Published: 6/7/66

Identification

Interactive debugging aids
D. B. Wagner

Purpose

The need for an "arsenal of new exterminators" for the "bugs" which have plagued programmers since the earliest days of computing has been thoroughly discussed elsewhere (e.g. see B0021). The collection of programs described here (probe, tracer, breaker, and monitor, BX.10.01-BX.10.04) form an interactive debugging aid which gathers into a very general framework most of the ideas in debugging which have been floating around in separate programs in different systems. This debugging aid is intended for interactive use but will certainly be usable by the batch-oriented user (simply read "control card" for "request" throughout).

A great deal of flexibility is provided through the use of the macro facility (described in BX.1.01) of the command language. One very important feature of this macro facility is that within a macro definition a mixture of commands (lines acted upon by the Shell) and requests (this is the most common word for lines acted upon by individual interactive programs) is possible. Users will not normally communicate directly with the debugging programs but use macros defined in terms of the "bare-bones" requests described in this and the following sections. A collection of "system macros" will be defined, documented, and made available so that the user will not have to know about the full generality of the debugging language unless he wishes to define macros himself.

Notice

A number of points in this and the following four Sections (BX.10.00-BX.10.04) are intentionally vague because at this writing certain parts of the System are not completely "nailed down." This is particularly true of the macro facility. Intentionally vague points are marked with "*" in the margin.

Debugging Facilities

Interrogation: At an interruption or normal termination of a program, the user may interrogate the values of variables and the contents of machine locations; a rather complete expression language makes it possible to conduct these interrogations in terms of the source language of the program. For example if a user, noting some peculiar program output, hits the quit button while a PL/I program is running and types the command

```
probe
```

followed by the request to probe,

```
print a+b
```

he means that probe is to find the storage assigned to the variables a and b in the program, add their values together in the same manner as a compiled PL/I program, and print the result on the console.

Breakpoints: A user may specify that program execution is to be interrupted upon the occurrence of certain (more or less hardware-oriented) events such as control reaching a certain point or a certain amount of time being used up. For example a standard macro named trap could be defined which makes arrangements so that the program will be interrupted when control reaches a certain point (label) in the program. (This example is enlarged upon in section BX.10.03) A user would then type

```
trap sym
```

to cause execution to be interrupted when control reached the statement labelled sym in his program. Then the user would start up his program (probably with a call through the Shell) and wait for the break to occur. When and if it did occur (i.e. when and if control reached sym), he would perhaps type print requests and snoop around in the values of variables at this point in the program's execution exactly as if he had just hit the quit button as discussed above. Finally he might allow execution to be continued, by typing

```
proceed
```

or cause execution to be resumed at some other point, by typing

```
transfer sym2
```

where sym2 is a statement label in the source program.

Tracing: Breakpoints may be used in another way. The tracer Command may be used to store up commands to be executed at specific breakpoints so that what takes place at the break is automatic. A macro named mvar might be defined which Causes the value of a variable to be printed every 10 milliseconds. (This macro would contain the command breaker, several requests to breaker, the command tracer, and again several requests. See the enlargement of this example in BX.10.03.) The user could then type,

```
                         mvar a+b
```

start his program by a call through the Shell, and receive the output

```
a+b            3.265
a+b            3.123
a+b            3.145
a+b            3.142
```

interspersed of course with any output his program produces.

Process History: One of the actions which can be specified to be performed at breakpoints is that of saving the state of a process so that it can be restored later. One may for example specify that the process state is to be saved every 10 ms. Then for example when and if something goes wrong in the program, probe requests can be used to back conditions up in time so that the user can search through time for clues to what went wrong in the program.

Limitations

The debugger is designed to be most convenient to users of PL/I and the standard assembly language. Users of algebraic languages other than PL/I, such as FORTRAN IV, will have to learn some new and occasionally confusing conventions, or else supply a replacement for the expression-evaluating machinery used by the debugging programs. Users of the languages sometimes unkindly called "oddball," such as COMIT, LISP, DYNAMO, ELIZA, and their ilk, will find the debugger as presently conceived less useful, although the trace and breakpoint facilities will probably see some use in connection with these languages. It seems unwise to build in any aids to users of specific special-purpose languages at this time since only an active user of LISP, for example, can have any clear idea of what facilities are useful in debugging LISP programs.

The Programs

Probe (described in BX.10.01) allows the user to examine and modify machine conditions and the contents of his segments using both machine- and PL/I-oriented formats. This is the core of any debugging aid. Considerable experience has been acquired in the matter of machine-oriented formats (e.g., in DDT, FAPDBG, FAPBUG, and GEBUG), but higher-language oriented formats are still in a rather primitive state.

Tracer (described in BX.10.02) provides a convenient tracing facility. In order to use it, the user inserts at strategic points in a program calls to a certain entry in the tracer command. Various ways of making these calls occur automatically at specific events will be available, e.g. the breaker and monitor commands and possibly a debug mode in the compiler. The tracer command accepts requests which specify "When argument 1 of the trace call is thus, do this." ("This" may be any sequence of commands and requests to commands.)

The breaker command (described in BX.10.03) accepts requests from a console or macro expansion to place a variety of event breakpoints into a program. It makes arrangements with the System to gain control whenever specified events occur. Breaker amounts to one way of causing trace calls to occur automatically.

The monitor command (described in BX.10.04) accepts requests from a console or macro expansion which indicate that certain blocks of machine code are to be executed interpretively instead of being allowed to run free. Whenever an "execution" access is made to such a block of code, a trap occurs and an interpreter is called. The interpreter calls the trace entry with appropriate arguments after the execution of each machine instruction.

The Debugging Language

An interactive program is an interpreter for a kind of computer language--an "interaction language" rather than a "programming language." The "debugging language" described here uses a number of the conventions of PL/I, e.g., the form of expressions and the control functions if, else, do, and end.

A request is a line which is read and acted upon by one of the programs probe, tracer, breaker, and monitor. (A better word might be primitive, since the requests which are actually seen by the programs will only rarely be typed by the user at his console. As was mentioned earlier, they will normally be used only in macro expansions.) A request consists in general of the request name followed by arguments delimited by blanks. The conventions of the Basic Command Syntax (see BX.1.00) are followed wherever applicable, especially with respect to the "Shell escape character" and the semicolon convention.

Expressions

An expression is something like "a+b" or "sin(a)+6" which can be evaluated to yield a value. Expressions are used in the debugging language in references to variables in the user's program and also wherever numbers, strings, etc. are arguments to requests (as in the specification of loops, see do request, below). Symbols used in these expressions are normally identifiers from the source program associated with the object program under examination. It is absolutely necessary that assemblers and compilers make available to the debugger the details of each compilation: this has * traditionally been done with the "symbol table file," a list of the identifiers defined by the programmer in the source program and an indication of "what was done" in implementing that identifier. (The standard format for these symbol tables is described in BD.2.)

A quick description of the debugging expression language would be that it is the PL/I expression language with the values of expressions limited to scalars (a PL/I expression may have a vector or structure value) but with the addition of the data type "address." (The data type "address" may turn out to be identical in implementation to the PL/I data type "pointer", but it seems worthwhile to keep the two concepts separate.) Expressions are divided into two classes, "machine-oriented expressions" and "algebraic expressions." The difference hangs primarily upon whether the "value" of a symbol referred to in the expression is taken to be the address (if any) associated with the symbol or the contents of the storage region (again, if any) associated with the symbol. The values of machine-oriented expressions are not constrained to be addresses, since a "contents" function is part of the language. This function takes an address and returns its contents in the form of a 36-bit bit-string which may then be used in any of the usual ways that bit-strings are used in PL/I expressions.

An algebraic expression is any valid PL/I scalar expression in which the variables referred to come from programs written in algebraic languages such as PL/I or FORTRAN IV. The value of a variable is taken to be the contents of the associated storage at the time expression evaluation takes place. If the variable is internal to a (PL/I) block which is not now active, the expression-evaluating machinery attempts to find its value at the last exit from the block. This information may or may not currently exist, depending for instance on the declaration of the variable (e.g. static or automatic) and the strategy used for dynamic storage allocation. The debugger attempts to find a symbol in any of the symbol tables it "knows about." A number of ambiguities present themselves: A name may be used for variables in different separately compiled programs or in different blocks of the same program, and one variable may have more than one generation active (e.g. when a recursive procedure calls itself). To provide a notation for "this symbol in this block," the question-mark (?) is used. For example "a?b" refers to the variable b in the block a. File or segment names may be used in the same way as block names. If a block has no name, its number (counted linearly through the source program) is used instead, so that "a?c?3?b" refers to the variable b in the third block internal to the block c internal to the block a. In the case of a "multiply active" variable (one for which more than one generation * exists), the latest generation (representing the deepest recursion) will arbitrarily be used.

A machine-oriented expression is an expression in which the "variables" are symbols from assembled source programs. Here symbols represent either addresses, base-offsets (such as stack symbols), or integers (symbols defined with some analog of the SET pseudo-operation in FAP). Expression syntax remains that of PL/I. In order to allow the expression of complicated Boolean conditions, such as those needed in the specification of searches for machine words with certain content or effective address, several special built-in functions are provided: the "content" function c, the "effective address" function ea, and the Boolean function safe which tells whether it is "safe" to use the effective-address function. This last is made necessary by the fact that in the 645 there are numerous funny kinds of indirection that do not yield proper addresses. The "contents of register" function cr recognizes mnemonics for special registers, so that for example "cr(a)" refers to the contents of the accumulator as a 36-bit string.

The treatment of the dollar-sign ($) in debugging expressions is slightly different from its treatment in PL/I. It is an operator whose preceding operand is a segment name, segment number, or base-register name and whose following operand is an integer giving relative address. The result is of course an address. Thus "alpha$7" means location 7 in the segment named alpha, but "6$7" means location 7 in segment number 6.

"Mixed" expressions, those which include both algebraic identifiers and machine-oriented identifiers, most emphatically do not have an official interpretation. These probably will not cause an error condition but will be interpreted in some reasonably intelligent manner, and may be useful in some contexts; nothing more will be said about these here.

The Control Requests

The four parts of the debugger will recognize, through a common interface, the control requests if, else, do, and end. The request

if condition then request

causes the request to be performed if and only if the conditional expression evaluates true. Then

else request

causes the specified request to be performed if and only if the conditional expression in the last balanced if request evaluated false. The request do (same options as in PL/I)

causes requests following, up to a balanced

end

to be executed under control of the options specified (options are loop-control specifications as in "do j=1 by 1 while a=b;").

In a <u>do</u> specification such as "do j=..." the variable specified is a "pseudo-variable" which is to be specially set up for the purpose. The variable is assigned a data-type consistent with that of the value of the expression to which it is being set, storage is assigned, and the variable name is placed in the symbol table. When the range of the <u>do</u> is left, the storage is freed and the name removed from the symbol table.

In addition to the above requests, each of the four parts of the debugger recognizes the request

exit

which means to return to the calling program, normally the Shell.