MSB-107

TO: Distribution

FROM: J.M. Broughton

DATE: 9 July 1973

SUBJECT: Proposal for a Symbolic Debugger

The purpose of this debugger is to allow the user to deal with the constructs and environment of his program in symbolic terms and with the minimum of fuss. It is also designed to provide him with great flexibility in specifying what he wants done via a unique command syntax.

The debugger has the ability to print and alter the value of variables. It allows the user to set breaks and "insert" debugger commands, effectively programming what is to happen at the break. A facility for examining and manipulating the stack is also provided, as is the ability to effect (possibly non-local) transfers of control. Finally, it is possible to call procedures, and to a somewhat limited extent, invoke functions.

I. Basic Concepts

The debugger is orienter toward individual source program statements; as a result, one does not set a break at a particular location (i.e., machine instruction), but rather, before or after some specified statement. Along these lines, the debugger also provides a simple set of commands for listing the source program.

This debugger maintains three "pointers" which most commands reference in some fashion:

source pointer
symbolic pointer
control pointer

The source pointer gives the number of the source statement currently being dealt with. The symbol pointer indicates the current block and generation of storage from which to evaluate reference to variables. The control pointer marks the point at which execution was last suspended, that is, the debugger was entered. All three may be altered separately.

II. How the Debugger is Invoked

The debugger may be called from command level by issuing the command:

```
          debug <path>
```

where <path> is an optional argument giving the pathname of the segment to be examined. If it is present, it will be "used" as the current procedure (see the "use" command); otherwise, the debugger will look for a faulting frame, and if present, the owner will be used; failing that, the immediately preceding frame will be used.

If a break is encountered while the a program is executing, the debugger will effectively be called. The source pointer will be set to the source of the statement at which the break was set; the symbol pointer will be set to the current frame and block; and the control pointer will be set to the next statement to be executed.

III. The Command Syntax

```
<command list>:: =<command>[{;|<nl>}<command>] ...

<command>:: =<simple command>|<conditional command>

<conditional command> :: = <predicate>{<simple command |
              <command list>}

<predicate> :: = {if|while} (<conditional>)

<conditional> :: =<bit value>|¬<bit value>|<value>
              {=|¬=|<=|>=|¬<|¬>}<value>
```

Essentially, one can give any number of commands on a line, if they are separated by semi-colons. The execution of a command or list of commands can be controlled by placing a predicate in front of it. For the case of "if (--)", the command(s) will be executed over and over again so long as the value is true. Since a full command list with this conditional feature can be "inserted", it is possible to control what happens at a break: for example, stop only if some condition is met.

IV. Commands

⟨brief command⟩:: = brief [on|off]

Controls the "verbosity" of the debugger.

⟨exec command⟩:: = exec "⟨string⟩"

Pass the ⟨string⟩ to the command processor.

⟨stack command⟩:: = stack [i [,n]] all

Trace the stack backward from the current or ith frame for n frames. If there are no limits given, the entire stack will be traced. Normally, the stack will be "cleaned up" by ignoring system routines; specifying "all" will cause them to be included in the trace.

⟨status command⟩status [⟨break specification⟩]
⟨break specification⟩:: = * |⟨procedure⟩|⟨ statement⟩

List information about breaks: for "*", give all segments containing breaks; for ⟨procedure⟩ or ⟨statement⟩ , give information about the breaks in that segment or at that line. The default is to list the break at the current statement.

⟨block⟩ :: = ⟨procedure⟩|⟨statement⟩ | level i
⟨use command⟩ :: = use ⟨block⟩

This may be used to set the symbol pointer. It will be set to the block specified by level i (from "stack"), an invocation of ⟨procedure⟩, or the block containing ⟨statement⟩. If the block given is not in the current source segment, the source pointer will be moved to that segment (at the point where execution was suspended). If the debugger has just been entered, the control pointer will be set to the point in the given procedure where execution was suspended.

⟨where command⟩:: = where

Prints status of all three pointers.

⟨list command⟩:: = list [i]

Print one or i lines beginning with the current statement.

⟨position command⟩:: = position ⟨statement⟩|⟨offset⟩

⟨offset⟩:: = [+|-] n

Make source pointer point at the line given by ⟨statement⟩ or by ⟨offset⟩ relative to the current statement.

⟨find command⟩:: = find "⟨string⟩"

All executable statements, starting at the current statement and wrapping around, will be searched for an occurrence of "⟨string⟩". If found, the source pointer will be set to that statement.

⟨insert command⟩:: = insert [⟨statement⟩] (⟨command list⟩)

Insert before the (current) statement the ⟨command list⟩ given, that is, set a break there.

⟨append command⟩:: = append [⟨statement⟩] (⟨command list⟩)

Same as above, except insert commands <u>after</u> the (current) statement.

⟨reset command⟩:: = reset [⟨break specification⟩]

Reset the breaks associated with the statement or procedures indicated (as with "status").

⟨stop command⟩:: = stop

This is essentially a call to the debugger. Input will be taken from the console as opposed to a break or a ("switched") file. If input is already from the console, another invocation of the debugger will still be created.

⟨pause command⟩:: = pause

This has the effect of making any break it appears at temporary. It is equivalent to "stop; reset".

⟨let command⟩:: = let ⟨element⟩ = ⟨value⟩

Assign value to the variable given by ⟨element⟩.

⟨goto command⟩:: = goto ⟨statement⟩

Transfer to ⟨statement⟩.

⟨call command⟩:: = call ⟨procedure⟩ [⟨argument list⟩]

Call the specified procedure.

⟨print command⟩:: = print {⟨expression⟩|⟨element⟩}

Print the items specified according to their data types. If the builtin functions addr or octal are used, the address or internal representation (in octal) will be given.

⟨step command⟩:: = step

Execute one statement and stop.

⟨integer command⟩ :: = [global] integer ⟨symbol⟩

This command defines a variable to be known only to the debugger. The variable will only be known in the current procedure (the one containing the current block) unless the global attribute appears. In that case, the variable will be known everywhere. Both definitions will last from process to process. Per-procedure names will be found before global names.

⟨macro command⟩ :: = [global] macro ⟨symbol⟩ (⟨command list⟩)

This command allows the user to define his own command macros for use in the debugger. The global and procedure distinctions are as above. The name may not be the same as a variable.

Note: In order to find an identifier the following procedure is followed: search the table of debugger variables for this procedure, then global variables, then search the procedure's symbol table and then (unless procedure is allowed) give up. If generation information appears, the search will begin with the procedures symbol table.

⟨do command⟩ :: = do ⟨symbol⟩

This will cause input to be changed to the source of the macro. The lines in the macro will be processed and executed as normal input.

⟨erase command⟩ :: = [global] erase ⟨symbol⟩

This will erase the definition of the variable or macro given by ⟨symbol⟩.

⟨continue command⟩ :: = continue

This will cause control to "continue" from where it left off -- that is, the debugger was entered.

⟨end command⟩ :: = end

This will cause the execution of a macro to be terminated, and the previous command stream to be re-entered.

V. <u>Expressions, Values, and References</u>

```
<expression>:: = <value>|<function>
   <function>:: = <procedure>[< argument list >]

<value>:: = <builtin> |<reference>|< constant>
   <builtin>:: = addr (<reference>) | octal (<reference>)

<reference>:: = {<simple> |<subscripted> |< structure>|< locator >}
         [<generation>] | <symbol>
   <simple>:: = <identifier>
   <subscripted>:: = <identifier> (<subscript>[,<subscript >]...)
      <subscript>:: = <reference>|<constant>
   <structure>:: = <member>[.<member>]...
      <member>:: = <simple>|< subscripted>
   <locator> :: =<reference> ->{<simple>|<subscripted>|<structure>}
   <generation>:: =   [block >| i]

<element>:: = <reference>|<iterated reference>
   <iterated reference>:: = a normal<reference> except a
        <subscript> may be:
           <iterated subscript>:: = <subscript>: <subscript>

<argument list> :: = (<value>[,<value>]...)
```

```
<constant>:: =<arithmetic> |<string> |< bit > |< pointer>
   <arithmetic>:: = [+|-] {< complex >|< real>|< imaginary >}
      <complex >:: = <real> {+1-}< imaginary >
      <imaginary>:: = <real>i
      <real>:: = <binary> |<octal >|< decimal >
        < binary>:: ={ 0| 1} ...b
         <octal>:: ={ 0|1|2|3|4|5|6|7} ...o
         <decimal>:: = <fixed> |<float >
            < fixed>:: =<integer> [<integer >]
            <float>:: = <fixed><exponent>
               <exponent> :: = e[+|-]< integer >
            <integer>:: ={0|1|2|3|4|5|6|7|8|9} ...
   <string>:: =  " [< character>]... "
   <bit> :: = "[0|1]..."b
   <pointer>:: = s|w [(b)]
```

< procedure>:: = any PL/I entry variable or constant. (Note:
if the symbol cannot be found, it will be assumed to be
an external entry and the user's search rules will be
followed to find it.)

```
<statement>:: =  {<label>|[f-] i | $ <special>} [,s]
   <special>:: = b|c
```

Where label is an identifier specifying a label constant or
variable, [f-] i represents the first statement on line i in file f,
$b gives the statement where a break last occurred, and $c gives
the current statement. "s" gives a statement offset from that
given by the above.

VI. Implementation Considerations

REQUIREMENTS OF EACH STATEMENT

	Symbol Table	Statement Map	Special Compile Mode
brief	-	-	-
exec	-	-	-
stack	-	x	-
status	(x)	-	-
switch	-	-	-
use	(x)	x	-
where	-	x	-
list	(x)	x	-
position	(x)	x	-
find	-	x	-
insert	(x)	x	-
append	(x)	x	-
reset	(x)	-	-
stop	-	-	-
pause	-	-	-
step	-	x	*
let	x	-	-
goto	(x)	-	-
call	x	-	-
print	x	-	-
return	-	-	-

(x) Indicates need for symbol table only to evaluate labels.

* In order to implement it in the easiest manner.