Date: July 30, 1974

To: Distribution

From: Jeff Broughton

Subject: A New Symbolic Debugger

The current system debugger, debug, is not well suited for use by the unsophisticated user. It is very much machine language oriented and has a confusing and error prone syntax. In addition, it is deficient in its handling of include files, quick blocks, and certain data types. Probe is intended to be more simple to use and to deal with the constructs of the user program in a more straightforward way. Notable differences between it and debug are:

1) Probe cannot modify or examine code.

2) Breakpoints are implemented in such a manner that an active invocation of probe need not be on the stack for a break to occur.

3) The syntax for breaks is potentially far more flexible.

4) Quick procedures and blocks, as well as normal begin blocks are recognized in a stack trace. Support procedures are excluded (at the user's discretion) from the stack trace.

5) Type checking and conversion is performed in assignments.

6) Arguments are converted to expected type in a call, if entry argument descriptors are present.

7) A wider range of constants, including decimal and complex, is supported.

There will soon be a version available for use in the Multics Library. Comments are welcome.

Due to a bug in the runtime symbol table, the address of entries in the program being examined cannot be found. As a result, the "call" and "use" commands currently cannot be used with those entries. Please report any other bugs you find to me.

Name: probe, pb

The probe command is a symbolic debugging aid that allows the user to interactively examine the state of his program. Commands are provided to display and alter the value of variables, to interrupt a running program at a particular statement by use of breakpoints, to list the source program, to examine the stack of block invocations, and to invoke external subroutines and functions.

In order to debug a program with probe, the program must have a standard symbol table that contains information about variables defined in the program and a statement map that gives the correspondence between source statements and object code. A symbol table and statement map is produced by the PL/I and Fortran compilers if the "-table" option is specified. (A program can also be compiled with the "-brief_table" option which will produce only the statement map and disable the ability to reference variables.)

To store certain information about programs being debugged, probe uses a segment in the user's home directory called Username.probe where Username is the user's personid. This segment is created automatically when needed.

Introduction:

The primary use of probe is to examine a program whose execution has been suspended. This can occur in one of several ways.

First, execution can be interrupted as a result of an error occurring in the program such as zerodivide or overflow. After an error message is printed on the user's console, and a new command level entered, probe can be called and commands issued to it to identify the cause of the error.

econd, the user can, as always, stop a run-away program by "quitting".

Third, the user can designate, by use of probe's break commands, statements on which the program is to stop and directly enter probe. A list of commands associated with the break would then be executed automatically. These commands could print a variable, tell what line was just executed, or cause probe to read additional commands from the console. In this way, the user can follow the progress of his program before an error occurs.

In all of the cases above, an active program has been suspended. This means that variables of all storage classes, in particular automatic, exist and can be displayed. Probe can also be used to examine a non-active program -- one that has never been run or that has completed. Used in this manner, probe can be used to look at static variables, and the program source, though the most common use is to set breaks before actually running the program.

Probe keeps track of a current statement, a current block, and a current control point. The current statement designates a particular source program statement and is referenced by many commands. The current block identifies a procedure, subprogram, or begin block whose variables are to be examined. Moreover, it specifies a particular stack frame occupied by that block so as to differentiate between different occurrences of a variable in a recursively invoked procedure. The current control point marks the statement at which execution was suspended in the user's program. For convenience, they will be refered to as the source pointer, the block pointer, and the control pointer, respectively.

Usage:

probe -<procedure>-

where <procedure> is an optional argument that gives the name of an entry in which the user is interested. If the procedure is active, the control and source pointers are set to the last statement executed, and the block pointer is set to the most recent invocation of the procedure. If it is not active, then the control and source pointers are set to point to the entry statement, and the block pointer designates the outermost block of the procedure.

If a <procedure> is not specified, probe checks if an error or quit has occurred and, by default, uses the procedure that was executing. The pointers are set as if the user had specified it explicitly. If no error has occurred, then probe prints a message and returns.

To execute a program that contains a breakpoint, the program can be called normally from command level, or from within probe by use of the call or value requests. Note well: for the breakpoint to take effect, proe must be must be invoked at least once in the process.

When probe is entered as the result of executing a procedure with a breakpoint set in it, the control and source pointers are

set to the statement on which the break was set, and the block pointer to the block that contains that statement.

In general, after an error, quit, or break, things are set up by default much as one would expect. The user should, however, explicitly name a <procedure> when he is interested in working with a non-active one.

Once probe has been entered, the user can issue commands to it in order to examine his program.

Command Syntax:

The command language recognizes three constructs: simple commands, command lists, and conditional commands. Loosely, a simple command is a basic probe request, and a command list is a list of commands separated by semi-colons (or newlines). A conditional command is a simple command or list (surrounded by parentheses) prefixed by a conditional predicate that controls when the request is to be performed. Examples follow in the next section.

In the discussion of commands that follows, meta-language symbols are used for certain constructs (e.g. <expression>). Their meaning should be apparent from context and from examples given. A complete discussion can be found later in this document.

Basic Commands

value, v

```
value {<expression>|<cross section>}
```

Output on the console the value of <expression>. The value request allows the user to display the value of variables, builtin functions such as addr and octal, and the value returned by an external function.

```
value var
value p -> a.b(j).c
value addr (i)
value octal (ptr)
value function (2)
```

Array cross-sections can be displayed by specifying the upper and lower bound of the cross-section as follows:

```
value array (1:5, 1)
```

which would print array(1,1), array(2,1), ..., array(5,1). More than one dimension can be iterated; for instance a(1:2,1:2) would print, in order, a(1,1), a(1,2), a(2,1), a(2,2).

let, l

```
let {<variable>|<cross section>} = <expression>
```

Set the <variable> specified to the value of the <expression>. If the types are not the same, conversion is performed according to the rules of PL/I. Array cross-sections can be used with the same syntax as in print. Note that one may not assign one array cross-section to another.

```
let var = 2
let array (2,3) = i + 1
let p -> a.b(1:2).c = 10b
let ptr = null
```

Warning: because of compiler optimization, the change may not have immediate effect in the program.

continue, c

```
continue
```

Cause probe to return to its caller. If entered from command level, probe returns to command level. After a break, the user's program is, in effect, restarted. To abort a debugging session, the quit button must be used.

call, cl

```
call <procedure([<expression[,<expression>]...])
```

Call the subroutine with the arguments given. If the procedure has descriptors that gives the type of the arguments expected, the ones given are converted to the expected type; otherwise, they are passed without conversion. The print request can be used to invoke a function, with the same sort of argument conversion taking place. Note: if the procedure has no arguments, a null argument list, "()", must be given.

```
call sub ("abc", p -> p2 -> bv, 25o, addr(j))
call sub_noargs ()
print function ("010"b)
```

goto, g

```
goto <label>
```

Cause an exit from probe and a non-local goto to the statement specified.

```
goto label_var          - transfer to value of label variable
goto action (3)         - transfer to label constant
goto 29                 - transfer to statement on line 29 of current program
goto $110               - transfer to line labeled 110 in the fortran program
goto $c,1               - transfer to the statement following the current statement
```

Warning: because of compiler optimization, unpredicatable results may occur.

Source Commands

source, sc

```
source [n]
```

Directs one or n statements beginning with the current statement (i.e. the source pointer) to be printed. Note: only executable statements for which code has been generated can be listed; however, if several statements are requested, intervening text such as comments and non-executable statements is included in the output.

position, ps

```
position [<label>]
position (+|-)n
```

Set the source pointer to the statement indicated or to plus or minus n executable statements relative to the current statement. If no label or offset is given then the statement designated by the control pointer is assumed.

```
position label          - set the source ptr to label: ...
position action (3)     - to action(3): ...
position 2-14           - to statement on line 14 of file 2
                          of the program
position +2             - move forward 2 statements in the
                          source
position -5             - move back 5 statements
```

In addition, the position command can be used to search for an executable statement that contains a specified string, and if found set the source pointer to that statement:

```
position "<string>"
```

The search begins after the current statement and continues around the program as in the editors edm and qedx. Note: because of reordering of statements by the compiler, which, among other things, moves subprograms to the end, the search may not necessarily find things in the same order as one would expect from a source listing of the program.

```
position "write (6,10)"  - locate the statement in the
                           program
position "str = ""a"     - locate str = "a
position "q+2"; source   - locate and print the statement
```

Symbol Commands

stack, sk

stack [[i,]n] [all]

Trace the stack backward from the ith frame for n frames. If no limits are given, the entire stack is traced. The trace consists of a list of active procedures and block invocations (including quick blocks) beginning with the most recent. In addition to the name of the block, a frame or level number is given, as is the name of any conditions raised in the frame.

stack	- trace the whole stack
stack 3	- trace the three most recent frames
stack 3, 2	- trace th 3rd and 4th frames

Normally, system or subsystem support procedures will not be included in the stack trace. If desired, they may be included by specifying "all".

```
stack all
stack 3,5 all
```

use, u

use [<block>]

Selects a new block or procedure to be examined. If no <block> is given, then the block originally used when probe was entered is assumed. The block pointer is set to the <block> specified so that variables in that block can be referenced. In addition, the source pointer is set to the last statement executed in the block; in this way, the point at which the block exited can be found with the help of the source command. Acceptable <block>s include:

```
<procedure>
<label>
level i
- n
```

Here <procedure> is the name of a procedure whose frame is desired; its usage is essentially the same as if used on the command line. A <label> denotes the block that contains the statement identified by the label or line number -- for instance, the label on a begin statement denotes that begin block. If the <label>s block is not active, the source pointer is set to the

statement specified. "level i" uses the ith block frame from a stack trace. "-n" uses the nth previous instance of the current block, allowing one to move back to a previous recursion level. (If more frames are requested than actually exist, the last one found is used.)

use sub	- use block procedure sub occupies
use label	- use block that contains label:
use level 2	- use second frame in stack trace
use -1	- use previous instance of current block
use -999	- use last (oldest) instance

Note: when a level is specified, the last trace mode specified (support procedures included or excluded) is used to find the level requested.

symbol, sb

symbol <identifier>

Display the attributes of the variable specified and the name of the block in which its declaration is found. If the variable has variable size or dimensions, an attempt is made to evaluate the size or extent expression; if the value is not available, then "*" is used instead.

where, wh

where [source|block|control]

Display the current value of one or all of the pointers. Source and control give the statement number of the corresponding statement. Block gives the name of the block currently being used; if the block is active, its level number is also given.

where	- give value of all three pointers
where source	- give the value of the source pointer

Break Commands

before, b

before [<label>][: (<command>|(<command list>))]

Set a breakpoint before the statement specified by <label> and cause the command(s) given to be associated with the break. If no <label> is given, the current statement is assumed. If no commands are given, "halt" is assumed. When the running program arrives at the statement, probe is entered before the statement is executed, and the commands are processed automatically. When finished with the commands, probe returns, and the program resumes at the statement at which the break was set. In effect, the user can "insert" probe commands into his program.

```
before: (value var; value var2)
                              - set a break before the current
                                statement
before quick: value x         - set a break before the statement
                                labeled quick
before                        - set a break with the "halt"
                                command before the current
                                statement
```

Note that the command list may extend across line boundaries if necessary.

after, a

after [<label>][: (<command>|(<command list>))]

is the same as before except that the break is set after the statement designated. This means that the command list is interpreted after the statement has been executed. If the statement branches to another location in the program, probe is not entered. The difference between setting a break after one statement and setting another before the next is that a transfer to the next statement causes a break for the before case but not for the after case.

halt, h

halt

Causes probe to stop processing its current input and read commands from the console. A new invocation of probe is created with new pointers set to the values at the time "halt" was executed. It is of primary use as part of a break command list

as it enables the the user to enter commands while a program is suspended by a break. In effect, he can halt a running program. A subsequent continue command causes probe to resume what it was doing before it stopped -- for instance, finish a break command list and return to the program. The command:

```
before 29: halt
```

causes the program to halt at statement 29 and allows the user to enter probe commands. Continue would restart the program. Similarly:

```
after: (value a; halt; value b)
```

causes the value of a to be printed before the program halted; later, after the user entered a "continue" command, the value of b would be printed, and the execution of the program resumed.

reset, r

```
reset
reset (at|after|before) <label>
reset <procedure>
reset *
```

Delete breaks set by the before an after commands. Just "reset" deletes the last break that occurred; the <label> form deletes breaks set before and/or after a statement; <procedure> and "*" can be used to reset all the breaks in a segment, and all breaks in all segments, respectively.

```
reset                   - delete the current break
reset at 34             - delete breaks set before and
                          after 34
reset after 34          - delete the break set after 34
reset sub               - delete all breaks in sub
reset *                 - delete all known breaks
```

status, st

```
status
status (at|after|before) <label>
status <procedure>
status *
```

Give information about what breaks have been set. The scope of the requests is similar to "reset":

status	- list the current break
status before label	- list the break set before the statement at label:
status sub	- tell what breaks have been set in sub
status *	- tell what procedures have breaks set in them

pause, pa

pause

Equivalent to "halt; reset" in a break command list, it causes the procedure to execute a break only once -- stopping, then reseting the break.

step, s

step

Set break consisting of "pause" after the statement following the control pointer and "continue". It enables the user to step through his program one statement at a time. Note: if a statement transfers elsewhere, the break does not happen until sometime later, if ever.

Miscellaneous Commands

mode

mode (brief|long)

Turn brief message mode on or off. In brief mode, most messages generated by probe are much shorter and others are surpressed altogether. The default is long.

execute, e

execute "<string>"

Pass <string> to the command processor to be executed as a normal Multics command.

Conditional Predicates

if

```
if <conditional>: (<simple command>|(<command list>))
```

The command(s) are executed if the <conditional> evaluates to true. The <conditional> can be of the form <expression><op><expression> with <=, <, =, ^=, >, >= allowed as <op>s.

```
if a < b: let p = addr (a)
```

This predicate is of most use in a break command list as it can be used to cause a conditional stop:

```
before: if z ^= "10"b: stop
```

would cause the program to stop only when z ^= "10"b.

while, wl

```
while <conditional>: (<simple command>|(<command list>))
```

Allows iteration by executing the command(s) as long as the <conditional> is true.

```
while p ^= null: (print p -> r.val; let p = p -> r.next)
```

Expressions:

Allowable <expression>s include simple scalar variables, constants, and probe builtin functions. The sum and difference of computational values can also be used.

Variables can be simple identifiers, subscripted references, structure qualified references, and locator qualified references. Subscripts are also expressions. Locators must be offsets or pointer variables or constants.

```
running_total
salaries (p -> i - 2)
a.b(2).c(3) or a.b.c(2,3) etc.
x.y -> var
```

Arithmetic, string, bit, and pointer constants are supported. Arithmetic constants can be either decimal or binary, fixed or float, real or complex. Also, octal numbers are permitted as abbreviations for binary integers (e.g. 12o = 10).

```
-123          45.37          2.1-0.3i
10b           4.73e10        12345670o
```

Character and bit strings without repetition factors are allowed. Character strings can include newline characters. Octal strings can be used in the place of bit strings (e.g. "123"o = "001010011"b).

```
"abc"                    "1010"b
"quote""instring"        "01234567"o
```

Pointer constants are of the form: seg#|word#(bit#). The seg# and word# must be in octal. The bit# is optional and must be in decimal. They can be used as locators.

```
214|5764                 232|7413(9)
```

Three builtin functions are provided by probe: addr, null, and octal. The addr function takes one argument and returns a pointer to that argument. Null, taking no arguments, returns a null pointer. They are the same as in PL/I. The function octal acts very much like PL/I's unspec builtin in that it treats its argument as a bit string of the same length as the raw data value, and can be used in a similar manner as a psuedo-variable. However, when used in the print command the value is displayed in octal. (Data items not occupying a multiple of three bits will be padded on the right.)

Label References:

A <label> identifies a source program statement and can be a label variable or constant, a line number as it appears on a source listing (i.e. [file-]line), or a special statement designator: $c representing the "current statement", $b representing the statement on which the last break occurred, and $number for fortran labels. An optional offset of the form ",s" is also allowed.

label	- statement at label: ...
label_var	- statement to which label_var is set
17	- statement on line 17 of program
3-14,2	- statement 2 on line 14 of file 3
$b	- statement at which last break occurred
$c,1	- statement after current statement
$100	- fortran statement labeled 100

Generally, a label can also be the name on a procedure or entry statement.

Procedure References:

A <procedure> is considered to be a reference to an entry variable or constant. External names can be used.

Evaluation of Variable References:

When a variable is referenced in a command, probe attempts to evaluate it by first checking for an applicable declaration in the current block, and if necessary in its parents. If not found, the list of builtin functions is searched. Finally, when the context allows a <procedure>, a search is made following the user's search rules.

The block in which to look for a variable can be altered by the use command that sets the current block. For example, if "value var" displays the value of var in the current block, then "use -1; value var" displays the value of var at the previous level of recursion. A shorthand is available for referencing variables in other blocks -- an optional block specification:

<variable> [<block>]

where block is the same as in the use command. The use of <block>s in this manner does not alter the block pointer.

var[-1]	- looks for previous value of var
abc[other_block]	- looks in "other_block" for abc
xyz[39]	- looks in block that contains line 39
n.m[level 4]	- looks in block at level 4
q(2)[sub]	- looks in procedure sub

A block specification can be used on an identifier anywhere the variable could be used. However, a block specification on a label or entry constant is ignored unless 1) the relative (-n) format is used, and 2) the label or entry is itself used in a block specifcation. In such a case, it is taken to mean the nth previous instance of the block designated by the label or entry; that is, "var[sub[-2]]" references var in the second previous invocation (third on the stack) of sub.

Sample Debugging Session:

The following is a sample attempt at debugging a program. It is not claimed that the program does anything useful, or that this is the best way to debug the program. The purpose is merely to give an example of how certain probe commands can be applied. A listing of the source of the program, test, is given on the next page; the sample outout follows with ">" used to denote lines typed by the user.

In order to use probe to debug a program, the program must be compiled with the "-table" option. Generally, the user should generate a symbol table for any program that he does not have good reason to believe will work.

On line 5, the user calls his program; noticing that it seems to be looping, he stops it by hitting the quit button. After the user invokes probe, it responds by telling that the internal function "fun" was executing line 38 when interrupted. Since the source pointer was automatically set to that line, a request to print the current statement with "source", displays the source. The statement causing an error could be displayed in a similar manner.

The stack command was then used to see what called what. The ouput shows that procedure "test" was called from command level, and then, in turn, called fun. While fun was executing, a quit occured and established a new command level. To determine whether fun was called from line 17 or line 27 of test, the use command is used to find the point at which test exited. Since "use" also sets the block pointer at the same time, the user can check if "s.num" has the correct value with the value command.

The user decides that it would be worthwhile to trace the value of i. Rather than recompiling his program with a put statement added in a strategic location, probe allows him to set a break containing a value command to accomplish the same thing. Wanting to set the break after the do statement on line 16, the user searches for it with the position command. "source" is used to verify that the correct line was found. The continue command then causes probe to return (to command level).

To abort the suspended program test, the user gives the release command to Multics. If he had done this just after quitting, he could not have used probe to find out much about what happened.

```
 1 test: procedure;
 2
 3      declare
 4
 5           (i, j) fixed binary,
 6           1 s structure based (p),
 7             2 num fixed binary,
 8             2 b (n refer (s.num)) float binary,
 9           p pointer, n fixed binary,
10           sysprint file;
11
12
13      n = 5;
14      allocate s set (p);
15
16      do i = 1 to s.num;
17           s.b(i) = fun (i, 1);
18      end;
19      put skip list (s.b);
20
21      do j = s.num to 1 by -1;
22           s.b(j) = fun (-j, -1);
23      end;
24      put skip list(s.b);
25
26      return;
27
28
29      fun: procedure (b, i) returns (float binary);
30
31           declare
32                (b, i) fixed binary;
33
34           if b = 0
35                then return (1);
36                else do;
37                     b = b - i;
38                     return (2**b + fun (b, i));
39                end;
40
41      end fun;
42
43
44 end test;
```

The program is started once again, but now, after each time line 16 is executed, the break occurs and probe displays the value of i. Clearly, it is not being incremented as it should. Since this approach is not producing any useful information, the user aborts the program and trys to delete the break. The status command is used to tell what breaks have been set in the segment test, and then to see the break set. The break is then deleted with the reset command. Note that if there had also been a "Break before 16", then the command "reset at 16" would have deleted both.

The user next decides to see what is going on in fun, so he sets a break to halt it every time it is invoked. By looking at the listing, he knows that the first statement in fun is on line 34, so he "positions" the source pointer to that statement and sets a break to halt the program. To accomplish the same thing, "before 34: halt" could have been used.

The program halts when the break before line 34 is reached. The user displays b and i getting the values he expected. The where command is also used to see what the state of things is. Continue ("c") restarts fun which calls itself recursively and stops again. The stack command (showing the last five frames) verifies that fact. The user displays the b in the current instance of fun (at level 2) and in the previous one (at level 3). Mistakenly expecting the b's at different levels to be different, he gets suspicious. The variable "i" has the value expected, but the symbol command shows that it is wrong one — the parameter to fun, not the loop index. To get the correct one, he must look in the frame belonging to the procedure test. This "i" has been set to zero. The user then realizes his error. The function is modifying its argument (the loop index "i") on line 37. Done with debugging the program, "reset" is used to delete the currently active break (the one that just occurred), and the program is aborted.

```
      > pl1 test -table
        PL/I
        r 1248 3.211 28.336 280

      > test
(quit)
        QUIT
        r 1250 5.371 6.702 52  level 2, 10

      > probe
        Condition quit raised at line 38 of fun.
      > source
                              return (2**b + fun (b, i));
      > stack
          1    command_processor_
          2    release_stack
          3    unclaimed_signal
          4    real_sdh_
          5    return_to_ring_0_
          6    fun                                quit
          7    test
          8    command_processor_
          9    listen_
         10    process_overseer_
         11    real_init_admin_
      > use level 7
      > source
                    s.b(i) = fun (i, 1);
      > value s.num
            5
      > position "i = 1"; source
           do i = 1 to s.num;
      > after: value i
      > continue
        r 1252 1.375 16.394 354  level 2, 10

      > release
        r 1252 .126 .922 19

      > test
            1
            1
            1
            1
(quit)
        QUIT
        r 1252 3.069 .650 25  level 2, 12
```

```
> release
  r 1253 .092 .937 20

> probe test
> status test
  Break after line 16.
> status after 16
  Break after line 16:  value i
> reset at 16
  Break reset after line 16 of test.
> position 34
> source
                if b = 0
                     then return (1);
> before: stop
> continue
  r 1255 .781 12.356 333

> test
  Stopped before line 34 of fun.
> value b
      1
> where
  Current line is line 34 of test.
  Using level 2: fun.
  Control at line 34 of fun.
> value i
      1
> c
  Stopped before line 34 of fun.
> stack 5
     1    break
     2    fun
     3    fun
     4    test
     5    command_processor_
> value b
      0
> value b[-1]
      0
> value i
      1
> symbol i
  Attributes are: fixed binary(17,0) aligned parameter.
  Declared in: fun.
> use test
> value i
      0
```

```
 97        > reset
 98          Break reset before line 34 of test.
 99   (quit)
100          QUIT
101          r 1307 4.870 64.788 1544  level 2, 18
102
103        > release
104          r 1307 .076 .992 31
```

Summary of Requests:

after	a	Set a break after a statement.
before	b	Set a break before a statement.
call	cl	Call an external procedure.
continue	c	Return from probe.
execute	e	Execute a Multics command.
goto	g	Transfer to a statement.
halt	h	Stop the program.
if		Execute commands if condition is true.
let	l	Assign a value to a variable.
mode		Turn brief message mode on or off.
pause	pa	Stop a program once.
position	ps	Examine a specified statement or locate a string in the program.
reset	r	Delete one or more breaks.
source	sc	Display source statements.
stack	sk	Trace the stack.
status	st	Display information about breaks.
step	s	Advance one statement and halt.
symbol	sb	Display the attributes of a variable.
use	u	Examine the block specified.
value	v	Display the value of a variable.
where	wh	Display the value of probe pointers.
while	wl	Execute commands while condition is true.