Assembly Language Programming

It is possible to embed assembly language code within an XPL program using the READ function and the WRITE statement. The source of the instruction is specified in the READ while the destination is specified in the WRITE. The following form would be used for a source to destination instruction:

```
write (<destination>) = read (<source>);
```

For example, to add two fixed point values together:

```
dcl r0   lit '"300"';
dcl r1   lit '"301"';
dcl add0 lit '"210"';

dcl (i, j) fixed;
...
write (r0) = i;              /* I to R0    */
write (r1) = j;              /* J to R1    */
write (add0) = read (r1);    /* R1 to ADD0 */
i = read (r0);               /* R0 to I    */
```

The XPL compiler uses the computer's registers in a well-defined way that is described in detail in the "ABLE Series Operating System Reference Manual." In general, registers zero through three are always available for assembly language programming. Furthermore, register thirteen (octal) is never used by the XPL compiler and is available at all times for use by the programmer.

For more information about assembly language programming and a full list of sources and destinations, refer to the "ABLE Series Hardware Reference Manual".

Section IX - Interrupt and Exception Processing

Scientific XPL includes the capability to process many different hardware interrupts with the WHEN statement. Many WHEN statements may appear in one program, and each WHEN statement is used to process a different interrupt or exception. This section contains information about the ABLE Series computer that is explained in detail in the "ABLE Series Hardware Reference Manual".

The WHEN Statement

Interrupts and exceptions are processed in XPL by using the WHEN statement. The format of a WHEN statement is:

```
when <interrupt> then <statement>;
```

The keyword WHEN is followed by an interrupt (or exception) identifier to indicate which interrupt is to be processed by that WHEN statement. The keyword THEN appears after the cell identifier and is followed by any legal XPL statement (usually a BEGIN-END block). WHEN automatically returns from the interrupt at the end of the statement, but can be forced to return with a RETURN statement.

ENABLE and DISABLE

Interrupts are disabled when a program begins execution. In order to use interrupts in a program, interrupts must be explicitly enabled with the ENABLE statement. Interrupts can only be received by the system after the execution of an ENABLE statement and interrupts will be ignored after the execution of a DISABLE statement. Interrupts can be selectively enabled or disabled many times during the course of program execution.

The ENABLE and DISABLE statements take the form:

```
enable;   /* interrupts enabled starting here */
disable;  /* interrupts disabled starting here */
```

The two special exception processing WHEN statements WHEN DISKERROR and WHEN BREAK are active at all times and do not require interrupts to be enabled.

Interrupt Processing

The possible XPL interrupt identifiers are shown below.

TTOINT	-	terminal output interrupt
TTIINT	-	terminal input interrupt
BDB15INT	-	custom user device interrupting on BDB15
BDB14INT	-	custom user device interrupting on BDB14
DxINT	-	interrupt from device X, where X can be any of the following devices (octal): 03, 16, 24, 30, 31, 32, 33, 34, 35, 36, 37 40, 42, 44, 46, 66, 107, 136, 137, 140

Using WHEN statements requires a great deal of care. In complex situations where several devices are simultaneously interrupting the computer, there are endless possibilities for user traps and subtle programming errors. There are some simple rules that should be followed when using WHEN statements that will help minimize such errors.

Interrupts should not be enabled (using ENABLE) inside a WHEN statement. During the processing of a WHEN statement, interrupts are disabled automatically by the system, to avoid the overwhelmingly confusing situation where a device might interrupt its own interrupt processing WHEN statement. Interrupts are reenabled automatically after processing a WHEN statement, as control is returned to the point of interrupt.

Never use a GOTO statement to exit from a WHEN statement. Instead, a RETURN statement should always be used to return control to the point of original interrupt. If another operation must be initiated from within a WHEN statement, the proper means is by flags, queues, and list processing structures.

Swapping procedures should not be called from a WHEN statement.

The following are explanations of some of the available WHEN statements.

WHEN TTOINT

When the terminal interface has finished printing a character on the terminal, a completion interrupt is generated to the computer. If a WHEN TTOINT statement is in the user program and interrupts are enabled, this WHEN statement is activated upon the completion of character transmission. The normal response is to the print another character on the terminal using PRINT CHARACTER. If there is no more information to be printed on the terminal then the WHEN statement should simply RETURN. This interrupt is automatically cleared by the XPL runtime system.

WHEN TTIINT

When a character is typed on the computer terminal, a hardware interrupt is generated. This interrupt is passed to the user's WHEN statement if interrupts are enabled when the character is typed.

As the XPL runtime system processes the interrupt, the character is read from the terminal (clearing the interrupt) and stored in a special memory location. The built-in function RCVDCHARACTER extracts the character from the memory location and returns it to the program (see example below). A typical WHEN TTIINT statement will read the character from RCVDCHARACTER, examine the character, and branch accordingly.

WHEN D03INT

The D3 Real Time Clock can be used in the interrupt mode. When interrupts are enabled, the D3 generates interrupts at a continuous 200 Hertz rate. This interrupt is automatically cleared by the XPL runtime system.

WHEN D136INT

The D136 Real Time Clock can be used in the interrupt mode. When D136 interrupts are enabled, the D136 generates interrupts at a continuous 200 Hertz rate. This interrupt is automatically cleared by the XPL runtime system.

WHEN D16INT

The D16 Scientific Timer can be used effectively in interrupt driven systems. When the status flag (device "17") of the timer becomes a one, an interrupt is generated if interrupts have been enabled by the ENABLE statement. A typical WHEN D16INT statement would reset the Scientific Timer to interrupt the computer again at an appropriate time interval, and then proceed to increment a timing variable or perform some experimental measurement. This interrupt is automatically cleared by the XPL runtime system.

WHEN D140INT

The WHEN D140INT is used to process interrupts of all types that are generated by the D140 Communications Processor. The D140 is an asynchronous communications subsystem that will support up to 64 terminals. Upon interrupt, the XPL runtime system selects the card and port that generated the interrupt. Separate documentation is available on the D140 entitled "Using the D140 Communications Processor".

WHEN BDB14INT, WHEN BDB15INT

Certain custom devices can be constructed that are to be used in an interrupt mode. These devices will be constructed to ground the Most Significant (BDB15) or Second Most Significant (BDB14) Data Bus Line during the ACKnowledge phase of the interrupt. In this situation, control is automatically transferred to the WHEN BDB14INT or BDB15INT statement, as appropriate.

The following example demonstrates the use of a WHEN statement:

```
when ttiint then begin;
   dcl c fixed; /* character typed by user */

   c = rcvdcharacter;
   print 'You have just typed the character: ', chr (c);
   return;
end;

enable;            /* enable interrupts here before loop */
do while (true);   /* infinite loop */
   i = 1;          /* do something here - cannot interrupt an
end;                          empty infinite loop */
```

The WHEN statement in the above example processes a terminal input interrupt. Note that the WHEN statement starts with the word WHEN and includes all statements in the BEGIN-END block.

In the example, a hardware interrupt is generated by the serial interface when a character is typed on the computer terminal and interrupts are enabled. If the appropriate WHEN statement is located anywhere in the user program (WHEN TTIINT), the XPL runtime system accepts the hardware interrupt, saves all necessary registers, and transfers control to the statements inside the WHEN statement.

The WHEN statement will read the character that was typed in (using the built-in function RCVDCHARACTER) and then print a message which includes that character on the terminal. After the interrupt has been processed by the WHEN statement, control is returned to the point of original interruption.

Exception Processing

There are two WHEN identifiers that are used to handle special exceptions. These identifiers differ from the others in that interrupts do not have to be enabled for them to work. They are active at all times.

WHEN BREAK

The WHEN BREAK statement is activated if the BREAK sequence is received from the terminal while the PRINT or SEND statement is being used, or while the INPUT or LINPUT statement is being used to read data from the terminal.

WHEN DISKERROR

The WHEN DISKERROR statement is activated if the program tries to read or write a sector of the disk that does not exist, or if a data error is encountered when reading or writing the disk. WHEN DISKERROR is often used to print an appropriate error message on the terminal in the case of a disk error.

The INVOKE Statement

It is sometimes necessary to invoke an interrupt routine in a program when the appropriate interrupt has not occurred. This most often happens when the interrupt routine cannot afford to call a procedure, but the interrupt code needs to be shared. This also occurs when an exception is detected by user software (such as a break typed by the user in some terminal processing code or a bad disk sector found within a user device driver). The INVOKE statement is used to accomplish this:

```
invoke <interrupt>;
```

For example:

```
invoke diskerror;  /* a disk error has occurred */
```

SCIENTIFIC XPL REFERENCE MANUAL

Reference Appendices

Appendix A - Built-In Functions

This appendix describes the many built-in functions that Scientific XPL provides. Following a section on devices and storage device input and output, all the built-in functions are listed by category of operation. Following this is an alphabetic listing (sorted by routine name) that describes each routine in detail.

Storage Device I/O

Scientific XPL provides several routines that are used to access storage device media directly. Some of the routines read data from storage devices into internal, external, or polyphonic sampling memory, and others write data from any memory location to storage devices. These routines can be used with any Winchester, floppy, optical disk, or tape that is configured in the system.

The parameters of these routines should be checked carefully, especially with routines that write to devices, because valuable data can easily be overwritten. It is also necessary to make sure that device directories are not destroyed. Refer to the manual "The XPL Catalog Routines" for this directory information before writing data to storage devices.

When using routines to access Winchester systems with more than one drive attached to a device, XPL will automatically determine which physical Winchester disk contains the specified logical sector.

If the storage device you are using with the storage device routines is a SCSI device, you must insert either :-XPL:SCSI or :-XPL:SCSISWAP into your program to make the device routines operate correctly.

Device Numbers

In order to access storage devices, a numbering convention has been established for all possible storage devices in the system. These numbers should be used to specify which device is to be read from or written to.

0: system device (W0 on Winchesters, F0 on floppy systems)
1: current device (set under MONITOR by the user with ENTER)
2: F0 (Leftmost floppy)
3: F1 (Rightmost floppy)
4: R0 (Remote floppy)
5: R1 (Remote floppy)
6: W0 (Winchester disk)
7: W1 (Winchester disk)
8: T0 (Tape drive)
9: T1 (Tape drive)
10: O0 (Optical disk)
11: O1 (Optical disk)

Only devices that are configured and present in a system should be used. This can be checked with a call to FIND_DEVICE.

Storing Floating Point Data

Special procedures must be followed when accessing floating point data stored on disk. When fixed point numeric quantities are stored, each word of disk storage represents one fixed point number. But floating point quantities are stored inside the computer (and on disk) in an encoded form requiring two words of memory.

The standard routines such as READDATA and WRITEDATA can be used for accessing floating point data stored on disk. Since each data element requires two words, however, the parameter specifying the number of words to transfer should be twice as large as when processing fixed point quantities. Remember that floating point data will require twice as much disk storage also. Note the following examples:

```
declare buf (255) fixed;
call readdata (0, 40, buf, 256);
```

and

```
declare buf (255) floating;
call readdata (0, 40, buf, 512);
```

In the first example, 256 words of data corresponding to 256 fixed point numbers are read from sector forty of the system device. In the second example, however, 512 words must be read with READDATA in order to recover only 256 floating point numbers.

XPL Built-In Functions

The XPL built-in functions are listed below by category. Following this list, the routines are explained in detail, listed alphabetically by name.

Storage Device I/O

READDATA - reads data from a device into an array (p. 124)
WRITEDATA - writes data from an array to a device (p. 133)
EXTREAD - reads data from a device into external memory (p. 113)
EXTWRITE - writes data from external memory to a device (p. 115)
POLYREAD - reads data from a device into polyphonic memory (p. 121)
POLYWRITE - writes data from polyphonic memory to a device (p. 122)

Block Manipulation

BLOCKMOVE - copies from one array to another (p. 108)
BLOCKSET - initializes an array (p. 108)
IMPORT - copies from external memory into an array (p. 116)
EXPORT - copies from an array into external memory (p. 112)
EXTSET - initializes a block of external memory (p. 114)

Arithmetic Functions

ABS - absolute value (p. 106)
SQR - square root (p. 129)
SIN - sine (p. 129)
COS - cosine (p. 111)
TAN - tangent (p. 132)
ATN - arctangent (p. 107)
LOG - natural logarithm (p. 119)
EXP - natural anti-logarithm (p. 112)
INT - converts floating point to fixed point (p. 117)

String Functions

BYTE - reads a byte from a string (p. 109)
PBYTE - writes a byte into a string (p. 120)

Bit Manipulation

SHL - shift left (p. 127)
SHR - shift right (p. 128)
ROT - rotate left (p. 125)

Pointers

CORE - an array representing internal memory (p. 110)
ADDR - finds the memory address of a variable (p. 106)
LOCATION - references memory during procedure calls (p. 118)

Program Termination

EXIT - terminates program execution (p. 111)
STOP - halts the computer (p. 130)

Systems Programming

FIND_DEVICE - finds a device in the configuration (p. 116)
SET_CURDEV - sets the current device (p. 126)
SWAPINIT - reinitializes swapping mechanism (p. 131)
RCVDCHARACTER - gets last character from terminal interrupt (p. 123)

Hardware

READ - reads a word from an interface device (p. 123)
WRITE - writes a word to an interface device (p. 132)

ABS — computes absolute value — ARITHMETIC

Synopsis:

```
abs: proc (fixed)    returns (fixed);
abs: proc (floating) returns (floating);
```

Usage:

```
result = abs (num);
```

where NUM can be either a fixed or floating point number. ABS returns the absolute value of NUM.

Example:

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

i = abs (j);
y = x*y + abs (x);
```

ADDR — finds memory address of a variable — POINTERS

Synopsis:

```
addr: proc (scalar variable) returns (pointer);
```

Usage:

```
ptr = addr (var);
```

where VAR is any simple variable or subscripted variable. ADDR returns a pointer to the absolute location of memory in which the variable is stored.

Example:

```
dcl ptr pointer;

ptr = addr (i);          /* simple variable */
ptr = addr (buf (i));    /* subscripted variable */
```

See also: LOCATION
CORE

Synopsis:
```
atn: proc (floating) returns (floating);
```

Usage:
```
radians = atn (num);
```

ATN returns the arctangent of NUM in radians.

XPL does not provide built-in functions for computing arcsine and arccosine, but these functions can be derived from ATN as follows:

```
arcsine (x)   = atn (x/sqr (1 - x*x))
arccosine (x) = 2*atn (sqr ((1 - x)/(1 + x)))
```

Example:
```
dcl (x, y) floating;

x = 2*atn (y);
```

See also: TAN
SIN
COS

BLOCKMOVE copies from one array to another BLOCK MANIPULATION

Synopsis: blockmove: proc (fixed array, fixed array, fixed);

Usage: call blockmove (source, dest, length);

This routine copies LENGTH words from the SOURCE buffer to the DESTination buffer. The copy works correctly if the two buffers overlap; the destination buffer will always contain the original contents of the source buffer.

The LOCATION and ADDR functions can be used to create pointers to specific array or memory locations.

Example:

```
call blockmove (input_buf, string, 8);
call blockmove (buf, loc (addr (data (ptr))), 512);
call blockmove (loc (ptr), buf, len);
```

See also: BLOCKSET

BLOCKSET initializes an array BLOCK MANIPULATION

Synopsis: blockset: proc (fixed array, fixed, fixed);

Usage: call blockset (buffer, length, value);

This routine assigns VALUE to the first LENGTH words of BUFFER.

The LOCATION and ADDR functions can be used to create pointers to specific array or memory locations.

Example:

```
call blockset (input_buf, input_len, 0);
call blockset (match, 10, false);
call blockset (loc (ptr), 256, 0);
```

See also: BLOCKMOVE

Synopsis: byte: proc (fixed array, fixed) returns (fixed);

Usage: char = byte (string, byte_num);

where STRING is an XPL string and BYTE_NUM is the byte position in that array (starting from zero). BYTE returns the 8-bit value that is stored in that position.

BYTE is usually used with fixed arrays that are XPL strings, as each character is represented by one byte (two characters per word). When using BYTE, the first byte (byte position 0) corresponds to the lower half of word one of the array, because the string length is stored in the first word (element 0) of the array.

BYTE can be used with fixed arrays that are not strings, to manipulate data elements by byte rather than by word. In this case the LOCATION and ADDR functions must be used to access the first word (element 0) of the array, as shown below:

```
dcl list (100) fixed array;

char = byte (loc (addr (list (0)) - 1), byte_num);
```

Example:

```
dcl str (10) fixed;  /* string of 20 characters */
dcl i        fixed;

linput str; /* get a string from the user */
str (0) = str (0) - 1; /* get rid of carriage return */

do i = 0 to str (0) - 1; /* loop through characters */
   print chr (byte (str, i));  /* print each one */
end;

i = byte (str, 3);  /* get the fourth character */
```

See also: PBYTE

Synopsis:
```
core: proc (pointer) returns (fixed);
core (pointer) = expression;
```

Usage:
```
number = core (ptr);
core (ptr) = number;
```

CORE is an array that is used to read and write locations of internal memory. CORE starts at location zero of memory and has as many elements as there are words in memory. It is used in an arithmetic expression with a subscript PTR that represents the word of internal memory to read from or write to.

Care should be taken when writing data to memory that the locations used by the program are not overwritten.

Example:
```
dcl ptr pointer;

x = core (1);          /* get contents of word one */
core (ptr) = x + y;    /* store a value in location PTR */
```

See also:
ADDR
LOCATION