Swapping Procedures

For very large programs, it is possible to store certain procedures in external memory, and to swap them into internal memory when they are needed by the program. Procedures that can swap are identified by the keyword SWAP in the procedure definition:

```
<proc name>: proc (<parm list>) returns (<type>) swap;
```

SWAP must always appear last in the list of procedure options (RETURNS, RECURSIVE, then SWAP).

The XPL runtime system keeps at most one swapping procedure in internal memory at a time. If a procedure that swaps calls another procedure that swaps, the original procedure will be read back into internal memory after the called procedure returns. Thus, swapping procedures can potentially execute many times slower than procedures that do not swap. It is important to make sure that frequently called procedures are not swapping procedures.

The compiler includes special logic to detect string constants in swapping procedures that are not eventually passed to another swapping procedure. It also detects data arrays that are not passed as actual parameters to other procedures. Such string constants and data arrays are stored in external memory with the swapping procedure they are contained within, and swap into internal memory with that procedure when it is called. This feature saves a great deal of internal memory in large programs.

To minimize the amount of internal memory used by a program, all procedures that are not of a time critical nature should be defined to be both recursive and swapping. This causes a program to use only as much internal memory as it really needs at any given moment.

Programs that have been compiled with swapping procedures cannot run on systems that do not have external memory.

Forward Reference Procedure Declarations

It is possible to invoke a procedure that is defined later on in the program. This feature is used so that procedures can be grouped logically within a program rather than in the order in which they are used. The declaration statement for a procedure must occur before any references to the procedure, but the actual procedure definition can exist later in the program. The syntax allows any type of procedure to be forward referenced:

```
dcl <proc name> proc (<parm list>) returns (<type>) <class>;
```

where <proc name> is the name of the procedure, <parm list> is a list of the parameter types separated by commas, <type> is the type of the value the procedure returns, and class is RECURSIVE if the procedure is recursive. If no parameters are passed, the <parm list> should be omitted. If no value is returned, the RETURNS attribute should be omitted.

If a procedure is recursive, the forward reference declaration must include the keyword RECURSIVE. However, if a procedure is a swapping procedure, the keyword SWAP cannot appear in the forward reference declaration.

Some examples of valid forward reference procedure declarations are:

```
dcl x proc;
dcl p procedure;       /* no SWAP attribute here */
dcl r proc recursive; /* RECURSIVE attribute must be here */
dcl y proc (fixed);
dcl z proc (fixed, floating, fixed array) returns (floating);

p: proc swap;
   print 'Program name and date';
end p;

r: proc (i) recursive;
   dcl i fixed;
   ...
end r;

z: proc (temp, div, values) returns (floating);
   dcl temp   fixed;
   dcl div    floating;
   dcl values fixed array;
   dcl result floating;
   ...
end z;
```

Section VII - Modules and Libraries

Scientific XPL provides several ways in which to divide programs into separate sections, as with the use of procedures and blocks. Another way to modularize a program is to place the program code in several different source files instead of one large file and to draw the code together at compile-time using the INSERT and ENTER statements. Different parts of a program can even be compiled separately, and then linked into the main program at a later time. Such program sections are called modules, and after they have been compiled they are referred to as libraries.

There are many reasons why dividing a program into separate files or modules is a useful programming tool. First of all, as programs become larger and larger, it becomes desirable to divide them into several independent sections, making the editing process easier. Secondly, a common library of procedures can be developed that can then be included in several main programs. For example, one can write a procedure to perform a special numeric calculation, store that procedure in a separate file, and then include it in any main program with proper use of the INSERT or LIBRARY statement. Finally, different sections of the program can be compiled and tested individually, simplifying the debugging phase of program development and reducing the time needed to compile the main program.

Treenames

The topics discussed in this section require that the programmer understand treenames. Treenames are used so that files in any catalog or on any device in the system can be accessed from the current catalog. They take the same format when used with the statements in this section as they do with Monitor commands, such as OLD and ENTER. When a single filename is specified, that file is searched for in the current catalog, and if it is not found, the system catalog is searched. If the file is located elsewhere, a pathname must be specified before the filename. A pathname is comprised of an optional device specifier and then one or more catalog names, separated by colons. A pathname with a filename appended is the complete treename for that file.

A leading colon is used to indicate the top level of the current device. If the colon is used with no subcatalog name, as in ENTER ':';, the top level catalog of the current device will be accessed. If the colon precedes a subcatalog name, as in ENTER ':SUBCAT1';, the specified subcatalog located in the top level catalog will be accessed.

The following are all valid treenames:

```
'a'         /* file A in current catalog */
':a'        /* file A from top-level catalog */
':test:a'   /* file A from catalog TEST in top-level */
'x:p:a'     /* file A from catalog P in cat X in current cat */
'w1:st:a'   /* file A from catalog ST on W1 */
'f0:a'      /* file A from F0 floppy disk */
```

The INSERT Statement

The XPL INSERT statement is used to include lines of program statements that are stored in a separate file. The INSERT statement allows full treenames to be specified:

```
insert '<treename>';
```

When the Scientific XPL Compiler processes an INSERT statement, the system is searched for the file specified. If the file is not found, a compiler error will occur. The inserted file may or may not be line-numbered. The body of textual material in the specified file is compiled into the final object code, exactly as if it were typed into the current source file at the location of the INSERT statement.

INSERT statements can be nested. The treename or filename is enclosed in apostrophes and appears after the keyword INSERT.

The following examples are all valid INSERT statements:

```
insert 'x';          /* insert file X from current catalog */
insert ':x';         /* insert file X from top-level catalog */
insert 'abc:hh:x';   /* insert file X from catalog ABC:HH */
insert ':abc:x';     /* insert file X from catalog :ABC */
insert 'f0:x';       /* insert file X from floppy disk in F0: */
```

Optimization of Procedure Source Libraries

The Scientific XPL Compiler includes an interesting optimization feature that simplifies the use of procedure source libraries. Many times, the programmer wishes to assemble a library of common procedures that are used in more than one program. When the compiler processes procedure definitions, it will specifically exclude from compilation any procedures that are not actually called by the program being compiled. By taking advantage of this feature, a programmer can construct a general purpose procedure source library that contains many common subroutines, knowing that only those procedures required by a particular program will actually be compiled.

The ENTER Statement

The ENTER statement is used in conjunction with the INSERT statement to include source files from different devices and subcatalogs into the program currently being compiled.

The ENTER statement changes the catalog that the compiler is currently accessing for inserted files. Without the use of an ENTER statement, all files are searched for starting from the catalog where the COMPILE command was issued (the current catalog). By using ENTER to change this default catalog, any source files that are then inserted will be searched for from the catalog specified in the ENTER statement. Note that the XPL statement ENTER works somewhat differently from the Monitor command ENTER, in that it does not actually change the current catalog.

The syntax for ENTER is similar to that of the INSERT statement:

```
enter '<pathname>';
```

The following examples are all valid ENTER statements:

```
enter 'ab';       /* access files from subcatalog AB */
enter ':ab:hi:c'; /* access files from subcatalog :AB:HI:C */
enter ':';        /* access files from the top-level catalog */
enter 'f0:';      /* access files from the floppy disk in F0: */
```

The asterisk can be used to indicate a return to the current catalog, as follows:

```
enter '*';
```

where the current catalog is the catalog from which the command COMPILE or RUN was issued.

Here is an example of ENTER used with INSERT:

```
insert 'source';     /* inserts SOURCE from current catalog */
enter  'subcat';     /* changes access to subcatalog SUBCAT */
insert 'source2';    /* inserts SOURCE2 from SUBCAT */
```

Using ENTER with INSERT

The ENTER statement causes the specified catalog to be searched for and opened before succeeding files are inserted. This means the full treename does not need to be searched for every INSERT from that catalog. So, in the example:

```
enter ':abc:def:ghi';
insert 'x';
insert 'y';
insert 'z';
```

the catalog :ABC:DEF:GHI is found during the ENTER statement and then remembered for all the following INSERT statements. Contrast this with:

```
insert ':abc:def:ghi:x';
insert ':abc:def:ghi:y';
insert ':abc:def:ghi:z';
```

In the above example, it would seem that the pathname :ABC:DEF:GHI is searched for each INSERT statement. However, the INSERT processing is optimized so that after a pathname has been searched, each level of that pathname (up to some maximum) is remembered until a different pathname is specified. The pathname is therefore only searched once and the results are identical (with one exception) to using ENTER.

For example, the sequence:

```
insert ':abc:def:ghi:jkl:x';   /* search for :ABC:DEF:GHI:JKL */
insert ':abc:d';               /* we know where :ABC is */
insert ':abc:def:ghi:j';       /* we know where :ABC:DEF:GHI is */
insert ':abc:def:ghi:jkl:y';   /* this is the original pathname */
```

would also only cause one pathname search. However, the sequence:

```
insert ':abc:def:ghi:jkl:x';   /* search for :ABC:DEF:GHI:JKL */
insert 'xyz:d';                /* search for XYZ in current cat */
insert ':abc:d';               /* search for :ABC */
insert ':abc:def:ghi:p';       /* search for DEF:GHI in :ABC */
insert ':abc:def:ghi:jkl:y';   /* seek JKL in :ABC:DEF:GHI */
```

causes a variety of different searches.

The main difference between INSERT with a treename and ENTER has to do with what happens when an inserted file has its own INSERTs. What if file X (in the first example) has the statement INSERT 'a'; buried in it? Where do we search for file A in each case?

In the case of ENTER (the first example), the compiler will search for A in the subcatalog :ABC:DEF:GHI. In the second example (INSERT with a treename), the compiler will search for A in the last catalog entered. There is a tremendous difference.

Modules

It is often advantageous to split a large program into much smaller pieces, each of which can be compiled separately and only brought together (linked) when the final program is compiled. These smaller pieces are called modules.

The obvious advantage to this approach is that only the parts of a program that change need to be recompiled to create a new version of a program. For a large program under active development, the time savings can be immense.

The more important advantages lie in the fundamental concepts of structured programming. A well written program (i.e., one that is reliable, flexible, efficient, and maintainable) depends on the ability to partition a large problem into smaller, more manageable parts. The parts generally take the form of black boxes. In other words, their inputs are known, their outputs are known, their function (i.e., what they do to their inputs to create their outputs) is known, but how they do their function is not known or at least it does not need to be known. A properly designed black box can be implemented in many different ways without affecting the user of the box; the box's interface and function do not need to change when it changes. An integrated circuit is a good example of a black box.

XPL procedures provide a very simple black box mechanism. To use a procedure, it is only necessary to know its parameters (inputs), return value (output), and what it does (function). It is totally irrelevant how it does it.

Unfortunately, many black boxes provide multiple functions for the same well defined object. For example, a tape deck allows you to "stop", "play", "rewind", "fast forward", or "record" a tape. A black box that is suitable for managing stacks would allow you to "push" an item onto the stack, "pop" an item off the stack, check if the stack is "empty" or "full", and "clear" all items from the stack (force it empty). These functions all operate on a stack. XPL procedures in and of themselves do not provide this kind of mechanism. There must be some way to group related procedures together with their data structures.

Thus, XPL provides a more generalized black box facility: the module. A module is a region of code which is to be treated as an independent set of procedures which will later be linked with other procedures and some main body of code (referred to by the compiler as MAIN). It is defined using the MODULE statement.

A file which contains a module can contain nothing else, including other modules. Furthermore, the first XPL statement (note that comments are not statements) in that file must be a MODULE statement of the form:

```
module <module name>;
```

and the last statement in the file must be an END statement of the form:

```
end <module name>;
```

where the module name in the END statement is optional, but must match the module name in the MODULE statement if it appears. Any statements found outside this module are disallowed. Nesting of modules is also disallowed. Between the MODULE statement and the END statement, any valid XPL statements can occur.

A word of caution about WHEN statements in modules: only one WHEN statement for any given condition can occur in the final linked program. For example, if WHEN D03INT occurs in the main body of a program, it must not appear in the source module of any referenced library. Furthermore, if a module is compiled for a Model C processor, a program that will run on a Model B processor cannot use that compilation of that module.

Public Variables and Procedures

Variables and procedures declared within a module are, by default, local to that module. That is, they cannot be accessed from any of the other modules in the program nor from MAIN.

Any variable or procedure, however, can have its scope extended to the entire program (i.e., made global) with the PUBLIC storage class. Like the storage classes STATIC and AUTOMATIC, PUBLIC must appear directly after the variable type in the variable declaration. The following are examples of public variable declarations:

```
dcl i     fixed public;
dcl x     floating public;
dcl a (9) fixed public;
dcl l     label public;
dcl d     data public (0, 1, 2, 3, 4, 5);

p: proc (x, y) returns (fixed) public swap;
   dcl (x, y) fixed;

   return (x + y);
end p;
```

In the case of recursive or swapping procedures, PUBLIC must appear first, followed by RECURSIVE and then SWAP.

A public variable should not be declared inside a procedure unless it is certain that the procedure in question will be called. Although the compiler does not enforce this restriction, it does print a warning message to this effect.

Public variables are a special case of static variables. Like static variables, they are initialized (to zero) once at the start of program execution and exist for the life of the program. Their scope, also like static variables, is from the point of declaration to the end of the block in which they are defined. They differ from normal static variables in that they can be referenced (and hence accessed) from anywhere in the program through the use of the EXTERNAL storage class (see below). Thus, their scope is really the entire program. A given variable identifier can be declared to be PUBLIC only once in a program.

The module designer and programmer are encouraged to restrict the number of procedures and especially variables that are accessible from outside a module (i.e., public). A well designed module performs a defined task (e.g., stack manipulation) on localized data (e.g., a stack). Any access to this data is conducted through global functions (e.g., push, pop, empty, full, clear) rather than through global variables. Any internal partitioning of those functions as well as the internal representation of the data is invisible outside the module. A well designed module is, for all intents and purposes, a black box. The advantages to this approach are information hiding (the user doesn't need to know the internal details), data integrity (only the module's functions operate on the data), implementation independence (the implementation can be upgraded to be more efficient or provide additional capabilities without affecting code that accesses the module), and easy maintenance of both the module and any programs that use it.

External Variables and Procedures

In order to reference a public variable or procedure from a part of the program that is not in the scope of the original public declaration (e.g., from another module), it is necessary to redeclare the variable with the EXTERNAL storage class. This storage class is unique in that there is no storage associated with a variable declared in this way. Rather, it references a variable that has already been declared to be public elsewhere in the program.

Being a storage class, the keyword EXTERNAL must appear directly after the variable type in the variable declaration. In the case of procedures, an external declaration takes the form of a forward reference procedure declaration with EXTERNAL appearing at the end (RECURSIVE and SWAP are unnecessary and cannot appear). The following examples of EXTERNAL declarations correspond to the PUBLIC declarations above:

```
dcl i fixed external;
dcl x floating external;
dcl a fixed array external;
dcl l label external;
dcl d data external;
dcl p proc (fixed, fixed) returns (fixed) external;
```

Each variable which is declared with the EXTERNAL storage class must have a corresponding declaration with the PUBLIC storage class somewhere in the program. Any variable that is declared to be external, but does not have a corresponding PUBLIC declaration is called an "unresolved reference" and is not allowed.

A variable can be declared with the EXTERNAL storage class many times throughout a program. It is good style, however, to declare an external variable only in modules where it is actually used. In this way, access to public variables can be restricted.

Libraries

When a module is compiled, a relocatable binary (or library) is created. The LIBRARY statement references a relocatable binary that needs to be "inserted" into the program, much as the INSERT statement references a source file that needs to be inserted into the program. The LIBRARY statement can appear anywhere an INSERT statement can appear and shares the same syntax:

```
library '<treename>';
```

The file specified must be a relocatable binary (a compiled module of file type RELOC).

Libraries are always linked in before the main body of code (MAIN) and so any executable statements that lie outside procedure bodies in libraries (e.g., local or global variable initialization) will be executed before the main body of code. The libraries will appear in the final program in the order in which the compiler finds the LIBRARY statements. Therefore, if only the main body of code has LIBRARY statements, the corresponding libraries will be linked in order. But if the first library inserted in the main body of code has LIBRARY statements within it, all those libraries will be linked in before the next main code LIBRARY statement is processed.

This information is important to know to guarantee that any executable statements located outside procedure bodies in the original modules are executed in the proper order in the final program. A list of the libraries linked in and the order in which they appear in the final program can be obtained by setting the appropriate compile-time option (see the appendix "Compilation Control").

For example, if the main body of code has the following statements:

```
library 'output';
library 'set';

dcl output proc external;
dcl set proc (fixed) external;

call set (3);
call output;
```

The source module for library OUTPUT contains:

```
module output;              /* output the contents of X */
   output: proc public;
      dcl x fixed external;
      print x;
   end output;

   call output;
end output;
```

The source module for library SET contains:

```
module set;                /* set X to some value */
   library 'init';
   dcl init proc external;

   call init;

   set: proc (new_x) public;
      dcl new_x fixed;
      dcl x     fixed external;

      x = new_x;
   end set;
end set;
```

And the source module for library INIT contains:

```
module init;               /* initialize X */
   dcl x fixed public;
   x = 5;

   init: proc public;
      x = -1;
   end init;
end init;
```

The order of the LIBRARY statements in the main body of code determines the order of the libraries in the final program and hence the order of the executable statements that lie outside procedure bodies in those libraries. In the example above, the final position of each library in the final program would be as follows: OUTPUT, SET, INIT, and then MAIN. Therefore, the final program will execute the following statements in exactly this order:

```
call output;  /* from OUTPUT - prints a 0 */
call init;    /* from SET    - sets X to -1 */
x = 5;        /* from INIT   - sets X to 5 */
call set (3); /* from MAIN   - sets X to 3 */
call output;  /* from MAIN   - prints a 3 */
```

The intended order was presumably:

```
x = 5;        /* from INIT   - sets X to 5 */
call init;    /* from SET    - sets X to -1 */
call output;  /* from OUTPUT - prints a -1 */
call set (3); /* from MAIN   - sets X to 3 */
call output;  /* from MAIN   - prints a 3 */
```

To achieve this order, INIT must be libraried in the main body of code and must precede the LIBRARY statement for SET which must precede the LIBRARY statement for OUTPUT as follows:

```
library 'init';
library 'set';
library 'output';
```

Libraries that are referenced from modules do not have to actually exist until the linking process commences (which happens when you compile the main program). This feature is useful when designing a system with a top-down approach and you want to compile a module (or make sure a module compiles) that references a low-level library that perhaps has not been written yet.

Using ENTER with LIBRARY

The ENTER statement behaves somewhat differently when used with the LIBRARY statement than with the INSERT statement. Above, it was mentioned that libraries referenced from within modules do not have to exist until the main body of code is compiled. That is only true for the LIBRARY statement. If the ENTER statement is used to access a library, the catalog specified in the ENTER statement must exist at compile-time. For example:

```
module x;                 /* define module X */
   library 'abc:x:z'; /* ABC:X:Z need not exist at compile */
   enter ':def:xx';   /* :DEF:XX must exist at compile-time */
   insert 'ccc';      /* CCC must exist at compile-time */
   library 'ddd';     /* DDD need not exist at compile-time */
end x;                    /* end of module X */
```

Recall that when the ENTER statement is used in conjunction with the INSERT statement, any files which are inserted from the inserted file are searched for in the catalog entered at the time of the initial INSERT. For example, if file A contains:

```
enter 'xyz';
insert 'b';
```

and file B contains:

```
insert 'c';
```

then file C is searched for in catalog XYZ instead of the current catalog. This is not true when the ENTER statement is used in conjunction with the LIBRARY statement. When a library is inserted, all searches initiated from within that library commence from the current catalog.

For example if file A is the main body of code and contains:

```
enter 'xyz';
library 'b';
library 'd';
```

and file B's source module (remember B is a precompiled module) contains:

```
module p;
   library 'c';
end p;
```

then file B and file D are searched for in subcatalog XYZ, but file C is searched for in the current catalog. What this means is that when ENTER is used with the LIBRARY statement, ENTER is a direct substitution for specifying the full pathname in each LIBRARY statement (except the ENTERed catalog must exist at compile-time). So the following two examples are exactly equivalent:

```
library ':abc:def:ghi:jkl:x';
library ':abc:def:ghi:jkl:y';
library ':abc:def:ghi:jkl:z';
```

and

```
enter ':abc:def:ghi:jkl';
library 'x';
library 'y';
library 'z';
```

except, in the second case, the catalog :ABC:DEF:GHI:JKL must exist at compile-time if these statements are in a module. Note there may have been side-effects to the ENTER statement in the second example if the LIBRARY statements were INSERT statements.

An Example Module

When writing modules, it is important to make them as self-contained as possible. There should not be a large number of external variables. Any person writing a program or procedure which uses the module should not have access to anything except procedure interfaces to the module. For more information about designing modules, consult The Practical Guide To Structured Systems Design by Meiler Page-Jones (published by Yourdon Press in 1980).

The module below is an example of a good module. There are no variables the user can directly manipulate. All system interfaces are restricted to the module. The user just needs to library this module, declare the procedures, and call them.

```
/* This module implements the hardware/software timer interface
  for the D3 real time clock. */

module timer;          /* define module TIMER */
   dcl time fixed;     /* this LOCAL variable counts D3 ticks */
   time = 0;           /* initialize the time counter */

   when d03int then time = time + 1;  /* count D3 ticks */
   enable;                            /* enable interrupts */

   get_time: proc returns (fixed) public; /* current time */
      return (time);
   end get_time;

   set_time: proc (new_time) public;  /* set starting time */
      dcl new_time fixed;             /* new starting time */

      time = new_time;
      write ("3") = 0; write ("3") = 0; /* reset the clock */
   end set_time;

   wait_time: proc (seconds) public;  /* wait time SECONDS */
      dcl seconds    fixed;           /* seconds to wait */
      dcl start_time fixed;           /* starting time */
      dcl ticks      fixed;           /* ticks to wait */

      start_time = get_time;          /* get starting time */
      ticks = seconds*200;            /* ticks to wait */

      /* wait for TICKS */
      do while (get_time - start_time ilt ticks); end;
   end wait_time;
end timer;                          /* end of module TIMER */
```

It would be far easier for the user of a library if the person who writes a module provides an insert file which describes what the library does and how to use it, references the library, and declares all the external variables and procedures. Such an insert file is shown below (the file is called TIMER).

```
/* Timer subroutines:
.
.  This library manages the D03 hardware interface, allowing the
.  user to deal with timer values (in units of 5 ms) rather than
.  the hardware/software interface for the timer.
.
.  Three routines are provided:
.    GET_TIME:      returns the current time relative to the
.                   last time set.
.    SET_TIME (X):  sets the timer counting from X.
.    WAIT_TIME (X): wait X seconds.
.
.  WARNING:  This library enables interrupts.  Do not use this
.            library for time critical applications. */

library ':libs:timer'; /* reference library TIMER */

/* reference these three procedures */

dcl get_time  proc returns (fixed) external;
dcl set_time  proc (fixed) external;
dcl wait_time proc (fixed) external;
```

It is then trivial for the user to write the following program without any special knowledge of the way the routines are implemented:

```
/* $Dump statistics, $Map memory */

insert ':timer';          /* reference timer routines */

dcl i fixed;

print get_time;           /* print starting time */
call wait_time (5);       /* wait five seconds */
print get_time;           /* verify the ending time */
call set_time (0);        /* initialize the timer */

do i = 0 to 32767; end;   /* increment a variable 32K times */

print get_time;           /* print the elapsed time */
call set_time (0);        /* initialize the timer */

/* count to three, outputting one number every second */

print 'zero ',;
do while (get_time < 200); /* wait until 1st second had passed */
end;
print 'one ',;
do while (get_time < 400); /* wait until 2nd second has passed */
end;
print 'two ',;
do while (get_time < 600); /* wait until 3rd second has passed */
end;
print 'three';
print get_time;            /* print the final time */
```

Section VIII - Hardware Manipulation

This section contains information about using the hardware of the ABLE Series computer. This topic is expanded in the "ABLE Series Hardware Reference Manual".

The Scientific XPL language offers two mechanisms for performing real time input and output. The READ function and WRITE statement operate on 16-bit fixed point quantities and are used to monitor and control interface devices connected to the system.

The READ Function

The READ function is used to transfer a 16-bit data word from an interface device into the user's program. Most such data transfers require only one microsecond to perform. The reserved word READ is followed by a device number enclosed in parentheses.

```
<variable> = read (<device number>);
```

The device number is often expressed in octal (e.g., "34" or "16") although literal declarations can be used to increase program readability. Observe the following two examples:

```
dcl i fixed;
i = read ("34"); /* read data from digital interface device */
print i;

dcl timer literally '"16"'; /* D16 clock */
dcl i     fixed;
i = read (timer) + 2;
print i;
```

The READ function can be used in any arithmetic expression, as if it were a function requiring one fixed point parameter and returning a fixed point result. The device number can also be a variable expression, although in this case the READ will take many microseconds.

The WRITE Statement

The WRITE statement is used to transfer a 16-bit data word from inside the user's program to an interface device connected to the system. The format for the WRITE statement is:

```
write (<device number>) = <expression>;
```

The keyword WRITE is following by a device number enclosed in parentheses (similar to READ). An equals sign (=) is then followed by any allowed XPL arithmetic expression.

The WRITE statement operates similarly to an assignment statement. The arithmetic expression to the right of the equals sign is first evaluated. The computed number is converted to fixed point automatically if a floating point expression is used. The number is then written out to the specified device.

```
write ("60") = i + 1;
write ("71") = i;
```

The following example uses a literal to improve program readability:

```
dcl DAC literally '"66"'; /* Digital-to-Analog Convertor */
dcl i   fixed;

do i = 0 to 1023;
   write (DAC) = i;
end;
write (DAC) = 0;
```

Precautions for READ and WRITE

If a program attempts to READ and WRITE data to a device that is not connected to the system (or does not exist), the computer will halt and program execution will not continue. This feature is often used for diagnostic purposes. However, if a Hand Operated Processor is not connected to the system, it will be impossible to tell which device number is being addressed. Without such information, it is very difficult to diagnose errors in the program. The user should be careful to always specify the correct device number in both the READ function and the WRITE statement. Information on device numbers is available in the manual "ABLE Series Hardware Reference Manual".