Published: 6/7/66

Identification

Machine- and PL/I-oriented interrogation and modification of the contents of segments
Probe
D. B. Wagner

Purpose

Probe is an interactive program which allows the user to peek into and modify the segments of a process at any interruption of the process or after a normal termination. It reads its requests through a common interface as mentioned in BX.10.00, so that for example the if and do Requests facilitate the definition of macros which perform various kinds of searches.

Usage

The command

probe

causes probe to begin reading requests from the console. The user may type any of the requests listed below or any of the "control" requests (if, else, do, and end) described in BX.10.00. He may also type macro invocations (in the same * form as in the command language: see BX.1.01) which expand to sequences of these requests. If a line received by probe (after macro expansion) is not recognizable as a request, it is treated as a command. The line is given to the Shell, which gives an appropriate diagnostic if it is not a command either.

Interrogation Requests

One request, backed up by numerous special functions built into the expression-evaluating machinery, provides the basic interrogation facility. This is

print expression expression ...

The values of the expressions are printed on one line on the console, separated by tab characters. Each expression is normally an invocation of one of the built-in "format functions." A format function takes some argument and returns a character-string which "represents" the value of that argument according to some interpretation. (See the example below.)

The format functions are listed below. Operation of most is obvious. Each takes a PL/I scalar or an address and returns

a character string which is suitable as a representation of the value of the scalar according to some interpretation.

```
decimal
floating
octal
binary
ascii
instruction
indirect
symbaddr
```

Most of these are from GEBUG, and behave in essentially the same way as the corresponding GEBUG output formats. The instruction format function takes an address and produces a representation of the contents of that address which looks like a line from an assembly, i.e., something like "lda sym,4". If a symbol table is available for the segment involved the address is printed symbolically. (One of the lessons of FAPDBG and GEBUG is that this sort of symbolic instruction printing involves a great number of aesthetic problems which can be only partially solved. A number of known bugs exist in FAPDBG and GEBUG, such as the one which causes printing of "ALS 11" as "ALS SYMBOL-4763", but usefulness is only slightly impaired by such nonsense.) Indirect again takes an address and interprets its contents as an indirect word. An optional second argument specifies the type of indirect modifier the instruction referring to the indirect word would have (e.g. *, SC, CI, etc.--modifier mnemonics from the assembler). Symbaddr takes an address and produces a representation of the address in the form "segnam$expression", where the expression is in terms of whatever symbol tables may be available for the segment.

A macro (defined to perform the same operation as the GEBUG "peek" request), invoked to print the contents of locations 69 through 105 of segment alpha interpreted as decimal and octal, might produce the following sequence of requests:

```
do q=alpha$69 by 1 to alpha$105
print symbaddr(q) decimal(c(q)) octal(c(q))
end
```

A "search" macro invoked to search the same area for a word with bit 29 on might expand to the following:

```
do q=alpha$69 by 1 to alpha$105
if substr(c(q),29+1,1)="1"b then print symbaddr(q)
end
```

Modification Requests

A user may wish to alter the contents of his segments at an interruption of the execution of a program for either of two reasons: he may wish to correct a bug without retranslating (despite the fact that "patching" will be less desirable in Multics than it has been in other systems), or he may be trying to answer such a question as "What would be the effective address of this if index register 2 contained that?" or "Would this complicated PL/I conditional expression evaluate true if alpha were 25.7?"

To allow both of these uses, with emphasis on the second, the requests set, reset, revert, and fix are provided. The request

set variable=expression

saves the current value of the variable in a stack and alters the value to that of the expression. "The value of the variable" means, for a symbol used in an assembly, the address it represents (the value of the expression must be an address), and for an algebraic variable the contents of the storage region associated with the variable. To patch machine locations, the format

set c(address)=expression

is used, meaning set the contents of the location.

A patch normally remains in force only until probe is left, either by a normal return (exit request) or by a transfer into the program (transfer or proceed request), at which point the saved value is restored. The following requests modify the application of this rule:

fix variable
fix c(address)
reset variable
reset c(address)
revert variable
revert c(address)

Fix makes the patch permanent, so that there will be no automatic resetting of the value. Reset changes the value of the symbol or the contents of the location specified back to its value the last time probe was entered (the value at the bottom of the stack). Revert changes it to the next previous value saved by the set request (the value at the top of the previous-values stack).

Handling the Trace File

A trace file may be used to keep a history of a process. This is a ring file of size determined either by default or by specific declaration into which the snap request

described below stores snapshots of interesting storage regions. Then the forward and backward requests allow the user to roll machine conditions back and forth in time (loosely speaking: see the discussion below) so that the full power of probe may be used to peek at conditions at various times in the history of the process. The trace file will also see considerable use in conjunction with a "time control" attached to displays analogous to the Scheduling Algorithm Display in CTSS and for statistics-gathering.

Normally the following request will be used only through the tracer command.

snap region, region, ...

Each region specification takes either of the forms

variable
address to address

where the symbol is an algebraic variable and the expressions are addresses. This request causes a snapshot of the specified storage region to be placed into the trace file. The request

snapall

takes a snapshot of every impure (changeable) segment attached to the process. (Or rather, a snapshot of enough information so that the state of the process can be brought back to this instant in its history. Clearly there are problems concerning the parts of the process which are inaccessible to the user.) The amount of information stored can easily become rather gross and this request must be used with some care. The usefulness of this sort of all-inclusive snapshot, however, is clear: It makes possible a "time-machine" which allows the user to jump freely about in history and to experiment with changes in the properties of the primordial ooze from which a set of machine conditions sprang. The requests

forward
backward

roll machine conditions forward or backward in the trace file. That is, they read the next and previous snapshot respectively and set (see above) the contents of the corresponding storage regions accordingly. It is to be noted that this can lead to an inconsistent muddle if not enough information is saved in the trace file, e.g. if snap requests are used to do the saving instead of snapall. The trace file is a tool whose use is not always appropriate.

A "time-search" macro, invoked to find an instant in the history of the process when the variables a and b in some PL/I program were equal, might expand to the following sequence of requests (note that "¬n" is the escape for the negation sign):

```
do while a¬n=b
backward
end
```

Miscellaneous Requests

The request

proceed

causes program execution to be resumed at the point at which it was last interrupted, either by a quit or by a breakpoint or other call to the trace entry (see BX.10.02-03). This * involves a "synthetic epilogue," described in detail in EPL design journal #4 (B0005), which deactivates active blocks of the debugging programs which are above the program in the stack, as in a PL/I "non-local go to."

The request

transfer address

performs a synthetic epilogue and transfers control to the * location specified.

The request

exit

causes probe to return to its caller, normally the Shell.