Acid Reference Manual

Phil Winterbottom
philw@plan9.bell-labs.com

Introduction

Acid is a general purpose, source level symbolic debugger. The debugger is built around a simple command language. The command language, distinct from the language of the program being debugged, provides a flexible user interface that allows the debugger interface to be customized for a specific application or architecture. Moreover, it provides an opportunity to write test and verification code independently of a program's source code. Acid is able to debug multiple processes provided they share a common set of symbols, such as the processes in a threaded program.

Like other language-based solutions, Acid presents a poor user interface but provides a powerful debugging tool. Application of Acid to hard problems is best approached by writing functions off-line (perhaps loading them with the `include` function or using the support provided by *acme*(1)), rather than by trying to type intricate Acid operations at the interactive prompt.

Acid allows the execution of a program to be controlled by operating on its state while it is stopped and by monitoring and controlling its execution when it is running. Each program action that causes a change of execution state is reflected by the execution of an Acid function, which may be user defined. A library of default functions provides the functionality of a normal debugger.

On Plan 9, a process is controlled by writing messages to a control file in the *proc*(3) file system. Each control message has a corresponding Acid function, which sends the message to the process. These functions take a process id *pid*) as an argument. The memory and text file of the program may be manipulated using the indirection operators. The symbol table, including source cross reference, is available to an Acid program. The combination allows complex operations to be performed both in terms of control flow and data manipulation.

Input format and `whatis`

Comments start with `//` and continue to the end of the line. Input is a series of statements and expressions separated by semicolons. At the top level of the interpreter, the builtin function `print` is called automatically to display the result of all expressions except function calls. A unary `+` may be used as a shorthand to force the result of a function call to be printed.

Also at the top level, newlines are treated as semicolons by the parser, so semicolons are unnecessary when evaluating expressions.

When Acid starts, it loads the default program modules, enters interactive mode, and prints a prompt. In this state Acid accepts either function definitions or statements to be evaluated. In this interactive mode statements are evaluated immediately, while function definitions are stored for later invocation.

The `whatis` operator can be used to report the state of identifiers known to the interpreter. With no argument, `whatis` reports the name of all defined Acid functions; when supplied with an identifier as an argument it reports any variable, function, or type definition associated with the identifier. Because of the way the interpreter handles semicolons, the result of a `whatis` statement can be returned directly to Acid without adding semicolons. A syntax error or interrupt returns Acid to the normal evaluation mode; any partially evaluated definitions are lost.

Using the Library Functions

After loading the program binary, Acid loads the portable and architecture-specific library functions that form the standard debugging environment. These files are Acid source code and are human-readable. The following example uses the standard debugging library to show how language and program interact:

Previously appeared with minor differences as the ''Acid Manual'' in *Plan 9 Programmer's Manual, Volume 2 (Second Edition)*.

```
% acid /bin/ls
/bin/ls:mips plan 9 executable

/sys/lib/acid/port
/sys/lib/acid/mips
acid: new()
75721: system call  _main ADD  $-0x14,R29
75721: breakpoint   main+0x4   MOVW  R31,0x0(R29)
acid: bpset(ls)
acid: cont()
75721: breakpoint   ls    ADD  $-0x16c8,R29
acid: stk()
At pc:0x0000141c:ls /sys/src/cmd/ls.c:87
ls(s=0x0000004d,multi=0x00000000) /sys/src/cmd/ls.c:87
    called from main+0xf4 /sys/src/cmd/ls.c:79
main(argc=0x00000000,argv=0x7ffffff0) /sys/src/cmd/ls.c:48
    called from _main+0x20 /sys/src/libc/mips/main9.s:10
acid: PC
0xc0000f60
acid: *PC
0x0000141c
acid: ls
0x0000141c
```

The function `new()` creates a new process and stops it at the first instruction. This change in state is reported by a call to the Acid function `stopped`, which is called by the interpreter whenever the debugged program stops. `Stopped` prints the status line giving the pid, the reason the program stopped and the address and instruction at the current PC. The function `bpset` makes an entry in the breakpoint table and plants a breakpoint in memory. The `cont` function continues the process, allowing it to run until some condition causes it to stop. In this case the program hits the breakpoint placed on the function `ls` in the C program. Once again the `stopped` routine is called to print the status of the program. The function `stk` prints a C stack trace of the current process. It is implemented using a builtin Acid function that returns the stack trace as a list; the code that formats the information is all written in Acid. The Acid variable `PC` holds the address of the cell where the current value of the processor register `PC` is stored. By indirecting through the value of `PC` the address where the program is stopped can be found. All of the processor registers are available by the same mechanism.

Types

An Acid variable has one of four types: *integer*, *float*, *list*, or *string*. The type of a variable is inferred from the type of the right-hand side of the assignment expression which last set its value. Referencing a variable that has not yet been assigned draws a "used but not set" error. Many of the operators may be applied to more than one type; for these operators the action of the operator is determined by the types of its operands. The action of each operator is defined in the *Expressions* section of this manual.

Variables

Acid has three kinds of variables: variables defined by the symbol table of the debugged program, variables that are defined and maintained by the interpreter as the debugged program changes state, and variables defined and used by Acid programs.

Some examples of variables maintained by the interpreter are the register pointers listed by name in the Acid list variable `registers`, and the symbol table listed by name and contents in the Acid variable `symbols`.

The variable `pid` is updated by the interpreter to select the most recently created process or the process selected by the `setproc` builtin function.

Formats

In addition to a type, variables have formats. The format is a code letter that determines the printing style and the effect of some of the operators on that variable. The format codes are derived from the format letters used by *db*(1). By default, symbol table variables and numeric constants are assigned the format code `X`, which specifies 32-bit hexadecimal. Printing a variable with this code yields the output `0x00123456`.

The format code of a variable may be changed from the default by using the builtin function `fmt`. This function takes two arguments, an expression and a format code. After the expression is evaluated the new format code is attached to the result and forms the return value from `fmt`. The backslash operator is a short form of `fmt`. The format supplied by the backslash operator must be the format character rather than an expression. If the result is assigned to a variable the new format code is maintained in the variable. For example:

```
acid: x=10
acid: print(x)
0x0000000a
acid: x = fmt(x, ’D’)
acid: print(x, fmt(x, ’X’))
10 0x0000000a
acid: x
10
acid: x\o
12
```

The supported format characters are:

`o`	Print two- byteinteger in octal.
`O`	Print four- byteinteger in octal.
`q`	Print two- byteinteger in signed octal.
`Q`	Print four- byteinteger in signed octal.
`B`	Print four- byteinteger in binary.
`d`	Print two- byteinteger in signed decimal.
`D`	Print four- byteinteger in signed decimal.
`Y`	Print eight- byteinteger in signed decimal.
`x`	Print two- byteinteger in hexadecimal.
`X`	Print four- byteinteger in hexadecimal.
`u`	Print two- byteinteger in unsigned decimal.
`U`	Print four- byteinteger in unsigned decimal.
`Z`	Print eight- byteinteger in unsigned decimal.
`f`	Print single- precisionfloating point number.
`F`	Print double- precisionfloating point number.
`g`	Print a single precision floating point number in string format.
`G`	Print a double precision floating point number in string format.
`b`	Print byte in hexadecimal.
`c`	Print byte as an ASCII character.
`C`	Like `c`, with printable ASCII characters represented normally and others printed in the form `\x`*nn*.
`s`	Interpret the addressed bytes as UTF characters and print successive characters until a zero byte is reached.
`r`	Print a two- byteinteger as a rune.
`R`	Print successive two- byteintegers as runes until a zero rune is reached.
`Y`	Print successive eight- byteintegers in hexadecimal.
`i`	Print as machine instructions.
`I`	As `i` above, but print the machine instructions in an alternate form if possible: `sunsparc` and `mipsco` reproduce the manufacturers’ syntax.
`a`	Print the value in symbolic form.

Complex types

Acid permits the definition of the layout of memory. The usual method is to use the `-a` flag of the compilers to produce Acid- languagedescriptions of data structures (see *2c*(1)) although such definitions can be typed interactively. The keywords `complex`, `adt`, `aggr`, and `union` are all equivalent; the compiler uses the synonyms to document the declarations. A complex type is described as a set of members, each containing a format letter, an offset in the structure, and a name. For example, the C structure

```
struct List {
    int         type;
    struct List *next;
};
```

is described by the Acid statement

```
complex List {
    'D' 0   type;
    'X' 4   next;
};
```

Scope

Variables are global unless they are either parameters to functions or are declared as `local` in a function body. Parameters and local variables are available only in the body of the function in which they are instantiated. Variables are dynamically bound: if a function declares a local variable with the same name as a global variable, the global variable will be hidden whenever the function is executing. For example, if a function `f` has a local called `main`, any function called below `f` will see the local version of `main`, not the external symbol.

Addressing

Since the symbol table specifies addresses, to access the value of program variables an extra level of indirection is required relative to the source code. For consistency, the registers are maintained as pointers as well; Acid variables with the names of processor registers point to cells holding the saved registers.

The location in a file or memory image associated with an address is calculated from a map associated with the file. Each map contains one or more quadruples (t, b, e, f), defining a segment named t (usually `text`, `data`, `regs`, or `fpregs`) mapping addresses in the range b through e to the part of the file beginning at offset f. The memory model of a Plan 9 process assumes that segments are disjoint. There can be more than one segment of a given type (e.g., a process may have more than one text segment) but segments may not overlap. An address a is translated to a file address by finding a segment for which $b + a < e$; the location in the file is then $address + f - b$.

Usually, the text and initialized data of a program are mapped by segments called `text` and `data`. Since a program file does not contain bss, stack, or register data, these data are not mapped by the data segment. The text segment is mapped similarly in the memory image of a normal (i.e., non- kernel)process. However, the segment called `*data` maps memory from the beginning to the end of the program's data space. This region contains the program's static data, the bss, the heap and the stack. A segment called `*regs` maps the registers; `*fpregs` maps the floating point registers (if they exist).

Sometimes it is useful to define a map with a single segment mapping the region from 0 to 0xFFFFFFFF; such a map allows the entire file to be examined without address translation. The builtin function `map` examines and modifies Acid's map for a process.

Name Conflicts

Name conflicts between keywords in the Acid language, symbols in the program, and previously defined functions are resolved when the interpreter starts up. Each name is made unique by prefixing enough `$` characters to the front of the name to make it unique. Acid reports a list of each name change at startup. The report looks like this:

```
/bin/sam: mips plan 9 executable
/lib/acid/port
/lib/acid/mips
Symbol renames:
    append=$append T/0xa4e40
acid:
```

The symbol `append` is both a keyword and a text symbol in the program. The message reports that the text symbol is now named `$append`.

Expressions

Operators have the same binding and precedence as in C. For operators of equal precedence, expressions are evaluated from left to right.

Boolean expressions

If an expression is evaluated for a boolean condition the test performed depends on the type of the result. If the result is of *integer* or *floating* type the result is true if the value is non- zero.If the expression is a *list* the result is true if there are any members in the list. If the expression is a *string* the result is true if there are any characters in the string.

primary-expression:
  identifier
  identifier `:` *identifier*
  constant
  `(` *expression* `)`
  `{` *elist* `}`

elist:
  expression
  elist , expression

An identifier may be any legal Acid variable. The colon operator returns the address of parameters or local variables in the current stack of a program. For example:

```
*main:argc
```

prints the number of arguments passed into main. Local variables and parameters can only be referenced after the frame has been established. It may be necessary to step a program over the first few instructions of a breakpointed function to properly set the frame.

Constants follow the same lexical rules as C. A list of expressions delimited by braces forms a list constructor. A new list is produced by evaluating each expression when the constructor is executed. The empty list is formed from `{}`.

```
acid: x = 10
acid: l = { 1, x, 2\D }
acid: x = 20
acid: l
{0x00000001 , 0x0000000a , 2 }
```

Lists

Several operators manipulate lists.

list-expression:
  primary-expression
  `head` *primary-expression*
  `tail` *primary-expression*
  `append` *expression* `,` *primary-expression*
  `delete` *expression* `,` *primary-expression*

The *primary-expression* for `head` and `tail` must yield a value of type *list*. If there are no elements in the list the value of `head` or `tail` will be the empty list. Otherwise `head` evaluates to the first element of the list and `tail` evaluates to the rest.

```
acid: head {}
{}
acid: head {1, 2, 3, 4}
0x00000001
acid: tail {1, 2, 3, 4}
{0x00000002 , 0x00000003 , 0x00000004 }
```

The first operand of `append` and `delete` must be an expression that yields a *list*. `Append` places the result of evaluating *primary- expression* at the end of the list. The *primary- expression* supplied to `delete` must evaluate to an integer; `delete` removes the *n*'th item from the list, where *n* is integral value of *primary- expression.* List indices are zero- based.

```
acid: append {1, 2}, 3
{0x00000001 , 0x00000002 , 0x00000003 }
acid: delete {1, 2, 3}, 1
{0x00000001 , 0x00000003 }
```

Assigning a list to a variable copies a reference to the list; if a list variable is copied it still points at the same list. To copy a list, the elements must be copied piecewise using `head` and `append`.

Operators

postfix- expression:
  list- expression
  postfix- expression `[` *expression* `]`
  postfix- expression `(` *argument- list* `)`
  postfix- expression `.` *tag*
  postfix- expression `->` *tag*
  postfix- expression`++`
  postfix- expression`--`

argument- list:
  expression
  argument- list, expression

The `[` *expression* `]` operator performs indexing. The indexing expression must result in an expression of *integer* type, say *n*. The operation depends on the type of *postfix- expression*. If the *postfix- expression* yields an *integer* it is assumed to be the base address of an array in the memory image. The index offsets into this array; the size of the array members is determined by the format associated with the *postfix- expression*. If the *postfix- expression* yields a *string* the index operator fetches the *n*'th character of the string. If the index points beyond the end of the string, a zero is returned. If the *postfix- expression* yields a *list* then the indexing operation returns the *n*'th item of the list. If the list contains less than *n* items the empty list `{}` is returned.

The `++` and `--` operators increment and decrement integer variables. The amount of increment or decrement depends on the format code. These postfix operators return the value of the variable before the increment or decrement has taken place.

unary- expression:
  postfix- expression
  `++` *unary- expression*
  `--` *unary- expression*

unary- operator:one of
  `* @ + - ~ !`

The operators `*` and `@` are the indirection operators. `@` references a value from the text file of the program being debugged. The size of the value depends on the format code. The `*` operator fetches a value from the memory image of a process. If either operator appears on the left- hand side of an assignment statement, either the file or memory will be written. The file can only be modified when Acid is invoked with the `-w` option. The prefix `++` and `--` operators perform the same operation as their postfix counterparts but return the value after the increment or decrement has been performed. Since the `++` and `*` operators fetch and increment the correct amount for the specified format, the following function prints correct machine instructions on a machine with variable length instructions, such as the 68020 or 386:

```
defn asm(addr)
{
    addr = fmt(addr, 'i');
    loop 1, 10 do
        print(*addr++, "\n");
}
```

The operators ~ and ! perform bitwise and logical negation respectively. Their operands must be of *integer* type.

> *cast- expression:*
>   *unary- expression*
>   *unary- expression*\ *format- char*
>   (*complex- name*) *unary- expression*

A unary expression may be preceded by a cast. The cast has the effect of associating the value of *unary-expression* with a complex type structure. The result may then be dereferenced using the . and -> operators.

An Acid variable may be associated with a complex type to enable accessing the type's members:

```
acid: complex List {
    'D' 0   type;
    'X' 4   next;
};
acid: complex List lhead
acid: lhead.type
10
acid: lhead = ((List)lhead).next
acid: lhead.type
-46
```

Note that the `next` field cannot be given a complex type automatically.

When entered at the top level of the interpreter, an expression of complex type is treated specially. If the type is called `T` and an Acid function also called `T` exists, then that function will be called with the expression as its argument. The compiler options `-a` and `-aa` will generate Acid source code defining such complex types and functions; see *2c*(1).

A *unary- expression* may be qualified with a format specifier using the \ operator. This has the same effect as passing the expression to the `fmt` builtin function.

> *multiplicative- expression:*
>   *cast- expression*
>   *multiplicative- expression** *multiplicative- expression*
>   *multiplicative- expression*/ *multiplicative- expression*
>   *multiplicative- expression*% *multiplicative- expression*

These operate on *integer* and *float* types and perform the expected operations: * multiplication, / division, % modulus.

> *additive- expression:*
>   *multiplicative- expression*
>   *additive- expression*+ *multiplicative- expression*
>   *additive- expression*- *multiplicative- expression*

These operators perform as expected for *integer* and *float* operands. Unlike in C, + and - do not scale the addition based on the format of the expression. This means that `i=i+1` will always add 1 but `i++` will add the size corresponding to the format stored with `i`. If both operands are of either *string* or *list* type then addition is defined as concatenation. Subtraction is undefined for these two types.

> *shift- expression:*
>   *additive- expression*
>   *shift- expression*<< *additive- expression*
>   *shift- expression*>> *additive- expression*

The >> and << operators perform bitwise right and left shifts respectively. Both require operands of *integer* type.

> *relational- expression:*
> *relational- expression*< *shift- expression*
> *relational- expression*> *shift- expression*
> *relational- expression*<= *shift- expression*
> *relational- expression*>= *shift- expression*

> *equality- expression:*
> *relational- expression*
> *relational- expression*== *equality- expression*
> *relational- expression* != *equality- expression*

The comparison operators are < (less than), > (greater than), <= (less than or equal to), >= (greater than or equal to), == (equal to) and != (not equal to). The result of a comparison is 0 if the condition is false, otherwise 1. The relational operators can only be applied to operands of *integer* and *float* type. The equality operators apply to all types. Comparing mixed types is legal. Mixed integer and float compare on the integral value. Other mixtures are always unequal. Two lists are equal if they have the same number of members and a pairwise comparison of the members results in equality.

> *AND- expression:*
> *equality- expression*
> *AND- expression*& *equality- expression*

> *XOR- expression:*
> *AND- expression*
> *XOR- expression*^ *AND- expression*

> *OR- expression:*
> *XOR- expression*
> *OR- expression* | *XOR- expression*

These operators perform bitwise logical operations and apply only to the *integer* type. The operators are & (logical and), ^ (exclusive or) and | (inclusive or).

> *logical- AND- expression:*
> *OR- expression*
> *logical- AND- expression*&& *OR- expression*

> *logical- OR- expression:*
> *logical- AND- expression*
> *logical- OR- expression*|| *logical- AND- expression*

The && operator returns 1 if both of its operands evaluate to boolean true, otherwise 0. The || operator returns 1 if either of its operands evaluates to boolean true, otherwise 0.

Statements

> `if` *expression* `then` *statement* `else` *statement*
> `if` *expression* `then` *statement*

The *expression* is evaluated as a boolean. If its value is true the statement after the `then` is executed, otherwise the statement after the `else` is executed. The `else` portion may be omitted.

> `while` *expression* `do` *statement*

In a while loop, the *statement* is executed while the boolean *expression* evaluates true.

> `loop` *startexpr, endexpr* `do` *statement*

The two expressions *startexpr* and *endexpr* are evaluated prior to loop entry. *Statement* is evaluated while the value of *startexpr* is less than or equal to *endexpr*. Both expressions must yield *integer* values. The value of *startexpr* is incremented by one for each loop iteration. Note that there is no explicit loop variable; the *expressions* are just values.

`return` *expression*

`return` terminates execution of the current function and returns to its caller. The value of the function is given by expression. Since `return` requires an argument, nil- valuedfunctions should return the empty list `{}`.

`local` *variable*

The `local` statement creates a local instance of *variable*, which exists for the duration of the instance of the function in which it is declared. Binding is dynamic: the local variable, rather than the previous value of *variable*, is visible to called functions. After a return from the current function the previous value of *variable* is restored.

If Acid is interrupted, the values of all local variables are lost, as if the function returned.

`defn` *function- name*`(` *parameter- list*`)` *body*

parameter- list:
variable
parameter- list, variable

body:
`{` *statement* `}`

Functions are introduced by the `defn` statement. The definition of parameter names suppresses any variables of the same name until the function returns. The body of a function is a list of statements enclosed by braces.

Code variables

Acid permits the delayed evaluation of a parameter to a function. The parameter may then be evaluated at any time with the `eval` operator. Such parameters are called *code variables* and are defined by prefixing their name with an asterisk in their declaration.

For example, this function wraps up an expression for later evaluation:

```
acid: defn code(*e) { return e; }
acid: x = code(v+atoi("100")\D)
acid: print(x)
(v+atoi("100"))\D;
acid: eval x
<stdin>:5: (error) v used but not set
acid: v=5
acid: eval x
105
```

Source Code Management

Acid provides the means to examine source code. Source code is represented by lists of strings. Builtin functions provide mapping from address to lines and vice- versa.The default debugging environment has the means to load and display source files.

Builtin Functions

The Acid interpreter has a number of builtin functions, which cannot be redefined. These functions perform machine- or operating system- specificfunctions such as symbol table and process management. The following section presents a description of each builtin function. The notation `{}` is used to denote the empty list, which is the default value of a function that does not execute a `return` statement. The type and number of parameters for each function are specified in the description; where a parameter can be of any type it is specified as type *item*.

integer `access(`*string*`)` Check if a file can be read

`Access` returns the integer 1 if the file name in *string* can be read by the builtin functions `file`, `readfile`, or `include`, otherwise 0. A typical use of this function is to follow a search path looking for a source file; it is used by `findsrc`.

```
if access("main.c") then
   return file("main.c");
```

float `atof(`*string*`)` Convert a string to float

`atof` converts the string supplied as its argument into a floating point number. The function accepts strings in the same format as the C function of the same name. The value returned has the format code `f`. `atof` returns the value 0.0 if it is unable to perform the conversion.

```
acid: +atof("10.4e6")
1.04e+07
```

integer `atoi(`*string*`)` Convert a string to an integer

`atoi` converts the argument to an integer value. The function accepts strings in the same format as the C function of the same name. The value returned has the format code `D`. `atoi` returns the integer 0 if it is unable to perform a conversion.

```
acid: +atoi("-1255")
-1255
```

`{}` `error(`*string*`)` Generate an interpreter error

`error` generates an error message and returns the interpreter to interactive mode. If an Acid program is running, it is aborted. Processes being debugged are not affected. The values of all local variables are lost. `error` is commonly used to stop the debugger when some interesting condition arises in the debugged program.

```
while 1 do {
   step();
   if *main != @main then
      error("memory corrupted");
}
```

list `file(`*string*`)` Read the contents of a file into a list

`file` reads the contents of the file specified by *string* into a list. Each element in the list is a string corresponding to a line in the file. `file` breaks lines at the newline character, but the newline characters are not returned as part each string. `file` returns the empty list if it encounters an error opening or reading the data.

```
acid: print(file("main.c")[0])
#include  <u.h>
```

integer `filepc(`*string*`)` Convert source address to text address

`filepc` interprets its *string* argument as a source file address in the form of a file name and line offset. `filepc` uses the symbol table to map the source address into a text address in the debugged program. The *integer* return value has the format `X`. `filepc` returns an address of -1 if the source address is invalid. The source file address uses the same format as *acme*(1). This function is commonly used to set breakpoints from the source text.

```
acid: bpset(filepc("main:10"))
acid: bptab()
   0x00001020 usage  ADD  $-0xc,R29
```

item `fmt(`*item,fmt*`)` Set print, `@` and `*` formats

`fmt` evaluates the expression *item* and sets the format of the result to *fmt*. The format of a value determines how it will be printed and what kind of object will be fetched by the `*` and `@` operators. The `\` operator is a short- handform of the `fmt` builtin function. The `fmt` function leaves the format of the *item* unchanged.

```
acid: main=fmt(main, 'i') // as instructions
acid: print(main\X, "\t", *main)
0x00001020 ADD   $-64,R29
```

list `fnbound(`*integer*`)` Find start and end address of a function

`fnbound` interprets its *integer* argument as an address in the text of the debugged program. `fnbound` returns a list containing two integers corresponding to the start and end addresses of the function containing the supplied address. If the *integer* address is not in the text segment of the program then the empty list is returned. `fnbound` is used by `next` to detect stepping into new functions.

```
acid: print(fnbound(main))
{0x00001050, 0x000014b8}
```

`{}` `follow(`*integer*`)` Compute follow set

The follow set is defined as the set of program counter values that could result from executing an instruction. `follow` interprets its *integer* argument as a text address, decodes the instruction at that address and, with the current register set, builds a list of possible next program counter values. If the instruction at the specified address cannot be decoded `follow` raises an error. `follow` is used to plant breakpoints on all potential paths of execution. The following code fragment plants breakpoints on top of all potential following instructions.

```
lst = follow(*PC);
while lst do
{
    *head lst = bpinst;
    lst = tail lst;
}
```

`{}` `include(`*string*`)` Take input from a new file

`include` opens the file specified by *string* and uses its contents as command input to the interpreter. The interpreter restores input to its previous source when it encounters either an end of file or an error. `include` can be used to incrementally load symbol table information without leaving the interpreter.

```
acid: include("/sys/src/cmd/acme/syms")
```

`{}` `interpret(`*string*`)` Take input from a string

`interpret` evaluates the *string* expression and uses its result as command input for the interpreter. The interpreter restores input to its previous source when it encounters either the end of string or an error. The `interpret` function allows Acid programs to write Acid code for later evaluation.

```
acid: interpret("main+10;")
0x0000102a
```

string `itoa(`*integer*`)` Convert integer to string

`itoa` takes an integer argument and converts it into an ASCII string in the `D` format. This function is commonly used to build `rc` command lines.

```
acid: rc("cat /proc/"+itoa(pid)+"/segment")
Stack    7fc00000 80000000    1
Data     00001000 00009000    1
Data     00009000 0000a000    1
Bss      0000a000 0000c000    1
```

`{}` `kill(`*integer*`)` Kill a process

`kill` writes a kill control message into the control file of the process specified by the *integer* pid. If the process was previously installed by `setproc` it will be removed from the list of active processes. If the *integer* has the same value as `pid`, then `pid` will be set to 0. To continue debugging, a new process must be selected using `setproc`. For example, to kill all the active processes:

```
while proclist do {
   kill(head proclist);
   proclist = tail proclist;
}
```

list `map(`*list*`)` Set or retrieve process memory map

`map` either retrieves all the mappings associated with a process or sets a single map entry to a new value. If the *list* argument is omitted then `map` returns a list of lists. Each sublist has four values and describes a single region of contiguous addresses in the memory or file image of the debugged program. The first entry is the name of the mapping. If the name begins with `*` it denotes a map into the memory of an active process. The second and third values specify the base and end address of the region and the fourth number specifies the offset in the file corresponding to the first location of the region. A map entry may be set by supplying a list in the same format as the sublist described above. The name of the mapping must match a region already defined by the current map. Maps are set automatically for Plan 9 processes and some kernels; they may need to be set by hand for other kernels and programs that run on bare hardware.

```
acid: map({"text", _start, end, 0x30})
```

integer `match(`*item,list*`)` Search list for matching value

`match` compares each item in *list* using the equality operator `==` with *item*. The *item* can be of any type. If the match succeeds the result is the integer index of the matching value, otherwise - 1.

```
acid: list={8,9,10,11}
acid: print(list[match(10, list)]\D)
10
```

`{}` `newproc(`*string*`)` Create a new process

`newproc` starts a new process with an argument vector constructed from *string*. The argument vector excludes the name of the program to execute and each argument in *string* must be space separated. A new process can accept no more than 512 arguments. The internal variable `pid` is set to the pid of the newly created process. The new pid is also appended to the list of active processes stored in the variable `proclist`. The new process is created then halted at the first instruction, causing the debugger to call `stopped`. The library functions `new` and `win` should be used to start processes when using the standard debugging environment.

```
acid: newproc("-l .")
56720: system call  _main  ADD $-0x14,R29
```

string `pcfile(`*integer*`)` Convert text address to source file name

`pcfile` interprets its *integer* argument as a text address in the debugged program. The address and symbol table are used to generate a string containing the name of the source file corresponding to the text address. If the address does not lie within the program the string `?file?` is returned.

```
acid: print("Now at ", pcfile(*PC), ":", pcline(*PC))
Now at ls.c:46
```

integer `pcline(`*integer*`)` Convert text address to source line number

`pcline` interprets its *integer* argument as a text address in the debugged program. The address and symbol table are used to generate an integer containing the line number in the source file corresponding to the text address. If the address does not lie within the program the integer 0 is returned.

```
acid: +file("main.c")[pcline(main)]
main(int argc, char *argv[])
```

`{}` `print(`*item,item,...*`)` Print expressions

`print` evaluates each *item* supplied in its argument list and prints it to standard output. Each argument will be printed according to its associated format character. When the interpreter is executing, output is buffered and flushed every 5000 statements or when the interpreter returns to interactive mode. `print` accepts a maximum of 512 arguments.

```
acid: print(10, "decimal ", 10\D, "octal ", 10\o)
0x0000000a decimal 10 octal 000000000012
acid: print({1, 2, 3})
{0x00000001 , 0x00000002 , 0x00000003 }
acid: print(main, main\a, "\t", @main\i)
0x00001020 main  ADD $-64,R29
```

`{}` `printto(`*string,item,item,...*`)` Print expressions to file

`printto` offers a limited form of output redirection. The first *string* argument is used as the path name of a new file to create. Each *item* is then evaluated and printed to the newly created file. When all items have been printed the file is closed. `printto` accepts a maximum of 512 arguments.

```
acid: printto("/env/foo", "hello")
acid: rc("echo -n $foo")
hello
```

string `rc(`*string*`)` Execute a shell command

`rc` evaluates *string* to form a shell command. A new command interpreter is started to execute the command. The Acid interpreter blocks until the command completes. The return value is the empty string if the command succeeds, otherwise the exit status of the failed command.

```
acid: rc("B "+itoa(-pcline(addr))+" "+pcfile(addr));
```

string `readfile(`*string*`)` Read file contents into a string

`readfile` takes the contents of the file specified by *string* and returns its contents as a new string. If `readfile` encounters a zero byte in the file, it terminates. If `readfile` encounters an error opening or reading the file then the empty list is returned. `readfile` can be used to read the contents of device files whose lines are not terminated with newline characters.

```
acid: ""+readfile("/dev/label")
helix
```

string `reason(`*integer*`)` Print cause of program stoppage

`reason` uses machine- dependent information to generate a string explaining why a process has stopped. The *integer* argument is the value of an architecture dependent status register, for example `CAUSE` on the MIPS.

```
acid: print(reason(*CAUSE))
system call
```

integer `regexp(`*pattern,string*`)` Regular expression match

`regexp` matches the *pattern* string supplied as its first argument with the *string* supplied as its second. If the pattern matches the result is the value 1, otherwise 0.

```
acid: print(regexp(".*bar", "foobar"))
1
```

`{}` `setproc(`*integer*`)` Set debugger focus

`setproc` selects the default process used for memory and control operations. It effectively shifts the focus of control between processes. The *integer* argument specifies the pid of the process to look at. The variable `pid` is set to the pid of the selected process. If the process is being selected for the first time its pid is added to the list of active processes `proclist`.

```
acid: setproc(68382)
acid: procs()
>68382: Stopped at main+0x4 setproc(68382)
```

`{}` `start(`*integer*`)` Restart execution

`start` writes a `start` message to the control file of the process specified by the pid supplied as its *integer* argument. `start` draws an error if the process is not in the `Stopped` state.

```
acid: start(68382)
acid: procs()
>68382: Running at main+0x4 setproc(68382)
```

{} `startstop(`*integer*`)` Restart execution, block until stopped

`startstop` performs the same actions as a call to `start` followed by a call to `stop`. The *integer* argument specifies the pid of the process to control. The process must be in the `Stopped` state. Execution is restarted, the debugger then waits for the process to return to the `Stopped` state. A process will stop if a startstop message has been written to its control file and any of the following conditions becomes true: the process executes or returns from a system call, the process generates a trap or the process receives a note. `startstop` is used to implement single stepping.

```
acid: startstop(pid)
75374: breakpoint   ls  ADD $-0x16c8,R29
```

string `status(`*integer*`)` Return process state

`status` uses the pid supplied by its *integer* argument to generate a string describing the state of the process. The string corresponds to the state returned by the sixth column of the *ps*(1) command. A process must be in the `Stopped` state to modify its memory or registers.

```
acid: ""+status(pid)
Stopped
```

{} `stop(`*integer*`)` Wait for a process to stop

`stop` writes a `stop` message to the control file of the process specified by the pid supplied as its *integer* argument. The interpreter blocks until the debugged process enters the `Stopped` state. A process will stop if a stop message has been written to its control file and any of the following conditions becomes true: the process executes or returns from a system call, the process generates a trap, the process is scheduled or the process receives a note. `stop` is used to wait for a process to halt before planting a breakpoint since Plan 9 only allows a process's memory to be written while it is in the `Stopped` state.

```
defn bpset(addr) {
   if (status(pid)!="Stopped") then {
      print("Waiting...\n");
      stop(pid);
   }
   ...
}
```

list `strace(`*pc,sp,linkreg*`)` Stack trace

`strace` generates a list of lists corresponding to procedures called by the debugged program. Each sublist describes a single stack frame in the active process. The first element is an *integer* of format `X` specifying the address of the called function. The second element is the value of the program counter when the function was called. The third and fourth elements contain lists of parameter and automatic variables respectively. Each element of these lists contains a string with the name of the variable and an *integer* value of format `X` containing the current value of the variable. The arguments to `strace` are the current value of the program counter, the current value of the stack pointer, and the address of the link register. All three parameters must be integers. The setting of *linkreg* is architecture dependent. On the MIPS linkreg is set to the address of saved `R31`, on the SPARC to the address of saved `R15`. For the other architectures *linkreg* is not used, but must point to valid memory.

```
acid: print(strace(*PC, *SP, linkreg))
{{0x0000141c, 0xc0000f74,
{{"s", 0x0000004d}, {"multi", 0x00000000}},
{{"db", 0x00000000}, {"fd", 0x000010a4},
{"n", 0x00000001}, {"i", 0x00009824}}}}
```

`{}` `waitstop(`*integer*`)` Wait for a process to stop

`waitstop` writes a waitstop message to the control file of the process specified by the pid supplied as its *integer* argument. The interpreter will remain blocked until the debugged process enters the `Stopped` state. A process will stop if a waitstop message has been written to its control file and any of the following conditions becomes true: the process generates a trap or receives a note. Unlike `stop`, the `waitstop` function is passive; it does not itself cause the program to stop.

```
acid: waitstop(pid)
75374: breakpoint   ls  ADD $-0x16c8,R29
```

Library Functions

A standard debugging environment is provided by modules automatically loaded when Acid is started. These modules are located in the directory `/sys/lib/acid`. These functions may be overridden, personalized, or added to by code defined in `$home/lib/acid`. The implementation of these functions can be examined using the `whatis` operator and then modified during debugging sessions.

`{}` `Bsrc(`*integer*`)` Load text editor with source

`Bsrc` interprets the *integer* argument as a text address. The text address is used to produce a path-name and line number suitable for the external `B` command of the text editor (eg, *acme*(1)). `Bsrc` builds a shell command to invoke `B`, which either selects an existing source file or loads a new source file into the editor. The line of source corresponding to the text address is then selected. In the following example `stopped` is redefined so that the editor follows and displays the source line currently being executed.

```
defn stopped(pid) {
   pstop(pid);
   Bsrc(*PC);
}
```

`{}` `Fpr()` Display double precision floating registers

For machines equipped with floating point, `Fpr` displays the contents of the floating point registers as double precision values.

```
acid: Fpr()
F0   0.   F2   0.
F4   0.   F6   0.
F8   0.   F10  0.
...
```

`{}` `Ureg(`*integer*`)` Display contents of Ureg structure

`Ureg` interprets the integer passed as its first argument as the address of a kernel `Ureg` structure. Each element of the structure is retrieved and printed. The size and contents of the `Ureg` structure are architecture dependent. This function can be used to decode the first argument passed to a *notify*(2) function after a process has received a note.

```
acid: Ureg(*notehandler:ur)
   status 0x3000f000
   pc  0x1020
   sp  0x7ffffe00
   cause  0x00004002
...
```

`{}` `acidinit()` Interpreter startup

`acidinit` is called by the interpreter after all modules have been loaded at initialization time. It is used to set up machine specific variables and the default source path. `acidinit` should not be called by user code.

`{} addsrcdir(`*string*`)` Add element to source search path

`addsrcdir` interprets its string argument as a new directory `findsrc` should search when looking for source code files. `addsrcdir` draws an error if the directory is already in the source search path. The search path may be examined by looking at the variable `srcpath`.

```
acid: rc("9fs fornax")
acid: addsrcpath("/n/fornax/sys/src/cmd")
```

`{} asm(`*integer*`)` Disassemble machine instructions

`asm` interprets its integer argument as a text address from which to disassemble machine instructions. `asm` prints the instruction address in symbolic and hexadecimal form, then prints the instructions with addressing modes. Up to twenty instructions will be disassembled. `asm` stops disassembling when it reaches the end of the current function. Instructions are read from the file image using the `@` operator.

```
acid: asm(main)
main     0x00001020 ADD    $-0x64,R29
main+0x4 0x00001024 MOVW   R31,0x0(R29)
main+0x8 0x00001028 MOVW   R1,argc+4(FP)
main+0xc 0x0000102c MOVW   $bin(SB),R1
```

`{} bpdel(`*integer*`)` Delete breakpoint

`bpdel` removes a previously set breakpoint from memory. The *integer* supplied as its argument must be the address of a previously set breakpoint. The breakpoint address is deleted from the active breakpoint list `bplist`, then the original instruction is copied from the file image to the memory image so that the breakpoint is removed.

```
acid: bpdel(main+4)
```

`{} bpset(`*integer*`)` Set a breakpoint

`bpset` places a breakpoint instruction at the address specified by its *integer* argument, which must be in the text segment. `bpset` draws an error if a breakpoint has already been set at the specified address. A list of current breakpoints is maintained in the variable `bplist`. Unlike in *db*(1), breakpoints are left in memory even when a process is stopped, and the process must exist, perhaps by being created by either `new` or `win`, in order to place a breakpoint. (`Db` accepts breakpoint commands before the process is started.) On the MIPS and SPARC architectures, breakpoints at function entry points should be set 4 bytes into the function because the instruction scheduler may fill `JAL` branch delay slots with the first instruction of the function.

```
acid: bpset(main+4)
```

`{} bptab()` List active breakpoints

`bptab` prints a list of currently installed breakpoints. The list contains the breakpoint address in symbolic and hexadecimal form as well as the instruction the breakpoint replaced. Breakpoints are not maintained across process creation using `new` and `win`. They are maintained across a fork, but care must be taken to keep control of the child process.

```
acid: bpset(ls+4)
acid: bptab()
    0x00001420 ls+0x4  MOVW R31,0x0(R29)
```

`{} casm()` Continue disassembly

`casm` continues to disassemble instructions from where the last `asm` or `casm` command stopped. Like `asm`, this command stops disassembling at function boundaries.

```
acid: casm()
main+0x10 0x00001030    MOVW   $0x1,R3
main+0x14 0x00001034    MOVW   R3,0x8(R29)
main+0x18 0x00001038    MOVW   $0x1,R5
main+0x1c 0x0000103c    JAL Binit(SB)
```

{} `cont()` Continue program execution

`cont` restarts execution of the currently active process. If the process is stopped on a breakpoint, the breakpoint is first removed, the program is single stepped, the breakpoint is replaced and the program is then set executing. This may cause `stopped()` to be called twice. `cont` causes the interpreter to block until the process enters the `Stopped` state.

```
acid: cont()
95197: breakpoint   ls+0x4 MOVW   R31,0x0(R29)
```

{} `dump(`*integer,integer,string*`)` Formatted memory dump

`dump` interprets its first argument as an address, its second argument as a count and its third as a format string. `dump` fetches an object from memory at the current address and prints it according to the format. The address is incremented by the number of bytes specified by the format and the process is repeated count times. The format string is any combination of format characters, each preceded by an optional count. For each object, `dump` prints the address in hexadecimal, a colon, the object and then a newline. `dump` uses `mem` to fetch each object.

```
acid: dump(main+35, 4, "X2bi")
0x00001043: 0x0c8fa700 108 143 lwc2 r0,0x528f(R4)
0x0000104d: 0xa9006811   0   0 swc3 r0,0x0(R24)
0x00001057: 0x2724e800   4  37 ADD  $-0x51,R23,R31
0x00001061: 0xa200688d   6   0 NOOP
0x0000106b: 0x2710c000   7   0 BREAK
```

{} `findsrc(`*string*`)` Use source path to load source file

`findsrc` interprets its *string* argument as a source file. Each directory in the source path is searched in turn for the file. If the file is found, the source text is loaded using `file` and stored in the list of active source files called `srctext`. The name of the file is added to the source file name list `srcfiles`. Users are unlikely to call `findsrc` from the command line, but may use it from scripts to preload source files for a debugging session. This function is used by `src` and `line` to locate and load source code. The default search path for the MIPS is `./`, `/sys/src/libc/port`, `/sys/src/libc/9sys`, `/sys/src/libc/mips`.

```
acid: findsrc(pcfile(main));
```

{} `fpr()` Display single precision floating registers

For machines equipped with floating point, `fpr` displays the contents of the floating point registers as single precision values. When the interpreter stores or manipulates floating point values it converts into double precision values.

```
acid: fpr()
F0   0.   F1   0.
F2   0.   F3   0.
F4   0.   F5   0.
...
```

{} `func()` Step while in function

`func` single steps the active process until it leaves the current function by either calling another function or returning to its caller. `func` will execute a single instruction after leaving the current function.

```
acid: func()
95197: breakpoint   ls+0x8 MOVW   R1,R8
95197: breakpoint   ls+0xc MOVW   R8,R1
95197: breakpoint   ls+0x10   MOVW   R8,s+4(FP)
95197: breakpoint   ls+0x14   MOVW   $0x2f,R5
95197: breakpoint   ls+0x18   JAL utfrrune(SB)
95197: breakpoint   utfrrune  ADD $-0x18,R29
```

{} `gpr()` Display general purpose registers

`gpr` prints the values of the general purpose processor registers.

```
acid: gpr()
R1  0x00009562 R2 0x000010a4 R3 0x00005d08
R4  0x0000000a R5 0x0000002f R6 0x00000008
...
```

{} `labstk(`*integer*`)` Print stack trace from label

`labstk` performs a stack trace from a Plan 9 *label.* The kernel and C compilers store continuations in a common format. Since the compilers all use caller save conventions a continuation may be saved by storing a `PC` and `SP` pair. This data structure is called a label and is used by the C function `longjmp` and the kernel to schedule threads and processes. `labstk` interprets its *integer* argument as the address of a label and produces a stack trace for the thread of execution. The value of the function `ALEF_tid` is a suitable argument for `labstk`.

```
acid: labstk(*mousetid)
At pc:0x00021a70:Rendez_Sleep+0x178 rendez.l:44
Rendez_Sleep(r=0xcd7d8,bool=0xcd7e0,t=0x0) rendez.l:5
    called from ALEF_rcvmem+0x198 recvmem.l:45
ALEF_rcvmem(c=0x000cd764,l=0x00000010) recvmem.l:6
...
```

{} `lstk()` Stack trace with local variables

`lstk` produces a long format stack trace. The stack trace includes each function in the stack, where it was called from, and the value of the parameters and automatic variables for each function. `lstk` displays the value rather than the address of each variable and all variables are assumed to be an integer in format `X`. To print a variable in its correct format use the `:` operator to find the address and apply the appropriate format before indirection with the `*` operator. It may be necessary to single step a couple of instructions into a function to get a correct stack trace because the frame pointer adjustment instruction may get scheduled down into the body of the function.

```
acid: lstk()
At pc:0x00001024:main+0x4 ls.c:48
main(argc=0x00000001,argv=0x7fffefec) ls.c:48
    called from _main+0x20 main9.s:10
    _argc=0x00000000
    _args=0x00000000
    fd=0x00000000
    buf=0x00000000
    i=0x00000000
```

{} `mem(`*integer,string*`)` Print memory object

`mem` interprets its first *integer* argument as the address of an object to be printed according to the format supplied in its second *string* argument. The format string can be any combination of format characters, each preceded by an optional count.

```
acid: mem(bdata+0x326, "2c2Xb")
P = 0xa94bc464 0x3e5ae44d  19
```

{} `new()` Create new process

`new` starts a new copy of the debugged program. The new program is started with the program arguments set by the variable `progargs`. The new program is stopped in the second instruction of `main`. The breakpoint list is reinitialized. `new` may be used several times to instantiate several copies of a program simultaneously. The user can rotate between the copies using `setproc`.

```
acid: progargs="-l"
acid: new()
60: external interrupt _main  ADD $-0x14,R29
60: breakpoint   main+0x4  MOVW   R31,0x0(R29)
```

`{}` `next()` — Step through language statement

`next` steps through a single language level statement without tracing down through each statement in a called function. For each statement, `next` prints the machine instructions executed as part of the statement. After the statement has executed, source lines around the current program counter are displayed.

```
acid: next()
60: breakpoint   Binit+0x4 MOVW   R31,0x0(R29)
60: breakpoint   Binit+0x8 MOVW   f+8(FP),R4
binit.c:93
 88
 89 int
 90 Binit(Biobuf *bp, int f, int mode)
 91 {
>92    return Binits(bp, f, mode, bp->b, BSIZE);
 93 }
```

`{}` `notestk(`*integer*`)` — Stack trace after receiving a note

`notestk` interprets its *integer* argument as the address of a `Ureg` structure passed by the kernel to a *notify*(2) function during note processing. `notestk` uses the `PC`, `SP`, and link register from the `Ureg` to print a stack trace corresponding to the point in the program where the note was received. To get a valid stack trace on the MIPS and SPARC architectures from a notify routine, the program must stop in a new function called from the notify routine so that the link register is valid and the notify routine's parameters are addressable.

```
acid: notestk(*notify:ur)
Note pc:0x00001024:main+0x4 ls.c:48
main(argc=0x00000001,argv=0x7fffefec) ls.c:48
   called from _main+0x20 main9.s:10
   _argc=0x00000000
   _args=0x00000000
```

`{}` `pfl(`*integer*`)` — Print source file and line

`pfl` interprets its argument as a text address and uses it to print the source file and line number corresponding to the address. The output has the same format as file addresses in *acme*(1).

```
acid: pfl(main)
ls.c:48
```

`{}` `procs()` — Print active process list

`procs` prints a list of active process attached to the debugger. Each process produces a single line of output giving the pid, process state, the address the process is currently executing, and the `setproc` command required to make that process current. The current process is marked in the first column with a `>` character. The debugger maintains a list of processes in the variable `proclist`.

```
acid: procs()
>62: Stopped at main+0x4 setproc(62)
 60: Stopped at Binit+0x8 setproc(60)
```

`{}` `pstop(`*integer*`)` — Print reason process stopped

`pstop` prints the status of the process specified by the *integer* pid supplied as its argument. `pstop` is usually called from `stopped` every time a process enters the `Stopped` state.

```
acid: pstop(62)
0x0000003e: breakpoint  main+0x4  MOVW   R31,0x0(R29)
```

`{}` `regs()` — Print registers

`regs` prints the contents of both the general and special purpose registers. `regs` calls `spr` then `gpr` to display the contents of the registers.

{} `source()` Summarize source data base

`source` prints the directory search path followed by a list of currently loaded source files. The source management functions `src` and `findsrc` use the search path to locate and load source files. Source files are loaded incrementally into a source data base during debugging. A list of loaded files is stored in the variable `srcfiles` and the contents of each source file in the variable `srctext`.

```
acid: source()
/n/bootes/sys/src/libbio/
/sys/src/libc/port/
/sys/src/libc/9sys/
/sys/src/libc/mips/
    binit.c
```

{} `spr()` Print special purpose registers

`spr` prints the contents of the processor control and memory management registers. Where possible, the contents of the registers are decoded to provide extra information; for example the `CAUSE` register on the MIPS is printed both in hexadecimal and using the `reason` function.

```
acid: spr()
PC  0x00001024 main+0x4  ls.c:48
SP  0x7fffef68 LINK  0x00006264 _main+0x28 main9.s:12
STATUS 0x0000ff33 CAUSE 0x00000024 breakpoint
TLBVIR 0x000000d3 BADVADR  0x00001020
HI  0x00000004 LO    0x00001ff7
```

{} `src(`*integer*`)` Print lines of source

`src` interprets its *integer* argument as a text address and uses this address to print 5 lines of source before and after the address. The current line is marked with a `>` character. `src` uses the source search path maintained by `source` and `addsrcdir` to locate the required source files.

```
acid: src(*PC)
ls.c:47
 42 Biobuf bin;
 43
 44 #define       HUNK   50
 45
 46 void
>47 main(int argc, char *argv[])
 48 {
 49     int i, fd;
 50     char buf[64];
 51
 52     Binit(&bin, 1, OWRITE);
```

{} `step()` Single step process

`step` causes the debugged process to execute a single machine level instruction. If the program is stopped on a breakpoint set by `bpset` it is first removed, the single step executed, and the breakpoint replaced. `step` uses `follow` to predict the address of the program counter after the current instruction has been executed. A breakpoint is placed at each of these predicted addresses and the process is started. When the process stops the breakpoints are removed.

```
acid: step()
62: breakpoint   main+0x8  MOVW   R1,argc+4(FP)
```

{} `stk()` Stack trace

`stk` produces a short format stack trace. The stack trace includes each function in the stack, where it was called from, and the value of the parameters. The short format omits the values of automatic variables. Parameters are assumed to be integer values in the format `X`; to print a parameter in the correct format use the `:` to obtain its address, apply the correct format, and use the `*` indirection operator to find its value. It may be necessary to single step a couple of instructions into a function to get a correct stack trace because the frame pointer adjustment instruction may get scheduled down into the body of the function.

```
acid: stk()
At pc:0x00001028:main+0x8 ls.c:48
main(argc=0x00000002,argv=0x7fffefe4) ls.c:48
    called from _main+0x20 main9.s:10
```

{} `stmnt()` Execute a single statement

`stmnt` executes a single language level statement. `stmnt` displays each machine level instruction as it is executed. When the executed statement is completed the source for the next statement is displayed. Unlike `next`, the `stmnt` function will trace down through function calls.

```
acid: stmnt()
62: breakpoint   main+0x18 MOVW   R5,0xc(R29)
62: breakpoint   main+0x1c JAL Binit(SB)
62: breakpoint   Binit     ADD $-0x18,R29
binit.c:91
 89 int
 90 Binit(Biobuf *bp, int f, int mode)
>91 {
```

{} `stopped(`*integer*`)` Report status of stopped process

`stopped` is called automatically by the interpreter every time a process enters the `Stopped` state, such as when it hits a breakpoint. The pid is passed as the *integer* argument. The default implementation just calls `pstop`, but the function may be changed to provide more information or perform fine control of execution. Note that `stopped` should return; for example, calling `step` in `stopped` will recur until the interpreter runs out of stack space.

```
acid: defn stopped(pid) {
    if *lflag != 0 then error("lflag modified");
    }
acid: progargs = "-l"
acid: new();
acid: while 1 do step();
<stdin>:7: (error) lflag modified
acid: stk()
At pc:0x00001220:main+0x200 ls.c:54
main(argc=0x00000001,argv=0x7ffffe8) ls.c:48
    called from _main+0x20 main9.s:10
```

{} `symbols(`*string*`)` Search symbol table

`symbols` uses the regular expression supplied by *string* to search the symbol table for symbols whose name matches the regular expression.

```
acid: symbols("main")
main   T  0x00001020
_main  T  0x0000623c
```

`{}  win()` Start new process in a window

`win` performs exactly the same function as `new` but uses the window system to create a new window for the debugged process. The variable `progargs` supplies arguments to the new process. The environment variable `$8½srv` must be set to allow the interpreter to locate the mount channel for the window system. The window is created in the top left corner of the screen and is 400x600 pixels in size. The `win` function may be modified to alter the geometry. The window system will not be able to deliver notes in the new window since the pid of the created process is not passed when the server is mounted to create a new window.

```
acid: win()
```