The Portable CLU Debugger

Dorothy Curtis

February 17, 1992

CLU programs compiled with portable CLU can be debugged by specifying the `-debug` option consistently on each portable CLU invocation, and then running the resulting executable. At that point, the debugger takes control and allows you to debug the program.

1 Basics

The debugger enables users to monitor execution of a program. Users can single step a program, executing one line at a time, or they can set *breakpoints* in the program. The debugger suspends execution of a program whenever it reaches a breakpoint, so that the user can determine how control reached that breakpoint and/or examine the values of program variables.

Whenever a procedure is invoked, information about the invocation is pushed onto a runtime *stack*. This information is known as a *stack frame*. By examining the stack when a breakpoint is reached, the user can determine how control reached the current procedure.

The debugger enables users to print the CLU objects referred to by program variables. Users can control how much of the representations of such objects is displayed. The printing *depth* is the maximum number of levels of nesting of an object that will be displayed. The printing *width* is the maximum number of components of an object that are displayed at each depth. The greater the width, the more elements in an array and the more fields in a record that are printed. The greater the depth, the more information about each element in an array or each field in a record that is displayed.

2 Hints

2.1 Interrupting the debugger

When a program is running, typing `control-c` (i.e., holding down the control or ctrl key and then pressing the letter `c`), will cause control to be passed back to the debugger. This assumes that `control-c` is your `INTR` (interrupt) character. (See *man (2) stty.*) For `control-c` to take effect, the program must pass through a procedure entry/exit point or a procedure with line breakpoints set in it (these lines do not need to be executed). A program in an infinite loop, which does not call a procedure, will not respond to `control-c`. To kill off such a program, type `control-z` (or your `SUSP` (suspend) character), use the `jobs` shell command to locate the errant program, and type

```
kill -9 %number_associated_with_program
```

You may need to use the `reset` shell command to restore normal tty behavior.

2.2 Capturing output

The (Unix) `script` command provides a method for capturing output created during a debugging session.

2.3 Line editing

The debugger supports a limited amount of line editing. The following files are used to specify key bindings for line editing: *~/.lineedit.keys* and *~/.inputrc*. These files override the default settings given by the following table.

Key	Action	Alternative key
ctrl-A	move to beginning of the line	ESC
ctrl-B	move back (left) one character	left-arrow on lk-201's
ctrl-D	delete current character	
ctrl-E	move to end of current line	
ctrl-F	move forward (right) one character	right-arrow on lk-201's
ctrl-J	complete entry	
ctrl-K	delete to the end of the line	remove-key on lk-201's
ctrl-M	complete entry	
ctrl-N	move to next history item	down-arrow on lk-201's
ctrl-P	move to previous history item	up-arrow on lk-201's
ctrl-U	delete line	
ctrl-W	delete word	
del	delete previous character	
	move forward one word	next-screen on lk-201's
	move back one word	prev-screen on lk-201's

2.4 Fatal errors

When running a program, a fatal error may occur and result in a Unix `Segmentation Violation` or `Bus Error`. When this happens, the debugger executes a `where` command to display the stack and then terminates. These errors are frequently caused by uninitialized variables.

If a program accesses uninitialized variables without triggering a fatal error, the state of the debugger may be corrupted.

If a circularly-linked object is printed, the debugger may terminate when the stack fills up.

2.5 How Printing Works

When printing an object of type t, the debugger uses `t$print`, if it exists and takes an object of type t (or cvt) as its first argument and a pstream as its second argument. If such an operation does not exist, `rep$print` for type t is used.

3 Command Overview

A debugger command consists of a command name followed by a sequence of arguments separated by spaces. The following describes related groups of debugger commands. The next section contains an alphabetical list of all debugger commands and describes them in detail.

- **Running**

  The `run` and `continue` commands cause the program to run continuously. The `next` command causes execution of a single program line.

- **Breakpoints**

  The `break` command sets breakpoints. The `show` command shows the current breakpoints. The `delete` command deletes breakpoints. The `step` command turns on single stepping for a procedure. The `unstep` command turns off single stepping for a procedure. The `trace` command turns on tracing for exceptions. The `untrace` command turns off tracing for exceptions.

- **The Stack**

  The `where` command displays the stack. The `up` and `down` commands select different frames in the stack.

- **Printing**

  The `print` command displays the values of variables. The `width` and `depth` commands control how much of the value is displayed.

- **Invocation**

  The `eval` command evaluates procedure invocations.

- **Source code**

  The `list` command displays source code. The `func` command selects a procedure to be displayed. Reaching a breakpoint also selects the procedure containing as the one to be displayed. Moving up and down the stack changes the procedure to be displayed.

- **Miscellaneous**

  The `help` command gives brief information on debugger commands. All commands can be displayed via `help all`. The `quit` command terminates the debugger.

4 Commands

This section describes each command in detail. Commands are listed alphabetically. Users can repeat execution of the `continue`, `eval`, `up`, `list`, `down`, `next` commands simply by pressing the `<RETURN>` key.

- `break` {*breakpoint ...*}

  The `break` command instructs the debugger to stop when a particular procedure or iterator is entered or exited or before a particular line in a file is executed. Each *breakpoint* is either the name of a procedure, the name of an iterator, or a line number, which refers to a line in the file containing the current procedure (or iterator). Note that the `func`, `up`, and `down` commands change the current procedure. Some examples:

```
break start_up         % sets breakpoints on entry to and
                       % exit from the top-most procedure


break 10               % sets a breakpoint at line 10
                       % in the current procedure


break 10 20            % sets breakpoints at lines 10 and 20
                       % in the current procedure


break t$op1 t$op2      % sets breakpoints on entry to and
                       % exit from procedures t$op1 and t$op2


break t$op1 t$op2 10   % sets breakpoints on entry to and
                       % exit from procedures t$op1 and t$op2
                       % and at line 10 in the current procedure
                       % (which may not be either t$op1 or t$op2)


func t$op1             % select function t$op1
break 10               % sets a breakpoint at line 10 in t$op1
```

  Whenever the debugger reaches a breakpoint, it displays the corresponding line in the source file and stops before executing that line.

- `continue`

  The `continue` command directs the debugger to continue running the program, either until the next breakpoint or until the next instruction in a procedure (or iterator) that is being single-stepped. It terminates any single-stepping set by the `next` command in the current procedure (or iterator).

- `delete` *breakpoint* {*breakpoint ...*}

  The `delete` command removes the specified breakpoints. (Note: If there are two breakpoints with the same line number, only the first will be deleted.) The argument `all` causes all breakpoints to be removed. Some examples:

```
delete t$op1                % deletes the breakpoints on entry to
                            % and exit from the procedure t$op1


delete all                  % deletes all breakpoints
```

- `depth` {*integer*}

  The `depth` command controls the printing depth for objects printed by the `print` command. If the `depth` command has no argument, the debugger prints the current depth. See also the `width` command.

- `down` {*integer*}

  The `down` command moves down the specified number of stack frames. If no argument is specified, then 1 is assumed. See also the `where` command and the `up` command.

- `eval` *expression*

  The `eval` command evaluates the specified expression. The following section describes the legal expressions. Some examples:

```
eval a = "xyz"                     % sets the variable a to "xyz"
eval po = stream$primary_output()  % sets po to primary output
eval stream$putl(po, a)            % prints a on po
```

  The assignment to the variable `a` in the example above changes the value of the variable `a` in the current stack frame, if such a variable exists. Otherwise, it creates a debugger variable named `a` and assigns the value of the expression to that variable.

- `func` {*name*}

  The `func` specifies the named procedure (or iterator) to be the one of current interest, either for listing source code or for setting breakpoints at line numbers. If no argument is given, the current line is printed, along with the five preceding and five succeeding lines

```
func t$op1                  % select function t$op1
list                        % list lines surrounding t$op1
break 10                    % set a breakpoint at line 10 in
                            % the file containing t$op1
```

- `help` {*command*}

  The `help` command, with no arguments, gives a summary of command names. When a command name is present as an argument, the `help` command will give a short description of the specified command. If the argument is the keyword `all`, short descriptions of all commands are displayed.

- `list` {*integer ...*}

  The `list` command displays source code in the current procedure (or iterator). By default, the current procedure is the routine that last reached a breakpoint. When the debugger first starts up, the `start_up` procedure is selected as the current procedure. The `func`, `up`, and `down` commands change the current procedure to the specified procedure. The arguments to `list` specify the line numbers of interest (use a space to separate the line numbers). The `where` command can be used to show the current line numbers in the currently active procedures.

```
list                        % list lines at current breakpoint
func t$op1                  % select function t$op1
list                        % list lines surrounding t$op1
```

- `next` {*integer*}

  The `next` command directs the debugger to single step to the next line in the current procedure. When a numeric argument n is present, the debugger executes the next n lines in the current procedure and prints out the corresponding source code as each line is executed.

- `print` *variable*

  The `print` command prints the values of variables, procedure arguments, and **own** variables. Uninitialized variables are printed as ???.

- `quit`

  The `quit` command causes the debugger to terminate.

- `rebuild`

  The `rebuild` command causes the debugger to run `make` and then the program.

- `restart`

  The `restart` command causes the debugger to restart the program: a new process is created, causing **own** variables to be initialized and the stack to be empty.

- `run`

  The `run` command causes the debugger to continue running the program.

- `show`

  The `show` command displays the location of the current breakpoints.

- `step` *name*

  The `step` command turns on single stepping for the specified procedure.

- `trace` *name*

  The `trace` command turns on tracing for the named signal.

  ```
  trace bounds                % trace the bounds exception
  trace all                   % trace all exceptions
  ```

- `unstep` *name*

  The `unstep` command turns off single stepping for the specified procedure.

- `trace` *name*

  The `untrace` command turns off tracing for the named signal.

  ```
  untrace bounds              % stop tracing the bounds exception
  untrace all                 % do not trace any exceptions
  ```

- `up` {`integer`}

  The `up` command moves up the specified number of stack frames. If no argument is specified, 1 is assumed. See also the `where` command and the `down` command.

- `where` {`integer`}

  The `where` command displays the frames on the stack. A numeric argument n limits the output to the last n frames pushed on the stack.

- `width` {`integer`}

  The `width` command controls the printing width for objects printed by the `print` command. If the `width` command has no argument, the debugger prints the current width. See also the `depth` command.

5 Expressions

The `eval` command currently accepts the following expressions:

- Constants:
  - booleans: `0` for false, `1` for true
  - characters: `'a'`
  - strings: `"abc"`
  - integers: `1`
  - reals: `1.2`
- Variables:

  The debugger first looks for a debugger variable with the given name, then for a local variable in the current procedure (or iterator), then for an **own** variable in the current procedure (or iterator) and finally for an **own** variable in the current type.

  The current version of the debugger does not always print the value of the variable that one expects: if two parallel scopes use the same name to denote variables of different types, the first declaration determines the print operation.
- Invocations:

  The programmer may invoke stand-alone procedures and procedures from unparameterized clusters. The current version of the debugger does not allow the user to invoke parametrized cluster procedures or iterators. Arguments to procedures can be constants or variables, as described above. The current debugger does not recognize sugars: users must type `eval int$add(1,2)` rather than `eval 1+2`. The debugger does not type-check arguments for procedure invocations; irrecoverable errors may result if arguments do not have the appropriate types.

6 Idiosyncrasies

6.1 Auxiliary files

When a program is executed under the debugger, an auxiliary file with symbol information is created. If the program is named *ps1*, then the file is named *ps1.sym*.

To preserve break point settings between debugging sessions, the debugger maintains the current breakpoints in a file. If the program is named *ps1*, then the file is named *ps1.bkpts*.

6.2 Stacks

When an iterator is active, the invoking procedure will appear on the stack twice: once for the invocation and once for the body of the **for** statement. Also, low-level routines that are in-lined may not appear on the stack, so setting breakpoints at such low-level routines may have no effect. Own initialization is not traceable and does not appear on the stack.