SCIENTIFIC XPL REFERENCE MANUAL

for

New England Digital Corporation's

ABLE SERIES COMPUTERS

June 1987

NEW ENGLAND DIGITAL CORPORATION
49 North Main Street
P.O. Box 546
White River Junction, Vermont 05001

REFERENCE APPENDICES

This manual describes the Scientific XPL language for version 6.00 and beyond (June 1987).

The material in this manual is for informational purposes and is subject to change without notice.

New England Digital Corporation assumes no responsibility for any errors which may appear in this manual.

Introduction to Scientific XPL

New England Digital's Scientific XPL is a high-level structured programming language that is used to write real-time programs for ABLE computers. Scientific XPL is an efficient and easy to use language that has been specifically designed for high speed scientific and industrial applications. Special features are included in Scientific XPL that allow data collection, bit manipulation, and other real-time functions to be easily performed. These features work together to provide an effective software tool that can be used to quickly and reliably solve challenging software problems.

Scientific XPL is one of a breed of modern programming languages that has become very popular with a wide range of computer users. A minimal subset of PL/I, Scientific XPL is a block-structured language that supports many variable types and includes powerful control structures such as IF statements and DO WHILE loops. These features of Scientific XPL make it vastly superior to older languages such as FORTRAN, COBOL, and BASIC.

Modern concepts of structured language programming are combined in Scientific XPL with the high speed and efficiency produced by a three pass optimizing compiler. This provides the programmer with the programming ease of a good high level language, yet also provides the ability to process data with an efficiency that on other systems is only available with assembly language programs. Scientific XPL has been used successfully in many applications where execution speed, program development time, program maintainability, and software reliability are important.

The efficiency of Scientific XPL derives from the three pass architecture that is used in the Scientific XPL Compiler. The three pass approach has many advantages when compared with the interpretive or single pass approach used on other systems. Since all of the symbol table processing is performed during the first pass, a great deal of memory is made available during the second pass for a powerful optimization routine. This routine creates a tree structure for each arithmetic expression which is then carefully inspected for common subexpressions, use of the multiply-divide unit, and so forth. Many Scientific XPL statements are compiled into a single directly executable machine instruction that takes on the order of one microsecond to perform.

Basic Features of Scientific XPL

A Scientific XPL program is a list of program statements. Most statements direct the computer to calculate an arithmetic expression, test for certain logical conditions, or perform input and output. Other statements allow the programmer to control the allocation of storage, create simple textual substitutions (literals), and define procedures. Scientific XPL is a block-structured language; procedures can contain further declarations which control storage allocation and define other procedures.

The procedure definition facility of Scientific XPL allows modular programming, so that a program can be divided into sections (e.g, numeric input, conversion from binary to decimal forms, or printing output messages). Each of these sections can be written as a procedure. Such procedures are conceptually simple, easy to formulate and debug, and easily incorporated into a large program. They can form a basis for a procedure library, if a family of similar programs is being developed.

Scientific XPL operates with two basic data types: fixed point and floating point numeric values. Fixed point numbers are stored inside the computer in a single 16-bit word of memory and are normally treated as signed integers in the range of -32,768 to +32,767 (narrower ranges apply in certain situations such as when performing arithmetic comparisons). A floating point variable is stored in two locations of memory and is used to represent a number in the range of (plus and minus) 5.50 E-20 to 9.00 E18. Two other data types are also available that are derived from the fixed point type: boolean and pointer. One-dimensional arrays (vectors) can be declared to let one variable name represent many data elements.

Executable statements specify the computational processes that are to take place. To achieve this, arithmetic, logical, and comparative (relational) operators are defined for variables and constants of all data types. These operators and operands are combined to form expressions which resemble those of elementary algebra. For example, the expression

```
x*(y - 3)/r
```

represents this calculation: the value of X multiplied by the quantity Y - 3, divided by the value of R.

Expressions are included in many Scientific XPL statements. A simple statement is the assignment statement, which computes a result and stores it in a memory location defined by a variable name. The assignment statement:

```
q = x*(y - 3)/r
```

first causes the evaluation of the expression to the right of the equals sign, as described above. Then the result of this computation is stored in the memory location associated with the variable name Q.

Other statements perform conditional testing and branching (IF), loop control (DO, DO WHILE), procedure invocation with parameter passing (CALL), interrupt processing (WHEN), and terminal input and output (INPUT, LINPUT, PRINT).

A particularly important feature of Scientific XPL is its real-time input/output capability. The READ function and WRITE statement transfer 16-bit quantities to and from interface modules connected to the system. In most cases, the data transfer occurs in a single machine instruction, requiring on the order of one microsecond to perform. The interface modules are each identified by a unique device address. The ABLE Series architecture allows for 256 devices.

A method of automatic text substitution (more specifically, a compile-time macro facility) is also provided in Scientific XPL. Symbolic names can be declared to be equivalent to an arbitrary sequence of characters. As each occurrence of the symbolic name is encountered by the compiler, the declared character sequence is substituted, so the compiler actually processes the substituted character string instead of the symbolic name. Such LITERAL declarations can be used to noticeably increase program readability and provide for self-documenting program statements.

A sample program that shows many features of Scientific XPL appears on the following pages. Notice that XPL statements end with a semicolon, and that spaces can be used throughout a program to improve readability. Comments begin with the character pair /* and end with */, as in the following example:

```
/* This is a comment, not a program statement */
```

```
/*  Statistics Program

    The following program will compute the average,
    standard deviation and median of a list of
    floating point numbers.

    Vector data structure:
      list(0) = n = number of elements in array
      list(1) ... list(n) = vector values
*/

dcl program_date data ('1 May 1987'); /* date of this program version */

dcl forever lit 'while (true)'; /* loop control */

dcl vector_length lit '200';    /* number of vector elements */

/* LEN returns the length of the vector, which is stored in the first
   word of the vector (list (0)).   */

len: proc (list) returns (fixed); /* compute length of vector */
   dcl list floating array;     /* array to process */

   return int (list (0));       /* return length of vector */
end len;

/* AVERAGE adds all the vector elements and then finds the average
   value of the vector.  */

average: proc (list) returns (floating); /* compute average of array */
   dcl list floating array;     /* array to process */
   dcl sum  floating;           /* local variables */
   dcl i    fixed;

   sum = 0;                     /* initialize sum */

   do i = 1 to len (list);      /* do for all values in the vector */
      sum = sum + list (i);     /* sum values */
   end;

   return (sum/len (list));     /* return average to main program */
end average;

/* STANDARD_DEVIATION computes the standard deviation of the vector. */

standard_deviation: proc (list) returns (floating); /* computes the standard deviation of vector */
   dcl list floating array;     /* array to process */
   dcl sum  floating;           /* sum of array elements */
   dcl ave  floating;           /* average of array elements */
   dcl i    fixed;

   if len (list) < 2 then do;   /* error - print message and stop */
      print 'Error in standard deviation - less than two vector elements';
      call exit (0);            /* exit the program */
   end;

   sum = 0;                     /* initialize sum */
   ave = average (list);        /* get the average */

   do i = 1 to len (list);      /* do for all vector values */
      sum = sum + (list (i) - ave)*(list (i) - ave);
   end;

   return (sqr (sum/(len (list) - 1)));
end standard_deviation;
```

```
67  /* $page */
68
69  /* MEDIAN sorts the list, and returns the middle value.  Not the
70       fastest algorithm but it works.  This algorithm no faster than
71       O(n log n) but known algorithms can do it in O(n) time. */
72
73  median: proc (list) returns (floating);  /* find median of vector */
74     dcl list floating array;     /* array to process */
75
76     /* This SORT procedure uses a selection sort - replace it with
77        a faster algorithm later if needed. */
78
79     sort: proc (list);           /* sort a vector */
80        dcl list      floating array; /* array to process */
81        dcl smallest  fixed;      /* index of current smallest element */
82        dcl temp      floating;   /* temporary storage value */
83        dcl (i, j, n) fixed;
84
85        n = len (list);           /* number of elements to sort */
86
87        do i = 1 to n - 1;        /* do one pass from each array location */
88           smallest = i;          /* assume first is smallest */
89
90           do j = i + 1 to n;     /* check the rest of the elements */
91              if list (j) < list (smallest) then smallest = j; /* if it's smaller, save it */
92           end;
93
94           /* place smallest in position I */
95           temp = list (i); list (i) = list (smallest); list (smallest) = temp;
96        end;
97     end sort;
98
99     call sort (list);            /* sort the vector */
100    return (list ((len (list) + 1)/2)); /* return the middle element */
101
102 end median;
103
104 /* USER_INPUT gets all the vector values from the user. */
105
106 user_input: proc (list);        /* get the vector values from the terminal */
107    dcl list   floating array;   /* array to process */
108    dcl (n, i) fixed;
109
110    print; print 'Enter the length of the data list',;
111    input n;                     /* get the length */
112
113    if n > 0 then do;            /* do only if there is something to input */
114       print 'Please enter data elements when prompted.';
115
116       do i = 1 to n;            /* get all the values */
117          print 'Value',;
118          input list (i);        /* put the value in the array */
119       end;
120       list (0) = n;             /* save length of vector in first slot */
121       print;                    /* print a blank line */
122    end;
123    else call exit (0);          /* exit the program if no length specified */
124 end user_input;
125
126
127 /**   MAIN PROGRAM   **/
128
129 dcl vector (vector_length) floating array; /* vector to process */
130
131 print 'Standard Deviation Program - ', string (program_date);
132
133 do forever;                     /* infinite loop */
134    call user_input (vector);    /* input vector values */
135    print 'Average            = ', average (vector); /* print the average */
136    print 'Median             = ', median (vector); /* print the median */
137    if len (vector) >= 2         /* must have at least two elements to do this */
138    then print 'Standard deviation = ', standard_deviation (vector);
139 end;
```

Section I - Constants and Variables

Numeric quantities in XPL can be either constants or variables. A constant always has the same value, while a variable's value can change during execution of the program. The expression:

```
x*(y - 3)/r
```

involves the constant 3 and the variables X, Y, and R. This section will describe the use of constants and variables in detail.

Data Types

Scientific XPL has four basic data types: FIXED, FLOATING, BOOLEAN, and POINTER. All of these types are stored in memory in a binary format. ABLE computers are 16-bit computers, which means that each location of memory can store 16 bits (a bit is a binary digit) of information. Each fixed point, boolean, or pointer data element is in a 16-bit, single word location; each floating point data element is in a 32-bit, double word encoded form.

Fixed point numbers can be interpreted as signed integers in the range of -32,768 to +32,767 or as unsigned integers in the range of 0 to 65,535. Signed integers are used mostly in arithmetic expressions while unsigned integers are often used as array indexes. Both signed and unsigned integers are processed identically inside the computer; the difference lies in the user interpretation of the bit patterns stored in memory. Unsigned integers allow the processing of larger numbers in a single location of memory, but do not have the capability to represent negative quantities. Some of the arithmetic operators available in Scientific XPL such as multiplication, division, and the relational operators are designed to be used strictly with signed quantities. These limitations will be explained in detail later on.

Floating point numbers are signed reals in the range of plus and minus 5.5 E-20 to 9.0 E18.

Booleans are logic values restricted to TRUE and FALSE. This data type is most often used for flags. In Scientific XPL, all even numbers evaluate to false and all odd numbers evaluate to true. Presently, BOOLEAN is an alias for fixed, TRUE is the number one, and FALSE is the number zero.

Pointers are memory addresses and thus are unsigned integers in the range of 0 to 65,535. Pointers usually contain the address of another variable. They are presently defined to be fixed point numbers. The constant NULL is provided to distinguish between active and inactive pointers. The actual value for NULL is zero.

The internal representation of all data types is discussed in the appendix "Internal Representations".

Constants

A constant is a value known at compile-time which does not change during execution of the program. Both numeric constants and string constants are supported.

Fixed point constants can be expressed in decimal, octal, or hexadecimal form. Decimal values are specified by a sequence of up to five digits that optionally start with a minus sign (-) indicating a negative quantity:

```
25
-14
10456
```

Octal constants are specified by a sequence of up to six digits (zero through seven only) enclosed in quotation marks:

```
"002476"
"773"
"177777"
```

Hexadecimal constants are specified by an H followed by a sequence of up to four digits (or the letters A through F), all enclosed in quotation marks:

```
"H0001"
"Ha1b"
"HFFFF"
```

Floating point constants are specified by an optional minus sign followed by up to eight digits, optionally followed by a decimal point and up to eight more digits:

```
3.14159
-12.0
152.77124455
.0001
```

Although a decimal point is not required in a floating point constant, it is good practice to always include one. This prevents the compiler from ever treating a floating point constant as a fixed point constant, which could cause overflow.

String constants are specified by a series of up to 128 characters enclosed in apostrophes:

```
'Hello there'
'This is a string.'
```

To include an apostrophe in a string, use two consecutive apostrophes:

```
'If you can do that, that''s great'
'Don''t touch that dial!'
```

String constants are most often used to send messages to the user (with the PRINT statement) and to define literals (macros).

Variable Identifiers

An identifier is used to name a variable, procedure, or statement label. An identifier can be up to thirty-two characters in length, and is composed of letters, digits, or the characters number sign (#), dollar sign ($), period (.), and underscore (_). The first character of an identifier cannot be a digit or a period. There is no distinction made between uppercase and lowercase letters. Some examples of identifiers are:

```
get_user_input
CHAR2
array.length
TestCase
$element_23
f#ms_sector
```

There are a number of otherwise valid identifiers whose meanings are fixed in advance (but are not reserved). These are keywords such as IF, DO, and DECLARE. Because they are actually part of the XPL language, it is recommended that they not be used as programmer defined identifiers.

Variable Declarations

Each variable used in a Scientific XPL program must be declared in a declaration statement before it is used. The declaration defines the variable name and tells the compiler what data type to associate with it. The simplest form of a variable declaration is:

```
declare <identifier> <type>;
```

The identifier is followed by one of the four XPL data type keywords: FIXED, FLOATING, BOOLEAN, or POINTER. Variables of the same type can be grouped together in the same declaration statement by separating the variable names with commas and enclosing them in parentheses. Some sample XPL declarations are:

```
declare i              fixed;
declare (a, b, c, d)   fixed;
declare (x, y, z)      floating;
declare (found, done)  boolean;
declare ptr            pointer;
```

The reserve word DECLARE can optionally be shortened to DCL.

A separate declaration statement is not required for each and every declaration. Instead of writing the two declaration statements:

```
dcl chr    fixed;
dcl (x, y) floating;
```

both declarations can be written in a single declaration statement like this:

```
dcl chr fixed, (x, y) floating;
```

If a declaration statement contains more than one declaration element, each element is separated by a comma. The declaration elements appearing in a single statement are completely independent of each other; it is as if they had been declared in different statements. The only difference is that the keyword DECLARE has not been repeated.

It is good programming style to declare each variable on a separate line with a comment on that same line explaining the function of that variable in that program or procedure.

Literals

The LITERALLY declaration defines a macro for expansion as the program is being compiled. An identifier is declared to represent a character string, which is then substituted for each occurrence of the identifier in subsequent text. The form of the literal declaration is:

```
declare <identifier> literally <string>;
```

The keyword LITERALLY (which can be shortened to LIT) is followed by a string constant that is to be sustituted for <identifier> wherever it appears in the program. The following program section illustrates the use of this literal feature:

```
dcl forever lit 'while (true)';
dcl pi      lit '3.14159';

do forever;
   x = r*pi;
   ...
end;
```

In the above example, the first literal uses the XPL boolean value true to create a meaningful notation for specifying infinite loops. The second statement allows the word pi to be used in the program instead of the number 3.14159 to increase program readability.

A common use of literals is defining numeric constants that are fixed for one compilation, but may change from one compilation to the next. Often it is desirable to make a literal declaration for some quantity such as a data buffer size, a string length, or a program sampling interval. Then by changing one line at the top of the program, all occurrences of that quantity can be automatically changed throughout the program.

Section II - Terminal Input and Output

Scientific XPL has several statements that allow for the transfer of data to and from the computer terminal. The PRINT and SEND statements convert numeric quantities from an internal format to a sequence of printable characters for presentation on a terminal or transmission to a remote computer system. For entering information into a program while it is running, the INPUT statement is used to enter numeric information and LINPUT is used for textual information.

The PRINT and SEND Statements

The PRINT and SEND statements are powerful statements that enable the computer to produce output in a format that is easy to read. PRINT is used to present numerical information or textual material on the computer terminal. SEND is used to send this same information to a remote computer or to a printer. The formats for PRINT and SEND are identical.

The keyword PRINT (or SEND) is followed by a series of subfields separated by commas. Each subfield can be an arithmetic expression, a string constant, or an expression starting with one of the keywords CHARACTER, OCTAL, or STRING. Each subfield in the PRINT statement is printed on the terminal screen in order from left to right. If a comma appears at the end of the last subfield (just before the semicolon), then a carriage return/linefeed pair is not transmitted at the end of the output line. Otherwise, a carriage return/linefeed is performed at the end of the PRINT statement.

A PRINT statement subfield which is a fixed point arithmetic expression is printed on the terminal in a six character (including sign), zero-filled field. A floating point value is printed in a nine character field, including decimal point and sign. If the floating point value is larger than this format allows, asterisks (*) are printed instead.

For example, the following program:

```
dcl x floating;
dcl i fixed;

x = 3.14159;
i = 2;

print 'The value of pi is: ', x;
print 'The value of', i, '*pi is ', i*x;
```

produces the following output:

```
The value of pi is: +3.141589
The value of 00002*pi is +6.283178
```

If a subfield in a PRINT or SEND statement contains an expression that includes the multiplication or division of two fixed point numbers, that operation will be performed in floating point to provide as much resolution as possible:

```
dcl (i, j) fixed;

print i/j;  /* floating point format division */
```

This feature can be overwritten by using the INT function:

```
print int (i/j);  /* fixed point division */
```

The PRINT statement includes a special feature to simplify its use with a video terminal. If CONTROL-S is typed while the computer is performing output to the terminal, the computer will halt temporarily until CONTROL-Q is typed. This freezes the video display so that the user can read it. CONTROL-S can also be used to halt the output to hard copy terminals temporarily to change the paper or make other adjustments. Note that this feature is disabled when interrupts are enabled.

Special PRINT Formats: CHARACTER, OCTAL, STRING

There are three special formats that can be used in conjunction with the PRINT or SEND statements: CHARACTER, OCTAL and STRING.

The CHARACTER format is used to print individual characters on the terminal. The keyword CHARACTER (which can be shortened to CHR) is followed by an arithmetic expression enclosed in parentheses. The result of the expression is written to the terminal directly. The ASCII character format is often used to write control information to terminals or printers or to send 8-bit bytes to a remote computer. Any 8-bit value (0-255) can be sent. The following program segment demonstrates the use of CHR:

```
dcl a.ff  lit '"014"'; /* ASCII formfeed */
dcl a.can lit '"030"'; /* ASCII cancel */
dcl a.esc lit '"033"'; /* ASCII escape */
dcl a.gs  lit '"035"'; /* ASCII group separator  */

print chr (a.gs),;              /* go into graphics mode */
print chr (a.esc), chr (a.ff),; /* clear graphics display */
print chr (a.can),;             /* get out of graphics mode */
```

The OCTAL format is used when an octal (base 8) printout of a fixed point number is desired. The keyword OCTAL is followed by the arithmetic expression to print enclosed in parentheses. A six character zero-filled field is used for octal numbers. The following program demonstrates the use of OCTAL.

```
dcl (i, j) fixed;

i = 73;
j = 1125;

print octal (i), ' ', octal (j);  /* print base 8 equivalents */
print 'Octal sum is ', octal (i + j);  /* print sum in octal */
```

The STRING format is used to print the contents of a fixed point array which contains an XPL string (such as that returned by LINPUT). The keyword STRING is followed by the name of the string enclosed in parentheses. Observe how PRINT STRING is used in the following program.

```
dcl buf (64) fixed;      /* array for holding text string */

linput buf;              /* get the text string from terminal */
buf (0) = buf (0) - 1;   /* get rid of final carriage return */
print string (buf);      /* echo string back to terminal */
```

The INPUT Statement

The INPUT statement allows the user to type numeric quantities into an executing program. The keyword INPUT is followed by one or more variable names, separated by commas. When an INPUT statement is started, the computer will print a question mark on the terminal. The user should then type in the requested numbers, separated by commas (with no spaces). The following program demonstrates this:

```
dcl x floating;
dcl i fixed;

print 'Type in two numbers',;  /* final comma so no CR/LF */
input x, i;
print 'The numbers you typed were:  ', x, ' ', i;
```

When executed, this program presents the user with the following (underlined text is typed by user):

```
Type in two numbers? .2,500
The numbers you typed were:  +.2000000  00500
```

Be sure to PRINT an intelligent prompt line before using the INPUT statement, so that the user will know what is being requested.

Special Use of INPUT

The INPUT statement is designed be used for simple interactive applications. If a single carriage return is pressed in response to a request for numeric output, a value of zero will be returned and a format error will not occur. This is often convenient when user actions are required. The following program segment is an example of this:

```
dcl i fixed;

print 'Please insert the second diskette';
print 'and press RETURN when ready',; /* input prints a ? */
input i;
```

The LINPUT Statement

The LINPUT statement is used to input a string of characters from the terminal into a fixed point array. The keyword LINPUT is followed by the name of the array where the string should be stored. Only one array can be specified. The entire line of input (including any commas, spaces, and the final carriage return) will be stored in the array. Observe the following program:

```
dcl user_input (64) fixed;

print 'Start typing!';
linput user_input;
user_input (0) = user_input (0) - 1; /* get rid of CR at end */
print 'You typed: ', string (user_input);
```

A fixed point array that is used with LINPUT must always be declared to contain 65 elements (0-64). Up to 128 characters (including the final carriage return) can be typed in. Always be sure to print a message on the terminal using a PRINT statement before doing the LINPUT so that the user will know what information is being requested and when to type it in. The LINPUT statement does not print a prompt character (such as a question mark) on the terminal so that flexible input formats can be implemented.

Character strings are discussed in more detail in the section "Arrays and Pointers".

Notes on INPUT and LINPUT

During processing of both numeric input (INPUT) and string input (LINPUT), control characters can be used to strike out typing errors or to delete the input line. CONTROL-X is used to delete and restart the entire line, and DELETE or CONTROL-Z is used to erase the last character typed.

Parity information is masked off automatically by the LINPUT routine. Terminals that use odd, even, 1 or 0 parity can be used. The information stored in the user array will always represent a 7-bit ASCII code (0-127). LINPUT allows entering of both uppercase and lowercase letters. If the programmer wishes to ignore case when processing the line of input, he should explicitly check for commands typed in either uppercase or lowercase.

Make sure to always declare an array being used with LINPUT to have at least 65 elements. LINPUT assumes it can accept up to 128 characters, so if an array is used that is not large enough, memory might be overwritten.

Section III - Operators and Expressions

A well-formed expression consists of operands (variables, constants, or function calls) combined with the various arithmetic, logical, and relational operators, in accordance with simple algebraic notation. Examples are:

```
a + b - c       x < y        d mod 3
a*2 + c/d       c >= 32      a xor b
```

If an arithmetic expression used in a Scientific XPL program contains only fixed point constants, the compiler will evaluate the expression at compile-time and use the resulting number in place of the expression. This reduces the amount of memory required by a compiled program and increases the speed at which the program executes.

Arithmetic Operators

Scientific XPL provides two unary arithmetic operators and seven binary arithmetic operators. The unary operators are:

-	unary minus (two's complement)
+	unary plus

Unary minus is used to negate a value. Unary plus has no effect when used, but is provided for consistency.

The seven binary arithmetic operators are listed below:

+	Addition
-	Subtraction
*	Multiplication
/	Division
mod	Modulus (remainder)
%	Fractional multiply
fdiv	Fractional divide

All seven of these operators are defined for fixed point operands, while only +, -, *, and / can be used with floating point operands.

Addition and subtraction are identical for both signed and unsigned fixed point variables as long as the result of the operation is within the range of interest (-32,768 to +32,767 for signed quantities, or 0 to 65,535 for unsigned quantities).

Fixed point multiplication (*) produces a signed 16-bit product from two signed fixed point operands. Overflow will result from the multiplication of two fixed point numbers if the product is less than (more negative than) -32,768 or greater than +32,767. Fixed point division (/) produces a signed 16-bit quotient and is valid for all signed fixed point operands. Remember that fixed point numbers are always integers and that the result of a fixed point division is technically the greatest integer part of the quotient (the fractional part being represented by the remainder).

The MOD operator computes the remainder of a fixed point division and is valid for all signed fixed point operands. The definition of a remainder indicates that the result of MOD will always be a positive number (or zero).

```
if (i mod 2) = 0 then ...  /* check if I is an even number */
```

The fractional multiply (%) and fractional divide (FDIV) operators perform computations that are useful in some situations. They can be used to extend the range of fixed point variables to provide an effective 32-bit resolution. Fractional multiply first computes a signed 32-bit product by multiplying two signed operands (-32,768 to +32,767). Then this product is divided by the number 65,536 to obtain the 16-bit result (i.e., the result is the upper 16 bits of the 32-bit product). This is effectively equivalent to multiplying the first operand by a fraction that is the second operand divided by 65,536. Fractional multiply can be used in this way to scale a number by a fraction between -.500000 and +.4999847 with a resolution of 1/65536. In the following example fractional multiply is used to save the full 32 bits of the multiplication of two fixed point numbers:

```
dcl ms_word fixed; /* most  significant word of product */
dcl ls_word fixed; /* least significant word of product */
dcl (i, j)  fixed; /* operands */

ms_word = i % j;   /* upper 16 bits of product */
ls_word = i * j;   /* lower 16 bits of product */
```

The fractional divide operator is used to perform 32-bit divisions with fixed point variables. As with fractional multiply, the operands must be signed integers in the range -32,768 to +32,767. Also, to prevent overflow, the first operand must be less than one-half of the second operand. In a fractional divide, the first operand is multiplied by the number 65,536 (this is done by loading a different register) to form a signed 32-bit product. This product is then divided by the second operand to produce the 16-bit result. This is effectively equivalent to computing a fraction by dividing the first operand by the second, and then multiplying the fraction by 65,536. Fractional divide is often used to generate numbers that are used as one operand of the fractional multiply operator.

```
j = i % (4 fdiv 9);    /* multiply I by 4/9 */
```

The *, /, MOD, %, and FDIV operators can be used with fixed point operands in Scientific XPL to achieve high speed computations involving signed integer quantities. Fixed point operations are performed using a single computer instruction which takes on the order of one microsecond to perform. If a Hardware Multiply/Divide unit is connected to the system, these functions will be performed using in-line computer instructions to achieve the highest possible throughput. A multiplication operation will require on the order of two microseconds to perform, while the other operations will require nearer ten microseconds since a sign correction must be performed. Unsigned divisions can be performed at higher rates (approaching two microseconds) by using the READ function and the WRITE statement.

If there is no Hardware Multiply/Divide unit in the system, an internal routine is used to perform each operation. These routines generally require 100 to 300 microseconds to compute the result.

Floating point variables are processed in Scientific XPL by using internal subroutines that perform the various operations. The actual amount of time it will take to perform a floating point operation will depend on the hardware configuration. Generally speaking floating point operations can be performed at a rate of 1000 to 5000 operations per second.

Floating point variables in the range of plus or minus one billion to one billionth and the number zero (0.0) can be added, subtracted, multiplied, or divided without causing arithmetic overflow. Accuracies of seven significant digits are obtained when using floating point variables. All internal products should be kept in the allowable floating point range of plus or minus 5.50 E-20 to 9.00$E18.

Scientific XPL includes a special feature that provides extended precision for multiplications and divisions of fixed point operands. When two fixed point numbers are multiplied together, a 32-bit signed product is computed internally by the compiler. If the multiplication is immediately followed by division by a fixed point number, then the entire 32-bit signed product is divided by the third operand to form the desired quotient. This special feature is only activated if both the multiplication and division occur in the same program statement, as shown below:

```
dcl (i, j, k) fixed;
i = i*j/k;  /* multiply i*j, divide 32-bit product by k */
```

Note that there is a limitation in the current XPL compiler when computing equations that use constants, or rather, any equation that the compiler tries to precompute. If the above equation was written using constants (e.g., 1000*250/469), the compiler would precompute the expression using 16-bit (rather than 32-bit) arithmetic, and the result would be incorrect. To prevent the compiler from precomputing this expression, the second constant must be a variable (e.g., 1000*i/469 where i has been set to 250).

Here are some examples of the arithmetic operators:

```
dcl (i, j) fixed;
dcl (x, y) floating;

i = -j;
j = i + 256;
x = (i - 2) + y;
y = k*count;
x = total/sector_count;
i = count mod 8;
j = i % count;
i = i fdiv j;
```

Relational Operators

The Scientific XPL operators that perform arithmetic comparisons between numeric quantities are shown below.

<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
=	Equal to
~=	Not equal to

Each of these operators requires two operands that can be of any data type. The result of any of these operations is a boolean value true or false. The symbols ^= and <> are equivalent to ~=.

If both operands are fixed point, then a fixed point comparison is performed in one computer instruction. This is done by subtracting the two operands and testing for a negative or positive result. This requires that the two numbers being compared must differ by no more than +32,767 or -32,768 if the correct relationship is to be determined. In general, numbers in the range of -16,384 to +16,383 may be compared using all relationals without overflow. When comparing numbers outside this range, the numbers should differ by no more than 32,767 if correct results are to be obtained. These limitations do not apply, however, to the = and ~= relational operators.

If either operand is floating point value, then a floating point comparison will be performed. In this case, each operand can have a value anywhere in the allowed floating point number range. Remember that floating point comparisons may take much longer than fixed point comparisons, depending on the system configuration. There are no overflow problems associated with floating point comparisons.

Relational operators are often used in the IF or DO WHILE statements:

```
if a = b then ...
do while (a >= 256);
```

The result of a relational operation is the boolean constant true if the relation is true and the boolean constant false if the relation is false. The result of a relational operation can be used in an arithmetic expression:

```
i = i + (j = k);
```

which increments i by one if and only if j = k.

Unsigned Relational Operators

There are six relational operators that are designed to operate on unsigned fixed point quantities. These relational operators are useful for comparing pointers or subscripts for large data arrays. The symbols for the these operators are listed below:

ilt	Is less than
ile	Is less than or equal to
igt	Is greater than
ige	Is greater than or equal to
ieq	Is equal to
ine	Is not equal to

Each of these operators requires two fixed point operands. The result of any of these operations is either true or false. The IEQ and INE functions are equivalent to the standard = and ~= relational operators and are included only for compatibility and standardization. The other relationals operate on unsigned 16-bit integers in the range of 0 to 65,535. These relational operators may be used with any fixed point variable without overflow.

```
if i igt j then ...
do while (ptr ine null);
```