Burroughs Corporation		Inter-Office Correspondence
Corporate Unit Computer Systems Group	Location Pasadena	Dept. Prog. Activity
TO: Name Programming Activity SPRITE Users		Date March 4, 1983
From Belinda Wilkinson	Dept. & Location Architecture Department	

Subject: New Release of SPRITE

On Monday, March 14, 1983, SPRITX (6601) will become SPRT66, SPRITE (6505) will become SPRT65 and a new version of SPRITE will be available. **SPRT65 and SPRT66 will be removed from the system a month later (April 14, 1983).**

The new version of SPRITE fixes bugs (see Appendix A and D) and provides new features (see Appendix B, C and E).

This SPRITE is version 6700 and is not MID-compatible with version 6601 or earlier versions. This inconvenience is necessary due to the changes within the SYSTEM information. **This requires that your MIDs be recompiled with SPRITE before any of your modules will recompile.**

The Y===== series of the intrinsics libraries are compatible only with SPRT65-emitted ICMs, and the X===== series of the intrinsics libraries are compatible only with SPRT66-emitted ICMs. Two new set of intrinsics libraries, S===== and E=====, are compatible with SPRITE-emitted ICMs. However, the E===== series of intrinsics libraries can only be used with ICMs produced with $$ EXTENDED (see Appendix B, item 18).

Concurrent with the release of the SPRITE compiler, a new version of XREF and COMPRS will also be available on the system. The new XREF program shows the correspondence between the module name and the file name used for that module. The old COMPRS program used to bomb when running on B3900 . The new one will now run on both B3900 and B4800.

Please report any problems to a member of the Implementation Systems Section with appropriate listings for screening before entering them into the BUGS system. All actual bugs will be entered into the BUGS system by the reporting user.

Belinda Wilkinson

Belinda Wilkinson, Manager
Implementation Systems Section
Architecture Department

This release document contains:
APPENDIX A: BUGS FIXED
APPENDIX B: GENERAL ENHANCEMENTS
APPENDIX C: OMEGA-RELATED ENHANCEMENTS
APPENDIX D: INTRINSICS BUGS FIXED
APPENDIX E: INTRINSICS ENHANCEMENTS

1. $MAP option with normal tag fields

The $MAP option did not list the normal tag fields of a STRUC declaration. Entries for these will now appear if $MAP has been set.

2. Nil pointer values (changed!!!)

The value of nil no longer changes or causes overflow when being moved as a 7sn value. (Regular pointers are 7sn for OMEGA programs.) This has been accomplished by changing nil to "CEEEEEEE" for regular pointers, "000000CEEEEEEE" for parametric pointers and "0CEEEEEEE" for procedure pointers. The new values are used for both OMEGA and non-OMEGA programs. **Pointer kludgers beware!!!**

3. Logical operations on hex strings over 100 digits (B2781)

Logical operations on fixed length hex strings over 100 digits now work for the entire string (it used to work only for the first 100 digits).

4. String comparison

In a relational expression where the left and right operands are both strings, SPRITE will now coerce the shorter one to the length of the longer (it used to coerce the right operand to the left operand no matter which one was longer).

5. DISPLAY ---> STRING coercion

SPRITE no longer allows the coercion from DISPLAY to STRING if the string is bigger. It will put out an error message if the string is fixed length. For variable length strings, it puts out a warning and then generates optional run time code to make sure that the string is not bigger than the display integer. SPRITE now puts out overflow testing code for the coercion from DISPLAY to STRING whenever the display integer is or could possibly be (in the case of variable length strings) bigger than the string.

6. Conditional subscript checking

The value of the BOUNDS dollar option is now checked before generating bounds checking code for array and subport indices. If BOUNDS has been reset or set to a value less than 4, the subscript checking code will not be generated.

7. Standard proc VAR parameter checking

Several standard procedures now correctly enforce their VAR access parameter requirements. Also, the standard function translate now requires only that its second parameter not be a constant (because a constant translate table is not mod 1000). It used to require VAR access.

8. File record size > 39996 (B2881)

Declaring a file with a record size greater than 39,996 digits no longer causes a compiler failure.

9. FOR ... DESCENDING, et al (B2882)

SPRITE no longer tries to optimize to MVW or MVA when either of the operands is a number. It now generates a MVN as before, which sets the comparison indicators properly. This was necessary to produce the right code for the FOR ... DESCENDING statement. This bug in turn had caused the compiler to fail when processing a call to a procedure with ten parameters.

10. Pointer coercion (B2819)

SPRITE now generates the right code to coerce between a pointer to a parametric string and a pointer to a fixed length string. It was generating bad code for a RETURN statement when the expression is one kind of pointer and the RETURN type is the other kind.

11. Variable length string to DISPLAY coercion (B2581)

SPRITE now generates the right code to coerce from a variable length string to DISPLAY, even when the dollar card option "BOUNDS" is reset. Also, "$$ BOUNDS" no longer resets the "BOUNDS" option.

12. Ptr function as parameter (B2701)

SPRITE now generates the right code for a parameter even when: the formal parameter is either UNIV or a non-parametric pointer passed by VALUE; the actual parameter is a call to the standard function "ptr"; and the actual parameter to "ptr" is either a constant, a data block variable, or a STATIC variable.

13. Dereferencing a constant pointer (B2605)

SPRITE now generates the right code to dereference a constant pointer (defined using a structure with an omitted tagfield).

14. Translating a variable length HEX string (B2824)

SPRITE no longer bombs when processing a call to the standard function "translate" with a variable length HEX string as the first parameter.

15. $$ LISTP (B2783)

SPRITE now lists all patches when compiling with "$$ LISTP, RESET LIST".

16. Finding BIT fields in an array (B2713)

SPRITE now puts out an error message if the type of the find primary in a FIND statement is BIT.

17. Range check subscripts for array slice (B2681)

SPRITE now generates range checking code for array slice subscripts.

18. Scale_ptr

The standard function scale_ptr (allowed only when producing assembly code) now generates the right offset for the destination address.

19. String concatenation with bad "edit_number" (B2893)

String concatenation with bad "edit_number" no longer causes compiler failure.

20. Macro's parameter (B2906)

SPRITE now catches the following error: The macro's formal parameter is VAR access, but the actual parameter is CONST access.

21. Mod 1000 flag reset for STATIC blocks

This flag was not being reset. When a STATIC block contained a TRANSLATE_TABLE, the mod_1000 flag was set and every STATIC block following it would also have the mod_1000 flag set. Needless to say, this wastes a bit of memory, so it has been remedied.

22. Nil to procedure pointer coercion (B2878)

SPRITE no longer generates the bad code when compiling "IF procedure_pointer = nil", and no longer bombs when compiling "IF nil = procedure_pointer".

23. FIND with SN key no longer generates bad code

SPRITE no longer tries to optimize the code for FIND statement if the type of the key is SN. It used to optimize the SN key to UA.

1. Port_io.open_available

A regular file identifier is now an acceptable parameter to the standard procedure port_io.open_available.

2. PORTRESULTS

An inquiry to the PORTRESULTS attribute will no longer modify the PFIB subport index field.

3. FIND with non-statement local pointers

For any of the find statement types, the FIND statement's result pointer may now be 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 and which on a non-hit will receive the nil pointer value.

find_pointer_spec

```
|                                             pointer:
\___________________________________________ ident _____________
                 \   \                     /              /      \
                  \   \___ LOCAL ___/              /        |
                   \                         pointer:  /          |
                    \__ EXTERNAL _____ primary __/           |
```

Examples:

```
     % p is local and available in THEN only
     % no change to current syntax
  FIND p AND i INTO array1 WHERE p@.num = 0
     % q is local and available in THEN only
  FIND LOCAL q INTO array2 WHERE q@.char = "X"
     % r is external, previously defined,
     % and available in its scope
  FIND EXTERNAL r INTO array3 WHERE r@.name = current_name
```

4. FIND with pointers delimiting array slice bounds

The pointer-to-pointer type of the FIND statement is now available. It 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 pointers must all be pointing to the type of the array's component. The array primary may not be an array slice.

find_statement

```
|
 \___FIND___find_pointer_spec___find_control___
                                               \
   ___________________________________________/
  /
  \___WHERE___find_condition___
                               \
   ___________________________/
  /
  \__THEN__statements____ELSE____statements____DNIF____
                        \___________________/          \
                                                        |
```

find_control

```
|
|\________________________________INTO__array:primary___
|    \              index:          /                    \
|     \___AND___identifier___/                            |
|                                                         |
|        base pointer:   limit pointer:       array:      |
 \___OVER__primary__ .. ___primary____INTO____primary___  |
                          \         /                    \|
                           \__ END _/                     |
                                                          |
```

Examples:

```
   % f and g used the ptr function
   % to point  at elements of array1
FIND p OVER f .. g INTO array1 WHERE p@.num = 0
   % ptr function itself may be used;
   % END gets last element of array2
FIND LOCAL q OVER ptr(array2 [2])..END INTO array2
   WHERE q@.char = "X"
   % pointer values have been previously
   % stored in a global structure
FIND EXTERNAL r OVER global.tbl3_begin .. global.tbl3_end
   INTO tbl3  WHERE r@.name = current_name
```

5. Logical operations and concatenations enhancement

It is now legal to do logical operations and concatenations between hex strings and display integers. The display integer involved in the operation will be coerced to a string of its own length with the base type set to the base type of the counterpart string. Example:
put.string ("dint2 = " + dint2);

6. Complex wait enhancement

The complex wait function now includes four new event types. They are: stoque_parameter_block.STOQINPUTEVENT, stoque_parameter_block.STOQOUTPUTEVENT, prog_name.CRCRINPUTEVENT and prog_name.CRCROUTPUTEVENT, where prog_name is a primary of type STRING (6) CHAR (EBCDIC), and stoque_parameter_block is a structure beginning with a field of type STRING (6) CHAR (EBCDIC) which is the STOQUE queue name. It is the user's responsibility to make sure that the queue_name is a 6 UA field STOQUE queue name and the prog_name is a 6 UA field program name.

7. Literal hex string optimization

The code produced for string operations involving a constant hex string has been changed in two respects. First, if the constant is either 1 or 2 digits long, an ORR/AND is replaced with a BST/BRT instruction. Secondly, if the constant is less than 7 digits long, it is made a literal in the A address field of the ORR/AND instruction.

8. New file types supported

For regular files, DCP and ISC are now legal values for the KIND attribute.

9. User-defined buffer io

For regular files, it is now possible to do input/output from different buffers. To allow this direct buffer access, five standard procedures (prepare_user_defined_buffer_io, write_buffer, read_buffer, read_random_buffer and write_random_buffer) under the standard module "io" have been added. Following is the module description that describes these new standard procedures and their parameters.

```
io
MOD
    prepare_user_defined_buffer_io
    PROC (file      FILE,
          buffer    UNIV PARAMETRIC_HEX_STRING); % Modulo and size
                                                 % must be mod 4
    read_buffer
    PROC (file      FILE,
          buffer    UNIV PARAMETRIC_HEX_STRING); % Modulo and size
                                                 % must be mod 4
    read_random_buffer
```

```
     PROC (file      FILE,
           buffer    UNIV PARAMETRIC_HEX_STRING,  % Modulo and size
           key       1..99999999);                % must be mod 4

     write_buffer
     PROC (file      FILE,
           buffer    UNIV PARAMETRIC_HEX_STRING); % Modulo and size
                                                  % must be mod 4
     write_random_buffer
     PROC (file      FILE,
           buffer    UNIV PARAMETRIC_HEX_STRING,  % Modulo and size
           key       1..99999999);                % must be mod 4

DOM;
```

There are five restrictions placed on these user-defined-buffer-io procedures.

(a) Prepare_user_cefined_buffer_io must be called before any of the read/write procedures can be used.

(b) There is only one buffer (i.e. only one pair of FIB_AA and FIB_BB) declared on the file to be used.

(c) The size and modulo of the buffer must be mod 4. Deferenced parametric pointers, variable length strings and fixed length substrings with varible offset are the user's reponsibility. The system will kill you if they are not mod 4.

(d) Once the buffer is used in prepare_user_defined_buffer_io, SPRITE will generate optional run time code to make sure that the **same** buffer is used for read_buffer, write_buffer, read_random_buffer and write_random_buffer.

(e) Using both the regular I/O procedures and the direct buffer I/O procedures for the same file is not allowed.

Following is an example that shows how to use these new standard procedures.

Example:

```
direct_buffer_io
MOD
```

```
TYPE
     REAL_RECORD = STRUC
                               reocrd     STRING (10000)
                                          % takes 20000 digits
                         CURTS,

     DUMMY_RECORD =,STRUC
                               dummy      STRING (2)
                                          % takes only 4 digits
                          CURTS;

file_block
FILE
     reader  [MYUSE = IN, KIND = DISK]
             OF DUMMY_RECORD, % Allocate 4-digit buffer
                              % instead of 20000-digit buffer
     printer [MYUSE = OUT, KIND = PRINTER]
             OF DUMMY_RECORD; % Allocate 4-digit buffer
                              % instead of 20000-digit buffer
buffer_block
DATA
     blk_buf_ptr     PTR TO REAL_RECORD;

driver
PROC

     prepare_direct_buffer_io;
     do_direct_buffer_io;

CORP;     % driver


prepare_direct_buffer_io
PROC;

     SHARES file_block, buffer_block;


     GENERATE EXTERNAL blk_buf_ptr;
     io.prepare_user_cefined_buffer_io (reader, blk_buf_ptr@);
     io.prepare_user_defined_buffer_io (printer, blk_buf_ptr@);

CORP;     % prepare_direct_buffer_io


do_direct_buffer_io
PROC;

     SHARES    file_bolck, buffer_block;


     io.read_buffer  (reader, blk_buff_ptr@);
     io.write_buffer (printer, blk_buf_ptr@);

CORP;     % do_direct_buffer_io
```

DOM; % direct_buffer_io

10. CASE statement optimization

The CASE statement will now use a multiply and indirect branch to select an alternate, rather than a search and indirect branch, but only if these conditions are satisfied:

1. There must be at least 12 alternates in the CASE statement. A search is faster for 11 or fewer labels.

2. The selector expression must be unsigned numeric or unpacked ORDERED or SYMBOLIC.

3. The result of the selector expression must have a length in the range 2..6.

4. A certain percentage of the possible alternate labels must be specified; otherwise, the case table will be much larger. For the lengths 2..6 these percentages are 80%, 67%, 67%, 57% and 57%. For example, if the selector expression result type is 100300..100399, then at least 57% of (100399-100300+1) or 57 alternates must be specified before a multiply will be generated.

For large CASE statements, the multiply is more than an order of magnitude faster than the search. In most cases (a little pun there) the compiler will automatically use the multiply, but if a few alternate labels must be manually added to the CASE statement, the rewards are worth it.

11. ICM_TOKEN definition change

Three new ICM_TOKEN fields were added for COBOL and FORTRAN. The "segment_threshold" field in MODULE_HEADER was changed from 4-UN to 2-UN to make room for a 2-UN "version_number". Both fields are set to zero, as before. The new BIT field "no_code_list" was added after "local" in MODULE_HEADER. Also, the new BIT field "fortran_external" was added after "returnseg_on_stack" in PROC_INTERFACE. Both BIT fields are set to false, instead of being "F"ed out.

12. File buffers in HIGH DATA

SPRITE no longer marks the buffer blocks as high data. The name of the buffer block for a particular file block is: "uwa_buff_XXXX_YYYYYY", where XXXX is a 4-digit block number assigned to the file block by SPRITE and YYYYYY is (the first six characters of) the file block name. This enhancement allows the user to put the buffer block in the appropriate overlay when binding together his program.

13. Parametric arrays

Parametric one-dimensional arrays parallel parametric strings in syntax and use. The same capabilities and restrictions apply. The syntax for a parametric array type definition is:

```
parametric array type defn
|
|                                             upper bound:
\__ TYPE __indicant__(__param:ident___subrange___)_ = _
                                                       \
 ______________________________________________________/
/                 lower bound:     param:            element:
\__ ARRAY __ [ __constant__ .. __ident__ ] __ OF __type_
                                             \____/      \
                                                          |
```

A parametric array must be one-dimensional. The lower bound constant must be an integer less than or equal to the lower bound of the upper bound subrange, which must be an integer range type.

A parametric array type can be the base type of a pointer as well as the type of a formal parameter. The standard operators "upb" and "lwb" can be used to discover the upper and lower bounds of the parametric array. The lwb function always returns the lower bound contant used in the parametric array type definition and the upb function returns a value which is: the lower bound value + the number of elements in the corresponding actual array - 1. Within the procedure, the index type of the parametric array is lwb..upb, and the semantics of fixed arrays apply. For example:

```
TYPE
     VECTOR (upbnd 7..100) = ARRAY [4..upbnd] OF 0..10000,
     VECTOR_PTR = PTR TO VECTOR;

build_vector
PROC;

     VAR  sum        0..1000000,
```

```
          vector    ARRAY [1..10] OF 0..10000;

     vector := [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
     sum := sum_vector (vector);

CORP;      % build_vector

sum_vector
PROC (vector   VECTOR)   RETURNS 0..1000000;

     VAR  sum  0..1000000 := 0;

     FOR i OVER lwb(vector, 1)..upb (vector, 1)
     DO                           % lwb returns 4
          sum +:= vector [i];   % upb returns 4 + 10 - 1 = 13
     OD;
     RETURN sum;

CORP;      % sum_vector
```

To allow the GENERATE statement to work for parametric array pointers, the syntax was modified as follows:

```
generate statement
|                                                              integer:
\__ GENERATE __ EXTERNAL ___primary___ ELEMENTS ___simple_expr___
                \__ LOCAL __/              \_ LENGTH _/             / \
                                             \_______________________/  /
                                                                       /
                                   ___________________________________/
                                  /              memory area:
                                  \____ IN ____identifier___________
                                                                    \
                                                                    |
```

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

14. Optional stack overflow check

The stack overflow check generated by SPRITE now has optional code markers (level=bounds_checking_code) around it.

15. Conditional compilation

The facilities for conditional compilation that COBOL and PASCAL have provided are now available in SPRITE. Each boolean type CCI (except "TITLE" and "CONTENTS") have been implemented on its own boolean stack, and $POP option has been added. when $SET or $RESET is used, the previous value of each boolean option specified will be stacked, and the current value will be set according to the boolean expression or default value. There are also up to 12 user-defined boolean options. The SET syntax is:

```
|            ______________________,___________________
|          /                                            \
\_ SET _\___boolean option name_______________________/
         \                      / \     boolean   /    \
          \_user-defined option_/   \_ =   expr  _/     |
                                                        |
```

```
boolean expr
|
|
\_______________  TRUE  ____________________________
 \                                          /       \
  |\____________  FALSE  ____________________/        |
  |                                        |          |
  |                 ___ OR ___              |
  |                /          \             |
  |  ___________/____ AND ___\______________|
  | /                                    \  |
  \_\_______________boolean option name___/__|
     \           / \     user-defined    /
      \_ NOT _/   \_______option______/
```

The boolean option name and user-defined option name in the above boolean expression must be declared before they can be referenced. The $POP option discards the current setting of each option in a list of boolean options, and restores the immediately previous setting. You will get a syntax error if you hvae too many POPs. The POP syntax is:

```
|               _____________ , ____________
|             /                             \
\______POP___\_____ boolean option name ___/_________
              \                         /            \
               \_ user-defined option _/              |
                                                      |
                                                      |
```

The RESET syntax is:

```
|              ____________ , ________________
|             /                               \
\_____RESET___\______ boolean option name _____/_____
                \                          /         \
                 \__ user-defined option ___/          |
                                                       |
```

Example:

```
$$ SET RELEASEVERSION = FALSE % Change to TRUE, when
          .                   % switch to release version
          .
$$ SET OMIT = RELEASEVERSION  % Instead of using "% COM;"
     dbwrite.string (.......);
               .
               .
     dbwrite.string (.......);
$$ POP OMIT                   % Instead of using "% MOC;"
```

Debug source can be surrounded with "%COM" and "%MOC". To delete the debug code from a release version, one would then change these to "COM" and "MOC". With conditional compilation, one can achieve the same thing by changing a single line, as shown in the above example.

16. Summary of virtual file utilization

Statistics of each virtual file's high bound, logical I/O, physical I/O, size (in digits) and overall hit ratio are now available. To have this information and CPU time for each pass, use a lower case "j" in the second program parameter : <compiler name>/dummy/j

17. Maximum modules

SPRITE now allows a maximum of 250 modules in one program.

18. EXTENDED option

The EXTENDED option allows a large program to address over one million digits. When this option is set, the internal representation of pointers will be 7 SN (rather than an address controller digit, hex "C" and 6 digits of address).

To take advantage of this new feature, you must set $$ EXTENDED in your MID and use the SYSTEM file produced to recompile all the modules for that program. Remember, you don't need to set this option when you compile your modules. Your must bind your program with the E=====

series of the intrinsics libraries.

19. Enhancement of one-dimensional array operations

It is now legal to move a one-dimensional array (slice) to another one-dimensional array (slice) provided that both arrays (slices) have (1) the same number of elements, and (2) the equivalent elememt type.

For parametric arrays and variable array slices, SPRITE will generate optional run time code to make sure the number of elements in both arrays is the same.

example:

```
   TYPE
        P_ARRAY (no 1..50) = ARRAY [1..no] OF ELEMENT;
   VAR
        array1    ARRAY [1..10] OF ELEMENT,
        array2    ARRAY [0..9]  OF ELEMENT,
        p_array   PTR TO P_ARRAY;

   array1 [1..5]  := array2 [4..8];
   p_array@       := array1;
   p_array@       := array2 [i..j];
```

20. FIND statement warning

SPRITE now puts out a warning for the FIND statement under the following two conditions:

(1). When the unit size of the find key is bigger than that of the find primary.

(2). When the types of the find primary and the find key are both type subrange and the range of the find key is not completely within the range of the find primary.

This is because SPRITE coerces the find key to the type of the find primary. Optional overflow testing or range checking code will be generated under the above conditions. This optional code will be deleted when you bind together a non-debug version of your program. This may produce strange results, such as a false match for p@ < key, where key is all F's because of the coercion failure.

21. MAXRECSIZE

MAXRECSIZE is no longer a required attribute for port files at declaration time. Its default value is 19998 bytes, if it is not declared.

22. Direct buffer io for PORT files

SPRITE used to allocate a buffer for each port file declared. Input from and output to the port file was then done by moving data between the buffer and the user's record (work area). This approach requires extra space for the buffer in addition to the space for the user's work area. To save the space for the buffer, SPRITE no longer allocates a buffer for each port file. Port I/O is now done directly from the user's work area.

1. Linked list FIND statement

The linked list FIND statement provides the ability to search a linked list for an element which satisfies a specified condition. 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:

```
find statement
|
\___FIND___find pointer spec______find control_____
                                                   \
  _________________________________________________/
 /
 \_WHERE_______find condition__________
                                       \
  _____________________________________/
 /
 \_THEN______statements___ELSE___statements___DNIF_____
                         \___________________/          \
                                                         |
```

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 list element 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. The syntax is:

```
find control
|                         predecessor pointer:
\__________________WITH_____identifier______
                  \______________________/   \
   __________________________________________/
  /           list pointer:
  \__FROM_____primary_____USING___link field__________
                                                      \
                                                       |
```

link field

```
|             ------------ - -----------
|            /        field name:        \
\___________\________identifier________/________________
                                                         \
                                                         |
```

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 preceding element. If no element in the list satisfies the find condition, the predecessor pointer points to the link of the last element. If the list is empty, or the first element satisfies the condition, it 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 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
|
|                                                      bit mask:
|\__ ANY_ONE_BIT_IN ___find primary___MATCHES____expr___
|\__ NO_ONE_BIT_IN  _/                                   \
|                                                        |
|                                           key:         |
 \__find primary________ = __________expression________  |
                 \____  ¬= ____/                        \|
                 |\___  <  ___/|                         |
                 |\___  <= ___/|                         |
                 |\___  >  ___/|                         |
                  \___  >= ___/                          |
```

If the ANY_ONE_BIT_IN or NO_ONE_BIT_IN form is used, all corresponding bits in each find primary and the specified bit mask expression are examined until an element is found which satisfies the match condition. A match occurs if any (ANY_ONE_BIT_IN) or no (NO_ONE_BIT_IN) pair of corresponding bits are both set. The bit mask must be a fixed length hex string the same size as the find primary.

If a relational form of the find condition is used, the array/list is searched for an element satisfying the relational condition.

2. Prog.read_timer and mcp.set_timer update

Prog.read_timer's return type and the parameter for OMEGA's version of mcp.set_timer were changed to 17-UN. This complies with the revised specifications for OMEGA's RDT and STT opcodes.

3. INT, APE, WHR

The following new standard procedures have been added to implement OMEGA's new INT, APE, and WHR opcodes:

```
mcp
MOD

interrupt                                               % INT
PROC;

make_page_table_entry_unused                            % APE 00
PROC (descriptor          UNIV P_STR_8_HEX);

copy_page_table_entry                                   % APE 01
PROC (source_descriptor,
      dest_descriptor     UNIV P_STR_8_HEX);
          % user-defined, 8-digit structures
          % describing which PTE's are involved
          % (descriptors themselves are not changed)

update_reinstate_list_address                           % WHR 00
PROC (new_address  UN_8);

update_snap_picture_address                             % WHR 01
PROC (new_address  UN_8);

update_memory_error_address                             % WHR 02
PROC (new_address  UN_8);

read_clear_processor_status                             % WHR 03
PROC (status  VAR UNIV P_STR_2_HEX);
          % user-defined, 2-digit structure
DOM;
```

4. Prog.lock_conditional

There is now an exception clause for the standard procedure prog.lock_conditional. It works just like "IF EOF" for I/O standard procs. The syntax and semantics are as follows:

lock_conditional exception clause

```
|
|
\__IF__~~LOCKED~~__THEN__statements___ELSE__statements___FI__
       IN_USE                         \_______________/        \
                                                                |
                                                                |
```

```
prog.lock_conditional (lock)                         % no semicolon
     IF    ~~LOCKED~~ IN_USE
     THEN       % it was already locked
          do_something_else_instead;

     ELSE       % now I have it
          do_something_with_it;
     FI;
```

5. MCPCAL and BGOVL calls

When calling an overlay module entry point, SPRITE now generates a VEN to either MCPCAL or BGOVL, depending on where the call is from. If the call is from an overlay module, SPRITE generates a VEN to MCPCAL. Otherwise, it generates a VEN to BGOVL. The overlay modules are specified in the MID by the overlay statement, as follows:

```
overlay statement
|                   ________ , ________
|                  /                   \
\_____OVERLAY_____\___module: ident___/______ ; _______
                                                       \
                                                        |
```

6. Scale_ptr

The standard function scale_ptr can now be used to initialize data block pointer variables at compile time. Also, a call to scale_ptr may now appear wherever the context clearly defines the resulting pointer type (such as the actual parameter to another procedure). (The above is also true for $SMCPVI.)

7. INCLUDE markers

The following INCLUDE markers have been added to our MID:

```
     general_and_vf_defn,
     max_image_and_text_length,
     position_info_type,
     symbol_table_and_token_defn,
     file_attr_symbolics,
```

```
id_and_attr_defn,
vf_file_data,
icm_defn_one,
icm_defn_two,
icm_put_module,
symbol_table_module.
```

1. put.go_to_col (1)

When called with a parameter of 1 (one), "put.go_to_col" now correctly recalculates "pt.char_used" and "pt.char_left" in the "put_line_info" DATA area.

2. dbwrite and f_dbwrite labels

The dbwrite and f_dbwrite modules use put.string to print the label fields, causing labels longer than 100 characters to be incorrectly printed.

The MID for these modules has been changed to limit labels to a maximum of 100 characters.

1. put.swap_line

A new procedure, "swap_line" has been added to the "put" module. This allows a program to construct two or more lines simultaneously by exchanging all of the information in the "put_line_info" DATA area.

The "dbwrite" module now uses this procedure to create its output lines while preserving whatever the rest of the program has done with the "put" module.

Burroughs Corporation ______________________ Inter-Office Correspondence

Corporate Unit	Location	Dept.
Computer Systems Group	Pasadena	Prog. Activity
TO: Name		Date
Programming Activity SPRITE Users		March 26, 1982
From	Dept. & Location	
Belinda Wilkinson	Architecture Department	

Subject: New Release of SPRITE and SPRITX

On Tuesday, April 6, 1982, SPRITX (6505X) will become SPRITE and a new version of SPRITX will be available.

The new version of SPRITX fixes bugs (see Appendix A) and provides new features (see Appendices B and C).

The new SPRITX is version number 6601 and is <u>not</u> MID-compatible with version 6505X or earlier versions. This inconvenience is necessary due to changes within the SYSTEM information.

Both SPRITE and SPRITX emit Type III Format 7 ICMs. However, ICMs created by SPRITE are <u>not</u> compatible with ICMs created by SPRITX, as the interface to the debug module has changed. If you wish to use SPRITX, you must recompile your MID, <u>all</u> of your MODs, refilter your BPL-created ICMs with the new version of FILTX, and un-truncate bind-deck names which are greater than 24 characters in your program source before you bind your code file.

The Y===== series of the intrinsics libraries are compatible only with SPRITE-emitted ICMs. The X===== series of the intrinsics libraries are compatible only with SPRITX-emitted ICMs. Intrinsics enhancements and bug fixes will be only in the X===== series (see Appendices D and E).

Please report any problems to a member of the Implementation Systems Section for screening before entering them into the BUGS system. Bring the appropriate listings and whatever else we might need to determine that the problem is truly a SPRITE bug. All actual bugs will be entered into the BUGS system by the reporting user.

Belinda

Belinda Wilkinson, Manager
Implementation Systems Section
Architecture Department

This release document contains:

APPENDIX A: BUGS FIXED
APPENDIX B: GENERAL ENHANCEMENTS
APPENDIX C: MCP-RELATED ENHANCEMENTS
APPENDIX D: INTRINSICS BUGS FIXED
APPENDIX E: INTRINSICS ENHANCEMENTS

1. Eliminate unnecessary calls to the "move" intrinsic (B2708)

   Certain special conditions no longer cause the compiler to generate unneeded calls to the move intrinsic.

2. Variable "prog.bct" parameters restored after BCT

   If you use a variable string of hex as the parameter to prog.bct, SPRITE now moves the string back to your variable after the BCT has been executed. Thus you may now access any information which has been changed by the MCP as a result of the BCT.

3. RETURN statements disallowed in MACRO definitions

   You may not define MACROs which contain RETURN statements. This used to cause SPRITE to generate an exit from the procedure which "called" the macro.

4. DATA declarations cause incorrect syntax errors

   The last variable in a DATA declaration will no longer cause certain things (such as a FILE declaration) to be incorrectly found to have syntax errors in some cases.

5. Heap overflow detection (B2670)

   The code SPRITE generates to detect heap overflow now checks to see if the next available heap location is > the heap limit (rather than >= the limit).

6. Bad code for ptr function when destination indirect

   SPRITE now generates correct code for the ptr function even when the destination has indirection involved.

7. Revised heap/stack collision code for HIGHHEAP (B2717)

   If you set the HIGHHEAP dollar card option in your module, SPRITE now generates procedure prologue heap/stack collision code which calls err.error (unless the ERRORCALLS option is reset, in which case it generates a hex "EC" opcode to cause a processor error at run-time). Previously, SPRITE unconditionally generated the hex "EC" opcode.

8. RESET multiple dollar card options

If you use RESET on a dollar card, it will now apply to all of the following options on the card (or until you specify SET). Previously, just the first option was reset while the remainder were set.

1. "filler" for unused fields in a structure definition or a data declaration

You may use the word "filler" as an identifier anywhere in a structure definition or a data declaration. You cannot reference the parts of the structure or the fields in the data declaration thus defined. You may use "filler" any number of times in a given structure definition or data declaration.

The word "filler" is now a predefined identifier in the SPRITE language. Use of this word 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 still 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.

2. Standard functions "zone_index_any" and "zone_index_none"

These new functions (each requiring 2 EBCDIC strings as parameters) allow you to scan strings for particular zone digits. They perform in a manner similar to "index_any" and "index_none", save that only EBCDIC strings are allowed as parameters. They generate SZE (scan zone equal) and SZU (scan zone unequal) machine instructions.

Zone_index_any returns the index of the first character in string 2 which has a zone digit equal to a zone digit in any character of string 1. If none is found, it returns a zero.

Zone_index_none returns the index of the first character in string 2 which has a zone digit not equal to a zone digit in any character of string 1. If none is found, it returns a zero.

For example,

```
number_ix := zone_index_any ("0", card_image);  % find
    %  first character "0" thru "9" (also hex FA, etc.)
IF zone_index_none ("AJS", word) = 0
THEN     %  there are no uppercase letters in this word
    ...
```

3. MAP dollar card option

MAP is a new option which you may set or reset on a dollar card. The default value of this option is reset.

Within the range of SPRITE source code that this option is set, the output listing lines of STRUCture definitions and DATA definitions are modified to show the internal details of the structure or the data block.

The card-image origin field of these output lines (normally "EDITOR", "INCLUDE", "PATCH", etc.) now contains 3 columns of information as follows:

1. **size** (if BIT, then "." plus allocated bit)
2. **offset**
3. **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 you to get the most information from this option, each DATA variable or STRUCture component must be on a separate source line.

4. Strings <= 100 characters allowed as VALUE parameters

You may now use strings of up to 100 characters as VALUE parameters to a procedure. The previous limit was 50 characters.

5. 30 characters of identifiers and indicants now used

Your identifiers and indicants must now be unique within the first 30 characters, rather than 24.

NOTE: Be sure to change your bind decks in those cases where you previously had to truncate an identifier to 24 characters. SPRITX and BINDX now truncate identifiers in the same manner.

6. Move words or move alpha done where possible

The compiler now generates MVW or MVA code in certain cases which used to be handled less efficiently.

7. Standard procedure "move_words"

This new standard procedure, which should be used with extreme caution, allows you to force the compiler to generate MVW code in circumstances 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 out to be caused by misuse of this standard procedure.

For example,

```
move_words (source_field, destination_field);
```

Clarification of SPRITX Release Memo
Appendix B, Items 6 and 7
Move Optimizations

Most people thinking of using the new standard function move_words will have no need of it. SPRITX now 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 SPRITX 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.

8. Initialization of pointer variables

Variables of type pointer (but not pointer to procedure) may now be initialized at compile time. The syntax, semantics and rules are:

SYNTAX: ptr_variable PTR [TO] <access> <level> <any_type> [STATIC] := ptr (referent);

SEMANTICS: ptr_variable is initialized to point to referent.

RULES: (a) Subject to all rules that apply to the use of ptr function.
(b) The address of the referent must be determinable at compile time.
(c) The following table shows the kinds of of pointers and the valid referents each kind of the pointer can point to.

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);
```

CONTENTS dollar card option

The CONTENTS option has the same format as the TITLE option. However, the string you specify appears <u>only</u> in the table-of-contents at the end of the compile listing. You may use this option for easily finding things within your MIDs and modules without affecting your present page headings. For example,

```
$$ CONTENTS "3.7    Virtual File TYPES"
```

10. MODULO allowed for data types

MODULO allows you to specify the modulo boundary at which a data object is 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 not start with "MODULO" (i.e. 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.

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 (don't forget 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.

The type checking has been changed so that 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
```

11. Heap overflow check code is now optional

The compare, branch, and call to err.error are now marked as optional code.

12. New port file attributes

The following port file attributes are now available for your use. They apply only to ports (not to subports), yet these fields have fresh information available for your inquiry after every port or subport operation.

Attribute	Type	Port Get	Port Set
ATTERR	STRING (2) OF HEX	Yes	No
MYPORTADDRESS	STRING (4) OF HEX	Yes	No
PORTRESULTS	STRING (100) OF HEX	Yes	No

The following enhancements apply only if you set the $ MCPVI option in your MID.

1. Pointers are 7 SN

The internal representation of pointers is now 7 SN, rather than an address controller digit, hex "C", and 6 digits of address.

2. Pointer arithmetic with standard functions "ptr_add" and "ptr_sub"

Two new standard functions, ptr_add and ptr_sub, allow you to perform some basic pointer operations. These functions, **which should be used with extreme caution**, help produce better code when stepping through an array or a string. Their syntax, semantics and rules are:

PTR_ADD

```
SYNTAX:    pointer_1 := ptr_add (pointer_2, num);

SEMANTICS: pointer_1 := pointer_2 + num * size (pointer_2@);

           where size (pointer_2@) is the size of the
                 referenced type rounded up to a
                 multiple of the modulo of the type.

RULES:     (1) Pointer_1 and pointer_2 are pointers with
               equivalent referenced types.
           (2) Parametric pointers are not allowed.
           (3) Pointers to procedures are not allowed.
           (4) num is any numeric expression whose value is
               in 0..9999999
```

PTR_SUB

```
SYNTAX:    pointer_1 := ptr_sub (pointer_2, num);

SEMANTICS: pointer_1 := pointer_2 - num * size (pointer_2@);

           where size (pointer_2@) is the size of the
                 referenced type rounded up to a
                 multiple of the modulo of the type.

RULES:     same as that of PTR_ADD.
```

*** WARNING *** No compile-time or run-time checks are made to protect the integrity of the pointer. It is your responsibility to ensure that these functions will work properly when your program runs.

These port file attributes are now available, but only for the use of BNA's Port Manager program. They apply only to **subports** (not ports).

Attribute	Type	Subport Get	Set
HISCODEFILEFAMILY	STRING (6)	Yes	No
HISCODEFILENAME	STRING (6)	Yes	No
HISCOMPRESSIONFLAG	STRING (1) OF HEX	Yes	Yes
HISFLOWSTATUS	BOOLEAN	No	Yes
HISMYNAME	STRING (100)	Yes	No
HISNULLFLAGS	STRING (1) OF HEX	Yes	No
HISOPENTYPE	0 .. 99	Yes	No
HISPORTADDRESS	STRING (4) OF HEX	Yes	No
HISSUBFILEERROR	NOERROR, DISCONNECTED, DATALOST, NOBUFFER, NOFILEFOUND, UNREACHABLEHOST	No	Yes
HISSUBPORTADDRESS	STRING (4) OF HEX	Yes	No
HISUSERCODE	STRING (17)	Yes	No
HISYOURNAME	STRING (100)	Yes	No
PLMCHARACTERSETS	STRING (1) OF HEX	Yes	Yes
PLMMATCHRESP	BOOLEAN	No	Yes
PLMMAXMSGTEXTSIZE	2 .. 19998	No	Yes
PLMMYCODEFILEFAMILY	STRING (6)	Yes	Yes
PLMMYCODEFILENAME	STRING (6)	Yes	Yes
PLMMYHOSTNAME	STRING (17)	Yes	Yes
PLMMYNAME	STRING (100)	Yes	Yes
PLMSECURITYGUARD	STRING (6)	Yes	Yes
PLMSECURITYTYPE	GUARDED, PRIVATE, PUBLIC	Yes	Yes
PLMSECURITYUSE	IO	Yes	Yes
PLMTITLE	STRING (17)	Yes	Yes

3. Generates assembly code for MCPGEN

SPRITE creates for you an 80 blocked 5 disk file containing card images in ASMBLR format. The internal name of this file (for label equation) is MCPASM.

The contents of this file is for use with a new flavor of MCPGEN (currently named "MCGXbp").

Procedure pointers, calls to SPRITE's intrinsics, and GENERATE EXTERNAL are not currently supported. Attempts to use these will cause syntax errors.

SPRITE still produces an ICM file, even though it is currently useless.

4. ALIAS statement

The ALIAS statement allows you to equate SPRITE names with MCPGEN labels. This statement goes in your MID, and you may not use a KNOWS list with this statement.

The syntax of the ALIAS statement is:

```
alias statement
|                 ________________ , ____________________
|                /                          assembly name:  \
\__ ALIAS __\__ sprite name __ = __ string literal __/__
                                                         \
                                                          |
                                                          |
```

```
sprite name
|
|      alias primary:
\__ ident or indicant ____ alias selections ____
                             \__________________/  \
                                                    |
                                                    |
```

```
alias selections
|     ____________________________________________
|  /                                              \
|  |       const array or string selection:        |
\__\____ [ ____________ exor ____________ ] ____/__
       \                                       /     \
        \___ . ___ proc or field:ident _______/       |
                                                      |
```

The assembly name cannot be more than six characters long.

The alias primary must be either: (a) a module, (b) a data variable (or the entire block), or (c) an indicant. Depending on which kind of primary is used, the ALIAS statement serves any of three purposes:

(a) It specifies the label for SPRITE to use when defining and calling a procedure (or module). This avoids the default "Pmodule#proc#" (or "Mmodule#), which can change when a new module or procedure is added (even if just to a knows list).

(b) It generates an EQIV (or BIT#) command to declare a label for data in a data block. This avoids the default "Dblock#offset" (using the inc field), which can change when the block changes or a new data block is added. However, SPRITE modules still use the default label.

(c) It generates an EQIV (or BIT#) command to declare a label for an indicant (and its selections). This label is used with an index register containing the address of a variable of the indicant's type.

For example,

```
ALIAS  preterm_module.terminate_this_program  =  "PRETRM";

ALIAS  sm_io  =  "SM-IO",
       kbo    =  "KBO";

ALIAS  Q_ELEM                             =  "Q-AREA",
       Q_ELEM.next                        =  "Q-LINK",
       Q_ELEM.io_descr                    =  "Q-DESC",
       Q_ELEM.io_descr.opcode [1::2]      =  "Q-OP"   ;

ALIAS  q_elem                             =  "Q/AREA",
       q_elem.next                        =  "Q/LINK",
       q_elem.io_descr                    =  "Q/DESC",
       q_elem.io_descr.opcode [1::2]      =  "Q/OP"   ;

ALIAS  MASTER_AVAIL                                 =  "MST-AV",
       MASTER_AVAIL [0]                             =  "MST-EL",
       MASTER_AVAIL [0] .avail_disk_addr            =  "MST-SS",
       MASTER_AVAIL [0] .avail_disk_addr.eu         =  "MST-EU",
       MASTER_AVAIL [1]                             =  "MST$EL";
```

5. OVERLAY statement

The OVERLAY statement allows you to specifiy which modules are located in the MCP's overlay area (as opposed to global or extension modules). This statement may only be used in your MID. SPRITE must handle calls to entry procedures in these modules by generating an NTR to MCPCLL in order to make the overlay present.

The syntax for this statement is:

```
     overlay statement
     |
     |                    _______ , ________
     |                   /                  \
     |                   |  module name:    |
     \___ OVERLAY ___\_____ ident ______/___
                                              \
                                               |
                                               |
```

The modules named in this statement must already have been defined in a MOD statement. For example,

```
OVERLAY keyboard_out, preterminate;
```

6. MCPCLL code for calls to procedure in overlay (MCPVI)

Whenever you make a call to a procedure which is an entry point into a module which is in an overlay (known by means of the OVERLAY statement), SPRITE will generate an NTR to MCPCLL with the appropriate parameters required by MCPCLL.

If the module from which you make the call is <u>not</u> in an overlay, SPRITE will print a warning, since MCPCLL calls from global are usually an error.

7. PROCESS_RUN statement

The PROCESS_RUN statement allows you to initiate an asynchronous call on a procedure in an overlay. The syntax for this statement is:

```
     process run statement
     |
     |                       proc call:                 string:
     \__ PROCESS_RUN ___ primary ___ USING __ simple expr __
                                                              \
                                                               |
                                                               |
```

The procedure call includes any necessary parameters. Note that all passed parameters must be VALUE parameters, that the total size of all parameters may not exceed 26 digits, and that no function may be called.

The string is the particular entry point into the MCP's BEGOVL routine which you wish to call. You will get a warning if it is not one of the following:

```
BEGOVM    BEGOVH    BEGOVL    BEGCTL
```

For example,

```
PROCESS_RUN pretrm.prog_err (inv_read) USING "BEGOVL";
```

8. REMAPS declaration

The REMAPS declaration permits you to redefine the way that a DATA block looks, so that you may save space (and possibly restrict knowledge of various redefinitions) without resorting to omitted tagfield structures.

You may use this declaration only in your MID. The syntax of this declaration (similar to a DATA declaration) is:

```
remaps declaration
|
|  remap DATA name:               target DATA name:
\_____ ident ________ REMAPS _______ ident ___
                                              \
                    _________________________/
                   /
                   \___ variable list ___
                                         \
                                          |
                                          |
```

For example,

```
sm_io
DATA
    basic_definition  STRING (200) OF HEX;

sm_io_for_keyboard_output
REMAPS sm_io:
    keyboard_command  BOOLEAN,
    message_number    0..999;
```

To access variables in a REMAPS declaration, use the remap data name in a SHARES list, just as you would for a DATA declaration.

9. Standard function "scale_ptr"

You may use this standard function to scale a number by some power of ten and convert the result into a pointer. This function may be used only in an assignment statement whose left-hand side is a pointer to a non-parametric type (note: no other type checking is done on this pointer).

The scale_ptr function requires 2 parameters. The first is the number upon which you wish to operate. The second is a positive integer constant power of ten by which the first is scaled (for example, a value of 2 means multiply by 100).

You will get a syntax error if the maximum possible value of the scaled number exceeds the size of the largest possible pointer address.

For example,

```
VAR  program_ptr  PTR TO STRING (100000) OF HEX;

program_ptr := scale_ptr (mix.base_addr_in_kd, 3);
```

1. "err.error" starts error message on new line

The run-time error message you get from err.error will now start at the beginning of the line, even if your program uses "out" module procedures.

2. Debug prints EXT lines when "db_monitor_all" is set

If you program sets db_monitor_all, your output listing will now show procedure EXT lines as well as procedure NTR lines.

3. Better statistics from statistics version of debug

You will now get the correct active time for the program entry procedure. Previously, the active time for this procedure might be off by a bit.

If you had explicit call to debug.summary in your program, this bug might also have affected the active times of other procedures.

This bug could also cause processor errors (invalid arithmetic data) on B2900/3900s.

his

1. Debug terminates on errors in extended input

If you use the "//X" option and your extended input to debug has errors, debug will now immediately terminate the execution of your program.

2. Debug checks for NTR / EXT mismatch

Debug now checks to see if your program has mismatched NTR / EXT problems during execution. If so, it prints one warning message the first time that it detects this problem.

3. The "hrtime" intrinsic has been deleted

Having received no reaction to our warning in the last SPRITE release letter, we have now deleted the "hrtime" module from the intrinsics which SPRITE supports.

Burroughs Corporation — Inter-Office Correspondence

Corporate Unit	Location	Dept.
Computer Systems Group	Pasadena	Prog. Activity

TO: Name	Date
Programming Activity SPRITE Users	October 10, 1985

From	Dept. & Location
Charlie C. Chan	Architecture Department

Subject: New Release of SPRITE

On Monday, October 14, 1985, a new version of SPRITE and a new set of S===== and E===== series of intrinsics libraries will be released for in-house use. The new version fixes bugs (see Appendix A and D) and provides new features (see Appendix B, C, and E).

This SPRITE is version 1000 and is **not** mid-compatible with version 6700 or earlier versions. This inconvenience is necessry due to changes with the SYSTEM information. **This requires that your MIDs be recompiled before any of your modules will recompile.**

The new version of SPRITE emits both type 3 (old style format) and type 4 (OMEGA style format) ICMs. To get type 4 ICMs, you have to compile your MIDs with $PAGING and use "FILE **ICM4** = " instead of "FILE ICM = ".

Please report any problems to a member of the SPRITE project group for screening before entering them into the BUGS system. Bring the appropriate listings and whatever else we might need to determine that the problem is truly a SPRITE bug. All actual bugs should be entered into the BUGS system by the reporting user.

For more copies of this memo, do "SYS COMP 8987:RM10 ON ALS".

Charlie

Charlie C. Chan
Implementation Systems Section
Language Department

This release document contains:

APPENDIX A: BUGS FIXED
APPENDIX B: GENERAL ENHANCEMENTS
APPENDIX C: OMEGA-RELATED FEATURES
APPENDIX D: INTRINSICS BUGS FIXED
APPENDIX E: INTRINSICS ENHANCEMENTS

1. FIND with a key of type pointer

SPRITE no longer generates bad code for FIND statement if (1) the key is a field selection through pointer dereference, (2) the type of the key is pointer or (3) SPRITE tries to optimize the code by converting the key from un to ua.

2. The code for indexing into arrays and sets (B2996)

(1) The SEA instruction no longer gives a false match when the index type is PACKED ORDERED and over one digit. (2) SPRITE no longer generates a SDE instruction instead of SEA when the index type is EBCDIC (or PACKED ORDERED and over 100 digits).

3. 8-digit filler in front of each file buffer (B3042)

SPRITE now allocates 8-digit space in front of each buffer for all kinds of files instead of just the first buffer for PRINTER and PUNCH files.

4. $XREF

SPRITE no longer gives the message "DUP LIB sxxtyd DSK" when you compile your module with $XREF.

1. FIND with OVER and AND clauses

The OVER and AND clauses in the FIND statement are no longer mutually exclusive.

Example:

```
FIND a_ptr AND a_idx
OVER base_ptr .. limit_ptr INTO array
WHERE a_ptr@ := key
DO
   ....
OD;
```

2. Division optimization

Currenly, SPRITE optimizes division via truncation (MVN) when the divisor is a constant power of 10. The new SPRITE takes another step further by trading a DIV instruction with MPY and MVN instructions if the divisor is not explicitly a power of 10, but is a factor of a power of 10. For example, the expression "a/2" is equivalent to "(a*5)/10". The result would be a MPY instruction (t := a*5) followed by a MVN instruction (r := t/10).

3. Proc_ptr and forward procedure definition in MID

The user can now use proc_ptr function to initialize a MID data block variable of type PTR TO PROC. The referenced procedure can be forward defined in MID. The parameter list and return type defined in proc ptr declaration must match those of the forward defined referenced procedure.

4. SPRITE I/O enhancements

(a) Shared files

It is now possible to declare disk or diskpack files to be shared between different multiple processors. Shared files are assumed to be random. The user declares shared files by setting the ACCESSMODE to SHARED. A new file attribute STALEMATE was added which allows the user to specify the procedure to be called by MCP to handle stalemate coditions. **It is the user's reponsibility to make sure that this procedure is** in segment 1

File attribute descriptions for ACCESSMODE and STALEMATE are as follows:

ACCESSMODE

```
DISK/DISKPACK : Read: anytime, Write: closed
                Mnemonic: SEQUENTIAL, RANDOM,
                          SHARED
                Default : SEQUENTIAL
```

Specifies the disk access technique.

STALEMATE:

```
DISK/DISKPACK : Read: never, Write: closed
                Address constant: mod_name.proc_name
                Default: none
```

Specifies the name of the procedure to be called by the MCP to handle stalemate condition. It must be a procedure without any parameters. The only way to get out of this procedure is by calling io.exitroutine.

The following standard procedures were added to allow the user do I/O's from the shared files.

```
(1) io.open_lock (file        FILE);

     % Once the file is opened with "lock", no
     % other program will be able to open the
     % file until the locking program closed it


(2) io.open_lock_access (file      FILE);

     % Once the file is opened with "lock_access",
     % any other program may open the file as
     % input but not output.


(3) io.read_no_unlock  (file          FILE,
                        record  VAR   RECORD,
                        key           1..99999999);

     % Lock the record, read the record
     % and leave the record locked


(4) io.read_with_unlock (file       FILE,
```

```
                                   record VAR RECORD,
                                   key        1..99999999);

      % Lock the record, read the record
      % unlock the record


(5) io.write_no_unlock  (file          FILE,
                         record        RECORD,
                         key           1..999999999);

      % Lock the record, write the record
      % and leave the record locked


(6) io.write_with_unlock (file        FILE,
                          record      RECORD,
                          key         1..999999999);

      % Lock the record write the record
      % and unlock the record


(7) io.lock (file          FILE,
             key           1..999999999);

      % Lock  the  record  only,  no  data  transfer.
      % If the record is  locked  by another program,
      % the program waits until it has been unlocked.


(8) io.unlock (file        FILE,
               key         1..999999999);

      % Unlock  the  record  only,  no data transfer
      % the program will be terminated if the record
      % has not been previously locked


(9) io.seek_no_unlock (file      FILE,
                       key       1..999999999);

      % Lock the record, request the MCP to make the
      % record  available in the  program buffer and
      % leave the record locked.


(10) io.seek_with_unlock (file      FILE,
                          key       1..99999999);

      % Same as seek_no_unlock except it unlock the
      % the record at the end


(11) io.exitroutine (file      FILE);
```

```
% The  only  way  to get out the  procedure
% which handles the stalemate conditions is
% by calling this procedure.
```

(3), (4), (5), (6) and (7) may take the "IF INVALID_KEY THEN ...." exception clause.

Example:

```
shared_file_block
FILE
     shared_file [MYUSE = IN, KIND = DISKPACK
                  STALEMATE = mod.stalemate,
                  ACCESSMODE = SHARED
                 ];

SHARES
     shared_file_block;

......

io.open_lock (shared_file);

......
```

(b) io.open_no_rewind (file FILE);

This procedure is used to open magnetic tape files without positioning to the beginning of tape. This is primarily used when opening the second and all subsequent files on a multi-file reel of magnetic tape.

(c) io.open_reverse (file FILE);

This procedure can only be used with single reel, single file, tape files. When the file is opened with this procedure, the subsequent read will make the data records available in the reverse record order starting with the last record.

```
(c) io.open_get_dhdr (file                FILE,
                      dhdr  VAR UNIV   P_STR_40_HEX);

     % dhdr must be 40 digits long
```

% for format of header, see Programmer Guide

The io.open_get_dhdr standard procedure will let the user open the file and get the disk file header at the same time. Following is the procedure description.

```
(d) io.set_buffer_addr (file          FILE,
                        buffer  UNIV  PARAMETRIC_HEX_STRING);
```

% modulo and size of buffer must be mod 4, must be
% called after io.prepare_user_defined_buffer_io

Currently, the "io.prepare_user_defined_buffer_io" adjusts the blocksize of the file and sets the fib_aa and fib_bb to point to the buffer location. To avoid additional code generated each time a different buffer of the same length is used, the "io.set_buffer_addr" may be used to set the fib_aa and fib_bb addresses without generating the code to adjust the blocksize.

5. File pointers

File pointers in SPRITE allow the users to make runtime determinations about the files used in their program. The syntax for a file pointer is:

```
file pointer type defn
|
|
\___PTR___TO_________________FILE____OF_____type__
     \____/  \_CONST_/          \____/           \
              \_VAR_/                            |
                                                 |
```

where type is the record type of the file being pointed to

This syntax has a precedent in SPRITE's current parametric pointers, where one can define a pointer to a type, yet one can not define a variable of that type.

File pointers may be passed as parameters to procedures. Two file pointers are equivalent to each other if they have equivalent file record types. Equivalent file pointers may be compared for equality with = and ¬=. Just like regular pointers, file pointers are built by using the "ptr" standard functions. No operations (+, -,

>, <, etc.) other than ":=" are allowed.

Example:

```
TYPE
    FILE_PTR = PTR TO FILE OF RECORD;

VAR
    file_ptr     FILE_PTR;

file_ptr := ptr (shared_file);
file_ptr@.MYUSE := IN;

io.open_lock (file_ptr@);
```

6. Return of the RETURN statement in MACRO

Once again, RETURN statement in MACRO is back. The RETURN statement causes an exit from a MACRO and the control passes to the instruction following the end of the body of the MACRO call.

7. New standard function -- search_string

The new standard function, search_string, is a more generalized search routine than the current index functions (index, index_any and index_inc). The following is the description of this new standard function.

PARAMETERS:	#	Type	Access	Description
	1	String_1	CONST	Key; can be variable length or parametric
	2	0..2	CONST	key_datatype 0 : un 1 : sn 2 : ua
	3	String_2	CONST	String to be searched; Can be variable length or parametric
	4	1..100	CONST	Increment between comparison
	5	0..2	CONST	search kind

0 : search for equal
1 : search for low
2 : search for lowest

RETURN TYPE : 0..length (string_2)

FUNCTION : Returns the index of the first occurrence of string_1 in string_2 where the occurrence begins on a multiple of the 4th numeric parameter (1, n+1, 2n+1, etc.) if the search kind is 0 (EQUAL).

Returns the index of the first occurrence of string_1 in string_2 which is less than string_1 where the ocurrence begins on a multiple of the 4th numeric parameter (1, n+1, 2n+1, etc.) if the search kind is 1 (LOW).

Returns the index of the lowest of all the occurrences of string_1 in string_2 where the occurrences begin on a multiple of the 4th numeric parameter (1, n+1, 2n+1, etc) if the search kind if 2 (LOWEST).

Returns 0 if the search condition fails

Examples:

```
indx := search_string (key [1::3], keytype,
                       str [1::pos], 3, 0);

indx := search_string (key_ptr@ [2::incr], 2,
                       str [1::pos], incr,
                       search_kind);

indx := search_string (key, keytype, str_ptr@@,
                       incr, search_kind);
```

8. $INDEX

The new $INDEX CCI allows the user to create an alphabetized index of where all of a module's procedures , identifiers and types are defined.

The syntax of the INDEX CCI is:

```
|  ____________________,__________________
| /                                       \
\_\___INDEX_______________ALL ______________/____
                 \                 |              \
                 | \____PROCEDURES____/|              |
                 |                     |              |
                 | \___IDENTIFIERS___/ |
                 |                     |
                 | \______TYPES______/ |
                 |                     |
                  \______MODULES______/
```

The ALL option of the INDEX CCI will create an index for each of the other INDEX option. If the MODULES option is set in a module, it has no effect (in a MID it indexes the module descriptions). If the PROCEDURES option is set in the MID, it creates an index of where the procedure descriptions occur. If the PRODEDURES option is set in a module, then it will create an index containing both procedures and MACROS.

The INDEX IDENTIFIERS option will produce an index of where all of the constants, data blocks, data block variables, file blocks, file names, and local varaibles are defined.

The INDEX CCI options can be set anywhere in a moudle (or MID) and will index the specified item(s) over the entire module (or MID) and not from the point it was set to the end of the module (or MID). All of the INDEX options default to being not set and there is no way to reset an INDEX option once it has been set.

Examples:

```
$$ SET INDEX PROCEDURES, INDEX IDENTIFIERS

$$ INDEX ALL

$$ INDEX MOUDLES, INDEX TYPES

$$ SET INDEX PROCEDURES
```

9. Default MID

Currently, the SPRITE user always has to write a MID and provide his own bind deck to get the excutable code file even if the user has only one module in the program. With the new SPRITE, the user can avoid going through the hassle of creating the MID and bind deck by using "BD" or "BN" in the first slash parameter. To use this convenient feature, the following procedures must be followed.

(1) To invoke the default MID, you must use **"DFTMID"** as your system file name.

(2) To invoke the default bind deck, you must use **"BD"** or **"BN"** as the first parameter to your program for the debug and non-debug version, respectively.

(3) The control card "FILE ICM =" **must** be included in the compile deck.

(4) The name of the module and program entry point procedure must be "main" and "driver", respectively.

(5) Default name for the created codefile is "CODFIL". The user may use "$CODEFILE" to specify the name of the result codefile.

Example:

```
%? COMPILE ONEMOD  WITH SPRITE/BD .. debug version
%? FILE SYSTEM = (DFTMID)         .. default SYSTEM
%? FILE ICM    = (MODICM)         .. ICM card must
%? DATA CATD
$$ CODEFILE "TESTxx"              %  codefile name will
                                  %  be TEST under xx
main         % name of the module
MOD
      ......

      driver     % name of the program entry point

      ......

DOM;
```

NOTE: SPRITE will fire off a bind job for you if there are no errors on the compilation. The name of the bind job should be "XXXXXX/BINDER", where XXXXXX is the name of the codefile.

10. $XREF enhancement

The crossreference listing from $XREF will now indicate where an indentifier is modified.

For your own copy, please do "SYS COMP 8987:APDL ON ALS".

1. Lines left on page when calling print.line to print before or after advancing

When print.line is called specifying "print before" or "print after", the lines left on page is no longer calculated wrong.

1. Faster debug

The new version of debug will run considerably faster than the old version. However, you do lose something. The cumulative counts for each procedure all no longer kept. If you count them, you have to use a new option "C".

If you use the "slash option" C or "COUNTS" in a extended deck then you will get what you used to get. Note that even without the "C" option any procedures you mention with give you cumulative counts.

Note that unless you want the counts you can use a samller version because much less information has to be kept. Also the old version of debug killed the program if it ran out of space in its tables. This version will just switch off the "C" option and continue.

2. v_dbwrite

The new intrinsic, v_dbwrite, has the function identical to that of the f_dbwrite. The first parameter of V_dbwrite is VALUE parameter. This means that when calling v_dbwrite procedures SPRITE will not put the constant labels into the CONST pools.

If you are a heavy user of v_dbwrite.string you may want to consider that the second parameter has not been changed to a VALUE parameter and can still be up to 99999 characters long and will go into the CONST pool.

The new intrinsic INCLUDE library (SINLIB) contains the moudle description for this new intrinsic. To include it, do

```
$$ INCLUDE "SINLIB" dbwrite_types
$$ INLCUDE "SINLIB" v_dbwrite
```

TABLE OF CONTENTS

Appendix L SPRITE FOR OMEGA

L.1 Introduction

Existing programs and code files will continue to compile and run on P-series machines without change. The 7.0 release of SPRITE supports writing part of the operating system (MCPX) in SPRITE, producing OMEGA code files. The 7.1 release will allow others to build OMEGA code files.

When writing a small OMEGA program (defined below), there are several differences from non-OMEGA programs. (With OMEGA, a "small" program can have up to half a million bytes of data. Thus practically all existing programs are considered small. Most users need only read this list, and will not be affected by the rest of this appendix.)

a. A "$$ PAGING" card must be added to the MID.
   This tells SPRITE to build an OMEGA ICM for the MID, and all modules compiled with that MID.

b. A pointer variable cannot point to a constant.
   If this is needed, just declare (and point to) a dummy data block variable which is initialized to that constant. (See L.4.5 for the reason this restriction is necessary.)

c. The internal mapping of pointers is different.
   This only affects those who build or manipulate pointers by hand (see L.4.1).

d. Files and intrinsics are not implemented yet.
   The MOVE intrinsic is now generated in-line for OMEGA. The standard procedure mcp.move_repeat should help take the place of the FILL intrinsic (see L.7.3).

e. Any BPL modules must be rewritten in SPRITE.
   There is no BPL for OMEGA.

f. OMEGA ICMs are bound with LINKER, not BINDER.
   BINDER does not support OMEGA ICMs. Also, LINKER is much faster.

These differences also apply to large OMEGA programs (defined below). However, a pointer can point to a constant if it has the appropriate SEG clause (see L.4). Also, several new features were added to allow the user to take advantage of the flexibility of P-series machines. For details, read on.

L.1.1 Background

These terms are defined within the context of SPRITE. For a more complete description, see the OMEGA documentation.

With OMEGA, a program is divided into two or more **segments** of up to half a million bytes each which can be scattered throughout memory. (Note: the OMEGA documentation uses the word "area" rather than "segment".) Up to eight memory segments are accessible at any one time. They comprise the **current addressing environment,** and are specified by the **active memory area table.** The entries in this table (and thus the corresponding segments) are numbered zero through seven. Segment zero contains the stack and index registers (among other things). Segment one contains the currently executing code (and its constants, with SPRITE programs). The rest (if they exist) hold miscellaneous data.

With an OMEGA program, LINKER puts the code and constants in as many code segments as necessary. In a **small** program, the remaining data fits into a single segment (zero). Thus even when there are several code segments (and thus several environments), they all share the same non-constant data, and they always find it in segment zero. A **large** program can have any number of data segments, with up to seven in each environment. A given data segment might appear as segment two in some environments, as segment three in others, and not at all in the rest.

L.1.2 OMEGA Pointers

With OMEGA, the high-order two digits in a pointer include the **dimension override** of zero to seven. This is the index into the active memory area table for the segment containing the object pointed to. The pointer may not be valid outside the environment in which it was built, however, since the dimension override may no longer refer to the same physical segment. This includes passing parameters by reference, since that is implemented by passing a pointer to the actual parameter.

For large programs, we therefore provide the following constructs: declaring segments, declaring environments, and pointer SEG clauses. Together, they allow the user to declare many different environments, and still share pointers between them safely. We also include several other features to allow the user (but mainly the MCP writer) to exploit the flexibility of P-series machines.

L.1.3 Sample MID and Module

These examples should help clarify the following explanations. Refer back to them as you read the text.

```
$$ PAGING
prog
PROG ACCESSES (seg_zero ORIGINAL); % use "prog" as
                                   % SEG_TABLE name

seg_zero
SEG
    prm_seg_zero          % program reserved memory
    DATA
        filler    STRING (40) OF HEX,
        topstack  0..999999;       % 6 UN

    global_data
    DATA
        info_list        LINK TO SEG seg_zero INFO;
GES;

SEG_TABLE
    lex_parse_table (seg_zero, , lex_parse_seg ORIGINAL),
    sem_table       (seg_zero, , sem_seg       ORIGINAL),
    code_gen_table  (seg_zero, , code_gen_seg  ORIGINAL);
...

lex
MOD ACCESSES lex_parse_table;
    get_token PROC RETURNS TOKEN; % uses lex_parse_table
DOM;

utility
MOD       % uses program's SEG_TABLE ( prog )
    list_info PROC (i   INFO);    % uses prog
DOM;
GORP;
```

```
lex
MOD ACCESSES (seg_zero,                    % 0
               ,                           % 1
               lex_parse_seg,              % 2
               lex_seg        ORIGINAL)    % 3
lex_seg
SEG
    STATIC;   % forces all STATIC variables in lex_seg

    lex_seg_data
    DATA
         lex_seg_info   INFO;
GES;

lex             % module name forces this into code segment
SEG
    code_seg_data
    DATA
         code_seg_info  INFO;
GES;

more_zero_data      % no SEG/GES, so goes in seg_zero
DATA
    ptr_to_code_seg      PTR TO SEG lex INFO,
    ptr_no_seg_clause    PTR TO         INFO;

get_token
PROC RETURNS TOKEN;

    SHARES lex_seg_data, code_seg_data, more_zero_data;

    VAR  token_start,
         token_end       0..80 STATIC := 0;  % go in lex_seg

    ptr_to_code_seg   := ptr (code_seg_info);     % okay
  % ptr_no_seg_clause := ptr (code_seg_info);     % illegal

  % utility.list_info (code_seg_info);            % illegal
  % utility.list_info (lex_seg_info);             % illegal
         % should change formal param to pass by VALUE
...

CORP;
DOM;
```

L.2 Declaring Segments

In a small program, data is declared as before. SPRITE puts constants in segment one, and the rest in segment zero.

In `a large program, MID data must be explicitly declared within a specific segment. If module data is declared as before, SPRITE handles it the same way as in small programs. If desired, the user can force SPRITE to put module data in a specific segment.

L.2.1 Segment Declaration in MID

```
mid segment declaration
|                                   ________ ; ___________
|          segment:            /                program    \
\_knows____ident___SEG__\__knows____component__/___GES__
 \_____/                      \_______/_____________/         \
                                                               |
```

For a large program, a segment appears in the MID as a collection of data and file blocks (including port and nsp files), each of which can have its own knows list (as long as it is a subset of the segment's knows list). (As a convenience, a segment declaration may actually include any program component except a segment or module declaration.) If two segment declarations use the same name, their data is simply combined into the same segment. This allows decomposition of a segment into logically distinct parts.

For large programs, every data block must be declared within some segment; free-standing blocks are not allowed. If declarations within a segment have their own knows lists, the lists must be a subset of the segment's knows list. Also, REMAPS declarations can only remap data blocks in the same segment (see L.5.5). File declarations can only appear in segment zero (see L.3). Explicitly declared intrinsic data must go in segment zero. Finally, when declaring segment zero, the stack must not be mentioned; it is supplied implicitly by LINKER.

Files and intrinsics are not implemented for OMEGA in the 7.0 release.

L.2.2 Segment Declaration in Module

```
module segment declaration
|                                                ______ ; ______
|   segment:                                    /     module      \
\____ident____SEG___STATIC____\___component___/____GES__
                     \______/     \_____________/            \
                                                              |
```

If module data is declared as before, constants go in segment one and the rest go in segment zero. In a large program, a user can override SPRITE's default allocation with a segment declaration. The declaration may appear in the same place as a normal module data block declaration. It may either add data to an existing segment (by using a segment name which has already been defined), or define a new module local segment.

Declaring a module data or file block in a segment declaration forces SPRITE to allocate that block within that segment. If the segment name is the same as the module name, the data is put into the current code segment (but see L.4.5). The keyword "STATIC", if used, forces all following STATIC variable blocks into the segment being declared. This remains in effect until overridden by another segment declaration with "STATIC".

As a convenience, a module segment declaration may contain any module component except a segment, proc, or macro declaration. Again, files can only be declared in segment zero. (Remember, files are not implemented in 7.0.)

L.3 Declaring Addressing Environments

In a small program, segment zero is always the same; it contains all the non-constant data. At any point in time, segment one contains the currently executing procedure and its constants. Thus each procedure's environment is very simple and obvious; no explicit declaration is needed.

In a large program, a procedure can access at most seven of the declared data segments (in addition to the code segment). This group of data segments is called a

SEG_TABLE. An ACCESSES clause specifies which SEG_TABLE a given procedure uses. These constructs are only allowed in large programs.

A SEG_TABLE that never appears in an ACCESSES clause represents a dummy memory area table (see the OMEGA documentation). A SEG_TABLE that appears in an ACCESSES clause directly represents a procedure's active memory area table when the procedure starts executing. (The MCP writer can change the table within the procedure; see L.3.4.) The SEG_TABLE does not include the code segment, which SPRITE supplies implicitly.

```
seg table definition
|                                   _______ , ______
|                     seg table:   /                \
\____ SEG_TABLE _____ident_____\ ___seg table___/_______
                                                        \
                                                        |

seq table
|                                 ___ , ______/99\____
|                                /    seg:            \
\____________SYSTEM___ ( __\__ident___ORIGINAL__/__ ) __
             \______/         \_______\__________/       \
                                                          |

accesses
|
\____ACCESSES_____seq table: ident_____________________
                                                       \
                                                       |

accesses with decl
|
\____ACCESSES_____seq table_______________________________
                                                          \
                                                          |
```

A block of SEG_TABLE declarations can appear in a large program wherever a data block declaration can appear. The optional SYSTEM clause specifies an operating system table, rather than the default user table. (This clause should only be used in the operating system, of course.) The optional ORIGINAL clause specifies the original table entry, rather than a copy of the original elsewhere. Each segment must appear as an ORIGINAL entry exactly once, in the MID or module in which it is declared. A segment cannot appear twice in the same SEG_TABLE.

SEG_TABLEs that never appear in an ACCESSES clause are dummy tables, which may have up to 100 entries. An

entry is either a segment identifier, or a spot left empty by using consecutive commas. Any SEG_TABLE that appears in an ACCESSES clause is a real table (potentially active), which can have at most eight entries. The entries are numbered zero to seven. Further, the SEG_TABLE cannot skip segment zero, and it must skip segment one (leave it empty). SPRITE supplies the code segment implicitly.

Forward-defined segments are allowed, but not forward-defined SEG_TABLEs. **At the top of a MID or module, the "accesses with decl" form must be used (if any).** It allows the programmer to declare and use a table at the same time. If there is no such clause after the word PROG in a MID, it is a small program. If there is no such clause after the word MOD in a module, SPRITE uses the clause for that module from the MID. This clause at the top of a module is for adding module-local segments to the MID table.

L.3.1 ACCESSES Clause in MID

In a large program, an ACCESSES clause declaring a SEG_TABLE must appear at the beginning of the MID immediately after the keyword "PROG". This specifies the default addressing environment. It is not allowed in a small program.

```
interface description
|
|  program:
\___ident____PROG____accesses with decl____program tail_____
                 \___________________/                      \
                                                             |
program tail
|                      _________ ; _________
|                     /                     \
\_________________\__knows___component__/___GORP___ ; _______
                   \______/_____________/          \___/     \
                    \_____comment_____/                       |
```

```
module description
|
|                                            ________ ; __ ______
|  module:                        env    /            procedure   \
\__ident__MOD__accesses____dep__\_knows__description_/__DOM_
               |               |\___/ \______/_____________/      \
               |  accesses |          \____comment_______  /       |
               \_with decl_/                                       |
                \_________/                                        |
```

```
procedure description
|
|  proc:                                      return
\__ident______PROC____parameters____value_____accesses______
             \          \__________/   \______/   \________/      \
              \__ENTRY______________________/                     |
```

```
proc pointer type
|
|                                            return
\__PTR___TO___PROC____parameters____value_____accesses______
          \__/          \__________/   \______/   \________/      \
                                                                   |
```

Each module uses the program's ACCESSES clause, unless it has its own as shown above. Each procedure uses its module's ACCESSES clause, unless it has its own as shown above.

ACCESSES clauses on individual procedure declarations and definitions are not implemented in the 7.0 release.

An ACCESSES clause is required in proc pointer types in large programs, and is not allowed in small programs. The proc pointer ACCESSES clause tells SPRITE the environment of the procedure being called when the proc pointer is dereferenced. SPRITE needs this to verify that the calling and called procedures share the same segment zero (see L.3.3). SPRITE also needs it to process the parameters, as in a normal procedure call (see L.4.4).

L.3.2 ACCESSES Clause in Module

In large programs, an ACCESSES clause declaring a SEG_TABLE may appear in the actual module definition immediately after the keyword "MOD". The table specified for the module in the MID must be an **exact subset** of the table in the module. That is, the module table must be the same as the MID table, except that new segments may be added in entries which were empty in the MID table. (The entry for segment one must remain empty.) This allows for module local segments. If there is no ACCESSES clause at the top of the module, SPRITE just uses the one specified for that module in the MID.

```
module
|                                         _____ ; _____
| module:          accesses          /     module      \
\__ident__MOD___with decl___\__component__/__DOM___ ; __
                  \_________/    \___________/         \___/ \
                                                              |
procedure header
|
\___PROC___parameters____return value____accesses_______
           \__________/  \____________/  \________/        \
                                                            |
macro header
|
\___MACRO____parameters______accesses___________________
             \__________/     \________/                   \
                                                            |
```

Each procedure and macro uses the module's ACCESSES clause, unless it has its own as shown above. The SEG_TABLE specified in the procedure's clause represents its active memory area table at the start of the procedure. The procedure header is part of both the forward and actual procedure definitions in the module. If an ACCESSES clause appears in either place, the exact same clause must also appear in the other place. If the procedure is a module entry point, its corresponding SEG_TABLE in the MID must be an exact subset of the SEG_TABLE in the module, as defined above. Proc pointer ACCESSES clauses are treated the same way as in the MID.

ACCESSES clauses on individual procedures and macros are not implemented in the 7.0 release. (For additional restrictions needed to implement this, see L.4.2.1 and L.4.4.2.)

A procedure or macro uses the SEG_TABLE specified by its ACCESSES clause. It can access only the data in the segments in its SEG_TABLE (and segment one), regardless of KNOWS lists. (Of course, it must still know the data to access it.) A macro can only be "called" by a procedure that uses the same SEG_TABLE. Also, a procedure can only call another user-defined procedure when they both use the same segment zero (and thus the same stack). Otherwise, an interrupt or hypercall is required (which are not implemented).

Finally, procedures with matching SEG_TABLEs can go in the same addressing environment. That is, LINKER can put their code in the same segment one, if it fits. If not, LINKER can use as many segment ones, and thus environments, as necessary. (If procedures in the same module are split between different code segments, however, the module's CONST and ACON pools must be duplicated in each segment.) Note that the only difference between these environments will be segment one. Thus with special provisions for data in segment one (see L.4.5), these environments can be considered the same for everything except the need for a non-local VEN and VEX between them (which LINKER handles automatically).

L.3.3 Multiple Segment Zeroes

There can be more than one segment zero. Procedures with different segment zeroes cannot call each other, however, because they do not share the same stack. Transfering control between such procedures requires an interrupt or hypercall, which are not yet implemented in SPRITE.

L.3.4 Multiple Environments Within a Procedure

There are two standard procedures to implement the APE opcode: mcp.make_page_table_entry_unused, and mcp.copy_page_table_entry. They allow the MCP writer to change environments by directly altering memory area table entries.

WARNING: These procedures must be used with extreme caution. SPRITE will always make its checks and generate code based on the declared ACCESSES clauses. SPRITE cannot keep track of changes made to the environment by calling these standard procedures.

L.4 Pointer SEG Clause, and LINK

SPRITE tries to make sure that pointers remain valid when shared between environments. For small programs, this means taking two precautions with constants (see L.4.5). Large programs are handled as described below.

Declaring each module's addressing environment as above is sufficient to allow passing parameters by reference (see L.4.4.1). To allow sharing pointer variables between addressing environments, pointer type definitions have an optional SEG clause. Alternatively, a special LINK pointer is sometimes useful, such as with the FIND statement (see L.5.1). (LINKs are also allowed in small programs, but then their only advantage is they are two digits smaller.)

```
pointer type
|                           segment:
\__PTR_____TO____SEG__ident_____CONST____EXTERNAL___type__
 \_LINK_/ \__/  \___________/  \_VAR_/  \_LOCAL__/       \
                               \___/    \______/         |
```

A pointer without a SEG clause can point into any segment in the current addressing environment, except segment one (see L.4.5). However, it cannot be accessed outside that environment (but see L.4.2.1 and L.4.4.2). A pointer with a SEG clause can only point to objects in the specified segment. However, it can be shared freely between environments.

A LINK with a SEG clause is treated just like a pointer with a SEG clause. A LINK without a SEG clause can only point to the same segment it is in. Thus no dimension override (see L.1.2) need be stored in the pointer itself. It can be shared between environments, as long as it is not moved to a different segment in the process (see L.4.3 and L.4.4.3).

To dereference a pointer or LINK, SPRITE moves it into an index register or mobile register, along with the necessary dimension override. (P-series machines have four **mobile registers,** which can be used like index registers one and two. See the OMEGA documentation.) For a pointer without a SEG clause, SPRITE just moves the entire pointer (since it already contains the proper dimension override). For a pointer or LINK with a SEG clause, SPRITE uses the dimension override for that segment in the current environment. For a LINK without a SEG clause, SPRITE uses the dimension override needed to access the LINK. Of course, the pointer or LINK cannot be dereferenced unless the segment it points to is in the current environment.

In a module, the SEG clause identifier can be the module name. This specifies that the pointer or LINK points to the active code segment. See L.4.5 for cautions regarding the use of this feature.

L.4.1 Data Mapping

Pointers still occupy eight digits, or fourteen if the base type is parametric (six for the length). The lower six digits contain the offset from the beginning of the segment (or "EEEEEE" when nil). The upper two digits of the pointer contain the HEX value "C", followed by the dimension override of the segment for the environment in which the pointer was built (or if the pointer is nil, possibly "CE"). LINKs occupy only six digits, containing just the offset (or "EEEEEE" when nil).

Note that two equivalent pointers with a SEG clause may point to the same place, and yet not compare as equal in a boolean expression. This would happen if they were built in different environments which used a different dimension override for the segment in the SEG clause. Also, if both pointers are set to nil, one might have the upper two digits set to "CE" while the other contains the dimension override. When comparing a pointer with the constant "nil", SPRITE only compares the low six digits. SPRITE could make similar special arrangements when comparing some pointer variables, but this would be impractical when comparing structures or arrays with imbedded pointers (see L.4.2).

L.4.2 When Optional or Required

SEG clauses are not allowed in small programs. In large programs, they are optional in a module, as well as when defining indicants and LINK variable types in the MID. They are required in the types of MID files and MID data block variables, when these types contain (non-LINK) pointers. (If this restriction is lifted, checks very similar to those described in L.4.2.1 will also apply to accessing MID data blocks and files.) That includes **imbedded pointers**, such as a structure with a pointer field. (That does not include the parameters to procedure pointers.) They are also required in the FIND and GENERATE statements (see L.5.1 and L.5.2). For the 7.0 release, they are required in pointer types of module entry point parameters. (For additional parameter checks needed when we lift this restriction, see L.4.4.2.)

These restrictions help to guarantee that pointers without a SEG clause cannot be shared between different environments (but see L.4.2.1 and L.4.4.2). UNIV parameters and omitted tagfields can bypass this; **pointer kludgers beware!**

L.4.2.1 ACCESSES Clauses on Individual Procedures

When we allow a procedure or macro to have a different ACCESSES clause than its module, (non-LINK) pointers without a SEG clause will require further restrictions for safe use. (For additional checks when parameters contain such pointers, see L.4.4.2.)

A pointer without a SEG clause in a module file or data block can only point to segments in the module's SEG_TABLE. To enforce this restriction, a procedure with a different SEG_TABLE than the module cannot have VAR access to file or data blocks containing a pointer without a SEG clause. It can have CONST access to such blocks only if the module's SEG_TABLE is an exact subset of the procedure's SEG_TABLE (see L.3.2). In effect, the two-way communication of VAR access forces each SEG_TABLE to be a subset of the other. That is, VAR access is only allowed when both SEG_TABLES are the same.

L.4.3 Coercions

Equivalent pointers have the same SEG clause, access (CONST or VAR), and level (not implemented), as well as equivalent base types. (Having the same SEG clause here includes neither pointer having one.) Equivalent LINKs have the same requirements. Also, if the LINKs do not have a SEG clause, they must be in the same segment to be equivalent.

Compatible pointers also must have equivalent base types. Coercing from VAR to CONST and/or EXTERNAL to LOCAL is allowed, but not the other way around. If both pointers have a SEG clause, they must have the same clause. If one pointer has a SEG clause and the other does not, the coercion is legal in either direction. Note that even for two types to be compatible, any imbedded pointers and LINKs must be equivalent, and thus have the same SEG clause. The above also applies to LINKs, including compatibility between a LINK and a pointer.

When coercing to a pointer without a SEG clause, SPRITE supplies the proper dimension override in the current environment for the segment the source points to. Of course, this segment must be in the current environment. It cannot be the code segment, however (see L.4.5).

When coercing to a pointer (or LINK) with a SEG clause, SPRITE verifies that the source points to the segment specified by the destination's SEG clause.

When coercing to a LINK without a SEG clause, SPRITE verifies that the source points to the segment the destination is in. Note that if the source is also a LINK without a SEG clause, they are only compatible if they are in the same segment. This does not apply to imbedded LINKs without a SEG clause. (This is very much like passing imbedded LINKs by VALUE; see L.4.4.3.)

Several of the checks for LINK coercions are not yet fully implemented. Neither is the check when coercing from a pointer without a SEG clause to a pointer with a SEG clause. Users should take care. Most of the compile-time checks are implemented.

L.4.4 Parameters

There are three special concerns when passing parameters for OMEGA. First, the parameter must be in a segment accessible by the called procedure. Second, if the parameter contains any pointers, they must be meaningful to the called procedure. Third, if the parameter contains a LINK without a SEG clause, moving it to a different segment invalidates the LINK.

If the calling and called procedures have the same ACCESSES clause, and segment one is not involved (see L.4.5), the first two special concerns are satisfied. The case of a call between different environments is detailed below. So is the third concern.

L.4.4.1 Checking Segment the Parameter is In

VALUE parameters are moved to the stack in segment zero, so they are not a problem (unless they contain a pointer or a LINK; see below). That leaves pass-by-reference parameters.

If SPRITE cannot tell at compile time which segment the actual parameter is in, and the calling procedure's SEG_TABLE is not an exact subset of the called procedure's SEG_TABLE (see L.3.2), SPRITE puts out a warning. (The code will only work if the segment turns out to be in the called procedure's SEG_TABLE.) This includes a parameter which involves dereferencing a pointer without a SEG clause, such as one of the procedure's own pass-by-reference formal parameters. An actual parameter in segment one cannot be passed by reference (see L.4.5). It is an error if SPRITE knows at compile time that the parameter's segment is not in the called procedure's SEG_TABLE. Otherwise, SPRITE builds and passes a pointer (to the actual parameter) with the proper override for the called environment.

It does not matter which segment the destination of a function result is in, however, unless the RETURN type is a LINK without a SEG clause (see L.4.4.3). If necessary, SPRITE builds a stack temporary for the RETURN parameter, passes a pointer to it with override zero, and moves the result to the real destination after the VEN. This is only necessary if the destination's segment is segment one, or it is not in the called procedure's SEG_TABLE, or it is not known at compile time and the calling procedure's SEG_TABLE is not an exact subset of the called procedure's SEG_TABLE.

L.4.4.2 Checking Pointer(s) in the Parameter's Type

The above coercion rules and restrictions on formal parameters are sufficient even when the called procedure is in a different module with a different ACCESSES clause. Any pointer (including an imbedded pointer) in an entry point's formal parameter type must have a SEG clause. If the called procedure dereferences the pointer but does not have access to the segment in its SEG clause, SPRITE will report the error when it is dereferenced. If the pointer in the actual parameter does not point to that segment, SPRITE will report the error during the coercion from the actual parameter to the formal.

Again, coercing from a pointer without a SEG clause to a pointer with a SEG clause requires a run-time check. This check is not made in the 7.0 release, so users should be careful.

When we make pointer SEG clauses optional on module entry point parameters, or when we allow ACCESSES clauses on individual procedures, further parameter checks will be needed. When an actual parameter contains a pointer without a SEG clause, it may point anywhere in the calling procedure's SEG_TABLE. For this pointer to be meaningful to the called procedure, the called procedure must have access to every data segment in the calling procedure's SEG_TABLE (if not more). If the called procedure can change the pointer and pass it back, the reverse is also true (that is, the SEG_TABLEs must be the same.) Even if only the RETURN parameter contains a pointer without a SEG clause (passed just one way), the SEG_TABLEs must still be the same. This lets SPRITE make the subset check when the segment the destination is in is not known at compile-time (see L.4.4.1).

Therefore, if a procedure has a CONST or VALUE parameter type containing a pointer without a SEG clause, it can only be called by a procedure whose SEG_TABLE is an exact subset of the called procedure's SEG_TABLE. Further, if a procedure has such a VAR or RETURN parameter, it can only be called by another procedure with the same SEG_TABLE. So SPRITE can enforce this, if an entry point has such a VAR or RETURN parameter, its SEG_TABLE in the MID must be the same as the one in the module; a subset is no longer sufficient (see L.3.2).

L.4.4.3 Checking LINK(s) in the Parameter's Type

A LINK with a SEG clause is handled just like a pointer with a SEG clause (see L.4.4.2). The problems involving a LINK without a SEG clause revolve around the segment the parameter is in. Many of these are covered in sections L.4.3 and L.4.4.1. In addition, moving such a LINK between segments invalidates the LINK. The consequences of this are detailed below.

First, in a large program, a formal VALUE parameter type cannot be a LINK without a SEG clause. A formal VALUE parameter is on the stack in segment zero, but the actual parameter is typically in a different segment. Rather than wait until processing the call to report the error, we simply outlaw it even when the actual parameter is in segment zero. Of course, if passing a LINK in segment zero by VALUE is desired, the user can simply put a SEG clause on the formal parameter, specifying segment zero.

Second, if the RETURN type is a LINK without a SEG clause, the segment containing the actual destination must be in the called procedure's SEG_TABLE. That is, SPRITE cannot build a stack temporary as it can for other RETURN types (see L.4.4.1).

Third, VALUE and RETURN parameters can contain imbedded LINKs without a SEG clause, and still be moved between segments. SPRITE assumes the user is really trying to pass the non-LINK fields, and will rebuild the LINK fields by hand. SPRITE may eventually generate optional code to fill such LINK fields with "AAAAAA" (for access error) when the parameter must be moved between different segments. This will catch the error if the user dereferences them without rebuilding them.

Fourth, if the formal CONST or RETURN parameter is a LINK without a SEG clause, the actual parameter must point to the same segment that the actual parameter is in. SPRITE does not allow the actual parameter to be a pointer (or LINK) with a SEG clause to a different segment. If it did, to match the formal parameter type, SPRITE would have to build a temporary LINK without a SEG clause in the segment specified by the actual parameter's SEG clause. This is impractical. By insisting the actual parameter point to the same segment that it is in, SPRITE can match the formal parameter type by passing along the address of the offset part of the actual parameter. If processing the RETURN parameter and the actual destination is a pointer, SPRITE will set the dimension override appropriately as part of coercing from a LINK to a pointer.

Some of the checks for LINK parameters are not yet fully implemented. Users should take care. Most of the compile-time checks are implemented.

L.4.5 Data in the Code Segment

A code segment contains several procedures and their constants. It may contain several modules, which may even use different SEG_TABLEs. Conversely, procedures using the same SEG_TABLE may be in different code segments, and thus have environments which differ only in segment one. This is obviously necessary when all the procedures using a particular SEG_TABLE cannot fit in a single code segment.

SPRITE is thus designed to verify (as far as possible) that each procedure has access to the data it needs, while leaving the LINKER free to assign procedures to code segments in any way it likes (subject to the bind deck specifications). This requires special care with data in the code segment. That includes constant pools as well as (in a large program) data explicitly declared in the code segment. (See L.1.3 for some examples of the following checks.)

L.4.5.1 Passing Data in the Code Segment by Reference

SPRITE will not let data in segment one be passed by reference, since there is no guarantee that both procedures are in the same code segment. However, constants are simply moved to the stack first.

SPRITE does not do the same thing with CONST parameters because of possible aliasing problems. That is, the procedure may also have VAR access to the variable (such as by sharing its data block directly), giving strange results.

L.4.5.2 Pointing to Data in the Code Segment

A pointer (p) without a SEG clause cannot point to data in segment one. That even includes constants and small programs.

This restriction is closely related to the first. If we allowed (p) to point to segment one, then passing (p@) by reference would violate the first restriction. SPRITE would not catch this error, because it cannot tell at compile time which segment (p@) is in. By making sure (p) points to one of the data segments, (p@) can be passed freely as long as the calling procedure's SEG_TABLE is an exact subset of the called procedure's SEG_TABLE (see L.4.4.1).

This restriction also applies to a LINK without a SEG clause. Though not strictly necessary, this is safer and more consistent.

L.4.5.3 Explicitly Declaring Data in the Code Segment

SPRITE makes the above checks. In large programs, **those declaring data in the code segment (or a pointer or LINK to it) must verify that the data and all**

procedures that share it (or dereference the pointer or LINK) go in the same segment in the bind deck.

SPRITE helps the user make this check by requiring that all data in segment one must be declared using the module name as the segment name (see L.2.2). Also, every pointer to segment one must have the module name in its type (see L.3.2). (UNIV parameters and omitted tagfields can bypass this; **pointer kludgers beware!).** Thus finding all occurrences of the module name should quickly lead to all places where something funny is going on with the code segment.

L.5 Statements

Several miscellaneous statements were changed or added for OMEGA and/or MCPX. These are described below.

L.5.1 FIND

With OMEGA programs, SPRITE can generate an SLT or STB for the FIND statement, which provide features not available with SEA. SLT supports searching over a linked list. STB supports searching an array with elements over 100 digits long. In addition to the find conditions SEA supports, SLT and STB both support: finding the maximum value; finding any or no bit matching a mask (see below); or finding the first value with one of the following relations to the key: ¬=, <=, >, >=.

```
find condition
|
|                                                          bit mask:
\__ ANY_ONE_BIT_IN ___find primary___MATCHES____expr____
 \_  NO_ONE_BIT_IN _/                                        \
                                                             |
```

If the ANY_ONE_BIT_IN or NO_ONE_BIT_IN form of FIND condition is used, all corresponding bits in each find primary and the specified bit mask expression are examined until an element is found which satisfies the match condition. A match occurs if any (ANY_ONE_BIT_IN) or no (NO_ONE_BIT_IN) pair of corresponding bits are both set. The bit mask must be a fixed length hex string the same size as the find primary.

For a linked list FIND in a large program, the types of the list head pointer and the link field must be equivalent. If they are not LINKs, they must have a SEG clause. Also, if the list head pointer (or LINK) has a SEG clause, it must be in the specified segment. These restrictions guarantee that the list head pointer and every element of the list are in the same segment, which must be true for the SLT instruction to work properly.

LINKs without a SEG clause were designed specifically for use with linked list FIND statements. This explains the strange restrictions on their use (see L.4). By definition, they point to the same segment they are in, which is what the SLT instruction needs. Since no SEG clause is needed to guarantee this, the same FIND statment can be used to search linked lists in several different segments. The user can simply pass the list head pointer (defined as a LINK) by reference to the procedure containing the FIND statement.

L.5.2 GENERATE

"GENERATE LOCAL" generates space on the stack, which is in segment zero. Thus if the pointer or LINK has a SEG clause, it must be segment zero.

"GENERATE EXTERNAL" generates space in the heap. In a small program, the heap is in segment zero. In a large program, each data segment can have its own heap. When generating with a LINK without a SEG clause, SPRITE generates heap space in the segment the LINK is in. It is an error if this segment is not known at compile time. When generating with a pointer in a large program, it is an error if the pointer does not have a SEG clause, because the SEG clause specifies which segment to generate in. The segment must be in the current environment, but not segment one. Generating with a LINK with a SEG clause is handled the same way as generating with a pointer with a SEG clause.

L.5.3 ALIAS

The ALIAS statement allows the user to associate a given SPRITE name or primary with an assembly code label. This can be useful when combining SPRITE and assembly modules.

```
alias statement
|                 ________________ , ____________________
|               /                          assembly label:   \
\___ALIAS___\___sprite name___ = ______literal_______/__
                                                         \
                                                         |
```

```
sprite name
|
|\__module: ident_____ . ___ procedure: ident___________
|                     \_________________________/        \
|                                                         |
|\__data block, seq, or seq table: ident_______________ |
|                                                        \|
|                                                         |
|                                        _____________    |
|                                      /               \   |
|\__data block variable: ident_____\__selection__/_____ |
|                               / \________________/    \|
\___indicant__________________/                          |
                                                         |
```

The ALIAS statement appears in the MID, without a KNOWS list. The assembly code label is an EBCDIC string literal up to six characters long. Only constant selections are allowed on the sprite name. That is, fields and constant indexing are okay, but not variable indexing nor pointer dereferencing.

L.5.4 OVERLAY

The operating system includes several overlay modules, which are written in assembly code. (They cannot be written in SPRITE, since once they are, they will no longer be overlay modules.) When calling an overlay module entry point, SPRITE puts out a VEN to BGOVL with the appropriate parameters. The operating system writer can specify which modules are in overlays by using the OVERLAY statement. It appears in the MID, without a KNOWS list.

```
overlay statement
|                          ________ , ________
|                        /                     \
\______OVERLAY________\___module: ident___/_____________
                                                         \
                                                         |
```

WARNING: Proc pointers can be initialized to point to overlay module entry points, since this is needed to supply procedure addresses to assembly code modules.

However, these proc pointers must not be dereferenced in a SPRITE module, since SPRITE will not generate the necessary BGOVL call. SPRITE does not catch this illegal use of these proc pointers.

L.5.5 REMAPS

The REMAPS definition allows the user to redefine the variables within a DATA block, without resorting to structures with omitted tagfields.

```
remaps definition
|
|  remap            target            _______________ , ___
|  DATA             DATA             /    ___ , ___        \
|  name:            name:            |   /   var:   \      |
\__ident__REMAPS__ident__ : ___\__\__ident__/__type__/__
                                                          \
                                                          |
```

The REMAPS definition may appear wherever a DATA definition may appear, but only in the MID. If the target DATA block is declared within a segment (see L.2.1), the REMAPS block must be declared within the same segment. REMAPS block variables cannot be initialized (but this might be implemented later on). The REMAPS block cannot be bigger than the target DATA block being redefined. To access REMAPS block variables, use the remap DATA name in a SHARES list, just as you would for a normal DATA block.

```
sm_io
DATA
    basic_definition    STRING (200) OF HEX := "0";

sm_io_for_keyboard_output
REMAPS sm_io:
    message_number      0..999,
    keyboard_command,
    message_received    BOOLEAN;
```

L.6 Implicitly Declared Data

Program_reserved_memory and iheap are implicitly declared by SPRITE. If the user must have access to their data, they must be declared explicitly in the user's MID. The names are different in large programs, however, as shown below.

L.6.1 Program_reserved_memory

Program_reserved_memory is the first fifty bytes in segment zero, containing such things as the index registers. If it is declared explicitly, and the declared block is bigger than that, the declared size is used instead.

In a large program, there can be more than one segment zero. Since the same name cannot be used for two different MID data blocks, program_reserved_memory cannot be declared by that name for each segment zero. Thus large programs must instead use: "prm_" + <the segment name>. This means the segment zero names have to be unique in the first 26 characters (30, less 4 for "prm_").

L.6.2 Iheap

In small programs, SPRITE maintains the heap with the implicit data block iheap.

In large programs, each segment (other than one) can have its own heap and iheap. To maintain unique names, each segment's iheap is named "iheap_" + <the segment name>. Note that the segment names must be unique in the first 24 characters (30, less 6 for "iheap_"). The size of each heap can be declared individually in the bind deck. If the program never tries to generate anything in a particular segment's heap, that segment's heap and iheap are not bound in.

L.7 Standard Functions and Procedures

The standard function proc_ptr and the type construct PTR TO PROC work somewhat differently for OMEGA. Also, OMEGA supports locks, which SPRITE implements with new standard procedures and types. These and several other new standard functions and procedures are described below.

L.7.1 Proc Pointer Types

Non-OMEGA procedure pointer types and the standard function proc_ptr are described in section 13.5 of the

SPRITE Reference Manual. Briefly, proc_ptr returns a reference to a procedure. Dereferencing the destination procedure pointer (followed by any parameters) calls that procedure.

For OMEGA programs, a procedure pointer is stored as a twenty-digit non-local environ pointer (ENVP in assembler). The value nil is represented as "000000000000CEEEEEEE".

For large OMEGA programs, procedure pointer types must include an ACCESSES clause. This clause is not allowed in other programs. SPRITE uses the ACCESSES clause to verify that the calling and called procedures share the same segment zero (see L.3.3). SPRITE also uses this clause to set up the parameters properly for the called environment, just as it does with a normal call (see L.4.4).

```
procedure pointer type
|                                                        seg
|                                     return             table:
\__PTR__TO__PROC__params____value____ACCESSES__ident____
       \__/      \______/  \______/ \_______________/  \
                                                        |
```

To be compatible, procedure pointers must have the same ACCESSES clause. (They must also have the same parameter list and RETURN type, as in non-OMEGA programs.) A procedure pointer can only point to procedures with the same ACCESSES clause. It cannot be dereferenced within a procedure which uses a different segment zero (and thus a different stack).

In a later release, coercions and the proc_ptr function could relax the ACCESSES clause restriction, as follows. The ACCESSES clauses need only be the same when there is a VAR or RETURN parameter containing a pointer without a SEG clause. If there is such a VALUE parameter, the destination proc pointer's SEG_TABLE must be an exact subset of the source's (or procedure's) SEG_TABLE. The same subset is required when there are any pass by reference or RETURN parameters at all, regardless of their type. Otherwise, the only restriction is that both SEG_TABLEs must use the same segment zero. These restrictions are needed so SPRITE can check and process the parameters (see L.4.4).

L.7.2 Lock Types

There are two different kinds of locks, as follows:

```
lock type
|
|\___LOCK_____level: constant simple expr_______________
|                                                /         \
\____EVENT______________________________________/          |
                                                           |
```

Data block variables, structure fields, and array elements can be defined as type LOCK (synchronization lock) or EVENT (event lock). For LOCKs, the level must be in the range 1..9999. Variables must be initialized with the predefined constant "init_lock". The only valid use of these variables is to pass them to the lock standard procs: (prog.)lock, lock_conditional, unlock, and test_event. LOCKs and EVENTs are twenty digits big.

```
data
DATA
    info      STRUC
                   flag           BOOLEAN,
                   flag_lock      LOCK 3
              CURTS     := [true, init_lock];
proc
PROC;
    SHARES    data;

    prog.lock_conditional (info.flag_lock)        % no ";" !
         IF   IN_USE
         THEN
              handle_busy_case;
         ELSE
              info.flag := false;
              prog.unlock (info.flag_lock);
         FI;
CORP;
```

L.7.3 Definitions

```
    proc_ptr
    PROC (proc     PROC_NAME)          % (proc) or (mod.proc)
        RETURNS PTR TO PROC ...;
            % returns proc pointer with same parameters,
            % return value, and ACCESSES clause (if any)
            % as the passed proc

    ptr
    PROC (primary       T)
        RETURNS PTR_OR_LINK TO T;
            % destination must be a PTR or a LINK

    ptr_add              % to efficiently step through an array
    PROC (source_ptr    PTR TO T,      % a pointer into
                                       % an array of T
          elems         UN_6)
        RETURNS PTR TO T;
            % dest_ptr := source_ptr + (elems * T.SIZE)
            %    (size rounded up to multiple of T's modulo)
            % arithmetic on offset part of pointer, with
            %    dest_ptr getting source_ptr's dimension
            %    override

    ptr_sub
    PROC (source_ptr    PTR TO T,
          elems         UN_6)
        RETURNS PTR TO T;
            % same as ptr_add except "-" instead of "+"

    scale_ptr
    PROC (number        UN_7,      % must include dimension
                                   %    override digit!!
          exponent      0..6)      % must be constant
        RETURNS PTR TO T;
            % dest_ptr := number *
            %    10 to the power of exponent

prog
MOD
    halt_breakpoint                            % HBK
    PROC (break_id,                        % AF
          mask          STRING (2) OF HEX); % BF
            % both must be constant


    read_timer                                 % RDT
    PROC RETURNS UN_20;
DOM;
```

```
mcp       % strictly for MCP use
MOD

                                              % ATE 02               A
    alter_environment_table_entry                                    A
    PROC (source_descriptor  UNIV P_STR_10_HEX,                      A
          dest_descriptor    UNIV P_STR_06_HEX);                     A
             % source_descriptor must be 10 digits long              A
             % dest_descriptor must be 6 digits long                 A
    bind_date                                       % CDAT           A
    PROC RETURNS UN_6;              % ddmmyy

    bind_time                                       % CTIM
    PROC RETURNS UN_6;              % hhmmss

    build_ptr_no_seg
    PROC (size,
          offset  UN_6,
          seg     SEGMENT_NAME)
        RETURNS PTR TO T;    % with seg's D.O.
             % Destination must be a PTR
             %    without a SEG clause.
             % If dest is not a parametric pointer,
             %    just pass size of zero.
             % If dest is a parametric pointer,
             %    pass size part of pointer
             %    (digit size, but byte size when
             %    parametric CHAR or EBCDIC string).

    build_ptr_seg
    PROC (size,
          offset  UN_6)
        RETURNS PTR TO SEG s T;
             % destination must be a PTR
             %    with a SEG clause
             % size is same as above

    context_addr                                    % ACON
    PROC (primary UNIV P_STR_1_TO_999999_HEX)
                                          % parametric HEX string
         RETURNS STRING (8) OF HEX;
             % like ptr but returns a
             % context address, with
             % controller and extend digit
```

```
convert_io                                        % CIO
PROC (initial_desc      UNIV P_STR_40_HEX,
      result_desc    VAR UNIV P_STR_30_HEX);
         % exception clause:
         %    IF INVALID_ADDRESS ...   % LEQ
         % params must be exactly
         %    40 and 30 digits long

copy_desc                                         % PTNM
PROC (seg_table SEGMENT_TABLE,              % SEG_TABLE name
      seg       SEGMENT_NAME)               % segment in seg_table
                                            %  SEG_TABLE
     RETURNS STRING (8) OF HEX;             % 6-digit EN,
         % returns a copy descriptor,       % 2-digit MAN
         % for COPY TO part of APE.
         % use module name as seg_table
         % name for the module's run
         % time environment

copy_page_table_entry                             % ATE 01         C
PROC (source_descriptor,
      dest_descriptor   UNIV P_STR_8_HEX);
         % params must be 8 digits long

env_ptr                                           % ENVP
PROC (proc           PROC_NAME,     % (proc) or (mod.proc)
      reserved       STRING (8) OF HEX)
    RETURNS  PTR TO PROC ...;
         % same as proc_ptr, except it
         % lets user specify the eight
         % reserved digits in the ENVP
         % (proc_ptr sets them to zero)

halt_branch                                       % HBR
PROC (branch_to_self    BOOLEAN);
         % must be constant
         % false means branch to next
         %    instruction
                                                                     A
hyper_call                                  % HCL                    A
PROC (fucntion_number        0..999,                  % must be conAtant
      params            UNIV P_STR_1_TO_19998_HEX);   % size must bA
                                                      % be mod 2     A
                                                                     A
                                                                     A
```

```
initiate_io                                      % IIO
PROC (channel        0..77,
      desc     UNIV P_STR_6_TO_30_HEX);
         % desc must be  6 to 30 digits long
         % exception clause:
         %    IF INVALID_IO ...         % HIGH

interrupt                                        % INT
PROC (request        REQUEST);
         % REQUEST is any user-defined,
         % non-packed symbolic with
         % up to 99 elements

interrupt_data                                   % INT
PROC (request        REQUEST,
      data     UNIV P_STR_99_HEX);
         % same as interrupt, except
         % allows data to be passed

io_complete                                      % IOC
PROC (channel              0..77,
      mast                 UN_6,
      digit_count    VAR UN_8);
         % exception clause:
         %    IF INVALID_IO ...         % HIGH
lock                                             % LOKU or
PROC (l_or_e   VAR LOCK_OR_EVENT);               % WAIT
         % param must be a LOCK or an EVENT

lock_conditional                                 % LOKC or
PROC (l_or_e   VAR LOCK_OR_EVENT);               % SETL
         % param must be a LOCK or an EVENT
         % exception clause:  IF IN_USE ...

make_page_table_entry_unused                     % ATE 00
PROC (descriptor   UNIV P_STR_8_HEX);
         % desc must be 8 digits long

move_data                                        % MVD
PROC (source         UNIV P_STR_1_TO_999999_HEX,
      dest       VAR UNIV P_STR_1_TO_999999_HEX);

move_repeat                                      % MVR
PROC (pattern        UNIV P_STR_1_TO_999999_HEX,
      dest       VAR UNIV P_STR_1_TO_999999_HEX);
         % dest's size must be an exact
         %    multiple of pattern's size
         % if either is variable length,
         %    only one MVR will be generated
         %    (with at most 100 repetitions)
```

```
mvs                                                  % MVS
PROC (afbf          UNIV P_STR_4_HEX,
      source_desc,
      dest_desc     UNIV P_STR_34_HEX,
      padding            0..2);
         % afbf and padding must be constants
         % descs must be 34 digits long
         % padding:  0 = pad with zeroes
         %           1 = no padding
         %           2 = pad with blanks

offset_ptr                                           % MADR
PROC (primary       UNIV P_STR_1_TO_999999_HEX)
     RETURNS UN_6;
         % returns just the offset part of
         % a pointer to the parameter

read_begin_address                                   % RAD 00
PROC (channel            0..77,
      dest      VAR UNIV P_STR_4_CHAR); % parametric CHAR string
         % exception clause:  IF BUSY ...
         % dest must be MOD 2, 4 bytes long

read_end_address                                     % RAD 01
PROC (channel            0..77,
      dest      VAR UNIV P_STR_4_CHAR);
         % exception clause:  IF BUSY ...
         % dest must be MOD 2, 4 bytes long

read_result_descriptor                               % RAD 02
PROC (channel            0..77,
      dest      VAR UNIV P_STR_4_CHAR);
         % exception clause:  IF BUSY ...
         % dest must be MOD 2, 4 bytes long

reinstate_task                                       % BRV
PROC (addr     PTR TO REINSTATE_LIST_ENTRY);
         % REINSTATE_LIST_ENTRY is a
         % user-defined structure
```

```
scan_descriptors                                       % SRD
PROC (list_head          UN_4,                    % AFBF
      complete_r_d  VAR R_D_PTR);                 % IX1
         % exception clause:
         %    IF IO_COMPLETE ...                   % NEQ
         % list_head is address of first
         %    result descriptor to check
         % complete_r_d is PTR (with or
         %    without SEG clause) to first
         %    R_D (user-defined structure)
         %    found that says I/O complete
         %    (if any)

set_timer                                              % STT
PROC (time           UN_20);

stack_overflow                                         % HCL 1
PROC (task           UN_4);

stop_failure                                           % FAIL
PROC (condition      UN_6,
      fatal          BOOLEAN);
         % both must be constant

system_id                                              % SST 01
PROC (dest       VAR UNIV P_STR_200_HEX);
         % dest must be 200 digits long

system_status                                          % SST 00
PROC (dest       VAR UNIV P_STR_200_HEX);
         % dest must be 200 digits long

test_event                                             % TEST
PROC (event          EVENT);
         % exception clause:  IF IN_USE ...

unlock                                                 % UNLK or
PROC (l_or_e     VAR LOCK_OR_EVENT);                   % SGNL
         % param must be a LOCK or an EVENT
```

```
    update_mast_address                                     % WHR 03
    PROC (addr           UN_9);

    update_memory_error_address                             % WHR 02
    PROC (addr           UN_9);

    update_reinstate_list_address                           % WHR 00
    PROC (addr           UN_9);

    update_snap_picture_address                             % WHR 01
    PROC (addr           UN_9);

    write_begin_end_addresses                               % RAD 09
    PROC (channel              0..77,
          source         UNIV P_STR_4_CHAR);
             % exception clause:  IF BUSY ...
             % dest must be MOD 2, 4 bytes long
DOM;
```