ITHACA INTERSYSTEMS
INTERPEST
PASCAL ERROR SOLVING TOOL
AN INTERACTIVE SYMBOLIC DEBUGGER
FOR PASCAL/Z
REVISION 1.1

InterPEST
(InterSystems Pascal Error Solving Tool)

An Interactive Symbolic Debugger for Pascal/Z

Reference Manual

Revision 1.1

TABLE OF CONTENTS

INTRODUCTION TO InterPEST

InterPEST (Intersystems Pascal Error Solving Tool) is the interactive symbolic debugger for use with programs generated by the Pascal/Z compiler and software package. InterPEST is designed to aid in isolating and correcting faults in a Pascal program.

InterPEST allows the user to set both absolute and conditional breakpoints, to display these breakpoints as well as variables and statement/module numbers, to display runtime requirements at any point in the program, to display the last ten statements executed, and to display the procedure/function stack. InterPEST also allows the user to modify both global and local variables, including subscripts of arrays, fields of records, and enumeration types.

The interactive nature of InterPEST obviates the need for manually "tracing" through a program, and permits the user great freedom in setting up a series of conditions which can quickly be changed as the program flow demands. The wide range of available commands in InterPEST provides the Pascal/Z user with a powerful and versatile software debugging tool.

GENERAL INFORMATION ON InterPEST

The debugger adds approximately 12K when linked with a Pascal/Z program.

Only types which are declared globally may be accessed by the debugger. Locally declared variables of global types may be accessed.

The T (Trace) and E (Extended error messages) compiler options will have no effect when using the debugger. The C (Control C checking) option should be disabled, otherwise user output files will not be closed when exitting the debugger.

When using InterPEST, any command involving a symbolic reference (a variable or procedure/function name, such as DV, MV, SW, SR, etc.) will require one or more disk accesses, since the symbol and type information is stored on diskette. The SR (set a watch on a routine) command will require one disk access for each procedure/function entry. Thus disk I/O may slow down program execution significantly when using the debugger. (If using the Cache BIOS as provided with the InterSystems Pascal Development System (PDS), this does not apply.)

If the program being executed contains READ statements, the user will have to input information from the console. Since it is not always apparent that the program is demanding input, often it appears that the debugger is "hung". If it appears that this is so, try inputting some data.

If the program being executed contains WRITE statements, the debugger output may often appear somewhat garbled since it will be interspersed with the program output. Using WRITELN statements rather than WRITEs during debugging can alleviate this problem.

Another problem may arise if debugging a program which requires additional parameters in the command tail. When the debugger asks for the SYM/TYP file name, it accepts only a single CP/M file name as input. This problem may be circumvented by inserting a breakpoint in the program before the end of line (EOLN) condition is tested, or before the program deals with the remainder of the command tail. Then enter CB (continue until the breakpoint) followed by the remaining parameters, as follows:

--->~CB <parameter> .. <parameter>~

as the next command to the debugger.

E.G. When debugging an editor requiring a text file as input, you might enter: CB manual.txt

One note of caution: CP/M converts all input to capitals before processing. The debugger does not do this; therefore the program will be receiving lower case input, which it may not accept if it was expecting normal input from the command tail. Thus when using the above procedure, it is advisable to enter the entire

command in capital letters.

HOW TO USE InterPEST

To use the debugger, compile your Pascal program, specifying a total of five drive letters, as follows:

~A>PASCAL48 [or PASCAL54] PRIMES.AABAX~

The compiler generates five files during compilation.

The first letter after the dot specifies the drive from which the compiler should take the Pascal source (.PAS) file as input, in this case drive A.

The second letter specifies the drive to which the Z-80 source (.SRC) file should be output, in this case drive A.

The third letter specifies the drive to which the listing (.LST) file should be output, in this case drive B.

The fourth letter specifies the drive to which the symbol (.SYM) and type (.TYP) files should be output. These files contain symbolic reference information for the debugger.

The fifth letter must always be X, and indicates that the debugger is being invoked. When the fifth letter is specified, the compiler outputs a fifth file of the form ##.DBG, where ## is a module number. If not using separate compilation, the default is 00. If using separate compilation, a unique .DBG file will be output for each module compiled. There is a maximum of sixteen modules permitted (0..15), therefore debugging more than sixteen modules at one time will cause problems when using the debugger.

(The program will be compiled as described on pages 46-49 of the PASCAL/Z IMPLEMENTATION MANUAL.)

The program must then be assembled using XMAIN.SRC (instead of the ordinary MAIN.SRC) or XEMAIN.SRC for separately compiled modules. The debugger may not be used for debugging external assembly language routines.

To assemble, type:

~A>ASMBL XMAIN,PRIMES/REB~

This indicates that the assembler should generate a relocatable object code module (.REL file) to be linked with the debugger. XMAIN.SRC or XEMAIN.SRC must always be the first file in the command line to the assembler.

The next step is to link the program with the debugger and the library. To do so, type:

~A>LINK PRIMES DEBUG/N:PRIMES/E~

This command links the debugger with PRIMES. The /N:PRIMES specifies

that the linker should generate a command (.COM) file with the name PRIMES. The library (LIB.REL) is automatically linked in, and /E specifies that control should then be returned to the operating system.

RUNNING InterPEST

To invoke the debugger, type the name of the .COM file, as follows:

```
~A>PRIMES~
```

The screen will come back with:

```
Pascal/Z debugger -- v-1.0
SYM/TYP file name --
```

At this point, enter the file name with the .SYM and .TYP extensions, in this example, PRIMES. The debugger will return with the statement and module numbers of the beginning of the main program and a prompt, to which the user may respond with any of the commands summarized on page 7 and described in detail on pages 12-35.

```
SYM/TYP file name -- ~PRIMES~
<MAIN> Statement 16, module 0
--->
```

If any of the following files is not on the logged-in drive, the drive letter must be specified:

```
<file name>.COM
<file name>.SYM
<file name>.TYP
##.DBG
```

Once debugging is finished, the program must be reassembled and relinked without the debugger information, as described on pages 46-49 of the PASCAL/Z IMPLEMENTATION MANUAL.

(For further information on compiling, assembling and linking Pascal/Z programs, see pages 46-49 of the PASCAL/Z IMPLEMENTATION MANUAL.)

USING InterPEST

Once the debugger has been linked with the user's program, and execution of the program has begun, all commands are given by typing the necessary code followed by a carriage return line feed (CRLF).

The following is a summary of the available commands, which are described in detail in the succeeding pages:

MB [# #]	Modify the breakpoint table
MC [var]	Modify the break conditions
MV [var]	Modify the variable value
DB	Display the breakpoint table
DC	Display the break conditions
DN	Display the last ten statements executed
DP	Display the procedure/function stack
DR	Display the current runtime requirements
DS	Display current statement and module numbers
DV	Display the variable type and value
CB	Continue execution until the next breakpoint
CN [#]	Continue execution for # more statements, or until the next breakpoint.
Q	Quit the debugger and return to the operating system
SE	Set the procedure/function entry/exit indicator
SP	Set the program counter to the beginning of the main program
SR <routine>	Set a watch on a routine (procedure or function)
ST	Set the trace indicator
SW [var]	Set a watch on the variable

RELATIONAL OPERATORS ("RELOPS")

LT	Less than (<)
GT	Greater than (>)
LE	Less than or equal to (<=)
GE	Greater than or equal to (>=)
EQ	Equal to (=)
NE	Not equal to (<>)

Typing HELP, H, or ? gives the menu of commands.

Hitting the DELETE key at any point while in the debugger will halt execution of the current debugger command and return control to the debugger.

SPECIFICATIONS

SIZE -- The debugger adds approximately 12K when linked to the Pascal/Z program.

BREAKPOINT TABLE -- The breakpoint table will hold a maximum of ten entries at any one time.

CONDITIONAL BREAKPOINTS -- There may be a maximum of ten break conditions in effect at any one time.

MODULES -- a maximum of sixteen modules may be debugged simultaneously.

MODIFYING VARIABLES -- All global variables or locally declared variables of global types may be modified, with the exception of the following: any REAL variables, Pascal files variables, pointers, strings, sets, arrays of arrays, and arrays of other excepted variables.

DISPLAYING VARIABLES -- All global variables or locally declared variables of global types may be displayed, with the exception of the following: Pascal file variables, arrays of arrays, and arrays of other excepted variables.

POINTERS -- When a pointer variable is displayed, the value given will be that of the variable pointed to.

CN COMMAND -- the maximum number of statements which may be specified using the CN command is MAXINT (32767).

PROCEDURE/FUNCTION STACK -- only the outermost twenty-five calls on the procedure/function stack will be displayed. The debugger will give a message specifying the number of calls not displayed.

Any operation which is dependent on a procedure or function name (e.g. anything involving local variables: modifying, displaying, setting a watch on a local variable or a routine, displaying the procedure/function stack) will not function properly if nested more than twenty-five levels.

NOTE: The result of a statement is not available until the execution of the statement is completed.

EXAMPLE

If A gets set equal to TRUE in statement 1, a DV of A will not return TRUE until statement 2 is the current statement (as displayed when commanding DS).

Similarly, when a breakpoint is specified, program execution will halt before the execution of the break statement.

ERRORS

Any errors in the debugger command syntax or in input will be caught by the debugger and the user will be warned and reprompted. Once an error message is displayed on the screen, ignore all debugger output until the next prompt ("--->" or "Value -- " for enumeration types). (Sometimes the debugger may output strange messages after an error message is displayed, because of the way in which it stores information.)

If the message "Enumeration:<badval>" is generated when trying to display or modify an enumeration type, this means that the ordinal value of the enumerator given exceeded the ordinal value of any possible enumerators.

When running the debugger, any error in the user's program will return control to the debugger. The program counter will automatically be reset to the beginning of the main program (as in the SP debugger command). Program execution will not be continued beyond the error until the error is corrected and the program is recompiled, reassembled, and relinked with the debugger.

Any debugger error will return with the message "Fatal debugger error" and control will be returned to the operating system. Examples of fatal debugger errors are "stack overflow", "file not found", "type error on input", and "read beyond end of file".

TROUBLESHOOTING

This section contains a description of some common problems encountered when using InterPEST, as well as methods of solving them.

One problem which occurs fairly often is that a breakpoint has been set and when the debugger finds the breakpoint it displays "Breakpoint encountered -- " and then hangs. This is due to the fact that the debugger has accessed the wrong ##.DBG file.

If two programs are being debugged using the same disk, the .DBG files generated during compilation may be output into an already existing .DBG file with the same module number. This utterly confuses the debugger, so care must be taken when debugging more than one program on a single disk.

When displaying a variable, if the results shown by the debugger are inaccurate, this may be due to uninitialized variables. Making certain that all variables are initialized at the very beginning of the program will eliminate any spurious results.

Other problems which may arise when using InterPEST (e.g. if the debugger "hangs") may be solved by recompiling, reassembling and relinking the Pascal/Z program with the debugger. If this does not correct the problem, contact InterSystems for assistance.

DESCRIPTION OF COMMAND SYNTAX

The following sections of the manual contain a detailed description of InterPEST command syntax.

Note that anything specified in square brackets ([]) is optional. If this information is not provided in the initial command line, the user will be prompted by the debugger. For the purposes of explanation, in the following description it is assumed that the user provides a minimum of information in the command line. Anything specified in angle brackets (<>) must be included in the command line.

In the following pages, user commands and responses are displayed in ~bold~ face. Most of the example interchanges refer to the PRIMES program found on the Pascal/Z Distribution Diskette.

MODIFY

The commands under the modify heading allow the user to set or to modify various aspects of the program.

~MB~ [stmt # [mod #]] Modify the breakpoint table

This command allows the user to set and to modify breakpoints -- statement and module numbers (NOT line numbers) at which program execution should be halted. Program execution will halt before the break statement is executed.

To add a breakpoint, simply type MB and when prompted, type the statement and module numbers of the desired breakpoint. The numbers should be separated by a space. If a module number is not specified, the default will be module 0.

To delete a breakpoint, follow the same procedure; if the statement and module numbers specified are those of an existing breakpoint, that breakpoint will be eliminated from the breakpoint table.

The breakpoint table will hold a maximum of ten breakpoints at any one time.

EXAMPLE

```
--->~MB~
Breakpoint -- ~22~
Breakpoint -- added.
--->~MB 22 0~
Breakpoint -- deleted.
--->
```

~MC~ [variable] Modify break conditions

This command allows the user to set conditional breakpoints -- breakpoints which will become effective only if certain conditions occur during program execution. A conditional breakpoint will cause program execution to halt only if the variable reaches the condition specified.

To set a conditional breakpoint, type MC and when prompted, type the variable to be watched. If the variable is a structured type, the user will be prompted for a subscript or field name. The subscript must be an integer.

The user will then be prompted for a "relop", a relational operator. The permissible relops are:

LT	Less than (<)
GT	Greater than (>)
LE	Less than or equal to (<=)
GE	Greater than or equal to (>=)
EQ	Equal to (=)
NE	Not equal to (<>)

Once a relop has been specified, the user will be prompted with: "Var or Const -- "

At this point, specifying Const or C (constant) will elicit a prompt for an constant value. Specifying Var or V (variable) will elicit another prompt for the name of the variable. If the variable given is a structured type, the user will again be prompted for a subscript or field name. The subscript must be an integer.

The debugger will always look first for a global variable of the name given. If the variable initially specified is not found globally, the debugger looks at the procedure/function stack, starting with the innermost call, and finds the first local variable of that name. The break condition is then set on that variable. When the break condition is reached, the debugger will display the break condition, indicate that the variable is a local variable, and give the name of the procedure or function in which the variable may be found.

Local variables may be accessed only when the procedure/function is active, i.e. when the program counter is inside the procedure or function.

A break condition may be specifically set on a local variable by specifying the module number, procedure/function name, and the variable name separated by colons. The user will be prompted as usual if the variable is a structured type.

If a break condition is set and the variable does not reach the condition before the procedure/function is exitted, that condition will usually have no effect.

Problems can arise, however, since the object of the break

is stored in a specific memory location. If the contents of that memory location are changed later in the program (i.e. after the procedure/function is exitted), the break condition will be displayed, but will not be accurate since that memory location is no longer associated with the local variable specified. Therefore caution is advised when setting break conditions (including a watch, described later) involving local variables.

Once the conditional breakpoint has been reached, it is disabled.

Conditional breakpoints may only be set for integers, characters, booleans and enumeration types.

There may be a maximum of ten conditional breakpoints specified at any one time.

EXAMPLE 1

```
--->~MC~
Variable -- ~count~
Relop --- ~lt~
Var or Const -- ~c~
Value -- ~5~
--->
```

EXAMPLE 2

```
--->~MC~
Variable -- ~a~
Relop --- ~eq~
Var or Const -- ~v~
Variable -- ~employee~
Field name -- ~age~
--->
```

EXAMPLE 3

```
--->~MC~
Variable -- ~0:factor:i~
Relop -- ~ge~
Var or Const -- ~c~
Value -- ~4~
--->~MC~
Variable -- ~i~
Local: FACTOR
Relop -- ~ge~
Var or Const -- ~c~
Value -- ~4~
--->
```

Thus the final expression for Example 1 will equivalent to:

```
When count < 5, halt
```

and the final expression for Example 2 will be equivalent to:

```
When count = employee.age, halt
```

The final expressions for Example 3 will be equivalent to:

```
When i (a variable local to the procedure/function
FACTOR) >= 4, halt
```

~MV~ Modify the variable value

This command allows the user to modify the value of a variable, but not its type. Both global and local variables may be modified, although the procedures differ with the scope of the variable, as described below.

When given the name of the variable to modify, the debugger will first look for a global variable of that name. If the variable specified is not found globally, then the debugger looks at the procedure/function stack, starting with the innermost call, and finds the first local variable of that name.

The following types of variables may not be modified:

Any REAL variables (may only be displayed)
Pascal file variables (cannot be modified or displayed)
Pointers (may only display the value pointed to)
Strings (may only be displayed)
Sets (may only be displayed)
Arrays of: arrays or file variables (cannot be modified or displayed), and other excepted variables (cannot be modified).

Fields of records and elements of arrays may be modified only if they contain types which may be legally modified.

Note that a variable may not be modified by specifying the the name of another variable as the new value.

Modifying Global Variables

~MV~ [variable]

To modify a global variable, type MV and when prompted, specify the variable name. If the variable is a structured type, the user will be prompted for a subscript or a field name. The subscript must be an integer.

When prompted again, specify any constant as the new value for the variable.

EXAMPLE

```
--->~MV~
Variable -- ~count~
New value -- ~6~
--->~MV~
Variable -- ~employee~
Field name -- ~age~
New value -- ~23~
--->~MV~
Variable -- ~days~
Subscript -- ~monday~
New value -- ~thursday~
--->
```

Modifying Local Variables

~MV~ [module #:procedure/function name:local variable]

Local variables may be accessed only when the procedure/function is active, i.e. only when the program counter is inside the procedure or function.

To specifically modify a local variable, type MV and when prompted, specify the variable by module number, procedure/function name and the local variable name, separated by colons. As usual, if the variable is a structured type, the user will be reprompted.

Parameters being passed to a procedure or function may also be accessed in the same way.

EXAMPLE

```
--->~MV~
Variable -- ~0:factor:divide~
New value -- ~false~
--->~MV~
Variable -- ~0:update:employee~
Field -- ~age~
New value -- ~32~
--->
```

DISPLAY

The commands under the display heading allow the user to view the status of various aspects of the program: the breakpoints, the break conditions, the procedure/function stack, the runtime requirements, statement and module numbers, and variables.

~DB~ Display the breakpoint table

This command displays the statement and module numbers of currently set breakpoints. This will not display break conditions.

Note that the debugger will allow the user to set breakpoints after the end of the program, but that this will have no effect upon program execution.

EXAMPLE

```
--->~DB~
Statement -- 12, module 0
Statement -- 23, module 0
Statement -- 14, module 1
Statement -- 32, module 3
--->
```

~DC~ Display break conditions

This command displays any conditional breakpoints (including watches) currently set. This will not display ordinary breakpoints.

EXAMPLE

```
--->~DC~
COUNT      LT Integer: 5
A          EQ AGE
PRIMES     GT Integer: 4
--->
```

~DN~ Display the last ten statements executed

This displays the statement and module numbers of the last ten statements executed.

EXAMPLE

```
--->~DN~
Statement -- 9, module 0
Statement -- 10, module 0
Statement -- 11, module 0
Statement -- 12, module 0
Statement -- 13, module 0
Statement -- 1, module 0
Statement -- 2, module 0
Statement -- 14, module 0
Statement -- 22, module 0
Statement -- 27, module 0
--->
```

~DP~ Display the procedure/function stack

This displays the procedure/function stack, specifying each procedure or function called and the statement and module numbers from which it was called.

A maximum of twenty-five calls will be displayed. If there have been more than twenty-five calls, only the outermost calls will be displayed. The debugger will give a message specifying the number of calls which are not displayed.

EXAMPLE

```
--->~DP~
FACTOR   called from   Statement -- 13, module 0
FACTOR   called from   Statement -- 21, module 0
--->
```

~DR~ Display the current runtime requirements

This displays the current runtime requirements, including the locations of the stack and heap pointers, the amounts of used stack and heap space, and the amount of memory remaining.

Note that the debugger occupies 12K, so that this extra space will be available when the debugger is not invoked.

EXAMPLE

```
--->~DR~
Stack pointer  Stack usage    Heap pointer  Heap usage    Free
BC71           2739           4FD3          0000          6C7E
--->
```

~DS~ Display current statement and module numbers

This displays the statement and module numbers of the current location of the program counter, as well as the program block.

EXAMPLE 1

```
--->~DS~
<MAIN>   Statement -- 16, module 0
--->
```

EXAMPLE 2

```
--->~DS~
FACTOR   Statement -- 13, module 0
--->
```

~DV~ Display the variable type and value

This command allows the user to display the type and value of any variable. Both global and local variables may be displayed, although the procedures differ with the scope of the variable, as described below.

If the variable specified is not global, the debugger looks at the procedure/function stack, starting with the innermost call, and finds the first local variable of that name. The variable name, type and value are then displayed, as well as the name of the procedure or function in which the variable can be found.

The only variables which may not be displayed are Pascal file variables, arrays of arrays, and arrays of file variables. Note that when displaying a pointer variable, the value displayed will be that of the variable pointed to.

Displaying Global Variables

~DV~ [variable]

To display a global variable, type DV and when prompted, specify the variable. If the variable is a structured type, the user will be prompted for a subscript or field name. The subscript must be an integer.

EXAMPLE

```
--->~DV~
Variable -- ~count~
Integer: 1
--->~DV~
Variable -- ~primes~
Subscript -- ~4~
Integer: 5
--->~DV~
Variable -- ~prime~
Boolean: TRUE
--->~DV~
Variable -- ~record~
Field name -- ~alpha~
Char: g
--->~DV~
Variable -- ~name~
String: laurie
--->~DV~
Variable -- ~i~
Local: FACTOR
Integer: 3
--->~DV~
Variable -- ~season~
Set contains:
Enumeration: SPRING
Enumeration: SUMMER
Enumeration: AUTUMN
Enumeration: WINTER
--->~DV~
Variable -- ~collection~
Subscript -- ~2~
Field name -- ~c~
Boolean: FALSE
--->~DV~
Variable -- ~amount~
Real: 3.399999E+00
--->
```

CONTINUE

The commands under the continue heading allow continuation of program execution under user control.

~CB~ Continue execution to the next breakpoint

This command will cause program execution to begin at the current statement and to continue until the next specified breakpoint or break condition has been reached. The statement and module numbers of the breakpoint (or the break condition met including the statement and module numbers at which it was reached) will then be displayed.

If no breakpoint has been set, the execution of the program will continue until finished.

EXAMPLE 1

```
--->~CB~

Break encountered -- FACTOR   Statement -- 12, module 0

--->
```

EXAMPLE 2

```
--->~CB~

Break encountered -- COUNT   LT Integer: 5
<MAIN>   Statement -- 17, module 0

--->
```

EXAMPLE 3

```
--->~SR FACTOR~
--->~CB~

Break encountered -- FACTOR   Statement -- 1, module 0

--->
```

~CN~ [#] Continue execution for [#] statements, or until the next breakpoint

This command will cause program execution to begin at the current statement and to continue for the number of statements specified in brackets. The statement and module numbers of the breakpoint or break condition will then be displayed.

If no number is given, the default is 1. Once a number is specified, that number will become the default until it is changed by giving the full CN [#] command again.

If a breakpoint or break condition is reached before the number of statements specified in brackets is completed, execution will halt at the breakpoint.

EXAMPLE

```
--->~DS~
<MAIN>   Statement 16, module 0
--->~CN~
<MAIN>   Statement 17, module 0
--->~CN 2~
<MAIN>   Statement 19, module 0
--->~CN~
<MAIN>   Statement 21, module 0
--->
```

QUIT

~Q~ Quit the debugger

This command will cause an exit from the debugger and control will be returned to the operating system.

Note that Control C checking must be disabled or user output files will not be closed upon exitting the debugger.

EXAMPLE

```
--->~Q~

Cache CP/M  4h   3/25/81

A>
```

SET

The commands under the set heading allow the user to set or enable various functions of the debugger.

~SE~ Set the procedure/function entry/exit indicator

This command will cause the name of each procedure and function to be displayed as it is entered and exitted.

Giving this command will reverse its current status: if enabled, the command will return TRUE, and if disabled, the command will return FALSE.

EXAMPLE

```
--->~SE~
TRUE
--->~CN~
Entering FACTOR
Leaving Factor
--->~SE~
FALSE
--->
```

~SP~ Set the program counter to the beginning of the main program

This command will set the program counter to the beginning statement of the main program, not to the first line of the program.

EXAMPLE

```
--->~SP~
<MAIN>   Statement -- 16, module 0
--->
```

~SR <routine>~ Set a watch on the routine

This command allows the user to set a watch on routine (a procedure or function). Each time the procedure or function is about to be entered, a breakpoint will be generated by the debugger.

To enable the routine watch, type SR followed by the name of the procedure or function to be watched. This information must be included in the initial command line: there will be no prompt as in other commands where additional information is optional.

To disable the watch, simply type SR. There is no need to enter the routine name.

Only one watch on a routine may be in effect at any time.

EXAMPLE

```
--->~SR FACTOR~
--->
```

~ST~ Set the trace indicator

This command will toggle the "trace" compiler option, and display the statement and module numbers of each statement as it is executed.

Giving this command will reverse its current status: if enabled, the command will return TRUE, and if disabled, the command will return FALSE.

EXAMPLE

```
--->~ST~
TRUE
--->~CN 3~
<MAIN>   Statement -- 17, module 0
<MAIN>   Statement -- 18, module 0
<MAIN>   Statement -- 19, module 0
--->~ST~
FALSE
--->
```

~SW~ [variable] Set a watch on the variable

This command will set a watch on the specified variable. When the value of the variable changes in any way, program execution will be halted and the break condition and statement/module numbers will be displayed. A watch is essentially the same as a break condition, only no specific condition is to be met--change is the only criterion.

To set a watch, type SW and when prompted, type the variable to be watched. If the variable is a structured type, the user will be prompted again for a subscript or field name. The subscript must be an integer.

To set a watch on a local variable, see the section on "Modifying break conditions" (MC) earlier in this manual.

When a watch is set on a variable, the break condition [variable] NE [constant] will appear in the table of break conditions (not the breakpoint table) and can be displayed with the DC (display break conditions) command.

The watch is disabled the first time the variable changes.

A watch may only be set on the following variables: integers, characters, booleans and enumeration types.

EXAMPLE

```
--->~SW~
Variable -- ~count~
--->~CN~

Break encountered -- COUNT   NE Integer: 3
<MAIN>   Statement -- 18, module 0

--->
```