Vulcan Loupe: A Modula-$2^{+\epsilon}$ Debugger

John DeTreville
Michael Sclafani
DEC Systems Research Center

April 29, 1991

1 Invocation

To use the Modula-$2^{+\epsilon}$ symbolic debugger *Loupe*, you must first have a Vulcan server running. If the server isn't running, use the `vulcan` program to start one. You can start a server and a debugging session in one invocation; see the `vulcan(1)` manual page for details.

The `vulcan` program has two switches, `-core` and `-debug`, which cause it to contact a Vulcan server and start a debugging session. For each session, the server creates a new window containing a debugger typescript.

- `vulcan -core` starts a post-mortem debugging session on a core dump. It is invoked as:

  `vulcan -core` *[corefile]*

  where *corefile* is the optional name of the core file (by default, `core`).

  The `quit` command exits the core-file debugger; typing `^C` terminates its current action.

- `vulcan -debug` starts a teledebugging session on a Topaz address space, such as a running program, Taos, or the Nub. It is invoked as:

  `vulcan -debug` *pid [machine]*

  Pid names the process to be debugged; it is a process number or the special value `taos` or `nub`. *Machine* names the machine on which the process resides; it can be either the string `self` or an IP address of the form *w.x.y.z*. The default is `self`.

Contents

```
srcff 162> vulcan -debug 108
a new window appears
...debugger
  --- PC ---   --- SP ---   --- name ---
  07f886cfeH  07f77fcf0H  _TPSpecialStub + 017eH
  07f8834f0H  07f77fcfcH  Debugger.DefaultCallHandler
  07f883304H  07f77fd9cH  Debugger.Call
  07f884ad8H  07f77fdacH  VuException.Raise
  07f89daf5H  07f77fe94H  M2Base.Raise
  07f881d48H  07f77feacH  Trap.RaiseTKArithmetic
      04514H  07f77febcH  _$ROOT + 0437H
      05a76H  07f77ff00H  Test.Compute
"Unhandled exception"
raised Trap.TKArithmetic with Hardware.AEIntegerDBZ
thread 0824d3124H, help trap
"Unhandled exception"
stopped
<1> Loupe commands
```

Here, the program has stopped because of an unhandled arithmetic trap raised in the procedure `Test.Compute` or a procedure nested within it.

The `continue`, `resume`, and `stepping` commands return control to the program; the `quit` command exits the debugger.

When the debugger is executing, typing `^C` terminates its current action, but does not stop the program. When the debugger is first connected, it waits for the program to stop. Simple inspections of the program can be performed while it is running; use the `stop` command to stop it.

`vulcan -debug` normally uses the HighTTD server; it uses the LowTTD server when debugging the Nub.

If a file named `.louperc` exists in the user's home directory, the debugger executes the commands stored in it during initialization. See Section 2.10 for more information.

2 Commands

Loupe's command prompt is <*n*>, where *n* is a sequence number.

```
<1> print Module.x
5
<2> set Module.x := 6
<3> print Module.x
6
```

A command has a command-name followed by arguments. If a command line begins with a space, the command-name `print`, `set` or `call` is understood.

```
<4>  Module.x
6
<5>  Module.x := 7
<6>  Module.x
7
```

Otherwise, statements may appear on the same line separated by semicolons.

```
<7> print 1; print 2
1
2
```

Statements may extend over more than one line.

```
<8> print 1+
    1
2
```

The result of command *n*, if any, is named $*n*. The most recent result is named $.

```
<9> print $
2
<10> print $5
7
```

Similarly, !*n* re-executes command *n*, and ! re-executes the previous command.

```
<11> !7
1
2
```

The `help` command lists and describes all of the available commands.

- `help` *[name]*
  Describe all commands, or a single command if *name* is given.

2.1 Examining values

The `print` command displays values.

- `print` *[expression [format] [, ...]]*
  Print the value of a Modula-$2^{+\epsilon}$ expression. The optional *format* may be `decimal`, `hexadecimal`, `octal`, `binary`, `address`, `referent`, or `roman`, which may be abbreviated `dec`, `hex`, `oct`, `bin`, `adr`, `ref` or `rom`; the default format depends on the type of the expression. `follow` is also an alias for `referent`.

```
<12> print Module.integer
123456789
<13> print $ DIV 11
11223344
<14> print $ hex
0ab4130H
<15> print LOOPHOLE($13, WORD)
0ab4130H
```

Values printed in a numeric format will be displayed in the appropriate base. If you specify the format `address`, the values will be printed symbolically as long as they are *compatible* with `ADDRESS`. If you specify the format `referent`, Loupe will print the referent of a pointer rather than the value of the reference itself, or `NIL^` when the reference is `NIL`.

```
<16> print typeMapSubArray
[0 = NIL^,
 1 = POINTER TO <VuBase.TypeDefRec at <01c8f8H>>,
 2 = POINTER TO <VuBase.TypeDefRec at <01d3c8H>>]
<17> print $ ref
[0 = NIL^,
 1 = <type = 1, slot = 1, npr = 0, cleanUpQ = NIL^>,
 2 = <type = 2, slot = 2, npr = 0, cleanUpQ = NIL^>]
```

The `print` command may be abbreviated `pr`.

The command-name `print` may be omitted by beginning the command-line with a space.

```
<18>  Module.integer
123456789
```

To print two items on the same line of output, separate them with commas. To leave off the trailing newline, use a trailing comma on the command line.

The `prinwidth` command sets the width of the page.

- `prinwidth` *n*
  `prinwidth *`
  Set the maximum desired output line length. If `*` is given, lines are of unbounded length. The initial line length is `79`.

The `prindepth` command sets the level to which complex objects print.

- `prindepth` *n*
  Set the maximum print depth. Complex objects are elided below this depth. Arrays and `Text.T` values are truncated to a length based on the print depth. The initial maximum print depth is `6`.

The low-level `printf` command provides fixed-format printing.

- `printf` *format [,expression]...*
  Formatted print, as in Modula-$2^{+\epsilon}$'s `PRINTF`, with the addition of `%b` as a binary format.

```
<19> printf "There were %5d hits.\n", n
There were    10 hits.
```

The low-level `hexdump` command displays memory.

- `hexdump` *[address] [, count]*
  Dump a block of memory, formatted in hex and ASCII, that spans the *count* bytes at *address*. The default *count* is 256; the default *address* starts where the previous `hexdump` command left off.

The low-level `disassemble` command displays instructions.

- `disassemble` *[address] [, count]*
  Disassemble *count* instructions at *address*. The default *count* is 1; the default *address* starts where the previous `disassemble` command left off. The `disassemble` command may be abbreviated `da`.

The low-level `dumpprocedure` command displays a detailed description of a procedure.

- `dumpprocedure` *procedure*
  Dump information about a Modula-$2^{+\epsilon}$ procedure, including instruction disassembly, procedure header information, and variable liveness. *Procedure* should be the name of a procedure or an address within a procedure.

2.2 Changing values

The `set` command changes the value of a Modula-$2^{+\epsilon}$ variable. You cannot change values in a core file.

- `set` *target* `:=` *expression*
  Assign a variable. As an extension, the left-hand side may be an arbitrary expression that names memory, as in:

```
<20> set LOOPHOLE(Module.integer, BITSET)
         := BITSET{6..10}
<21> print $
BITSET{6..10}
```

  The command-name `set` may be omitted by beginning the command-line with a space.

```
<22>  Module.integer := 0
```

2.3 Calling procedures

Only built-in procedures and Loupe closures may be called. Procedures that return a value may appear in expressions, and so they may be called within a `print` statement. The `call` command invokes procedures regardless of whether they return a value.

- `call` *procedure* `(`*[args]*`)`
  Call a procedure.

  ```
  <23> print Module.integer
  0
  <24> call INC($, 5)
  <25> print Module.integer
  5
  ```

  The command-name `call` may be omitted by beginning the command-line with a space.

2.4 Examining the stack

The `showframe`, `showstack` and `frames` commands display the stack. There is a *current* stack frame; frames preceding it in time are *active*.

- `showframe` *[format]*
  Show the contents of the current stack frame, using *format* as for the `print` command. The `showframe` command may be abbreviated `sf`.

- `showstack` *[format]*
  Show the contents of all active stack frames (a verbose stack trace), using *format* as for the `print` command. The `showstack` command may be abbreviated `ss`.

- `frames`
  List all active stack frames (a terse stack trace). If `precise` is set, then the listing includes source locations of the form *index*+*length* (in characters), otherwise only a top-level procedure name is displayed. The `precise` command is described below.

The `previousframe`, `nextframe`, `topframe`, and `bottomframe` commands walk the stack.

- `previousframe` *[scope]*
  Move the current stack frame back by one: the current stack frame's caller becomes current. If *scope* (a module or procedure) is given, `previousframe` moves to the closest previous frame executing in that scope. The `previousframe` command may be abbreviated `pf`.

- `nextframe` *[scope]*
  Move the current stack frame forward by one: the current stack frame's callee becomes current. If *scope* is given, `nextframe` moves to the closest frame forward executing in that scope. The `nextframe` command may be abbreviated `nf`.

- `topframe`
  Return to the top of the stack, undoing any `previousframe` commands. The `topframe` command may be abbreviated `tf`.

- `bottomframe`
  Move to the bottom of the stack. The `bottomframe` command may be abbreviated `bf`.

On each entry, Loupe may perform automatic `previousframe` operations to back up over certain uninteresting stack frames; undo this action by invoking `topframe`.

The `precise` command sets the value of a flag which controls whether the address translations performed by the `frames` and `threads` commands find an exact source location, or only display a top-level procedure name. Finding the name of a top-level procedure is faster than computing the exact statement, but will hide nested procedure names as well as omitting source locations.

- `precise` *bool*
  Set the value of the precision flag. The initial value is `FALSE`.

2.5 Examining threads

The `threads` command lists the program's threads.

- `threads` *[scope][,n]*
  List threads. Each thread is identified by a unique handle and its current context. The top *n* frames in each thread are listed; the default *n* is 4. The listing is in the same format as the `frames` command, and the `precise` flag controls the detail of the listing. If *scope* (a module or procedure) is given, only those threads executing in that scope (within the top *n* frames) will be displayed. When teledebugging the Nub server, `threads` lists processors instead.

The `thread` command focuses attention on a current thread.

- `thread` *t*
  Select the thread named by *t*, which is either the `ADDRESS` value of the thread's `ThreadsPort.Handle` (as shown by the `threads` command) or the `Thread.T` value itself. The selected thread's stack becomes current. When teledebugging the Nub server, `thread` focuses on a processor instead.

2.6 Breakpoints

The `break` and `unbreak` commands set and clear breakpoints. The `breaks` command lists breakpoints. You cannot use these commands when debugging a core file.

- `break` *loc [*`when` *predicate]*
  Set a breakpoint at *loc*. If *loc*'s type is *compatible* with `ADDRESS`, a breakpoint is placed at that PC value. If *loc*'s type is a `PROCEDURE` type, a breakpoint is placed at the entry to the procedure. Any prior breakpoint at the location is cleared. If a *predicate* is given, the breakpoint will be taken only when the *predicate* is true.

  To set a breakpoint at a particular statement, use the overloading of subscripting described in Section 5 on Operators to obtain the address of the statement.

  ```
  <26> break Text[2094]
  <Text.Cat, loc 2065+44 (07f7b9b76H)>
  ```

- `unbreak` *loc*
  `unbreak *`
  Clear the breakpoint at *loc*. If `*` is given, clear all breakpoints (including Loupe's own internal breakpoints, as shown by `breaks`).

- `breaks`
  List all breakpoints.

2.7 Control

The `continue`, `stepin`, `stepup`, `stepdown`, `step` and `singlestep` commands return control to the program. You cannot use these commands when debugging a core file.

- `continue`
  Resume execution of the program. The `continue` command may be abbreviated `co`.

The `stepping` commands return control for some unit of execution. Each takes an optional trailing `until`-clause, specifying repetition of `stepping` until a *predicate* becomes true.

- `stepin` *[*`until` *predicate]*
  Resume execution of the program until the next statement in the current stack frame, or until the current stack frame returns. The `stepin` command may be abbreviated `si`.

- `stepup` *[*`until` *predicate]*
  Resume execution of the program until the current stack frame returns. The `stepup` command may be abbreviated su.

- `stepdown` *[*`until` *predicate]*
  Resume execution of the program until the top frame performs a call or a return. The `stepdown` command may be abbreviated `sd`.

- `step` *[*`until` *predicate]*
  Resume execution of the program until the next statement. The `step` command may be abbreviated `st`.

- `singlestep` *[*`until` *predicate]*
  Resume execution of the program for one machine instruction.

The `wait`, `stop`, and `resume` commands allow asynchronous control and inspection of the program. You cannot use these commands when debugging a core file.

- `wait`
  If the program is not stopped, wait for it to stop. Display the current state of the program.

- `stop`
  Issue a request to stop the program, wait for it to be stopped, and then display the current state of the program.

- `resume`
  Restart the program and immediately return, not waiting for the program to stop. Allows inspection of a running program. Use the `stop` or `wait` commands to resynchronize with the program.

2.8 Programmability

Loupe provides a number of simple programmability commands. The `lambda` command evaluates to a closure.

- `lambda` (*args*) *form* *[*end*]*
  Create a closure.

```
    <27> lambda (x) x - (x MOD 512) end
    <lambda (<<"x">>) <<<...>> (<...>, <...>)>>
    <28> print $(1984)
    1536
```

A *form* is a Modula-$2^{+\epsilon}$ expression. An expression is formed from a command sequence by enclosing it between "{" and "}". As a consequence of this syntax, an end is optional in the `lambda` command and the other programmability commands.

```
<29> lambda (x) {print x DIV 512; print x MOD 512}
<lambda (<<"x">>) <<<...>> (<...>, <...>)>>
<30> call $(1984)
3
448
```

The `value` command allows an expression to appear in a command context.

- `value` *expr*
  Evaluate an expression and return the result. This can be used to return a result from a *form*.

```
<31> value 1984
1984
<32> break Module.WatchPoint when
     {print x; value FALSE}
<Module.WatchPoint, loc 866+288 (05a80H)>
```

The `let` and `letrec` commands evaluate with `lambda`-bindings.

- `let` *name[(args)]* `=` *expr* ... `in` *form [*`end`*]*
  `let` *name* `:=` *expr* ... `in` *form [*`end`*]*
  `let` *name*`:` *type [*`:=` *expr]* ... `in` *form [*`end`*]*
  Evaluate *form* with the *names* bound to the *exprs*. In the `:=` variants, the values may be assigned to using the `set` command.

```
<33> let x = 123456789, y = 11 in x DIV y end
11223344
```

- `letrec` *name[(args)]* `=` *expr* ... `in` *form [*`end`*]*
  Evaluate *form* with the *names* bound to the *exprs*; the *exprs* are evaluated in the inner scope.

```
<34> letrec nn(i) =
     {print i; call nn(i+1)}
     in nn(0) end
0
1
2
3
4 ...
```

The `if` command provides conditional evaluation.

- `if` *expr* `then` *form [*`elsif` *form* `then` *form]* ... *[*`else` *form] [*`end`*]*
  If-command.

```
<35> letrec fact(n) =
     {if n <= 1 then 1 else n*fact(n-1) end}
     in fact(5) end
120
```

The `while`, `repeat`, `for`, `loop`, `exit`, `valof`, and `resultis` commands provide syntactic sugar for control, as in Modula-2$^{+\epsilon}$ and BCPL.

- `while` *expr* `do form` *[*`end`*]*
  Evaluate *expr*. If the result is `TRUE`, evaluate *form* and repeat.

- `repeat` *form* `until` *expr*
  Evaluate *form* and *expr*. If the result of *expr* is `TRUE`, repeat.

- `for` *name* `:=` *expr* `to` *expr* *[*`by` *expr]* `do` *form* *[*`end`*]*
  For-command. The *name* is `lambda`-bound within the *form.*

- `loop` *form* *[*`end`*]*
  Repeatedly evaluate *form* until a lexically enclosed `exit` is reached.

- `exit`
  End execution of a lexically enclosing `loop` command.

- `valof` *form* *[*`end`*]*
  The *form* is evaluated until a lexically enclosed `resultis` is reached.

- `resultis` *expr*
  Returns a value from an enclosing `valof`.

The `LET` command allows modification of `lambda`-bindings.

- `LET` *name[*`(`*args*`)`*]* `=` *expr ...*
  Replace a `lambda`-binding.

```
<36> let i = 0 in
     {lambda () {LET i = i+1; value i}}
<lambda () <<<...>> (<...>, <...>)>>
<37> print $36()
1
<38> print $36()
2
<39> print $36()
3
```

2.9 Definitions

The `define` and `DEFINE` commands define (top-level) names in Loupe.

- `define` *name[*`(`*args*`)`*]* `=` *expr ...*
  `define` *name* `:=` *expr ...*
  `define` *name*`:` *type [*`:=` *expr] ...*
  Globally bind names to values. In the `:=` variants, the value may be assigned to using the `set` command.

```
<40> define foo = Module.refOpenArray^
<41> print foo[5]
5
```

- `DEFINE` *name[(args)] = expr ...*
  `DEFINE` *name[: type] [:= expr] ...*
  Like `define`, but override any old bindings.

2.10 Customization

The `alias` and `unalias` commands create, display, and remove command-name aliases.

- `alias` *[name [expr]]*
  Display or create command aliases. If no arguments are given, all aliases are displayed. If *name* is given, the command or alias with that name is displayed. If *expr* is a simple name, then *name* will be created as an alias for *expr*. If *expr* is a *form*, then *name* will be created as a no-argument command alias for *form*. All of the command abbreviations are built-in name aliases.

```
<42> alias ss {singlestep; disassemble REG(15)}
<43> ss
<Text.Cat, loc 1807+977 (07f7b9997H)>:
stopped
----------- Text.Cat, loc 1807+977 (07f7b9997H):
07f7b9997H:    movl    ap,(sp)
```

- `unalias` *name*
  Removes the alias for *name*.

When a debugging session is started, the debugger checks for a file named `.louperc` in your home directory. If it exists, commands are read from that file and executed. It is an ideal place to put `alias` and `DEFINE` commands which customize the debugging environment.

2.11 Miscellaneous commands

Other Loupe commands include:

- `quit`
  Exit Loupe.

- `<` *"file"*
  Execute the Loupe commands in *file*.

- `>` *"file"*, *form*
  Evaluate *form*, storing the output in *file*.

- `ivy` *"file"*
  This experimental command communicates with Ivy by writing a magic control sequence which Ivy typescripts intercept. This command should cause Ivy to open *file*.

- `flushcache` *[address [, count]]*
  Invalidate data in the debugger's cache of target memory. This low-level command is useful if you suspect the target's memory has been altered by a third party. If no arguments are given, then the entire cache is flushed. Otherwise, cache entries for the *count* bytes starting at *address* are invalidated; the default *count* is `1`.

3 Names

By default, unqualified names are top-level names. These include module names, as well as built-in type names, the constants `TRUE`, `FALSE` and `NIL`, and so on. Names defined by `define` or `DEFINE` are top-level names. The top-level name "`@`" stands for the top-level scope itself.

`Module.x` names the `x` within the `Module` within the top-level scope. Within a module, names defined in the `DEFINITION MODULE` or the `IMPLEMENTATION MODULE` are treated identically.

If a stack frame is current, names visible in it also become visible at the top level. Names may be qualified with a procedure name, if that procedure has some active stack frame; the context of that procedure's first active stack frame is used.

The environment variables of the Vulcan server are also available in the top-level scope as `$`*name*.

```
<44> print $HOME
"/udir/sclafani"
```

A Loupe name (`$`, `$`*n*, a `lambda`-bound, `define`d or `DEFINE`d name) shares storage with any corresponding Modula-$2^{+\epsilon}$ value.

```
<45> print Module.integer
5
<46> let x = Module.integer in
     {set Module.integer := 0; value x}
0
<47> let x = Module.integer in
     {set x := 5; value Module.integer}
5
```

The characters `_` and `@` may appear freely in names; include arbitrary characters by preceding them with a \.

4 Types

All Modula-$2^{+\epsilon}$ language types are printable:

```
<48> print Module.record
<char = '!', boolean = TRUE,
 five = <plus = 5, minus = -5>>
<49> print Module.ptrBitSet
POINTER TO <BITSET at 06ce18H>
<50> print $^
BITSET{1..3}
```

Values of simple types print in the natural fashion (*e.g.*, integers print in decimal). ADDRESS and PROCEDURE values may print symbolically. WORD and BYTE values print in hexadecimal.

Arrays print between "[" and "]". If the element type is CHAR, the array prints as a string. Otherwise, the array prints as a list of "*index* = *value*" pairs.

POINTER values print as "POINTER TO <*typename* at *loc*>". REF values print as "REF <*typename* at *loc*>". REFANY values as print "*typename*: REF <*typename* at *loc*>".

Impossible values print in hexadecimal. For example, a POINTER to an obviously bad address will print in hexadecimal.

Loupe also models many values that are not first-class citizens in Modula-2$^{-\epsilon}$, and allows them to print.

```
<51> print Module.Short
BITS 16 FOR [0..65535]
<52> print BITSET
SET OF [0..31]
<53> print Module
MODULE Module IMPLEMENTS Module, ModulePrivate
```

5 Operators

All standard Modula-2$^{+\epsilon}$ operators are implemented: ".", "+", "-", "*", "/", DIV, MOD, "=", "#", "<", "<=", ">", ">=", IN, subscripting, unary "+", unary "-", AND, OR, NOT and "^". Some extensions have been provided for easier debugging.

- REFANY values may be dereferenced.
- Subscripting is overloaded: *scope*[*index*] computes the code address for the innermost statement within *scope* at character position *index* in the source file. Module[0] is defined to be the address of the main body of Module.
- The left-hand side of an assignment statement may be an arbitrary expression that names memory.
- If r is a reference to a record, then r.f is shorthand for r^.f
- The binary infix operator "&" performs string and text concatenation.

- Types may be compared for equality and inequality. The debugger cannot positively determine whether two opaque word types are not equal.

- Some type constructors may be used as expressions. Subranges, fixed-sized arrays, "`BITS FOR`", "`POINTER TO`", and "`SET OF`" are supported.

```
<54> print ['a'..'z']
['a'..'z']
<55> print ARRAY $ OF BITS 1 FOR BOOLEAN
ARRAY ['a'..'z'] OF BITS 1 FOR BOOLEAN
```

- An array may be subscripted by a subrange type, producing a subarray.

```
<56> print Module.charArray
[John DeTreville]
<57> print TYPEOF($)
Module.CharArray
<58> print Module.CharArray
ARRAY [1..15] OF CHAR
<59> print Module.charArray[[1..4]]
[John]
```

6 Procedures

Most Modula-$2^{+\epsilon}$ built-in procedures are implemented, including:

ABS	HIGH	MIN	ADDR	BITAND
ASSERT	INC	NARROW	BITSIZE	BITOR
CHR	INCL	NUMBER	BYTESIZE	BITXOR
DEC	LAST	ODD	COPY	BITSHIFTLEFT
EXCL	LONGFLOAT	ORD	TYPECODE	BITSHIFTRIGHT
FIRST	LOOPHOLE	TRUNC	ZERO	BITEXTRACT
FLOAT	MAX	VAL	BITNOT	BITINSERT

- The type-checking for some procedures is slightly looser than in the compiler. This should cause no problem.

- The built-in procedures `DISPOSE`, `HALT`, `NEW` and `RAISE` are not implemented.

Some new built-in procedures have been added for debugging.

- `TYPEOF(`*x*`)` returns the type of *x*.

- `LOCOF(`*x*`)` returns a description of *x*'s location.

- `AT(`*loc*`, `*type*`)` returns a value of type *type* at *loc*.

```
<60> print Module.x
123456789
<61> print TYPEOF(Module.x)
INTEGER
<62> print ADDR(Module.x)
<Module.x (02388cH)>
<63> print LOCOF(Module.x)
<the 32 bits at <loc 02388cH>>
<64> print AT($62, INTEGER)
123456789
<65> print AT($63, INTEGER)
123456789
```

Note the overloading of locations.

- `COPYOF(`*x*`)` returns a copy of *x* that does not share storage with *x*.
- `GETWORDAT(`*loc*`, `*n*`)` returns the *n*-bit word at *loc*; *n* must be `16` or `32`. The operation is atomic and uncached.
- `SETWORDAT(`*loc*`, `*n*`, `*value*`)` sets the *n*-bit word at *loc* to *value*; *n* must be 16 or 32. The operation is atomic and uncached.
- `REG(`*n*`)` returns the value of register *n* in the current stack frame. Register `16` is the VAX PSL. The value often shares "storage" with the register.
- `ACCESSIBLE(`*x*`)` returns `FALSE` if *x* is dead or has an inaccessible location.
- `BITALIGN(`*x*`)` returns the bit alignment of *x* or values of type *x*.
- `MINBITSIZE(`*x*`)` returns the minimum number of bits needed to store *x* or values of type *x*.
- `NEXTTHREAD(`*handle [,scope [,n]]*`)` returns the handle of the next thread executing in *scope* (in the top *n* frames). If *n* is omitted, then the entire thread stack is searched, and if *scope* is omitted, then the next thread is returned. `NIL` starts and ends the iteration.
- `REFTYPE(`*n*`)` returns the concrete `REF` type that has typecode *n*.
- `TIMENOWSECONDS()` returns `Time.NowSeconds()`.
- `VESTAUID(`*x*`)` returns the Vesta UID of the source of the module defined by *x* or containing *x*, where *x* can be a module, an interface, a procedure, or an address.

7 Problems

The following restrictions apply.

- Calling user procedures (procedure call in target) is currently unimplemented.
- Loupe values that share storage with values in the target program continue to exist even when the target values disappear.
- `Text.T` literals are not implemented.
- Loupe does not adequately understand exceptions.
- The debugger cannot positively determine whether two opaque word types are not equal.
- The `for` command does not support `UNSIGNED` iteration.
- Vulcan Loupe does not support debugging code written in other languages, such as C and Pascal. For example, if the Vulcan procedure calling sequence is not followed, then a stack trace may not be available while executing such "foreign" code.

Index