NAME

cdb – C debugger

SYNOPSIS

cdb [a.out [core]]

DESCRIPTION

Cdb is a debugger for use with C programs. It is useful for both post-mortem and interactive debugging. An important feature of *cdb* is that even in the interactive case no advance planning is necessary to use it; in particular it is not necessary to compile or load the program in any special way nor to include any special routines in the object file.

The first argument to *cdb* is an object program, preferably containing a symbol table; if not given ''a.out'' is used. The second argument is the name of a core-image file; if it is not given, ''core'' is used. The core file need not be present.

Commands to *cdb* consist of an address, followed by a single command character, possibly followed by a command modifier. Usually if no address is given the last-printed address is used. An address may be followed by a comma and a number, in which case the command applies to the appropriate number of successive addresses.

Addresses are expressions composed of names, decimal numbers, and octal numbers (which begin with ''0'') and separated by ''+'' and ''–''. Evaluation proceeds left-to-right.

Names of external variables are written just as they are in C. For various reasons the external names generated by C all begin with an underscore, which is automatically tacked on by *cdb*. Currently it is not possible to suppress this feature, so symbols (defined in assembly-language programs) which do not begin with underscore are inaccessible.

Variables local to a function (automatic, static, and arguments) are accessible by writing the name of the function, a colon '':'', and the name of the local variable (e.g. ''main:argc''). There is no notion of the ''current'' function; its name must always be written explicitly.

A number which begins with ''0'' is taken to be octal; otherwise numbers are decimal, just as in C. There is no provision for input of floating numbers.

The construction ''name[expression]'' assumes that *name* is a pointer to an integer and is equivalent to the contents of the named cell plus twice the expression. Notice that *name* has to be a genuine pointer and that arrays are not accessible in this way. This is a consequence of the fact that types of variables are not currently saved in the symbol table.

The command characters are:

/*m* print the addressed words. *m* indicates the mode of printout; specifying a mode sets the mode until it is explicitly changed again:
- **o** octal (default)
- **i** decimal
- **f** single-precision floating-point
- **d** double-precision floating-point

\ Print the specified bytes in octal.

= print the value of the addressed expression in octal.

´ print the addressed bytes as characters. Control and non-ASCII characters are escaped in octal.

" take the contents of the address as a pointer to a sequence of characters, and print the characters up to a null byte. Control and non-ASCII characters are escaped as octal.

& Try to interpret the contents of the address as a pointer, and print symbolically where the pointer points. The typeout contains the name of an external symbol and, if required, the smallest possible positive offset. Only external symbols are considered.

? Interpret the addressed location as a PDP-11 instruction.

$*m* If no *m* is given, print a stack trace of the terminated or stopped program. The last call made is listed first; the actual arguments to each routine are given in octal. (If this is inappropriate, the arguments may be examined by name in the desired format using ''/''.) If *m* is ''r'', the contents of the PDP-11 general registers are listed. If *m* is ''f'', the contents of the floating-point registers are listed. In all cases, the reason why the program stopped or terminated is indicated.

%*m* According to *m,* set or delete a breakpoint, or run or continue the program:

b An address within the program must be given; a breakpoint is set there. Ordinarily, breakpoints will be set on the entry points of functions, but any location is possible as long as it is the first word of an instruction. (Labels don't appear in the symbol table yet.) Stopping at the actual first instruction of a function is undesirable because to make symbolic printouts work, the function's save sequence has to be completed; therefore *cdb* automatically moves breakpoints at the start of functions down to the first real code.

It is impossible to set breakpoints on pure-procedure programs (−n flag on cc or ld) because the program text is write-protected.

d An address must be given; the breakpoint at that address is removed.

r Run the program being debugged. Following the ''%r'', arguments may be given; they cannot specify I/O redirection (''>'', ''<'') or filters. No address is permissible, and the program is restarted from scratch, not continued. Breakpoints should have been set if any were desired. The program will stop if any signal is generated, such as illegal instruction (including simulated floating point), bus error, or interrupt (see signal(II)); it will also stop when a breakpoint occurs and in any case announce the reason. Then a stack trace can be printed, named locations examined, etc.

c Continue after a breakpoint. It is possible but probably useless to continue after an error since there is no way to repair the cause of the error.

SEE ALSO

cc (I), db (I), C Reference Manual

BUGS

Use caution in believing values of register variables at the lowest levels of the call stack; the value of a register is found by looking at the place where it was supposed to have been saved by the callee.

Some things are still needed to make *cdb* uniformly better than *db:* non-C symbols, patching files, patching core images of programs being run. It would be desirable to have the types of variables around to make the correct style printout more automatic. Structure members should be available.

Naturally, there are all sorts of neat features not handled, like conditional breakpoints, optional stopping on certain signals (like illegal instructions, to allow breakpointing of simulated floating-point programs).