MODEL #'s: 30063, 30064, 30165

1. Summary

The purpose of the product release notice is to provide the user with specific information about the product which is not available in the product manuals (information may be repeated in subsequent release notices if the appropriate manual is not yet available).

Between revisions of the product, periodic updates to the product may be issued. The purpose of an update is to reduce the time required to respond to problems by providing a level of corrections which does not require a release of the complete product. Each update of a product release supercedes the previous update.

A release of the product consists of two parts, as defined below:

	Part Description	Part Number
1.	SP/Pascal Rev 2.20 Release Notice	085-000234-07
2.	SP/Pascal Rev 2.20 Release Media	Defined by this release notice in section 6A.

Included in this release notice are:

2. Environment

A. Prerequisites

MP/AOS SP/Pascal Model: 30063

MP/AOS-SU Operating System Rev 1.00.
MP/AOS Operating System Rev 1.30 or later.

AOS SP/Pascal Model: 30064

AOS Rev 3.20 or later.

AOS/VS SP/Pascal Model: 30165

AOS/VS Rev 1.50 (with all patches installed) or later.

AOS/VS SP/PASCAL Model: 30165/G on 96TPI mini diskette

Require AOS/VS Rev 5.00 or later.

B. Dependent Products

None

3. Fixes

1. A compiler bug in preserving the contents of FPAC's across the evaluation of conditional expressions when the expressions contained pointer checks has been fixed.

2. A compiler bug in the generation of temporary file names used by the compiler has been fixed. This problem prevented more than one concurrent compilation in the same working directory.

3. The word_count argument to the predefined procedures SET_BLOCK and BLOCK_MOVE and the byte_count argument to procedure BYTE_MOVE will now be range checked for values in 1..32767. This check will only be generated when the rangecheck option "R" is enabled.

4. A compiler bug in adjusting the order in which parameters were passed to parametric procedures and functions has been fixed. In previous revisions, the parameters were passed in reverse order from what the procedures and functions expected.

5. A runtime bug in converting from long_whole to integer has been fixed. This bug could generate an incorrect error: "CONVERSION ERROR: LONG_INTEGER OR LONG_WHOLE TO INTEGER OR WHOLE".

6. A compiler bug in sharing references to the first word of a record (either explicitly through a pointer or implicitly through a with variable) when the reference to the structure occurred just before a loop and inside the loop has been fixed. The compiler incorrectly preserved the first value and used it after the loop was finished.

7. A compiler bug when optimizing incrementation of a long_whole variable for variables that contained pointer dereferences or array subscripts has been fixed.

8. A compiler bug in handling indirect references in real expressions has been fixed. The bug occurred in expressions whose left operand contained the indirection and whose right operand required all the FPAC's.

9. A compiler bug in initializing string variables inside variant records has been fixed. The compiler generates code to reset the current length to zero and the maxlength to the declared size.

10. A compiler bug in generating assignments to long_whole variables of the form: X := X + small_constant, when the small_constant was not equal to 1 has been fixed.

11. A compiler bug in generating short-circuit evaluation of the conditional expression in a while statement has been fixed.

12. A compiler bug in folding frame relative array addresses has been fixed. Previously, references to arrays containing large elements from a stack frame in low memory could incorrectly address the base of the array.

13. A compiler bug in generating real constants from integer operands, e.g. const r = 10 / 2, has been fixed. Previously, using these expressions inside a structured constant or in a redefined declaration could cause the compiler to abort.

4. Enhancements

None

5. Notes and Warnings

1. Revisions 1.20 and later of SP/Pascal place storage for overlay module names (.ENTO values passed as arguments to OV?LD and OV?RL) in the pure data area of the program instead of page zero. Allocation of the overlay module names in the pure area allows more space for user data in page zero, and a greater number of overlay modules to be used in the program. To support this feature, all programs created by the MP/AOS binder must use revision 1.30 or later of the MP/AOS binder. Programs created by LINK under AOS or AOS/VS will function correctly.

2. The files CODEMERGE.PAS, CODEMERGE.PR, CODEMERGE.DOC, and CODEMERGE.CLI are provided as a working example of a SP/Pascal program. The CODEMERGE utility can be used to interleave the assembly language code listing and the source listing files produced by the compiler into one listing file that shows each source line followed by the code generated for it. An interleaved listing file provides a useful debugging aid. The file CODEMERGE.DOC contains additional information about the operating procedures and arguments for the CODEMERGE utility.

3. The O option for integer overflow checking is not implemented.

4. For cross development under AOS/VS, the SPCLINK macro must be used to bind programs for execution under AOS/VS. The MP/AOS binder cannot be used to bind AOS/VS programs. To bind programs under AOS/VS for MP/AOS use the MPAOS_BIND macro. For revision 1.60 of AOS/VS, patch number 42 must be applied to AGENT.PR. The assembly language file MERMES.TXT is supplied with the AOS/VS release to allow users to integrate MP/AOS and AOS/VS error messages.

5. The EXTERNAL ASSEMBLY designator is included in SP/Pascal for compatiblity with MP/Pascal. It generates the same calling sequence as the MP/Pascal ASSEMBLY designator. When calling an assembly language program from SP/Pascal programs only, the standard SP/Pascal calling sequence is preferred. It is more efficient than the ASSEMBLY interface and is generated like a Pascal routine simply by declaring the routine as EXTERNAL. See Appendix C in the manual.

6. For AOS Rev 4.28 on S/120's, patch #4 must be installed.

7. For AOS/VS Rev 2.00, patch #8, patch #25, and patch #36 must be applied to the agent. For AOS/VS Rev 3.00, patch #7 and patch #25 must be applied to the agent.

8. Using the /STANDARD switch may impact the efficiency and amount of runtime storage required for existing programs. Large structures are copied when passed as value parameters, and unpacked arrays of char are word instead of byte aligned.

9. A non-local GOTO cannot be used to transfer control from one task to another.

6. Product Organization

A. Software

MP/AOS SP/Pascal on 1.25MB diskette

Model: 30063Q

Status	Part Number	Description
R	062-000264-05	MP/AOS SP/Pascal

MP/AOS SP/Pascal on mag tape and cartridge tape

Model: 30063M/H/C

Status	Part Number	Description
R	071-000637-05	MP/AOS SP/Pascal

AOS SP/Pascal on 1.25MB diskette

Model: 30064Q

Status	Part Number	Description
R	062-000265-05	AOS SP/Pascal

AOS SP/Pascal on mag tape and cartridge tape

Model: 30064M/H/C

Status	Part Number	Description
R	071-000638-05	AOS SP/Pascal

AOS/VS SP/Pascal on mag tape and cartridge tape and 120 mb cartridge tape

Model: 30165H/C/J

Status	Part Number	Description
R	071-000704-04	AOS/VS SP/Pascal

AOS/VS SP/PASCAL ON 96TPI MINI DISKETTE

Model: 30165/G

STATUS	PART NUMBER	DESCRIPTION
R	081-000315-02	AOS/VS SP/PASCAL
R	081-000316-02	AOS/VS SP/PASCAL

AOS/VS SP/PASCAL ON 20 MB CARTRIDGE

Model: 30165B

STATUS	PART NUMBER	DESCRIPTION
R	061-000287-01	AOS/VS SP/PASCAL

B. Documentation

Status	Part Number	Description
R	069-400203-01	SP/Pascal Programmer's Reference

7. Documentation Changes

1. On page 201 in the discussion of calling sequences, remove the sentence: All other accumulators are undefined. Substitute the following: SP/Pascal assumes that the accumulators, AC0, AC1, and AC2, are preserved across a call to an assembly language routine. Thus, all assembly routines should be coded with a save and rtn or should save and restore any of these registers that it changes.

2. On page 56, add the following to the discussion of the CASE statement: The OTHERWISE keyword in a CASE statement may be followed by a list of statements not just one statement.

3. On page 58, add the following paragraph to the discussion of the FOR statement:

The initial and final expressions are treated as signed values when making tests for loop entry and termination. Thus FOR statements with whole-type control variables must be programmed carefully since whole values in the range +32768 to +65536 are treated as signed initial or final expressions. For example:

```
    FOR W:= 0 TO whole-exp DO S;
```

Statement S will not be executed if the value of whole-exp is greater than 32767.

4. On page 101, add the following sentences to the discussion of the INCLUDE facility:

The SP/Pascal compiler ignores all text appearing after the semicolon on the same line as an INCLUDE statement. Therefore, all INCLUDE statements must appear on a separate line from other source text in the program.

5. On page 140, the description of REAL2STR.PAS has been changed to provide conversion of double precision real values. The argument data type for parameter INPUT is now DOUBLE_REAL. Delete the note on the bottom of the page and change the last paragraph to:

The FSIZE parameter determines the type of the numeric represention. If FSIZE is greater than 0, a fixed-point notation is generated with FSIZE digits after the decimal point. Otherwise, scientific (E) notation is used. At most, fourteen non-zero digits are printed. When less than 14 digits are requested, the remainder are used for rounding. If WIDTH is not positive an error is generated. The minimum number of characters printed for scientific notation is eight. If the requested numeric representation does not fit into the output string, an error is generated. In this case, the output string contains the first MAXLENGTH characters.

6. On page 142, the name of the constant for conversion errors should be S2IN_CNVRT_ERR instead of S2IN_CONVRT_ERR for consistency

with the definition in include file STR2DINT.PAS. The same change should be made again on page 144 for consistency with the definition in the include file STR2SINT.PAS.

7. On page 145 and 146, the definitions of the functions contained in BOOLEAN.PAS should be changed to use WHOLE parameter and result type definitions, instead of INTEGER types.

8. On page 162, under the discussion of task stack size, add the following paragraph before the example program in the section:

The /STACK switch gives the static stack requirements for each user routine. In addition to this space, each task also has extra stack storage allocated as part of a default (hidden) stack frame, and references to SP/Pascal runtime routines may consume additional stack space not included as part of the static requirements. The stack requirements for the runtime routines may dynamically depend on the application program and its data. For this reason, fixed upper bounds on the stack usage cannot be provided for all the runtimes. The stack space needed for each task must carefully balance all of these factors. The /STACK switch may be used as an initial estimate of the minimum stack requirements. In most programs, the addition of 50 words to the minimum stack value should be sufficient.

9. On page 147, the description of the function DDDIV should be changed to:

The division function DDDIV returns the quotient and remainder of its arguments. The remainder is defined as r:= x MOD y.

```
EXTERNAL FUNCTION DDDIV (X,Y:DOUBLE; VAR R:DOUBLE):DOUBLE;

Z:= DDDIV(X,Y,R);
```

8. New Documentation

A. New documentation for revision 1.20

A.1 Long_whole and long_integer types
Add to chapter 3, Data declarations

A.1.1 Definition
Add to the section Predefined Simple Data Types

A long_whole is a 32 bit unsigned integer (32 significant digits).

A long_integer is a 32 bit signed integer (31 significant digits and a sign bit).

Long_wholes and long_integers are included in the set of ordinal types. Thus they can be used in the same places as the other arithmetic ordinals, integer and whole, except where noted. This includes 32-bit expressions using standard infix notation, constants, parameters, I/O, and other useful features.

A.1.2 Constants
Add to the section Numeric Constants

Long_whole and Long_integer constants are supported.

By default, unsigned ordinal constants are whole or long_whole; but if they are signed as in unary operations, then they are integers or long_integers.

Range for ordinal constants

Constant	Lower bound	Upper bound
1) whole	0	65535
2) integer	-32768	+32767
3) long_whole	65536	4294967295
4) long_integer	-2147483648 +32768	-32769 +2147483647

In most instances, arithmetic operations on constants follow the rules of implicit coercion (see section 3.4). But, the value of a constant does not in all cases denote the type of the constant. There is an overlap in the values of whole and long_integer constants (i.e. 32768 to 65535). These unsigned constants with bit 0 as a significant digit are treated specially. By default, these constants are wholes. But in an expression, the precision of these constants depends on the precision of the other operand.

A.1.2.1 Binary operations on unsigned constants (32678 to 65535)

  Constant Operator x   (e.g.   40000 + x)

For binary operations, if one operand is an unsigned constant in the range 32768 to 65535 (bit 0 is on), then its value depends upon the type of the other operand (x).

The constant is

1) a whole constant.

If x is an integer or a whole, then the constant is a whole and the operation is single precision. This is consistent with revision 1.10 of SP/Pascal.

2) a long_integer constant.

If x is a long_integer, then the constant is a long_integer and a 32-bit signed operation is used.

3) a long_whole constant.

If x is a long_whole, then the constant is a long_whole and a 32-bit unsigned the operation is used.

4) coerced to real.

If x is a real or double_real, then the constant is coerced to a real or double_real and the operation is real or double_real.

For coercions of whole constants to real or double_real, the whole constant is treated as an unsigned quantity as expected. However, for coercions of long_whole constants to real or double_real, the long_whole constants are treated as signed quantities because of hardware restrictions.

If both operands are constants in this range (e.g. 40000 + 50000), then by default they are wholes, and the result may cause an overflow. Note that explicit coercions can be used to prevent overflow.

A.1.2.2   Unary operations on constants (32768 to 65535)

  Operator Constant   (e.g. x := + 40000)

If the constant is signed and if it is in the range 32768 to 65535, then the constant is represented as a long_integer.

A.1.2.3   Long_whole constants
in the range 2147483648 to 4294967295 (bit 0 on)

Unary operations on long_wholes in this range are errors.
Exception: Unary - 2147483648 is acceptable. This is represented as as a long_integer word, 1b0.

A.2   Structured data types
Add to the section Structured Data types

Long_whole and long_integer types can be the element type of an array, a field type of a record, and the resolution type of a pointer.

Exclusions

1) No subrange types of long_whole or long_integer.

2) No set base-types of long_whole or long_integer. The set's base-type must be in the range 0..255.

3) No variant tag-types of long_whole or long_integer.
Variant tag fields must be in a subrange of 0..127.

4) No array index-types of long_whole or long_integer.

A.3 Expressions

Add to chapter 4, Expressions

A.3.1 Arithmetic operations

Add to the section Arithmetic Operators

* Addition
* Subtraction
* Multiplication
* Division
* Modulus

A.3.2 Relational Operators

Add to the section Relational Operators

All relational operators are implemented for long_wholes and long_integers. In rev 1.10, comparisons of whole constants were signed. As of rev 1.20, comparisons of whole or of long_whole constants are unsigned.

A.3.3 Overflow

Add to the section Compatibility Rules

Operations on wholes or integers (variables or constants) is treated the same as in revision 1.10. When there is an overflow, significant digits will be lost. However, overflow and underflow checking is always performed for long_wholes and long_integers.

Example 1. (no overflow)

```
whole1 := 30000;
whole2 := whole1 + whole1;
```

In this case, the result's accuracy is retained.

Example 2. (overflow)

```
whole1 := 30000;
```

```
whole2 := 40000;
whole2 := whole1 + whole2;
```

The result of this operation will not be 70000. Significant digits are lost.
30000 + 40000 = 4464 (in decimal)

Example 3. (explicit coercion to override the default and
force the operands to long_wholes)

```
whole1 := 30000;
whole2 := 40000;

1) long_whole1 := whole1 + whole2    or

2) x := long_whole( whole1 + whole2 );
```

The result of this operation will be 70000; but, it will be represented in two words. In the second case, the result of the operation will also be coerced to the result type of x.

A.3.4 Compatibility and implicit coercion

Add to the section Compatibility Rules

There are compatibility checks for expressions with long_whole and/or long_integer operands, which are coerced when necessary. If possible, expressions with a long_whole or long_integer operand and a whole or integer operand are evaluated using more efficient operations than an expression with two long_whole or long_integer operands.

A.3.4.1 Operand compatibility

A long_whole and long_integer operand is compatible with all arithmetic types. By default, in an expression, x + y, if x and y are different types, then either x or y or both may be coerced. Below are the rules for compatibility and the default coercions for an expression with a long_whole and/or long_integer operand. If one of the operands in signed, then the other operand will be coerced to a signed operand. If one of the operands is long, then the other will coerced to a long operand.

			O P E R A N D			
O P E R A N D	integer	whole	long_integer	long_whole	real	double_real
long_ integer	Convert integer to long_integer	Convert whole to long_ integer	OK	Convert long_whole to long_ integer	Convert long_integer & real to double_real	Convert long_ integer to double_real
long_ whole	Convert integer to long_integer & long_whole to long_integer	Convert whole to long_ whole	Convert long_whole to long_ integer	OK	Convert * long_whole & real to double_real	Convert * long_whole to double_real

* Because of hardware restrictions, conversions from long_whole to real or to double_real are the same as long_integer to real or to double_real.

A.3.4.2 Arithmetic assignment compatibility

For assignments, operand1 := operand2, if the types are different, operand2 may be coerced to the same type as operand1 or there may be an error. All arithmetic types except real and double_real may be assigned to long_whole and long_integer. Below are the compatibility rules and default coercions for arithmetic assignments.

	O P E R A N D 2					
	integer	whole	long_integer	long_whole	real	double_real
integer	OK	Convert [1] whole to integer	Convert [2] long_integer to integer	Convert [2] long_whole to integer	error	error
whole	Convert [1] integer to whole	OK	Convert [2] long_integer to whole	Convert [2] long_whole to whole	error	error
O P E R A N D 1 long_ integer	Convert integer to long_ integer	Convert whole to long_ integer	OK	Convert [1] long_whole to long_integer	error	error
long_ whole	Convert integer to long_whole	Convert whole to long_whole	Convert [1] long_integer to long_whole	OK	error	error
real	Convert integer to real	Convert whole to real	Convert long_integer to real	Convert [3] long_whole to real	OK	Convert double_real to real
double_ real	Convert integer to double_ real	Convert whole to double_ real	Convert long_integer to double_ real	Convert [3] long_whole to double_real	Convert real to double_ real	OK

1 If the whole option is on, perform a range check.

2 If conversion from a long_integer precision to a 16-bit ordinal is impossible without loss in accuracy, flag as an error.

3 Because of hardware restrictions, conversions from long_whole to real or to double_real are the same as long_integer to real or to double_real.

A.4 Program statements

Add to chapter 5, Program Statements

Exclusions

1) FOR loops can not have a long_whole or long_integer as the control variable. An alternative must be used. One possibility is a REPEAT or WHILE loop with a statement that increments the long_whole or long_integer value.

2) No case selectors or case constants of long_whole or long_integer.

A.5 Functions and parameters
Add to chapter 6, SP/Pascal Routines

Function result types and parameters may be long_wholes or long_integers. Constants that are passed as parameters are coerced to the appropriate precision automatically.

A.6 I/O
Add to chapter 7, Input/Output

Reading and Writing of long_wholes and long_integers to files of long_whole and long_integer and to text files are now supported.

A.7 Predefined routines
Add to chapter 9, Predefined routines

A.7.1 New predefined routines

* Procedure Block_move(source, destination, word_count)
* Procedure Byte_move(bp_source, bp_destination, byte_count)
* Function Get_byte(byte_pointer) : char
* Procedure Intds
* Procedure Inten
* Procedure Set_block(array, word_count, value)
* Procedure Store_byte(byte_pointer, char)

Routine name	Operation	Number of args	Type of args	Result Type if function
1) Block_move	moves a block of contiguous words from source to destination	3	array or record array or record integer or whole	
2) Byte_move	moves a block of contiguous bytes from source to destination	3	byte_pointer byte_pointer integer or whole	
3) Get_byte	Get the character that is pointed to *	1	byte_pointer	character
4) Intds	Interrupt disable	0		
5) Inten	Interrupt enable	0		

6) Set_block	sets an array of specified length in words to a value	3	array integer or whole integer or whole
7) Store_byte	stores the character in the location pointed to	2	byte_pointer character

*There is no check that the byte pointer is to a character.

Example 1.

```
Var buffer1, buffer2 : array[ 1..1024 ] Of Char;
```

Block_move and Byte_move can be used to move a part of buffer1 to buffer2.

```
Begin

{ Move the first 100 bytes from buffer1 to buffer2 }
Byte_move( Byteaddr( buffer1 ), Byteaddr( buffer2 ), 100 );

                    or

{ Move the first 50 words from buffer1 to buffer2 }
Block_move( buffer1, buffer2, 50 );

End;
```

Example 2.

Set_block can be used to initialize buffer1.

```
{ Initialize buffer 1 to nul }
Set_block( buffer1, 512, 0 );
```

Example 3.

Get_byte and Store_byte are useful when manipulating byte addresses.

```
Var bp_ch : Whole;

Begin

bp_ch := Byteaddr( buffer1[ 0 ] );

 ...

{ Get the character that bp_ch points to
  (i.e. buffer1[ 0 ]) and put it in buffer2[ 0 ] }
buffer2[ 0 ] := Get_byte( bp_ch );
 ...
{ Store buffer2[ 0 ] at the location
```

```
  pointed to by bp_ch (i.e. buffer1[ 0 ]) }
Store_byte( bp_ch, buffer2[ 0 ] );

End;
```

A.7.2 Modified routines
now allow long_whole and long_integer arguments.

Routine	Argument type	Result type
	Mathematical functions	
Abs	any arithmetic type	same as argument

Whole and long_whole arguments are treated as integer and long_integer arguments respectively. This is because the result of an Abs is considered to be a signed quantity.

Routine	Argument type	Result type
Arctan	any arithmetic type	Real or double_real
Cos	any arithmetic type	Real or double_real
Exp	any arithmetic type	Real or double_real
Ln	any arithmetic type	Real or double_real
Sin	any arithmetic type	Real or double_real
Sqrt	any arithmetic type	Real or double_real

If the argument is a long_whole or a long_integer then by default, double_real functions will be used, and the result will be double_real.

Routine	Argument type	Result type
Max	any ordinal or arithmetic type	same as arguments
Min	any ordinal or arithmetic type	same as arguments

Min and Max formerly only allowed ordinal arguments.

Coercion function

Routine	Argument type	Result type
Float	whole, integer long_integer, long_whole	real double_real

If the argument is a long_whole, it is treated as a long_integer because of hardware restrictions.

Ordinal functions

Chr	whole, integer, long_whole, and long_integer	Character
Ord	any ordinal type	integer
Pred	any ordinal type	same as argument
Succ	any ordinal type	same as argument

Routines to return addresses

Byteaddr	any variable	whole
Wordaddr	any variable	whole

Routines to return field size

Bitsize	type name	whole
Bytesize	type name	whole

Miscellaneous routines

Odd	any ordinal type	boolean
Sqr	any arithmetic type	same as argument

The functions Trunc and Round will always return an integer result. This is because there is no way of determining the result type based on the argument. To truncate to a long_integer, use the Longint function.

A.7.3 Functions for explicit coercions
Add to the section Type-handling Routines

Explicit coercions were implemented to give the user a full range of options to override default coercions. This gives the user the privilege of determining the cost of arithmetic operations (the more precision, the more costly the operation).

1) New functions

* Long_whole(expression)

evaluates the expression as a long_whole. The expression must be a non-real arithmetic type. If the argument is a real, it is an error.

* Long_integer(expression)

evaluates the expression as a long_integer. The expression must be an arithmetic type.

2) Long_whole and long_integer arguments

* Whole, Integer, Real, and Double_real(expression)

also accept long_whole and long_integer arguments.

3) Real arguments

* Integer, Whole, Long_whole (real expression)

Integer accepts a real argument, but Whole and Long_whole with a real argument is an error.

Example 1. x := Integer(whole1 * double_real1)

Whole1 will be coerced to a double_real by implicit coercion, the operands will be multiplied using double real arithmetic, and then the result will be coerced to an integer.

4) Expressions as arguments

* Real, Double_real (expression)

Previously, only the functions, Real and Double_real, would evaluate expressions according to the precision of the coercion function. Now, all the arithmetic type coercion functions will evaluate the expression according to the function type and precision.

* Whole, Integer, Long_whole, and Long_integer(expression)

a) If the expression is ordinal, then it will be evaluated according to the type and precision of the function.

b) If the expression has one or more real operands, then it is considered real. It will first be evaluated, and then the result will be coerced to the ordinal result type.

5) Assignments

An assignment to a long_whole or long_integer acts like an explicit coercion for the expression on the right side of the assignment. In this case, all ordinal expressions will be evaluated using long_whole or long_integer operations. Real expressions will be evaluated using real operations and then coerced to long_integer. An assignment of a real to a long_whole or to a long_integer is an error. For an assignment to long_integer (or integer), this can be circumvented by using explicit coercions.

For an assignment,
x := y + z, if x is long_integer or long_whole, then any ordinal operation, y + z, would be long_integer or long_whole respectively. If x is any other type, then the precision of y + z is not affected

by x.

Example 2. long_integer1 := integer1 * integer2

This assignment is equivalent to

```
long_integer1 := Long_integer( Integer1 * Integer2 )
```

Integer1 and integer2 will first be coerced to long_integers, and then multiplied. No precision would be lost.

6) Nested coercions

When nesting coercion functions, the ordinal function closest to the expression takes precedence over ordinal operations, and the closest real function takes precedence over real operations.

Example 3. Real1 := Real(Integer(whole1 * double_real1))

Double_real1 is coerced to real because of the explicit coercion. Whole1 is coerced to real because of the implicit coercion. The multiplication uses single precision real operations. The result is coerced to an Integer and then to a Real.

This is not the same as
 Real1 := Integer(whole1 * double_real1).
Since Real1 is a single precision real, the precision of the expression is not affected by it.

These coercions whether functions or assignments to variables of long_integer precision are particularly useful for operations that may overflow. By using them, one is assured that the operation will always be performed using 32-bit arithmetic.

Example 4.

```
a) long_integer1 :=  Whole( whole1 + long_whole2 ) +
                      integer1 * integer2

               (which is equivalent to)

b) long_integer1 := Long_integer( Whole( whole1 + long_whole2 ) +
                       integer1 * integer2 )
```

Long_whole2 is coerced to whole and added to whole1 using unsigned single precision arithmetic. Integer1 and integer2 are multiplied using signed 32-bit arithmetic. The results are then added using long_integer arithmetic.

6) Functions

Explicit coercions must be used directly in the actual parameter list to coerce the arguments. If the coercion surrounds the function call, then only the result will be coerced. Also, for the trigonometric functions, the function's precision is based on the argument's precision.

Example 5.

a) Double_real(Sin(real1))

This will compute the single precision sine of real1 and then force the result to double_real.

b) Sin(Double_real(real1))

This will compute the double precision sine of real1, and the result will be double_real.

A.7.4 Explicit coercions for compatibility with AOS/VS Pascal

* Function Longint(expression) : long_integer

is equivalent to Long_integer(expression).
It evaluates the expression as a long_integer. The expression must be arithmetic.

* Function Shortint(x) : Integer

is equivalent to Integer(expression) except that it will only accept arithmetic types. (Integer will accept all scalar types which includes booleans, characters, etc.)
Shortint evaluates the expression as an integer.

A.8 Compile-time evaluable predefined routines

The following predefined routines are now compile-time evaluable (if their arguments are constants):

* Arctan, Cos, Exp, Float, Ln, Longint, Round, Sin,
  Shortint, Sqrt, Trunc

in addition to the previous list:

* Abs, Bitsize, Bytesize, Chr, Length, Maxlength, Odd,
  Ord, Pred, Sqr, Succ.

The following type coercions functions are now compile-time evaluable:

* Long_whole, Long_integer, Real, Double_real

in addition to the previous list:

* Boolean, Char, Integer, Whole, enumerated types,
  and subranges.

A.9 Range checks

For all the compile-time predefined routines with constant arguments, there are range checks on the arguments.

A.10 Routines now predefined and generated in line

These boolean routines that were external assembly routines are now predefined and are generated in line. They can have any one-word ordinal as arguments.

* Xand, Xior, Xnot, Xshft, Xxor

A.11 DGC runtimes Add to chapter 10, External Routines Supplied by DGC

* DI2ST, ST2DI

The conversion functions for double to string and string to double allow long_whole and long_integer arguments. However, by default the argument will be unsigned. To specify a signed quantity, the option i2st_signd must be used.

A.12 Compiler option Add to chapter 13, Operating Procedures to the section Compiler Options

The whole compiler option has been extended to long_wholes for checking an assignment between integers and wholes.

A.13 Miscellaneous

1) All rules are the same for constant expressions in the declaration block and in executable code.

2) Variables of type long_whole and long_integer are word-aligned.

3) Three new error codes were added.

* epdsc
  Conversion error: Long_integer or Long_whole to Integer or Whole

* eplou
  Long_integer or Long_whole overflow/underflow

* ep1b0
  Whole <-> Integer conversion error

B. New documentation for revision 2.00

B.1 Label declaration

Add to chapter 5, Program Statements

A label declaration introduces integer constants, which must be in the range 0 to 9999, as statement labels. The form of a label declaration is:

```
        LABEL
                integer-const [... ,integer-const] ;
```

Every label declared in a label declaration must have a unique integral value and appear on exactly one of the statements in the routine that contains the label declaration. The SP/Pascal compiler will signal an error if a label is declared and not used on a statement. Labels follow the same scope rules as identifiers for constants, types, and variables. That is, a nested routine may declare a label with the same integral value as another label declared in its parent. Unlike other declarations that require integer constants, the labels in a label declaration must be a simple digit sequence. They may not be constant expressions or use an alternate radix specification. As with other declarations, SP/Pascal permits more than one label declaration to be specified in the declaration-part, and allows label declarations to be placed in any order with other declarations. However, labels are not permitted to be repeated in any label declarations appearing in the same block. (This feature, called benign redefinition is described in chapter 3 of the SP/Pascal manual.)

To define a label on a statement, use the form:

```
        label : statement
```

where label is one of the integer constants in the label declaration.

B.2 GOTO statement

A GOTO statement unconditionally transfers control to the statement identified by the label in the GOTO statement. For the transfer to be valid, the label must be on a statement satisfying one of the following conditions:

1) The statement contains the GOTO statement.
2) The statement is one of the statements in a compound statement containing the GOTO statement.
3) The statement is one of the statements at the outermost level of a routine containing the GOTO statement.

The above restrictions prohibit transfer of control into any conditional, iteration, exception, or compound statement. However, transferring to the beginning of one of these statements is possible. The last rule allows a transfer from a nested procedure to an enclosing procedure, provided the labelled statement is at the outermost level (i.e. not contained by any other statement) in the procedure. The syntax of the goto statement is:

```
        GOTO label
```

For example:

```
        procedure p;
        label 1,2,3,4;

            procedure nested;
            label 4,5,6;

            begin
                x := z;
            4:  if x > y then
                   begin
                       x:= x - y;
                       goto 4; { legal by rule 1 }
                   end;
                for i:= lower to upper do
                   if a[i] = x then
                      goto 6; { legal by rule 3 }
                goto 2; { legal by rule 3 }
            5:  y:= f(y);
                goto 4; { legal by rule 2 }
            6:  if i < max then
                   goto 5; { legal by rule 3 }
            end; { nested }

        begin { p }
        1:  nested;
            goto 3; { legal by rule 2 }
        2:  write('Fail');
            goto 4; { legal by rule 2 }
        3:  write('Ok');
        4:  { label on empty statement }
        end;
```

B.3 File buffers and procedures GET and PUT

Add to chapter 7, Input/Output

File buffers are special variables associated with each SP/Pascal file. The file buffer is implicitly defined as part of the declaration of a file variable and provides a low-level interface to a single component of the file. The file buffer is referenced like a pointer variable by specifying the name of the file variable followed by an up-arrow (^) or at-sign (@) character. For example:

```
        var
          f:text;
          g:file of integer;
```

```
     f^ is a file buffer variable of CHAR type, and
     g^ is a file buffer variable of INTEGER type.
```

In general, using file buffers is much less efficient than using a corresponding READ or WRITE operation. The READ and WRITE operations can be defined in terms of file buffer operations. For example:

```
        read(f,ch); { is equivalent to } ch:= f^; get(f);
        write(g,i); { is equivalent to } g^:= i;  put(g);
```

When a file is opened for reading, the file buffer variable is defined to be the next component to be transferred from the file. Thus, the file buffer provides a one component lookahead in the file. Referencing the file buffer does not change the current file position. To advance the file buffer to the next component, the predefined procedure GET is used. The procedure GET takes a file variable as its only argument. If EOF is true, then the value of the file buffer is undefined; and if GET is called, SP/Pascal will signal an end-of-file error. For example:

```
        rewrite(f);
        writeln(f,'abc');

        reset(f);     { f^ is now 'a' }
        ch:= f^;      { f^ is still 'a' }
        get(f);       { f^ is now 'b' }
        read(f,ch);   { ch is now 'b' and f^ is now 'c' }
        readln(f,ch); { ch is now 'c' and f^ is undefined }
```

When a file is opened for writing, the file buffer is used to hold the next component to be transferred to the file. An assignment to the file buffer does not change the contents of the file. To write the file buffer to the file, the predefined procedure PUT is used. After a PUT or WRITE operation, the value of the file buffer is undefined. The procedure PUT takes a file variable as its only argument. For example:

```
        rewrite(g);  { g is the empty file }
        g^:= 2;
        i:= g^ * j;
        put(g);      { g has one component, g^ is undefined }
        g^:= i;
        write(g,i);  { g has two components, g^ is undefined }
```

File buffers and the procedures GET and PUT can be used in place of the procedures READ and WRITE. However, for text files, READ and WRITE convert the file elements to the type of the argument. GET and PUT do not perform any conversion. For example:

```
        rewrite(f);
        writeln(f,'1');
        reset(f);
        i:= ord(f^);  { assigns i the value 49 }
        read(f,i);    { assigns i the value 1 }
```

B.4 PACK and UNPACK procedures
Add to chapter 9, Predefined Routines

The predefined procedure PACK initializes an entire packed array from an unpacked array having the same component type. The predefined procedure UNPACK assigns a slice of an unpacked array from an entire packed array having the same component type. In both of these procedures, the entire packed array is used, but not necessarily the entire unpacked array. The syntax for PACK and UNPACK is:

```
     PACK(unpacked-array-var, starting-index, packed-array-var)
   UNPACK(packed-array-var, unpacked-array-var, starting-index)
```

The starting-index always applies to the unpacked-array-var and specifies the starting position in the unpacked array for values to be taken (PACK) or assigned (UNPACK). The value of the starting-index must be assignment-compatible with the index-type of the unpacked array. For example:

```
        var
          x,y,z:packed array[1..10] of char;
          a:array[1..20] of char;
          i:1..20;

        begin
          pack(a,i,z);
          pack(a,1,x);
          unpack(z,a,10);
        end;
```

In SP/Pascal, only packed arrays of char have storage compacted for their representation (see the discussion of the /STANDARD switch). For all other structures, the packed attribute does not affect the storage allocation. Therefore, the PACK and UNPACK procedures should only be used on arrays of char. Note that, BIT qualified structures, defined on page 32 of the SP/Pascal programmer's reference manual, may not be passed as arguments to PACK and UNPACK. Instead, the programmer must write a simple loop to accomplish the same function. For example:

```
        var x:array[1..16] of boolean bit 1;
            a:array[1..16] of boolean;

        for j := 1 to 16 do
            a[j]:= x[j]; { is equivalent to unpack(x,a,1) }
```

B.5 Parametric procedures and functions
Add to chapter 6, SP/Pascal Routines

Revision 2.00 of SP/Pascal allows the programmer to declare formal parameters that are procedures or functions. The declaration of the parameters looks the same as a procedure or function heading. The form of the syntax is:

```
    formal-parmlist -> formal-parameter [ ;formal-parameter ]
    formal-parameter -> [ VAR ] id-list : parm-type | pfhead
    pfhead -> PROCEDURE id [ ( formal-parmlist ) ] |
              FUNCTION id [ ( formal-parmlist ) ] : result-type
```

For example:

```
        procedure p( procedure q(var i:integer);
                     function f(x:real):real );

        function g( a,b:char;
                    procedure print(procedure x) ):char;
```

Procedure p has two parameters. The first is a procedure that has a single VAR integer parameter, and the second is a function that returns a real value and has one real value parameter. Function g has three parameters. The first two are char expressions, and the third is a procedure that has another parameterless procedure as its only parameter. Calls to p and g might look like:

```
        p( aproc, afunc );
        ch:= g( 'a', 'z', someproc );
```

When a procedure or function is passed as an argument, the number and type of its formal parameters must match exactly with the number and type of the parameters for the argument it is being passed to. Only the names of the formal parameters are allowed to be different. For example, the following procedures could be passed as the first argument to procedure p in the above example:

```
        procedure level( var j:integer );
        procedure next( var index:integer );
```

Within a procedure or function, references to the procedure or function parameters are made exactly like references to any other routines. For example, inside function g, a call to procedure print would be:

```
        print( y ); { y is another procedure }
```

Only user-defined procedures or functions can be passed as parameters. To use a predefined procedure or function as a parameter, encapsulate it in another procedure or function. For example:

```
        function mysin( arg: real ): real;
        begin
            mysin:= sin( arg );
        end;

        function mycos( arg: real ): real;
        begin
            mycos:= cos( arg );
        end;

        { plot successive values for trigonometric functions }

        procedure graph( function trig( arg: real): real );
        var xvalue: real;

        begin
            read( f, xvalue );
            repeat
                plot( xvalue, trig( xvalue ) );
                xvalue:= xvalue + 0.1;
            until xvalue >= 10.0;
        end;

        begin
            graph( mycos );
            graph( mysin );
        end.
```

When a procedure or function is used as an argument, two pieces of information are passed. The first is the address of the procedure or function, and the second is an environment pointer that is needed to access non-local variables. Because it is necessary to pass the environment pointer, whether or not the routine contains references to non-local variables, the CLRE calling convention is used.

This rule means that any procedures or functions that are declared with the EXTERNAL or ENTRY qualifiers must also be declared with the CLRE qualifier. The SP/Pascal compiler will signal an error if the CLRE qualifier is not given. For routines that are local to a module or nested inside another procedure or function, the SP/Pascal compiler will automatically generate the CLRE calling convention. In the declaration of a procedure or function formal parameter, no qualifiers (EXTERNAL, ENTRY, ASSEMBLY, or CLRE) are allowed. For example, the following declaration is illegal:

```
        Procedure burrito( ch: char; external procedure taco );
```

B.6 New and Dispose with variant tags
Add to chapter 9, Predefined Routines

Revision 2.00 of SP/Pascal allows the form of NEW and DISPOSE with variant tags. Compile-time checks on the variant tags are made, but no storage economization on the size of the record object is performed. That is, according to the standard, only enough storage to fit the specified record variant could be allocated. SP/Pascal always allocates enough storage for the entire record.

The form of the calls to NEW and DISPOSE is:

```
        NEW(p,c1,c2,...,cn);
        DISPOSE(q,k1,k2,...,km);
```

where p and q are pointers to objects of a record-type; and c1...cn, k1...km are constants that give the value of successively nested variant tags of the record. The values of the variant tags are specified in order from the outermost variant to the variants nested within it. If a nested variant is specified, all enclosing variants must also have been listed. Variants at a deeper nesting level than the last variant specified are not required. Note that both tagged and untagged variant values are specified in calls to NEW and DISPOSE. For example:

```
        type tags = (a,b,c);
             rec  = record
                case t:tags of
                a,b:(case boolean of
                     true:(f:integer);
                    false:(g:real));
                  c:(h:boolean);
                end;

        var p:^rec;

        { no variants specified }
        new(p);
        { one variant specified, no others defined }
        new(p,c);
        { one variant specified, nested variant undefined }
        new(p,a);
        { both variants specified }
        new(p,b,false);
```

B.7 The /STANDARD compiler switch

Add to chapter 13, Operating Procedures

The /STANDARD switch is used on the command line to the SP/Pascal compiler to control the treatment of existing SP/Pascal language features that must be changed for conformance with the Pascal standard. The three features affected by the /STANDARD switch are nested comments, unpacked arrays of char, and value parameters.

B.7.1 Nested comments

Add to chapter 2, Lexical Structure

SP/Pascal allows nested comments by requiring that comments beginning with one form of bracketing symbol be terminated with the corresponding bracketing symbol. For example, comments that begin with a left-brace "{" must terminate with a right-brace "}", and comments that begin with a left-paren-asterisk "(*" must terminate with an asterisk-right-paren "*)". To conform with the Pascal standard, revision 2.00 of SP/Pascal treats the lexical alternatives for comments identically if the /STANDARD switch is used.

B.7.2 Unpacked arrays of char
Add to chapter 3, Data Declarations

SP/Pascal automatically packs (byte aligns) unpacked arrays of char for compatibility with MP/Pascal. The Pascal standard allows elements of unpacked arrays of char to be passed as VAR parameters. Because these elements may fall on odd byte addresses, SP/Pascal forbids passing them as VAR parameters. To rectify this problem, revision 2.00 of SP/Pascal word aligns unpacked arrays of char if the /STANDARD switch is used. Note: This change in allocation means that unpacked arrays of char will now occupy twice as much storage as in previous revisions.

B.7.3 Value parameters
Add to chapter 6, SP/Pascal Routines

Prior to rev 2.00, SP/Pascal passed references to the actual argument for records, arrays, and strings passed as value parameters, and prohibited assignment inside the procedure or function to all value parameters. Values of scalar-type (integer, whole, char, boolean, enumeration, real, double-real, long-integer, and long-whole), pointer-type, and set-type had copies made and passed either the actual value for single word types, or the address of a stack temporarily containing the copied value. Revision 2.00 makes copies of all arguments, including structured-type arguments, and allows assignment to value parameters if the /STANDARD switch is used. Existing programs that pass large structures as value parameters may need to be changed by passing the structures as VAR parameters if the /STANDARD switch is used. SP/Pascal makes a copy of the value argument in the stack of the routine doing the procedure or function call.

B.8 Other enhancements for standard Pascal conformance

B.8.1 Recursive pointer types
Add to chapter 3, Data Declarations

Prior to revision 2.00, SP/Pascal did not permit pointer type definitions of the form:

```
        p = ^p;
```

Such definitions are typically used as placeholders in top down design. Revision 2.00 of SP/Pascal allows these recursive pointer declarations.

B.8.2 Files of pointer-type
Add to chapter 3, Data Declarations

Prior to revision 2.00, SP/Pascal did not permit a file to be declared with a pointer-type as part of its element type. For example:

```
        type
          p = ^r;
          r = record link:p; data:integer end;
          f1 = file of p;
          f2 = file of r;
```

The declarations of f1 and f2 are permitted in revision 2.00. Note, however, that writing pointers out to a file does not guarantee that the pointer values will be valid when they are read back in. If, for example, some operations deallocating the storage are performed on the heap, then the pointer values in the file will be invalid.

B.8.3 Changing record variant tags

Add to chapter 3, Data Declarations

Prior to rev 2.00, on an assignment to a variant tag, the storage for the variant was cleared if it contained a pointer, string, or structure with bit or byte fields. For conformance with the Pascal standard, the storage for tagged record variants will no longer be initialized. For example:

```
        var
            rec : record
                      case t:tags of
                    tag1:(f:integer);
                    tag2,tag3:(p:ptr);
                  end;

        begin
            rec.t:= tag2;
            new(rec.p);
            rec.t:= tag3; { rec.p should still be valid }
        end;
```

Prior to rev 2.00, the value of rec.p would be cleared during the assignment of tag3 to the tag field. Because tag2 and tag3 identify the same variant, assigning tag3 does not change the active variant. The variant storage could only be cleared if the active variant is changed by assigning tag1 instead of tag3. However, detecting this condition is prohibitively expensive. Any programs that relied on the initial values of tagged variant fields will have to be changed.

B.8.4 Set compatibility checking

Add to chapter 13, Operating Procedures

The compatibility rules for assignment to a set variable require an extra checking operation to be performed at runtime. For example:

```
        var
          x: set of 0..20;
          y: set of 0..40;

        x:= y; { check that the members of y are in 0..20 }
```

Revision 2.00 of SP/Pascal has implemented a compiler option (the T option) to control the insertion of this runtime checking. The handling of compiler options is described in chapter 13 of the SP/Pascal programmer's reference manual. By default, the set checking option is disabled.

B.8.5 For-statement checking
Add to chapter 5, Program Statements

The Pascal standard requires that the initial and final value of a for-statement be assignment-compatible with the type of the control-variable, but only if the body of the for-statement is executed. For example:

```
        var
          i: 1..20;
          j,k: integer;

        for i:= j to k do loop-body;
```

If the above loop-body is to be executed (j <= k), then the values of j and k must be in the range 1..20. Revision 2.00 of SP/Pascal inserts these runtime checks if the R option for subrange checking is specified. (See chapter 13 for a discussion of compiler options.)

B.8.6 Text files not terminated by an end-of-line
Add to chapter 6, Input/Output

The Pascal standard requires that when a non-empty textfile is opened for reading (RESET) and the last character in the file is not an end-of-line character, then one is inserted. Revision 2.00 of SP/Pascal diagnoses this condition and inserts a null delimiter in the input stream.

B.9 Number of routines in a module
Add to appendix H, SP/Pascal Implementation Limits

Revision 2.00 raises the limit of procedures and functions that can be compiled in a single module from 64 to 256. This limit is used for allocating an internal table in the SP/Pascal compiler.

B.10 Nesting of Control Flow Constructs
Add to appendix H, SP/Pascal Implementation Limits

The maximum depth for the nesting of any combination of control flow statements is 20. In counting the level of a control construct, include the nesting level of the routine body containing the statement. Control flow statements are if-statements, while-statements, repeat-statements, for-statements, case-statements, and exception-blocks.

B.11 Floating point underflow handling
Add to chapter 13, Operating Procedures

Revision 2.00 permits the programmer to specify that floating point underflow errors are to be ignored by the program. To enable this feature, specify the /NO_UNDERFLOW switch on the SPCLINK or SPCBIND macro. When this switch is used, all floating point underflow errors cause the value of the floating point accumulator that generated the underflow to be set to true zero; the execution of the program resumes at the point the error occurred. If this switch is not used, then floating point underflow errors will raise an exception.

B.12 Standards compliance issues

Add a new appendix after appendix H

The Pascal standard requires that a conforming processor document the treatment of error conditions, implementation-defined and implementation-dependent features, and exceptions to the requirements of the standard. To obtain a copy of the Pascal standard, designated ANSI/IEEE770X3.97-1983, write to ANSI or IEEE in New York City.

B.12.1 Implementation-defined features

The standard requires that a compliant processor provide documentation on the definition of all implementation-defined features. There are currently eleven such features:

1) The correspondence between the set of alternatives for string-elements in character string constants and the values of the predefined char-type.

SP/Pascal allows any printable ASCII character to appear in a string constant. String constants may not span more than one line.

2) The set of representations for the predefined real-type and the subset of values of the real numbers.

SP/Pascal uses the standard Data General hardware representation for single and double precision real values.

3) The set of values for the predefined char-type and the ordinal values of these characters.

SP/Pascal uses the ASCII character set.

4) The actions that are performed and the points in the program that they occur to satisfy the post-assertions of the file-operations. (RESET, REWRITE, GET, PUT, READ, WRITE, READLN, WRITELN, EOF, EOLN).

SP/Pascal uses an implementation technique called lazy I/O. This technique allows some of the post-assertions for the file-operations to be delayed until the next I/O operation in the program. For example, RESET(f) opens f, but does not actually read the first element from the file until it is needed by the program. The advantage of lazy I/O is that on interactive devices, conversational prompting can occur. However, programs that rely on lazy I/O may not be portable.

5) The value of MAXINT is 32767.

6) The default value of totalwidth for integer, real, and boolean values.

TYPE	DEFAULT WIDTH
integer	1
long_integer	1
boolean	5
real	12
double_real	20

7) The number of digit-characters in the exponent of the floating point representation of a value of real-type is 2.

8) The default case of each letter in writing boolean values.

SP/Pascal writes the first letter in upper case and the rest in lower case, i.e. 'True' and 'False'.

9) The effect of PAGE(f).

SP/Pascal writes a form feed character to the textfile f.

10) The effect of a RESET or REWRITE on the predefined file variables INPUT or OUTPUT.

SP/Pascal permits these operations with the effects defined in the standard for other file variables.

11) The binding of program parameters of file-types.

SP/Pascal defines the binding of INPUT and OUTPUT to external devices or files that are environment dependent. For example the generic files @INPUT and @OUTPUT under AOS and the default channels ?INCH and ?OUCH under MP/AOS. Program file parameters other than INPUT and OUTPUT are not bound to external entities. For example, file parameters could be bound with additional arguments in the cli command line.

B.12.2 Standard error detection and handling

The standard defines 58 error conditions that a conforming processor must cover. A conforming processor is allowed to treat each error in one of four ways:

1) Document that the error is not reported.
2) Report the error during program compilation.
3) Report the error during program binding.
4) Report the error during program execution.

Many of the error conditions in the Pascal standard are detected at the option of the programmer by inserting runtime checks into the program. These checks include:

1) Array subscript checking
2) Record variant checking
3) NIL pointer checking
4) Subrange checking for ordinal-types
5) Subrange checking for set-types
6) Integer division by 0

The following errors are either automatically detected at runtime or are detected at runtime if the /STD switch on the SPCBIND or SPCLINK macro is used:

```
     1)  File mode for PUT, WRITE, WRITELN, PAGE. (/STD)
     2)  EOF true for PUT, WRITE, WRITELN, PAGE. (/STD)
     3)  File mode for GET, READ, READLN. (/STD)
     4)  EOF false for GET, READ, READLN.
     5)  NIL pointer argument to DISPOSE.
     6)  EOF true on call to EOLN.
     7)  Invalid case statement selector.
     8)  Real divide by 0.0.
     9)  Real overflow and underflow.
     10) SQR(x) and x does not exist.
     11) LN(x) and x <= 0.
     12) SQRT(x) and x < 0.
     13) Definition of TRUNC(x):
          (x >= 0) -> 0 <= x - trunc(x) < 1
          (x <  0) -> -1 < x - trunc(x) <= 0.
     14) Definition of ROUND(x) in terms of TRUNC(x) as:
          (x >= 0) -> ROUND(x) = TRUNC(x + 0.5)
          (x <  0) -> ROUND(x) = TRUNC(x - 0.5).
     15) Reading an invalid integer representation.
     16) Reading an invalid real representation.
     17) Total width or field width < 1.
```

The following general classes of errors are not detected by SP/Pascal:

1) Use of uninitialized or undefined variables.
2) Assigning or referencing an identified record variable created by NEW(p,c1,...,cn).
3) Errors in DISPOSE when the pointer object was created by NEW(p,c1,...,cn).
4) CHR(x) and x not in 0..255 (for non-constant x).
5) SUCC(x) and x > max value of the type of x (for non-constant x).
6) PRED(x) and x < min value of the type of x (for non-constant x).
7) Detection of integer overflow or underflow.
8) Unassigned function results.

B.12.3 Implementation-dependent features.

The standard defines nine implementation-dependent conditions that a conforming processor must recognize. A conforming processor is required to detect, in a manner similar to that specified for error conditions, any use of an implementation-dependent feature. That is, a processor must determine if a program relies on a particular definition of an implementation-dependent feature. (Note that, even for the same conforming processor, the definition of an implementation-dependent feature may be indeterminate. For example, the evaluation order of the operands of a dyadic operator may be done one way for some expressions and a different way for other expressions.) Because SP/Pascal performs common subexpression optimizations, detecting the following implementation-dependent features is not possible:

1) The order of evaluation of index-expressions of an indexed-variable.
2) The order of evaluation of the member expressions of a member-designator, e.g. [x..y], and the order of evaluation of the member-designators in a set constructor, e.g. [a..b,x..y].
3) The order of evaluation of the operands of a dyadic operator.
4) The order of evaluation and binding of the actual-parameters of a function-designator.
5) The order of accessing the variable and evaluating the expression in an assignment-statement.
6) The order of evaluation and binding of the actual-parameters of a procedure-statement.

The following implementation-dependent features are not detected by SP/Pascal:

7) The effect of inspecting a textfile to which the predefined procedure PAGE has been applied.

8) The binding of variables other than file type that appear in the program parameter list. The SP/Pascal compiler issues a warning message for any identifiers other than INPUT or OUTPUT that appear in the program parameter list. SP/Pascal does not define any binding for these identifiers.

9) The relationship between end-of-line characters and values of the predefined char-type. SP/Pascal uses the default delimiters null, newline, form feed, and carriage return as end-of-line terminators.

B.12.4 Exceptions to the standard

SP/Pascal complies with the requirements of the ANSI/IEEE770X3.97-1983 with the following exceptions:

1) The following additional reserved words may not be used for program identifiers:

   ASSEMBLY, BIT, CLRE, ENTRY, ERETURN, EXCEPTION, EXITLOOP, EXTERNAL, INCLUDE, MODULE, OTHERWISE, OVERLAY, RECAST, RETURN, and ZREL.

2) Enumeration types must have more than 1 and less than 256 enumeration constants.

3) Record variant tag types must be non-negative and in the range 0..127.

4) Set base types must be non-negative and in the range 0..255.

5) Threatening references to control-variables that occur in nested routines are not detected.

6) The lexical structure of a program is broken into text lines. The maximum length of a line is 136 characters, including the end-of-line delimiter. This convention places limits on the maximum length of certain language lexemes, such as the number of characters in an identifier and a string literal.

7) Files or structures containing files may not be defined in the variant part of a record.

8) Other restrictions and implementation limits defined in the SP/Pascal Programmer's Reference Manual.

The following SP/Pascal language extensions provide additional features that are defined as errors in standard Pascal programs:

1) Lexical extensions

SP/Pascal permits the use of the question mark, underscore, and dollar characters in identifiers. SP/Pascal has an angle bracket extension for specifying non-printing character constants. SP/Pascal has an extension for specifying non-decimal radix integer constants and allows underscore to be used in numeric constants. SP/Pascal uses the percent character to implement and ignore the rest of the line form of comment. SP/Pascal allows the relational operators >= and <= to be written as => and =<. As part of the built-in string data type, SP/Pascal allows a null string constant to be specified by juxtaposing two apostrophe characters, e.g. ''.

2) Use of NIL

SP/Pascal permits the reserved word NIL to be used in a constant definition part. The standard only permits NIL to be used in an expression contained in the statement part of the routine.

3) MOD operator

SP/Pascal allows any non-zero modulus and defines the MOD operator as:

```
a MOD b = a - (b * trunc(floor(a/b)))
```

4) Constant expressions

SP/Pascal allows the use of constant expressions in data declarations.

5) Ordinal coercion functions

SP/Pascal allows an ordinal type identifier to be used as a function reference to coerce an expression from one ordinal type to another.

6) Relaxed declaration ordering and benign redefinition

SP/Pascal allows declarations to be repeated and appear in any order in a block provided the standard Pascal rules on declaration before use are followed. In addition, as a part of the separate compilation and include file facilities, SP/Pascal permits constant, type, and certain forms of variable and routine declarations to be duplicated in the same block. SP/Pascal also allows the parameter list to be repeated on forward declared procedures and functions.

9. STR Reporting

When a problem is discovered with the SP/Pascal compiler or the SP/Pascal runtime environment, the following information should be provided to your Data General representative:

A. A complete description of the environment, including:

   a. The revision number of the SP/Pascal compiler.
   b. The name and revision number of the operating system.
   c. The hardware configuration. (if appropriate)

B. A detailed description of the problem, including:

   a. The characteristics identifying the problem.
   b. Any dependent system activities.
   c. The suspected cause of the problem.
   d. The results that the user expected.

C. Sufficient information to reproduce the problem, including:

   a. Any source files, programs, libraries, or macros.
   b. A description of the actions necessary to cause the problem.
   c. Any other information that the user feels will be beneficial.

If possible, every attempt should be made to reduce the problem to the smallest number of source lines or sequence of actions necessary to reproduce it.

When the problem is determined to be in the SP/Pascal compiler, be sure to supply all the source files and user-defined include files necessary to perform the compilation. When the problem is determined to be in a program, be sure to include the program and symbol table files, and any data files required to execute the program. If the problem is in a program that uses multi-tasking, a breakfile of the program should also be submitted.

10. INSTALLATION INSTRUCTIONS

The following procedure should be used to bring up a SP/PASCAL system from the 96TPI mini diskette:

Load the appropriate files from the supplied diskette in the desired directory using the following commands:

```
OP ON
LOAD/V @LFD:VOL1:SP_PASCAL
OP OFF
```

To LOAD model # 30165B (061-000287-00) 20 mb cartridge tape, use the following CLI command:

```
LOAD_II/V/DEL/BUFF=16384 @MTJn:0
```

Where "n" is the unit number on which you mounted the tape.

****

END OF RELEASE NOTICE

****