TO: Distribution

FROM: J. W. Gintell

DATE: September 30, 1974

SUBJECT: New probe command

Attached is the documentation on the new probe command. Any
comments may be submitted to Jeff Broughton or John Gintell.

MULTICS PROGRAMMERS' MANUAL

```
 _____
|        |
| probe  |
|_____|
```

Command
07/16/74

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 giving 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 may 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 may 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 may be called and commands issued to it to identify the cause of the error.

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

Third, the user may 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 may be displayed. Probe may 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 maintains three "pointers" that can affect the execution of many commands. They are: the source pointer which marks a particular source program statement as the "current statement" and the program as the "current program"; the symbol pointer which indicates the "current block" and generation of storage (i.e. stack frame) in which to evaluate symbolic references to variables; and the control pointer which designates the statement at which control was suspended in the procedure of interest.

Usage:

    probe -<procedure>-

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

If a <procedure> is not specified, probe will check if an error or quit has occurred and, by default, use the procedure that was executing. The pointers will be set as if the user had specified it explicitly. If no error has occurred, then probe will print a message and return.

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 symbol pointer to the block that contains that statement.

In general, after an error, quit, or break, things will be 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 may 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 controlling when the request is to be performed. Examples follow in the next section.

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

Basic Commands

print, p

        print {<expression>¦<cross section>}

Output on the console the value of <expression>. The print
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.

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

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

        print array (1:5, 1)

which would print array(1,1), array(2,1), ..., array(5,1). More
than one dimension may 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 will be performed
according to the rules of PL/I. Array cross-sections may 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 will return to command level. After a break, the user's program will, in effect, be 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 giving the type of the arguments expected, the ones given will be converted to the expected type; otherwise, they will be passed as they are. The print request may 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, go to, go, 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

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

Source Commands

list, ls

list [n]

Direct 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 will be 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.

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

find, f

find "<string>"

Search for an executable statement containing the characters in <string> and if found, set the source pointer to that statement. 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.

find "write (6,10)"	— locate the statement in the program
find "str = ""a"	— locate str = "a
find "q+2"; list	— 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 will be 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 first three 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
will be assumed. The symbol 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 may be found with the help of the list 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 containing the
statement identified by the label or line number -- for instance,
the label on a begin statement denotes that begin block. If the
s block is not active, the source pointer will be set to
the statement specified. "level i" will use the ith block frame

from a stack trace. "-n" will use 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 will be used.)

use sub	— use block procedure sub occupies
use label	— use block containing label: ...
use level 2	— use second frame in stack trace
use -1	— use previous instance of current block
use -999	— use first (oldest) instance

symbol

        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 will be made to evaluate the size or extent expression; if the value is not available, then "*" will be used instead.

where, wh

        where [source!symbol!control]

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

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

Break Commands

insert, i

        insert [<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 will be assumed. When
the running program arrives at the statement, probe will be
entered before the statement is executed, and the commands will
be processed automatically.   When finished with the commands,
probe will return, and the program will resume at the statement
at which the break was set.   In effect, the user may "insert"
probe commands into his program.

        insert: (print var; print var2)
                                - set a break before the current
                                  statement
        insert quick: print x   - set a break before the statement
                                  labeled quick

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

append, a

        append [<label>]: (<command>¦(<command list>))

is the same as insert except that the break is set after the
statement designated.   This means that the command list will be
interpreted after the statement has been executed.   If the
statement branches to another location in the program, probe will
not be entered.   The difference between appending at one
statement and inserting at the next is that a transfer to the
next statement would cause a break for the insert case but would
not for the append one.

stop, sp

        stop

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 "stop" 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 may halt a running program.
A subsequent continue command would cause probe to resume what it

was doing before stopping — for instance, finish a break command
list and return to the program. The command:

        insert 29: stop

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

        append: (print a; stop; print b)

would cause 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 insert or append commands. Just "reset"
deletes   the   last break that occurred; the <label> form deletes
breaks set before and/or after a statement; <procedure> and "*"
may 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   inserted   and
                                      appended at 34
        reset after 34   — delete the break appended after
                                      34
        reset sub   — delete all breaks in sub
        reset *   — delete all breaks known

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 inserted 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 "stop; 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: that if a statement transfers elsewhere, the break will not happen until sometime later, if ever.

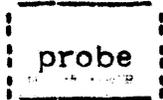## Miscellaneous Commands

brief, b

    brief [on:off]

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

execute, exec, ex

    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) will be executed if the <conditional> evaluates to true. The <conditional> may 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:
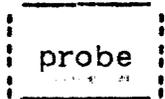
    insert: if z ^= "10"b: stop

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 may 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 may 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        123456700
```

Character and bit strings without repetition factors are allowed. Character strings may include newline characters. Octal strings may 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 may 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 may 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 may 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 that label_var is set to
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

Procedure References:

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

Evaluation of Variable References:

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

The block in which to look for a variable may be altered by the use command which sets the symbol pointer. For example, if "print var" displays the value of var in the current block, then "use -1; print 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 symbol pointer.

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

A block specification may 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 "list", 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 symbol pointer at the same time, the user can check if "s.num" has the correct value with the print 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 print 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 find command. "list" 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 prints 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
"inserts" a "stop". To accomplish the same thing, "insert 34:
stop" 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 prints 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.

```
 1          > pl1 test -table
 2          PL/I
 3          r 1248 3.211 28.336 280
 4
 5          > test
 6  (quit)
 7          QUIT
 8          r 1250 5.371 6.702 52   level 2, 10
 9
10          > probe
11          Condition quit raised at line 38 of fun.
12          > list
13                              return (2**b + fun (b, i));
14          > stack
15              1       command_processor_
16              2       release_stack
17              3       unclaimed_signal
18              4       real_sdh_
19              5       return_to_ring_0_
20              6       fun                         quit
21              7       test
22              8       command_processor_
23              9       listen_
24             10       process_overseer_
25             11       real_init_admin_
26          > use level 7
27          > list
28                      s.b(i) = fun (i, 1);
29          > print s.num
30              5
31          > find "i = 1"; list
32                  do i = 1 to s.num;
33          > append: print i
34          > continue
35          r 1252 1.375 16.394 354   level 2, 10
36
37          > release
38          r 1252 .126 .922 19
39
40          > test
41                  1
42                  1
43                  1
44                  1
45  (quit)
46          QUIT
47          r 1252 3.069 .650 25   level 2, 12
48
```

```
49        > release
50          r 1253 .092 .937 20
51
52        > probe test
53        > status test
54          Break after line 16.
55        > status after 16
56          Break after line 16:  print i
57        > reset at 16
58          Break reset after line 16 of test.
59        > position 34
60        > list
61                        if b = 0
62                            then return (1);
63        > insert: stop
64        > continue
65          r 1255 .781 12.356 333
66
67        > test
68          Stopped before line 34 of fun.
69        > print b
70              1
71        > where
72          Current line is line 34 of test.
73          Using level 2: fun.
74          Control at line 34 of fun.
75        > print i
76              1
77        > c
78          Stopped before line 34 of fun.
79        > stack 5
80              1       break
81              2       fun
82              3       fun
83              4       test
84              5       command_processor_
85        > print b
86              0
87        > print b[-1]
88              0
89        > print i
90              1
91        > symbol i
92          Attributes are: fixed binary(17,0) aligned parameter.
93          Declared in: fun.
94        > use test
95        > print i
96              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
```