Principia Softwarica: The Plan 9 Debuggers and Tracers

version 0.1

Yoann Padioleau
yoann.padioleau@gmail.com

with code from
Phil Winterbottom and Ken Thompson

April 28, 2026

Contents

Chapter 1

Introduction

The goal of this book is to present with full details the source code of a debugger and tracer.

1.1 Motivations

Why a debugger and tracer? Because I think you are a better programmer if you fully understand how things work under the hood, and debuggers and tracers are actually some of the best tools to understand how things work.

This is most obvious when you open an unfamiliar codebase: rather than reading the source top-down and trying to guess which branches are actually taken at runtime, you can set a breakpoint at `main()`, single-step through the first few calls, and let the debugger show you the real control flow. A tracer gives you the same information at the system-call granularity—you immediately see which files the program opens, in what order, and how it reacts when they are missing. In both cases the debugger is less a bug-finding tool than a program-reading tool: it turns a pile of source into a live, observable execution.

Of course, when a program does crash, the debugger also lets you inspect exactly what went wrong: which instruction faulted, what the registers contained, what the call stack looked like. When a program misbehaves silently, a tracer reveals every system call it makes, often pointing directly to the problem.

Yet very few programmers know how these tools actually work. Understanding the internals of a debugger also deepens your understanding of the kernel (which provides the debugging facilities) and the toolchain (which generates the metadata the debugger reads).

Here are a few questions I hope this book will answer:

- How does a system call tracer intercept and observe the calls made by another program?
- How does a debugger control another process? Is a debugger a kind of binary interpreter? What kernel facilities does it use?
- How does a debugger set a breakpoint? What happens at the machine level when a breakpoint is hit?
- How does single-stepping work? How does the debugger advance the target by exactly one instruction?
- How does a debugger map a machine address to a function name or a source line?
- How does a debugger reconstruct the call stack of a stopped program, walking back through function frames to show a backtrace?
- How does a debugger make sense of a core dump long after the process has died—what does the kernel save, and what does the debugger reconstruct from it?
- How does a debugger find a local variable or a struct field by name, and how does it know the type of the value at a given address?

1.2 Plan 9 `acid` and `ratrace`

In this book, I will explain the code of the Plan 9 debugger `acid` and the system call tracer `ratrace` (the Plan 9 equivalent of Linux's `strace`). Like for most books in Principia Softwarica, I chose Plan 9 programs because they are simple, small, elegant, open source, and they form together a coherent set.

`acid` is particularly interesting: unlike traditional debuggers that provide a fixed set of commands, `acid` is built as an interpreter for a custom C-like language. Debugging operations—setting breakpoints, inspecting variables, walking data structures—are all written as `acid` functions. The user can define new functions to automate complex debugging tasks, making `acid` a *programmable debugger*.

Another important property of `acid` is that most of its code is *architecture-independent*. The architecture-specific details—register names, instruction decoding, breakpoint instructions—are isolated in `libmach`, a library that provides a uniform interface to different object file formats and instruction sets. Thanks to `libmach`, the same `acid` binary can debug programs compiled for ARM, x86, MIPS, or any other architecture that Plan 9 supports. `libmach` is also used by the other debugger `db`, the Plan 9 emulators, and the profiling tools.

Historically, `acid` was designed to debug Alef programs, Plan 9's original concurrent-with-channels language (a precursor to `libthread` and, much later, Go). This shaped `acid`'s process-table and multi-process control features from the start: because an Alef program could spawn many cooperating processes, the debugger had to follow forks, attach to children, and stop or resume any subset of them. The same machinery works unchanged for today's `libthread` programs, which are just Alef's ideas re-expressed as a C library. This means `acid` can debug multi-threaded programs.

`ratrace` is much simpler than `acid`: it traces the system calls made by a process by reading the kernel's `/proc` interface. Despite being a small program, it is one of the most effective debugging tools available—often the first thing you reach for when a program misbehaves.

1.3 Other debuggers

Here are a few debuggers and tracers that I considered for this book, but which I ultimately discarded:

- `db`, the traditional Plan 9 debugger, descends from the original UNIX debugger `db` written by Ken Thompson, which evolved into `adb` ("advanced debugger") by Steve Bourne. Plan 9's `db` is a command-line debugger with a terse syntax inherited from UNIX. It supports breakpoints, memory inspection, disassembly, and limited source-level debugging. However, `db` offers a fixed set of commands and is not programmable. It is presented in Appendix E.

- `gdb`[1] (the GNU Debugger) is the most widely used open-source debugger. It supports dozens of architectures and languages, reads DWARF[2] debug information, and has a Python scripting interface. However, its codebase is enormous (over 1.5 million LOC including the `BFD` binary file library, which plays a role similar to Plan 9's `libmach`).

- `lldb`[3] is LLVM's debugger, designed from the ground up with a Python API and a modular architecture. It is the default debugger on macOS (via Xcode). Like `gdb`, it is far too large to explain in a book.

- `valgrind`[4] is not a traditional debugger but a dynamic analysis framework. It runs programs on a synthetic CPU, instrumenting every memory access to detect leaks, use-after-free, buffer overflows, and uninitialized reads. It is one of the most effective tools for finding memory bugs in C programs, but its approach (full instrumentation) is entirely different from `acid`'s.

[1] https://www.gnu.org/software/gdb/
[2] https://dwarfstd.org/
[3] https://lldb.llvm.org/
[4] https://valgrind.org/

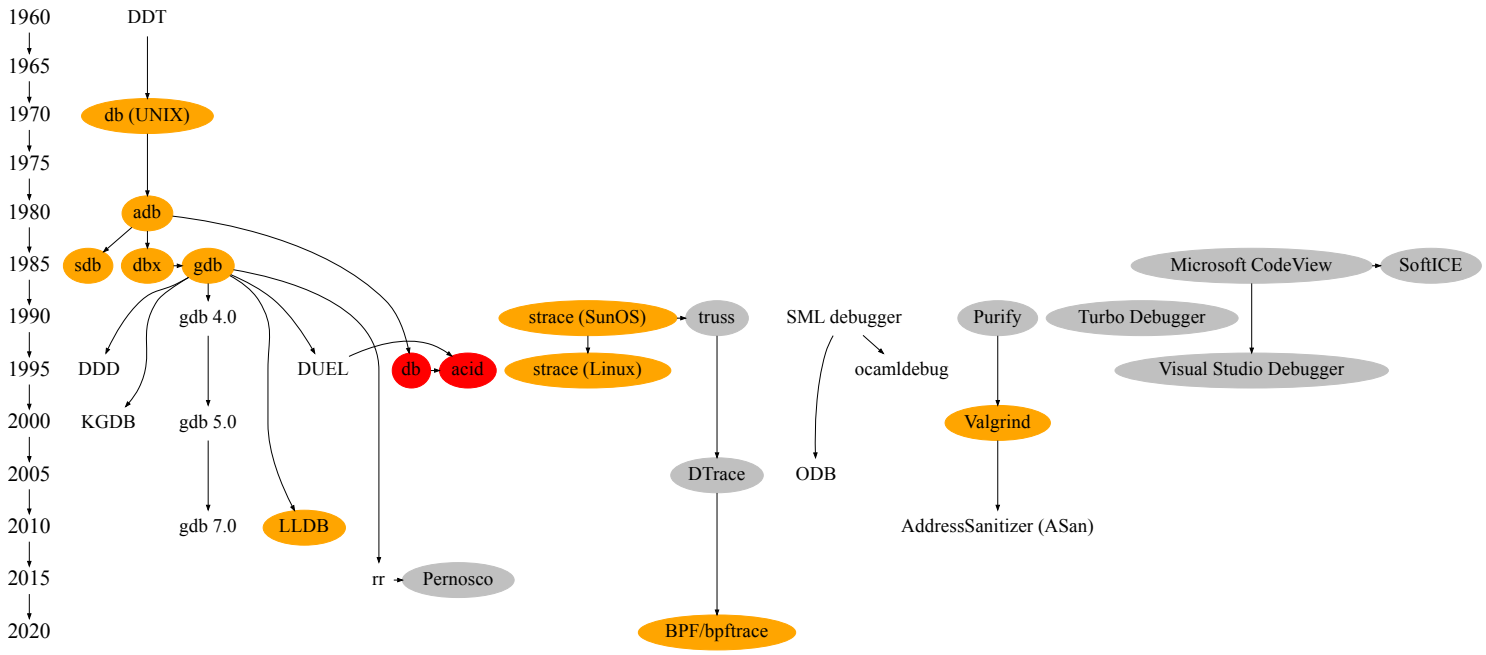


Figure 1.1: Debuggers and tracers timeline

- `ocamldebug`[5] is the OCaml debugger, notable for its time-travel capability: the user can step backward as well as forward through execution. This is a powerful feature, but the implementation relies on OCaml's bytecode interpreter and checkpointing mechanism, which makes it quite different from a native-code debugger.
- Linux `strace`[6] is functionally similar to `ratrace`: it traces system calls made by a process. On Linux, `strace` uses the `ptrace` system call to intercept syscalls, whereas `ratrace` reads Plan 9's `/proc` files. Linux also has `ftrace` and `uprobe` for tracing kernel and user-level functions.
- Solaris `dtrace`[7] is a comprehensive dynamic tracing framework that can instrument arbitrary points in the kernel and user programs without recompilation. It has its own scripting language (D) and was considered revolutionary when it appeared. Linux's eBPF[8] provides similar capabilities. However, DTrace is far more complex than the tools in this book.

Figure 1.1 presents a timeline of major debuggers and tracers. I think `acid` and `ratrace` represent the best compromise for this book: `acid`'s programmable approach is unique and makes it far more interesting to study than a traditional command-line debugger, while `ratrace` demonstrates system call tracing in its simplest form. Together they are small enough to explain in full detail.

1.4 Getting started

To play with `acid` and `ratrace`, you will first need to install the Plan 9 fork used in Principia Softwarica (see https://www.principia-softwarica.org/getting-started.html). To trace the system calls of a program:

```
$ ratrace -c /bin/ls /dev
...
```

[5] https://ocaml.org/manual/debugger.html
[6] https://strace.io/
[7] https://dtrace.org/
[8] https://ebpf.io/

```
75 ls Open  6a65 0xcfffeb3/"/dev" 0x0 = 3 "" 20000135 20000136
75 ls Brk   b834 0x20c98 = 0 "" 20000138 20000139
75 ls Pread b824 3 0xfbf8 65595 -1 0xfbf8/"=.M.....
...
/dev/bintime
/dev/cons
...
75 ls Close 6a6c 3 = 0 "" 20000140 20000142
75 ls Exits 6a74 0/""
$
```

To debug a program with `acid`:

```
$ acid /bin/ls
/bin/ls:386 plan 9 executable
/lib/acid/port.acid
/lib/acid/386.acid
acid: new()
72: system call _main      SUBL $0x48,SP
72: breakpoint  main+0x3   MOVL $bin(SB),AX
acid: stk()
main(argv=0xcfffefa4,argc=0x1)+0x3 /utilities/files/ls.c:91
_main+0x31 /libc/386/main9.s:16
acid: cont()
...
bin
boot
...
<stdin>:4 (error) msg: pid=72 startstop: process exited
acid:
```

`acid` is also available through both plan9port[9] and Goken9cc[10], so you can use it on Linux or macOS to debug native ELF binaries compiled with gcc or clang. On Linux, `libmach` uses `ptrace` under the hood for process control, memory access, single-stepping, and breakpoints—so the full debugging experience works. `acidtypes` extracts type information from DWARF debug sections.

1.5 Requirements

Because this book is made of C source code, you will need a good knowledge of the C programming language [KR88]. Reading the KERNEL book [Pad14] is helpful, especially the chapter on "Debugging Support": the kernel's `/proc` file system is the interface through which both `acid` and `ratrace` control and observe processes. The LINKER book [Pad15c] and COMPILER book [Pad16a] explain how the symbol tables and line number information that `acid` reads are generated.

For background on debugger internals in general, Rosenberg's *How Debuggers Work* [Ros96] is one of the few books on the subject, covering breakpoints, single-stepping, and symbol table interpretation.

If, while reading this book, you have specific questions on the command-line interface of `acid` or `ratrace`, or on the debugging facilities they rely on, I suggest you to consult the manual pages in my Plan 9 repository:

[9] https://9fans.github.io/plan9port/
[10] https://github.com/aryx/goken9cc

`docs/man/1/acid` and `docs/man/1/ratrace` for the tools themselves, `docs/man/1/db` for the ancestor debugger, `docs/man/1/trace` and `docs/man/1/prof` for the related observability tools, `docs/man/3/proc` for the `/proc` filesystem interface that backs both `acid` and `ratrace`, and `docs/man/2/debugger` and `docs/man/2/mach` for the `libmach` API.

The `debuggers/docs/` directory contains the two foundational papers by Phil Winterbottom: `debuggers/docs/a` ("ACID: A Debugger Built from a Language") and `debuggers/docs/acid.pdf`, together with a short tutorial in `debuggers/docs/acidtut.pdf`. These are essential reading for understanding the design choices behind `acid`. The "Libmach" paper at `docs/articles/libmach.pdf` in the top-level articles directory documents the portable machine abstraction that `acid` uses to read symbol tables, registers, and memory.

1.6 About this document

This document is a *literate program* [Knu92]. It derives from a set of files processed by a tool, `syncweb` [Pad09], generating either this book or the actual source code of the program. The code and its documentation are thus strongly connected.

1.7 Copyright

1.8 Acknowledgments

I would like to acknowledge of course the author of `acid`, Phil Winterbottom, who wrote in some sense most of this book.

Chapter 2

Overview

Before showing the source code of `acid` and `ratrace` in the following chapters, I first give an overview in this chapter of the general principles of tracers and debuggers. I also describe the services provided by `ratrace` and `acid`, walk through a small debugging session to show `acid` in action, and finally explain how the code is organized, what the software architecture of both tools looks like, and how the rest of this book is structured.

2.1 Tracer principles

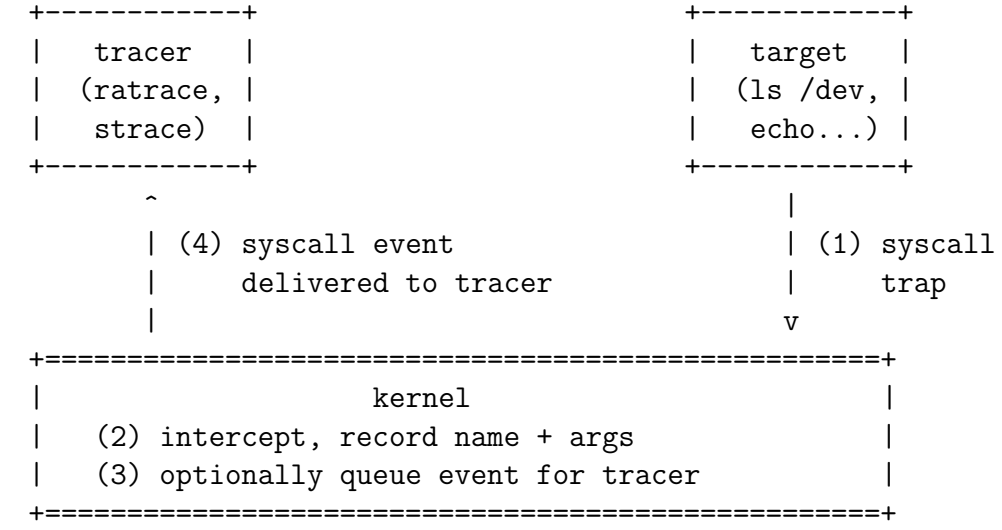
The most basic debugging technique is `printf`: insert print statements into the code and observe the output. This works, but it requires modifying the program, recompiling, and knowing where to look in advance.

A system call tracer provides automatic logging without any modification to the program. It intercepts every system call the program makes—`open`, `read`, `write`, `fork`, etc.—and prints the system call name, arguments, and return value.

2.1.1 Observing another program

Even the lightest form of debugging—a syscall tracer that only watches the program go by—is not trivial to implement: a user process cannot peek at another process's syscall stream directly, because the operating system isolates processes from each other. The tracer has to ask the kernel to report each syscall the target makes, and each OS exposes that facility differently.

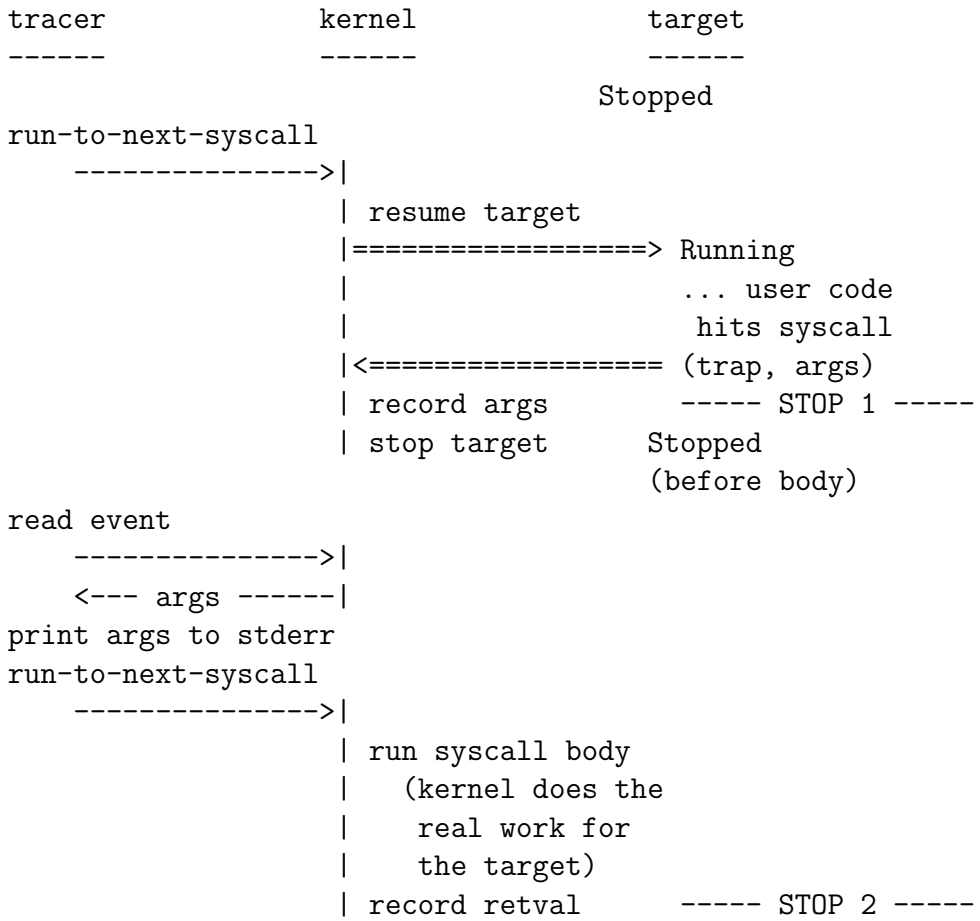
A tracing session involves three actors: the tracer process, the target process being observed, and the kernel sitting between them. The tracer never touches the target directly; it only receives events that the kernel forwards from the target's syscall entry and exit points:



The interface each OS provides for receiving syscall events falls into two broad patterns: a dedicated file that blocks until the next syscall, or a request code added to a catch-all debug system call.

- Plan 9 exposes syscall tracing as a file under `/proc/<pid>/`. A tracer opens that file and reads records describing each syscall the target makes. No special API, no request codes—just file I/O. This is exactly how `ratrace` is built (Chapter 4).
- UNIX and Linux route everything through the `ptrace` system call. `PTRACE_SYSCALL` asks the kernel to stop the target on each syscall entry and each syscall exit; the tracer then inspects the target's registers to recover the syscall number and arguments. `strace` is built on this single request code.
- macOS also has `ptrace` for compatibility but the preferred mechanism is `DTrace`'s `syscall` provider, which delivers syscall probe events without requiring a stopped target. DTrace is a much larger system than a tracer—a general dynamic instrumentation framework—but its syscall provider covers the same need.
- Windows exposes syscall-like events through Event Tracing for Windows (ETW), a kernel event bus that any subscriber can read. It is not specifically a debugger interface, but it plays the same role: the kernel forwards events to a separate observer.
- DOS had no process isolation, so "observation" was trivial: hook the interrupt vector for `INT 21h` (the DOS system call dispatcher) and see every call go by. No kernel cooperation needed because there was no kernel to speak of.

Whatever the OS-specific interface, a tracing cycle has the same shape. The tracer asks the kernel to run the target until its next syscall, the kernel does so and freezes the target inside the syscall, the tracer reads the recorded event, prints it, and resumes the target. The two never communicate directly—every arrow goes through the kernel:



```
                    | stop target      Stopped
                                       (after body)
  read event
      ---------------->|
      <-- retval -----|
  print retval to stderr
  run-to-next-syscall
      ---------------->|
                    | resume target
                    |==================> Running
                                           (back to user)
  ... loop to next syscall ...
```

Notice that each traced syscall produces *two* stops: one before the kernel runs the call (so the tracer can record arguments) and one after (so it can record the return value). The work of the call itself runs in between, executed by the kernel on behalf of the still-frozen target.

`ratrace` and `strace` both follow this two-stop, kernel-mediated protocol. The interfaces differ completely—`ratrace` writes `startsyscall` to a `/proc` file and reads records back, `strace` sends `PTRACE_SYSCALL` through a single `ptrace` system call—but the protocol shape is identical. Chapter 4 works through the concrete cycle for `ratrace` step by step.

So much for how the kernel lets us watch another process. The next question is what to watch. The answer for a tracer—as the interfaces above already hint—is system calls, and it is worth saying why.

2.1.2 Tracing boundaries

System calls are a good boundary for tracing because they represent every interaction between the program and the operating system. This makes tracers an effective first tool for many bugs: a program that fails silently might be trying to open a file that does not exist, or writing to a full disk. The tracer reveals these issues immediately, especially when the program does not check its error codes. Linux's `strace` and Plan 9's `ratrace` work this way. There is also `ltrace` on Linux, which traces library calls rather than system calls.

Modern observability tools (OpenTelemetry, Datadog APM, New Relic, Honeycomb, Jaeger) are a direct descendant of this same idea, just moved up the stack. Where `ratrace` traces *system calls* at the kernel boundary, an OpenTelemetry SDK traces *function calls* at API boundaries: an incoming HTTP request, an outgoing database query, a message publish, an RPC call. Each "span" records the function name, its arguments (tagged as attributes), the return value or error, and a high-resolution timestamp—exactly what `ratrace` prints, only for application-level operations instead of syscalls. The principle is identical: tracing at a well-chosen boundary turns silent bugs into visible ones, without modifying the program's logic. The main differences are that modern tracers report to a centralized collector instead of stderr, support distributed traces that span multiple machines by propagating a trace ID through request headers, and sample aggressively because tracing *every* HTTP call in a production system would overwhelm the collector—a scale problem `ratrace` never has to solve, because a single `ratrace` session targets a single process.

2.2 `ratrace` services

⟨*function usage (tracers/ratrace.c)* 17⟩≡ (371b)

```
void
usage(void)
{
    fprint(STDERR, "Usage: ratrace [-c cmd [arg...]] | [pid]\n");
    exits("usage");
}
```

`ratrace` has two modes of operation:

- `ratrace -c ls /dev` starts `ls /dev` and traces its system calls from the beginning.
- `ratrace 1234` attaches to an already running process with pid 1234 and traces its subsequent system calls.

Here is an example tracing a simple `echo hello` command:

```
$ ratrace -c echo hello
...
66 echo Pwrite 796c 1 0xa488/"hello." 6 -1hello
 = 6 "" 2000135 2000135
...
66 echo Exits 30bf 0/""
```

Each line shows the system call name, its arguments (file descriptor, buffer address, size, offset), and the return value after the `= `. Here we can see `echo` calling `Pwrite` to write 6 bytes (`hello\n`) to file descriptor 1 (standard output), followed by `Exits("")` to terminate successfully. When a program fails mysteriously, `ratrace` often reveals the cause immediately. For instance, an `Open` returning -1 or a `Pread` on a file descriptor that was never opened points directly to the bug.

Note that in the output above, the `= 6` return value appears on the line *after* the `Pwrite` call. This is because the program's own output (`hello`) is interleaved with `ratrace`'s trace output—both write to standard error, so they mix on screen. The `-1` in the `Pwrite` line is the file offset argument, not the return value.

2.3 Debugger principles

Where a tracer just observes system calls from outside, a debugger goes much further: it controls the target program itself and can stop it at arbitrary points, read and modify its memory and registers, and resume it. This section presents the principles common to most debuggers: how the debugger talks to the target process, the standard ways to stop it (breakpoints and watchpoints), the inspection facilities for examining its state, and the disassembler and source-level mechanisms that turn raw addresses and bytes back into something the user can read.

The three-actor picture introduced in Section 2.1.1 (debugger, target, and the kernel in the middle) applies here verbatim; a debugger just adds the write and control verbs on top of the read-only interface that a tracer already uses. Every facility below (controlling, breakpoints, watchpoints, inspection, disassembly, source-level mapping) fits into that same three-actor pattern.

2.3.1 Controlling another program

A debugger is a program that controls another program. One naive approach is full interpretation (as in the EMULATOR book [Pad15b]), which gives total control but is far too slow for real programs. Instead, debuggers rely on hardware and kernel support to control a target process efficiently.

The basic mechanism is simple: the debugger is a separate process that can read and write the target's memory and registers through kernel facilities. The debugger can stop the target at chosen addresses (via breakpoints, presented in the next subsection), inspect its state, modify it, and let it resume.

Debuggers have always relied on some form of kernel cooperation, even on systems with no `/proc` file system—the specific interface is the part that varies a lot across operating systems:

- UNIX and Linux extend the `ptrace` syscall we already met in the tracer case with write and control request codes: `PTRACE_ATTACH` to take control of an existing process, `PTRACE_POKEDATA` to write a word of target memory (needed to plant breakpoints, see below), `PTRACE_SETREGS` to modify registers, `PTRACE_CONT` to

resume, and `PTRACE_SINGLESTEP` to advance by one instruction. `ptrace` was introduced in Version 6 UNIX in 1975 as a single catch-all syscall whose behavior depends entirely on its request constant—an awkward API, but one `gdb` is built on.

- Plan 9, whose read side we already saw, exposes the write and control verbs through the same `/proc/<pid>/` files: writing to `mem` patches the target's code (so a debugger plants a breakpoint with an ordinary `write` syscall), writing to `regs` modifies its registers, and writing control messages such as `startstop` to `ctl` halts and resumes it. No extra syscall is needed, and the interface extends naturally across the network through 9P.

- macOS does not use `ptrace` (well, it has a partial implementation for compatibility, but it is unreliable). Instead, it relies on Mach exception ports, inherited from the Mach microkernel. A debugger registers itself as the exception handler for the target task, and the kernel delivers exceptions (faults, breakpoints) as Mach messages. Memory access goes through `mach_vm_read`/`mach_vm_write`, which operate on Mach virtual memory objects. This is why `lldb` on macOS uses a completely different backend than `lldb` on Linux.

- Windows provides a dedicated debugging API: `DebugActiveProcess` to attach, `WaitForDebugEvent` to wait for events, `ReadProcessMemory`/`WriteProcessMemory` for memory access, and `GetThreadContext`/`SetThr` for registers. Unlike UNIX, debug events are delivered synchronously to a dedicated debugger thread that polls `WaitForDebugEvent` in a loop. This is the model used by `WinDbg` and Visual Studio.

- DOS had no process isolation at all, so debugging was even simpler: the debugger could directly read and write any memory in the system. Breakpoints used the `INT 3` instruction (a single byte, `0xCC`), which triggered interrupt vector 3—the debugger had previously installed its own handler in this vector. Single-stepping used the trap flag in the FLAGS register, which causes `INT 1` to fire after every instruction. Tools like Borland's Turbo Debugger and SoftICE used this approach.

What differs across these systems is how the debugger receives fault notifications and how it accesses the target's state. The fundamental trap-based breakpoint mechanism, on the other hand, is universal—we look at it next. The debugger lineage (see also Figure 1.1 in the Introduction) runs from MIT's DDT (1961, the first interactive debugger, for the PDP-1) through AT&T's `adb` (Ritchie's UNIX address debugger), Berkeley's `dbx` (the first source-level debugger), GNU's `gdb` (Stallman, 1986, still the standard on Linux), LLVM's `lldb` (2010s, the macOS default), and Plan 9's `acid` (Rob Pike)—which is unusual in this lineage because it is *programmable*: you write debugging scripts in a C-like language rather than using a fixed command set.

2.3.2 Breakpoint

A breakpoint stops execution at a specific address. The debugger sets one by overwriting the instruction at the target address with a dedicated trap instruction that causes a fault. On x86 this trap is `INT 3`, which is the famous single-byte `0xCC`—deliberately one byte wide in the ISA for exactly this use case, so the debugger can poke it over any instruction without worrying about alignment. On ARM and most RISC architectures the trap is full-width: ARM `BKPT` is 4 bytes in A32 (2 in Thumb), ARM64 `BRK` is 4 bytes, RISC-V `EBREAK` is 4 bytes. The debugger saves the overwritten bytes and restores them when the breakpoint is removed or stepped over. When the target reaches that address, the fault stops execution and gives control back to the debugger; after inspection, the debugger restores the original instruction, single-steps over it, and re-inserts the trap so the breakpoint stays active. This mechanism is universal because it relies on hardware features that predate modern operating systems.

Breakpoints can be conditional (stop only when a condition is true) or counted (stop only after the n-th hit), but the underlying trap mechanism is the same.

An important detail is that the debugger modifies the instruction in the target process's memory image, not in the binary file on disk. The on-disk executable stays untouched, which means the breakpoint disappears

as soon as the process exits, and other processes running the same binary are unaffected. This requires the kernel to support writing to a process's text segment even when it is normally read-only and shared between processes—typically done via copy-on-write.

Here is the full breakpoint life cycle for a single hit, showing the three actors (debugger, kernel, debuggee) and the text-segment memory that changes across phases:

```
 debugger           kernel           debuggee text segment
 --------           ------           ---------------------
     |                 |             0x1020: MOVL $1,AX
set bp at 0x1020       |             0x1023: CALL foo
     |                 |             0x1028: ...
     |--- write /proc/<pid>/mem --->|
     |                 |             0x1020: INT3   <--- overwritten
     |                 |             0x1023: CALL foo
     |                 |             0x1028: ...
write "startstop" ---->|
     |                 |-- resume -->| (runs until 0x1020)
     |                 |             |
     |                 |<-- trap ----| (INT3 fires, fault
     |                 |              takes debuggee into
     |                 |              kernel mode, kernel
     |                 |              marks proc Stopped)
     |<-- returns -----|             | (startstop unblocks)
     |                 |
read regs, mem,        |
call stack             |
     |                 |
restore orig byte      |
     |--- write /proc/<pid>/mem --->|
     |                 |             0x1020: MOVL $1,AX  <--- restored
set bp at 0x1023       |             (next instruction after 0x1020)
     |--- write /proc/<pid>/mem --->|
     |                 |             0x1023: INT3
write "startstop" ---->|-- resume -->| (executes the MOVL,
     |                 |<-- trap ----| traps immediately at 0x1023)
     |<-- returns -----|
remove bp at 0x1023    |
re-insert bp at 0x1020 |
     |--- write /proc/<pid>/mem --->|
     |                 |             0x1020: INT3  <--- re-inserted
     |                 |             0x1023: CALL foo <--- restored
write "startstop" ---->|-- resume -->| (back to normal execution)
```

The diagram uses `startstop` (not `start`) because the debugger needs to *wait* for the target to hit the next trap before inspecting it; a plain `start` would resume the target without blocking. Note also how Plan 9 implements single-stepping: there is no dedicated "step" message in `/proc/<pid>/ctl`. Instead, `acid` uses `follow()`[411a] (backed by `machdata->foll`) to compute the set of possible next PCs from the current instruction, sets a temporary breakpoint at each of them, resumes with `startstop`, and then removes those temporary breakpoints. This software-based approach works on all architectures without requiring hardware single-step support (which ARM, unlike x86, does not have).

Two things are worth noting. First, the debugger never touches the debuggee directly—every arrow that reaches the target goes through the kernel via `/proc/<pid>/mem` and `/proc/<pid>/ctl`. Second, between "trap fires" and "`waitstop` wakes up", the debuggee is in the `Stopped` state and the scheduler simply does not run it; the kernel is free to schedule any other process in the meantime. The debugger regains control *asynchronously* when the scheduler next picks it, not immediately at the moment of the trap.

2.3.3 Watchpoint

A watchpoint (or data breakpoint) stops execution when a specific memory location is read or written. This is useful for finding out what code modifies a variable. Hardware watchpoints use the processor's debug registers to monitor an address without slowing down execution. Software watchpoints can be implemented by marking the memory page as inaccessible and catching the resulting page fault, though this is slower and very coarse-grained: since the unit of protection is a page (typically 4KB), *any* access anywhere in the page will trigger a fault, not just accesses to the watched variable. The debugger must then check whether the faulting access was the one of interest and silently resume otherwise—a lot of overhead if the watched variable lives among many others on the same page.

Here is how a hardware watchpoint works compared to a software (page-protection) watchpoint, watching variable `x` at address `0x4010`:

```
Hardware watchpoint              Software watchpoint
(debug register)                 (page protection)

CPU debug reg DR0 = 0x4010       Page containing 0x4010
DR7 = write, 4 bytes            marked PROT_NONE (no access)

Any instruction writes           Any instruction accesses
to 0x4010..0x4013:               anything on page 0x4000..0x4FFF:
  -> CPU raises #DB fault          -> CPU raises page fault
  -> kernel stops debuggee         -> kernel stops debuggee
  -> debugger inspects             -> debugger checks: was the
                                     faulting address 0x4010?
Precise, no false positives.       - yes: real watchpoint hit
Limited number (4 on x86,         - no: false positive,
typically 2-16 on ARM).              resume silently.
                                 Many false positives because
                                 page = 4KB granularity.
```

2.3.4 Inspector

The debugger can inspect the state of the stopped process. Even without any metadata, a debugger can do a lot if the kernel provides good introspection on the target process: register values, memory contents, the call stack.

Here is the map of process state a debugger can display, and the interface each OS provides to access it:

```
Process state            Plan 9              Linux
to inspect               (/proc/<pid>/)      (ptrace / /proc/)
====================     ================    ==================
General registers        read regs           PTRACE_GETREGS
Floating-point regs      read fpregs         PTRACE_GETFPREGS
Memory (code/data/       read/seek mem       PTRACE_PEEKDATA or
  stack/heap)                                  /proc/<pid>/mem
```

```
Open file descriptors    read fd            /proc/<pid>/fd/
Mapped segments          read segment       /proc/<pid>/maps
Namespace / mounts       read ns            /proc/<pid>/mounts
Process status           read status        /proc/<pid>/status
Pending signals/notes    read note          /proc/<pid>/status
Kernel registers         read kregs         (not exposed)
```

The key difference is that Plan 9 exposes everything as regular files you can `read` and `seek`—the same system calls used for ordinary I/O. Linux splits the interface between `ptrace` calls (for registers and memory) and `/proc` pseudo-files (for metadata like maps and fd). The Plan 9 approach is more uniform: any tool that can read files can inspect a process, and the interface extends naturally across the network through 9P.

However, for a good user experience, the debugger needs metadata generated by the toolchain: a symbol table mapping addresses to function and variable names (from the linker), and line number information mapping addresses to source lines (from the compiler). The "Debugging Support" chapters in the ASSEMBLER book [Pad15a], COMPILER book [Pad16a], and LINKER book [Pad15c] describe how this metadata is generated.

Here is what these two tables look like for a tiny C program `hello.c`:

```
/* hello.c */
int main(void) {       /* line 1 */
    int x = 42;        /* line 2 */
    return x + 1;      /* line 3 */
}                      /* line 4 */
```

Symbol table

name	address
main	0x1020
x (auto)	-0x4

Line number table

address	file	line
0x1020	h.c	1
0x1023	h.c	2
0x102a	h.c	3

With these two tables, the debugger can answer questions like "what function is at address `0x1025`?" (look up in the symbol table: `main+0x5`) and "what source line corresponds to address `0x1025`?" (look up in the line table: `hello.c:2`). It can also answer the reverse: "what address does `main` start at?" or "what address should I set a breakpoint at to stop on line 3?"

With symbol table information, the debugger can display function names and variable values instead of raw addresses and numbers. With type information (e.g., from `acid`'s type descriptions or DWARF on other systems), it can even display structured data types like structs and arrays in a readable format.

2.3.5 Stack trace

A stack trace (or backtrace) reconstructs the chain of function calls that led to the current point of execution. The debugger walks backward through the stack frames, reporting each caller's name and program counter. The challenge is that the debugger needs to know how large each frame is in order to find the next one. Different systems solve this differently:

- On x86 with the traditional frame pointer convention, each function starts with `push ebp; mov ebp,esp`, creating a linked list of frame pointers on the stack. The debugger follows the `ebp` chain: each saved `ebp` points to the previous frame, and the return address sits at `ebp+4`. This is simple but costs one register.

- Plan 9 uses a SP-PC table instead: the linker generates a table mapping each PC value to the corresponding stack pointer offset at that point. Given the current PC and SP, the debugger looks up the SP offset in the table, computes the caller's SP, reads the return address from the stack, and repeats. This avoids dedicating a register to frame pointers.

- Modern compilers on Linux typically omit the frame pointer (`gcc -fomit-frame-pointer`, the default since 2005) and rely on DWARF `.debug_frame` / `.eh_frame` tables, which encode the same information as Plan 9's SP-PC table but in a far more complex format (a stack machine bytecode called the DWARF Call Frame Information, or CFI).

All three approaches solve the same problem—"given the current PC and SP, how big is this frame?"—but trade off runtime cost (frame pointer burns a register), metadata size (DWARF is verbose), and simplicity (Plan 9's approach is a compact array).

2.3.6 Disassembler

A disassembler converts machine code back into assembly mnemonics, allowing the user to see what instructions the processor will execute. This is the reverse of what the assembler does. The debugger typically uses a library (e.g., `libmach` in Plan 9) to decode the binary instructions at a given address. Appendix D presents the ARM disassembler in `libmach/5db.c`, which decodes 32-bit ARM instructions back into Plan 9 assembly syntax.

2.3.7 Source-level debugging

With line number information from the compiler, the debugger can display source code instead of assembly, set breakpoints on source lines, and step one source statement at a time. This requires the debugger to map between addresses and source positions in both directions. Source-level debugging is what most developers expect, but it requires cooperation from the entire toolchain: the compiler must emit line tables, the linker must preserve them, and the debugger must read them. The diagram above already illustrates the two tables involved.

2.3.8 The Heisenbug

A *Heisenbug* (Jim Gray, 1986—a pun on the Heisenberg uncertainty principle) is a bug that changes behavior when you try to observe it. Add a `printf` and the timing shifts enough that the race condition disappears. Run under a debugger and the memory layout changes so the buffer overflow corrupts something different. Attach `acid` or `gdb` to a process and its scheduling is perturbed—context switches land in different places, caches warm differently, and the bug you were trying to reproduce quietly vanishes. The phenomenon is not rare: it is the defining challenge of debugging concurrent and real-time code, and every experienced developer has a war story about a bug that only appeared in production and never under the debugger.

The Heisenbug is the reason why different debugging techniques carry different *perturbation costs*. `printf` is low-perturbation: it adds a few microseconds per call and barely changes the program's memory layout, which is why even developers with access to sophisticated debuggers reach for `printf` first. A syscall tracer like `ratrace` is medium: it stops the target on every syscall but lets it run freely in between. A full debugger like `acid` is high-perturbation: breakpoints, single-stepping, and register inspection change the target's execution at every step. The practical lesson is that no single technique works for all bugs: you need the whole spectrum, from low-perturbation logging to high-perturbation inspection, and the choice depends on whether the bug survives observation.

2.3.9 Observing the unobservable

A running program is, from the outside, a black box: you can see what goes in (`argv`, `stdin`) and what comes out (`stdout`, exit status), but you cannot see what happens inside—the values of local variables, the call stack,

the contents of memory at a given address, the exact instruction being executed at the moment something goes wrong. The debugger is the instrument that makes the interior visible. `printf` debugging is the crude version: drill a hole in the box and peek through one variable at a time, recompiling after every change. A symbolic debugger like `acid` is the sophisticated version: stop the program at any point, inspect every register and memory cell, walk the stack, disassemble the code, modify a variable, and resume. The gap between the two is the gap between "I think the bug is here" and "I can *see* the bug"—and the tools in this book exist to close that gap.

2.4 `acid` services

⟨*function usage (acid/main.c)* 24⟩≡ (383c)

```
void
usage(void)
{
    fprint(STDERR, "usage: acid [-kqw] [-l library] [-m machine] [pid] [file]\n");
    exits("usage");
}
```

The main flags are:

- `-l library`: load an `acid` library file at startup. Libraries provide higher-level debugging functions; for instance, `acid -l truss` loads a system call tracer written entirely in `acid`'s own language (see Section 14.3 for more information).
- `-k`: debug kernel state rather than user state (see Section 14.6)
- `-w`: allow modifying the text file (for patching binaries). (see Section 14.5). This is useful when you need to permanently change a binary without recompiling—for example, replacing a hard-coded path, flipping a feature flag stored as a constant, or NOPing out a buggy instruction in a vendor binary you do not have the source for.
- `-m machine`: specify the CPU type for disassembly. (see Section 14.8). This is particularly useful in Plan 9 because `/proc` can be imported across the network: you can mount a remote machine's `/proc` (e.g., from an ARM terminal) on your local x86 desktop and run `acid -m arm` to debug processes running on the remote machine. The same trick works for kernel debugging on a different architecture.

You can specify a running process by `pid`, or a binary by `textfile` (defaulting to `8.out`). At startup, `acid` loads standard function definitions from `/lib/acid/port.acid` and architecture-specific functions from `/lib/acid/$objtype.acid`.

The key functions available at the `acid:` prompt include: `new()` (start the program), `bpset()` (set a breakpoint), `cont()` (continue execution), `step()` (single-step), `next()` (statement-step), `stk()` (print call stack with arguments), `src()` (show source around an address), `asm()` (disassemble), `regs()` (print registers), and `print()` (display a value).

Because these are regular `acid` functions defined in the standard library—not built-in commands—the user can redefine them or write new ones to automate complex debugging tasks. For example, with a programmable debugger you can write a function that walks a linked list and prints every element, or one that sets a breakpoint on every system call entry and prints the arguments (this is exactly what `/lib/acid/truss` does—a system call tracer written in 100 lines of `acid`). You could also write a function that detects memory leaks by hooking `malloc` and `free`, or one that records every function entry/exit to produce a trace. Scripts with similar goals already exist in `/lib/acid/leak` and `/lib/acid/coverage` (though, as we will see in Chapter 14, they take a different route). With `db` or basic `gdb` commands, these tasks would external script that drives the debugger through a command file—neither approach scales.

2.5 Debugging `hello_bug.c`

Here is a typical `acid` debugging session on a small program that divides by zero:

⟨hello_bug.c 25⟩≡

```
#include <u.h>
#include <libc.h>

int foo(int x, int y) {
    return x / y;
}

int bar(int a, int b) {
    return foo(a, b) + 3;
}

void
main(int argc, char **argv)
{
    int x, y, z;
    x = 41 + argc;
    y = argc - 1;
    z = bar(x, y);
    print("%d %d %d\n", x, y, z);
    exits(nil);
}
```

2.5.1 Running to a breakpoint

The first half of the session covers a typical happy-path inspection: start the program, set a breakpoint at a function of interest, continue to it, and look around. This is the basic debugging workflow you would use even on a program that is not yet failing—for example, to understand the state at the entry of a suspicious function.

We compile it, start `acid`, set a breakpoint on `bar`, run to it, and inspect the state:

```
$ cd /tests/acid/
$ 8c hello_bug.c
$ 8l -o hello_bug hello_bug.8
$ acid ./hello_bug
./hello_bug:386 plan 9 executable
/lib/acid/port.acid
/lib/acid/386.acid
acid: new()
66: system call  _main   SUBL $0x48, SP
66: breakpoint main+0x3  MOVL argc+0x0(FP), AX
acid: bpset(bar)
acid: cont()
66: breakpoint bar   SUBL $0x8, SP
acid: stk()
bar(a=0x2a,b=0x0)+0x0 /tests/acid/hello_bug.c:8
main(argc=0x1)+0x21 /tests/acid/hello_bug.c:25
_main+0x31  /lib_core/libc/386/main9.s:16
acid: src(*PC)
/tests/acid/hello_bug.c:8
 3
```

```
 4  int foo(int x, int y) {
 5      return x / y;
 6  }
 7
>8  int bar(int a, intb) {
 9     return foo(a, b) + 3;
 10 }
 11
 12 void
 13 main(int argc, char **argv)
```

`new()` starts the program and stops at the first `main` instruction. `bpset(bar)` sets a breakpoint at the entry of `bar`. `cont()` resumes execution until the breakpoint is hit. `stk()` prints the call stack with argument values, and finally `src(*PC)` shows the source code around the current program counter.

Up until this point, no bug. Let's continue.

2.5.2 Catching the divide-by-zero

The second half is where the actual debugging happens. We resume execution, hit a fault, then use `acid`'s inspection commands to find out what went wrong: read the parameters that caused the fault, look at the offending machine instruction, and check the register state.

```
acid: cont()
66: breakpoint bar+0x3   MOVL a+0x0(FP),CX
66: divide error    foo+0x5 IDIVL AX,y+0x4(FP)
Notes pending:
    sys: trap: divide error
acid: src(*PC)
 1  #include <u.h>
 2  #include <libc.h>
 3
 4  int foo(int x, int y) {
>5      return x / y;
 6  }
 7
 8  int bar(int a, intb) {
 9     return foo(a, b) + 3;
 10 }
acid: *foo:y
0x00000000
acid: *foo:x
0x0000002a
acid: *foo:x\d
42
acid: asm(foo)
foo     0x00001020   MOVL x+0x0(FP),AX
foo+0x4 0x00001024   CDQ
foo+0x5 0x00001025   IDIVL AX,y+0x4(FP)
foo+0x9 0x00001029   RET
bar 0x0000102a  SUBL  $0x8,SP
```

```
acid: regs()
PC  0x00001025 foo+0x5 /tests/acid/hello_bug.c:5
SP  0xcfffef20 ECODE 0xe010068d EFLAG 0x00000286
CS  0x00000023 DS 0x0000001b SS 0x0000001b
GS  0x0000001b FS 0x0000001b ES 0x0000001b
TRAP    0x000000000 divide error
AX  0x0000002a BX 0x0000002a CX 0x00000000 DX 0x00000000
DI  0x0002e641 SI 0x00000000 BP 0xcfffba48
acid:
```

`cont()` resumes execution, but this time the program hits a divide-by-zero fault inside `foo` before reaching `bar` breakpoint. The kernel stops the process and `acid` reports the fault: "divide error" at `foo+0x5`, the `IDIVL` instruction. `src(*PC)` confirms we are on line 5, the `return x / y` statement. Now we can inspect the arguments that caused the fault: `*foo:y` reads the value of parameter `y` from `foo`'s stack frame and shows `0x00000000`—the zero divisor. `*foo:x` shows `0x0000002a` (42). The `\d` format suffix in `*foo:x\d` displays it as decimal. `asm(foo)` disassembles `foo`, showing the three-instruction body: load `x` into `AX`, sign-extend with `CDQ`, then `IDIVL` which divides `AX` by `y`—the faulting instruction. Finally, `regs()` dumps the full register state at the time of the fault: `AX=0x2a` (42, the dividend) and `TRAP=divide error`. Note that the divisor is *not* in a register: looking at the `asm(foo)` output above, the `IDIVL` instruction reads its second operand from `y+0x4(FP)`—i.e., from the stack frame, not a register. That is why we had to use `*foo:y` to read it: it lives at offset `+0x4` from the frame pointer of `foo`'s stack frame. This is the typical acid debugging workflow: inspect source, read variables, check assembly, examine registers.

2.5.3 Variables are addresses, not values

A common surprise for first-time `acid` users is that *every variable behaves like a pointer*, even ones that are plain integers in the C source. In our example, `foo` takes `int x` and `int y`—two ordinary `int` parameters. Yet to see their values at the `acid` prompt, we wrote `*foo:y` and `*foo:x` with an explicit dereference. Why?

Because in `acid`, a name like `foo:x` resolves to the address where `x` lives in memory (the offset of `x` within `foo`'s stack frame, plus the current frame pointer), not the value stored there. To get the value, you have to dereference the address with `*`. The same is true for global variables: typing `main` at the prompt gives you the address of `main` (e.g., `0x1020`), not the bytes of code at that address. Even register names work this way: `PC` is an offset into `/proc/<pid>/regs`, and `*PC` is the actual program counter value. This convention makes `acid` more uniform than C: every identifier is an address, and dereferencing is always explicit. It also matches how a debugger thinks: the debugger already knows where every variable lives, what it cannot guess is when you want to look at the variable's location versus the value stored there.

2.5.4 `*` vs `@`: live process vs binary file

There are actually *two* dereference operators in `acid`: `*` and `@`. They look similar but read from two different places:

- `*addr` reads from `cormap`—the live process memory via `/proc/<pid>/mem`. This gives you the current runtime state: actual values of variables, contents of the stack, current instructions (which may include breakpoints that `acid` inserted).
- `@addr` reads from `symmap`—the executable file on disk. This gives you the original, unmodified contents: the code as the linker laid it out, before any breakpoint patches.

For *data* segments, the two are typically the same right after `exec` (the kernel maps the file's data segment into memory), but they diverge as soon as the program writes to a global variable: `*` sees the new value, `@` sees the

original. For *text*, they differ whenever `acid` has set a breakpoint: `*addr` returns the breakpoint instruction (`INT 3` / `BKPT`), while `@addr` returns the original instruction. This is exactly how `bpdel()` restores the original byte—it reads it back from `symmap` via `@` and writes it through `cormap` via `*`. Most of the time you want `*`: it is the equivalent of "inspect this running process." You only need `@` for unusual tasks like patching, restoring instructions, or comparing the file against the running image.

2.6 Code organization

Table 2.1 presents short descriptions of the source files of `acid` and `ratrace`, together with the main entities (e.g., structures, functions, globals) the file defines, and the corresponding chapter in this document in which the code contained in the file is primarily discussed.

Function	Ch.	File	Entities	LOC
system call tracer	4	`ratrace.c`	`threadmain()`[37] `reader()`[43] `writer()`[41b]	308
data structures	6	`acid.h`	`Node`[378b] `Value`[378b] `Store`[378b] `Lsym`[378b]	407
globals	6	`globals.c`	`symmap`[55a] `cormap`[55b] `interactive`[70a]	96
entry point	7	`main.c`	`main()`[69e] `readtext()`[74d] `loadmodule()`[78a]	736
process management	7	`proc.c`	`nproc()`[82b] `sproc()`[85b] `install()`[83b] `msg()`[56d]	326
executable parsing	8	`libmach/executable.c`	`crackhdr()`[90]	477
machine selection	8	`libmach/setmach.c`	`machbytype()`[91]	102
symbol table	8	`libmach/sym.c`	`syminit()`[94] `symbase()`[96a]	1518
memory map	8	`libmach/map.c`	`loadmap()`[92] `setmap()`[323b]	190
memory access	8	`libmach/access.c`	`get1()`[351a] `put1()`[352c] `attachproc()`[96b]	295
machine data	8	`libmach/machdata.c`	`symoff()`[356a]	493
ARM support	8	`libmach/5.c`, `5db.c`	`armreglist`[308f] `armdas()`[211d]	1514
lexer	9	`lex.c`	`yylex()`[103a] `numsym()`[104b] `pushfile()`[100d]	734
parser	9	`dbg.y`	`yyparse()X`	360
interpreter	10	`exec.c`	`execute()`[120a] `call()`[136b]	545
expression evaluator	10	`expr.c`	`oadd()`[129] `ocall()`[135d] `oasgn()`[406b]	1110
list operations	10	`list.c`	`al()`[63c] `addlist()`[393c] `construct()`[392c]	304
builtins	10	`builtin.c`	`tab`[68a] `bprint()`[144d] `follow()`[411a]	1439
printing	??	`print.c`	`bprint()` `patom()`[145b]	466
dot (complex access)	10	`dot.c`	`odot()`[398b] `oframe()`[403a]	168
utilities	??	`util.c`	`isnumeric()`[209] `gc()`[149]	338
Total				13400

Table 2.1: Chapters and associated `acid` and `ratrace` source files.

2.7 Software architecture

The overall architecture involves two processes: the debugging process (`acid`) and the debugged process (the debuggee). `acid` controls the debugged process by writing messages to its `/proc/<pid>/ctl` file (`start`, `stop`,

`waitstop`, `startstop`, `hang`), accesses and modifies its memory through `/proc/<pid>/mem`, and reads its registers through `/proc/<pid>/regs`.

The `libmach` library sits between `acid` and the raw binary data. It parses executables (`crackhdr()`[90]), loads symbol tables (`syminit()`[94]), disassembles instructions, and provides a uniform `Map` abstraction for memory access. Thanks to `libmach`, the same `acid` code works across ARM, x86, and MIPS—the architecture-specific details are isolated in per-architecture source files (`5.c`/`5db.c` for ARM, `8.c`/`8db.c` for x86).

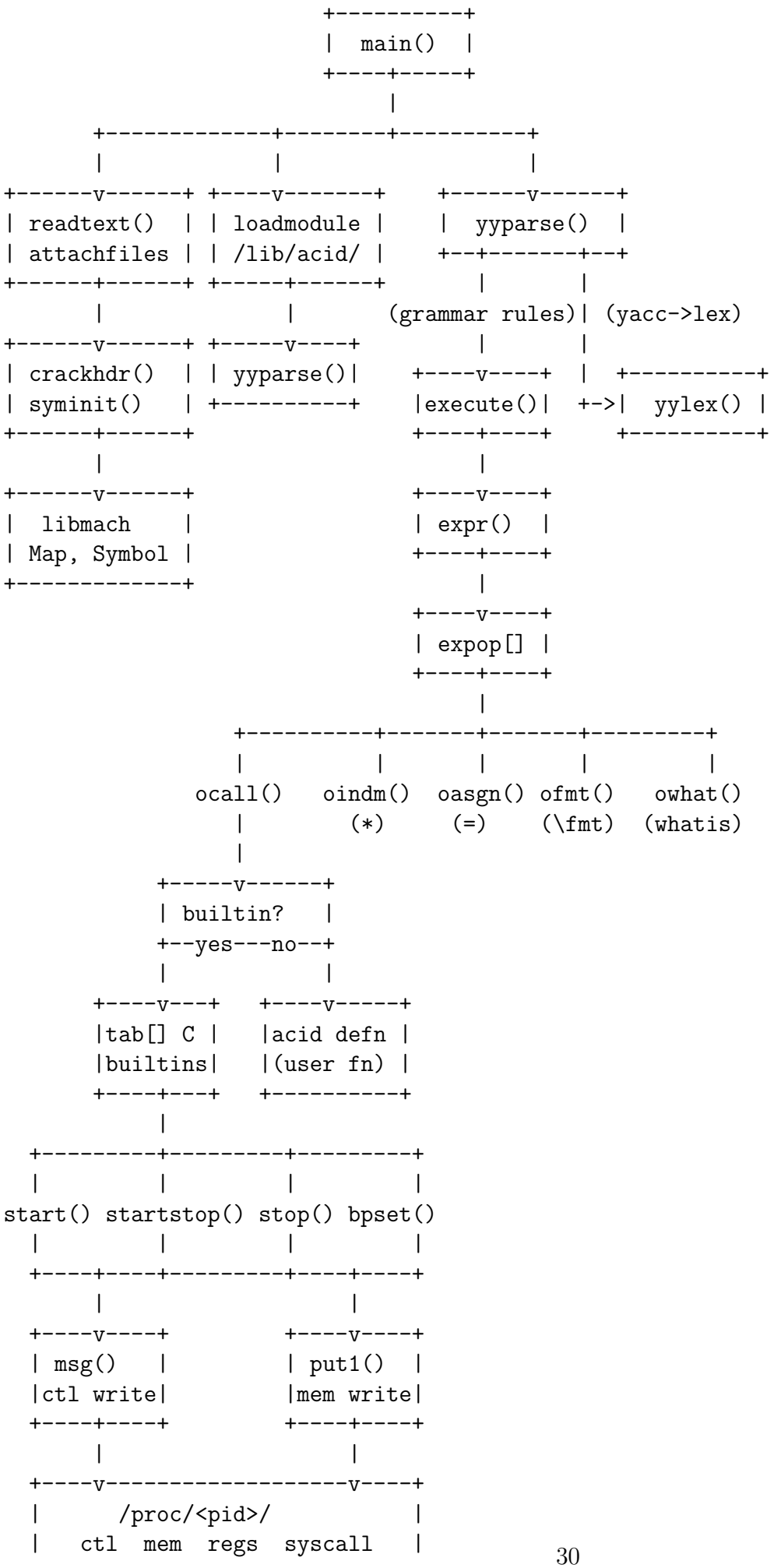
The toolchain also cooperates: the assembler, compiler, and linker all generate metadata that `acid` reads. The symbol table maps addresses to function and variable names, the line number table maps addresses to source positions, and the SP-PC table tracks stack pointer offsets for unwinding the call stack.

2.8 Book structure

You now have enough background to understand the source code of `acid` and `ratrace`. The rest of the book is organized as follows.

I will begin with the external support that `acid` relies on: Chapter 3 describes the kernel's `/proc` interface for process control and memory access. Then, Chapter 4 presents `ratrace`, the system call tracer—a small program that demonstrates how `/proc` can be used to observe a running process. I present it before `acid` because it is simple yet exercises all the kernel mechanisms that `acid` also relies on. Chapter 5 then reviews the metadata (symbol tables, line number information) generated by the assembler, compiler, and linker that `acid` uses to map addresses to source-level entities. Chapter 6 describes the core data structures of `acid` and `libmach` (executable headers, machine descriptions, memory maps, AST nodes, runtime values, symbol table). Chapter 7 presents `main()`, the REPL, the initialization sequence, and process attachment. Chapter 8 presents `libmach`, the library for parsing executables, loading symbol tables, and disassembling instructions, and how `acid` uses it. The following chapters describe the `acid` language implementation: Chapter 9 covers the lexer and parser, and Chapter 10 the interpreter and expression evaluator (including the garbage collector and the function call mechanism). Chapter 11 introduces the standard `acid` library files in `/lib/acid/` that build higher-level debugging commands on top of the language primitives. Chapter 12 then presents the source code of the most important builtins and library functions used in a typical debugging session: process control (`new`, `cont`, `step`, `bpset`), inspection (`stk`, `regs`, `print`), and disassembly (`asm`). Chapter 13 covers source-level debugging features (`src`, `pcline`, `pcfile`). Chapter 14 covers advanced features such as binary patching (`-w`), kernel debugging (`-k`), remote debugging (`-r`), and cross-architecture debugging (`-m`). Finally, Chapter 15 concludes and gives pointers to other books in the Principia Softwarica series.

Some appendices present the code of non-functional properties: code to help debug `acid` itself in Appendix A, error management in Appendix B, and utility functions in Appendix C. Appendix E presents `db`, the traditional Plan 9 debugger, for reference.


main()
readtext()
attachfiles
loadmodule
/lib/acid/
yyparse()
crackhdr()
syminit()
yyparse()
(grammar rules)
(yacc->lex)
libmach
Map, Symbol
execute()
yylex()
expr()
expop[]
ocall()
oindm()
(*)
oasgn()
(=)
ofmt()
(\fmt)
owhat()
(whatis)
builtin?
yes
no
tab[] C
builtins
acid defn
(user fn)
start()
startstop()
stop()
bpset()
msg()
ctl write
put1()
mem write
/proc/<pid>/
ctl
mem
regs
syscall

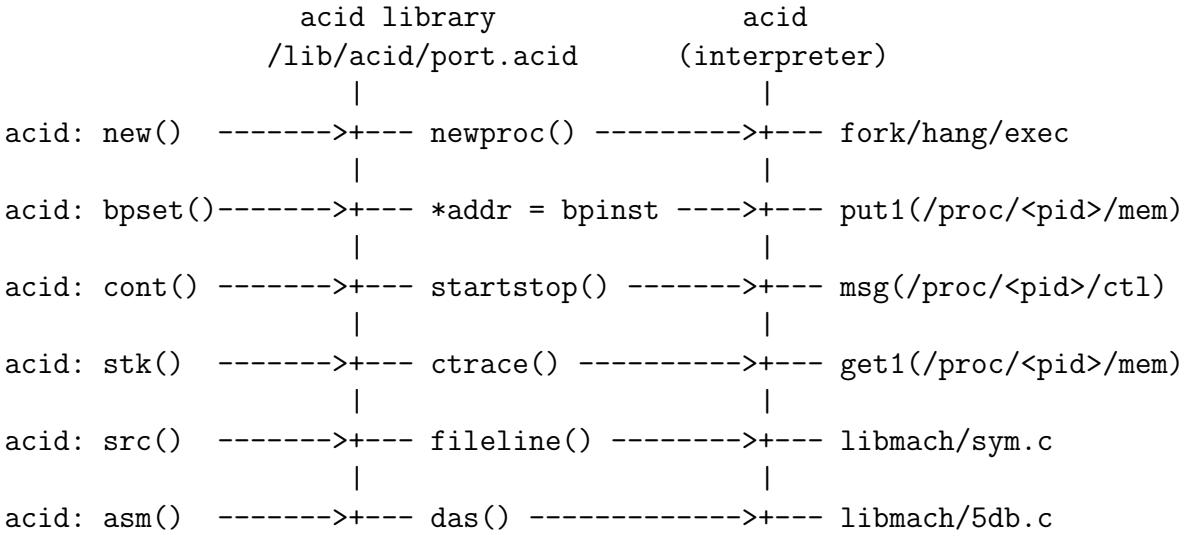


Figure 2.2: Data flow between `acid` library functions and the interpreter.

Chapter 3

Kernel Support

Much of the power of a debugger comes not from the debugger itself but from the kernel. The kernel provides the interface through which the debugger controls, inspects, and modifies the target process. In Plan 9, this interface is the `/proc` file system: each process is represented as a directory `/proc/<pid>/` containing files that expose process state and accept control messages.

Before looking at each file individually, here is the map of the `/proc/<pid>/` directory from the debugger's point of view—which files are control channels (write to command the target), which are data channels (read for state), and which are mixed:

```
/proc/<pid>/
+----- ctl        W    commands: hang, start, stop,
|                      waitstop, startstop, startsyscall
+----- status     R    name, state, user, wait chan
+----- mem        R/W  entire address space: text, data,
|                      bss, heap, stack (seek = vaddr)
+----- text       R    on-disk a.out (symbols, line table,
|                      SP-PC table; text+data segments)
+----- regs       R/W  general-purpose regs + PC + SP
+----- fpregs     R/W  floating-point regs
+----- segment    R    list of mapped segments (text,
|                      data, bss, stack, heap, shared)
+----- note       R/W  pending and posted notes (signals)
+----- fd         R    open file descriptors
+----- syscall    R    system call trace (used by ratrace)
+----- proc       R    per-thread info
+----- ns         R    namespace (mount table)
+----- kregs      R    kernel-mode saved registers
```

Roughly, `ctl` is how the debugger *drives* the target (one syscall per command, the kernel interprets the text), and all the other files are how it *observes* it. The split is deliberate: there is no single `ptrace()`-like fat syscall with a request enum; there is a small vocabulary of control words that any shell script or `acid` function can `echo` into `ctl`, plus a handful of regular files you `read` and `seek` through. Each section below walks through one of these files.

3.1 Process control: `/proc/<pid>/ctl`

The `/proc/<pid>/ctl` file is the main control channel. The debugger writes text messages to this file to control the target process:

- `hang`: tells the kernel to stop the process after its next `exec` system call. This is how the debugger intercepts a newly started child—it uses the classic fork/adjust/exec pattern, where the child writes `hang` to its own `ctl` before calling `exec`.
- `start`: resumes a stopped process. Used by `acid`'s `start()` builtin (called from the `cont()` library function when the user types `cont()` at the prompt) to resume execution after the debugger has finished inspecting state.
- `stop`: stops a running process. Used when the user wants to interrupt a long-running target—for instance, typing Ctrl-C in `acid` sends `stop` to halt the target without killing it.
- `waitstop`: blocks the writer (the debugger) until the target process enters the `Stopped` state—i.e., until the target hits a breakpoint, takes a fault, receives a signal, or completes a `startsyscall`/`hang` action. This provides synchronization between the debugging and debugged processes. For example, after `hang`/`exec` on a child, the debugger writes `waitstop` to block until the child has actually reached its first instruction.
- `startstop`: atomically starts the process and then waits for it to stop again. This is the core of the breakpoint cycle: resume the target until it hits a fault or breakpoint, then regain control. The atomicity matters: if the debugger did `start` then `waitstop` in two separate operations, the target could hit a breakpoint *between* the two writes, and the debugger would miss the stop event and block forever on `waitstop`.

The `ctl` file also provides synchronization: a write to `waitstop` will block until the target is actually stopped, so the debugger does not need to poll. This is crucial for correct operation—without atomic synchronization, the debugger might try to read registers while the target is still running.

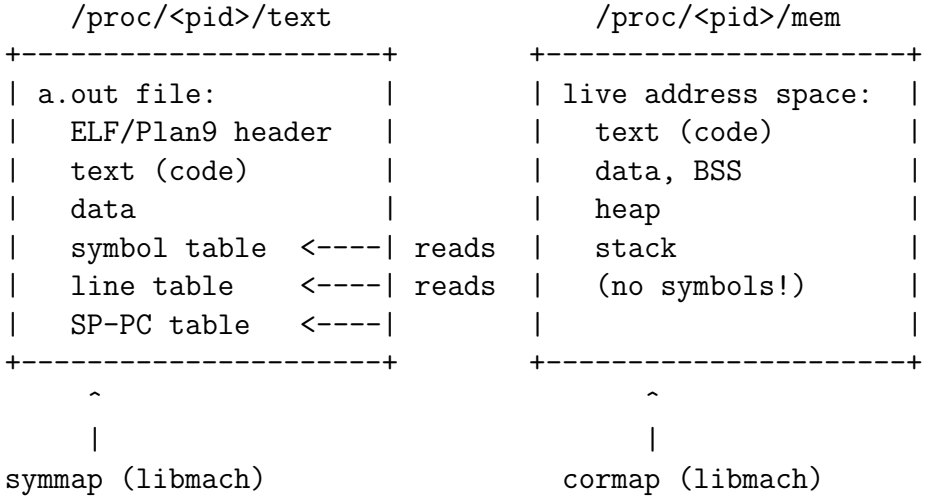
Here is a sequence diagram showing how `acid` starts a target program and continues it through a breakpoint:

```
    |------------------------>|                 |
    |<-- inspection results   |                 |
    |                         |                 |
```

The atomicity of `startstop` is what makes this safe: the kernel guarantees that the writer is blocked until the target stops again, with no window where a stop event could be lost.

3.2 Memory access: /proc/<pid>/mem

The `/proc/<pid>/mem` file gives the debugger direct access to the target process's entire address space: text (code), static data, BSS, heap, and stack. Reading and writing this file at a given offset accesses the corresponding virtual address in the target. This is how the debugger reads variable values, inspects the stack, and—crucially—writes breakpoint instructions into the target's code. There is also `/proc/<pid>/text`, which gives access to the on-disk executable file behind the running process. It is mostly a superset relationship from the debugger's point of view—except for one important thing: the symbol table and line number table live only in the executable file, not in the live process memory (only the text and data segments are loaded; the symbol table is discarded). So `acid` uses `/proc/<pid>/text` to read the symbols (via `libmach`), and `/proc/<pid>/mem` to read live process state:



3.3 Registers access: /proc/<pid>/regs

The `/proc/<pid>/regs` file exposes the target process's register state as a binary blob. The debugger reads it to inspect register values (program counter, stack pointer, general-purpose registers) and can write it to modify registers—for instance, to change the PC after handling a breakpoint. The layout of the register file is architecture-dependent; `libmach`'s `Reglist`[51c] structure describes the offset of each register within this blob.

Reading and writing another process's registers is something every debugger has to do, but the OS interface differs widely:

OS	read regs	write regs
\plan	read /proc/$pid/regs	write /proc/$pid/regs
Linux	ptrace(PTRACE_GETREGS)	ptrace(PTRACE_SETREGS)
Linux (modern)	ptrace(PTRACE_GETREGSET)	ptrace(PTRACE_SETREGSET)
FreeBSD	ptrace(PT_GETREGS, ...)	ptrace(PT_SETREGS, ...)
macOS / Mach	thread_get_state()	thread_set_state()
Windows	GetThreadContext()	SetThreadContext()

```
gdb remote (GDB)    ‘‘g’’ packet (hex blob)    ‘‘G’’ packet
```

Plan 9's choice is the simplest of these: the register state is a regular file. You can `cat /proc/$pid/regs | xd` from the shell, no debugger required, and a script that wants to nudge the PC can do it with `echo` and `seek` alone. The UNIX tradition multiplexes everything through one fat `ptrace` syscall whose request enum has grown to dozens of opcodes; Mach uses opaque thread ports; Windows packs everything into a giant `CONTEXT` struct whose layout is architecture-specific. The cost of Plan 9's everything-is-a-file approach is that the layout of the blob is undocumented in the file itself—you still need `libmach`'s `Reglist` to know which 4-byte slot is the PC. The benefit is that the mechanism composes with every other tool in the system: a crashed process is just a file you `grep` through. A typical example of writing the PC is the breakpoint cycle: when the target hits a breakpoint (e.g., `INT 3` on x86), the kernel reports the fault at PC = breakpoint address + 1 (because `INT 3` is one byte and PC has already advanced past it). The debugger needs to:

1. Restore the original instruction at the breakpoint address.
2. Decrement PC by 1 so that the now-restored instruction will re-execute when the process resumes.
3. Resume the process for one step (single-step), so the original instruction executes once.
4. Re-insert the breakpoint instruction so the breakpoint remains active for the next time.

Step 2 is where the debugger writes to `/proc/<pid>/regs` to modify the PC.

3.4 Breakpoint faulting instruction

Each architecture has a specific instruction that causes a trap when executed. On ARM, this is the `BKPT` instruction; on x86, it is `INT 3` (a single byte, `0xCC`). The `Machdata`[51e] structure stores the breakpoint instruction bytes in `bpinst` and its size in `bpsize`.

The encoding details vary considerably across ISAs, and the choice has consequences for `bpsize`:

ISA	mnemonic	encoding	size	notes
x86 / x64	INT3	0xCC	1 B	deliberately one byte so any instruction can be patched in place
ARM A32	BKPT	0xE1200070	4 B	fixed-width
ARM T32	BKPT	0xBE00	2 B	Thumb half-word
ARM64	BRK #0	0xD4200000	4 B	fixed-width
RISC-V	EBREAK	0x00100073	4 B	or 0x9002 (C.EBREAK)
MIPS	BREAK	0x0000000D	4 B	
PowerPC	TRAP	0x7FE00008	4 B	aka tw 31,0,0
SPARC	TA 1	0x91D02001	4 B	

The x86 `INT3` is the historical reason debuggers ever worked the way they do. Intel deliberately made `INT3` a *one-byte* instruction (0xCC) so a debugger could overwrite *any* instruction at *any* address with a breakpoint without disturbing the surrounding code, even if the original instruction was a single byte itself. On a fixed-width RISC like ARM, this is a non-issue: every instruction is already aligned to 4 bytes (or 2 in Thumb), so the breakpoint slot is the same width as anything you might patch over. For `acid`, `bpsize` is what tells `bpset()`[165a] how many bytes to save in `bplist` before overwriting, and how many to write back when `bpdel()`[165b] removes the breakpoint. On multi-byte architectures the save/restore must be atomic with respect to the running process: if the kernel happened to schedule the target between the save and the overwrite, it could fetch a half-patched instruction. Plan 9 side-steps this by requiring the process to be stopped (via `/proc/$pid/ctl`) before any

breakpoint operation. To set a breakpoint, the debugger saves the original instruction at the target address, then overwrites it with `bpinst` via `/proc/<pid>/mem`. When the target executes this address, the processor faults, the kernel stops the process, and the debugger regains control. In `acid`, the original instructions are not saved in a C-level global but in an `acid`-level list called `bplist`, maintained by the `bpset()`/`bpdel()` library functions in `/lib/acid/port.acid`. Each entry in `bplist` is a pair containing the breakpoint address and the original instruction byte(s). Storing it in `acid`'s own data structures (rather than in a hidden C global) is consistent with `acid`'s philosophy that debugger state should be inspectable and scriptable from the `acid` prompt.

3.5 Core dumps and broken processes

When a process crashes on Plan 9 (e.g., from a segmentation fault or divide error), the kernel does not kill it immediately. Instead, the process is placed in a `Broken` state, where it remains visible in `/proc` and can be inspected with `acid <pid>` (or `db <pid>`). This is much more convenient than traditional core dumps for several reasons:

- The full memory state is still live, including memory-mapped files, shared memory segments, and pages mapped from devices—none of which are typically saved in a core dump.
- Open file descriptors are still attached, so you can read `/proc/<pid>/fd` to see what files the process had open at the time of the crash.
- Threads are still in their stopped state, so you can inspect register values directly without reconstructing them from a saved blob.
- There is no disk space wasted writing a multi-megabyte core file (and no risk of `ulimit` truncating it).

You simply run `ps` to find the crashed process's PID, then attach the debugger.

Chapter 4

/bin/ratrace

Before diving into `acid`, it is worth looking at `ratrace` first. It is a much simpler program (about 300 lines of C), yet it already demonstrates the key kernel interfaces for process control: `/proc/<pid>/ctl` for stopping and starting the target, and `/proc/<pid>/syscall` for reading system call traces. The patterns we see here—fork/hang/exec, waitstop, startsyscall—will reappear in `acid`'s more complex process management code.

4.1 Entry point: `threadmain()`

`ratrace` uses the `libthread` library for concurrency (see LIBCORE book [Pad16b]). The entry point `threadmain()`[37] (used instead of `main()` because `ratrace` is a `libthread` program) processes the command-line arguments, then either forks a new child process (`ratrace -c cmd`) or attaches to an existing one (`ratrace pid`). It creates three channels for inter-thread communication, spawns the `writer()` thread, and then enters the `reader()` loop for the target process.

Why this thread architecture? When the traced program forks a child, `ratrace` needs to trace the child as well, which means running multiple `reader()` threads in parallel—one per traced process. To avoid interleaved output on stderr, all `reader()` threads forward their trace lines to a single `writer()` thread through the `out` channel; the writer is the only one that actually writes to stderr. The `forkc` channel lets readers notify the writer about new processes (so it can keep its reader count up to date), and `quit` signals when a reader is done. An alternative would be to use mutex-protected stderr writes, but channels match Plan 9's CSP-style concurrency model better and avoid the need for explicit locking.

4.1.1 `ratrace <pid>`

⟨*function* `threadmain` 37⟩≡ (371b)

```
void
threadmain(int argc, char **argv)
{
    int pid;
    ⟨threadmain() other locals 39a⟩

    ⟨threadmain() argv processing 39b⟩
    ⟨threadmain() if command given via -c 39c⟩
    else {
        if(argc != 2)
            usage();
        pid = atoi(argv[1]);
        //TODO? send a 'stop' to its ctl file?
    }

    out   = chancreate(sizeof(char*), 0);
```

```
    quit  = chancreate(sizeof(char*), 0);
    forkc = chancreate(sizeof(ulong *), 0);

    nread++;
    procrfork(writer, nil, Stacksize, 0);
    reader((void*)pid);
}
```

Uses Stacksize-3 38a, forkc 38d, nread 38b, out 38c, quit 38e, reader() 43, and writer() 41b.

⟨*enum* _anon_ (ratrace) 38a⟩≡ (371b)

```
enum {
    Stacksize   = 8*1024,
    Bufsize     = 8*1024,
};
```

⟨*global* nread 38b⟩≡ (371b)

```
int nread = 0;
```

Uses nread 38b.

4.1.2 Channels

The three channels implement a producer-consumer pattern: multiple `reader()` threads (one per traced process) send trace lines through `out`, signal new child processes through `forkc`, and announce their termination through `quit`. A single `writer()` thread listens on all three channels using `alt()` (a multiplexing receive), serializing the output and tracking the reader count. Why does the writer care about new children? Because the writer is also the program's lifecycle manager: it terminates `ratrace` when all `reader()` threads have exited (i.e., when `nread` reaches zero). Without `forkc`, the writer would not know that new readers had been spawned and might exit prematurely after the parent reader finishes, killing the children mid-trace.

⟨*global* out 38c⟩≡ (371b)

```
// chan<ref<string>> listener = writer(), sender = reader()
Channel *out;
```

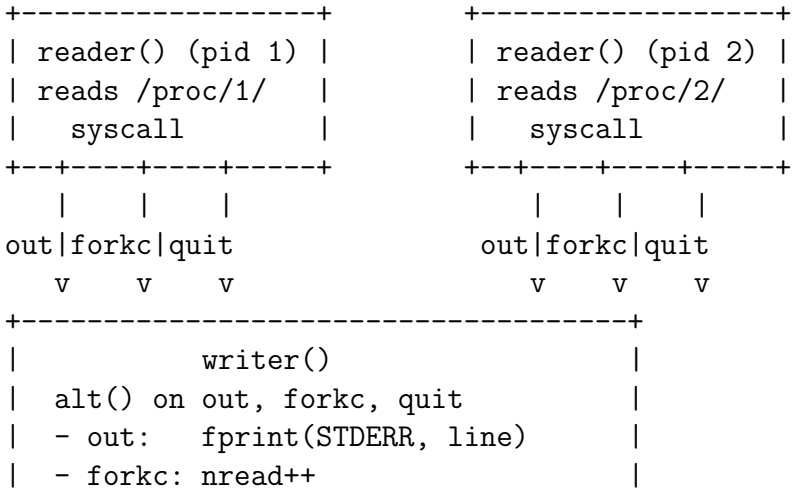
⟨*global* forkc 38d⟩≡ (371b)

```
// chan<pid> listener = writer(), sender = reader()
Channel *forkc;
```

⟨*global* quit 38e⟩≡ (371b)

```
// chan<nil> listener = writer(), sender = reader()
Channel *quit;
```

Here is the overall thread/channel architecture of `ratrace`:



```
|  - quit:  nread--, exit if zero     |
+-------------------------------------+
              |
              v
            STDERR
```

4.1.3 ratrace -c <cmd>

When `ratrace -c cmd` is used, `ratrace` forks a child process that will become the traced target. The child calls `hang()`[40] before `exec()`—this writes `hang` to the child's own `/proc/<pid>/ctl`, telling the kernel to stop the child after its `exec` completes. This is the same fork/adjust/exec pattern used by `acid`: the child adjusts its own state before `exec`, so the parent can intercept it at exactly the right moment.

⟨threadmain() *other locals* 39a⟩≡ (37)
```
char *cmd = nil;
char **args = nil;
```

⟨threadmain() *argv processing* 39b⟩≡ (37)
```
/*
 * don't bother with fancy arg processing, because it picks up options
 * for the command you are starting.  Just check for -c as argv[1]
 * and then take it from there.
 */
if (argc < 2)
    usage();
while (argv[1][0] == '-') {
    switch(argv[1][1]) {
    case 'c':
        if (argc < 3)
            usage();
        cmd = strdup(argv[2]);
        args = &argv[2];
        break;
    default:
        usage();
    }
    ++argv;
    --argc;
}
```

⟨threadmain() *if command given via* -c 39c⟩≡ (37)
```
/* run a command? */
if(cmd) {
    pid = fork();
    if (pid < 0)
        sysfatal("fork failed: %r");
    if(pid == 0) {
        hang();
        exec(cmd, args);
        // here when cmd didn't exist; try /bin/cmd is not given full path
        if(cmd[0] != '/')
            exec(smprint("/bin/%s", cmd), args);
        // should not be reached if the cmd exists
        sysfatal("exec %s failed: %r", cmd);
    }
}
```

Uses `hang()` 40.

The `hang()` function writes the `hang` message to the calling process's own `/proc/<pid>/ctl` file. After this, the next `exec` system call will cause the process to stop, giving the parent (the tracer) a chance to attach before any code in the new program runs.

⟨*function* `hang` 40⟩≡ (371b)

```
void
hang(void)
{
    fdt me;
    char *myctl;
    static char hang[] = "hang";

    myctl = smprint("/proc/%d/ctl", getpid());
    me = open(myctl, OWRITE);
    if (me < 0)
        sysfatal("can't open %s: %r", myctl);
    cwrite(me, myctl, hang, sizeof hang - 1);
    close(me);
    free(myctl);
}
```

Uses `cwrite()` 44c.

Here is what happens when `ratrace -c cmd` starts the target:

```
parent (ratrace)              child (will become target)
----------------              --------------------------
     fork()
       |---->----.
       |         |
       |         | open(/proc/<own_pid>/ctl)
       |         | write("hang") -> ctl
       |         | exec(cmd)
       |         |    (kernel stops child here)
       |
     reader()
       |
       | open(/proc/<child_pid>/ctl)
       | open(/proc/<child_pid>/syscall)
       | write("waitstop") -> ctl
       |    (blocks until child fully stopped)
       |
       | write("startsyscall") -> ctl
       |    (resumes child until next syscall)
       | read syscall trace
       |    -> send to writer via 'out'
       | write("startsyscall") -> ctl
       | ... loop ...
```

4.2 Output thread: `writer()`

The `writer()` thread serves two purposes: it serializes output from multiple reader threads so trace lines do not interleave, and it acts as a reference counter for active readers. When a reader finishes, it sends on `quit`; when all readers are done (`nread` reaches zero), the writer exits the program.

The `Str` structure is a small wrapper that bundles a buffer and its length. We need it because the `out` channel carries *pointers* to `Str` (not the buffer directly): a thread sends a pointer through the channel and the receiver reads at most one pointer-sized value. Without the wrapper, we would have to send length and buffer separately, requiring two channel operations or a more complex message structure. The buffer is allocated immediately after the struct in the same `malloc` (as we will see in `newstr()`[45b]), so freeing the `Str` also frees the buffer.

⟨*struct* `Str` 41a⟩≡ (371b)
```
struct Str {
    //ref_own<string> length = len, point to space after malloc'ed Str
    char *buf;
    int len;
};
```

The writer sets up an `Alt` array with three channels—quit, out, and forkc—and calls `alt()` in a loop. `alt()` is `libthread`'s multiplexing receive: it blocks until one of the channels has data and returns the index of the ready channel. Case 0 (quit) decrements the reader count; case 1 (out) prints a trace line to stderr; case 2 (forkc) increments the count for a newly spawned reader.

⟨*function* `writer` 41b⟩≡ (371b)
```
void
writer(void *)
{
    int newpid;
    Str *s;

    // TODO use literal array initializer?
    Alt a[4];

    a[0].op = CHANRCV;
    a[0].c = quit;
    a[0].v = nil;

    a[1].op = CHANRCV;
    a[1].c = out;
    a[1].v = &s;

    a[2].op = CHANRCV;
    a[2].c = forkc;
    a[2].v = &newpid;

    a[3].op = CHANEND;

    for(;;)
        switch(alt(a)){
        case 0:   /* quit */
            nread--;
            if(nread <= 0)
                goto done;
            break;
        case 1:   /* out */
            /* it's a nice null terminated thing */
            fprint(STDERR, "%s", s->buf);
            free(s);
            break;
        case 2:   /* forkc */
            // procrfork(reader, (void*)newpid, Stacksize, 0);
            nread++;
            break;
        }
```

```
done:
    exits(nil);
}
```

Uses forkc 38d, nread 38b, out 38c, and quit 38e.

4.3 Per-process reader thread: reader()

The reader() is the core of ratrace. It opens the target process's /proc/<pid>/ctl and /proc/<pid>/syscall files, then enters a loop: write startsyscall to ctl (which resumes the process and blocks until the next system call), then read the system call description from syscall, and send the result to the writer() through the out channel. The waitstop at the start ensures the child has actually stopped (after the hang/exec sequence) before we begin tracing.

To make the loop concrete, suppose ratrace has attached to an already-running target with ratrace 1234; the target is initially Stopped. (The ratrace -c echo hello mode is similar but starts with an extra fork+exec to launch the target before this loop begins.) Here is one full syscall cycle, showing the three actors and the two /proc/1234/ files they communicate through:

```
      |                 | systab[PWRITE] runs:
      |                 |   the kernel, still
      |                 |   in the target's
      |                 |   context, writes 6
      |                 |   bytes to fd 1
      |                 |   ret = 6
      |                 | sysretfmt() appends
      |                 |   " = 6 "" ...\n"
      |                 |   to up->syscalltrace
      |                 | target -> Stopped   | Stopped
      |                 |                     |  (in kernel
      |                 |                     |   mode, after
      |                 |                     |   Pwrite body)
  pread /proc/1234/syscall
      |---------------->|
      |<-- return record|
      |                 |                     |
  print to stderr                               |
      |                 |                     |
  write /proc/1234/ctl
    "startsyscall"
      |---------------->|
      |                 | ready(target)
      |                 |====== schedule ====>| kernel returns
      |                 |                     |  to user mode
      |                 |                     | Running
      |                 |                     | ... until the
      |                 |                     |   next syscall
  ... loop back to pread /proc/1234/syscall ...
```

Two things are worth noticing. First, the actual work of `Pwrite`—copying bytes out to the file—runs *in between* the two tracer stops, executed by the kernel's `systab` entry for `PWRITE` on behalf of the target. The target thread is suspended in kernel mode the whole time; it never gets back to user mode until both halves of the trace are done, which is why `ratrace`'s single `pread` loop iterates twice per traced syscall (once for the `syscallfmt` record, once for the `sysretfmt` record). Second, `ratrace` and the target never communicate directly: every arrow between the two outer columns goes through the kernel via `/proc/1234/ctl` and `/proc/1234/syscall`, matching the three-actor picture from Section 2.1.1.

⟨*function* `reader` 43⟩≡ (371b)

```
void
reader(void *v)
{
    char *ctl, *truss;
    fdt cfd, tfd;
    bool forking = false;
    bool exiting = false;
    int pid, newpid;
    Str *s;

    // /proc/<pid>/ctl messages
    static char waitstop[] = "waitstop";
    static char start[] = "start";

    pid = (int)(uintptr)v;
```

```
    ctl = smprint("/proc/%d/ctl", pid);
    if ((cfd = open(ctl, OWRITE)) < 0)
        die(smprint("%s: %r", ctl));

    truss = smprint("/proc/%d/syscall", pid);
    if ((tfd = open(truss, OREAD)) < 0)
        die(smprint("%s: %r", truss));

    /* child was stopped by hang msg earlier */
    cwrite(cfd, ctl, waitstop, sizeof waitstop - 1);
    // useful? if it was stopped, then why need waitstop?
    // because the fork() has been done but maybe the child
    // has not yet reached the exec() and got actually stopped ?
    // and also for 'ratrace pid' case ?

    cwrite(cfd, ctl, "startsyscall", 12);

    s = newstr();

    while((s->len = pread(tfd, s->buf, Bufsize - 1, 0)) >= 0){

        ⟨reader() if forking and special condition 46⟩

        ⟨reader() if Rfork syscall 45c⟩
         else
            ⟨reader() if Exits syscall 44a⟩

        sendp(out, s); /* print line from /proc/$child/syscall */

        ⟨reader() if exiting 44b⟩

        /* flush syscall trace buffer */
        cwrite(cfd, ctl, "startsyscall", 12);
        s = newstr();
    }

    sendp(quit, nil);
    threadexitsall(nil);
}
```

Uses Bufsize-4 38a, cwrite() 44c, newstr() 45b, out 38c, and quit 38e.

⟨reader() *if Exits syscall* 44a⟩≡ (43)

```
if (strstr(s->buf, " Exits") != nil)
  exiting = true;
```

⟨reader() *if* exiting 44b⟩≡ (43)

```
if (exiting) {
    s = newstr();
    strcpy(s->buf, "\n");
    sendp(out, s);
    break;
}
```

Uses newstr() 45b and out 38c.

`cwrite()`[44c] is a wrapper around `write()` that checks for errors and terminates the reader thread if the write fails (e.g., because the traced process has exited).

⟨*function* cwrite 44c⟩≡ (371b)

```
void
```

```
cwrite(fdt fd, char *path, char *cmd, int len)
{
    werrstr("");
    if (write(fd, cmd, len) < len) {
        fprint(STDERR, "cwrite: %s: failed writing %d bytes: %r\n",
            path, len);
        sendp(quit, nil);
        threadexits(nil);
    }
}
```

Uses quit 38e.

⟨*function* die 45a⟩≡ (371b)

```
void
die(char *s)
{
    fprint(STDERR, "%s\n", s);
    exits(s);
}
```

`newstr()`[45b] allocates a `Str` with the buffer placed immediately after the struct in a single allocation (the `s->buf = (char *)&s[1]` trick). This avoids a separate `malloc` for the buffer and ensures the buffer is freed with the struct.

⟨*function* newstr 45b⟩≡ (371b)

```
Str *
newstr(void)
{
    Str *s;

    s = mallocz(sizeof(Str) + Bufsize, true);
    if (s == nil)
        sysfatal("malloc");
    s->buf = (char *)&s[1];
    return s;
}
```

Uses Bufsize-4 38a.

4.4 Tracing forked children

When the traced process forks a child (via `Rfork` with the `RFPROC` flag), `ratrace` needs to start tracing the child as well. The reader detects this by looking for `Rfork` in the syscall trace output, checking that the `RFPROC` flag is set. On the next read, the return value gives the new child's PID, which is sent to the writer via `forkc`, and a new reader thread is spawned for the child. Why route through `forkc` rather than letting the new reader update `nread` directly? Because `nread` is owned by the `writer()` thread—it is the one that reads and modifies it (in case 0 of its `alt()` loop). Letting the new reader write to `nread` from a different thread would be a race condition. The `forkc` channel serializes the increment through the writer, keeping all updates to `nread` in one thread without needing locks. This is the standard CSP discipline: shared state is owned by one process, others communicate by message-passing.

⟨`reader()` *if Rfork syscall* 45c⟩≡ (43)

```
/*
 * There are three tests here and they (I hope) guarantee
 * no false positives.
 */
if (strstr(s->buf, " Rfork") != nil) {
    char *a[8];
```

```
        char *rf;

        rf = strdup(s->buf);
        if (tokenize(rf, a, 8) == 5 &&
            strtoul(a[4], 0, 16) & RFPROC)
            forking = true;
        free(rf);
    }
```

⟨`reader()` *if* `forking` *and special condition* 46⟩≡ (43)

```
    // ??
    if (forking && s->buf[1] == '=' && s->buf[3] != '-') {
        forking = false;
        newpid = strtol(&s->buf[3], 0, 0);
        sendp(forkc, (void*)newpid);
        procrfork(reader, (void*)newpid, Stacksize, 0);
    }
```

Uses `Stacksize-3` 38a, `forkc` 38d, and `reader()` 43.

Chapter 5

Toolchain Support

A debugger needs metadata generated by the toolchain to provide a good user experience. Without symbols, the debugger can only show raw addresses; without line numbers, it cannot display source code. This chapter briefly reviews what each stage of the toolchain contributes. For full details, see the Assembler book [Pad15a], Linker book [Pad15c], and Compiler book [Pad16a].

5.1 Assembler metadata

The assembler (`5a` for ARM, `8a` for x86) records a symbol table in each object file. Each symbol has a name, a type (text, data, BSS, auto, param), and a value (address or stack offset). Local variables and function parameters are recorded with type codes `a` (auto) and `p` (param), along with their offset from the frame pointer. The assembler also generates `z`-type entries that record source file names and line number history, which the debugger uses to map addresses back to source positions.

For example, given a tiny assembly source file `hello.s`:

```
TEXT main(SB), $8
    MOVL  x+0(FP), AX
    ADDL  $1, AX
    MOVL  AX, .ret+0(FP)
    RET
```

the assembler emits these symbol entries (which the linker will relocate to their final addresses):

```
value     type  name
--------  ----  ----------
0x1020    T     main      <- text symbol (entry address)
0x0008    m     .frame    <- frame size for unwinding
0x0000    p     x         <- parameter at FP+0
0x0001    z     hello.s   <- file/line history entry
```

Each function gets one `T` entry (its address), one `m .frame` entry (its frame size), and `a`/`p` entries for each local/parameter with its frame-pointer offset. The `z` entries form a sequence that the debugger replays to reconstruct the original `#include` hierarchy and the exact source line for any address.

5.2 Linker metadata

The linker (`5l` for ARM, `8l` for x86) combines the symbol tables from all object files into a single table in the executable. It also generates two important tables for the debugger: the SP-PC table (mapping program

counter values to stack pointer offsets, used for stack unwinding) and the line number-PC table (mapping program counter values to source line numbers, used by `src()` and `pcline()`). These tables are stored in the executable header, whose offsets are recorded in the `Fhdr` structure.

The two linker-generated tables look like this (continuing the `hello.s` example from above, with one extra function `bar`):

SP-PC table

PC	SP offset
0x1020	0
0x1023	8
0x102a	8
0x102c	0

Line number-PC table

PC	file	line
0x1020	hello.c	3
0x1023	hello.c	4
0x102a	hello.c	5
0x102c	hello.c	9

The SP-PC table tells the debugger "at PC `0x1023`, the stack pointer has been decremented by 8 bytes from its value at function entry"—essential for `stk()` to walk back through caller frames. The line number-PC table powers `src(*PC)` and breakpoint-on-line operations. Both tables are stored compactly using delta encoding (only differences are stored), so they take little space even for large programs. Their offsets in the executable are recorded in the `Fhdr` structure.

5.3 Compiler metadata

The compiler (`5c` for ARM, `8c` for x86) does not write symbol entries directly to the object file. Instead, it emits pseudo-instructions (`ANAME`) interleaved with real code in its assembly-like intermediate output, and the linker (`8l`) translates these into the final symbol table entries (`a`, `p`, `m`) we saw above. So the compiler is the *source* of local variable and parameter information, but the linker is what *writes* it to the final executable. This is unlike the assembler, which writes its symbol entries directly to the object file in their final form.

The C compiler also has a special feature for `acid`: the `-a` flag (implemented in `cc/acid.c`) makes the compiler output `acid` type definitions (using the `complex/aggr` syntax) for every C struct and union. This lets `acid` understand the layout of data structures: you just run `8c -a foo.c > foo.acid` and then load `foo.acid` into your `acid` session. This is functionally similar to DWARF debug information used by `gdb` and `lldb`, but much simpler. DWARF is more complete: it encodes type definitions, variable lifetimes, inlined function boundaries, optimized-out variables, and even macro definitions. Plan 9's approach is closer to a "poor man's DWARF"—it covers struct/union/enum layout but not lifetime or location information for optimized variables. The trade-off is simplicity: DWARF parsers are notoriously complex (>10K LOC in `libdwarf`), while `acid`'s type system is just a few hundred lines.

Chapter 6

Core Data Structures

This chapter presents the data structures used by `acid`, `libmach`, and the kernel's `/proc` interface. There are two broad categories: the `libmach` types that describe executables, machines, and memory maps (shared with `db` and the emulators), and the `acid`-specific types for the interpreter (AST nodes, values, symbol table, garbage collector).

6.1 `libmach` types and globals

The `libmach` library provides an architecture-independent interface for reading executables, accessing process memory, looking up symbols, and disassembling instructions.

6.1.1 Executable format: `Fhdr`

The `Fhdr` structure holds the parsed header of an executable file. It contains everything the debugger needs to locate code, data, and metadata within the binary: the text and data segment addresses and offsets, the symbol table location and size, and the line number and SP-PC table locations. The `type` field identifies the architecture (ARM, x86, MIPS) and whether this is a regular executable or a bootable kernel image.

⟨*struct* `Fhdr` 49⟩≡ (304c)

```
/*
 * Common a.out header describing all architectures
 */
struct Fhdr
{
    char *name;  /* identifier of executable */

    // enum<executable_type>
    byte type;  /* file type - see codes above */

    byte hdrsz;  /* header size */
    byte _magic;  /* _MAGIC() magic */
    byte spare;

    long magic;  /* magic number */

    uvlong txtaddr; /* text address */
    vlong txtoff;  /* start of text in file */

    uvlong dataddr; /* start of data segment */
    vlong datoff;  /* offset to data seg in file */

    vlong symoff;  /* offset of symbol table in file */
```

```
    uvlong entry;  /* entry point */

    vlong sppcoff; /* offset of sp-pc table in file */
    vlong lnpcoff; /* offset of line number-pc table in file */

    long txtsz;  /* text size */
    long datsz;  /* size of data seg */
    long bsssz;  /* size of bss */

    long symsz;  /* size of symbol table */
    long sppcsz;  /* size of sp-pc table */ // unused
    long lnpcsz;  /* size of line number-pc table */

};
```

⟨*enum* executable_type 50a⟩≡ (304c)

```
/* types of executables */
enum executable_type
{
    FNONE = 0,  /* unidentified */

    FI386,   /* 8.out */
    FI386B,   /* I386 bootable */
    FARM,   /* 5.out */
    FARMB,   /* ARM bootable */
    FMIPS,   /* v.out */
    FMIPSB,   /* mips bootable */
    FMIPS2BE,          /* 4.out */
};
```

6.1.2 Architecture description: `Mach` and `mach` (ARM)

The `Mach` structure describes a processor architecture: its register set, register file size, well-known register names (PC, SP, link register, static base), page size, memory layout constants, and address/register sizes. The global `mach` points to the current architecture's `Mach` instance, set when the executable is parsed by `crackhdr()`[90].

⟨*struct* Mach 50b⟩≡ (304c)

```
struct Mach{
    char *name;
    // enum<machine_type>
    int mtype;   /* machine type code */

    Reglist *reglist;  /* register set */
    long regsize;  /* sizeof registers in bytes */
    long fpregsize;  /* sizeof fp registers in bytes */

    char *pc;   /* pc name */
    char *sp;   /* sp name */
    char *link;   /* link register name */
    char *sbreg;   /* static base register name */
    uvlong sb;   /* static base register value */

    int pgsize;   /* page size */
    uvlong kbase;   /* kernel base address */
    uvlong ktmask;   /* ktzero = kbase & ~ktmask */
    uvlong utop;   /* user stack top */

    int pcquant;  /* quantization of pc */
```

```
    int szaddr;   /* sizeof(void*) */
    int szreg;   /* sizeof(register) */
    int szfloat;  /* sizeof(float) */
    int szdouble;  /* sizeof(double) */
};
```

⟨*global* mach 51a⟩≡ (321e)
```
Mach *mach = &mi386;   /* Global current machine table */
```
Uses mach 51a and mi386.

⟨*enum* machine_type 51b⟩≡ (304c)
```
/* machine types */
enum machine_type
{
    MI386,
    MARM,
    MMIPS,
};
```

⟨*struct* Reglist 51c⟩≡ (304c)
```
/*
 * machine register description
 */
struct Reglist {
    char *rname;  /* register name */

    short roffs;  /* offset in u-block */

    // bitset<enum<register_flag>>
    char rflags;  /* INTEGER/FLOAT, WRITABLE */
    char rformat; /* print format: 'x', 'X', 'f', '8', '3', 'Y', 'W' */
};
```

The `rflags` field is a bitset describing each register's type and access permissions: `RFLT` marks floating-point registers (the default `RINT` bit is zero for integer registers), and `RRDONLY` marks registers the debugger must not modify (e.g., the program counter on architectures where it cannot be assigned directly, or hard-wired zero registers like ARM's `PC` in some modes).

⟨*enum* register_flag 51d⟩≡ (304c)
```
enum {     /* bits in rflags field */
    RINT = (0<<0),
    RFLT = (1<<0),

    RRDONLY = (1<<1),
};
```

6.1.3 Architecture-specific operations: `Machdata` and `machdata` (ARM)

While `Mach` describes static properties of an architecture, `Machdata` provides the architecture-specific *methods* that the debugger needs: the breakpoint instruction bytes (`bpinst`), byte-swapping functions for cross-architecture debugging, the C stack tracer (`ctrace`), the disassembler (`das`), and the follow-set calculator (`foll`—given a PC, what are the possible next PCs?). The global `machdata` points to the current architecture's instance.

⟨*struct* Machdata 51e⟩≡ (304c)
```
struct Machdata {  /* Machine-dependent debugger support */
    uchar bpinst[4];   /* break point instr. */
```

```
    short bpsize;    /* size of break point instr. */

    ushort (*swab)(ushort);  /* ushort to local byte order */
    ulong (*swal)(ulong);   /* ulong to local byte order */
    uvlong (*swav)(uvlong);  /* uvlong to local byte order */

    int (*ctrace)(Map*, uvlong, uvlong, uvlong, Tracer); /* C traceback */
    uvlong (*findframe)(Map*, uvlong, uvlong, uvlong, uvlong);/* frame finder */
    char* (*excep)(Map*, Rgetter); /* last exception */
    ulong (*bpfix)(uvlong);  /* breakpoint fixup */

    int (*sftos)(char*, int, void*); /* single precision float */
    int (*dftos)(char*, int, void*); /* double precision float */

    int (*foll)(Map*, uvlong, Rgetter, uvlong*);/* follow set */

    int (*das)(Map*, uvlong, char, char*, int); /* symbolic disassembly */
    int (*hexinst)(Map*, uvlong, char*, int);  /* hex disassembly */
    int (*instsize)(Map*, uvlong); /* instruction size */
};
```

The `armmach` structure is the ARM instance of the `Machdata` vtable described in Chapter 6. It wires up the disassembler, follow-set, and stack-tracing functions so that `acid` can call them through the `machdata` pointer without knowing which architecture it is debugging. Note `bpinst`: the ARM `BKPT` instruction is `0xE1200070` (4 bytes, stored little-endian as `70 00 20 E1`).

⟨*global* `armmach` *(arm)* 52a⟩≡ (309b)

```
/*
 *      Debugger interface
 */
Machdata armmach =
{
  .bpinst= {0x70, 0x00, 0x20, 0xE1},         /* break point */       /* E1200070 */
  .bpsize= 4,                   /* break point size */

  .swab= leswab,                      /* short to local byte order */
  .swal= leswal,                      /* long to local byte order */
  .swav= leswav,                      /* long to local byte order */

  .ctrace=  risctrace,          /* C traceback */
  .findframe=  riscframe,          /* Frame finder */

  .excep= armexcep,                    /* print exception */
  .bpfix= nil,                  /* breakpoint fixup */

  .sftos= nil,                  /* single precision float printer */
  .dftos= nil,                  /* double precision float printer */

  .foll= armfoll,            /* following addresses */
  .das=  arminst,            /* print instruction */
  .hexinst=  armdas,             /* dissembler */
  .instsize=  arminstlen,         /* instruction size */
};
```

Uses `armdas()` 211d, `armexcep()` 52b, `armfoll()` 233, `arminst()` 211a, `arminstlen()` 211c, `leswab()` 316a, `leswal()` 316b, `leswav()` 316c, `riscframe()` 362b, and `risctrace()` 361.

⟨*function* `armexcep` *(arm)* 52b⟩≡ (309b)

```
static char*
armexcep(Map *map, Rgetter rget)
{
```

```
    uvlong c;

    c = (*rget)(map, "TYPE");
    switch ((int)c&0x1f) {
    case 0x11:
        return "Fiq interrupt";
    case 0x12:
        return "Mirq interrupt";
    case 0x13:
        return "SVC/SWI Exception";
    case 0x17:
        return "Prefetch Abort/Breakpoint";
    case 0x18:
        return "Data Abort";
    case 0x1b:
        return "Undefined instruction/Breakpoint";
    case 0x1f:
        return "Sys trap";
    default:
        return "Undefined trap";
    }
}
```

6.1.4 Memory mapping abstraction: `Map`

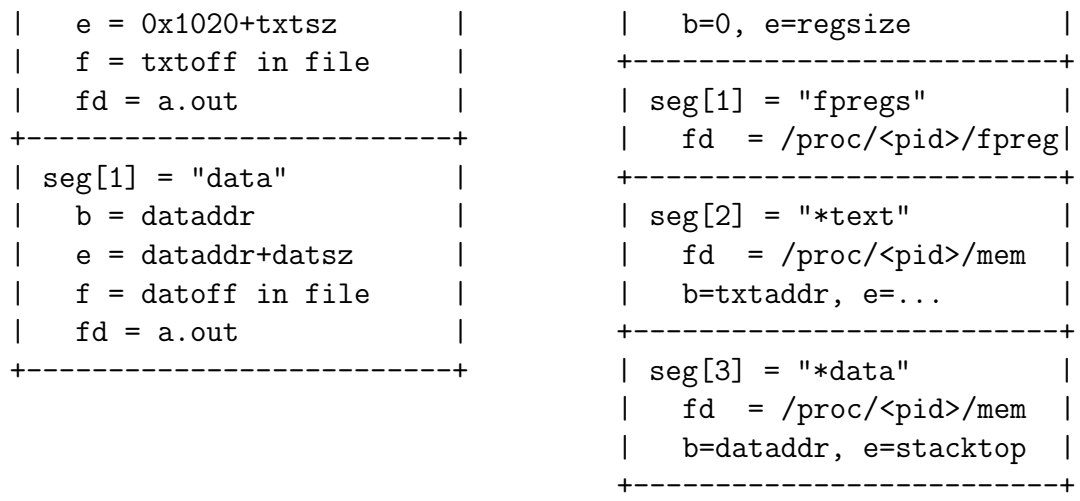
A `Map` is the key abstraction for memory access. It maps address ranges to file descriptors and offsets, so that reading address `0x1000` in the map translates to reading at the right offset in the right file. There are two main maps: `symmap`[55a] maps the executable file's text and data segments (for reading symbols and code from the binary), and `cormap`[55b] maps the live process's memory through `/proc/<pid>/mem` and register files. This abstraction lets the debugger use the same code to read from a file or from a running process.

⟨*struct* `Map` 53⟩≡ (304c)

```
/*
 *  Structure to map a segment to a position in a file
 */
struct Map {
    int nsegs;   /* number of segments */

    struct segment {  /* per-segment map */
        char *name;  /* the segment name */
        fdt fd;  /* file descriptor */

        bool inuse;  /* in use - not in use */
        bool cache;  /* should cache reads? */

        uvlong b;  /* base */
        uvlong e;  /* end */
        vlong f;  /* offset within file */

    } seg[1];   /* actually n of these */
};
```

Here is what the two main maps look like for a typical debugging session:

```
symmap (executable file)              cormap (live process)
+-------------------------+          +-------------------------+
| seg[0] = "text"         |          | seg[0] = "regs"         |
|   b = 0x1020            |          |   fd  = /proc/<pid>/regs |
```



`acid` reads symbols and code through `symmap` (the on-disk file), and reads/writes live process state through `cormap` (the `/proc` files). The `*` prefix on `*text`/`*data` distinguishes the live segments from the file-based ones with the same name in `symmap`; this distinction matters because the live text may differ from the file text after the debugger inserts breakpoints.

6.1.5 Symbol entries: `Symbol`

A `Symbol` is the result of a symbol table lookup. It contains the symbol's name, its value (an address for code and data symbols, a stack offset for locals), and its type and class. The type codes come from the assembler/linker conventions: `T`/`t` for text, `D`/`d` for data, `B`/`b` for BSS, `a` for autos, `p` for parameters. The capitalization distinguishes *global* from *static (file-local)* symbols: `T` is a globally visible function, `t` is a `static` function. Same for `D` vs `d` (global data vs static data) and `B` vs `b` (BSS). This convention comes from the original UNIX`a.out` format and is preserved by the Plan 9 toolchain. The debugger usually treats both cases the same way for display, but the linker uses the distinction to scope references during linking.

⟨*struct* `Symbol` 54a⟩≡ (304c)

```
/*
 * Internal structure describing a symbol table entry
 */
struct Symbol {
    void  *handle;  /* used internally - owning func */

    struct {
        char *name;
        vlong value;  /* address or stack offset */
        char type;  /* as in a.out.h */
        char class;  /* as above */
        int index;  /* in findlocal, globalsym, textsym */
    };
};
```

⟨*enum* `symbol_type` 54b⟩≡ (304c)

```
/* symbol table classes */
enum symbol_type
{
    CNONE  = 0,

    CAUTO,
    CPARAM,
    CSTAB,
    CTEXT,
```

```
    CDATA,

    CANY,   /* to look for any class */
};
```

6.1.6 Raw symbol records: `Sym`

The `Sym` type is defined in `include/a.out.h` that is included by libmach `mach.h`.

```
struct Sym
{
    vlong value;
    uint sig;
    char type;
    char *name;
};
```

`Sym` and `Symbol` are easy to confuse but play different roles:

- `Sym` is the raw on-disk record: just a value, a type character, and a name string. It is what `syminit()`[94] reads from the executable file into the `symbols[]` array. `Sym` is internal to `libmach/sym.c`.
- `Symbol` is the cooked structure returned to clients of `libmach` (like `acid`). It carries the same fields plus a `class` (CAUTO, CPARAM, CTEXT, ...), an `index`, and an opaque `handle` used by `libmach` to track the parent function for locals.

The lookup functions (`lookup()`[331], `findlocal()`[332c], `textsym()`[333b]) take a `Sym` from the array and "fill" a caller-provided `Symbol` via `fillsym()`. Clients only ever see `Symbol`; `Sym` is an implementation detail.

6.2 `acid`, `libmach`, and `/proc`

`acid` uses two `Map`[53] objects to access the target: `symmap`[55a] for the executable file (symbols and static code) and `cormap`[55b] for the live process (memory, registers). These are set up during initialization and process attachment (see `readtext()`[74d] and `sproc()`[85b])

6.2.1 `/proc/<pid>/text` and `symmap`

⟨*global symmap (acid/globals.c)* 55a⟩≡ (381a)
```
Map* symmap;
```

6.2.2 `/proc/<pid>/mem` and `cormap`

⟨*global cormap (acid/globals.c)* 55b⟩≡ (381a)
```
Map* cormap;
```

6.2.3 Process table: ptab

acid can debug multiple processes simultaneously (e.g., parent and child after a fork). The ptab[56b] array maps PIDs to open /proc/<pid>/ctl file descriptors. The msg()[56d] function looks up a PID in this table and writes a control message to the corresponding ctl fd. install()[83b] adds an entry, deinstall()[84] removes one. The same machinery is what lets acid debug multi-threaded programs. In Plan 9, there are no threads as a separate kernel concept: a multi-threaded program is just a group of processes created with rfork(RFPROC|RFMEM), which share their address space but each have their own pid, stack, and entry in /proc/. Linux takes essentially the same view under the hood—clone(CLONE_VM|...) creates a new task that shares memory with its parent, and each thread has its own tid visible in /proc/<pid>/task/. Because every thread is an independent entry in /proc/, acid debugs threads exactly the way it debugs separate processes: install each one in ptab, send stop/start/waitstop to its own ctl file, and read its registers from its own regs file. No thread-specific API is needed.

⟨*struct* Ptab 56a⟩≡ (378b)
```
struct Ptab
{
    int pid;
    fdt ctl;
};
```

⟨*global* ptab 56b⟩≡ (381a)
```
// hash<pid, fdt>
Ptab ptab[Maxproc];
```
Uses Maxproc 56c.

⟨*acid constants* 56c⟩≡ (378a) 66b▷
```
Maxproc  = 50,
```

6.2.4 Sending control messages: msg()

⟨*function* msg 56d⟩≡ (392a)
```
void
msg(int pid, char *msg)
{
    int i;
    int l;
    char err[ERRMAX];

    for(i = 0; i < Maxproc; i++) {

        if(ptab[i].pid == pid) {
            l = strlen(msg);
            if(write(ptab[i].ctl, msg, l) != l) {
                errstr(err, sizeof err);
                if(strcmp(err, "process exited") == 0)
                    deinstall(pid);
                error("msg: pid=%d %s: %s", pid, msg, err);
            }
            return;
        }
    }
    error("msg: pid=%d: not found for %s", pid, msg);
}
```
Uses Maxproc 56c, deinstall() 84, and ptab 56b.

6.3 Tokens

We now leave `libmach`'s data structures behind and turn to `acid`'s own. The remaining types in this chapter (tokens, AST nodes, runtime values, symbol table, process table) are all internal to the `acid` interpreter and exist only in `acid/` source files.

The `acid` language has a C-like syntax with a few additions: `head`/`tail`/`append` for list manipulation, `defn` for function definitions, `loop` for counted iteration, `complex`/`aggr` for type descriptions, `whatis` for introspection, and `eval` for dynamic evaluation. The token declarations below are used by the yacc-generated parser.

⟨*token declarations* 57a⟩≡ (112b)

```
%token <sym> Tid
%token <ival> Tconst
%token <fval> Tfconst
%token <string> Tstring

%token <ival> Tfmt

%token Tif Tdo Tthen Telse Twhile  Tloop
%token Thead Ttail Tappend
%token Tfn Tret Tlocal
%token Tcomplex Twhat Tdelete Teval Tbuiltin
```

⟨*union yacc* 57b⟩≡ (112b)

```
%union
{
  uvlong  ival;
  float   fval;
  String* string;

  Lsym *sym;
  ⟨union yacc other fields 112c⟩
}
```

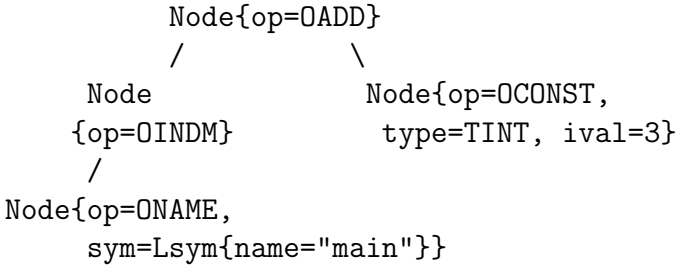
6.4 Abstract syntax tree

Like many interpreters, `acid` parses input into an abstract syntax tree (AST) and then walks it to evaluate expressions and execute statements. Each AST node is a `Node`[378b] with an `Opcode`, left and right children, and optional extra fields for the symbol, type, and literal value. Note that nodes are garbage-collected (they embed a `Gc`[378b] header).

For example, the `acid` expression

```
*main + 3
```

parses into the following tree:



The yacc grammar rules build this tree by calling `an()`[58c] bottom-up: first the `ONAME` leaf, then the `OINDM` unary, then the `OCONST` for the literal, and finally the `OADD` root that joins them. The interpreter (`execute()`[120a]) later walks the tree top-down, dispatching on each node's opcode through the `expop`[121f] table.

6.4.1 AST nodes: `Node`

The `Node` structure serves double duty: it represents both AST nodes (during parsing and execution) and temporary results (during expression evaluation, where `op` is set to `OCONST` and the embedded `Store` union holds the actual integer, float, string, or list value). The "value fields" are not direct members of `Node`: they live inside the embedded `Store` struct (added via the kencc anonymous struct feature, see chunk `<<{\tt{}Node} other fields>>` below). Thanks to this embedding, you can write `n->ival` or `n->fval` directly on a `Node` just like on a `Value`.

⟨*struct* `Node` 58a⟩≡ (378b)
```
struct Node
{
    ⟨Node first gc field 64h⟩

    // enum<opcode>
    char op;
    Node* left;
    Node* right;

    ⟨Node other fields 58d⟩
};
```

⟨*constant* `ZN` 58b⟩≡ (378b)
```
#define ZN (Node*)0
```

The `an()`[58c] function allocates a new node and adds it to the garbage collector's linked list.

⟨*function* `an` 58c⟩≡ (383c)
```
Node*
an(int op, Node *l, Node *r)
{
    Node *n;

    n = gmalloc(sizeof(Node));
    memset(n, 0, sizeof(Node));

    ⟨an() add node n to gcl 64e⟩

    n->op = op;
    n->left = l;
    n->right = r;

    return n;
}
```
Uses `gmalloc()` 151b.

⟨Node *other fields* 58d⟩≡ (58a) 65b▷
```
// enum<Type_kind> ?
char type;

// ??
int builtin;

// ??
Store;
```

6.4.2 Opcodes: `Opcode`

The opcodes cover the full range of `acid` operations: arithmetic (`OADD`, `OSUB`, ...), comparisons, logical operators, assignment, control flow (`OIF`, `OWHILE`, `ODO`, `ORET`), function calls (`OCALL`), list operations (`OHEAD`, `OTAIL`,

OAPPEND), memory indirection (OINDM for * dereference through cormap, OINDC for @ dereference through the text), and acid-specific features like OCOMPLEX for type declarations and OEVAL for dynamic evaluation.

⟨*enum* _anon_ (acid/acid.h)*3* 59⟩≡ (378b)

```
enum Opcode
{
    ONAME,
    OCONST,
    OLIST,

    OADD,
    OSUB,
    OMUL,
    ODIV,
    OMOD,

    ORSH,
    OLSH,

    OLT,
    OGT,
    OLEQ,
    OGEQ,
    OEQ,
    ONEQ,

    OLAND,
    OXOR,
    OLOR,
    ONOT,

    OCAND,
    OCOR,

    OEDEC,
    OEINC,
    OPINC,
    OPDEC,

    OIF,
    OELSE,
    OWHILE,
    ODO,
    ORET,

    OASGN,
    OINDM,
    OCALL,
    OINDEX,
    OINDC,
    ODOT,
    OCAST,

    OHEAD,
    OTAIL,
    OAPPEND,
    ODELETE,

    OLOCAL,
    OFRAME,
```

```
    OCTRUCT,
    OCOMPLEX,

    OFMT,
    OEVAL,
    OWHAT,
};
```

We will see the generation of these opcodes in Chapter 9 (the yacc grammar rules build `Node`s with the appropriate `op`) and their interpretation in Chapter 10 (the `expop` dispatch table maps each opcode to its evaluator function).

6.5 Type system

`acid` has a simple dynamic type system with five types: integers, floats, strings, lists, and code (unevaluated expressions passed by reference). Every `acid` value carries a type tag that is checked at runtime during expression evaluation.

⟨*enum* `_anon_ (acid/acid.h)2` 60a⟩≡ (378b)

```
enum Type_kind
{
    TINT,
    TFLOAT,

    TSTRING,
    TLIST,

    TCODE,
};
```

The five-type runtime `Type_kind` above is for `acid` *values*—what arithmetic and assignment operate on. The `Type` struct below is something else entirely: it describes a *C-level* aggregate (a struct or union from the program being debugged), so that `acid` can walk the fields of a `Proc` or a `Pgrp` from the running kernel and print them by name. Each `Type` is one row in a field list; multiple rows are linked through `next` to form the full layout of an aggregate. The interesting fields are:

- `offset`: byte offset of this field within the parent struct (the value the C compiler hardcoded into `loadword` instructions);
- `fmt`: a single-character format spec (`X` for hex, `D` for decimal, `s` for string, etc.) telling `acid` how to print the raw bytes;
- `depth`: pointer-indirection depth, so `fmt='X'`, `depth=2` means "pointer-to-pointer-to-hex";
- `type` / `tag` / `base`: links into the `Lsym` symbol table that name the field, the parent aggregate, and (for nested aggregates) the inner type.

The split between value-level `Type_kind` and field-level `Type` is what lets `acid` type-check its own scripting language while still letting the user write `complex Proc 'X' 0 p; 'X' 4 next;` and then say `print(*((Proc)addr)` without ever rebuilding the debugger. The C type description is data inside `acid`, not code.

⟨*struct* `Type` 60b⟩≡ (378b)

```
struct Type
{
    Type* next;
```

```
    int offset;
    char fmt;
    char depth;

    Lsym* type;
    Lsym* tag;
    Lsym* base;
};
```

6.6 Runtime values

After the AST and types, we now look at the data structures that hold actual values during execution. When `acid` evaluates an expression, the result is stored in a `Value` node containing both the data (`Store`) and a type tag. List elements share the same `Store` structure so that operations like `append` and `head` can move values between `Value` and `List` without conversion.

6.6.1 Formats

Each value also carries a format character that controls how it is printed. Formats are orthogonal to types: an integer can be displayed in hex, decimal, or even as a disassembled instruction. Here are the main format characters:

Format	Meaning
`'D'`	decimal integer
`'X'`	hexadecimal integer
`'o'`	octal integer
`'c'`	character (single byte)
`'C'`	rune (UTF-8 character)
`'b'`	byte
`'s'`	string
`'f'`	single-precision float
`'g'`	double-precision float
`'i'`	disassembled instruction
`'I'`	disassembled with file:line
`'a'`	complex/aggregate (struct)

The format characters double as format suffixes in the `acid` language: `x\D` prints `x` in decimal, `*PC\i` disassembles the instruction at PC, and so on.

6.6.2 Runtime values: `Value` and `Store`

A `Value` is a runtime value in `acid`: it contains a `Store` (the actual data), a type tag, and a scope pointer for managing local variables. `Store` is factored out as a separate struct because it is embedded by inclusion in both `Value` and `List` nodes—this is a kencc extension where an anonymous struct field makes its members directly accessible (like `v->ival` instead of `v->store.ival`).

⟨*struct* `Value` 61⟩≡ (378b)

```
struct Value
{
    Store;
    // enum<Type_kind>
```

```
    char type;
    ⟨Value other fields 62c⟩
};
```

⟨*struct* Store 62a⟩≡ (378b)
```
struct Store
{
    union {
        vlong ival;
        double fval;
        String* string;
        List* l;
        ⟨Store union other cases 62b⟩
    };

    // enum<Format_kind> 'X', 'D', ...
    char fmt;
    ⟨Store other fields 62e⟩
};
```

⟨Store *union other cases* 62b⟩≡ (62a)
```
Node* cc;
```

⟨Value *other fields* 62c⟩≡ (61) 62d▷
```
char set;
```

⟨Value *other fields* 62d⟩+≡ (61) ◁62c 134a▷
```
Lsym* scope;
```

⟨Store *other fields* 62e⟩≡ (62a)
```
Type* comt;
```

6.6.3 Compound values: `String` and `List`

`String` and `List` are the two compound value types in `acid`. Both are garbage-collected (they embed a `Gc` header). `String` holds a length-counted byte array. Like `Str` in `ratrace`, `String` uses the trick of placing the character data right after the struct in a single allocation—`strnodlen()` allocates `sizeof(String)+len+1` bytes and points `s->string` past the struct header.

⟨*struct* String 62f⟩≡ (378b)
```
struct String
{
    ⟨String first gc field 64f⟩
    char* string;
    int   len;
};
```

⟨*function* strnode 62g⟩≡ (387a)
```
String*
strnode(char *name)
{
    return strnodlen(name, strlen(name));
}
```
Uses `strnodlen()` 63a.

⟨*function* `strnodlen` 63a⟩≡ (387a)

```
String*
strnodlen(char *name, int len)
{
    String *s;

    s = gmalloc(sizeof(String)+len+1);
    s->string = (char*)s+sizeof(String);
    s->len = len;
    if(name != 0)
        memmove(s->string, name, len);
    s->string[len] = '\0';

    ⟨strnodlen() add String s to gcl 64c⟩

    return s;
}
```

Uses `gmalloc()` 151b.

`List` is a singly-linked list where each node contains a `Store` (so list elements can be integers, strings, or nested lists). You may wonder why `List` embeds a `Store` rather than a full `Value`. The reason is mostly historical: `Value` also carries scope and pop fields used by the symbol table (for shadowing in nested scopes), which are meaningless for list elements. Embedding `Store` keeps `List` nodes small and avoids dragging in scope-management state. The trade-off is that the type tag has to be duplicated in `List` (as a `type` field at the end of the struct), which is slightly redundant with the `fmt` field in `Store`. A cleaner design might split `Value` into a small core (used by both `List` and the symbol table) plus a wrapper for scope information, but the current code is small and works.

⟨*struct* `List` 63b⟩≡ (378b)

```
struct List
{
    ⟨List first gc field 64g⟩

    Store;

    List* next;
    // enum<Type_kind> ?
    char type;
};
```

⟨*function* `al` 63c⟩≡ (383c)

```
List*
al(int t)
{
    List *l;

    l = gmalloc(sizeof(List));
    memset(l, 0, sizeof(List));
    l->type = t;

    ⟨al() add List l to gcl 64d⟩

    return l;
}
```

Uses `gmalloc()` 151b.

6.6.4 Garbage collector header: `Gc`

`acid` uses a simple mark-and-sweep garbage collector for its heap-allocated objects (nodes, strings, and lists). Every collectible object starts with a `Gc` header containing a mark bit and a link pointer. All allocated objects are chained through `gcl` (the GC list). During collection, the GC first clears all marks, then scans the symbol table to mark reachable objects, and finally sweeps the list to free unmarked objects.

⟨*struct* `Gc` 64a⟩≡ (378b)
```
struct Gc
{
    char gcmark;
    Gc*  gclink;
};
```

⟨*global* `gcl` 64b⟩≡ (381a)
```
Gc* gcl;
```

⟨`strnodlen()` *add String* `s` *to* `gcl` 64c⟩≡ (63a)
```
s->gclink = gcl;
gcl = s;
```
Uses `gcl` 64b.

⟨`al()` *add List* `l` *to* `gcl` 64d⟩≡ (63c)
```
l->gclink = gcl;
gcl = l;
```
Uses `gcl` 64b.

⟨`an()` *add node* `n` *to* `gcl` 64e⟩≡ (58c)
```
n->gclink = gcl;
gcl = n;
```
Uses `gcl` 64b.

⟨`String` *first gc field* 64f⟩≡ (62f)
```
Gc;
```

⟨`List` *first gc field* 64g⟩≡ (63b)
```
Gc;
```

⟨`Node` *first gc field* 64h⟩≡ (58a)
```
Gc;
```

6.7 Symbol table

The `acid` interpreter maintains a hash table mapping names to `Lsym` ("lexical symbol") entries. This is `acid`'s own symbol table, distinct from the target program's symbol table in `libmach`. It stores everything the interpreter knows: variable values, function definitions, keyword tokens, builtin function pointers, and complex type descriptions. At startup, `varsym()`[79d] populates it with entries from the target's symbol table, so that typing `main` at the `acid` prompt gives the address of `main`.

The two symbol tables are easy to confuse. Here is the distinction:

	`libmach` symbol table	`acid` symbol table
data type	`Sym` (and `Symbol`)	`Lsym` (in `hash[]`)
source	target binary on disk	built at runtime
key	address	name (string)
contents	target's functions, globals locals	everything: targets's syms, keywords, user vars, fns, types
read by	`lookup()`[331], `getsym()`[328b]	`look()`[66d], `mkvar()`[67a]

At startup, `acid` reads the libmach symbol table (via `syminit()`[94]) and copies each entry into its own `hash[]` table (via `varsym()`), so that target program names become queryable as `acid` variables. From that point on, the user operates on the `acid`-side table.

6.7.1 Lexical symbols: `Lsym`

An `Lsym` has a name, a token kind (`lexval`—`Tid` for identifiers, keyword tokens for reserved words), and a current value (`v`).

⟨*struct* `Lsym` 65a⟩≡ (378b)
```
struct Lsym
{
    // ref_own<string>
    char* name;
    // enum<Token_kind>
    int lexval;

    // ref_own<Value>
    Value* v;

    ⟨Lsym other fields 65c⟩

    // Extra fields
    ⟨Lsym extra fields 66c⟩

};
```

⟨`Node` *other fields* 65b⟩+≡ (58a) ◁58d
```
// option<Lsym>, Some when op = ONAME
Lsym* sym;
```

⟨`Lsym` *other fields* 65c⟩≡ (65a) 65d▷
```
// option<Node>, Some when symbol is a user-defined acid function
Node* proc;
```

⟨`Lsym` *other fields* 65d⟩+≡ (65a) ◁65c 68b▷
```
Type* lt;
```

6.7.2 Hash table: `hash`

Concretely, after loading a target and defining a couple of user variables, the `hash[]` array looks like this (with `Hashsize` = 128 buckets, separate chaining via the `Lsym.hash` field):

```
hash[]                  buckets (linked via Lsym.hash)
+-------+
|   0   |---> nil
+-------+
|   1   |---> Lsym{"while", lexval=Twhile} -> nil
+-------+       (keyword, installed by kinit)
|  ...  |
+-------+
|  17   |---> Lsym{"main", lexval=Tid,     -> Lsym{"x", lexval=Tid,
+-------+           v=Value{TINT, 0x1020}}        v=Value{TINT, 42}} -> nil
|  ...  |     (target symbol from varsym)  (user var, from enter)
+-------+
```

```
|  42   |---> Lsym{"bpset", lexval=Tid,    -> nil
+-------+           proc=Node{OCALL...}, builtin=bpset_c}
|  ...  |     (function, both a user-defined proc AND a C builtin)
+-------+
|  127  |---> nil
+-------+
```

Three things are worth noting from this picture. First, the same bucket holds entries of very different kinds: keywords, target symbols, user variables, functions—they all share one flat namespace because `acid` is an untyped interpreter and `lexval` is how the lexer decides whether a name tokenises as `Tid` or as a reserved keyword. Second, a single `Lsym` can have *both* a `proc` AST and a `builtin` pointer, and at call time the interpreter prefers the user's `proc`—this is how `/lib/acid/port.acid` overrides the built-in `bpset` with its own library implementation while keeping the C version reachable. Third, because the scope of a name is always the whole table, there is no lexical scoping at the hash-table level; function locals get their separate stack mechanism described later.

The hash table uses a simple multiplicative hash (`h = h*3 + *p`) with separate chaining. `look()`[66d] searches for an existing entry; `enter()`[67b] creates a new one with a freshly allocated `Value` (defaulting to integer type with hexadecimal format). `mkvar()`[67a] combines both: look up first, create if not found.

⟨*global* `hash` 66a⟩≡ (381a)
```
// hash <string, ref_own<Lsym>, (next in bucket = Lsym.hash)
Lsym* hash[Hashsize];
```
Uses `Hashsize` 66b.

⟨*acid constants* 66b⟩+≡ (378a) ◁56c 82a▷
```
Hashsize = 128,
```

⟨`Lsym` *extra fields* 66c⟩≡ (65a)
```
// list<ref<Lsym>> (next = Sym.hash) bucket of hashtbl 'hash'
Lsym* hash;
```

6.7.3 `look()` and `mkvar()`

⟨*function* `look` 66d⟩≡ (381b)
```
Lsym*
look(char *name)
{
    Lsym *s;
    uint h;
    char *p;

    h = 0;
    for(p = name; *p; p++)
        h = h*3 + *p;
    h %= Hashsize;

    for(s = hash[h]; s; s = s->hash)
        if(strcmp(name, s->name) == 0)
            return s;
    // else
    return nil;
}
```
Uses `Hashsize` 66b.

⟨*function* `mkvar` 67a⟩≡ (381b)

```
Lsym*
mkvar(char *s)
{
    Lsym *l;

    l = look(s);
    if(l == nil)
        l = enter(s, Tid);
    return l;
}
```

Uses `Tid`, `enter()` 67b, and `look()` 66d.

6.7.4 Inserting new symbols: `enter()`

⟨*function* `enter` 67b⟩≡ (381b)

```
Lsym*
enter(char *name, int t)
{
    Lsym *s;
    uint h;
    char *p;
    ⟨enter() other locals 67c⟩

    // dupe of look()?
    h = 0;
    for(p = name; *p; p++)
        h = h*3 + *p;
    h %= Hashsize;

    s = gmalloc(sizeof(Lsym));
    memset(s, 0, sizeof(Lsym));
    s->name = strdup(name);
    s->lexval = t;

    s->hash = hash[h];
    hash[h] = s;

    ⟨enter() allocate value of symbol 67d⟩

    return s;
}
```

Uses `Hashsize` 66b and `gmalloc()` 151b.

⟨`enter()` *other locals* 67c⟩≡ (67b)

```
Value *v;
```

⟨`enter()` *allocate value of symbol* 67d⟩≡ (67b)

```
v = gmalloc(sizeof(Value));
s->v = v;

v->fmt = 'X';
v->type = TINT;
memset(v, 0, sizeof(Value));
```

Uses `TINT` 60a and `gmalloc()` 151b.

6.7.5 Builtins

Builtin functions are implemented in C and registered in the `tab`[68a] array. During initialization, `installbuiltin()`[73a] copies each entry's function pointer into the corresponding `Lsym`'s `builtin` field. When `acid` encounters a function call, it checks `s->builtin` first—if set (and no user-defined `proc` overrides it), the C function is called directly rather than interpreting an AST.

⟨*global* `tab` 68a⟩≡ (423b)
```
struct Btab
{
    char *name;
    void (*fn)(Node*, Node*);
} tab[] =
{
    ⟨tab entries 140a⟩
    0
};
```
Uses `Btab` 68a.

⟨`Lsym` *other fields* 68b⟩+≡ (65a) ◁65d 139c▷
```
void (*builtin)(Node*, Node*);
```

Chapter 7

/bin/acid

This chapter presents `acid`'s entry point, REPL, initialization, and process attachment. The overall flow is: parse command-line arguments, open the executable, load symbol tables, load the `acid` library files, then enter the interactive REPL loop.

7.1 Entry point: `main()`

⟨*global* `aout` 69a⟩≡ (381a)
```
// given program to debug (or /proc/<pid>/text when running acid <pid>)
char* aout;
```

⟨*global* `quiet` 69b⟩≡ (381a)
```
bool quiet;
```

⟨*global* `bioout` 69c⟩≡ (383c)
```
static Biobuf bioout;
```

⟨*global* `bout` 69d⟩≡ (381a)
```
// pointer to bioout
Biobuf* bout;
```

`main()`[69e] processes command-line flags, determines the executable and optional PID, then runs a long initialization sequence: format handlers, keyword table, lexer setup, builtins, symbol loading, module loading, and the user's `acidinit` function. Finally it enters the infinite REPL loop.

⟨*function* `main` 69e⟩≡ (383c)
```
void
main(int argc, char *argv[])
{
    int pid = 0;
    ⟨main()(acid) other locals 71f⟩

    argv0 = argv[0];
    aout = "8.out";
    quiet = true;
    mtype = nil;

    ARGBEGIN{
    ⟨main()(acid) command line processing 71c⟩
    default:
        usage();
    }ARGEND

    if(argc > 0) {
```

```
        ⟨main() (acid) if argc and remote adjust aout 191b⟩
        else
        ⟨main() (acid) if acid pid 88c⟩
        else {
            ⟨main() (acid) if kernel and no pid 190e⟩
            aout = argv[0];
        }
    } else
       ⟨main() (acid) if not argc and remote adjust aout 191c⟩

    ⟨main() (acid) format initializations 204a⟩
    ⟨main() (acid) initializations 72a⟩

    ⟨main() (acid) infinite loop 70c⟩
    /* not reached */
}
```

7.2 REPL

The `interactive` global controls whether `acid` is reading from a terminal (the user typing at the `acid:` prompt) or from a file (a library being loaded). When `interactive` is true, `acid` prints prompts and a newline ends a statement; when false, it processes input silently and expects explicit semicolons to end statements. This is similar to how `rc` distinguishes interactive from script mode.

⟨*global interactive (acid/globals.c)* 70a⟩≡ (381a)
```
bool interactive;
```

The `line` global tracks the current line number in the input source. It is used only for error reporting—when the lexer or parser fails, the error message includes `line` so the user can find the offending line. The lexer increments it on every newline.

⟨*global line (acid/globals.c)* 70b⟩≡ (381a)
```
int line;
```

The REPL (Read-Eval-Print Loop) is `acid`'s interactive command loop. It prints the `acid:` prompt, calls `yyparse()X` to parse and execute one statement, then loops back. An important subtlety: `yyparse()` does not just parse—it also *executes* the parsed statement immediately (via the `bigstmnt` grammar rule, which calls `execute()`[120a] inline). So parsing and execution are interleaved, not separate passes.

⟨main() (acid) infinite loop 70c⟩≡ (69e)
```
interactive = true;
line = 1;

for(;;) {

    ⟨main() (acid) in infinite loop, reinit and unwind if error 71b⟩
    stacked = 0;

    Bprint(bout, "acid: ");

    // yyparse() will internally call execute() !
    if(yyparse() != OK_1)
        die();
    restartio();

    unwind();
}
```

The `stacked` counter tracks brace nesting depth for multi-line input—when inside braces, `yylex()` prints a tab instead of the prompt.

⟨*global* `stacked` 71a⟩≡ (381a)
```
int stacked;
```

The REPL uses `setjmp/longjmp` for error recovery: if any error occurs during parsing or execution, `error()`[206b] does a `longjmp` back to here, which reinitializes the output buffer and unwinds any local variable scopes.

⟨`main()` *(acid) in infinite loop, reinit and unwind if error* 71b⟩≡ (70c)
```
if(setjmp(err)) {
    Binit(&bioout, STDOUT, OWRITE);
    unwind();
}
```

7.3 Arguments processing

7.3.1 Quiet mode: `acid -q`

⟨`main()` *(acid) command line processing* 71c⟩≡ (69e) 71g▷
```
case 'q':
    quiet = false;
    break;
```

7.3.2 Loading extra libraries: `acid -l <lib>`

⟨*global* `nlm` 71d⟩≡ (383c)
```
static int nlm;
```

⟨*global* `lm` 71e⟩≡ (383c)
```
// array<ref<string>> length = nlm
static char* lm[16];
```

⟨`main()` *(acid) other locals* 71f⟩≡ (69e) 77c▷
```
char *s;
```

⟨`main()` *(acid) command line processing* 71g⟩+≡ (69e) ◁71c 190a▷
```
case 'l':
    s = ARGF();
    if(s == nil)
        usage();
    lm[nlm++] = s;
    break;
```

7.4 Interpreter setup

Initialization is a multi-step process: set up the lexer keyword table (`kinit()`[112a]), push stdin as the input source, create the built-in variables (`loadvars()`[72b]) and register the builtin C functions (`installbuiltin()`[73a]), parse the executable and load its symbols (`attachfiles()`[73e]), load the standard `acid` library files, call the user's `acidinit` function if defined, and import the target's symbols as `acid` variables (`varsym()`[79d]).

⟨*global* `initialising` 71h⟩≡ (381a)
```
int initialising;
```

⟨main() *(acid) initializations* 72a⟩≡ (69e)

```
initialising = 1;

Binit(&bioout, STDOUT, OWRITE);
bout = &bioout;

// setup lexer
kinit();
// to read commands on stdin once all modules have been loaded
pushfile(nil);

loadvars();
installbuiltin();

⟨main() (acid) sanity check mtype 191g⟩
⟨main() (acid) attachfiles(aout, pid) 73c⟩

⟨main() (acid) load modules 77a⟩
userinit();
varsym();
⟨main() (acid) check acidmap 81b⟩

⟨main() (acid) notify setup 207f⟩

initialising = 0;
```

7.4.1 Predefined variables: `loadvars()`

`loadvars()`[72b] creates the predefined `acid` variables: `pid` (current process ID), `proc` (current process handle), `notes` (pending notes/signals), and `proclist` (list of attached processes). These are initialized to zero/empty and will be set to real values later when a process is attached or created via `new()`.

⟨*function* `loadvars` 72b⟩≡ (387a)

```
/// main -> <>
void
loadvars(void)
{
    Lsym *l;
    Value *v;

    l =  mkvar("proc");
    v = l->v;
    v->type = TINT;
    v->fmt = 'X';
    v->set = 1;
    v->ival = 0;

    l = mkvar("pid");  /* Current process */
    v = l->v;
    v->type = TINT;
    v->fmt = 'D';
    v->set = 1;
    v->ival = 0;

    mkvar("notes");   /* Pending notes */

    l = mkvar("proclist");  /* Attached processes */
    l->v->type = TLIST;
}
```

Uses TINT 60a, TLIST 60a, and mkvar() 67a.

7.4.2 Builtin functions: installbuiltin()

installbuiltin()[73a] iterates through the tab[68a] array and registers each C function as an acid builtin by setting the builtin field of the corresponding Lsym.

⟨*function* installbuiltin 73a⟩≡ (423b)
```
void
installbuiltin(void)
{
    Btab *b;
    Lsym *s;

    b = tab;
    while(b->name) {
        s = look(b->name);
        if(s == nil)
            s = enter(b->name, Tid);

        s->builtin = b->fn;
        ⟨installbuiltin() if bprint 73b⟩
        b++;
    }
}
```
Uses Tid, enter() 67b, look() 66d, and tab 68a.

⟨installbuiltin() *if* bprint 73b⟩≡ (73a)
```
if(b->fn == bprint)
    mkprint(s);
```
Uses mkprint() 409a.

7.5 Loading the executable

attachfiles()[73e] opens the executable and, if a PID was given, attaches to the running process. readtext()[74d] does the heavy lifting: it calls crackhdr()[90] to parse the executable header, loadmap()[92] to create the symbol map, and syminit()[94] to load the symbol table. It then prints the architecture identification line that you see at startup (e.g., "./buggy:386 plan 9 executable").

7.5.1 attachfiles()

⟨main()*(acid)* attachfiles(aout, pid) 73c⟩≡ (72a)
```
if (attachfiles(aout, pid) < 0)
    varreg();  /* use default register set on error */
```

⟨*global* text 73d⟩≡ (381a)
```
fdt text;
```

⟨*function* attachfiles 73e⟩≡ (383c)
```
/// main -> <>
static errorneg1
attachfiles(char *aout, int pid)
{
    interactive = false;
    ⟨attachfiles() if error 74a⟩
```

```
    if(aout) {    /* executable given */
        ⟨attachfiles() if wtflag 190b⟩
        else
            text = open(aout, OREAD);

        ⟨attachfiles() sanity check text 74b⟩
        readtext(aout);
    }
    ⟨attachfiles() if pid given 88d⟩
    return OK_0;
}
```

Uses interactive 70a, readtext() 74d, and text 73d.

⟨attachfiles() *if error* 74a⟩≡ (73e)

```
if(setjmp(err))
    return ERROR_NEG1;
```

Uses err 206a.

⟨attachfiles() *sanity check* text 74b⟩≡ (73e)

```
if(text < 0)
    error("%s: can't open %s: %r\n", argv0, aout);
```

Uses text 73d.

7.5.2 readtext()

⟨*global fhdr (acid/globals.c)* 74c⟩≡ (381a)

```
Fhdr fhdr;
```

⟨*function* readtext 74d⟩≡ (383c)

```
/// main -> attachfiles -> <>
void
readtext(char *s)
{
    ⟨readtext() locals 75b⟩

    ⟨readtext() if mtype != nil 191h⟩
    // else

    if(!crackhdr(text, &fhdr)) {
        print("can't decode file header\n");
        return;
    }

    symmap = loadmap(nil, text, &fhdr);
    ⟨readtext() sanity check symmap 75a⟩

    if(syminit(text, &fhdr) < 0) {
        print("%s: (error) syminit: %r\n", argv0);
        return;
    }
    // first output! the executable and archi
    print("%s:%s\n", s, fhdr.name);

    ⟨readtext() if mach->sbreg 76a⟩
    ⟨readtext() make objtype variable 75d⟩
    ⟨readtext() make textfile variable 75e⟩
```

```
    machbytype(fhdr.type);
    varreg();
}
```

Uses crackhdr() 90, fhdr 74c, loadmap() 92, machbytype() 91, syminit() 94, text 73d, and varreg() 76b.

The print call above produces the architecture identification line we saw in the Getting Started section—e.g., "./buggy:386 plan 9 executable". The fhdr.name field is set by crackhdr()[90] based on the magic number.

⟨readtext() *sanity check* symmap 75a⟩≡ (74d)
```
if(symmap == nil)
    print("%s: (error) loadmap: cannot make symbol map\n", argv0);
```

⟨readtext() *locals* 75b⟩≡ (74d) 75c▷
```
Dir *d;
uvlong length;
```

7.5.3 Setting objtype and textfile

After parsing the executable header, readtext()[74d] populates two predefined acid string variables: objtype (the architecture name from mach->name, e.g., "386" or "arm") and textfile (the path of the binary being debugged). These are used by the acid library files: for instance, /lib/acid/port.acid uses objtype to decide which architecture-specific library to load, and textfile is displayed in some prompts.

⟨readtext() *locals* 75c⟩+≡ (74d) ◁75b 75f▷
```
Lsym *l;
Value *v;
```

⟨readtext() *make objtype variable* 75d⟩≡ (74d)
```
l = mkvar("objtype");
v = l->v;
v->fmt = 's';
v->set = 1;
v->string = strnode(mach->name);
v->type = TSTRING;
```

Uses TSTRING 60a, mach 51a, mkvar() 67a, and strnode() 62g.

⟨readtext() *make textfile variable* 75e⟩≡ (74d)
```
l = mkvar("textfile");
v = l->v;
v->fmt = 's';
v->set = 1;
v->string = strnode(s);
v->type = TSTRING;
```

Uses TSTRING 60a, mkvar() 67a, and strnode() 62g.

7.5.4 Setting SB

Some architectures (notably ARM and MIPS in the Plan 9 toolchain) use a static base register to address global data: the register holds a fixed pointer near the data segment, and global references are encoded as offsets from this register. The linker reserves a symbol (e.g., setR12 on ARM) to record the static base value, and the debugger needs to know it to display addresses correctly. readtext()[74d] looks up the static base symbol in the target's symbol table (via lookup()[331]), records its value in mach->sb, and exposes it as the acid variable SB so the user can inspect or override it. On x86 there is no static base register, so mach->sbreg is null and this step is skipped.

⟨readtext() *locals* 75f⟩+≡ (74d) ◁75c
```
Symbol sym;
```

⟨readtext() *if* mach->sbreg 76a⟩≡ (74d)

```
if(mach->sbreg && lookup(0, mach->sbreg, &sym)) {
    mach->sb = sym.value;
    l = enter("SB", Tid);
    l->v->fmt = 'X';
    l->v->ival = mach->sb;
    l->v->type = TINT;
    l->v->set = 1;
}
```

Uses TINT 60a, Tid, enter() 67b, lookup() 331, and mach 51a.

7.5.5 Setting registers and bpinst

varreg()[76b] creates an acid variable for each machine register (e.g., R0, R1, PC, SP) using the register list from mach->reglist. Each variable's value is set to the register's offset in the /proc/<pid>/regs file, so that *R0 reads the actual register value from the live process via cormap. It also creates the bpinst variable containing the breakpoint instruction bytes, and a registers list variable for enumeration. Why use *R0 instead of just R0? Because acid treats register names as *addresses into* cormap, not as direct values. The variable R0 holds an offset (e.g., 0x10) into the /proc/<pid>/regs file. To get the actual register value, you dereference that offset through cormap using the * operator (which calls oindm()[402d], which calls get1()[351a] / get2()[350c] / geta()[349] on the map). This uniform approach lets the same * operator read register values, memory contents, and global variable values—all are just addresses into cormap from acid's point of view.

⟨*function* varreg 76b⟩≡ (387a)

```
/// main -> attachfiles -> readtext -> <>
void
varreg(void)
{
    Lsym *l;
    Value *v;
    Reglist *r;
    List **tail, *li;

    l = mkvar("registers");
    v = l->v;
    v->set = 1;
    v->type = TLIST;

    v->l = 0;
    tail = &v->l;

    for(r = mach->reglist; r->rname; r++) {
        l = mkvar(r->rname);
        v = l->v;
        v->set = 1;
        v->ival = r->roffs;
        v->fmt = r->rformat;
        v->type = TINT;

        li = al(TSTRING);
        li->string = strnode(r->rname);
        li->fmt = 's';

        *tail = li;
        tail = &li->next;
    }
```

```
    if(machdata == nil)
        return;

    l = mkvar("bpinst"); /* Breakpoint text */
    v = l->v;
    v->type = TSTRING;
    v->fmt = 's';
    v->set = 1;
    v->string = gmalloc(sizeof(String));
    v->string->len = machdata->bpsize;
    v->string->string = gmalloc(machdata->bpsize);
    memmove(v->string->string, machdata->bpinst, machdata->bpsize);
}
```

Uses TINT 60a, TLIST 60a, TSTRING 60a, al() 63c, gmalloc() 151b, mach 51a, machdata 355e, mkvar() 67a, and strnode() 62g.

7.6 Library and symbol loading

7.6.1 Loading library modules

At startup, acid loads two standard library files: /lib/acid/port.acid (architecture-independent functions like new(), bpset(), cont(), stk(), src()) and /lib/acid/<arch>.acid (architecture-specific functions). Additional libraries can be loaded with -l. Loading a module simply means parsing it with yyparse()X—since the parser executes defn statements as it goes, the function definitions become available immediately.

⟨main()*(acid) load modules* 77a⟩≡ (72a)

```
loadmodule("/lib/acid/port.acid");
loadmoduleobjtype();
⟨main()(acid) load -l modules 77d⟩
```

⟨*function* loadmoduleobjtype 77b⟩≡ (383c)

```
void
loadmoduleobjtype(void)
{
    char *buf;

    buf = smprint("/lib/acid/%s.acid", mach->name);
    loadmodule(buf);
    free(buf);
}
```

Uses loadmodule() 78a and mach 51a.

⟨main()*(acid) other locals* 77c⟩+≡ (69e) ◁71f 81a▷

```
int i;
```

⟨main()*(acid) load* -l *modules* 77d⟩≡ (77a)

```
for(i = 0; i < nlm; i++) {
    if(access(lm[i], AREAD) >= 0)
        loadmodule(lm[i]);
    else {
        s = smprint("/lib/acid/%s.acid", lm[i]);
        loadmodule(s);
        free(s);
    }
}
```

⟨*function* `loadmodule` 78a⟩≡ (383c)
```
void
loadmodule(char *s)
{
    interactive = false;
    ⟨loadmodule() unwind if error 78d⟩
    pushfile(s);
    silent = 0;
    yyparse();
    popio();
    return;
}
```
Uses `interactive` 70a, `popio()` 101e, `pushfile()` 100d, `silent` 78b, and `yyparse()`.

⟨*global* `silent` 78b⟩≡ (381a)
```
int silent;
```

⟨`error()` *(acid) if* `silent` 78c⟩≡ (206b)
```
if(silent)
    silent = 0;
```
Uses `silent` 78b.

⟨`loadmodule()` *unwind if error* 78d⟩≡ (78a)
```
if(setjmp(err)) {
    unwind();
    return;
}
```
Uses `err` 206a and `unwind()` 207a.

7.6.2 User customization: `$HOME/lib/acid` and `acidinit()`

`userinit()`[78e] loads the user's personal `acid` library from `~/lib/acid` (if it exists), then calls the `acidinit` function if one was defined in any of the loaded modules. This lets users customize their debugging environment—for instance, defining a custom `acidinit` that automatically sets breakpoints or loads type definitions.

⟨*function* `userinit` 78e⟩≡ (383c)
```
/// main -> <>
void
userinit(void)
{
    Lsym *l;
    Node *n;
    char *buf, *p;

    ⟨userinit() if user acid file 79b⟩

    interactive = false;
    ⟨userinit() unwind if error 79a⟩

    l = look("acidinit");
    if(l && l->proc) {
        n = an(ONAME, ZN, ZN);
        n->sym = l;
        n = an(OCALL, n, ZN);
        execute(n);
    }
}
```
Uses `OCALL` 59, `ONAME` 59, `ZN` 58b, `an()` 58c, `execute()` 120a, `interactive` 70a, and `look()` 66d.

⟨userinit() *unwind if error* 79a⟩≡ (78e)
```
if(setjmp(err)) {
    unwind();
    return;
}
```
Uses err 206a and unwind() 207a.

⟨userinit() *if user acid file* 79b⟩≡ (78e)
```
p = getenv("home");
if(p != nil) {
    buf = smprint("%s/lib/acid", p);
    silent = 1;
    loadmodule(buf);
    free(buf);
}
```
Uses loadmodule() 78a and silent 78b.

⟨*function* acidinit*(arm)* 79c⟩≡ (158)
```
defn acidinit()                 // Called after all the init modules are loaded
{
        bplist = {};
        bpfmt = 'X';

        srcpath = {
                "./",
                "/sys/src/libc/port/",
                "/sys/src/libc/9sys/",
                "/sys/src/libc/arm/"
        };

        srcfiles = {};                  // list of loaded files
        srctext = {};                   // the text of the files
}
```

7.6.3 Importing target symbols

varsym()[79d] is the bridge between libmach's symbol table and acid's interpreter. It iterates through every symbol in the executable (T, D, B, t, d, b, l, L types) and creates a corresponding acid variable with the symbol's address as its value. It also builds the symbols list variable, where each entry is a triple of {name, type, address}. After this, typing a function name at the acid prompt returns its address.

⟨*function* varsym 79d⟩≡ (387a)
```
/// main -> <>
void
varsym(void)
{
    int i;
    Sym *s;
    long n;
    Lsym *l;
    uvlong v;
    char buf[1024];
    List *list, **tail, *l2, *tl;

    tail = &l2;
    l2 = 0;

    symbase(&n);
```

```
    for(i = 0; i < n; i++) {
        s = getsym(i);
        switch(s->type) {
        case 'T':
        case 'L':
        case 'D':
        case 'B':
        case 'b':
        case 'd':
        case 'l':
        case 't':
            if(s->name[0] == '.')
                continue;

            v = s->value;
            tl = al(TLIST);
            *tail = tl;
            tail = &tl->next;

            l = unique(buf, s);

            l->v->set = 1;
            l->v->type = TINT;
            l->v->ival = v;

            if(l->v->comt == 0)
                l->v->fmt = 'X';

            /* Enter as list of { name, type, value } */
            list = al(TSTRING);
            tl->l = list;
            list->string = strnode(buf);
            list->fmt = 's';
            list->next = al(TINT);
            list = list->next;
            list->fmt = 'c';
            list->ival = s->type;
            list->next = al(TINT);
            list = list->next;
            list->fmt = 'X';
            list->ival = v;

        }
    }
    l = mkvar("symbols");
    l->v->set = 1;
    l->v->type = TLIST;
    l->v->l = l2;
    if(l2 == nil)
        print("no symbol information\n");
}
```

Uses TINT 60a, TLIST 60a, TSTRING 60a, al() 63c, getsym() 328b, mkvar() 67a, strnode() 62g, symbase() 96a, and unique() 384b.

7.6.4 The acidmap hook

acidmap is an optional callback hook the user (or a loaded library) may define. After acid finishes loading

symbols and modules, it looks up an `acid` function called `acidmap` and, if found, calls it with no arguments. The convention is that `acidmap` sets up complex type declarations for the target program (e.g., declaring `Node`, `Lsym`, and so on as `aggr` types) so that the user can immediately type `*addr.field` and get sensible output. Without `acidmap`, the user would have to load these declarations manually each session. The pattern below—building an `ONAME` node, wrapping it in an `OCALL` node, then calling `execute()`[120a]—is how C code inside `acid` invokes a user-defined `acid` function. We will see the same pattern again in `dostop()`[88a] and `userinit()`[78e].

⟨`main()` *(acid) other locals* 81a⟩+≡ (69e) ◁77c
```
Lsym *l;
Node *n;
```

⟨`main()` *(acid) check acidmap* 81b⟩≡ (72a)
```
l = look("acidmap");
if(l && l->proc) {
    n = an(ONAME, ZN, ZN);
    n->sym = l;
    n = an(OCALL, n, ZN);
    execute(n);
}
```

7.7 Starting a new process

When the user types `new()` at the `acid` prompt, the `acid` library function `new()`[161a] (defined in `/lib/acid/port.acid`) calls the builtin `newproc()`[81c], which forks a child process to run the target program. For reference, here is what `new()` looks like in the library:

```
defn new() {
    bplist = {};                // reset breakpoint list
    newproc(progargs);          // fork+exec the target
    bpset(follow(main)[0]);     // bp at main (handles delay slots)
    cont();                     // run until that breakpoint
    bpdel(*PC);                 // remove the bp
}
```

So `new()` is itself a small composition of more primitive operations: `newproc` (a C builtin) plus `bpset()`[165a] / `cont()`[166] / `bpdel()`[165b] (other library functions). The user can read this definition with `whatis new` at the prompt and override it freely. This is the essence of a programmable debugger: the "run a program" command is not built in, it is written in `acid` itself.

7.7.1 Argument parsing: `newproc()`

The `newproc()`[81c] builtin takes an optional string of arguments, splits it on whitespace (like a simple shell), prepends the executable name (`aout`), and calls `nproc()`[82b] to do the actual fork/exec.

⟨*function* `newproc` 81c⟩≡ (423b)
```
/// main -> yyparse -> new -> <> -> nproc -> sproc
void
newproc(Node *r, Node *args)
{
    int i;
    Node res;
    char *p, *e;
    char *argv[Maxarg], buf[Strsize];

    i = 1;
```

```
    argv[0] = aout;

    if(args) {
        expr(args, &res);
        if(res.type != TSTRING)
            error("newproc(): arg not string");
        if(res.string->len >= sizeof(buf))
            error("newproc(): too many arguments");
        memmove(buf, res.string->string, res.string->len);
        buf[res.string->len] = '\0';
        p = buf;
        e = buf+res.string->len;
        for(;;) {
            while(p < e && (*p == '\t' || *p == ' '))
                *p++ = '\0';
            if(p >= e)
                break;
            argv[i++] = p;
            if(i >= Maxarg)
                error("newproc: too many arguments");
            while(p < e && *p != '\t' && *p != ' ')
                p++;
        }
    }
    argv[i] = nil;
    r->op = OCONST;
    r->type = TINT;
    r->fmt = 'D';
    r->ival = nproc(argv);
}
```

⟨*acid constants* 82a⟩+≡ (378a) ◁66b 83a▷

```
Maxarg  = 512,
```

7.7.2 Fork/hang/exec: `nproc()`

`nproc()`[82b] implements the fork/hang/exec pattern, the same one used by `ratrace`. The child writes `hang` to its own `ctl`, closes all file descriptors, reopens the console for stdin/stdout/stderr, then calls `exec()`. The `hang` ensures the child stops after `exec`, giving the parent control. The parent calls `install()`[83b] to register the child's PID in the process table, then `msg(pid, "waitstop")` to block until the child is actually stopped. It then calls `sproc()`[85b] to set up the memory map and `dostop()`[88a] to invoke the `stopped()` callback if one is defined.

⟨*function* `nproc` 82b⟩≡ (392a)

```
/// main -> yyparse -> new -> newproc -> <> -> sproc
int
nproc(char **argv)
{
    char buf[128];
    int pid, i;
    fdt fd;

    pid = fork();
    switch(pid) {
    case -1:
        error("new: fork %r");
    case 0:
        // child
        rfork(RFNAMEG|RFNOTEG);
```

```
        snprint(buf, sizeof(buf), "/proc/%d/ctl", getpid());
        fd = open(buf, ORDWR);
        if(fd < 0)
            fatal("new: open %s: %r", buf);

        write(fd, "hang", 4);
        close(fd);

        close(STDIN);
        close(STDOUT);
        close(STDERR);
        for(i = 3; i < NFD; i++)
            close(i);

        open("/dev/cons", OREAD);
        open("/dev/cons", OWRITE);
        open("/dev/cons", OWRITE);
        // Exec!
        exec(argv[0], argv);
        // should never reach
        fatal("new: exec %s: %r");

    //parent
    default:
        install(pid);
        msg(pid, "waitstop");

        notes(pid);
        sproc(pid);
        dostop(pid);

        break;
    }

    return pid;
}
```

Uses NFD 83a, dostop() 88a, install() 83b, msg() 56d, and sproc() 85b.

⟨*acid constants* 83a⟩+≡ (378a) ◁82a 103b▷

```
NFD  = 100,
```

7.7.3 Registering the process: install()

install()[83b] registers a new process: it finds a free slot in ptab[56b], opens /proc/<pid>/ctl, and adds the PID to the proclist variable (so the acid user can see which processes are being debugged). deinstall()[84] does the reverse when a process exits.

⟨*function* install 83b⟩≡ (392a)

```
static void
install(int pid)
{
    Lsym *s;
    List *l;
    char buf[128];
    int i, fd, new, p;

    new = -1;
    for(i = 0; i < Maxproc; i++) {
```

```
            p = ptab[i].pid;
            if(p == pid)
                return;
            if(p == 0 && new == -1)
                new = i;
        }
        if(new == -1)
            error("no free process slots");

        snprint(buf, sizeof(buf), "/proc/%d/ctl", pid);
        fd = open(buf, OWRITE);
        if(fd < 0)
            error("pid=%d: open ctl: %r", pid);

        ptab[new].pid = pid;
        ptab[new].ctl = fd;

        s = look("proclist");
        l = al(TINT);
        l->fmt = 'D';
        l->ival = pid;
        l->next = s->v->l;
        s->v->l = l;
        s->v->set = 1;
    }
```

Uses Maxproc 56c, TINT 60a, al() 63c, look() 66d, and ptab 56b.

⟨*function* deinstall 84⟩≡ (392a)

```
    void
    deinstall(int pid)
    {
        int i;
        Lsym *s;
        List *f, **d;

        for(i = 0; i < Maxproc; i++) {
            if(ptab[i].pid == pid) {
                close(ptab[i].ctl);
                ptab[i].pid = 0;
                s = look("proclist");
                d = &s->v->l;
                for(f = *d; f; f = f->next) {
                    if(f->ival == pid) {
                        *d = f->next;
                        break;
                    }
                }
                s = look("pid");
                if(s->v->ival == pid)
                    s->v->ival = 0;
                return;
            }
        }
    }
```

Uses Maxproc 56c, look() 66d, and ptab 56b.

7.7.4 Reading pending notes: `notes()`

`notes()`[85a] reads pending notes (signals) from `/proc/<pid>/note` and stores them as a list of strings in the `acid` variable `notes`. This lets the user see why the process stopped—e.g., "sys: trap: divide error" or "sys: trap: fault read".

⟨*function notes (acid/proc.c)* 85a⟩≡ (392a)

```
void
notes(int pid)
{
    Lsym *s;
    Value *v;
    int i, fd;
    char buf[128];
    List *l, **tail;

    s = look("notes");
    if(s == nil)
        return;

    // else
    v = s->v;

    snprint(buf, sizeof(buf), "/proc/%d/note", pid);
    fd = open(buf, OREAD);
    if(fd < 0)
        error("pid=%d: open note: %r", pid);

    v->set = 1;
    v->type = TLIST;
    v->l = 0;
    tail = &v->l;
    for(;;) {
        i = read(fd, buf, sizeof(buf));
        if(i <= 0)
            break;
        buf[i] = '\0';
        l = al(TSTRING);
        l->string = strnode(buf);
        l->fmt = 's';
        *tail = l;
        tail = &l->next;
    }
    close(fd);
}
```

Uses `TLIST` 60a, `TSTRING` 60a, `al()` 63c, `look()` 66d, and `strnode()` 62g.

7.7.5 Setting up process memory: `sproc()`

`sproc()`[85b] sets up the live process for debugging: it opens `/proc/<pid>/mem` for reading and writing the target's memory, verifies that the executable matches the running process (via `checkqid()`[87a]), creates `cormap` with `attachproc()`[96b], and renames the map segments to `*text` and `*data` (the `*` prefix distinguishes live memory from the file-based segments in `symmap`).

⟨*function* `sproc` 85b⟩≡ (392a)

```
/// (main -> yyparse -> execute -> new -> newproc -> nproc) | main -> <>
void
sproc(int pid)
{
```

```
    char buf[64];
    fdt fcor;
    Lsym *s;
    int i;

    ⟨sproc() sanity check symmap 86a⟩

    snprint(buf, sizeof(buf), "/proc/%d/mem", pid);
    fcor = open(buf, ORDWR);
    ⟨sproc() sanity check fcor 86b⟩

    checkqid(symmap->seg[0].fd, pid);

    s = look("pid");
    s->v->ival = pid;

    nocore();

    cormap = attachproc(pid, kernel, fcor, &fhdr);
    ⟨sproc() sanity check cormap 86c⟩

    i = findseg(cormap, "text");
    if (i > 0)
        cormap->seg[i].name = "*text";

    i = findseg(cormap, "data");
    if (i > 0)
        cormap->seg[i].name = "*data";

    install(pid);
}
```

Uses attachproc() 96b, checkqid() 87a, fhdr 74c, findseg() 98, install() 83b, kernel 190c, look() 66d, and nocore() 87b.

⟨sproc() *sanity check* symmap 86a⟩≡ (85b)

```
if(symmap == nil)
    error("no map");
```

⟨sproc() *sanity check* fcor 86b⟩≡ (85b)

```
if(fcor < 0)
    error("setproc: open %s: %r", buf);
```

⟨sproc() *sanity check* cormap 86c⟩≡ (85b)

```
if (cormap == nil)
    error("setproc: can't make coremap: %r");
```

7.7.6 QID sanity check: checkqid()

checkqid()[87a] is a defensive sanity check: it compares the QID (a unique file identifier in Plan 9, similar to an inode number) of the executable file opened by acid with the QID of /proc/<pid>/text. If they differ, acid prints a warning ("image does not match text for pid ...")—it does not refuse to proceed. The check is only meaningful when the user explicitly passes *both* a PID and a separate executable path—e.g., acid 1234 /tmp/myprog. With just acid 1234, aout is set to /proc/1234/text (i.e., the file acid opens is the same one checkqid() then opens), so the QIDs always match by construction. The two-argument form acid <pid> <file> is a historical convention inherited from earlier UNIX debuggers like db. Why it survives in acid and

what its intended use cases are, I do not know—the source code does not say.

⟨*function* checkqid 87a⟩≡ (383c)

```
void
checkqid(int f1, int pid)
{
    fdt fd;
    Dir *d1, *d2;
    char buf[128];

    ⟨checkqid() return if [[kernel 190f⟩

    d1 = dirfstat(f1);
    if(d1 == nil){
        print("checkqid: (qid not checked) dirfstat: %r\n");
        return;
    }

    snprint(buf, sizeof(buf), "/proc/%d/text", pid);
    fd = open(buf, OREAD);
    if(fd < 0 || (d2 = dirfstat(fd)) == nil){
        print("checkqid: (qid not checked) dirstat %s: %r\n", buf);
        free(d1);
        if(fd >= 0)
            close(fd);
        return;
    }

    close(fd);

    if(d1->qid.path != d2->qid.path || d1->qid.vers != d2->qid.vers || d1->qid.type != d2->qid.type){
        print("path %llux %llux vers %lud %lud type %d %d\n",
            d1->qid.path, d2->qid.path, d1->qid.vers, d2->qid.vers, d1->qid.type, d2->qid.type);
        print("warning: image does not match text for pid %d\n", pid);
    }
    free(d1);
    free(d2);
}
```

7.7.7 Releasing the previous map: `nocore()`

⟨*function* nocore 87b⟩≡ (392a)

```
/// sproc -> <>
void
nocore(void)
{
    int i;

    if(cormap == nil)
        return;

    for (i = 0; i < cormap->nsegs; i++)
        if (cormap->seg[i].inuse && cormap->seg[i].fd >= 0)
            close(cormap->seg[i].fd);
    free(cormap);
    cormap = nil;
}
```

7.7.8 `dostop()` and `stopped` hook

`dostop()`[88a] looks for a user-defined `acid` function called `stopped` and, if found, calls it with the PID as argument. This is a callback hook: the user can define `stopped(pid)` in their `acid` library to automatically perform actions whenever a process stops—for example, printing registers or the current source line.

⟨*function* `dostop` 88a⟩≡ (392a)
```
/// main -> yyparse -> new -> newproc -> nproc -> sproc; <>
void
dostop(int pid)
{
    Lsym *s;
    Node *np, *p;

    s = look("stopped");
    if(s && s->proc) {
        np = an(ONAME, ZN, ZN);
        np->sym = s;
        np->fmt = 'D';
        np->type = TINT;
        p = con(pid);
        p->fmt = 'D';
        np = an(OCALL, np, p);
        execute(np);
    }
}
```
Uses `OCALL` 59, `ONAME` 59, `TINT` 60a, `ZN` 58b, `an()` 58c, `con()` 382a, `execute()` 120a, and `look()` 66d.

7.8 Attaching to a running process

When invoked as `acid <pid>`, `acid` attaches to an already-running (or `Broken`) process instead of starting a new one. It reads the executable path from `/proc/<pid>/text` and calls `sproc()`[85b] to set up `cormap`.

⟨*global* `prog` 88b⟩≡ (383c)
```
static char prog[128];
```

⟨`main()` *(acid) if* `acid pid` 88c⟩≡ (69e)
```
if(isnumeric(argv[0])) {
    pid = strtol(argv[0], 0, 0);
    snprint(prog, sizeof(prog), "/proc/%d/text", pid);
    aout = prog;

    if(argc > 1)
        aout = argv[1];
    else
      ⟨main() (acid) when acid pid if kernel 190g⟩
}
```

⟨`attachfiles()` *if* `pid` *given* 88d⟩≡ (73e)
```
if(pid)     /* pid given */
    sproc(pid);
```
Uses `sproc()` 85b.

Why call `sproc()` directly here, instead of going through `newproc()`[81c] / `nproc()`[82b] like we saw earlier for `new()`? Because the `acid pid` case attaches to a process that *already exists*—there is no need to fork or exec. `newproc()` / `nproc()` handle the fork/hang/exec dance required when `acid` is starting a fresh program; here, the target is already running (or `Broken`), so we just need to set up the `cormap` for memory access and register the PID in `ptab`. `sproc()` does exactly that.

Chapter 8

libmach/

`libmach` is the shared library that all Plan 9 binary tools use to read executables and object files. It does the reverse of the linker: where the linker writes magic numbers, symbol tables, and section headers, `libmach` reads them back. It is used by `acid`, `db`, the profilers, and the emulators.

8.1 Parsing executables

8.1.1 Header parsing: `Fhdr` and `crackhdr()`

`crackhdr()`[90] is the entry point for parsing an executable. It reads the first bytes of the file, extracts the magic number, and searches the `exectab`[89b] table for a matching entry. Each entry in `exectab` specifies the header size, a byte-swapping function (for cross-endian reading), and a header-specific parser. When a match is found, the global `mach` is set to the corresponding architecture, and the `Fhdr` structure is filled with the segment addresses, sizes, and symbol table offsets.

⟨*struct* `Exectable` 89a⟩≡ (321e)

```
/*
 *      definition of per-executable file type structures
 */
typedef struct Exectable{
    long        magic;                  /* big-endian magic number of file */
    char        *name;                  /* executable identifier */
    char        *dlmname;               /* dynamically loadable module identifier */
    uchar       type;                   /* Internal code */
    uchar       _magic;                 /* _MAGIC() magic */
    Mach        *mach;                  /* Per-machine data */
    long        hsize;                  /* header size */
    ulong       (*swal)(ulong);         /* beswal or leswal */
    int (*hparse)(int, Fhdr*, ExecHdr*);
} ExecTable;
```

Uses `Exectable` 89a.

⟨*global* `exectab` 89b⟩≡ (321e)

```
ExecTable exectab[] =
{
    { I_MAGIC,                  /* I386 8.out & boot image */
        "386 plan 9 executable",
        "386 plan 9 dlm",
        FI386,
        1,
        &mi386,
        sizeof(Exec),
        beswal,
```

```
        common },
    { ELF_MAG,                  /* any ELF */
        "elf executable",
        nil,
        FNONE,
        0,
        &mi386,
        sizeof(Ehdr),
        nil,
        elfdotout },
    { E_MAGIC,                  /* Arm 5.out and boot image */
        "arm plan 9 executable",
        "arm plan 9 dlm",
        FARM,
        1,
        &marm,
        sizeof(Exec),
        beswal,
        common },
    { V_MAGIC,                 /* Mips v.out */
        "mips plan 9 executable BE",
        "mips plan 9 dlm BE",
        FMIPS,
        1,
        &mmips,
        sizeof(Exec),
        beswal,
        adotout },

    { 0 },
};
```

Uses `adotout()` 317a, `beswal()` 315b, `common()` 317c, `elfdotout()` 320a, `marm` 308g, `mi386`, and `mmips`.

We can now look at `crackhdr()` itself, which we have referenced several times. It reads the first bytes of an executable file, extracts the magic number, and searches the `exectab` table for a matching entry. When a match is found, it sets the global `mach` to the corresponding architecture's `Mach` structure and fills the `Fhdr` with the parsed header fields (text/data addresses and sizes, symbol table offset, line number table offset).

⟨*function* `crackhdr` 90⟩≡ (321e)

```
int
crackhdr(fdt fd, Fhdr *fp)
{
    ExecTable *mp;
    ExecHdr d;
    int nb, ret;
    ulong magic;

    fp->type = FNONE;
    nb = read(fd, (char *)&d.e, sizeof(d.e));
    if (nb <= 0)
        return 0;

    ret = 0;
    magic = beswal(d.e.magic);  /* big-endian */
    for (mp = exectab; mp->magic; mp++) {
        if (nb < mp->hsize)
            continue;

        /*
         * The magic number has morphed into something
```

```
         * with fields (the straw was DYN_MAGIC) so now
         * a flag is needed in Fhdr to distinguish _MAGIC()
         * magic numbers from foreign magic numbers.
         *
         * This code is creaking a bit and if it has to
         * be modified/extended much more it's probably
         * time to step back and redo it all.
         */
        if(mp->_magic){
            if(mp->magic != (magic & ~DYN_MAGIC))
                continue;

//              if(mp->magic == V_MAGIC)
//                  mp = couldbe4k(mp);

            if ((magic & DYN_MAGIC) && mp->dlmname != nil)
                fp->name = mp->dlmname;
            else
                fp->name = mp->name;
        }
        else{
            if(mp->magic != magic)
                continue;
            fp->name = mp->name;
        }
        fp->type = mp->type;
        fp->hdrsz = mp->hsize;  /* will be zero on bootables */
        fp->_magic = mp->_magic;
        fp->magic = magic;

        mach = mp->mach;
        if(mp->swal != nil)
            hswal(&d, sizeof(d.e)/sizeof(ulong), mp->swal);
        ret = mp->hparse(fd, fp, &d);
        seek(fd, mp->hsize, 0);  /* seek to end of header */
        break;
    }
    if(mp->magic == 0)
        werrstr("unknown header type");
    return ret;
}
```

Uses `beswal()` 315b, `exectab` 89b, `hswal()` 316e, and `mach` 51a.

8.1.2 Selecting the architecture: `machbytype()`

`machbytype()`[91] selects the architecture-specific disassembler and debugging support (`machdata`) based on the executable type determined by `crackhdr()`[90]. It is called by `acid`'s `readtext()`[74d] after `crackhdr()` returns. You may wonder why `crackhdr()` sets the global `mach` but not the global `machdata`: they are filled in by two separate tables. `mach` comes from `exectab` (mapped from magic numbers in `libmach`'s `executable.c`), while `machdata` comes from the `machines` table in `setmach.c`. The split exists because `Mach` holds machine description (register names, page size, endianness) needed early during header parsing, while `Machdata` holds the higher-level operations (disassembler, breakpoint instruction, follow-set) that depend on `Mach` being already set up. Splitting the two avoids a circular initialization between them.

⟨*function* `machbytype` 91⟩≡ (371a)

```
/*
 * select a machine by executable file type
 */
```

```
void
machbytype(int type)
{
    Machtab *mp;

    for (mp = machines; mp->name; mp++){
        if (mp->type == type || mp->boottype == type) {
            asstype = mp->asstype;

            machdata = mp->machdata;

            break;
        }
    }
}
```

Uses `asstype` 355d, `machdata` 355e, and `machines` 370b.

8.2 Mapping the executable file: `loadmap()`

`loadmap()`[92] creates a `Map` for an executable file (not a live process). It sets up two segments—text and data—mapping address ranges to file offsets so that reading an address through the map reads the corresponding bytes from the executable file.

⟨*function* `loadmap` 92⟩≡ (324b)

```
Map*
loadmap(Map *map, int fd, Fhdr *fp)
{
    map = newmap(map, 2);
    if (map == nil)
        return nil;

    map->seg[0].b = fp->txtaddr;
    map->seg[0].e = fp->txtaddr+fp->txtsz;
    map->seg[0].f = fp->txtoff;
    map->seg[0].fd = fd;
    map->seg[0].inuse = 1;
    map->seg[0].name = "text";

    map->seg[1].b = fp->dataddr;
    map->seg[1].e = fp->dataddr+fp->datsz;
    map->seg[1].f = fp->datoff;
    map->seg[1].fd = fd;
    map->seg[1].inuse = 1;
    map->seg[1].name = "data";

    return map;
}
```

Uses `newmap()` 323a.

Why do we need `loadmap()` when `attachproc()`[96b] already maps the text and data segments? The two functions serve different purposes:

- `loadmap()` builds `symmap`, which maps text and data *from the executable file on disk*. Its segments point to the `a.out` file via offsets. This is what `acid` uses to read the symbol table, line number table, and SP-PC table—all of which live in the file but are *not* loaded into the running process's memory.

- `attachproc()` builds `cormap`, which maps text and data *from the running process's memory* via `/proc/<pid>/m` This is what `acid` uses to read live register values, current stack contents, and the actual current state of variables.

Both maps cover the same address ranges, but they back them with different files. `acid` always has both maps available so it can read symbols from one and live state from the other.

8.3 Loading the symbol table

The symbol table is stored in the executable as a sequence of records, each containing a value (address or offset), a type character, and a name. `syminit()`[94] reads the entire table into memory and classifies symbols by type: text (`T`/`t`), data (`D`/`d`/`B`/`b`), autos/params (`a`/`p`/`m`), and file history (`z`). It also loads the SP-PC and line number-PC tables if present.

⟨*global nsym (libmach/sym.c)* 93a⟩≡ (347)
```
static long nsym;   /* number of symbols */
```

⟨*global* `symbols` 93b⟩≡ (347)
```
static Sym *symbols;  /* symbol table */
```

8.3.1 Reading symbol records: `syminit()`

`syminit()`[94] is a single long function but its structure is straightforward:

1. Reset any previously loaded symbol table state (`cleansyms()`[327]).
2. Allocate the `symbols` array, sized conservatively based on the symbol table size (assuming a minimum record size of 7 bytes per entry).
3. Open the executable as a `Biobuf` and seek to the symbol table offset given by `fp->symoff`.
4. Loop reading one symbol record at a time: read the value (4 or 8 bytes depending on architecture), the type byte, and the name (via `decodename()`[326m], which handles both old fixed-length and new variable-length name encodings).
5. Classify each symbol by type and update counters (`ntxt` for text symbols, `nglob` for globals, `nauto` for locals/parameters, `nhist` / `nfiles` for file history entries).
6. Optionally read the SP-PC and line number-PC tables into separate buffers (`spoff` and `pcline`).

For example, given the `hello.s` symbols we showed earlier, the resulting `symbols` array would look like:

```
symbols[0] = { value=0x1020, type='T', name="main" }
symbols[1] = { value=0x0008, type='m', name=".frame" }
symbols[2] = { value=0x0000, type='p', name="x" }
symbols[3] = { value=0x0001, type='z', name="hello.s" }
```

with `ntxt=1`, `nauto=2` (the `m` and `p` entries), `nhist=1`, `nfiles=1`. Note that `syminit()` only loads the symbols into a flat array; it does not yet group locals under their containing function or sort them by address. That work is done lazily by `buildtbls()`[329] the first time the user looks up a symbol (see Section **??** below... if it exists).

⟨*function* `syminit` 94⟩≡ (347)

```
/*
 * initialize the symbol tables
 */
int
syminit(int fd, Fhdr *fp)
{
    Sym *p;
    long i, l, size;
    vlong vl;
    Biobuf b;
    int svalsz;

    if(fp->symsz == 0)
        return 0;
    if(fp->type == FNONE)
        return 0;

    cleansyms();
    textseg(fp->txtaddr, fp);
        /* minimum symbol record size = 4+1+2 bytes */
    symbols = malloc((fp->symsz/(4+1+2)+1)*sizeof(Sym));
    if(symbols == 0) {
        werrstr("can't malloc %ld bytes", fp->symsz);
        return -1;
    }
    Binit(&b, fd, OREAD);
    Bseek(&b, fp->symoff, 0);
    nsym = 0;
    size = 0;
    if((fp->_magic && (fp->magic & HDR_MAGIC)) || mach->szaddr == 8)
        svalsz = 8;
    else
        svalsz = 4;
    for(p = symbols; size < fp->symsz; p++, nsym++) {
        if(svalsz == 8){
            if(Bread(&b, &vl, 8) != 8)
                return symerrmsg(8, "symbol");
            p->value = beswav(vl);
        }
        else{
            if(Bread(&b, &l, 4) != 4)
                return symerrmsg(4, "symbol");
            p->value = (u32int)beswal(l);
        }
        if(Bread(&b, &p->type, sizeof(p->type)) != sizeof(p->type))
            return symerrmsg(sizeof(p->value), "symbol");

        i = decodename(&b, p);
        if(i < 0)
            return -1;
        size += i+svalsz+sizeof(p->type);

        /* count global & auto vars, text symbols, and file names */
        switch (p->type) {
        case 'l':
```

```
    case 'L':
    case 't':
    case 'T':
        ntxt++;
        break;
    case 'd':
    case 'D':
    case 'b':
    case 'B':
        nglob++;
        break;
    case 'f':
        if(strcmp(p->name, ".frame") == 0) {
            p->type = 'm';
            nauto++;
        }
        else if(p->value > fmax)
            fmax = p->value; /* highest path index */
        break;
    case 'a':
    case 'p':
    case 'm':
        nauto++;
        break;
    case 'z':
        if(p->value == 1) {  /* one extra per file */
            nhist++;
            nfiles++;
        }
        nhist++;
        break;
    default:
        break;
    }
}
if (debug)
    print("NG: %ld NT: %d NF: %d\n", nglob, ntxt, fmax);
if (fp->sppcsz) {   /* pc-sp offset table */
    spoff = (uchar *)malloc(fp->sppcsz);
    if(spoff == 0) {
        werrstr("can't malloc %ld bytes", fp->sppcsz);
        return -1;
    }
    Bseek(&b, fp->sppcoff, 0);
    if(Bread(&b, spoff, fp->sppcsz) != fp->sppcsz){
        spoff = 0;
        return symerrmsg(fp->sppcsz, "sp-pc");
    }
    spoffend = spoff+fp->sppcsz;
}
if (fp->lnpcsz) {   /* pc-line number table */
    pcline = (uchar *)malloc(fp->lnpcsz);
    if(pcline == 0) {
        werrstr("can't malloc %ld bytes", fp->lnpcsz);
        return -1;
    }
    Bseek(&b, fp->lnpcoff, 0);
    if(Bread(&b, pcline, fp->lnpcsz) != fp->lnpcsz){
        pcline = 0;
        return symerrmsg(fp->lnpcsz, "pc-line");
```

```
        }
        pclineend = pcline+fp->lnpcsz;
    }
    return nsym;
}
```

Uses beswal() 315b, beswav() 315c, cleansyms() 327, debug-98 325f, decodename() 326m, fmax-101 325i, mach 51a, nauto-106 325n, nfiles-107 326a, nglob-108 326b, nhist-109 326c, nsym-110 93a, ntxt-111 326d, pcline-112 326e, pclineend-113 326f, spoff-114 326g, spoffend-115 326h, symbols-116 93b, symerrmsg() 326l, and textseg() 328a.

8.3.2 Accessing the raw table: symbase()

⟨*function* symbase 96a⟩≡ (347)

```
/*
 * symbase: return base and size of raw symbol table
 *  (special hack for high access rate operations)
 */
Sym *
symbase(long *n)
{
    *n = nsym;
    return symbols;
}
```

Uses nsym-110 93a and symbols-116 93b.

8.4 Mapping process memory: attachproc()

attachproc()[96b] creates a Map for a live process by opening /proc/<pid>/regs (and fpregs for floating-point registers) and combining them with the /proc/<pid>/mem file descriptor. The resulting map has four segments: registers, FP registers, text, and data. The text and data segments are mapped through mem, so reading an address in the map reads the corresponding virtual address in the live process.

⟨*function* attachproc 96b⟩≡ (324b)

```
Map*
attachproc(int pid, int kflag, int corefd, Fhdr *fp)
{
    char buf[64], *regs;
    int fd;
    Map *map;
    uvlong n;

    map = newmap(0, 4);
    if (!map)
        return 0;
    if(kflag)
        regs = "kregs";
    else
        regs = "regs";
    if (mach->regsize) {
        sprint(buf, "/proc/%d/%s", pid, regs);
        fd = open(buf, ORDWR);
        if(fd < 0)
            fd = open(buf, OREAD);
        if(fd < 0) {
            free(map);
            return 0;
        }
```

```
        setmap(map, fd, 0, mach->regsize, 0, "regs");
    }
    if (mach->fpregsize) {
        sprint(buf, "/proc/%d/fpregs", pid);
        fd = open(buf, ORDWR);
        if(fd < 0)
            fd = open(buf, OREAD);
        if(fd < 0) {
            close(map->seg[0].fd);
            free(map);
            return 0;
        }
        setmap(map, fd, mach->regsize, mach->regsize+mach->fpregsize, 0, "fpregs");
    }
    setmap(map, corefd, fp->txtaddr, fp->txtaddr+fp->txtsz, fp->txtaddr, "text");
    if(kflag || fp->dataddr >= mach->utop) {
        setmap(map, corefd, fp->dataddr, ~0, fp->dataddr, "data");
        return map;
    }
    n = stacktop(pid);
    if (n == 0) {
        setmap(map, corefd, fp->dataddr, mach->utop, fp->dataddr, "data");
        return map;
    }
    setmap(map, corefd, fp->dataddr, n, fp->dataddr, "data");
    return map;
}
```

Uses `mach` 51a, `newmap()` 323a, and `setmap()` 323b.

After `attachproc()` returns, the `Map` has up to four segments wired up like this:

```
Map* (cormap)
+--------------------------------+
| nsegs = 4                      |
+--------------------------------+
| seg[0] "regs"                  |
|   fd  -> /proc/<pid>/regs      |
|   b=0, e=mach->regsize         |
+--------------------------------+
| seg[1] "fpregs"                |
|   fd  -> /proc/<pid>/fpregs    |
|   b=regsize, e=regsize+fpregsize|
+--------------------------------+
| seg[2] "text"                  |
|   fd  -> /proc/<pid>/mem       |
|   b=fp->txtaddr,               |
|   e=fp->txtaddr+fp->txtsz      |
+--------------------------------+
| seg[3] "data"                  |
|   fd  -> /proc/<pid>/mem       |
|   b=fp->dataddr,               |
|   e=stacktop(pid)              |
+--------------------------------+
```

Note that text and data both point to the same `/proc/<pid>/mem` fd, but with different address ranges; the `Map` abstraction demultiplexes them by checking which segment contains the address being read.

⟨*function* `findseg` 98⟩≡ (324b)

```
int
findseg(Map *map, char *name)
{
    int i;

    if (!map)
        return -1;
    for (i = 0; i < map->nsegs; i++)
        if (map->seg[i].inuse && !strcmp(map->seg[i].name, name))
            return i;
    return -1;
}
```

Chapter 9

Parsing `acid` code

This chapter presents the lexer and parser for the `acid` language. The lexer and parser follow the same patterns seen in the COMPILER book [Pad16a], SHELL book [Pad18], and ASSEMBLER book [Pad15a]: a hand-written lexer feeding a yacc-generated parser. I will not explain them in as much detail, since the focus of this book is on debugging, not parsing.

9.1 Files management

Like the C preprocessor's `#include`, `acid` can read input from multiple files (the standard library, user libraries, and stdin). The `IOstack` structure maintains a stack of open input sources. When `acid` encounters an `include()` call or loads a module, it pushes a new `IOstack` entry; when that file is exhausted, it pops back to the previous source.

9.1.1 Input source stack: `IOstack`

The stack is a plain singly-linked list rooted at the global `lexio`, with each frame remembering the line number at which it was suspended so that an error inside a nested include can still report the right location in the outer file. After `acid`'s startup has loaded the port library, the architecture file, and the user's `/lib/acid`, and the user is typing at the REPL prompt, `lexio` looks like:

```
lexio                                                  (top of stack)
 |
 v
+----------------------------+
| IOstack                    |
|   name = "<stdin>"         |     <-- current input
|   fin  = Biobuf(/fd/0)     |         (yylex reads here)
|   line = 5                 |
|   prev -+                  |
+---------|------------------+
          v
+----------------------------+
| IOstack                    |
|   name = "~/lib/acid"      |     <-- userinit file
|   fin  = Biobuf(a.out)     |         (was reading at line 12
|   line = 12                |         when it finished and was
|   prev -+                  |         NOT yet popped in this pic)
+---------|------------------+
```

```
          v
  +---------------------------+
  | IOstack                   |
  |   name = "/lib/acid/port" |
  |   fin  = Biobuf(...)      |
  |   line = 47               |
  |   prev -> nil             |     <-- bottom (initial push)
  +---------------------------+
```

`pushfile()`[100d] allocates a new frame, stashes the current global `line` into the *old* top's `line` field, and makes its own frame the new top. `popio()`[101e] reverses that: close the `Biobuf`, restore `line` from the frame it is about to discard, and point `lexio` at `prev`. Because all the state the lexer needs—the `Biobuf`, the filename for error messages, the line counter—lives in one struct, the lexer itself (`yylex()`[103a]) does not need to know about includes at all: it reads a character through `lexc()`[102a], and whenever the current `fin` hits EOF the pop happens transparently underneath.

⟨*struct* `IOstack` 100a⟩≡ (381b)
```
struct IOstack
{
    // ref_own<string>    included filename or "<stdin>"  or "<string>"
    char *name;
    // option<ref_own<Biobuf>> (None when <string>)
    Biobuf *fin;

    // saved global line to be restored in popio()
    int line;

    ⟨IOstack other fields 143a⟩
    // Extra
    ⟨IOstack extra fields 100c⟩
};
```

⟨*global* `lexio` 100b⟩≡ (381b)
```
// list<ref_own<IOstack> next = lexio.prev
IOstack *lexio;
```

⟨`IOstack` *extra fields* 100c⟩≡ (100a)
```
IOstack *prev;
```

9.1.2 Pushing a file: `pushfile()`

⟨*function* `pushfile` 100d⟩≡ (381b)
```
/// main | loadmodule -> <>
void
pushfile(char *file)
{
    Biobuf *b;
    IOstack *io;

    if(file)
        b = Bopen(file, OREAD);
    else{
        b = Bopen("/fd/0", OREAD);
        file = "<stdin>";
    }

    ⟨pushfile() sanity check b 101a⟩
```

```
    io = malloc(sizeof(IOstack));
    ⟨pushfile() sanity check io 101b⟩
    io->name = strdup(file);
    ⟨pushfile() sanity check io->name 101c⟩

    io->line = line;
    line = 1;
    io->fin = b;
    io->text = nil;

    // add_list(io, lexio)
    io->prev = lexio;
    lexio = io;
}
```

Uses lexio 100b.

⟨pushfile() *sanity check* b 101a⟩≡ (100d)

```
if(b == nil)
    error("pushfile: %s: %r", file);
```

⟨pushfile() *sanity check* io 101b⟩≡ (100d)

```
if(io == nil)
    fatal("no memory");
```

⟨pushfile() *sanity check* io->name 101c⟩≡ (100d)

```
if(io->name == nil)
    fatal("no memory");
```

9.1.3 Restoring input: restartio(), popio()

After yyparse() returns from processing one statement at the acid prompt, the Biobuf in lexio->fin may still have buffered input from the line the user just typed. restartio()[101d] flushes any leftover bytes and re-initializes the buffer to read fresh input from STDIN. Without this reset, stray characters could leak from one prompt to the next. popio()[101e] is the inverse of pushfile()[100d]: it closes the current input source and pops back to the previous one in the stack. It returns true as long as there is something to pop, so it can be used as a loop condition: while(popio()) ; unwinds the entire stack (used after errors).

⟨*function* restartio 101d⟩≡ (381b)

```
/// main -> yyparse; <>
void
restartio(void)
{
    Bflush(lexio->fin);
    Binit(lexio->fin, STDIN, OREAD);
}
```

Uses lexio 100b.

⟨*function* popio 101e⟩≡ (381b)

```
/// loadmodule | error -> <>
bool
popio(void)
{
    IOstack *s;

    if(lexio == nil)
        return false;
```

```
    if(lexio->prev == nil){
        if(lexio->fin)
            restartio();
        return false;
    }
    // else

    if(lexio->fin)
        Bterm(lexio->fin);
    else
       ⟨popio() when no fin 143f⟩
    free(lexio->name);

    // restore global line
    line = lexio->line;

    // s = pop_list(lexio)
    s = lexio;
    lexio = s->prev;
    free(s);

    return true;
}
```

Uses lexio 100b and restartio() 101d.

9.2 Lexer

With the input source machinery in place, we can now look at the lexer itself. The acid lexer is hand-written (not generated by lex/flex) and follows the classic character-by-character pattern: a low-level reader lexc()[102a] that gets one character at a time from the current input source, and a higher-level yylex()[103a] that recognizes tokens. The lexer is small enough (about 300 lines) that hand-writing it is simpler than learning lex's syntax.

9.2.1 Character reader: lexc() and unlexc()

lexc()[102a] is the low-level character reader. It reads from a Biobuf file in lexio->fin (or as we well see later from lexio->text, an in-memory string, used by interpret()[142c]). unlexc()[102b] pushes one character back.

⟨*function* lexc 102a⟩≡ (381b)

```
int
lexc(void)
{
    int c;

    if(lexio->fin) {
        c = Bgetc(lexio->fin);
        ⟨lexc() if gotint 208b⟩
        return c;
    }
    // else
    ⟨lexc() when no fin 143g⟩
}
```

Uses lexio 100b.

⟨*function* unlexc 102b⟩≡ (381b)

```
void
unlexc(int s)
```

```
{
    if(s == '\n')
        line--;

    if(lexio->fin)
        Bungetc(lexio->fin);
    else
        ⟨unlexc() when no fin 144a⟩
}
```

Uses lexio 100b.

9.2.2 Token reader: yylex()

yylex()[103a] is the main lexer function, called by the yacc-generated parser. It skips whitespace, handles comments (// style), and dispatches on the first character of each token. An important detail: in interactive mode, a bare newline (not inside braces) is returned as ;, making semicolons optional at the prompt.

⟨function yylex 103a⟩≡ (381b)

```
/// main -> yyparse -> <>
int
yylex(void)
{
    int c;
    ⟨yylex other locals 110b⟩

loop:
    Bflush(bout);
    c = lexc();
    switch(c) {
    ⟨yylex() switch c cases 103c⟩
    }
}
```

Uses bout 69d and lexc() 102a.

⟨acid constants 103b⟩+≡ (378a) ◁83a 108a▷

```
Eof  = -1,
```

⟨yylex() switch c cases 103c⟩≡ (103a) 103d▷

```
case Eof:
    ⟨yylex() when Eof, if gotint 208d⟩
    return Eof;
```

Uses Eof 103b.

⟨yylex() switch c cases 103d⟩+≡ (103a) ◁103c 103e▷

```
case '.':
    c = lexc();
    unlexc(c);
    if(isdigit(c))
        return numsym('.');

    return '.';
```

Uses lexc() 102a, numsym() 104b, and unlexc() 102b.

⟨yylex() switch c cases 103e⟩+≡ (103a) ◁103d 104c▷

```
default:
    return numsym(c);
```

Uses numsym() 104b.

⟨*global* `symbol` 104a⟩≡ (381a)
```
char symbol[Strsize];
```
Uses `Strsize` 108a.

⟨*function* `numsym` 104b⟩≡ (381b)
```
int
numsym(char first)
{
    char *p;
    int c;
    ⟨numsym() locals 105b⟩

    symbol[0] = first;
    p = symbol;

    ⟨numsym() if number 105c⟩
    // else
    ⟨numsym() when symbol 111a⟩
}
```
Uses `symbol` 104a.

9.2.3 Spaces and comments

⟨`yylex()` *switch* `c` *cases* 104c⟩+≡ (103a) ◁103e 104d▷
```
case ' ':
case '\t':
    goto loop;
```

⟨`yylex()` *switch* `c` *cases* 104d⟩+≡ (103a) ◁104c 105a▷
```
case '/':
    c = lexc();
    //pad: the '*' case is just to support syncweb comments
    if(c == '/' || c == '*') {
        eatnl();
        goto loop;
    }
    unlexc(c);
    return '/';
```

Uses `eatnl()` 104e, `lexc()` 102a, and `unlexc()` 102b.

⟨*function* `eatnl` 104e⟩≡ (381b)
```
void
eatnl(void)
{
    int c;

    line++;
    for(;;) {
        c = lexc();
        if(c == Eof)
            error("eof in comment");
        if(c == '\n')
            return;
    }
}
```
Uses `Eof` 103b and `lexc()` 102a.

9.2.4 Newlines

Newline handling depends on the `interactive` global we saw earlier. In non-interactive mode (loading a library file), a newline is just whitespace—the lexer increments `line` and loops back. In interactive mode, a newline at the top level (outside braces) terminates the statement and returns `';'` to the parser, so the user does not have to type explicit semicolons at the prompt. Inside braces (`stacked > 0`), the newline is consumed but the lexer prints a tab as a continuation prompt and loops back, allowing multi-line function definitions.

⟨yylex() *switch* c *cases* 105a⟩+≡ (103a) ◁104d 106a▷

```
case '\n':
    line++;
    if(!interactive)
        goto loop;
    if(stacked) {
        print("\t");
        goto loop;
    }
    // else when interactive and not stacked
    return ';';
```

Uses `interactive` 70a and `stacked` 71a.

9.2.5 Numbers

⟨numsym() *locals* 105b⟩≡ (104b) 110e▷

```
bool isbin, isfloat, ishex;
char *sel;
```

⟨numsym() *if number* 105c⟩≡ (104b)

```
ishex = false;
isbin = false;
isfloat = false;
if(first == '.')
    isfloat = true;

if(isdigit(*p++) || isfloat) {
    for(;;) {
        c = lexc();
        if(c < 0)
            error("%d: <eof> eating symbols", line);

        if(c == '\n')
            line++;
        sel = "01234567890.xb";
        if(ishex)
            sel = "01234567890abcdefABCDEF";
        else if(isbin)
            sel = "01";
        else if(isfloat)
            sel = "01234567890eE-+";

        if(strchr(sel, c) == 0) {
            unlexc(c);
            break;
        }
        if(c == '.')
            isfloat = true;
        if(!isbin && c == 'x')
            ishex = true;
        if(!ishex && c == 'b')
```

```
            isbin = true;
        *p++ = c;
    }
    *p = '\0';
    if(isfloat) {
        yylval.fval = atof(symbol);
        return Tfconst;
    }

    if(isbin)
        yylval.ival = strtoull(symbol+2, 0, 2);
    else
        yylval.ival = strtoull(symbol, 0, 0);
    return Tconst;
}
```

Uses Tconst, Tfconst, lexc() 102a, symbol 104a, unlexc() 102b, and yylval.

9.2.6 Characters

⟨yylex() *switch* c *cases* 106a⟩+≡ (103a) ◁105a 107b▷

```
case '\'':
    c = lexc();
    if(c == '\\')
        yylval.ival = escchar(lexc());
    else
        yylval.ival = c;
    c = lexc();
    if(c != '\'') {
        error("missing '");
        unlexc(c);
    }
    return Tconst;
```

Uses Tconst, escchar() 106b, lexc() 102a, unlexc() 102b, and yylval.

⟨*function* escchar 106b⟩≡ (381b)

```
int
escchar(char c)
{
    int n;
    char buf[Strsize];

    if(c >= '0' && c <= '9') {
        n = 1;
        buf[0] = c;
        for(;;) {
            c = lexc();
            if(c == Eof)
                error("%d: <eof> in escape sequence", line);
            if(strchr("0123456789xX", c) == 0) {
                unlexc(c);
                break;
            }
            if(n >= Strsize)
                error("string escape too long");
            buf[n++] = c;
        }
        buf[n] = '\0';
```

```
        return strtol(buf, 0, 0);
    }

    n = cmap[c];
    if(n == 0)
        return c;
    return n-1;
}
```

Uses Eof 103b, Strsize 108a, cmap 107a, lexc() 102a, and unlexc() 102b.

⟨*global* cmap 107a⟩≡ (381b)

```
char cmap[256] =
{
    ['0'] '\0'+1,
    ['n'] '\n'+1,
    ['r'] '\r'+1,
    ['t'] '\t'+1,
    ['b'] '\b'+1,
    ['f'] '\f'+1,
    ['a'] '\a'+1,
    ['v'] '\v'+1,
    ['\\'] '\\'+1,
    ['"'] '"'+1,
};
```

9.2.7 Strings

⟨yylex() *switch* c *cases* 107b⟩+≡ (103a) ◁106a 108b▷

```
case '"':
    eatstring();
    return Tstring;
```

Uses Tstring and eatstring() 107c.

⟨*function* eatstring 107c⟩≡ (381b)

```
void
eatstring(void)
{
    int esc, c, cnt;
    char buf[Strsize];

    esc = 0;
    for(cnt = 0;;) {
        c = lexc();
        switch(c) {
        case Eof:
            error("%d: <eof> in string constant", line);

        case '\n':
            error("newline in string constant");
            goto done;

        case '\\':
            if(esc)
                goto Default;
            esc = 1;
            break;

        case '"':
```

```
            if(esc == 0)
                goto done;

            /* Fall through */
        default:
        Default:
            if(esc) {
                c = escchar(c);
                esc = 0;
            }
            buf[cnt++] = c;
            break;
        }
        if(cnt >= Strsize)
            error("string token too long");
    }
done:
    buf[cnt] = '\0';
    yylval.string = strnode(buf);
}
```

Uses Eof 103b, Strsize 108a, escchar() 106b, lexc() 102a, strnode() 62g, and yylval.

⟨*acid constants* 108a⟩+≡ (378a) ◁103b 151c▷

```
Strsize  = 4096,
```

9.2.8 Operators and punctuations

⟨yylex() *switch* c *cases* 108b⟩+≡ (103a) ◁107b 108c▷

```
case '(':
case ')':
case '[':
case ']':
case ';':
case ':':
case ',':
case '~':
case '?':
case '*':
case '@':
case '^':
case '%':
    return c;
```

⟨yylex() *switch* c *cases* 108c⟩+≡ (103a) ◁108b 108d▷

```
case '{':
    stacked++;
    return c;
case '}':
    stacked--;
    return c;
```

Uses stacked 71a.

⟨yylex() *switch* c *cases* 108d⟩+≡ (103a) ◁108c 109a▷

```
case '!':
    c = lexc();
    if(c == '=')
        return Tneq;
    unlexc(c);
```

```
        return '!';
```

Uses Tneq, lexc() 102a, and unlexc() 102b.

⟨yylex() *switch* c *cases* 109a⟩+≡ (103a) ◁108d 109b▷
```
    case '+':
        c = lexc();
        if(c == '+')
            return Tinc;
        unlexc(c);
        return '+';
```

Uses Tinc, lexc() 102a, and unlexc() 102b.

⟨yylex() *switch* c *cases* 109b⟩+≡ (103a) ◁109a 109c▷
```
    case '&':
        c = lexc();
        if(c == '&')
            return Tandand;
        unlexc(c);
        return '&';
```

Uses Tandand, lexc() 102a, and unlexc() 102b.

⟨yylex() *switch* c *cases* 109c⟩+≡ (103a) ◁109b 109d▷
```
    case '=':
        c = lexc();
        if(c == '=')
            return Teq;
        unlexc(c);
        return '=';
```

Uses Teq, lexc() 102a, and unlexc() 102b.

⟨yylex() *switch* c *cases* 109d⟩+≡ (103a) ◁109c 109e▷
```
    case '|':
        c = lexc();
        if(c == '|')
            return Toror;
        unlexc(c);
        return '|';
```

Uses Toror, lexc() 102a, and unlexc() 102b.

⟨yylex() *switch* c *cases* 109e⟩+≡ (103a) ◁109d 110a▷
```
    case '<':
        c = lexc();
        if(c == '=')
            return Tleq;
        if(c == '<')
            return Tlsh;
        unlexc(c);
        return '<';

    case '>':
        c = lexc();
        if(c == '=')
            return Tgeq;
        if(c == '>')
```

```
        return Trsh;
    unlexc(c);
    return '>';
```

Uses Tgeq, Tleq, Tlsh, Trsh, lexc() 102a, and unlexc() 102b.

⟨yylex() *switch* c *cases* 110a⟩+≡ (103a) ◁109e 110d▷
```
  case '-':
      c = lexc();

      if(c == '>')
          return Tindir;

      if(c == '-')
          return Tdec;
      unlexc(c);
      return '-';
```

Uses Tdec, Tindir, lexc() 102a, and unlexc() 102b.

9.2.9 Format X\F

The backslash starts a format suffix: a single character that controls how an expression is printed. For example, x\D means "print x in decimal" and *PC\i means "disassemble the instruction at PC". The lexer checks the character after the backslash against vfmt (the table of valid format characters defined in print.c); if valid, it returns a Tfmt token carrying the format character as its value, which the parser combines into an OFMT AST node. If the character is not a known format, the lexer treats the backslash as a stray character and returns it literally.

⟨yylex *other locals* 110b⟩≡ (103a)
```
  extern char vfmt[];
```

⟨*global* vfmt 110c⟩≡ (423b)
```
  char vfmt[] = "aBbcCdDfFgGiIoOqQrRsSuUVWxXYZ38";
```
Uses vfmt 110c.

⟨yylex() *switch* c *cases* 110d⟩+≡ (103a) ◁110a
```
  case '\\':
      c = lexc();
      if(strchr(vfmt, c) == 0) {
          unlexc(c);
          return '\\';
      }
      yylval.ival = c;
      return Tfmt;
```

Uses Tfmt, lexc() 102a, unlexc() 102b, vfmt 110c, and yylval.

9.2.10 Symbols

⟨numsym() *locals* 110e⟩+≡ (104b) ◁105b
```
  Lsym *s;
```

⟨numsym() *when symbol* 111a⟩≡ (104b)

```
for(;;) {
    c = lexc();
    if(c < 0)
        error("%d <eof> eating symbols", line);
    if(c == '\n')
        line++;
    if(c != '_' && c != '$' && c <= '~' && !isalnum(c)) { /* checking against ~ lets UTF names through */
        unlexc(c);
        break;
    }
    *p++ = c;
}

*p = '\0';

s = look(symbol);
if(s == 0)
    s = enter(symbol, Tid);

yylval.sym = s;
return s->lexval;
```

Uses Tid, enter() 67b, lexc() 102a, look() 66d, symbol 104a, unlexc() 102b, and yylval.

9.2.11 Keywords

Keywords are handled through the symbol table rather than special-casing them in the lexer. `kinit()`[112a] enters all keywords into the hash table with their token codes; when `numsym()`[104b] looks up an identifier, keywords are found and their `lexval` (e.g., `Tif`, `Twhile`, `Tfn`) is returned instead of `Tid`.

⟨*global* keywds 111b⟩≡ (381b)

```
struct keywd
{
    char *name;
    int terminal;
}
keywds[] =
{
    "do", Tdo,
    "if", Tif,
    "then", Tthen,
    "else", Telse,
    "while", Twhile,
    "loop", Tloop,

    "head", Thead,
    "tail", Ttail,
    "append", Tappend,

    "defn", Tfn,
    "return", Tret,
    "local", Tlocal,
    "aggr", Tcomplex,
    "union", Tcomplex,
    "adt", Tcomplex,
    "complex", Tcomplex,
    "delete", Tdelete,
    "whatis", Twhat,
    "eval", Teval,
```

```
    "builtin", Tbuiltin,

    0,  0
};
```

Uses Tappend, Tbuiltin, Tcomplex, Tdelete, Tdo, Telse, Teval, Tfn, Thead, Tif, Tlocal, Tloop, Tret, Ttail, Tthen, Twhat, Twhile, and keywd 111b.

⟨*function* kinit 112a⟩≡ (381b)

```
void
kinit(void)
{
    int i;

    for(i = 0; keywds[i].name; i++)
        enter(keywds[i].name, keywds[i].terminal);
}
```

Uses enter() 67b and keywds 111b.

9.3 Grammar

The acid grammar is specified in yacc format and is relatively small—about 150 lines of rules. The grammar is C-like with a few differences: defn instead of function declarations, then/do keywords after conditions, list operations (head, tail, append), and complex/aggr for type descriptions. Most grammar rules build AST nodes using an()[58c].

9.3.1 Overview

⟨*acid/dbg.y* 112b⟩≡

```
%{
#include <u.h>
#include <libc.h>
#include <bio.h>
#include <mach.h>

#include "acid.h"
%}
⟨union yacc 57b⟩
⟨type declarations 112d⟩
⟨priority and associativity declarations 113a⟩
⟨token declarations 57a⟩
%%
⟨grammar 113b⟩
%%
```

⟨union yacc *other fields* 112c⟩≡ (57b)

```
Node *node;
```

⟨*type declarations* 112d⟩≡ (112b)

```
%type <node> expr monexpr term stmnt name args zexpr slist
%type <node> member members mname castexpr idlist
%type <sym> zname
```

⟨*priority and associativity declarations* 113a⟩≡ (112b)

```
%left ';'
%right '='
%left Tfmt
%left Toror
%left Tandand
%left '|'
%left '^'
%left '&'
%left Teq Tneq
%left '<' '>' Tleq Tgeq
%left Tlsh Trsh
%left '+' '-'
%left '*' '/' '%'
%right Tdec Tinc Tindir '.' '[' '('
```

⟨*grammar* 113b⟩≡ (112b)

```
prog
  : /* empty */
  | prog bigstmnt
  ;

⟨statements rules 114a⟩
⟨members rules 118c⟩
⟨expressions rules 116a⟩
⟨name rules 113c⟩
⟨other rules 114b⟩
```

A `name` is either a bare identifier (e.g., `main`, `x`) or a qualified name of the form `func:var` (e.g., `main:argc`). The qualified form is what lets the user access local variables and parameters of a specific function: `main:argc` means "the parameter `argc` in function `main`". The grammar produces an `OFRAME` node carrying the function symbol on the left and the variable name on the right; at runtime, `oframe()`[403a] uses `localaddr()`[355f] to compute the actual stack address from the function's frame pointer and the variable's frame offset (read from the `a/p` symbol table entries).

⟨*name rules* 113c⟩≡ (113b) 113d▷

```
name
  : Tid
  {
   $$ = an(ONAME, ZN, ZN);
   $$->sym = $1;
  }
  | Tid ':' name
  {
   $$ = an(OFRAME, $3, ZN);
   $$->sym = $1;
  }
  ;
```

⟨*name rules* 113d⟩+≡ (113b) ◁113c

```
zname
  :
  /* empty */
  { $$ = 0; }
  | Tid
  ;
```

9.3.2 Definitions

The `bigstmnt` rule is the top-level entry point. It handles three cases: a statement (which is immediately executed via `execute()`[120a] and then garbage-collected), a function definition (`defn name(args) \ body \`), and a complex type declaration (`aggr name \ members \`). The immediate execution is the key design choice—`acid` does not separate parsing from execution for interactive statements.

⟨*statements rules* 114a⟩≡ (113b) 114c▷

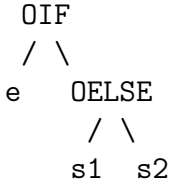
```
bigstmnt
  : stmnt
  {
   /* make stmnt a root so it isn't collected! */
   mkvar("_thiscmd")->proc = $1;
   execute($1);
   mkvar("_thiscmd")->proc = nil;
   gc();
   if(interactive)
      Bprint(bout, "acid: ");
  }
  | Tfn Tid '(' args ')' zsemi '{' slist '}'
  {
   $2->proc = an(OLIST, $4, $8);
  }
  | Tfn Tid
  {
   $2->proc = nil;
  }
  | Tcomplex name '{' members '}' ';'
  {
   defcomplex($2, $4);
  }
  ;
```

⟨*other rules* 114b⟩≡ (113b) 115a▷

```
zsemi
  : /* empty */
  | ';' zsemi
```

9.3.3 Statements

The `stmnt` rule covers all statement forms. Each rule builds a single AST node carrying the operator and one or two sub-trees. A subtle point is the encoding of `if/else` and `loop`: they pack two operands into one `Node` slot by introducing an intermediate `OELSE` or `OLIST` node. For example, `if e then s1 else s2` becomes:



This trick is how a binary tree encodes a 3-ary node. We will see the same trick in `loop start, end do body`: `start` and `end` are wrapped in an `OLIST` under the `ODO` node's left child.

⟨*statements rules* 114c⟩+≡ (113b) ◁114a 115b▷

```
stmnt
  : zexpr ';'
  | '{' slist '}'
  {
```

```
    $$ = $2;
   }
   | Tif expr Tthen stmnt
   {
    $$ = an(OIF, $2, $4);
   }
   | Tif expr Tthen stmnt Telse stmnt
   {
    $$ = an(OIF, $2, an(OELSE, $4, $6));
   }
   | Tloop expr ',' expr Tdo stmnt
   {
    $$ = an(ODO, an(OLIST, $2, $4), $6);
   }
   | Twhile expr Tdo stmnt
   {
    $$ = an(OWHILE, $2, $4);
   }
   | Tret expr ';'
   {
    $$ = an(ORET, $2, ZN);
   }
   | Tlocal idlist
   {
    $$ = an(OLOCAL, $2, ZN);
   }
   | Tcomplex Tid name ';'
   {
    $$ = an(OCOMPLEX, $3, ZN);
    $$->sym = $2;
   }
   ;
```

⟨*other rules* 115a⟩+≡ (113b) ◁114b

```
idlist
   : Tid
   {
    $$ = an(ONAME, ZN, ZN);
    $$->sym = $1;
   }
   | idlist ',' Tid
   {
    $$ = an(ONAME, $1, ZN);
    $$->sym = $3;
   }
   ;
```

⟨*statements rules* 115b⟩+≡ (113b) ◁114c

```
slist
   : stmnt
   | slist stmnt
   {
    $$ = an(OLIST, $1, $2);
   }
   ;
```

9.3.4 Expressions

⟨*expressions rules* 116a⟩≡ (113b) 116b▷

```
expr
  : castexpr
  | expr '*' expr { $$ = an(OMUL, $1, $3); }
  | expr '/' expr { $$ = an(ODIV, $1, $3); }
  | expr '%' expr { $$ = an(OMOD, $1, $3); }
  | expr '+' expr { $$ = an(OADD, $1, $3); }
  | expr '-' expr { $$ = an(OSUB, $1, $3); }

  | expr Trsh expr { $$ = an(ORSH, $1, $3); }
  | expr Tlsh expr { $$ = an(OLSH, $1, $3); }

  | expr '<' expr  { $$ = an(OLT, $1, $3); }
  | expr '>' expr  { $$ = an(OGT, $1, $3); }
  | expr Tleq expr { $$ = an(OLEQ, $1, $3); }
  | expr Tgeq expr { $$ = an(OGEQ, $1, $3); }
  | expr Teq expr  { $$ = an(OEQ, $1, $3); }
  | expr Tneq expr { $$ = an(ONEQ, $1, $3); }

  | expr '&' expr { $$ = an(OLAND, $1, $3); }
  | expr '^' expr { $$ = an(OXOR, $1, $3); }
  | expr '|' expr { $$ = an(OLOR, $1, $3); }

  | expr Tandand expr { $$ = an(OCAND, $1, $3); }
  | expr Toror expr   { $$ = an(OCOR, $1, $3); }

  | expr '=' expr { $$ = an(OASGN, $1, $3); }
  | expr Tfmt { $$ = an(OFMT, $1, con($2)); }
  ;
```

⟨*expressions rules* 116b⟩+≡ (113b) ◁116a 116c▷

```
castexpr
  : monexpr
  | '(' Tid ')' monexpr
  {
   $$ = an(OCAST, $4, ZN);
   $$->sym = $2;
  }
  ;
```

⟨*expressions rules* 116c⟩+≡ (113b) ◁116b 117▷

```
monexpr
  : term
  | '*' monexpr { $$ = an(OINDM, $2, ZN); }
  | '@' monexpr { $$ = an(OINDC, $2, ZN); }
  | '+' monexpr { $$ = an(OADD, $2, ZN); }
  | '-' monexpr { $$ = con(0); $$ = an(OSUB, $$, $2); }

  | Tdec monexpr { $$ = an(OEDEC, $2, ZN); }
  | Tinc monexpr { $$ = an(OEINC, $2, ZN); }

  | Thead monexpr { $$ = an(OHEAD, $2, ZN); }
  | Ttail monexpr { $$ = an(OTAIL, $2, ZN); }
  | Tappend monexpr ',' monexpr { $$ = an(OAPPEND, $2, $4); }
  | Tdelete monexpr ',' monexpr { $$ = an(ODELETE, $2, $4); }

  | '!' monexpr { $$ = an(ONOT, $2, ZN); }
  | '~' monexpr { $$ = an(OXOR, $2, con(-1)); }
```

```
    | Teval monexpr { $$ = an(OEVAL, $2, ZN);}
    ;
```

⟨*expressions rules* 117⟩+≡ (113b) ◁116c 118a▷

```
  term
    : '(' expr ')'
    {
     $$ = $2;
    }
    | '{' args '}'
    {
     $$ = an(OCTRUCT, $2, ZN);
    }
    | term '[' expr ']'
    {
     $$ = an(OINDEX, $1, $3);
    }
    | term Tdec
    {
     $$ = an(OPDEC, $1, ZN);
    }
    | term '.' Tid
    {
     $$ = an(ODOT, $1, ZN);
     $$->sym = $3;
    }
    | term Tindir Tid
    {
     $$ = an(ODOT, an(OINDM, $1, ZN), ZN);
     $$->sym = $3;
    }
    | term Tinc
    {
     $$ = an(OPINC, $1, ZN);
    }
    | name '(' args ')'
    {
     $$ = an(OCALL, $1, $3);
    }
    | Tbuiltin name '(' args ')'
    {
     $$ = an(OCALL, $2, $4);
     $$->builtin = 1;
    }
    | name
    | Tconst
    {
     $$ = con($1);
    }
    | Tfconst
    {
     $$ = an(OCONST, ZN, ZN);
     $$->type = TFLOAT;
     $$->fmt = 'f';
     $$->fval = $1;
    }
    | Tstring
    {
     $$ = an(OCONST, ZN, ZN);
     $$->type = TSTRING;
```

```
    $$->string = $1;
    $$->fmt = 's';
   }
   | Twhat zname
   {
    $$ = an(OWHAT, ZN, ZN);
    $$->sym = $2;
   }
   ;
```

⟨*expressions rules* 118a⟩+≡ (113b) ◁117 118b▷

```
  args
    : zexpr
    | args ',' zexpr
    {
     $$ = an(OLIST, $1, $3);
    }
    ;
```

⟨*expressions rules* 118b⟩+≡ (113b) ◁118a

```
  zexpr
    : /* empty */
    { $$ = 0; }
    | expr
    ;
```

9.3.5 Complex

⟨*members rules* 118c⟩≡ (113b)

```
  members
    : member
    | members member
    {
     $$ = an(OLIST, $1, $2);
    }
    ;

  mname
    : Tid
    {
     $$ = an(ONAME, ZN, ZN);
     $$->sym = $1;
    }
    ;

  member
    : Tconst Tconst mname ';'
    {
     $3->ival = $2;
     $3->fmt = $1;
     $$ = $3;
    }
    | Tconst mname Tconst mname ';'
    {
     $4->ival = $3;
     $4->fmt = $1;
     $4->right = $2;
     $$ = $4;
    }
```

```
| mname Tconst mname ';'
{
 $3->ival = $2;
 $3->left = $1;
 $$ = $3;
}
| '{' members '}' ';'
{
 $$ = an(OCTRUCT, $2, ZN);
}
;
```

Chapter 10

Interpreting `acid` code

This chapter presents the `acid` interpreter: the tree-walking evaluator that executes AST nodes produced by the parser. The interpreter is split into two parts: `execute()`[120a] handles statements (control flow, assignments, function calls), while the `expr()` macro dispatches expression evaluation through a function pointer table `expop`[121f].

10.1 Interpreter entry point: `execute()`

`execute()`[120a] is the entry point of the interpreter. It is called from `yyparse()` (via the `bigstmnt` grammar rule) on each top-level AST node and handles both statements and expressions by dispatching on the node's opcode. Control-flow statements (`OIF`, `OWHILE`, `ODO`), local declarations (`OLOCAL`), and returns (`ORET`) are handled directly by the switch; expressions fall through to the `default` case, which delegates to `expr()` (the expression evaluator described in the next section). Assignments and calls (`OASGN`, `OCALL`) get their own cases so that `expr()` is called without auto-printing the result.

⟨*function* `execute` 120a⟩≡ (391)
```
/// main (acidmap) | ?? -> <>
void
execute(Node *n)
{
    Node *l, *r;
    Value *v;
    Node res;
    ⟨execute() other locals 120b⟩

    gc();
    ⟨execute() if gotint 208c⟩
    if(n == nil)
        return;
    ⟨execute() if big statements count 121a⟩

    l = n->left;
    r = n->right;
    switch(n->op) {
    ⟨execute() switch n->op cases 121c⟩
    }
}
```
Uses `gc()` 149.

⟨`execute()` *other locals* 120b⟩≡ (120a) 121b▷
```
static int stmnt;
```

⟨execute() *if big statements count* 121a⟩≡ (120a)
```
if(stmnt++ > 5000) {
    Bflush(bout);
    stmnt = 0;
}
```
Uses bout 69d.

⟨execute() *other locals* 121b⟩+≡ (120a) ◁120b
```
Lsym *sl;
vlong i, s, e;
Node xx;
```

10.2 Expressions

Expression evaluation uses a dispatch table: the expop[121f] array maps each opcode to its evaluation function. The expr() macro indexes into this table and calls the corresponding function. Most expression evaluators follow the same pattern: evaluate left and right children recursively, check types, compute the result, and store it in res with op set to OCONST.

The default case is more subtle than it looks: after evaluating the expression, it auto-prints the result by splicing it into the prnt node and re-evaluating that. This is what makes the acid REPL behave like "type *main and the value appears"—there is no explicit print call, the interpreter prints any non-empty top-level expression result automatically. The ret || (...empty list...) condition skips auto-printing in two cases: (1) when a return is in flight (we should not print, we should propagate), and (2) when the result is the empty list (typically a function that returned nothing). OASGN and OCALL have their own cases because assignments and calls are evaluated for their side effects and should *not* auto-print—otherwise x = 42 would print 42 back after assigning, and every bpset(addr) call would dump its return value.

⟨execute() *switch* n->op *cases* 121c⟩≡ (120a) 121d▷
```
default:
    expr(n, &res);

    if(ret || (res.type == TLIST && res.l == 0 && n->op != OADD))
        break;

    prnt->right = &res;
    expr(prnt, &xx);

    break;
```
Uses OADD 59, TLIST 60a, prnt 144b, and ret 134b.

⟨execute() *switch* n->op *cases* 121d⟩+≡ (120a) ◁121c 135b▷
```
case OASGN:
case OCALL:
    expr(n, &res);
    break;
```
Uses OASGN 59 and OCALL 59.

⟨*macro* expr 121e⟩≡ (378b)
```
#define expr(n, r) do{(r)->comt=nil; (*expop[(n)->op])(n, r);}while(0)
```

⟨*global* expop 121f⟩≡ (407c)
```
void (*expop[])(Node*, Node*) =
{
    ⟨expop entries 122⟩
};
```

⟨expop *entries* 122⟩≡ (121f) 195a ▷

```
[ONAME] oname,
[OCONST] oconst,
[OMUL] omul,
[ODIV] odiv,
[OMOD] omod,
[OADD] oadd,
[OSUB] osub,
[ORSH] orsh,
[OLSH] olsh,
[OLT] olt,
[OGT] ogt,
[OLEQ] oleq,
[OGEQ] ogeq,
[OEQ] oeq,
[ONEQ] oeq,
[OLAND] oland,
[OXOR] oxor,
[OLOR] olor,
[OCAND] ocand,
[OCOR] ocor,
[OASGN] oasgn,
[OINDM] oindm,
[OEDEC] oeinc,
[OEINC] oeinc,
[OPINC] opinc,
[OPDEC] opinc,
[ONOT] onot,
[OIF] 0,
[ODO] 0,
[OLIST] olist,
[OCALL] ocall,
[OCTRUCT] octruct,
[OWHILE] 0,
[OELSE] 0,
[OHEAD] ohead,
[OTAIL] otail,
[OAPPEND] oappend,
[ORET] 0,
[OINDEX] oindex,
[OINDC] oindc,
[ODOT] odot,
[OLOCAL] 0,
[OFRAME] oframe,
[OCOMPLEX] 0,
[ODELETE] odelete,
[OCAST] ocast,
[OFMT] ofmt,
[OEVAL] oeval,
```

Uses oadd() 129, oappend() 404a, oasgn() 406b, ocall() 135d, ocand() 128a, ocast() 402c, oconst() 405c, ocor() 128c, octruct() 406a, odelete() 404b, odiv() 132a, odot() 398b, oeinc() 406c, oeq() 126, oeval() 402b, ofmt() 407b, ogeq() 125b, ogt() 124, ohead() 405a, oindc() 402e, oindex() 403b, oindm() 402d, oland() 127a, oleq() 125a, olist() 402a, olor() 127c, olsh() 133a, olt() 123b, omod() 132b, omul() 131, oname() 405d, onot() 128b, opinc() 407a, orsh() 133b, osub() 130, otail() 405b, and oxor() 127b.

10.2.1 Boolean operations

`fbool()`[123a] converts a value to boolean: nonzero integers, nonzero floats, nonempty strings, and nonempty lists are true. This is used by `if`, `while`, `&&`, and `||`.

⟨*function* `fbool` 123a⟩≡ (391)

```
int
fbool(Node *n)
{
    int true = 0;

    if(n->op != OCONST)
        fatal("fbool: not const");

    switch(n->type) {
    case TINT:
        if(n->ival != 0)
            true = 1;
        break;
    case TFLOAT:
        if(n->fval != 0.0)
            true = 1;
        break;
    case TSTRING:
        if(n->string->len)
            true = 1;
        break;
    case TLIST:
        if(n->l)
            true = 1;
        break;
    }
    return true;
}
```

Uses `OCONST` 59, `TFLOAT` 60a, `TINT` 60a, `TLIST` 60a, and `TSTRING` 60a.

⟨*function* `olt` 123b⟩≡ (407c)

```
void
olt(Node *n, Node *res)
{
    Node l, r;

    expr(n->left, &l);
    expr(n->right, &r);

    res->fmt = l.fmt;
    res->op = OCONST;
    res->type = TINT;
    switch(l.type) {
    default:
        error("bad lhs type <");
    case TINT:
        switch(r.type) {
        case TINT:
            res->ival = l.ival < r.ival;
            break;
        case TFLOAT:
            res->ival = l.ival < r.fval;
            break;
        default:
```

```
            error("bad rhs type <");
        }
        break;
    case TFLOAT:
        switch(r.type) {
        case TINT:
            res->ival = l.fval < r.ival;
            break;
        case TFLOAT:
            res->ival = l.fval < r.fval;
            break;
        default:
            error("bad rhs type <");
        }
        break;
    }
}
```

Uses OCONST 59, TFLOAT 60a, and TINT 60a.

⟨*function* ogt 124⟩≡ (407c)

```
void
ogt(Node *n, Node *res)
{
    Node l, r;

    expr(n->left, &l);
    expr(n->right, &r);
    res->fmt = 'D';
    res->op = OCONST;
    res->type = TINT;
    switch(l.type) {
    default:
        error("bad lhs type >");
    case TINT:
        switch(r.type) {
        case TINT:
            res->ival = l.ival > r.ival;
            break;
        case TFLOAT:
            res->ival = l.ival > r.fval;
            break;
        default:
            error("bad rhs type >");
        }
        break;
    case TFLOAT:
        switch(r.type) {
        case TINT:
            res->ival = l.fval > r.ival;
            break;
        case TFLOAT:
            res->ival = l.fval > r.fval;
            break;
        default:
            error("bad rhs type >");
        }
        break;
    }
}
```

Uses OCONST 59, TFLOAT 60a, and TINT 60a.

⟨*function* oleq 125a⟩≡ (407c)

```
void
oleq(Node *n, Node *res)
{
    Node l, r;

    expr(n->left, &l);
    expr(n->right, &r);
    res->fmt = 'D';
    res->op = OCONST;
    res->type = TINT;
    switch(l.type) {
    default:
        error("bad expr type <=");
    case TINT:
        switch(r.type) {
        case TINT:
            res->ival = l.ival <= r.ival;
            break;
        case TFLOAT:
            res->ival = l.ival <= r.fval;
            break;
        default:
            error("bad expr type <=");
        }
        break;
    case TFLOAT:
        switch(r.type) {
        case TINT:
            res->ival = l.fval <= r.ival;
            break;
        case TFLOAT:
            res->ival = l.fval <= r.fval;
            break;
        default:
            error("bad expr type <=");
        }
        break;
    }
}
```

Uses OCONST 59, TFLOAT 60a, and TINT 60a.

⟨*function* ogeq 125b⟩≡ (407c)

```
void
ogeq(Node *n, Node *res)
{
    Node l, r;

    expr(n->left, &l);
    expr(n->right, &r);
    res->fmt = 'D';
    res->op = OCONST;
    res->type = TINT;
    switch(l.type) {
    default:
        error("bad lhs type >=");
    case TINT:
        switch(r.type) {
        case TINT:
            res->ival = l.ival >= r.ival;
```

```
            break;
        case TFLOAT:
            res->ival = l.ival >= r.fval;
            break;
        default:
            error("bad rhs type >=");
        }
        break;
    case TFLOAT:
        switch(r.type) {
        case TINT:
            res->ival = l.fval >= r.ival;
            break;
        case TFLOAT:
            res->ival = l.fval >= r.fval;
            break;
        default:
            error("bad rhs type >=");
        }
        break;
    }
}
```

Uses OCONST 59, TFLOAT 60a, and TINT 60a.

⟨*function* oeq 126⟩≡ (407c)

```
void
oeq(Node *n, Node *res)
{
    Node l, r;

    expr(n->left, &l);
    expr(n->right, &r);
    res->fmt = 'D';
    res->op = OCONST;
    res->type = TINT;
    res->ival = 0;
    switch(l.type) {
    default:
        break;
    case TINT:
        switch(r.type) {
        case TINT:
            res->ival = l.ival == r.ival;
            break;
        case TFLOAT:
            res->ival = l.ival == r.fval;
            break;
        default:
            break;
        }
        break;
    case TFLOAT:
        switch(r.type) {
        case TINT:
            res->ival = l.fval == r.ival;
            break;
        case TFLOAT:
            res->ival = l.fval == r.fval;
            break;
        default:
```

```
            break;
        }
        break;
    case TSTRING:
        if(r.type == TSTRING) {
            res->ival = scmp(r.string, l.string);
            break;
        }
        break;
    case TLIST:
        if(r.type == TLIST) {
            res->ival = listcmp(l.l, r.l);
            break;
        }
        break;
    }
    if(n->op == ONEQ)
        res->ival = !res->ival;
}
```

Uses OCONST 59, ONEQ 59, TFLOAT 60a, TINT 60a, TLIST 60a, TSTRING 60a, listcmp() 394b, and scmp() 386c.

⟨*function* oland 127a⟩≡ (407c)

```
void
oland(Node *n, Node *res)
{
    Node l, r;

    expr(n->left, &l);
    expr(n->right, &r);
    res->fmt = l.fmt;
    res->op = OCONST;
    res->type = TINT;
    if(l.type != TINT || r.type != TINT)
        error("bad expr type &");
    res->ival = l.ival&r.ival;
}
```

Uses OCONST 59 and TINT 60a.

⟨*function* oxor 127b⟩≡ (407c)

```
void
oxor(Node *n, Node *res)
{
    Node l, r;

    expr(n->left, &l);
    expr(n->right, &r);
    res->fmt = l.fmt;
    res->op = OCONST;
    res->type = TINT;
    if(l.type != TINT || r.type != TINT)
        error("bad expr type ^");
    res->ival = l.ival^r.ival;
}
```

Uses OCONST 59 and TINT 60a.

⟨*function* olor 127c⟩≡ (407c)

```
void
olor(Node *n, Node *res)
{
```

```
    Node l, r;

    expr(n->left, &l);
    expr(n->right, &r);
    res->fmt = l.fmt;
    res->op = OCONST;
    res->type = TINT;
    if(l.type != TINT || r.type != TINT)
        error("bad expr type |");
    res->ival = l.ival|r.ival;
}
```

Uses OCONST 59 and TINT 60a.

⟨*function* ocand 128a⟩≡ (407c)

```
void
ocand(Node *n, Node *res)
{
    Node l, r;

    res->op = OCONST;
    res->type = TINT;
    res->ival = 0;
    res->fmt = 'D';
    expr(n->left, &l);
    if(fbool(&l) == 0)
        return;
    expr(n->right, &r);
    if(fbool(&r) == 0)
        return;
    res->ival = 1;
}
```

Uses OCONST 59, TINT 60a, and fbool() 123a.

⟨*function* onot 128b⟩≡ (407c)

```
void
onot(Node *n, Node *res)
{
    Node l;

    res->op = OCONST;
    res->type = TINT;
    res->ival = 0;
    expr(n->left, &l);
    res->fmt = l.fmt;
    if(fbool(&l) == 0)
        res->ival = 1;
}
```

Uses OCONST 59, TINT 60a, and fbool() 123a.

⟨*function* ocor 128c⟩≡ (407c)

```
void
ocor(Node *n, Node *res)
{
    Node l, r;

    res->op = OCONST;
    res->type = TINT;
    res->ival = 0;
    res->fmt = 'D';
```

```
    expr(n->left, &l);
    if(fbool(&l)) {
        res->ival = 1;
        return;
    }
    expr(n->right, &r);
    if(fbool(&r)) {
        res->ival = 1;
        return;
    }
}
```

Uses OCONST 59, TINT 60a, and fbool() 123a.

10.2.2 Arithmetic operations

The arithmetic operators handle mixed int/float types with the expected promotion rules (int+float gives float). The + operator is also overloaded for string concatenation and list concatenation, making it the most complex of the arithmetic operators.

⟨*function* oadd 129⟩≡ (407c)

```
void
oadd(Node *n, Node *res)
{
    Node l, r;

    if(n->right == nil){  /* unary + */
        expr(n->left, res);
        return;
    }
    expr(n->left, &l);
    expr(n->right, &r);
    res->fmt = l.fmt;
    res->op = OCONST;
    res->type = TFLOAT;
    switch(l.type) {
    default:
        error("bad lhs type +");
    case TINT:
        switch(r.type) {
        case TINT:
            res->type = TINT;
            res->ival = l.ival+r.ival;
            break;
        case TFLOAT:
            res->fval = l.ival+r.fval;
            break;
        default:
            error("bad rhs type +");
        }
        break;
    case TFLOAT:
        switch(r.type) {
        case TINT:
            res->fval = l.fval+r.ival;
            break;
        case TFLOAT:
            res->fval = l.fval+r.fval;
            break;
        default:
```

```
            error("bad rhs type +");
        }
        break;
    case TSTRING:
        if(r.type == TSTRING) {
            res->type = TSTRING;
            res->fmt = 's';
            res->string = stradd(l.string, r.string);
            break;
        }
        if(r.type == TINT) {
            res->type = TSTRING;
            res->fmt = 's';
            res->string = straddrune(l.string, r.ival);
            break;
        }
        error("bad rhs for +");
    case TLIST:
        res->type = TLIST;
        switch(r.type) {
        case TLIST:
            res->l = addlist(l.l, r.l);
            break;
        default:
            r.left = 0;
            r.right = 0;
            res->l = addlist(l.l, construct(&r));
            break;
        }
    }
}
```

Uses OCONST 59, TFLOAT 60a, TINT 60a, TLIST 60a, TSTRING 60a, addlist() 393c, construct() 392c, stradd() 386a, and straddrune() 386b.

⟨*function* osub 130⟩≡ (407c)

```
void
osub(Node *n, Node *res)
{
    Node l, r;

    expr(n->left, &l);
    expr(n->right, &r);
    res->fmt = l.fmt;
    res->op = OCONST;
    res->type = TFLOAT;
    switch(l.type) {
    default:
        error("bad lhs type -");
    case TINT:
        switch(r.type) {
        case TINT:
            res->type = TINT;
            res->ival = l.ival-r.ival;
            break;
        case TFLOAT:
            res->fval = l.ival-r.fval;
            break;
        default:
            error("bad rhs type -");
        }
```

```
            break;
        case TFLOAT:
            switch(r.type) {
            case TINT:
                res->fval = l.fval-r.ival;
                break;
            case TFLOAT:
                res->fval = l.fval-r.fval;
                break;
            default:
                error("bad rhs type -");
            }
            break;
        }
    }
```

Uses OCONST 59, TFLOAT 60a, and TINT 60a.

⟨*function* omul 131⟩≡ (407c)

```
    void
    omul(Node *n, Node *res)
    {
        Node l, r;

        expr(n->left, &l);
        expr(n->right, &r);
        res->fmt = l.fmt;
        res->op = OCONST;
        res->type = TFLOAT;
        switch(l.type) {
        default:
            error("bad lhs type *");
        case TINT:
            switch(r.type) {
            case TINT:
                res->type = TINT;
                res->ival = l.ival*r.ival;
                break;
            case TFLOAT:
                res->fval = l.ival*r.fval;
                break;
            default:
                error("bad rhs type *");
            }
            break;
        case TFLOAT:
            switch(r.type) {
            case TINT:
                res->fval = l.fval*r.ival;
                break;
            case TFLOAT:
                res->fval = l.fval*r.fval;
                break;
            default:
                error("bad rhs type *");
            }
            break;
        }
    }
```

Uses OCONST 59, TFLOAT 60a, and TINT 60a.

⟨*function* `odiv` 132a⟩≡ (407c)

```
void
odiv(Node *n, Node *res)
{
    Node l, r;

    expr(n->left, &l);
    expr(n->right, &r);
    res->fmt = l.fmt;
    res->op = OCONST;
    res->type = TFLOAT;
    switch(l.type) {
    default:
        error("bad lhs type /");
    case TINT:
        switch(r.type) {
        case TINT:
            res->type = TINT;
            if(r.ival == 0)
                error("zero divide");
            res->ival = l.ival/r.ival;
            break;
        case TFLOAT:
            if(r.fval == 0)
                error("zero divide");
            res->fval = l.ival/r.fval;
            break;
        default:
            error("bad rhs type /");
        }
        break;
    case TFLOAT:
        switch(r.type) {
        case TINT:
            res->fval = l.fval/r.ival;
            break;
        case TFLOAT:
            res->fval = l.fval/r.fval;
            break;
        default:
            error("bad rhs type /");
        }
        break;
    }
}
```

Uses `OCONST` 59, `TFLOAT` 60a, and `TINT` 60a.

⟨*function* `omod` 132b⟩≡ (407c)

```
void
omod(Node *n, Node *res)
{
    Node l, r;

    expr(n->left, &l);
    expr(n->right, &r);
    res->fmt = l.fmt;
    res->op = OCONST;
    res->type = TINT;
    if(l.type != TINT || r.type != TINT)
        error("bad expr type %");
```

```
    res->ival = l.ival%r.ival;
}
```

Uses OCONST 59 and TINT 60a.

⟨*function* olsh 133a⟩≡ (407c)

```
void
olsh(Node *n, Node *res)
{
    Node l, r;

    expr(n->left, &l);
    expr(n->right, &r);
    res->fmt = l.fmt;
    res->op = OCONST;
    res->type = TINT;
    if(l.type != TINT || r.type != TINT)
        error("bad expr type <<");
    res->ival = l.ival<<r.ival;
}
```

Uses OCONST 59 and TINT 60a.

⟨*function* orsh 133b⟩≡ (407c)

```
void
orsh(Node *n, Node *res)
{
    Node l, r;

    expr(n->left, &l);
    expr(n->right, &r);
    res->fmt = l.fmt;
    res->op = OCONST;
    res->type = TINT;
    if(l.type != TINT || r.type != TINT)
        error("bad expr type >>");
    res->ival = (uvlong)l.ival>>r.ival;
}
```

Uses OCONST 59 and TINT 60a.

10.2.3 List and string operations

The list and string operators (`head`, `tail`, `append`, `delete`, `+` for concatenation, `[]` for indexing) follow the same pattern as the arithmetic operators above: each is a small function that evaluates its operands recursively, type-checks them, builds a new `Value`, and stores it in `res`. The implementations live in `acid/list.c` and the relevant chunks of `Debugger\_extra.nw`; we do not detail them here as they do not introduce new mechanisms.

10.3 Call, locals, and return

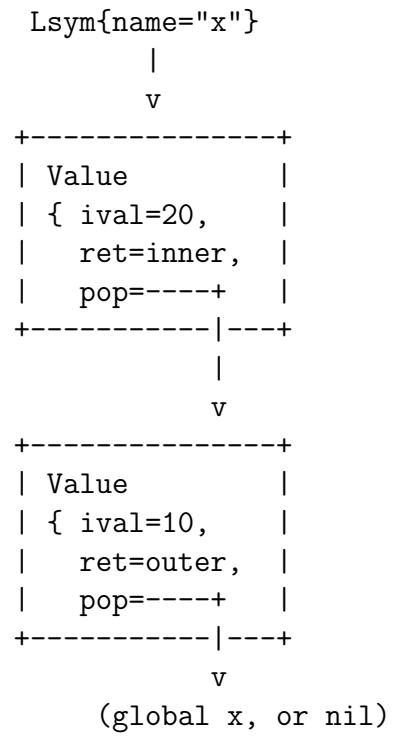
Function calls in `acid` use a scope-based approach for local variables. Each call pushes new `Value` nodes onto the symbol table entries (shadowing the previous values via the `pop` chain), and each return pops them off.

The shadowing trick is worth a diagram because the same `Lsym` entry can carry multiple values at once, one per active scope. Suppose the user defines:

```
defn outer() {
    local x;
    x = 10;
```

```
    inner();
}
defn inner() {
    local x;
    x = 20;
    /* HERE: both scopes are alive */
}
```

At the moment marked HERE, the `Lsym` for `x` looks like:



Each new `local` declaration in a function pushes a fresh `Value` onto the head of the chain via the `pop` field; each return pops the head off and frees it. The `ret` field records which call frame (`Rplace`) owns the binding, so `call()` can recognize "its" bindings during cleanup. This is just a per-symbol stack, but stored inside-out: the chain belongs to the `Lsym` entry rather than to a single global stack of frames.

⟨`Value` *other fields* 134a⟩+≡ (61) ◁62d 134c▷
```
Value* pop;
```

The `Rplace`[378b] structure captures the return context: a `setjmp` buffer for `return` statements (which use `longjmp`), the linked list of local variables to restore on return, and a pointer to the result node. You may wonder why `acid` uses `setjmp`/`longjmp` for `return` when a tree-walking interpreter can usually handle returns just by propagating a flag or a return value back up the recursion chain. The reason is that a `return` can occur *deep* inside nested control structures (loops, ifs, blocks) and using `longjmp` avoids having to check a "returning" flag at every single level of the call stack. With `longjmp`, `return` just jumps directly back to the `setjmp` in `call()` in one step, no matter how deep the nesting is. This is the same trick used by `error()` to unwind on errors. The downside is that local variable cleanup must be done after `setjmp` returns (by walking `ret->local`), since `longjmp` does not run any destructors.

⟨*global* `ret` 134b⟩≡ (381a)
```
Rplace* ret;
```

⟨`Value` *other fields* 134c⟩+≡ (61) ◁134a
```
Rplace* ret;
```

⟨*struct* `Rplace` 135a⟩≡ (378b)
```
struct Rplace
{
    jmp_buf rlab;
    Node* stak;
    Node* val;
    Lsym* local;
    Lsym** tail;
};
```

⟨`execute()` *switch* `n->op` *cases* 135b⟩+≡ (120a) ◁121d 135c▷
```
case ORET:
    if(ret == 0)
        error("return not in function");
    expr(n->left, ret->val);
    longjmp(ret->rlab, 1);
```
Uses `ORET` 59 and `ret` 134b.

⟨`execute()` *switch* `n->op` *cases* 135c⟩+≡ (120a) ◁135b 137▷
```
case OLOCAL:
    for(n = n->left; n; n = n->left) {
        if(ret == 0)
            error("local not in function");
        sl = n->sym;
        if(sl->v->ret == ret)
            error("%s declared twice", sl->name);
        v = gmalloc(sizeof(Value));
        v->ret = ret;

        v->pop = sl->v;
        sl->v = v;

        v->scope = 0;
        *(ret->tail) = sl;
        ret->tail = &v->scope;
        v->set = 0;
    }
    break;
```
Uses `OLOCAL` 59, `gmalloc()` 151b, and `ret` 134b.

`ocall()`[135d] dispatches between builtin C functions and user-defined `acid` functions. If the symbol has a `builtin` pointer and either the call uses the `builtin` keyword or no user-defined `proc` exists, the C function is called directly. Otherwise, `call()`[136b] is invoked for the user-defined function.

⟨*function* `ocall` 135d⟩≡ (407c)
```
void
ocall(Node *n, Node *res)
{
    Lsym *s;
    Rplace *rsav;

    res->op = OCONST;  /* Default return value */
    res->type = TLIST;
    res->l = 0;

    chklval(n->left);
    s = n->left->sym;

    if(n->builtin && !s->builtin){
        error("no builtin %s", s->name);
```

```
        return;
    }
    if(s->builtin && (n->builtin || s->proc == 0)) {
        (*s->builtin)(res, n->right);
        return;
    }
    if(s->proc == 0)
        error("no function %s", s->name);

    rsav = ret;
    call(s->name, n->right, s->proc->left, s->proc->right, res);
    ret = rsav;
}
```

Uses OCONST 59, TLIST 60a, call() 136b, chklval() 401b, and ret 134b.

The na global is a counter used by flatten()[419] to count the number of arguments while it walks the OLIST tree representing a comma-separated argument list. call() resets na to zero, calls flatten() on the actual parameters (filling avp), reads back na as np (the parameter count), then resets na and flattens the formal parameters into ava. The two counts must match—that is the arity check. Using a global counter avoids passing an extra in/out parameter through the recursive walk.

⟨*global* na 136a⟩≡ (381a)

```
int na;
```

call() implements the full function call protocol: flatten parameter and argument lists, check arity, push new Value nodes for each parameter (shadowing any existing bindings), execute the body, then pop all locals and restore the previous bindings. The setjmp captures the return point so that return can longjmp back.

⟨*function* call 136b⟩≡ (391)

```
void
call(char *fn, Node *parameters, Node *local, Node *body, Node *retexp)
{
    int np, i;
    Rplace rlab;
    Node *n, res;
    Value *v, *f;
    Lsym *s, *next;
    Node *avp[Maxarg], *ava[Maxarg];

    rlab.local = 0;

    na = 0;
    flatten(avp, parameters);
    np = na;
    na = 0;
    flatten(ava, local);
    if(np != na) {
        if(np < na)
            error("%s: too few arguments", fn);
        error("%s: too many arguments", fn);
    }

    rlab.tail = &rlab.local;

    ret = &rlab;
    for(i = 0; i < np; i++) {
        n = ava[i];
        switch(n->op) {
        default:
            error("%s: %d formal not a name", fn, i);
```

```
            case ONAME:
                expr(avp[i], &res);
                s = n->sym;
                break;
            case OINDM:
                res.cc = avp[i];
                res.type = TCODE;
                res.comt = 0;
                if(n->left->op != ONAME)
                    error("%s: %d formal not a name", fn, i);
                s = n->left->sym;
                break;
            }
            if(s->v->ret == ret)
                error("%s already declared at this scope", s->name);

            v = gmalloc(sizeof(Value));
            v->ret = ret;
            v->pop = s->v;
            s->v = v;
            v->scope = 0;
            *(rlab.tail) = s;
            rlab.tail = &v->scope;

            v->Store = res.Store;
            v->type = res.type;
            v->set = 1;
        }

        ret->val = retexp;
        if(setjmp(rlab.rlab) == 0)
            execute(body);

        for(s = rlab.local; s; s = next) {
            f = s->v;
            next = f->scope;
            s->v = f->pop;
            free(f);
        }
    }
```

Uses Maxarg 82a, OINDM 59, ONAME 59, TCODE 60a, execute() 120a, flatten() 419, gmalloc() 151b, na 136a, and ret 134b.

10.4 Statement handlers

The four control-flow opcodes (OLIST sequence, OIF, OWHILE, ODO counted loop) are handled directly inside execute()[120a]'s opcode switch. Each is a small recursive call back into execute() for the body, with expr() used to evaluate the condition. Note that loop bounds in ODO (loop start, end do body) must be integers—this is the only place acid enforces a type at statement level.

⟨execute() *switch* n->op *cases* 137⟩+≡ (120a) ◁135c 138a▷

```
  case OLIST:
      execute(n->left);
      execute(n->right);
      break;
```

Uses OLIST 59 and execute() 120a.

⟨execute() *switch* n->op *cases* 138a⟩+≡ (120a) ◁137 138b▷

```
case OIF:
    expr(l, &res);
    if(r && r->op == OELSE) {
        if(fbool(&res))
            execute(r->left);
        else
            execute(r->right);
    }
    else if(fbool(&res))
        execute(r);
    break;
```

Uses OELSE 59, OIF 59, execute() 120a, and fbool() 123a.

⟨execute() *switch* n->op *cases* 138b⟩+≡ (120a) ◁138a 138c▷

```
case OWHILE:
    for(;;) {
        expr(l, &res);
        if(!fbool(&res))
            break;
        execute(r);
    }
    break;
```

Uses OWHILE 59, execute() 120a, and fbool() 123a.

⟨execute() *switch* n->op *cases* 138c⟩+≡ (120a) ◁138b 139a▷

```
case ODO:
    expr(l->left, &res);
    if(res.type != TINT)
        error("loop must have integer start");
    s = res.ival;
    expr(l->right, &res);
    if(res.type != TINT)
        error("loop must have integer end");
    e = res.ival;
    for(i = s; i <= e; i++)
        execute(r);
    break;
```

Uses ODO 59, TINT 60a, and execute() 120a.

10.5 Complex

The `complex` (or `aggr`) system lets `acid` understand C data structures. A complex type declaration (typically generated by `cc -a`) describes the offset, format, and name of each field in a struct. When a variable is declared as a complex type, the `.` operator can access its fields by reading memory at the appropriate offset from the base address.

For example, the C struct

```
struct Point {
    int x;
    int y;
    char *label;
};
```

generates the following `acid` complex declaration when compiled with `8c -a`:

```
aggr Point {
    'D' 0  x;
    'D' 4  y;
    'X' 8  label;
};
```

Each line specifies the field's print format (`'D'` decimal, `'X'` hex), its byte offset from the base address, and its name. With this declaration loaded, the user can do `complex Point p; p = 0x1234;` and then access `p.x`, `p.y`, `p.label` through the `.` operator, which the interpreter translates into reads at the appropriate offsets via `cormap`.

⟨`execute()` *switch* `n->op` *cases* 139a⟩+≡ (120a) ◁138c
```
case OCOMPLEX:
    decl(n);
    break;
```
Uses `OCOMPLEX` 59 and `decl()` 139d.

`Frtype` is a small struct that records "variable `var` in this function should be displayed as complex type `type`." When the user writes `complex Foo main:p` (declaring that local variable `p` in function `main` is of complex type `Foo`), `decl()`[139d] adds an `Frtype` to the function's `local` list rather than tagging the `Lsym` for `p` directly. The function-scoped list is needed because the same local-variable name (e.g., `i`) can exist in different functions with different complex types.

⟨*struct* `Frtype` 139b⟩≡ (378b)
```
struct Frtype
{
    Lsym* var;
    Type* type;
    Frtype* next;
};
```

⟨`Lsym` *other fields* 139c⟩+≡ (65a) ◁68b
```
Frtype* local;
```

⟨*function* `decl` 139d⟩≡ (400b)
```
void
decl(Node *n)
{
    Node *l;
    Value *v;
    Frtype *f;
    Lsym *type;

    type = n->sym;
    if(type->lt == 0)
        error("%s is not a complex type", type->name);

    l = n->left;
    if(l->op == ONAME) {
        v = l->sym->v;
        v->comt = type->lt;
        v->fmt = 'a';
        return;
    }

    /*
     * Frame declaration
     */
    for(f = l->sym->local; f; f = f->next) {
```

```
            if(f->var == l->left->sym) {
                f->type = n->sym->lt;
                return;
            }
        }
        f = malloc(sizeof(Frtype));
        if(f == 0)
            fatal("out of memory");

        f->type = type->lt;

        f->var = l->left->sym;
        f->next = l->sym->local;
        l->sym->local = f;
    }
```

Uses ONAME 59.

10.6 Builtins

The builtin functions provide the bridge between `acid`'s interpreted language and the underlying system. They are grouped by category:

Category	Builtins
type conversion	`cvtatof()`[416a], `cvtatoi()`[416b], `cvtitoa()`[417]
file operations	`getfile()`[415], `readfile()`[414c], `doaccess()`[414b]
symbol lookup	`filepc()`[412b], `funcbound()`[411b], `pcfile()`[422a], `pcline()`[422b]
process control	`start()`[162c], `stop()`[162d], `startstop()`[163a], `waitstop()`[163b]
process create	`newproc()`[81c], `setproc()`[412a], `kill`[410a]
output	`bprint()`[144d], `printto()`[421b], `fmt`[147b]
formatting	`fmtof()`[423a], `dofmtsize()`[148b]
matching	`match`[409c], `regexp`[421a]
miscellaneous	`doerror()`[414a], `dosysr1()`[409b], `strace`[420a], `map`[418b], `rc`[413a], `reason`[410b]

Each builtin takes a result node and an argument node, evaluates its arguments, performs the operation, and stores the result back in the result node.

⟨tab *entries* 140a⟩≡ (68a) 140b▷

```
"atof", cvtatof,
"atoi", cvtatoi,
"itoa", cvtitoa,
```

Uses `cvtatof()` 416a, `cvtatoi()` 416b, and `cvtitoa()` 417.

⟨tab *entries* 140b⟩+≡ (68a) ◁140a 140c▷

```
"file",         getfile,
"readfile", readfile,
"access", doaccess,
```

Uses `doaccess()` 414b and `readfile()` 414c.

⟨tab *entries* 140c⟩+≡ (68a) ◁140b 141a▷

```
"filepc", filepc,
"fnbound", funcbound,
"pcfile", pcfile,
"pcline", pcline,
```

Uses `filepc()` 412b, `funcbound()` 411b, `pcfile()` 422a, and `pcline()` 422b.

⟨tab *entries* 141a⟩+≡ (68a) ◁140c 141b▷
```
"setproc", setproc,
"kill", kill,
```
Uses `kill()` 410a and `setproc()` 412a.

⟨tab *entries* 141b⟩+≡ (68a) ◁141a 141c▷
```
"printto", printto,
"fmtof", fmtof,
```
Uses `fmtof()` 423a and `printto()` 421b.

⟨tab *entries* 141c⟩+≡ (68a) ◁141b 141d▷
```
"match", match,
"regexp", regexp,
```
Uses `match()` 409c and `regexp()` 421a.

⟨tab *entries* 141d⟩+≡ (68a) ◁141c 141e▷
```
"follow", follow,
```
Uses `follow()` 411a.

⟨tab *entries* 141e⟩+≡ (68a) ◁141d 141f▷
```
"error", doerror,
```
Uses `doerror()` 414a.

⟨tab *entries* 141f⟩+≡ (68a) ◁141e 141g▷
```
"rc", rc,
```
Uses `rc()` 413a.

⟨tab *entries* 141g⟩+≡ (68a) ◁141f 141h▷
```
"reason", reason,
```
Uses `reason()` 410b.

⟨tab *entries* 141h⟩+≡ (68a) ◁141g 141i▷
```
"map", map,
```
Uses `map()` 418b.

⟨tab *entries* 141i⟩+≡ (68a) ◁141h 141j▷
```
"strace", strace,
"sysr1", dosysr1,
```
Uses `dosysr1()` 409b and `strace()` 420a.

10.6.1 Including a file: `include()`

We will not look at the implementation of every builtin. Most of them follow the same boilerplate: check argument count, evaluate arguments via `expr()`, call a small helper, and store the result. Instead, we focus on a few that illustrate non-trivial mechanisms: `include()` (loading another `acid` file at runtime), `interpret()` (evaluating a string as code, also known as "eval"), and a few others as we encounter them in later chapters.

⟨tab *entries* 141j⟩+≡ (68a) ◁141i 142b▷
```
"include", include,
```
Uses `include()` 142a.

⟨*function* `include` 142a⟩≡ (423b)

```
void
include(Node *r, Node *args)
{
    Node res;
    bool isave;

    if(args == 0)
        error("include(string): arg count");
    expr(args, &res);
    if(res.type != TSTRING)
        error("include(string): arg type");

    pushfile(res.string->string);

    isave = interactive;
    interactive = false;
    r->ival = yyparse();
    interactive = isave;
    popio();
    r->op = OCONST;
    r->type = TINT;
    r->fmt = 'D';
}
```

Uses `OCONST` 59, `TINT` 60a, `TSTRING` 60a, `interactive` 70a, `popio()` 101e, `pushfile()` 100d, and `yyparse()`.

10.6.2 Eval: `interpret()`

The `interpret()` builtin takes a string and evaluates it as `acid` code. It pushes the string as an input source (via `pushstr()`[143b], which sets `lexio->text` instead of `lexio->fin`), calls `yyparse()X` to parse and execute it, then pops the input source. This enables dynamic code generation: an `acid` function can build a string containing `acid` code and execute it at runtime. A typical use case in the standard library is the `asm()` function in `/lib/acid/port.acid`, which formats the result of `das()X` as a string and uses `interpret()` to display it. More generally, `interpret` is what enables "eval-style" debugging recipes: write a function that decides what to inspect based on runtime data, then build the right `print` expression on the fly.

⟨`tab` *entries* 142b⟩+≡ (68a) ◁141j 144c▷

```
"interpret",interpret,
```

Uses `interpret()` 142c.

⟨*function* `interpret` 142c⟩≡ (423b)

```
/// acid: interpret() -> <>
void
interpret(Node *r, Node *args)
{
    Node res;
    bool isave;

    if(args == nil)
        error("interpret(string): arg count");
    expr(args, &res);
    if(res.type != TSTRING)
        error("interpret(string): arg type");

    pushstr(&res);

    isave = interactive;
    interactive = false;
```

```
        r->ival = yyparse();
        interactive = isave;
        popio();
        r->op = OCONST;
        r->type = TINT;
        r->fmt = 'D';
    }
```

Uses OCONST 59, TINT 60a, TSTRING 60a, interactive 70a, popio() 101e, pushstr() 143b, and yyparse().

⟨IOstack *other fields* 143a⟩≡ (100a)

```
    char *text;
    char *ip;
```

⟨*function* pushstr 143b⟩≡ (381b)

```
    /// acid: interpret() -> interpret -> <>
    void
    pushstr(Node *s)
    {
        IOstack *io;

        io = malloc(sizeof(IOstack));
        ⟨pushstr() sanity check io 143c⟩
        io->line = line;
        line = 1;
        io->name = strdup("<string>");
        ⟨pushstr() sanity check io->name 143d⟩
        io->line = line;
        line = 1;

        // this time we do not use io->fin but io->text
        io->fin = nil;
        io->text = strdup(s->string->string);
        ⟨pushstr() sanity check io->text 143e⟩
        io->ip = io->text;

        io->prev = lexio;
        lexio = io;
    }
```

Uses lexio 100b.

⟨pushstr() *sanity check* io 143c⟩≡ (143b)

```
    if(io == nil)
        fatal("no memory");
```

⟨pushstr() *sanity check* io->name 143d⟩≡ (143b)

```
    if(io->name == nil)
        fatal("no memory");
```

⟨pushstr() *sanity check* io->text 143e⟩≡ (143b)

```
    if(io->text == 0)
        fatal("no memory");
```

⟨popio() *when no* fin 143f⟩≡ (101e)

```
    free(lexio->text);
```

Uses lexio 100b.

⟨lexc() *when no* fin 143g⟩≡ (102a)

```
    c = *lexio->ip++;
    if(c == 0)
        return -1;
    return c;
```

Uses lexio 100b.

⟨unlexc() *when no* fin 144a⟩≡ (102b)
```
lexio->ip--;
```
Uses lexio 100b.

10.6.3 Printing: print()

This section covers acid's output: the print builtin, the prnt node used to auto-print expression results in the REPL.

⟨*global* prnt 144b⟩≡ (381a)
```
Node* prnt;
```

⟨tab *entries* 144c⟩+≡ (68a) ◁142b 147a▷
```
"print", bprint,
```

⟨*function* bprint 144d⟩≡ (423b)
```
void
bprint(Node *r, Node *args)
{
    int i, nas;
    Node res, *av[Maxarg];

    USED(r);
    na = 0;
    flatten(av, args);
    nas = na;
    for(i = 0; i < nas; i++) {
        expr(av[i], &res);
        switch(res.type) {
        default:
            if(comx(res))
                break;
            patom(res.type, &res.Store);
            break;
        case TCODE:
            pcode(res.cc, 0);
            break;
        case TLIST:
            blprint(res.l);
            break;
        }
    }
    if(ret == 0)
        Bputc(bout, '\n');
}
```
Uses Maxarg 82a, TCODE 60a, TLIST 60a, blprint() 144e, bout 69d, comx() 145a, flatten() 419, na 136a, patom() 145b, pcode() 198b, and ret 134b.

⟨*function* blprint 144e⟩≡ (423b)
```
void
blprint(List *l)
{
    Bprint(bout, "{");
    while(l) {
        switch(l->type) {
        default:
            patom(l->type, &l->Store);
            break;
        case TSTRING:
```

```
            Bputc(bout, '"');
            patom(l->type, &l->Store);
            Bputc(bout, '"');
            break;
        case TLIST:
            blprint(l->l);
            break;
        case TCODE:
            pcode(l->cc, 0);
            break;
        }
        l = l->next;
        if(l)
            Bprint(bout, ", ");
    }
    Bprint(bout, "}");
}
```

Uses TCODE 60a, TLIST 60a, TSTRING 60a, blprint() 144e, bout 69d, patom() 145b, and pcode() 198b.

⟨*function* comx 145a⟩≡ (423b)

```
int
comx(Node res)
{
    Lsym *sl;
    Node *n, xx;

    if(res.fmt != 'a' && res.fmt != 'A')
        return 0;

    if(res.comt == 0 || res.comt->base == 0)
        return 0;

    sl = res.comt->base;
    if(sl->proc) {
        res.left = ZN;
        res.right = ZN;
        n = an(ONAME, ZN, ZN);
        n->sym = sl;
        n = an(OCALL, n, &res);
            n->left->sym = sl;
        expr(n, &xx);
        return 1;
    }
    print("(%s)", sl->name);
    return 0;
}
```

Uses OCALL 59, ONAME 59, ZN 58b, and an() 58c.

⟨*function* patom 145b⟩≡ (423b)

```
/// bprint | printto -> <>
void
patom(char type, Store *res)
{
    int i;
    char buf[512];
    extern char *typenames[];

    switch(res->fmt) {
    case 'c':
        Bprint(bout, "%c", (int)res->ival);
```

```
    break;
case 'C':
    if(res->ival < ' ' || res->ival >= 0x7f)
        Bprint(bout, "%3d", (int)res->ival&0xff);
    else
        Bprint(bout, "%3c", (int)res->ival);
    break;
case 'r':
    Bprint(bout, "%C", (int)res->ival);
    break;
case 'B':
    memset(buf, '0', 34);
    buf[1] = 'b';
    for(i = 0; i < 32; i++) {
        if(res->ival & (1<<i))
            buf[33-i] = '1';
    }
    buf[35] = '\0';
    Bprint(bout, "%s", buf);
    break;
case 'b':
    Bprint(bout, "%.2x", (int)res->ival&0xff);
    break;
case 'X':
    Bprint(bout, "%.8lux", (ulong)res->ival);
    break;
case 'x':
    Bprint(bout, "%.4lux", (ulong)res->ival&0xffff);
    break;
case 'D':
    Bprint(bout, "%d", (int)res->ival);
    break;
case 'd':
    Bprint(bout, "%d", (ushort)res->ival);
    break;
case 'u':
    Bprint(bout, "%d", (int)res->ival&0xffff);
    break;
case 'U':
    Bprint(bout, "%lud", (ulong)res->ival);
    break;
case 'Z':
    Bprint(bout, "%llud", res->ival);
    break;
case 'V':
    Bprint(bout, "%lld", res->ival);
    break;
case 'W':
    Bprint(bout, "%.8llux", res->ival);
    break;
case 'Y':
    Bprint(bout, "%.16llux", res->ival);
    break;
case 'o':
    Bprint(bout, "0%.11uo", (int)res->ival&0xffff);
    break;
case 'O':
    Bprint(bout, "0%.6uo", (int)res->ival);
    break;
case 'q':
```

```
            Bprint(bout, "0%.11o", (short)(res->ival&0xffff));
            break;
        case 'Q':
            Bprint(bout, "0%.6o", (int)res->ival);
            break;
        case 'f':
        case 'F':
        case '3':
        case '8':
            if(type != TFLOAT)
                Bprint(bout, "*%c<%s>*", res->fmt, typenames[type]);
            else
                Bprint(bout, "%g", res->fval);
            break;
        case 's':
        case 'g':
        case 'G':
            if(type != TSTRING)
                Bprint(bout, "*%c<%s>*", res->fmt, typenames[type]);
            else
                Bwrite(bout, res->string->string, res->string->len);
            break;
        case 'R':
            if(type != TSTRING)
                Bprint(bout, "*%c<%s>*", res->fmt, typenames[type]);
            else
                Bprint(bout, "%S", (Rune*)res->string->string);
            break;
        case 'a':
        case 'A':
            symoff(buf, sizeof(buf), res->ival, CANY);
            Bprint(bout, "%s", buf);
            break;
        case 'I':
        case 'i':
            if(type != TINT)
                Bprint(bout, "*%c<%s>*", res->fmt, typenames[type]);
            else {
                if (symmap == nil || (*machdata->das)(symmap, res->ival, res->fmt, buf, sizeof(buf)) < 0)
                    Bprint(bout, "no instruction");
                else
                    Bprint(bout, "%s", buf);
            }
            break;
        }
    }
```

Uses TFLOAT 60a, TINT 60a, TSTRING 60a, bout 69d, machdata 355e, symoff() 356a, and typenames 203b.

10.6.4 Formatting: fmt()

⟨tab *entries* 147a⟩+≡ (68a) ◁144c 148a▷

```
    "fmt", fmt,
```

Uses fmt() 147b.

⟨*function* fmt 147b⟩≡ (423b)

```
    void
    fmt(Node *r, Node *args)
    {
```

```
    Node res;
    Node *av[Maxarg];

    na = 0;
    flatten(av, args);
    if(na != 2)
        error("fmt(obj, fmt): arg count");
    expr(av[1], &res);
    if(res.type != TINT || strchr(vfmt, res.ival) == 0)
        error("fmt(obj, fmt): bad format '%c'", (char)res.ival);
    expr(av[0], r);

    r->fmt = res.ival;
}
```

Uses Maxarg 82a, TINT 60a, flatten() 419, na 136a, and vfmt 110c.

10.6.5 Instruction size: fmtsize()

⟨tab *entries* 148a⟩+≡ (68a) ◁147a 161c▷

```
"fmtsize", dofmtsize,
```

Uses dofmtsize() 148b.

⟨*function* dofmtsize 148b⟩≡ (423b)

```
void dofmtsize(Node *r, Node *args)
{
    Node *av[Maxarg];
    Node res;
    Store * s ;
    Value v ;

    na = 0;
    flatten(av, args);
    if(na < 1)
        error("fmtsize(obj): no argument");
    if(na > 1)
        error("fmtsize(obj): too many arguments") ;
    expr(av[0], &res);

    v.type = res.type ;
    s = &v.Store ;
    *s = res ;

    r->op = OCONST;
    r->type = TINT ;
    r->ival = fmtsize(&v) ;
    r->fmt = 'D';
}
```

Uses Maxarg 82a, OCONST 59, TINT 60a, flatten() 419, fmtsize() 148c, and na 136a.

⟨*function* fmtsize 148c⟩≡ (407c)

```
int
fmtsize(Value *v)
{
    int ret;

    switch(v->fmt) {
    default:
        return  fsize[v->fmt];
```

```
    case 'i':
    case 'I':
        if(v->type != TINT || machdata == 0)
            error("no size for i fmt pointer ++/--");
        ret = (*machdata->instsize)(cormap, v->ival);
        if(ret < 0) {
            ret = (*machdata->instsize)(symmap, v->ival);
            if(ret < 0)
                error("%r");
        }
        return ret;
    }
}
```

Uses TINT 60a, fsize-17 401a, and machdata 355e.

10.7 Garbage collection

The garbage collector runs periodically (when dogc[151a] exceeds 1MB of allocated memory). It uses a three-phase mark-and-sweep: (1) clear all marks, (2) scan the symbol table and mark all reachable objects (strings, lists, nodes, and code values), (3) sweep the gcl list and free unmarked objects. The scan phase traverses both the procedure bodies (marktree) and the value chain (v->pop for shadowed locals).

10.7.1 Mark-and-sweep: gc()

⟨*function* gc 149⟩≡ (383c)

```
void
gc(void)
{
    int i;
    Lsym *f;
    Value *v;
    Gc *m, **p, *next;

    ⟨gc() return if still enough memory 151d⟩

    /* Mark */
    for(m = gcl; m; m = m->gclink)
        m->gcmark = 0;

    /* Scan */
    for(i = 0; i < Hashsize; i++) {
        for(f = hash[i]; f; f = f->hash) {
            marktree(f->proc);
            if(f->lexval != Tid)
                continue;
            for(v = f->v; v; v = v->pop) {
                switch(v->type) {
                case TSTRING:
                    v->string->gcmark = 1;
                    break;
                case TLIST:
                    marklist(v->l);
                    break;

                case TCODE:
                    marktree(v->cc);
```

```
                    break;
                }
            }
        }
    }

    /* Free */
    p = &gcl;
    for(m = gcl; m; m = next) {
        next = m->gclink;
        if(m->gcmark == 0) {
            *p = next;
            free(m); /* Sleazy reliance on my malloc */
        }
        else
            p = &m->gclink;
    }
}
```

Uses Hashsize 66b, TCODE 60a, TLIST 60a, TSTRING 60a, Tid, gcl 64b, marklist() 150b, and marktree() 150a.

10.7.2 Marking phase: marktree() and marklist()

⟨*function* marktree 150a⟩≡ (383c)

```
void
marktree(Node *n)
{

    if(n == nil)
        return;

    marktree(n->left);
    marktree(n->right);

    n->gcmark = 1;
    if(n->op != OCONST)
        return;

    switch(n->type) {
    case TSTRING:
        n->string->gcmark = 1;
        break;
    case TLIST:
        marklist(n->l);
        break;

    case TCODE:
        marktree(n->cc);
        break;
    }
}
```

Uses OCONST 59, TCODE 60a, TLIST 60a, TSTRING 60a, marklist() 150b, and marktree() 150a.

⟨*function* marklist 150b⟩≡ (383c)

```
void
marklist(List *l)
{
    while(l) {
        l->gcmark = 1;
```

```
        switch(l->type) {
        case TSTRING:
            l->string->gcmark = 1;
            break;
        case TLIST:
            marklist(l->l);
            break;
        case TCODE:
            marktree(l->cc);
            break;
        }
        l = l->next;
    }
}
```

Uses TCODE 60a, TLIST 60a, TSTRING 60a, marklist() 150b, and marktree() 150a.

10.7.3 Allocation: gmalloc() and dogc

gmalloc()[151b] is a thin wrapper around malloc that tracks the total amount of memory allocated since the last GC in dogc. When dogc exceeds Mempergc (1MB), the next call to gc()[149] will actually run a collection; otherwise gc() returns immediately. This amortizes the cost of garbage collection over many allocations.

⟨*global* dogc 151a⟩≡ (381a)
```
long dogc;
```

⟨*function* gmalloc 151b⟩≡ (383c)
```
void*
gmalloc(long l)
{
    void *p;

    dogc += l;
    p = malloc(l);
    if(p == nil)
        fatal("out of memory");
    return p;
}
```

Uses dogc 151a.

⟨*acid constants* 151c⟩+≡ (378a) ◁108a
```
Mempergc = 1024*1024,
```

⟨gc() *return if still enough memory* 151d⟩≡ (149)
```
if(dogc < Mempergc)
    return;
dogc = 0;
```

Uses Mempergc 151c and dogc 151a.

Chapter 11

The acid library

Most of acid's user-facing functionality is implemented not in C but in acid's own language, in library files loaded at startup. This is the essence of acid's design: the C code provides low-level primitives (process control, memory access, expression evaluation), and the acid library builds higher-level debugging commands on top. Users can redefine any library function or write new ones.

11.1 The portable library: /lib/acid/port.acid

The portable library defines the core debugging functions: new() (start a process, set a breakpoint at main, continue to it), bpset()/bpdel() (breakpoint management), cont() (continue execution), step() (single-step), stk() (print call stack), src() (show source), asm() (disassemble), regs() (print registers), mem() (display memory), and stkrace() (stack trace). These functions use the builtins from the Interpreting and Process Control chapters.

⟨*lib/acid/port.acid* 152⟩≡

```
// portable acid for all architectures

defn pfl(addr)
{
        print(pcfile(addr), ":", pcline(addr), "\n");
}

defn
notestk(addr)
{
        local pc, sp;
        complex Ureg addr;

        pc = addr.pc\X;
        sp = addr.sp\X;

        print("Note pc:", pc, " sp:", sp, " ", fmt(pc, 'a'), " ");
        pfl(pc);
        _stk(pc, sp, linkreg(addr), 1);
}

defn
notelstk(addr)
{
        local pc, sp;
        complex Ureg addr;

        pc = addr.pc\X;
```

```
        sp = addr.sp\X;

        print("Note pc:", pc, " sp:", sp, " ", fmt(pc, 'a'), " ");
        pfl(pc);
        _stk(pc, sp, linkreg(addr), 1);
}

defn labstk(l)                          // trace from a label
{
        _stk(*(l+4), *l, linkreg(0), 0);
}
```

⟨*function* params 170c⟩

```
stkprefix = "";
stkignore = {};
stkend = 0;
```

⟨*function* locals 170b⟩

```
defn _stkign(file)
{
        s = stkignore;
        while s do {
                if regexp(head s, file) then
                        return 1;
                s = tail s;
        }
        return 0;
}
```

⟨*function* _stk 168c⟩

```
defn findsrc(file)
{
        local lst, src;

        if file[0] == '/' then {
                src = file(file);
                if src != {} then {
                        srcfiles = append srcfiles, file;
                        srctext = append srctext, src;
                        return src;
                }
                return {};
        }

        lst = srcpath;
        while head lst do {
                src = file(head lst+file);
                if src != {} then {
                        srcfiles = append srcfiles, file;
                        srctext = append srctext, src;
                        return src;
                }
                lst = tail lst;
        }
}

defn line(addr)
```

```
{
        local src, file;

        file = pcfile(addr);
        src = match(file, srcfiles);

        if src >= 0 then
                src = srctext[src];
        else
                src = findsrc(file);

        if src == {} then {
                print("no source for ", file, "\n");
                return {};
        }
        line = pcline(addr)-1;
        print(file, ":", src[line], "\n");
}

defn addsrcdir(dir)
{
        dir = dir+"/";

        if match(dir, srcpath) >= 0 then {
                print("already in srcpath\n");
                return {};
        }

        srcpath = {dir}+srcpath;
}

defn source()
{
        local l;

        l = srcpath;
        while l do {
                print(head l, "\n");
                l = tail l;
        }
        l = srcfiles;

        while l do {
                print("\t", head l, "\n");
                l = tail l;
        }
}

defn Bsrc(addr)
{
        local lst;

        lst = srcpath;
        file = pcfile(addr);
        if file[0] == '/' && access(file) then {
                rc("B "+file+":"+itoa(pcline(addr)));
                return {};
        }
        while head lst do {
                name = head lst+file;
```

```
                if access(name) then {
                        rc("B "+name+":"+itoa(pcline(addr)));
                        return {};
                }
                lst = tail lst;
        }
        print("no source for ", file, "\n");
}

defn srcline(addr)
{
        local text, cline, line, file, src;
        file = pcfile(addr);
        src = match(file,srcfiles);
        if (src>=0) then
                src = srctext[src];
        else
                src = findsrc(file);
        if (src=={}) then
        {
                return "(no source)";
        }
        return src[pcline(addr)-1];
}
```

⟨*function* src 172⟩

⟨*function* step 167⟩

⟨*function* bpset 165a⟩

```
defn bptab()                                    // print a table of breakpoints
{
        local lst, addr;

        lst = bplist;
        while lst do {
                addr = head lst;
                print("\t", fmt(addr, 'X'), " ", fmt(addr, 'a'), "  ", fmt(addr, 'i'), "\n");
                lst = tail lst;
        }
}
```

⟨*function* bpdel 165b⟩

⟨*function* cont 166⟩

```
defn stopped(pid)               // called from acid when a process changes state
{
        pstop(pid);             // stub so this is easy to replace
}

defn procs()                    // print status of processes
{
        local c, lst, cpid;

        cpid = pid;
        lst = proclist;
        while lst do {
                np = head lst;
```

```
                setproc(np);
                if np == cpid then
                        c = '>';
                else
                        c = ' ';
                print(fmt(c, 'c'), np, ": ", status(np), " at ", fmt(*PC, 'a'), " setproc(", np, ")\n");
                lst = tail lst;
        }
        pid = cpid;
        if pid != 0 then
                setproc(pid);
}
```

⟨*constant* _asmlines 170a⟩

⟨*function* asm 169c⟩

```
defn casm()
{
        asm(lasmaddr);
}

defn win()
{
        local npid, estr;

        bplist = {};
        notes = {};

        estr = "/sys/lib/acid/window '0 0 600 400' "+textfile;
        if progargs != "" then
                estr = estr+" "+progargs;

        npid = rc(estr);
        npid = atoi(npid);
        if npid == 0 then
                error("win failed to create process");

        setproc(npid);
        stopped(npid);
}

defn win2()
{
        local npid, estr;

        bplist = {};
        notes = {};

        estr = "/sys/lib/acid/transcript '0 0 600 400' '100 100 700 500' "+textfile;
        if progargs != "" then
                estr = estr+" "+progargs;

        npid = rc(estr);
        npid = atoi(npid);
        if npid == 0 then
                error("win failed to create process");

        setproc(npid);
        stopped(npid);
```

```
}
```

⟨*function* new 161a⟩

```
defn stmnt()                    // step one statement
{
        local line;

        line = pcline(*PC);
        while 1 do {
                step();
                if line != pcline(*PC) then {
                        src(*PC);
                        return {};
                }
        }
}

defn func()                     // step until we leave the current function
{
        local bound, end, start, pc;

        bound = fnbound(*PC);
        if bound == {} then {
                print("cannot locate text symbol\n");
                return {};
        }

        pc = *PC;
        start = bound[0];
        end = bound[1];
        while pc >= start && pc < end do {
                step();
                pc = *PC;
        }
}
```

⟨*function* next 174⟩

```
defn dump(addr, n, fmt)
{
        // see definition of dump in acid manual: it does n+1 iterations
        loop 0, n do {
                print(fmt(addr, 'X'), ": ");
                addr = mem(addr, fmt);
        }
}

defn mem(addr, fmt)
{

        local i, c, n;

        i = 0;
        while fmt[i] != 0 do {
                c = fmt[i];
                n = 0;
                while '0' <= fmt[i] && fmt[i] <= '9' do {
                        n = 10*n + fmt[i]-'0';
                        i = i+1;
```

```
                }
                if n <= 0 then n = 1;
                addr = fmt(addr, fmt[i]);
                while n > 0 do {
                        print(*addr++, " ");
                        n = n-1;
                }
                i = i+1;
        }
        print("\n");
        return addr;
}

⟨function symbols 170d⟩

defn spsrch(len)
{
        local addr, a, s, e;

        addr = *SP;
        s = origin & 0x7fffffff;
        e = etext & 0x7fffffff;
        loop 1, len do {
                a = *addr++;
                c = a & 0x7fffffff;
                if c > s && c < e then {
                        print("src(", a, ")\n");
                        pfl(a);
                }
        }
}

⟨global progargs 161b⟩

print("/lib/acid/port.acid");
```

11.2 The ARM library: `/lib/acid/arm.acid`

The architecture-specific library files define functions that depend on register names and calling conventions: `regs()` prints the architecture's register set, `stk()` uses the frame pointer chain to walk the stack, and `step()` uses `follow()` to find the next instruction(s) and single-step correctly over branches and delay slots.

⟨*lib/acid/arm.acid* 158⟩≡

```
// ARM support

⟨function acidinit(arm) 79c⟩

⟨function linkreg(arm) 168b⟩
⟨function stk(arm) 168a⟩

defn lstk()                             // trace with locals
{
        _stk(*PC, *SP, linkreg(0), 1);
}

⟨function gpr(arm) 169b⟩
⟨function regs(arm) 169a⟩
```

```
defn pstop(pid)
{
        local l;
        local pc;

        pc = *PC;

        print(pid,": ", reason(*TYPE), "\t");
        print(fmt(pc, 'a'), "\t", fmt(pc, 'i'), "\n");

        if notes then {
                if notes[0] != "sys: breakpoint" then {
                        print("Notes pending:\n");
                        l = notes;
                        while l do {
                                print("\t", head l, "\n");
                                l = tail l;
                        }
                }
        }
}

sizeofUreg=72;
aggr Ureg
{
        'U' 0 r0;
        'U' 4 r1;
        'U' 8 r2;
        'U' 12 r3;
        'U' 16 r4;
        'U' 20 r5;
        'U' 24 r6;
        'U' 28 r7;
        'U' 32 r8;
        'U' 36 r9;
        'U' 40 r10;
        'U' 44 r11;
        'U' 48 r12;
        'U' 52 r13;
        'U' 56 r14;
        'U' 60 type;
        'U' 64 psr;
        'U' 68 pc;
};

defn
Ureg(addr) {
        complex Ureg addr;
        print(" r0      ", addr.r0, "\n");
        print(" r1      ", addr.r1, "\n");
        print(" r2      ", addr.r2, "\n");
        print(" r3      ", addr.r3, "\n");
        print(" r4      ", addr.r4, "\n");
        print(" r5      ", addr.r5, "\n");
        print(" r6      ", addr.r6, "\n");
        print(" r7      ", addr.r7, "\n");
        print(" r8      ", addr.r8, "\n");
        print(" r9      ", addr.r9, "\n");
        print(" r10     ", addr.r10, "\n");
        print(" r11     ", addr.r11, "\n");
```

```
        print(" r12     ", addr.r12, "\n");
        print(" r13     ", addr.r13, "\n");
        print(" r14     ", addr.r14, "\n");
        print(" type    ", addr.type, "\n");
        print(" psr     ", addr.psr, "\n");
        print(" pc      ", addr.pc, "\n");
};

print("/lib/acid/arm.acid");
```

Chapter 12

The acid Commands

We are now finally in a position to see and understand the code behind most of the commands we used in the hello_bug debugging session in Section 2.5: new(), bpset(), cont(), stk(), src(), asm(), regs(), and so on. Most are implemented as combinations of acid library functions (in /lib/acid/port.acid and /lib/acid/$objtype.acid) calling C builtins from acid/builtin.c.

Before diving into the commands, it is worth recalling the ambient state that most of them rely on. During initialization and process attachment, acid sets a handful of globals that the library commands read without passing them around explicitly. loadvars()[72b] creates the per-machine register variables (PC, SP, R0...) and a few predefined identifiers like bpinst, bplist, and notes; newproc()[81c] (and sproc()[85b] for acid pid) sets pid to the current target's process id and adds an entry to ptab[56b] so msg()[56d] knows which ctl file to write to; acidinit (the user's library-level hook) typically binds proc to pid as well. The commands below treat these as implicit arguments: bpset()[165a] reads bpinst and updates bplist, cont()[166] reads pid to drive start, and stk()[168a] reads PC and SP through cormap. Once you keep this set of globals in mind, the short bodies of the commands stop looking mysterious—they are mostly one-liners that route a control message or a memory read to "the current process", where the current process is whichever pid was last made the focus.

12.1 Start the program: new()

The new() function is the typical entry point for a debugging session. It clears the breakpoint list, calls newproc()[81c] to fork/exec the target, sets a breakpoint at main, continues to it, then removes the breakpoint. After new(), the target is stopped at the first instruction of main, ready for inspection.

⟨*function* new 161a⟩≡ (152)
```
defn new()
{
        bplist = {};
        newproc(progargs);
        // Dont miss the delay slot calls
        bpset(follow(main)[0]);
        cont();
        bpdel(*PC);
}
```

Uses bplistX, newproc(), bpset()[165a], cont()[166], bpdelX

⟨*global* progargs 161b⟩≡ (152)
```
progargs="";
```

⟨tab *entries* 161c⟩+≡ (68a) ◁148a 162a▷
```
"newproc", newproc,
```

12.2 Process control

This section presents the `acid` builtins for controlling the debugged process. They all follow the same pattern: evaluate the PID argument, then write a message to the process's `/proc/<pid>/ctl` file via `msg()`[56d]. The different messages correspond to the kernel operations described in Chapter 3: `start` resumes, `stop` halts, `startstop` resumes then waits, and `waitstop` just waits.

⟨`tab` *entries* 162a⟩+≡ (68a) ◁161c 162b▷
```
"start", start,
"startstop",startstop,
"stop", stop,
"waitstop", waitstop,
```
Uses `start()` 162c, `startstop()` 163a, `stop()` 162d, and `waitstop()` 163b.

⟨`tab` *entries* 162b⟩+≡ (68a) ◁162a
```
"status", status,
```
Uses `status()` 163c.

12.2.1 Resume: `start()`

⟨*function* `start` 162c⟩≡ (423b)
```
void
start(Node *r, Node *args)
{
    Node res;

    USED(r);
    if(args == nil)
        error("start(pid): no pid");
    expr(args, &res);
    if(res.type != TINT)
        error("start(pid): arg type");

    msg(res.ival, "start");
}
```
Uses `TINT` 60a and `msg()` 56d.

12.2.2 Halt: `stop()`

⟨*function* `stop` 162d⟩≡ (423b)
```
void
stop(Node *r, Node *args)
{
    Node res;

    USED(r);
    if(args == 0)
        error("stop(pid): no pid");
    expr(args, &res);
    if(res.type != TINT)
        error("stop(pid): arg type");

    Bflush(bout);

    msg(res.ival, "stop");

    notes(res.ival);
```

```
    dostop(res.ival);
}
```

Uses TINT 60a, bout 69d, dostop() 88a, and msg() 56d.

12.2.3 Resume and wait: startstop()

startstop()[163a] is the most important process control builtin: it resumes the target and blocks until the target stops again (at a breakpoint, fault, or system call). This is the core of the "continue until something happens" operation. After the target stops, it reads pending notes and calls the stopped callback.

⟨*function* startstop 163a⟩≡ (423b)

```
void
startstop(Node *r, Node *args)
{
    Node res;

    USED(r);
    if(args == nil)
        error("startstop(pid): no pid");
    expr(args, &res);
    if(res.type != TINT)
        error("startstop(pid): arg type");

    msg(res.ival, "startstop");

    notes(res.ival);
    dostop(res.ival);
}
```

Uses TINT 60a, dostop() 88a, and msg() 56d.

12.2.4 Wait for stop: waitstop()

⟨*function* waitstop 163b⟩≡ (423b)

```
void
waitstop(Node *r, Node *args)
{
    Node res;

    USED(r);
    if(args == 0)
        error("waitstop(pid): no pid");
    expr(args, &res);
    if(res.type != TINT)
        error("waitstop(pid): arg type");

    Bflush(bout);
    msg(res.ival, "waitstop");
    notes(res.ival);
    dostop(res.ival);
}
```

Uses TINT 60a, bout 69d, dostop() 88a, and msg() 56d.

12.2.5 Process status: status()

⟨*function* status 163c⟩≡ (423b)

```
void
```

```
status(Node *r, Node *args)
{
    Node res;
    char *p;

    USED(r);
    if(args == 0)
        error("status(pid): no pid");
    expr(args, &res);
    if(res.type != TINT)
        error("status(pid): arg type");

    p = getstatus(res.ival);

    r->string = strnode(p);
    r->op = OCONST;
    r->fmt = 's';
    r->type = TSTRING;
}
```

Uses OCONST 59, TINT 60a, TSTRING 60a, getstatus() 164, and strnode() 62g.

⟨*function* getstatus 164⟩≡ (392a)

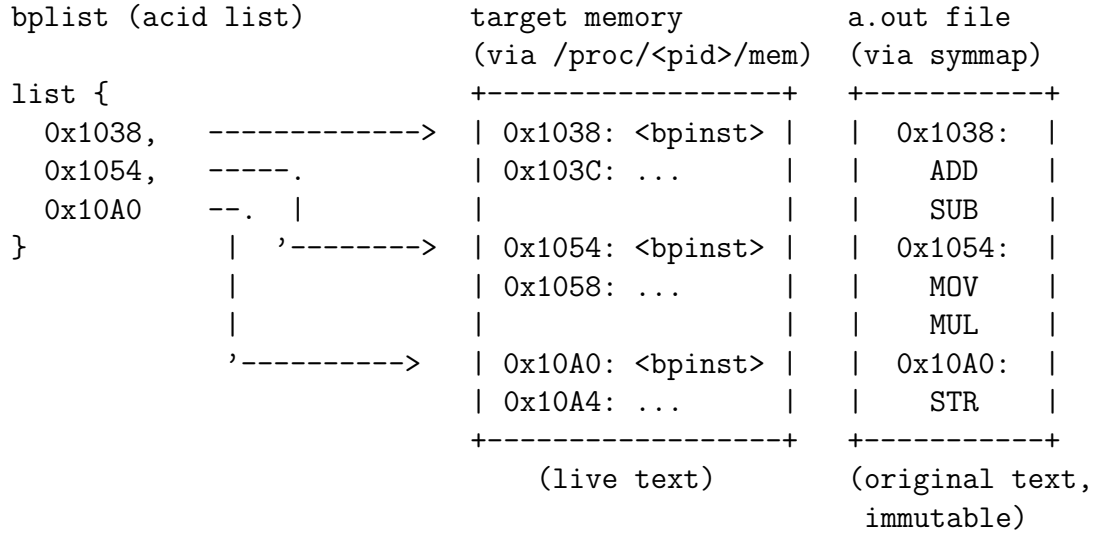
```
char *
getstatus(int pid)
{
    int fd, n;
    char *argv[16], buf[64];
    static char status[128];

    snprint(buf, sizeof(buf), "/proc/%d/status", pid);
    fd = open(buf, OREAD);
    if(fd < 0)
        error("open %s: %r", buf);

    n = read(fd, status, sizeof(status)-1);
    close(fd);
    if(n <= 0)
        error("read %s: %r", buf);
    status[n] = '\0';

    if(tokenize(status, argv, nelem(argv)-1) < 3)
        error("tokenize %s: %r", buf);

    return argv[2];
}
```

12.3 Breakpoints

The four breakpoint commands (`bpset`, `bpdel`, `cont`, `step`) are written entirely in the `acid` language, in `/lib/acid/port.acid`. They use the same building blocks the user would use at the prompt: `*addr` to write through `cormap`, `@addr` to read the original byte from `symmap`, `bpinst` (the architecture's breakpoint instruction string), and `startstop()` to resume the target. The actual state—which addresses currently have breakpoints—lives in the global `acid` list `bplist`, which the user can also inspect or manipulate directly.

Conceptually, the bookkeeping a traditional debugger stores in a C-level "breakpoint table" is kept here in two places: the active breakpoint instruction lives in the target's own memory (patched over the original via `/proc/<pid>/mem`), and the list of patched addresses lives in `acid`'s `bplist`. The "saved original instruction"

column that a gdb-style debugger would store in RAM is, in acid, looked up on demand from symmap via the @addr operator (read through the on-disk text) whenever the breakpoint is removed:



So bpdel(addr)'s *addr = @addr idiom reads right: "write to the live process ('*') the byte that the on-disk file ('@') has at the same address"—exactly what is needed to restore the original instruction, and the reason the symmap/cormap split exists in the first place. No separate saved-instruction column is needed because the on-disk text is treated as an immutable reference copy. Keeping bplist as a plain acid list means the user can print bplist at any prompt to see what is patched, or script batch operations over it with head/tail.

12.3.1 Setting a breakpoint: bpset(<func>)

bpset() is the simplest of the four. It first ensures the target is stopped (otherwise we cannot patch its memory), then checks for duplicates, and finally writes the breakpoint instruction at the target address (*fmt(addr, bpfmt) = bpinst) and adds it to bplist. The fmt(addr, bpfmt) dance is needed because bpinst is a multi-byte string and the * write needs to know the format width.

⟨*function* bpset 165a⟩≡ (152)

```
defn bpset(addr)                                // set a breakpoint
{
        if status(pid) != "Stopped" then {
                print("Waiting...\n");
                stop(pid);
        }
        if match(addr, bplist) >= 0 then
                print("breakpoint already set at ", fmt(addr, 'a'), "\n");
        else {
                *fmt(addr, bpfmt) = bpinst;
                bplist = append bplist, addr;
        }
}
```

bpdel() is the inverse: it restores the original instruction at the breakpoint address using the *addr = @addr idiom (read original from symmap via @, write to cormap via *). It then walks bplist and rebuilds it without the deleted entry—a typical functional-style list filter written in acid's own language.

⟨*function* bpdel 165b⟩≡ (152)

```
defn bpdel(addr)                                // delete a breakpoint
{
        local n, pc, nbplist;

        n = match(addr, bplist);
```

```
        if n < 0  then {
                print("no breakpoint at ", fmt(addr, 'a'), "\n");
                return {};
        }

        addr = fmt(addr, bpfmt);
        *addr = @addr;

        nbplist = {};                           // delete from list
        while bplist do {
                pc = head bplist;
                if pc != addr then
                        nbplist = append nbplist, pc;
                bplist = tail bplist;
        }
        bplist = nbplist;                       // delete from memory
}
```

12.3.2 Continue: `cont()`

`cont()` resumes execution. The non-obvious case is when the target is currently *stopped on a breakpoint*: if we just sent `startstop` now, the breakpoint instruction would fire again immediately and we would be stuck. So `cont()` first checks whether the current PC is in `bplist`, and if so, restores the original instruction, single-steps over it (so the original is executed once), then re-inserts the breakpoint before resuming. This is the same step-over-the-bp trick that real-hardware debuggers use; in `acid` it lives in the `acid` library rather than in C.

⟨*function* `cont` 166⟩≡ (152)

```
defn cont()                                     // continue execution
{
        local addr;

        addr = fmt(*PC, bpfmt);
        if match(addr, bplist) >= 0 then {      // Sitting on a breakpoint
                *addr = @addr;
                step();                         // Step over
                *addr = bpinst;
        }
        startstop(pid);                         // Run
}
```

12.3.3 One step execution: `step()`

`step()` is where the breakpoint dance from Chapter 3 actually happens. It is one of the cleverer pieces of `acid` because the ARM does not have a hardware single-step flag (unlike x86's `TF` bit or `ptrace(PTRACE\_SINGLESTEP)` on Linux). `acid` has to simulate single-step using only breakpoints:

1. If the PC is currently sitting on a breakpoint, save the breakpoint address and restore the original instruction (`*bput = @bput` reads the original byte from `symmap` and writes it through `cormap`).
2. Compute the follow set—the addresses of every instruction the next-executed instruction can possibly jump to (`follow(*PC)`). For a normal instruction this is just `*PC + 4`; for a branch it could be `*PC + 4` OR the branch target; for a return it is `*LR`; for a conditional branch it is two addresses.
3. Insert breakpoint instructions at every address in the follow set.
4. Resume the target with `startstop(pid)`. The target runs exactly one instruction and hits one of the new breakpoints.

5. Remove all the breakpoints from the follow set, restoring the original instructions.

6. If we cleared a real breakpoint at step 1, re-insert it.

The follow-set technique is the standard way to single-step on ARM and other architectures without a hardware step flag. The downside is that the follow-set computation needs to understand every branch instruction, which is why `machdata->foll` is architecture-specific.

⟨*function* `step` 167⟩≡ (152)

```
defn step()                                     // single step the process
{
        local lst, lpl, addr, bput;

        bput = 0;
        if match(*PC, bplist) >= 0 then {       // Sitting on a breakpoint
                bput = fmt(*PC, bpfmt);
                *bput = @bput;
        }

        lst = follow(*PC);

        lpl = lst;
        while lpl do {                          // place break points
                *(head lpl) = bpinst;
                lpl = tail lpl;
        }

        startstop(pid);                         // do the step

        while lst do {                          // remove the breakpoints
                addr = fmt(head lst, bpfmt);
                *addr = @addr;
                lst = tail lst;
        }
        if bput != 0 then
                *bput = bpinst;
}
```

12.4 Inspecting

We have already met the basic printing facilities: `print()` to dump a variable or register from the `acid` prompt, the REPL's automatic printing of any expression typed at top level, the `fmt()` helper that attaches a print format to a value, and the `\F` format modifier that applies a format inline to an expression. This section covers the step up from printing a value to inspecting process state: the call stack with `stk()`, the full register file with `regs()`, the disassembly around the current PC with `asm()`, and the enumeration of local, parameter, and global symbols via `locals()`, `params()`, and the `symbols` list.

12.4.1 Stack trace: `stk()`

The `stk()` command builds its output by walking the chain of call frames down from the current PC/SP. The heavy lifting is delegated to the `strace` builtin, which returns an `acid` list of 4-element frame records; `_stk()`[168c] just iterates over that list, printing one line per frame and updating `pc` to the caller's return address. Each frame record has this shape:

```
frame = { fnpc, retpc, params, locals }
```

```
|     |      |        |
|     |      |        +-- list { {name,addr}, ... }
|     |      +---------- list { {name,val},  ... }
|     +----------------- return PC into caller
+---------------------- entry PC of current fn
```

Visually, a three-deep stack trace for a crash in `baz()` called from `bar()` called from `main()` looks like:

```
                                  stk = strace(*PC,*SP,*R14)
                                  |
  high address                    |
  +---------------------+  <---- SP at entry to main
  | main frame          |         |
  |   .frame size       |         v
  |   params, locals    |    frame[0]: {main,  start_addr, params, locals}
  +---------------------+         |   ^
  | saved R14 (ret PC) -|---------|---'  = frame[1] of caller below
  +---------------------+         |
  | bar frame           |         v
  |   ...               |    frame[1]: {bar,   retpc_in_main, ...}
  +---------------------+         |
  | saved R14           |---.     v
  +---------------------+   |
  | baz frame           |   |     v
  |   ...               |   '---> frame[2]: {baz,   retpc_in_bar, ...}
  +---------------------+
  | current *SP         |  <---- walk starts here
  low address
```

At each iteration `_stk()` pulls the head frame, prints `fnpc(params)+offset file:line` (offset = `pc - frame[0]`), optionally dumps the `locals` list, and then sets `pc = frame[1]` so the next iteration's line number resolves inside the caller's body rather than at the call site. `stkignore` is a cosmetic layer on top: when a run of leading frames lives in files matching ignored regular expressions (typically runtime glue or system-call wrappers), only the last one of that run is printed, so the user sees their own frames without the bookkeeping noise.

⟨*function* `stk`*(arm)* 168a⟩≡ (158)
```
defn stk()                          // trace
{
        _stk(*PC, *SP, linkreg(0), 0);
}
```

⟨*function* `linkreg`*(arm)* 168b⟩≡ (158)
```
defn linkreg(addr)
{
        return *R14;
}
```

⟨*function* `_stk` 168c⟩≡ (152)
```
// print a stack trace
//
// in a run of leading frames in files matched by regexps in stkignore,
// only print the last one.
defn _stk(pc, sp, link, dolocals)
{
        local stk, ign, last, lastpc;
```

```
        stk = strace(pc, sp, link);
        if stkignore then
                ign = 1;
        else
                ign = 0;
        last = stk;
        lastpc = pc;
        while stk do {
                if ign then {
                        if !_stkign(pcfile(pc)) then {
                                ign = 0;
                                stk = last;
                                pc = lastpc;
                        }
                }
                frame = head stk;
                if !ign then {
                        print(stkprefix, fmt(frame[0], 'a'), "(");
                        params(frame[2]);
                        print(")+", itoa(pc-frame[0], "%ux"), " ");
                        pfl(pc);
                        if dolocals then
                                locals(frame[3]);
                }
                last = stk;
                lastpc = pc;
                stk = tail stk;
                pc = frame[1];
        }
        print(stkprefix, fmt(pc, 'a'), " ");
        pfl(pc);
}
```

12.4.2 Register dump: `regs()`

⟨*function* `regs` *(arm)* 169a⟩≡ (158)

```
defn regs()                             // print all registers
{
        gpr();
}
```

⟨*function* `gpr` *(arm)* 169b⟩≡ (158)

```
defn gpr()                      // print general purpose registers
{
        print("R0\t", *R0, " R1\t", *R1, " R2\t", *R2, "\n");
        print("R3\t", *R3, " R4\t", *R4, " R5\t", *R5, "\n");
        print("R6\t", *R6, " R7\t", *R7, " R8\t", *R8, "\n");
        print("R9\t", *R9, " R10\t", *R10, " R11\t", *R11, "\n");
        print("R12\t", *R12, " R13\t", *R13, " R14\t", *R14, "\n");
        print("R15\t", *R15, "\n");
}
```

12.4.3 Disassembly: `asm(<coderef>)`

⟨*function* `asm` 169c⟩≡ (152)

```
defn asm(addr)
{
```

```
        local bound;

        bound = fnbound(addr);

        addr = fmt(addr, 'i');
        loop 1,_asmlines do {
                print(fmt(addr, 'a'), " ", fmt(addr, 'X'));
                print("\t", @addr++, "\n");
                if bound != {} && addr > bound[1] then {
                        lasmaddr = addr;
                        return {};
                }
        }
        lasmaddr = addr;
}
```

⟨*constant* _asmlines 170a⟩≡ (152)

```
_asmlines = 30;
```

12.4.4 Locals and params: `locals()` and `params()`

⟨*function* locals 170b⟩≡ (152)

```
defn locals(l)
{
        local sym;

        while l do {
                sym = head l;
                print(stkprefix, "\t", sym[0], "=", itoa(sym[1], "%ux"), "\n");
                l = tail l;
        }
}
```

⟨*function* params 170c⟩≡ (152)

```
defn params(param)
{
        while param do {
                sym = head param;
                print(sym[0], "=", itoa(sym[1], "%ux"));
                param = tail param;
                if param then
                        print (",");
        }
}
```

12.4.5 Global symbols: `symbols` and `symbols()`

⟨*function* symbols 170d⟩≡ (152)

```
defn symbols(pattern)
{
        local l, s;

        l = symbols;
        while l do {
                s = head l;
                if regexp(pattern, s[0]) then
                        print(s[0], "\t", s[1], "\t", s[2], "\n");
                l = tail l;
```

```
        }
}
```

Chapter 13

Source level C Debugging

Source-level debugging requires cooperation between the compiler (which generates line number tables), the linker (which preserves them in the executable), and `libmach` (which provides lookup functions like `fileline()` and `pc2line()`). The `acid` library functions `src()` and `Bsrc()` use these to display source code around a given address.

13.1 Showing source: `src(<coderef>)`

The `src()` command takes an address (typically `*PC`, the current program counter) and prints the source code around that location, with the matching line marked. The library implementation is straightforward:

⟨*function* `src` 172⟩≡ (152)

```
defn src(addr)
{
        local src, file, line, cline, text;

        file = pcfile(addr);
        src = match(file, srcfiles);

        if src >= 0 then
                src = srctext[src];
        else
                src = findsrc(file);

        if src == {} then {
                print("no source for ", file, "\n");
                return {};
        }

        cline = pcline(addr)-1;
        print(file, ":", cline+1, "\n");
        line = cline-5;
        loop 0,10 do {
                if line >= 0 then {
                        text = src[line];
                        if text == {} then
                                return {};
                        if line == cline then
                                print(">");
                        else
                                print(" ");
                        print(line+1, "\t", text, "\n");
                }
                line = line+1;
```

```
        }
}
```

The two builtins doing the real work are:

- `pcfile()`[422a]: given an address, return the source file that contains it. Implemented in `libmach` by walking the `z`-type history entries to reconstruct the file path.
- `pcline()`[422b]: given an address, return the source line number. Implemented by binary-searching the line number-PC table the linker emitted.

Once the file and line are known, `src()` reads a window of nearby lines and prints them. The user can also use `Bsrc()` which opens the source file in an external editor (typically `acme`) at the right line—a primitive form of IDE integration.

13.2 One statement execution: `next()`

The `next()` command is the source-level "step over": it advances by one statement of the current function, treating any function call inside that statement as a single opaque step. This is what the user wants 99% of the time when single-stepping—raw `step()` (which advances by one machine instruction) would dive into every callee, including the entire C library.

The implementation is short but the algorithm deserves a walkthrough because it shows the trade-off `acid` makes relative to gdb. The idea is:

```
sp    = current stack pointer  (used as a "depth" marker)
bound = [start-pc, end-pc] of current function
         (from the symbol table)

stmnt()              ; advance by one source statement worth
                     ; of machine instructions, using the line
                     ; table to know when we've crossed a line
if PC still within bound -> done; we're on the next stmt

; otherwise we either fell out of the function or stepped
; into a callee. step() until we're back in the same
; function AND not deeper than where we started:
while PC outside bound and SP >= original sp:
    step()
```

The `sp >= *SP` guard is the trick. When you call a function, the stack *grows downward*, so the new `SP` is *less* than the saved one; the guard becomes false and the loop exits, leaving you stepping inside the callee. But wait—that is the opposite of what we want, isn't it? Read it again: the loop continues only while `sp >= *SP`, i.e. while we are at the same call depth or deeper. So as long as we are inside a callee, we keep `step`ping; the moment the callee returns and the stack pointer pops back up (making `*SP > sp`), the guard fails and we stop. This is exactly the "step over" semantics: dive in, run to completion, surface back.

The same problem in gdb is solved with DWARF-level frame descriptors and a hardware-assisted "step until SP changes" loop, but the underlying idea is identical. `acid` gets away with a four-line algorithm because it has only two levers it needs: the symbol table (for `fnbound`) and `*SP` (for depth comparison). The cost is

that `next()` cannot tell the difference between a normal return and a `longjmp` that unwinds several frames at once—both will satisfy `*SP > sp` and stop the loop, possibly far from the user's expected next statement.

⟨*function* `next` 174⟩≡ (152)

```
defn next()
{
        local sp, bound;

        sp = *SP;
        bound = fnbound(*PC);
        stmnt();
        pc = *PC;
        if pc >= bound[0] && pc < bound[1] then
                return {};

        while (pc < bound[0] || pc > bound[1]) && sp >= *SP do {
                step();
                pc = *PC;
        }
        src(*PC);
}
```

13.3 Displaying data-structures

Source-level inspection of *values* (not just lines) needs two more pieces:

- Frame walking: to display local variables, the debugger needs to find each function's stack frame on the call stack. The library function `stk()` uses the builtin `strace`[420a], which calls `libmach`'s `ctrace` follow-set walker. For each frame, `ctrace` returns the PC, frame pointer, and a list of local variable addresses (from the `a/p` symbol entries). The library then prints them via `print()`.
- Complex types: to display structs and unions, the user declares them as `complex` (or `aggr`) types using the syntax we saw in the Interpreting chapter. With a complex declaration loaded, the `.` operator dispatches to `odot()`[398b] which reads the right offset through `cormap`. Without it, `acid` only sees raw addresses and integers.

Together, these mechanisms approximate what gdb does with DWARF. The result is more limited (no inlining info, no support for optimized-out variables) but the implementation is two orders of magnitude smaller.

Chapter 14

Advanced Features

This chapter covers features that go beyond basic debugging: memory leak detection and code coverage (both implemented as `acid` library scripts), binary patching with `-w`, kernel debugging with `-k`, remote debugging with `-r`, and cross-architecture debugging with `-m`.

14.1 Memory leak detection: `/lib/acid/leak.acid`

The `/lib/acid/leak.acid` library implements memory leak detection in the style of a conservative garbage collector. It never sets a breakpoint and never intercepts `malloc` or `free`. Instead, it dumps the heap and every root region (bss, stack, and each thread's registers) from the target process through `acid`'s `*addr` read-from-target-memory operator, and prints the raw data for an external tool to post-process. Two facts make this work. First, Plan 9's pool allocator tags each block with a magic number: `ALLOC_MAGIC` for live blocks, `FREE_MAGIC` for freed ones (see `lib_core/libc/port/pool.c`). The script walks each arena's block chain by advancing `B2NB(b)` and inspecting `addr.magic`, so it can list every live allocation without ever running code in the target. Second, `malloc()` calls `setmalloctag()` (see `lib_core/libc/port/malloc.c`), which stores the caller PC in the padding longs sitting just before the user data. The script reads `*(addr+8)` and `*(addr+12)` to recover the allocator and reallocator PCs—so each dumped block carries a back-reference to the line of source that allocated it.

The "reachability" half of the algorithm runs outside `acid`. `dumpmem()` and `dumpregs()` walk every 4-byte word in bss, heap, stack, and register file, and print those whose value happens to fall inside the heap range (as judged by `isptr`). Feeding that to companion `rc` scripts (the `% also leak.rc, umem.rc, kmem.rc` comment) gives a graph of "which block is pointed to by which root or which other block", from which the set of unreachable blocks is the leak report. This is exactly how Boehm's conservative garbage collector (and its leak-detection mode) works: treat every aligned word as a potential pointer, and call anything not reachable from the roots garbage. The Plan 9 variant is striking because it needs no runtime support and no recompilation: pool tags and the padlong PC are already there, and `acid`'s `*addr` primitive does the rest. Compare Valgrind, which has to interpret every load and store through its synthetic CPU to track reachability, or ASan, which inserts redzone checks at compile time—both give more precise answers at a much higher engineering cost.

⟨*lib/acid/leak.acid* 175⟩≡

```
//
// usage: acid -l pool -l leak
//
include("/sys/src/libc/port/pool.acid");

defn
dumppool(p, sum)
{
        complex Pool p;
        a = p.arenalist;
```

```
        print("A: ", p.arenalist\X, "\n");
        while a != 0 && a < 0xff000000 do {
                complex Arena a;
                dumparena(a, sum);
                a = a.down;
        }
}

defn
dumparena(arena, sum)
{
        local atail, b, nb;

        atail = A2TB(arena);
        complex Bhdr arena;
        b = a;
        print("B: ", b\X, " ", atail\X, "\n");
        while b < atail && b.magic != ARENATAIL_MAGIC do {
                dumpblock(b, sum);
                nb = B2NB(b);
                if nb == b then {
                        print("B2NB(", b\X, ") = b\n");
                        b = atail;      // end loop
                }
                if nb > atail then {
                        b = (Bhdr)(b+4);
                        print("lost at block ", (b-4)\X, ", scanning forward\n");
                        while b < atail && b.magic != ALLOC_MAGIC && b.magic != FREE_MAGIC do
                                b = (Bhdr)(b+4);
                        print("stopped at ", b\X, " ", *b\X, "\n");
                }else
                        b = nb;
        }
        if b != atail then
                print("found wrong tail to arena ", arena\X, " wanted ", atail\X, "\n");
}

defn
isptr(a)
{
        if end <= a && a < xbloc then
                return 1;
        if 0xdefff000 <= a && a < 0xdffff000 then
                return 1;
        return 0;
}

lastalloc = 0;
lastcount = 0;
lastsize = 0;
defn
emitsum()
{
        if lastalloc then
                print("summary ", lastalloc\a, " ", lastcount\D, " ", lastsize\D, "\n");
        lastalloc = 0;
        lastcount = 0;
        lastsize = 0;
}
```

```
defn
dumpblock(addr, sum)
{
        complex Bhdr addr;

        if addr.magic == ALLOC_MAGIC || (!sum && addr.magic == FREE_MAGIC) then {
                local a, x, s;

                a = addr;
                complex Alloc a;

                x = addr+8;
                if sum then {
                        if *(addr+8) != lastalloc then {
                                emitsum();
                                lastalloc = *(addr+8);
                        }
                        lastcount = lastcount+1;
                        lastsize = lastsize+a.size;
                }else{
                        if addr.magic == ALLOC_MAGIC then
                                s = "block";
                        else
                                s = "free";
                        print(s, " ", addr\X, " ", a.size\X, " ");
                        print(*(addr+8)\X, " ", *(addr+12)\X, " ",
                                *(addr+8)\a, " ", *(addr+12)\a, "\n");
                }
        }
}

defn
dumprange(s, e, type)
{
        local x, y;

        print("range ", type, " ", s\X, " ", e\X, "\n");
        x = s;
        while x < e do {
                y = *(x\X);
                if isptr(y) then print("data ", x\X, " ", y\X, " ", type, "\n");
                x = x + 4;
        }
}

defn
stacktop()
{
        local e, m;

        m = map();
        while m != {} do {
                e = head m;
                if e[0] == "*data" then
                        return e[2];
                m = tail m;
        }
        return 0xdffff000;
}
```

```
defn
dumpmem()
{
        local s, top;

        xbloc = *bloc;
        // assume map()[1] is "data"
        dumprange(map()[1][1], end, "bss");     // bss
        dumprange(end, xbloc, "alloc"); // allocated

        top = stacktop() - 8;
        if top-0x01000000 < *SP && *SP < top then
                s = *SP;
        else
                s = top-32*1024;

        dumprange(s, top, "stack");
}

defn
dumpregs()
{
        dumprange(0, sizeofUreg, "reg");
}


defn
leakdump(l)
{
        print("==LEAK BEGIN==\n");
        dumppool(*mainmem, 0);
        dumpmem();
        dumpregs();
        while l != {} do {
                setproc(head l);
                dumpregs();
                l = tail l;
        }
        print("==LEAK END==\n");
}

defn
blockdump()
{
        print("==BLOCK BEGIN==\n");
        dumppool(*mainmem, 0);
        print("==BLOCK END==\n");
}

defn
blocksummary()
{
        print("==BLOCK BEGIN==\n");
        dumppool(*mainmem, 1);
        emitsum();
        print("==BLOCK END==\n");
}
```

14.2 Code coverage: `/lib/acid/coverage.acid`

The `/lib/acid/coverage.acid` library implements basic-block coverage, not function-level coverage. It walks every instruction in the text segment and calls `follow(e)` at each address, which returns the possible successor PCs. Most instructions have exactly one successor (fall-through), so `tail l ==` and they are ignored. When `follow` returns two successors—i.e. the instruction is a branch—the script adds *both* the fall-through and the branch target to the `bblock` set. These are exactly the basic-block entry points. `coverage()` then patches a breakpoint instruction into every entry in `bblock` and calls `cont()` in a loop. Each time the target stops on a known breakpoint, the script removes it from `bblock` (a hit block never stops the program again, so the target runs at near-native speed after the first hit of each block). When execution ends, whatever remains in `bblock` is the set of basic blocks that were never reached. `report()` then uses `pcline()` and `pcfile()` to translate those addresses back to source lines, giving a list of unexercised lines—the same granularity as `gcov`'s line coverage, but produced with no compiler instrumentation at all: `acid`'s symbol table and the architecture-specific `follow()` helper supply everything `gcov` normally gets from `-fprofile-arcs`.

⟨*lib/acid/coverage.acid* 179⟩≡

```
// Coverage library

defn coverage()
{
        local lmap, lp, e, pc, n, l;

        new();

        bblock = {};

        // find the first location in the text
        e = (map()[0][1])\i;

        while e < etext-4 do {
                l = follow(e);
                if tail l != {} then {
                        if match(l[0], bblock) < 0 then
                                bblock = append bblock, l[0];
                        if match(l[1], bblock) < 0 then
                                bblock = append bblock, l[1];
                }
                e++;
        }

        l = bblock;
        while l != {} do {
                *fmt(head l, bpfmt) = bpinst;
                l = tail l;
        }

        while 1 do {
                cont();
                pc = *PC;
                n = match(pc, bblock);
                if n >= 0 then {
                        pc = fmt(pc, bpfmt);
                        *pc = @pc;
                        bblock = delete bblock, n;
                }
                else {
                        pstop(pid);
                        return {};
```

```
                }
        }
}

defn eblock(addr)
{
        addr = addr\i;

        while addr < etext do {
                if (tail follow(addr)) != {} then
                        return pcline(addr);
                addr++;
        }
        return 0;
}

defn basic(stsrc, ensrc, file)
{
        local src, text;

        if stsrc >= ensrc then
                return {};

        print(file, ":", stsrc, ",", ensrc, "\n");
        src = match(file, srcfiles);

        if src >= 0 then
                src = srctext[src];
        else
                src = findsrc(file);

        if src == {} then
                print("no source for ", file, "\n");
        else {
                while stsrc <= ensrc do {
                        text = src[stsrc];
                        if text != {} then
                                print("\t", stsrc, ":", text, "\n");
                        stsrc = stsrc+1;
                }
        }
}

defn analyse(fnaddr)
{
        local addr, l, tfn;

        new();

        tfn = fnbound(fnaddr);

        l = bblock;
        while l do {
                addr = head l;

                if addr >= tfn[0] && addr < tfn[1] then
                        basic(pcline(addr), eblock(addr), pcfile(addr));

                l = tail l;
        }
```

```
        kill(pid);
}

defn report()
{
        local addr, l;

        new();

        l = bblock;
        while l do {
                addr = head l;

                basic(pcline(addr), eblock(addr), pcfile(addr));

                l = tail l;
        }
        kill(pid);
}

defn stopped(pid)
{
        return {};
}

print("/sys/lib/acid/coverage");
```

14.3 Yet another system call tracer: `/lib/acid/truss.acid`

The `/lib/acid/truss.acid` library implements system call tracing entirely in `acid`'s own language—it is a software-only alternative to `ratrace`. Loading it with `acid -l truss` adds functions that set breakpoints on system call entry points, decode the arguments, and print them in a readable format. This demonstrates `acid`'s programmability: even a tracer can be written as a debugging script.

⟨*lib/acid/truss.acid* 181⟩≡

```
// poor emulation of SVR5 truss command - traces system calls

include("/lib/acid/syscall.acid");

_stoprunning = 0;

defn stopped(pid) {
        local l;
        local pc;
        pc = *PC;
        if notes then {
                if (notes[0]!="sys: breakpoint") then
                {
                        print(pid,": ",trapreason(),"\t");
                        print(fmt(pc,97),"\t",fmt(pc,105),"\n");
                        print("Notes pending:\n");
                        l = notes;
                        while l do
                        {
                                print("\t",head l,"\n");
                                l = tail l;
                        }
```

```
                        _stoprunning = 1;
                }
        }
}

defn _addressof(pattern) {
        local s, l;
        l = symbols;
        pattern = "^\\$*"+pattern+"$";
        while l do
        {
                s = head l;
                if regexp(pattern, s[0]) && ((s[1] == 'T') || (s[1] == 'L')) then
                        return s[2];
                l = tail l;
        }
        return 0;
}

stopPC = {};
readPC = {};
fd2pathPC = {};
errstrPC = {};
awaitPC = {};
_waitPC = {};
_errstrPC = {};
trusscalls = {
                "sysr1",
                "_errstr",
                "bind",
                "chdir",
                "close",
                "dup",
                "alarm",
                "exec",
                "_exits",
                "_fsession",
                "fauth",
                "_fstat",
                "segbrk",
                "_mount",
                "open",
                "_read",
                "oseek",
                "sleep",
                "_stat",
                "rfork",
                "_write",
                "pipe",
                "create",
                "fd2path",
                "brk_",
                "remove",
                "_wstat",
                "_fwstat",
                "notify",
                "noted",
                "segattach",
                "segdetach",
                "segfree",
```

```
                "segflush",
                "rendezvous",
                "unmount",
                "_wait",
                "seek",
                "fversion",
                "errstr",
                "stat",
                "fstat",
                "wstat",
                "fwstat",
                "mount",
                "await",
                "pread",
                "pwrite",
        };

trussapecalls = {
                "_SYSR1",
                "__ERRSTR",
                "_BIND",
                "_CHDIR",
                "_CLOSE",
                "_DUP",
                "_ALARM",
                "_EXEC",
                "_EXITS",
                "__FSESSION",
                "_FAUTH",
                "__FSTAT",
                "_SEGBRK",
                "__MOUNT",
                "_OPEN",
                "__READ",
                "_OSEEK",
                "_SLEEP",
                "__STAT",
                "_RFORK",
                "__WRITE",
                "_PIPE",
                "_CREATE",
                "_FD2PATH",
                "_BRK_",
                "_REMOVE",
                "__WSTAT",
                "__FWSTAT",
                "_NOTIFY",
                "_NOTED",
                "_SEGATTACH",
                "_SEGDETACH",
                "_SEGFREE",
                "_SEGFLUSH",
                "_RENDEZVOUS",
                "_UNMOUNT",
                "__WAIT",
                "_SEEK",
                "__NFVERSION",
                "__NERRSTR",
                "_STAT",
                "__NFSTAT",
```

```
                "__NWSTAT",
                "__NFWSTAT",
                "__NMOUNT",
                "__NAWAIT",
                "_PREAD",
                "_PWRITE",
        };

defn addressof(pattern) {
        // translate to ape system calls if we have an ape binary
        if _addressof("_EXITS") == 0 then
                return _addressof(pattern);
        return _addressof(trussapecalls[match(pattern, trusscalls)]);
}

defn setuptruss() {
        local lst, offset, name, addr;

        trussbpt = {};
        offset = trapoffset();
        lst = trusscalls;
        while lst do
        {
                name = head lst;
                lst = tail lst;
                addr = addressof(name);
                if addr then
                {
                        bpset(addr+offset);
                        trussbpt = append trussbpt, (addr+offset);
                        // sometimes _exits is renamed $_exits
                        if(regexp("exits|exec", name)) then stopPC = append stopPC, (addr+offset);
                        if(regexp("read", name)) then readPC = append readPC, (addr+offset);
                        if(regexp("fd2path", name)) then fd2pathPC = append fd2pathPC, (addr+offset);
                        if(regexp("^\\$*await", name)) then awaitPC = append awaitPC, (addr+offset);
                        if(regexp("^\\$*errstr", name)) then errstrPC = append errstrPC, (addr+offset);
                        // compatibility hacks for old kernel
                        if(regexp("_wait", name)) then _waitPC = append _waitPC, (addr+offset);
                        if(regexp("_errstr", name)) then _errstrPC = append _errstrPC, (addr+offset);
                }
        }
}

defn trussflush() {
        stop(pid);              // already stopped, but flushes output
}

defn new() {
        bplist = {};
        newproc(progargs);
        bpset(follow(main)[0]);
        cont();
        bpdel(*PC);
        // clear the hang bit, which is left set by newproc, so programs we fork/exec don't hang
        printto("/proc/"+itoa(pid)+"/ctl", "nohang");
}

defn truss() {
        local pc, lst, offset, prevpc, pcspret, ret;
```

```
        offset = trapoffset();

        stop(pid);
        _stoprunning = 0;
        setuptruss();
        pcspret = UPCSPRET();

        while !_stoprunning do {
                cont();
                if notes[0]!="sys: breakpoint" then {
                        cleantruss();
                        return {};
                }
                pc = *PC;
                if match(*PC, stopPC)>=0 then {
                        print(pid,": ",trapreason(),"\t");
                        print(fmt(pc,'a'),"\t",fmt(pc,'i'),"\n");
                        cleantruss();
                        return {};
                }
                if match(*PC, trussbpt)>=0 then {
                        usyscall();
                        trussflush();
                        prevpc = *PC;
                        step();
                        ret = eval pcspret[2];
                        print("\treturn value: ", ret\D, "\n");
                        if (ret>=0) && (match(prevpc, readPC)>=0) then {
                                print("\tdata: ");
                                printtextordata(*((eval pcspret[1])+4), ret);
                                print("\n");
                        }
                        if (ret>=0) && (match(prevpc, fd2pathPC)>=0) then {
                                print("\tdata: \"", *(*((eval pcspret[1])+4)\s), "\"\n");
                        }
                        if (ret>=0) && (match(prevpc, errstrPC)>=0) then {
                                print("\tdata: \"", *(*(eval pcspret[1])\s), "\"\n");
                        }
                        if (ret>=0) && (match(prevpc, awaitPC)>=0) then {
                                print("\tdata: ");
                                printtextordata(*(eval pcspret[1]), ret);
                                print("\n");
                        }
                        // compatibility hacks for old kernel:
                        if (ret>=0) && (match(prevpc, _waitPC)>=0) then {
                                print("\tdata: ");
                                printtextordata(*(eval pcspret[1]), 12+3*12+64);
                                print("\n");
                        }
                        if (ret>=0) && (match(prevpc, _errstrPC)>=0) then {
                                print("\tdata: ");
                                printtextordata(*(eval pcspret[1]), 64);
                                print("\n");
                        }
                }
                trussflush();
        }
}

defn cleantruss() {
```

```
        local lst, offset, addr;

        stop(pid);
        offset = trapoffset();
        lst = trussbpt;
        while lst do
        {
                addr = head lst;
                lst = tail lst;
                bpdel(addr);
        }
        trussbpt = {};
        **PC = @*PC;    // repair current instruction
}

defn untruss() {
        cleantruss();
        start(pid);
}

print("/lib/acid/truss");
```

⟨*lib/acid/syscall.acid* 186⟩≡

```
// print system calls
defn printstring(s)
{
        print("\"", s, "\"");
}

defn printtextordata(addr, n)
{
        local a, i;

        a = addr\c;
        i = 0;
        loop 1, n do {
                if (a[i]>=127) then {
                        print(fmt(addr, 'X'), ", ", n\D);
                        return {};
                }
                i = i+1;
        }

        print("\"");
        printstringn(addr, n);
        print("\"");
}

defn printstringn(s, n)
{
        local m;

        m = n;
        if (m > 100) then m = 100;
        loop 1,m do {
                print(*(s\c)); s=s+1;
        }
        if(m != n) then print("...");
}
```

```
defn printsyscall(name, fmt, arg) {
        local f, i, a, argp, sl;

        print(name, "(");
        i = 0;
        a = eval arg;
        while fmt[i] != 0 do {
                if fmt[i] == 's' then {
                        if *a == 0 then
                                print("nil");
                        else
                                printstring(*(*a\s));
                } else if fmt[i] == 'S' then {
                        argp = *a;
                        argl = {};
                        while *argp != 0 do {
                                argl = append argl, *(*argp\s);
                                argp++;
                        }
                        print(argl);
                } else if (fmt[i] == 'Z') && (~*a == 0) then {
                        print("-1");
                        a++;    // advance extra word for quadword
                } else if (fmt[i] == 'Y') || (fmt[i] == 'V') then {
                        print(fmt(*a, fmt[i]));
                        a++;    // advance extra word for quadword
                } else if (fmt[i] == 'T') then {
                        if *a == 0 then
                                print("nil");
                        else
                                printtextordata(*a, a[1]);
                } else
                        print(fmt(*a, fmt[i]));
                if fmt[i+1] != 0 then
                        print(", ");
                i = i+1;
                a++;
        }
        print(")\n");
}

defn code(*e) { return e; }

syscalls = {
        { 0, {"sysr1",          "s",            code(0)}},
        { 1, {"_errstr",                "s",            code(*sys_errstr:arg)}},
        { 2, {"bind",           "ssX",          code(*sysbind:arg)}},
        { 3, {"chdir",          "s",            code(*sysbind:arg)}},
        { 4, {"close",          "D",            code(*sysclose:arg)}},
        { 5, {"dup",            "DD",           code(*sysdup:arg)}},
        { 6, {"alarm",          "D",            code(*sysalarm:arg)}},
        { 7, {"exec",           "sS",           code(*sysexec:arg)}},
        { 8, {"exits",          "s",            code(*sysexits:arg)}},
        { 9, {"_fsession",      "DX",           code(*sys_fsession:arg)}},
        {10, {"fauth",          "DX",           code(*sysfauth:arg)}},
        {11, {"_fstat",         "DX",           code(*sys_fstat:arg)}},
        {12, {"segbrk",         "XX",           code(*syssegbrk:arg)}},
        {13, {"_mount",         "DsXs",         code(*sys_mount:arg)}},
        {14, {"open",           "sD",           code(*sysopen:arg)}},
```

```
        {15, {"_read",          "DXD",          code(*sys_read:arg)}},
        {16, {"oseek",          "DDD",          code(*sysoseek:arg)}},
        {17, {"sleep",          "D",            code(*syssleep:arg)}},
        {18, {"_stat",          "sX",           code(*sys_stat:arg)}},
        {19, {"rfork",          "X",            code(*sysstat:arg)}},
        {20, {"_write",         "DXD",          code(*sys_write:arg)}},
        {21, {"pipe",           "X",            code(*syspipe:arg)}},
        {22, {"create",         "sDO",          code(*syscreate:arg)}},
        {23, {"fd2path",        "DXD",          code(*sysfd2path:arg)}},
        {24, {"brk_",           "X",            code(*sysbrk_:arg)}},
        {25, {"remove",         "s",            code(*sysremove:arg)}},
        {26, {"_wstat",         "sX",           code(*sys_wstat:arg)}},
        {27, {"_fwstat",                "DX",           code(*sys_fwstat:arg)}},
        {28, {"notify",         "X",            code(*sysnotify:arg)}},
        {29, {"noted",          "D",            code(*sysnoted:arg)}},
        {30, {"segattach",      "DsXD",         code(*syssegattach:arg)}},
        {31, {"segdetach",      "X",            code(*syssegdetach:arg)}},
        {32, {"segfree",        "XD",           code(*syssegfree:arg)}},
        {33, {"segflush",       "XD",           code(*syssegflush:arg)}},
        {34, {"rendezvous",     "XX",           code(*sysrendezvous:arg)}},
        {35, {"unmount",        "ss",           code(*sysunmount:arg)}},
        {36, {"_wait",          "X",            code(*sys_wait:arg)}},
        {39, {"seek",           "XDVD",         code(*sysseek:arg)}},
        {40, {"fversion",       "DDsD",         code(*sysfversion:arg)}},
        {41, {"errstr",         "TD",           code(*syserrstr:arg)}},
        {42, {"stat",           "sXD",          code(*sysstat:arg)}},
        {43, {"fstat",          "DXD",          code(*sysfstat:arg)}},
        {44, {"wstat",          "sXD",          code(*syswstat:arg)}},
        {45, {"fwstat",         "DXD",          code(*sysfwstat:arg)}},
        {46, {"mount",          "DDsXs",                code(*sysmount:arg)}},
        {47, {"await",          "TD",           code(*sysawait:arg)}},
        {50, {"pread",          "DXDZ",         code(*syspread:arg)}},
        {51, {"pwrite",         "DTDZ",         code(*syspwrite:arg)}},
};

defn syscall() {
        local n, sl, h, p;

        map({"*data", 0, 0xffffffff, 0});
        n = *syscall:scallnr;
        sl = syscalls;
        while sl != {} do {
                h = head sl;
                sl = tail sl;

                if n == h[0] then {
                        p = h[1];
                        printsyscall(p[0], p[1], p[2]);
                }
        }
}

defn UPCSPRET() {
        // return sys call number, address of first argument, location of syscall return value
        if objtype == "386" then
                return { code(*(*PC-4)), code(*SP+4), code(*AX) };
        if (objtype == "mips") || (objtype == "mips2") then
                return { code(*(*PC-4) & 0xffff), code(*SP+4), code(*R1) };
        if objtype == "arm" then
                return { code(*(*PC-4) & 0xffff), code(*SP+4), code(*R0) };     // untested
```

```
        if objtype == "alpha" then
                return { code(*(*PC-4) & 0xffff), code(*SP+4), code(*R0) };     // untested
}

defn trapoffset() {
        // return offset from entry point to trap instr
        if objtype == "386" then return 5;
        if objtype == "mips" then return 8;
        if objtype == "mips2" then return 8;
        if objtype == "arm" then return 8;      // untested
        if objtype == "alpha" then return 8;    // untested
}

defn trapreason() {
        // return reason for trap
        if objtype == "386" then return reason(*TRAP);
        if objtype == "mips" then return reason(*CAUSE);
        if objtype == "mips2" then return reason(*CAUSE);
        if objtype == "arm" then return "unknown trap"; // untested
        if objtype == "alpha" then return reason(cause);        // untested
}


defn usyscall() {       // gives args for system call in user level; not useful with -k
        local n, sl, h, p;

        // stopped at TRAP instruction in system call library
        pcsp = UPCSPRET();
        n = eval pcsp[0];
        sl = syscalls;
        while sl != {} do {
                h = head sl;
                sl = tail sl;

                if n == h[0] then {
                        p = h[1];
                        printsyscall(p[0], p[1], pcsp[1]);
                }
        }
}
```

14.4 Multi-threaded debugging

14.5 Modifying code: `acid -w`

Normally `acid` opens the executable file read-only. With `-w`, it opens it for writing too, allowing the user to patch the binary on disk—for instance, to replace an instruction with a `NOP` or change a constant. This is a powerful but dangerous feature; changes are permanent in the file. A typical use case is patching a vendor binary you do not have the source for: change a hard-coded path, disable an expired license check, fix a buggy instruction, or insert a permanent breakpoint at a function entry. Another use is "binary surgery" on a self-modifying binary or one whose source is lost.

⟨*global wtflag (acid/globals.c)* 189⟩≡ (381a)

```
bool wtflag;
```

⟨main() *(acid) command line processing* 190a⟩+≡ (69e) ◁71g 190d▷
```
case 'w':
    wtflag = true;
    break;
```

⟨attachfiles() *if* wtflag 190b⟩≡ (73e)
```
if(wtflag)
    text = open(aout, ORDWR);
```
Uses text 73d.

Once the file is opened with ORDWR, the symmap segments inherit a writable file descriptor, and writes through the Map abstraction reach the underlying file. The user-facing way to modify the binary is the @ operator (the OINDC opcode in oindc()[402e]), which dereferences through symmap (the file) rather than cormap (live memory). So @*0x1020 = 0x90 patches the byte at file offset corresponding to virtual address 0x1020 to 0x90 (an x86 NOP). Without -w, symmap is opened read-only and any @ write fails.

14.6 Kernel debugging: acid -k

With -k, acid debugs the kernel rather than a user process. It reads kernel registers from /proc/<pid>/kregs instead of /proc/<pid>/regs, and uses the kernel's text file as the executable. You may wonder: "the kernel has no PID, how can you attach to it via /proc/<pid>?" The trick is that on Plan 9, every kernel-mode *thread* (or processor in the SMP case) has a PID just like user processes. The scheduler, the page fault handler, and so on all run in kernel space but inside a process structure with a PID, visible in /proc like any other process. So acid -k attaches to a specific kernel thread's PID and inspects its kernel-mode register state. The chicken-and-egg concern ("the kernel provides /proc, how can it debug itself?") is real but limited to truly fatal crashes: as long as the kernel is running well enough to serve /proc requests, acid -k works. For deeper crashes you need a separate kernel debugger (typically running on another machine) connected over a serial line, or a post-mortem snapshot. acid -k is meant for inspecting a live but well-behaved kernel, not for catastrophic crashes.

⟨*global* kernel 190c⟩≡ (381a)
```
int kernel;
```

⟨main() *(acid) command line processing* 190d⟩+≡ (69e) ◁190a 191d▷
```
case 'k':
    kernel++;
    break;
```

⟨main() *(acid) if kernel and no pid* 190e⟩≡ (69e)
```
if(kernel) {
    fprint(STDERR, "acid: -k requires a pid\n");
    usage();
}
```

⟨checkqid() return if [[kernel 190f⟩≡ (87a)
```
if(kernel)
    return;
```
Uses kernel 190c.

⟨main() *(acid) when* acid pid *if* kernel 190g⟩≡ (88c)
```
if(kernel)
   aout = system();
```

14.7 Remote debugging: `acid -r`

Remote debugging exploits Plan 9's `import` command: you can import a remote machine's `/proc` into your local namespace, then run `acid -r` against it as if the process were local. The `-r` flag implies `-k` (kernel mode) and defaults PID to 1. This is a beautiful demonstration of Plan 9's "everything is a file" philosophy—the same `/proc` interface works across network boundaries.

⟨*global* `remote` 191a⟩≡ (381a)
```
int remote;
```

⟨`main()` *(acid) if argc and remote adjust* `aout` 191b⟩≡ (69e)
```
if(remote)
    aout = argv[0];
```

⟨`main()` *(acid) if not argc and remote adjust* `aout` 191c⟩≡ (69e)
```
if(remote)
    aout = "/mips/9ch";
```

⟨`main()` *(acid) command line processing* 191d⟩+≡ (69e) ◁190d 191f▷
```
case 'r':
    pid = 1;
    remote++;
    kernel++;
    break;
```

14.8 Cross architecture debugging: `acid -m`

With `-m`, `acid` can debug a binary compiled for a different architecture than the host. For instance, you can run `acid -m arm` on an x86 machine to disassemble and debug an ARM binary. Combined with `import`'ed `/proc` from a remote ARM machine, this gives full cross-architecture debugging. The `-m` flag overrides the automatic architecture detection from `crackhdr()`[90].

⟨*global* `mtype` 191e⟩≡ (383c)
```
static char* mtype;
```

⟨`main()` *(acid) command line processing* 191f⟩+≡ (69e) ◁191d
```
case 'm':
    mtype = ARGF();
    break;
```

⟨`main()` *(acid) sanity check* `mtype` 191g⟩≡ (72a)
```
if(mtype && machbyname(mtype) == 0)
    print("unknown machine %s", mtype);
```

⟨`readtext()` *if* `mtype != nil` 191h⟩≡ (74d)
```
if(mtype != nil){
    symmap = newmap(0, 1);
    if(symmap == 0)
        print("%s: (error) loadmap: cannot make symbol map\n", argv0);
    length = 1<<24;
    d = dirfstat(text);
    if(d != nil){
        length = d->length;
        free(d);
    }
    setmap(symmap, text, 0, length, 0, "binary");
    return;
}
```
Uses `mtype-16` 191e, `newmap()` 323a, `setmap()` 323b, and `text` 73d.

Chapter 15

Conclusion

You now know how the Plan 9 debugger `acid` works, to the smallest details, and more generally how many debuggers work.

A debugger may seem like magic—setting breakpoints, inspecting registers, single-stepping through instructions—but the underlying mechanism is remarkably simple: replace an instruction with one that faults, let the kernel stop the process, then read and write the process's memory and registers through `/proc`. The entire debugger fits in a few thousand lines of C because the kernel's `/proc` file system does the heavy lifting: `ctl` for control, `mem` for memory access, `regs` for registers, `text` for the executable. Along the way, you have seen a programmable debugger, the `libmach` library for reading symbol tables and disassembling instructions, and the breakpoint state machine that saves and restores the original instruction around each stop.

15.1 Patterns and techniques

`acid` is small, but the techniques behind it apply far beyond debugging:

- *Embedded interpreter for tool extensibility.* `acid` is a debugger built as an interpreter for its own language, so the user can write new debugging functions instead of being limited to a fixed set of commands. This pattern is shared by many successful tools: Emacs (Emacs Lisp), Vim (Vimscript), AutoCAD (AutoLISP), GIMP (Script-Fu), Maya (MEL), Blender (Python). Once a tool grows past a few dozen commands, exposing them as functions of a small embedded language scales much better than adding more flags or special syntax.

- *Synthetic file system as a uniform control surface.* The kernel exposes process state as files under `/proc` rather than as a custom system call. The same idea reappears in Linux's `/proc` and `/sys`, FUSE, `cgroups`, and the FreeBSD `procfs`. Once your interface is files, you get scriptability, remote access, and access control for free—all standard filesystem operations apply.

- *Save/modify/restore for transparent instrumentation.* The breakpoint cycle (save the original instruction, write a trap, wait for the fault, restore the instruction, step one, re-insert) is a save/modify/restore pattern that also shows up in database transactions, copy-on-write filesystems, text-editor undo/redo, and live patching of running programs. The original is preserved so the modification can be reverted at any time.

- *Abstraction layer for binary formats.* `libmach` hides the differences between executable formats and architectures behind a uniform interface (parse header, read symbol, disassemble instruction, walk stack). LLVM's `libObject` does the same for ELF/Mach-O/COFF, image libraries do it for PNG/JPEG/GIF, and JDBC does it for relational databases. This separation lets higher-level code work with the *meaning* of the data without caring about its *encoding*.

15.2 Connections to other books

- KERNEL book [Pad14]: the kernel provides the infrastructure that makes debugging possible—the `/proc` file system, the `Broken` process state (which lets you attach to a crashed process without needing a core dump), and the `note` mechanism for delivering signals to the debugged process.
- LINKER book [Pad15c]: `5l` generates the symbol table and line number information that `acid` reads through `libmach` to map addresses to function names and source lines.
- ASSEMBLER book [Pad15a]: understanding the ARM instruction set is essential for reading disassembly output and understanding how breakpoints work (replacing an instruction with `BKPT`).
- COMPILER book [Pad16a]: source-level debugging requires metadata generated by `5c`—local variable locations, type information, and the mapping between source lines and machine addresses.

15.3 Beyond the Plan 9 debuggers

Modern debuggers have evolved far beyond `acid` and `db`'s command-line interface. Here are some of the features they provide:

- *DWARF debug information*: `acid` reads Plan 9's simple symbol table and line number format. Modern debuggers (GDB, LLDB) consume DWARF, a rich and complex standard that encodes variable locations (including optimized-out variables and register assignments), type hierarchies, inlined function boundaries, and macro definitions. DWARF is what makes source-level debugging of optimized code possible.
- *Scriptable debuggers*: GDB has a Python scripting interface, and LLDB is built around a Python API from the ground up. Plan 9's `acid` takes a different approach—it is a debugger built as an interpreter for a custom C-like language, letting users write debugging scripts that understand data structures and can walk linked lists or dump hash tables.
- *Time-travel debugging*: tools like `rr` (for Linux) can record an entire program execution and replay it deterministically, allowing you to step backwards as well as forwards. This is transformative for debugging race conditions and heisenbugs.
- *Graphical and IDE integration*: most developers today use debuggers through their IDE (VS Code, CLion, Xcode), with visual breakpoint management, variable watches, and call stack navigation. The Debug Adapter Protocol (DAP) provides a standard interface between editors and debuggers.
- *Dynamic tracing*: Linux's `strace` is functionally similar to Plan 9's `ratrace`, but tools like DTrace and eBPF go much further, allowing programmable tracing of kernel and user-space functions with minimal overhead. These have largely replaced traditional system call tracing for production debugging.
- *Memory debuggers*: tools like Valgrind and AddressSanitizer detect memory errors (use after free, buffer overflows, uninitialized reads) by instrumenting memory accesses, which can find subtle bugs that breakpoint-based debugging cannot catch. `acid` addresses a small slice of this through the `/lib/acid/leak.acid` script (conservative reachability analysis over the heap and root regions, using the pool allocator's magic tags and the per-block caller `PC` stored by `setmalloctag`) and the `/lib/acid/malloc.acid` script (heap inspection), but these are far less comprehensive than Valgrind: they catch leaks but not buffer overflows, uninitialized reads, or use-after-free. That is essentially a consequence of the "passive observer" approach: `leak.acid` only ever *reads* target memory through the `*addr` primitive, so it can tell which blocks are unreachable but cannot see an out-of-bounds store or an uninitialized load as they happen. Catching those requires either a synthetic CPU (Valgrind) or compile-time rewriting (ASan)—both of which intercept every load and store, at much higher engineering and runtime cost.

The core idea, however, has not changed: a debugger is a program that controls another program, using kernel facilities to stop, inspect, and modify the target's execution. `acid` presents this idea in its simplest form, with the Plan 9`/proc` interface making the mechanism especially transparent.

Appendix A

Debugging

This appendix collects the facilities `acid` provides for debugging itself—the meta question every programmable debugger has to face eventually. A user writing non-trivial `acid` functions inevitably hits a bug in their own code (or in a library function like `stk()`) and wants the same inspection tools for `acid` that `acid` provides for C programs. Rather than shipping a separate meta-debugger, Winterbottom wrote a handful of builtins—`whatis` for self-introspection, the trace mode flags on the interpreter, and a few helpers described below—that let `acid` debug itself from inside the `acid` prompt.

A.1 `acid: whatis`

Like Emacs, `acid` is a self-documenting program. The `whatis` command lets the user inspect any name in the system: variable values and their formats, function definitions (printed back as `acid` source), type descriptions (the `complex/aggr` layouts from `cc -a`), and even lists of all defined functions. This means the user never has to leave the debugger to look up what a library function does or how a type is laid out—just type `whatis stk` or `whatis Proc`.

Note that `whatis` is not a builtin function dispatched through `tab[]` like `start()` or `stop()`. It is a keyword of the `acid` language, parsed directly by the yacc grammar (the `OWHAT` token) and dispatched through the `expop` table like other expression operators. This means you write `whatis name` (no parentheses, no arguments), not `whatis("name")`.

⟨`expop` *entries* 195a⟩+≡ (121f) ◁122

```
  [OWHAT] owhat,
```

Uses `owhat()` 195b.

⟨*function* `owhat` 195b⟩≡ (407c)

```
  void
  owhat(Node *n, Node *res)
  {
      res->op = OCONST;           /* Default return value */
      res->type = TLIST;
      res->l = 0;
      whatis(n->sym);
  }
```

Uses `OCONST` 59, `TLIST` 60a, and `whatis()` 195c.

⟨*function* `whatis` 195c⟩≡ (400c)

```
  /// user:whatis -> execute -> expr -> expop -> owhat -> <>
  void
  whatis(Lsym *l)
  {
      int t;
```

```
    bool def = false;
    Type *ti;

    if(l == nil) {
        fundefs();
        return;
    }

    if(l->v->set) {
        t = l->v->type;
        Bprint(bout, "%s variable", typenames[t]);
        if(t == TINT || t == TFLOAT)
            Bprint(bout, " format %c", l->v->fmt);
        if(l->v->comt)
            Bprint(bout, " complex %s", l->v->comt->base->name);
        Bputc(bout, '\n');
        def = true;
    }
    if(l->lt) {
        Bprint(bout, "complex %s {\n", l->name);
        for(ti = l->lt; ti; ti = ti->next) {
            if(ti->type) {
                if(ti->fmt == 'a') {
                    Bprint(bout, "\t%s %d %s;\n",
                    ti->type->name, ti->offset,
                    ti->tag->name);
                }
                else {
                    Bprint(bout, "\t'%c' %s %d %s;\n",
                    ti->fmt, ti->type->name, ti->offset,
                    ti->tag->name);
                }
            }
            else
                Bprint(bout, "\t'%c' %d %s;\n",
                ti->fmt, ti->offset, ti->tag->name);
        }
        Bprint(bout, "};\n");
        def = true;
    }
    if(l->proc) {
        Bprint(bout, "defn %s(", l->name);
        pexpr(l->proc->left);
        Bprint(bout, ") {\n");
        pcode(l->proc->right, 1);
        Bprint(bout, "}\n");
        def = true;
    }
    if(l->builtin) {
        Bprint(bout, "builtin function\n");
        def = true;
    }
    if(!def)
        Bprint(bout, "%s is undefined\n", l->name);
}
```

Uses TFLOAT 60a, TINT 60a, bout 69d, fundefs() 197a, pcode() 198b, pexpr() 200, and typenames 203b.

A.1.1 Variable

A.1.2 Function

A.1.3 Builtin

A.1.4 Complex

A.1.5 List of functions

⟨*function* `fundefs` 197a⟩≡ (400c)

```
void
fundefs(void)
{
    Lsym *l;
    char **vec;
    int i, j, n, max, col, f, g, s;

    max = 0;
    f = 0;
    g = 100;
    vec = malloc(sizeof(char*)*g);
    if(vec == 0)
        fatal("out of memory");

    for(i = 0; i < Hashsize; i++) {
        for(l = hash[i]; l; l = l->hash) {
            if(l->proc == 0 && l->builtin == 0)
                continue;
            n = strlen(l->name);
            if(n > max)
                max = n;
            if(f >= g) {
                g *= 2;
                vec = realloc(vec, sizeof(char*)*g);
                if(vec == 0)
                    fatal("out of memory");
            }
            vec[f++] = l->name;
        }
    }
        qsort(vec, f, sizeof(char*), cmp);
    max++;
    col = 60/max;
    s = (f+col-1)/col;

    for(i = 0; i < s; i++) {
        for(j = i; j < f; j += s)
            Bprint(bout, "%-*s", max, vec[j]);
        Bprint(bout, "\n");
    }
}
```

Uses `Hashsize` 66b, `bout` 69d, and `cmp()` 198a.

A.2 AST dumper

⟨*global* `tabs` 197b⟩≡ (400c)

```
static char *tabs = "\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t";
```

Uses tabs-65 197b.

⟨*function* cmp 198a⟩≡ (400c)
```
int
cmp(void *va, void *vb)
{
    char **a = va;
    char **b = vb;

    return strcmp(*a, *b);
}
```

A.2.1 Statement dumper: pcode()

⟨*function* pcode 198b⟩≡ (400c)
```
/// (user:whatis -> ... -> whatis) | bprint -> <>
void
pcode(Node *n, int d)
{
    Node *r, *l;

    if(n == 0)
        return;

    r = n->right;
    l = n->left;

    switch(n->op) {
    default:
        Bprint(bout, "%.*s", d, tabs);
        pexpr(n);
        Bprint(bout, ";\n");
        break;
    case OLIST:
        pcode(n->left, d);
        pcode(n->right, d);
        break;
    case OLOCAL:
        Bprint(bout, "%.*slocal", d, tabs);
        while(l) {
            Bprint(bout, " %s", l->sym->name);
            l = l->left;
            if(l == 0)
                Bprint(bout, ";\n");
            else
                Bprint(bout, ",");
        }
        break;
    case OCOMPLEX:
        Bprint(bout, "%.*scomplex %s %s;\n", d, tabs, n->sym->name, l->sym->name);
        break;
    case OIF:
        Bprint(bout, "%.*sif ", d, tabs);
        pexpr(l);
        d++;
        Bprint(bout, " then\n");
        if(r && r->op == OELSE) {
            slist(r->left, d);
            Bprint(bout, "%.*selse\n", d-1, tabs);
```

```
            slist(r->right, d);
        }
        else
            slist(r, d);
        break;
    case OWHILE:
        Bprint(bout, "%.*swhile ", d, tabs);
        pexpr(l);
        d++;
        Bprint(bout, " do\n");
        slist(r, d);
        break;
    case ORET:
        Bprint(bout, "%.*sreturn ", d, tabs);
        pexpr(l);
        Bprint(bout, ";\n");
        break;
    case ODO:
        Bprint(bout, "%.*sloop ", d, tabs);
        pexpr(l->left);
        Bprint(bout, ", ");
        pexpr(l->right);
        Bprint(bout, " do\n");
        slist(r, d+1);
    }
}
```

Uses OCOMPLEX 59, ODO 59, OELSE 59, OIF 59, OLIST 59, OLOCAL 59, ORET 59, OWHILE 59, bout 69d, pcode() 198b, pexpr() 200, slist() 199a, and tabs-65 197b.

⟨*function* slist 199a⟩≡ (400c)

```
void
slist(Node *n, int d)
{
    if(n == 0)
        return;
    if(n->op == OLIST)
        Bprint(bout, "%.*s{\n", d-1, tabs);
    pcode(n, d);
    if(n->op == OLIST)
        Bprint(bout, "%.*s}\n", d-1, tabs);
}
```

Uses OLIST 59, bout 69d, pcode() 198b, and tabs-65 197b.

A.2.2 Expression dumper: pexpr()

⟨*global* binop 199b⟩≡ (400c)

```
static char *binop[] =
{
    [OMUL] "*",
    [ODIV] "/",
    [OMOD] "%",
    [OADD] "+",
    [OSUB] "-",
    [ORSH] ">>",
    [OLSH] "<<",
    [OLT] "<",
    [OGT] ">",
    [OLEQ] "<=",
```

```
    [OGEQ] ">=",
    [OEQ] "==",
    [ONEQ] "!=",
    [OLAND] "&",
    [OXOR] "^",
    [OLOR] "|",
    [OCAND] "&&",
    [OCOR] "||",
    [OASGN] " = ",
};
```

⟨*function* pexpr 200⟩≡ (400c)

```
/// whatis | pcode -> <>
void
pexpr(Node *n)
{
    Node *r, *l;

    if(n == 0)
        return;

    r = n->right;
    l = n->left;

    switch(n->op) {
    case ONAME:
        Bprint(bout, "%s", n->sym->name);
        break;
    case OCONST:
        switch(n->type) {
        case TINT:
            Bprint(bout, "%lld", n->ival);
            break;
        case TFLOAT:
            Bprint(bout, "%g", n->fval);
            break;
        case TSTRING:
            pstr(n->string);
            break;
        case TLIST:
            break;
        }
        break;
    case OMUL:
    case ODIV:
    case OMOD:
    case OADD:
    case OSUB:
    case ORSH:
    case OLSH:
    case OLT:
    case OGT:
    case OLEQ:
    case OGEQ:
    case OEQ:
    case ONEQ:
    case OLAND:
    case OXOR:
    case OLOR:
    case OCAND:
```

```
case OCOR:
    Bputc(bout, '(');
    pexpr(l);
    Bprint(bout, binop[n->op]);
    pexpr(r);
    Bputc(bout, ')');
    break;
case OASGN:
    pexpr(l);
    Bprint(bout, binop[n->op]);
    pexpr(r);
    break;
case OINDM:
    Bprint(bout, "*");
    pexpr(l);
    break;
case OEDEC:
    Bprint(bout, "--");
    pexpr(l);
    break;
case OEINC:
    Bprint(bout, "++");
    pexpr(l);
    break;
case OPINC:
    pexpr(l);
    Bprint(bout, "++");
    break;
case OPDEC:
    pexpr(l);
    Bprint(bout, "--");
    break;
case ONOT:
    Bprint(bout, "!");
    pexpr(l);
    break;
case OLIST:
    pexpr(l);
    if(r) {
        Bprint(bout, ",");
        pexpr(r);
    }
    break;
case OCALL:
    pexpr(l);
    Bprint(bout, "(");
    pexpr(r);
    Bprint(bout, ")");
    break;
case OCTRUCT:
    Bprint(bout, "{");
    pexpr(l);
    Bprint(bout, "}");
    break;
case OHEAD:
    Bprint(bout, "head ");
    pexpr(l);
    break;
case OTAIL:
    Bprint(bout, "tail ");
```

```
        pexpr(l);
        break;
    case OAPPEND:
        Bprint(bout, "append ");
        pexpr(l);
        Bprint(bout, ",");
        pexpr(r);
        break;
    case ODELETE:
        Bprint(bout, "delete ");
        pexpr(l);
        Bprint(bout, ",");
        pexpr(r);
        break;
    case ORET:
        Bprint(bout, "return ");
        pexpr(l);
        break;
    case OINDEX:
        pexpr(l);
        Bprint(bout, "[");
        pexpr(r);
        Bprint(bout, "]");
        break;
    case OINDC:
        Bprint(bout, "@");
        pexpr(l);
        break;
    case ODOT:
        pexpr(l);
        Bprint(bout, ".%s", n->sym->name);
        break;
    case OFRAME:
        Bprint(bout, "%s:%s", n->sym->name, l->sym->name);
        break;
    case OCAST:
        Bprint(bout, "(%s)", n->sym->name);
        pexpr(l);
        break;
    case OFMT:
        pexpr(l);
        Bprint(bout, "\\%c", (int)r->ival);
        break;
    case OEVAL:
        Bprint(bout, "eval ");
        pexpr(l);
        break;
    case OWHAT:
        Bprint(bout, "whatis");
        if(n->sym)
            Bprint(bout, " %s", n->sym->name);
        break;
    }
}
```

Uses OADD 59, OAPPEND 59, OASGN 59, OCALL 59, OCAND 59, OCAST 59, OCONST 59, OCOR 59, OCTRUCT 59, ODELETE 59, ODIV 59, ODOT 59, OEDEC 59, OEINC 59, OEQ 59, OEVAL 59, OFMT 59, OFRAME 59, OGEQ 59, OGT 59, OHEAD 59, OINDC 59, OINDEX 59, OINDM 59, OLAND 59, OLEQ 59, OLIST 59, OLOR 59, OLSH 59, OLT 59, OMOD 59, OMUL 59, ONAME 59, ONEQ 59, ONOT 59, OPDEC 59, OPINC 59, ORET 59, ORSH 59, OSUB 59, OTAIL 59, OWHAT 59, OXOR 59, TFLOAT 60a, TINT 60a, TLIST 60a, TSTRING 60a, binop-64 199b, bout 69d, pexpr() 200, and pstr() 203a.

⟨*function* pstr 203a⟩≡ (400c)

```
void
pstr(String *s)
{
    int i, c;

    Bputc(bout, '"');
    for(i = 0; i < s->len; i++) {
        c = s->string[i];
        switch(c) {
        case '\0':
            c = '0';
            break;
        case '\n':
            c = 'n';
            break;
        case '\r':
            c = 'r';
            break;
        case '\t':
            c = 't';
            break;
        case '\b':
            c = 'b';
            break;
        case '\f':
            c = 'f';
            break;
        case '\a':
            c = 'a';
            break;
        case '\v':
            c = 'v';
            break;
        case '\\':
            c = '\\';
            break;
        case '"':
            c = '"';
            break;
        default:
            Bputc(bout, c);
            continue;
        }
        Bputc(bout, '\\');
        Bputc(bout, c);
    }
    Bputc(bout, '"');
}
```

Uses bout 69d.

A.2.3 Type dumper

⟨*global* typenames 203b⟩≡ (400c)

```
char *typenames[] =
{
    [TINT] "integer",
    [TFLOAT] "float",
    [TSTRING] "string",
```

```
    [TLIST] "list",
    [TCODE] "code",
};
```

A.3 Format dumpers

⟨main()*(acid) format initializations* 204a⟩≡ (69e) 383a▷
```
fmtinstall('L', Lfmt);
```

⟨*function* Lfmt 204b⟩≡ (381b)
```
int
Lfmt(Fmt *f)
{
    int i;
    char buf[1024];
    IOstack *e;

    e = lexio;
    if(e) {
        i = snprint(buf, sizeof(buf), "%s:%d", e->name, line);
        while(e->prev) {
            e = e->prev;
            if(initialising && e->prev == 0)
                break;
            i += snprint(buf+i, sizeof(buf)-i, " [%s:%d]", e->name, e->line);
        }
    } else
        snprint(buf, sizeof(buf),  "no file:0");
    fmtstrcpy(f, buf);
    return 0;
}
```
Uses initialising 71h and lexio 100b.

Appendix B

Error Management

B.1 Quitting: `die()`

⟨*function die (acid/main.c)* 205a⟩≡ (383c)

```
/// main -> <>
void
die(void)
{
    Lsym *s;
    List *f;

    Bprint(bout, "\n");

    s = look("proclist");
    if(s && s->v->type == TLIST) {
        for(f = s->v->l; f; f = f->next)
            Bprint(bout, "echo kill > /proc/%d/ctl\n", (int)f->ival);
    }
    exits(nil);
}
```

Uses `TLIST` 60a, `bout` 69d, and `look()` 66d.

B.2 Unrecoverable errors: `fatal()`

⟨*function fatal (acid/main.c)* 205b⟩≡ (383c)

```
/// -> <>
void
fatal(char *fmt, ...)
{
    char buf[128];
    va_list arg;

    va_start(arg, fmt);
    vseprint(buf, buf+sizeof(buf), fmt, arg);
    va_end(arg);
    fprint(STDERR, "%s: %L (fatal problem) %s\n", argv0, buf);
    exits(buf);
}
```

B.3 Parser errors: `yyerror()`

⟨*function* `yyerror` 205c⟩≡ (383c)

```
void
yyerror(char *fmt, ...)
{
    char buf[128];
    va_list arg;

    if(strcmp(fmt, "syntax error") == 0) {
        yyerror("syntax error, near symbol '%s'", symbol);
        return;
    }
    va_start(arg, fmt);
    vseprint(buf, buf+sizeof(buf), fmt, arg);
    va_end(arg);
    print("%L: %s\n", buf);
}
```

Uses `symbol` 104a and `yyerror()` 205c.

B.4 `error()` and exception management

`error()`[206b] is `acid`'s exception mechanism: it prints the error message, unwinds any stacked I/O channels, resets the `interactive` flag, and `longjmp`s back to the REPL's `setjmp` point. This is the C equivalent of throwing an exception—any error during parsing or execution is caught at the top level, and the user gets back to the `acid:` prompt.

⟨*global* `err` 206a⟩≡ (381a)

```
jmp_buf err;
```

⟨*function error (acid/exec.c)* 206b⟩≡ (391)

```
/// ??? -> <>
void
error(char *fmt, ...)
{
    int i;
    char buf[2048];
    va_list arg;

    ⟨error() (acid) unstack io channels 207d⟩

    ret = 0;
    ⟨error() reset gotint 208e⟩

    Bflush(bout);
    ⟨error() (acid) if silent 78c⟩
    else {
        va_start(arg, fmt);
        vseprint(buf, buf+sizeof(buf), fmt, arg);
        va_end(arg);
        fprint(STDERR, "%L: (error) %s\n", buf);
    }
    while(popio())
        ;
    interactive = true;
    longjmp(err, 1);
}
```

Uses `bout` 69d, `err` 206a, `interactive` 70a, `popio()` 101e, and `ret` 134b.

`unwind()`[207a] restores all symbol table entries to their global values by popping the entire `pop` chain for every symbol. This cleans up local variable bindings left behind when an error occurs inside a function call.

⟨*function* `unwind` 207a⟩≡ (391)
```
void
unwind(void)
{
    int i;
    Lsym *s;
    Value *v;

    for(i = 0; i < Hashsize; i++) {
        for(s = hash[i]; s; s = s->hash) {

            while(s->v->pop) {
                v = s->v->pop;
                free(s->v);
                s->v = v;
            }
        }
    }
}
```
Uses `Hashsize` 66b.

⟨*global* `io` 207b⟩≡ (381a)
```
Biobuf* io[32];
```

⟨*global* `iop` 207c⟩≡ (381a)
```
int iop;
```

⟨`error()` *(acid) unstack io channels* 207d⟩≡ (206b)
```
/* Unstack io channels */
if(iop != 0) {
    for(i = 1; i < iop; i++)
        Bterm(io[i]);
    bout = io[0];
    iop = 0;
}
```
Uses `bout` 69d, `io` 207b, and `iop` 207c.

B.5 Interrupt management

When the user presses Ctrl-C, the kernel delivers an interrupt note to `acid`. The `catcher()`[208a] signal handler sets `gotint` to true and continues. The `gotint` flag is checked at three points: in `lexc()`[102a] (while reading input), in `execute()`[120a] (while running code), and in `yylex()`[103a] (at EOF). This ensures that long-running `acid` scripts can be interrupted without corrupting internal state.

⟨*global* `gotint` 207e⟩≡ (381a)
```
bool gotint;
```

⟨`main()` *(acid) notify setup* 207f⟩≡ (72a)
```
notify(catcher);
```

⟨*function* `catcher` 208a⟩≡ (383c)

```
void
catcher(void *junk, char *s)
{
    USED(junk);

    if(strstr(s, "interrupt")) {
        gotint = true;
        noted(NCONT);
    }
    noted(NDFLT);
}
```

Uses `gotint` 207e.

⟨`lexc()` *if* `gotint` 208b⟩≡ (102a)

```
if(gotint)
    error("interrupt");
```

Uses `gotint` 207e.

⟨`execute()` *if* `gotint` 208c⟩≡ (120a)

```
if(gotint)
    error("interrupted");
```

Uses `gotint` 207e.

⟨`yylex()` *when* `Eof`, *if* `gotint` 208d⟩≡ (103c)

```
if(gotint) {
    gotint = false;
    stacked = 0;
    Bprint(bout, "\nacid: ");
    goto loop;
}
```

Uses `bout` 69d, `gotint` 207e, and `stacked` 71a.

⟨`error()` *reset* `gotint` 208e⟩≡ (206b)

```
gotint = false;
```

Uses `gotint` 207e.

Appendix C

Utilities

⟨*function* `isnumeric` 209⟩≡ (383c)

```
bool
isnumeric(char *s)
{
    while(*s) {
        if(*s < '0' || *s > '9')
            return false;
        s++;
    }
    return true;
}
```

Appendix D

ARM Disassembler

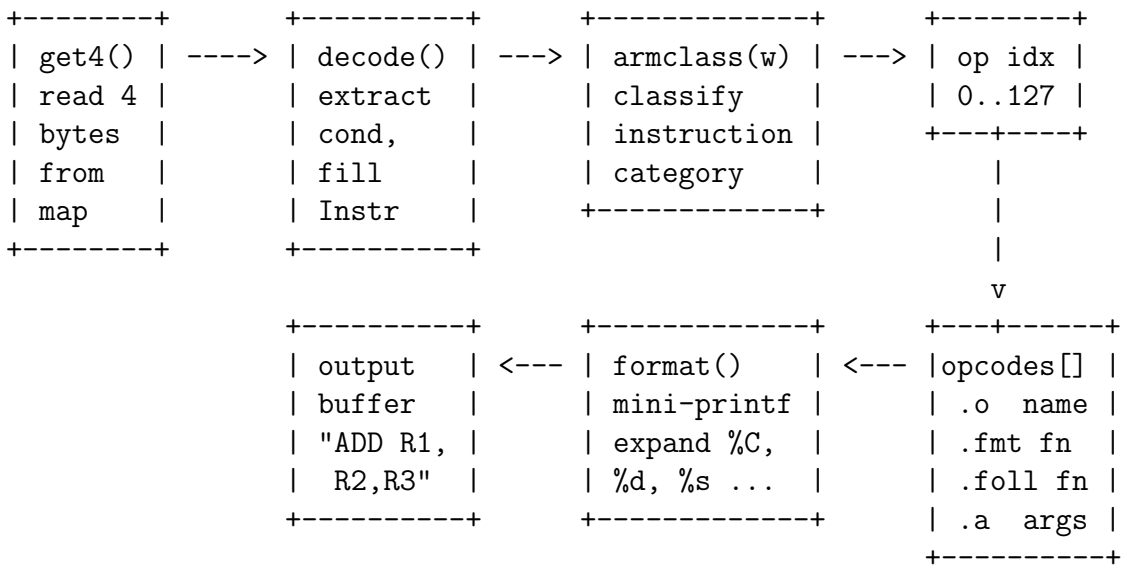
This appendix presents the ARM disassembler in `libmach/5db.c`. It is the reverse of the ASSEMBLER book [Pad15a]: where `5a` turns assembly mnemonics into 32-bit machine words, `5db.c` turns machine words back into Plan 9 assembly syntax. The disassembler serves two roles inside `acid`:

1. **Display**: the `asm()` and `das()` commands print disassembled instructions for the user to read.
2. **Follow-set**: the `armfoll()`[233] function computes where execution *will go next* from a given PC, which is how `acid` implements single-stepping (set a breakpoint at the follow address, resume, wait).

D.1 Overview

D.1.1 Decoding pipeline

The pipeline from a raw 32-bit word to a printed instruction is:



`armclass()`[219] is the heart: a 150-line switch statement that decodes the top 3 bits of the instruction word (bits 25–27, the "major opcode") and then refines with sub-fields to produce an index into the `opcodes`[214b] table. Each table entry names the Plan 9 mnemonic, a format handler that extracts operand fields into the `Instr`[314c] struct, and an optional follow-set function for single-stepping.

D.1.2 `Machdata` disassembler ARM methods

The four entry points below are wired into the `armmach`[52a] vtable and called through `machdata->` pointers:

- `arminst()`[211a] / `printins()`[211b]: disassemble one instruction into Plan 9 assembly syntax. This is what `acid`'s `asm()` command calls.
- `armdas()`[211d]: the "hex dump" disassembler—produces just the raw hex bytes, used by `acid`'s hex-format display.
- `arminstlen()`[211c]: returns the instruction length (always 4 for ARM A32).
- `armfoll()`[233]: the follow-set entry point. It decodes the instruction, then delegates to the `foll` function pointer from the `opcodes`[214b] entry (or returns `pc+4` if the instruction cannot change control flow). This is the function that `acid`'s single-stepper calls to decide where to place the temporary breakpoint.

⟨*function* `arminst` *(arm)* 211a⟩≡ (309b)
```
/// bprint -> patom (when 'i' format) -> armmach.das -> <>
static int
arminst(Map *map, uvlong pc, char modifier, char *buf, int n)
{
    USED(modifier);
    return printins(map, pc, buf, n);
}
```

⟨*function* `printins` *(arm)* 211b⟩≡ (309b)
```
static int
printins(Map *map, uvlong pc, char *buf, int n)
{
    Instr i;

    i.curr = buf;
    i.end = buf+n-1;
    if(decode(map, pc, &i) < 0)
        return -1;

    (*opcodes[i.op].fmt)(&opcodes[i.op], &i);
    return 4;
}
```

⟨*function* `arminstlen` *(arm)* 211c⟩≡ (309b)
```
/// fmtsize -> armmach.instsize
static int
arminstlen(Map *map, uvlong pc)
{
    Instr i;

    if(decode(map, pc, &i) < 0)
        return -1;
    return 4;
}
```

⟨*function* `armdas` *(arm)* 211d⟩≡ (309b)
```
/// ?? -> armmach.hexinst -> <>
static int
armdas(Map *map, uvlong pc, char *buf, int n)
{
    Instr i;
```

```
    i.curr = buf;
    i.end = buf+n;
    if(decode(map, pc, &i) < 0)
        return -1;
    if(i.end-i.curr > 8)
        i.curr = _hexify(buf, i.w, 7);
    *i.curr = 0;
    return 4;
}
```

Uses _hexify() 212a.

⟨*function* _hexify 212a⟩≡ (362c)

```
char *
_hexify(char *buf, ulong p, int zeros)
{
    ulong d;

    d = p/16;
    if(d)
        buf = _hexify(buf, d, zeros-1);
    else
        while(zeros--)
            *buf++ = '0';
    *buf++ = "0123456789abcdef"[p&0x0f];
    return buf;
}
```

Uses _hexify() 212a.

D.1.3 Bit manipulation

Four macros for bit manipulation on 32-bit instruction words. `BITS(a,b)` produces a bitmask from bit `a` to bit `b` inclusive. `LSR`, `ASR`, and `ROR` are logical shift right, arithmetic shift right, and rotate right—the same operations as the ARM barrel shifter, used here to decode immediate operands with rotation (the ARM "flexible second operand").

⟨*function* BITS*(arm)* 212b⟩≡ (309b)

```
#define BITS(a, b)      ((1<<(b+1))-(1<<a))
```

⟨*function* LSR*(arm)* 212c⟩≡ (309b)

```
#define LSR(v, s)       ((ulong)(v) >> (s))
```

⟨*function* ASR*(arm)* 212d⟩≡ (309b)

```
#define ASR(v, s)       ((long)(v) >> (s))
```

⟨*function* ROR*(arm)* 212e⟩≡ (309b)

```
#define ROR(v, s)       (LSR((v), (s)) | (((v) & ((1 << (s))-1)) << (32 - (s))))
```

⟨*function* nbits*(arm)* 212f⟩≡ (309b)

```
static int
nbits(ulong v)
{
    int n = 0;
    int i;

    for(i=0; i < 32 ; i++) {
        if(v & 1) ++n;
        v >>= 1;
    }

    return n;
}
```

D.2 Data structures

The ARM disassembler is built around three data structures: the `Instr` struct, which holds the decoded fields of one instruction; the `Opcode` table entry, which describes the class handler and format template for every opcode; and the shared `Map` abstraction from `libmach/access.c`, used to read the raw instruction bytes from either an executable file or a live process.

D.2.1 `Instr`

The `Instr` struct holds the decoded state of one instruction. After `decode()`[217] reads the raw 32-bit word `w` from the map, it extracts the condition code (`cond`, bits 28–31) and calls `armclass()`[219] to set `op` (the index into `opcodes`[214b]). The format handler then fills `rd`, `rn`, `rs`, `imm`, and `store` from the remaining bit fields—these correspond directly to the fields in the ARM instruction encoding:

```
31..28  27..25  24..21  20  19..16  15..12  11..0
+------+-------+-------+---+-------+-------+--------+
| cond | major | minor | S |  Rn   |  Rd   | operand|
+------+-------+-------+---+-------+-------+--------+
   .cond   (armclass)  .store  .rn     .rd    .rs/.imm
```

⟨*struct* `Instr` *(arm)* 213⟩≡ (309b)

```
struct  Instr
{
    Map *map;
    ulong       w;
    uvlong      addr;
    uchar       op;                     /* super opcode */

    uchar       cond;                   /* bits 28-31 */
    uchar       store;                  /* bit 20 */

    uchar       rd;                     /* bits 12-15 */
    uchar       rn;                     /* bits 16-19 */
    uchar       rs;                     /* bits 0-11 (shifter operand) */

    long        imm;                    /* rotated imm */
    char*       curr;                   /* fill point in buffer */
    char*       end;                    /* end of buffer */
    char*       err;                    /* error message */
};
```

D.2.2 `Opcode`

The `Opcode` struct is the schema for one row in the `opcodes`[214b] table. Each instruction class has one entry:

- `o`: the Plan 9 mnemonic template, e.g., `"ADD\%C\%S"`. The `%C` and `%S` are expanded by `format()`[222] into the condition suffix and the S-bit flag.
- `fmt`: the format handler that extracts operand fields from the raw word into the `Instr` struct, then calls `format()` to print the result.
- `foll`: the follow-set function. If non-nil, it computes the next PC when `acid` single-steps through this instruction. Only instructions that can change control flow (branches, moves into PC) have a non-nil `foll`.
- `a`: the operand template, e.g., `"R\%s,R\%n,R\%d"`. `format()` expands `%s`, `%n`, `%d` to the register numbers extracted by the format handler.

⟨*struct* `Opcode` *(arm)* 214a⟩≡ (309b)

```
struct Opcode
{
    // opcode template
    char*       o;
    // operand extractor
    void        (*fmt)(Opcode*, Instr*);
    // option<func> non-nil when branch or MOVE PC instruction
    uvlong      (*foll)(Map*, Rgetter, Instr*, uvlong);
    // operands template
    char*       a;
};
```

D.2.3 `opcodes` table

The `opcodes`[214b] table has 128 entries, laid out so that `armclass()`[219] can compute the index with simple arithmetic on bit fields. The layout mirrors the ARM instruction encoding:

Index range	Group	Handler
0..15	data proc reg, no shift	armdps
16..31	data proc reg, imm shift	armdps
32..47	data proc reg, reg shift	armdps
48..63	data proc immediate	armdpi
64..67	multiply, swap	armdpi
68..71	halfword load/store	armhwby
72..75	word/byte load/store imm	armsdti
76..79	word/byte load/store reg	armsdts
80..81	block data transfer	armbdt
82..83	branch / branch-link	armb
84..87	coprocessor	armco
88..91	long multiply	armdpi
92	unknown	armunk
93..98	ARMv7 (LDREX, barriers)	armdpi/armunk
99	RFE	armbdt
100..123	VFP instructions	armdps/armvstdi
124..127	BKPT, BX, BLX	armbpt/armdps

For example, entry 82 is `{ "B%C", armb, armfbranch, "%b" }`: the mnemonic is `"B"` with an optional condition suffix, `armb()`[232a] extracts the 24-bit offset, `armfbranch()`[235b] computes the follow address, and `%b` formats the target as a symbol name.

⟨*global* `opcodes` *(arm)* 214b⟩≡ (309b)

```
static Opcode opcodes[] =
{
    "AND%C%S",  armdps, 0,      "R%s,R%n,R%d",
    "EOR%C%S",  armdps, 0,      "R%s,R%n,R%d",
    "SUB%C%S",  armdps, 0,      "R%s,R%n,R%d",
    "RSB%C%S",  armdps, 0,      "R%s,R%n,R%d",
    "ADD%C%S",  armdps, armfadd,        "R%s,R%n,R%d",
    "ADC%C%S",  armdps, 0,      "R%s,R%n,R%d",
    "SBC%C%S",  armdps, 0,      "R%s,R%n,R%d",
    "RSC%C%S",  armdps, 0,      "R%s,R%n,R%d",
    "TST%C%S",  armdps, 0,      "R%s,R%n",
    "TEQ%C%S",  armdps, 0,      "R%s,R%n",
```

```
    "CMP%C%S",  armdps, 0,       "R%s,R%n",
    "CMN%C%S",  armdps, 0,       "R%s,R%n",
    "ORR%C%S",  armdps, 0,       "R%s,R%n,R%d",
    "MOVW%C%S", armdps, armfmov,         "R%s,R%d",
    "BIC%C%S",  armdps, 0,       "R%s,R%n,R%d",
    "MVN%C%S",  armdps, 0,       "R%s,R%d",

/* 16 */
    "AND%C%S",  armdps, 0,       "(R%s%h%m),R%n,R%d",
    "EOR%C%S",  armdps, 0,       "(R%s%h%m),R%n,R%d",
    "SUB%C%S",  armdps, 0,       "(R%s%h%m),R%n,R%d",
    "RSB%C%S",  armdps, 0,       "(R%s%h%m),R%n,R%d",
    "ADD%C%S",  armdps, armfadd,         "(R%s%h%m),R%n,R%d",
    "ADC%C%S",  armdps, 0,       "(R%s%h%m),R%n,R%d",
    "SBC%C%S",  armdps, 0,       "(R%s%h%m),R%n,R%d",
    "RSC%C%S",  armdps, 0,       "(R%s%h%m),R%n,R%d",
    "TST%C%S",  armdps, 0,       "(R%s%h%m),R%n",
    "TEQ%C%S",  armdps, 0,       "(R%s%h%m),R%n",
    "CMP%C%S",  armdps, 0,       "(R%s%h%m),R%n",
    "CMN%C%S",  armdps, 0,       "(R%s%h%m),R%n",
    "ORR%C%S",  armdps, 0,       "(R%s%h%m),R%n,R%d",
    "MOVW%C%S", armdps, armfmov,         "(R%s%h%m),R%d",
    "BIC%C%S",  armdps, 0,       "(R%s%h%m),R%n,R%d",
    "MVN%C%S",  armdps, 0,       "(R%s%h%m),R%d",

/* 32 */
    "AND%C%S",  armdps, 0,       "(R%s%hR%M),R%n,R%d",
    "EOR%C%S",  armdps, 0,       "(R%s%hR%M),R%n,R%d",
    "SUB%C%S",  armdps, 0,       "(R%s%hR%M),R%n,R%d",
    "RSB%C%S",  armdps, 0,       "(R%s%hR%M),R%n,R%d",
    "ADD%C%S",  armdps, armfadd,         "(R%s%hR%M),R%n,R%d",
    "ADC%C%S",  armdps, 0,       "(R%s%hR%M),R%n,R%d",
    "SBC%C%S",  armdps, 0,       "(R%s%hR%M),R%n,R%d",
    "RSC%C%S",  armdps, 0,       "(R%s%hR%M),R%n,R%d",
    "TST%C%S",  armdps, 0,       "(R%s%hR%M),R%n",
    "TEQ%C%S",  armdps, 0,       "(R%s%hR%M),R%n",
    "CMP%C%S",  armdps, 0,       "(R%s%hR%M),R%n",
    "CMN%C%S",  armdps, 0,       "(R%s%hR%M),R%n",
    "ORR%C%S",  armdps, 0,       "(R%s%hR%M),R%n,R%d",
    "MOVW%C%S", armdps, armfmov,         "(R%s%hR%M),R%d",
    "BIC%C%S",  armdps, 0,       "(R%s%hR%M),R%n,R%d",
    "MVN%C%S",  armdps, 0,       "(R%s%hR%M),R%d",

/* 48 */
    "AND%C%S",  armdpi, 0,       "$#%i,R%n,R%d",
    "EOR%C%S",  armdpi, 0,       "$#%i,R%n,R%d",
    "SUB%C%S",  armdpi, 0,       "$#%i,R%n,R%d",
    "RSB%C%S",  armdpi, 0,       "$#%i,R%n,R%d",
    "ADD%C%S",  armdpi, armfadd,         "$#%i,R%n,R%d",
    "ADC%C%S",  armdpi, 0,       "$#%i,R%n,R%d",
    "SBC%C%S",  armdpi, 0,       "$#%i,R%n,R%d",
    "RSC%C%S",  armdpi, 0,       "$#%i,R%n,R%d",
    "TST%C%S",  armdpi, 0,       "$#%i,R%n",
    "TEQ%C%S",  armdpi, 0,       "$#%i,R%n",
    "CMP%C%S",  armdpi, 0,       "$#%i,R%n",
    "CMN%C%S",  armdpi, 0,       "$#%i,R%n",
    "ORR%C%S",  armdpi, 0,       "$#%i,R%n,R%d",
    "MOVW%C%S", armdpi, armfmov,         "$#%i,R%d",
    "BIC%C%S",  armdpi, 0,       "$#%i,R%n,R%d",
    "MVN%C%S",  armdpi, 0,       "$#%i,R%d",
```

```
/* 48+16 */
    "MUL%C%S",  armdpi, 0,      "R%M,R%s,R%n",
    "MULA%C%S", armdpi, 0,      "R%M,R%s,R%n,R%d",
    "SWPW",             armdpi, 0,      "R%s,(R%n),R%d",
    "SWPB",             armdpi, 0,      "R%s,(R%n),R%d",

/* 48+16+4 */
    "MOV%u%C%p",        armhwby, 0,     "R%d,(R%n%UR%M)",
    "MOV%u%C%p",        armhwby, 0,     "R%d,%I",
    "MOV%u%C%p",        armhwby, armfmov,       "(R%n%UR%M),R%d",
    "MOV%u%C%p",        armhwby, armfmov,       "%I,R%d",

/* 48+24 */
    "MOVW%C%p", armsdti, 0,     "R%d,%I",
    "MOVB%C%p", armsdti, 0,     "R%d,%I",
    "MOVW%C%p", armsdti, armfmov,       "%I,R%d",
    "MOVBU%C%p",        armsdti, armfmov,       "%I,R%d",

    "MOVW%C%p", armsdts, 0,     "R%d,(R%s%h%m)(R%n)",
    "MOVB%C%p", armsdts, 0,     "R%d,(R%s%h%m)(R%n)",
    "MOVW%C%p", armsdts, armfmov,       "(R%s%h%m)(R%n),R%d",
    "MOVBU%C%p",        armsdts, armfmov,       "(R%s%h%m)(R%n),R%d",

    "MOVM%C%P%a",       armbdt, armfmovm,               "[%r],(R%n)",
    "MOVM%C%P%a",       armbdt, armfmovm,               "(R%n),[%r]",

    "B%C",              armb, armfbranch,               "%b",
    "BL%C",             armb, armfbranch,               "%b",

    "CDP%C",    armco, 0,               "",
    "CDP%C",    armco, 0,               "",
    "MCR%C",    armco, 0,               "",
    "MRC%C",    armco, 0,               "",

/* 48+24+4+4+2+2+4 */
    "MULLU%C%S",        armdpi, 0,      "R%M,R%s,(R%n,R%d)",
    "MULALU%C%S",       armdpi, 0,      "R%M,R%s,(R%n,R%d)",
    "MULL%C%S", armdpi, 0,      "R%M,R%s,(R%n,R%d)",
    "MULAL%C%S",        armdpi, 0,      "R%M,R%s,(R%n,R%d)",

/* 48+24+4+4+2+2+4+4 = 92 */
    "UNK",              armunk, 0,      "",

    /* new v7 arch instructions */
/* 93 */
    "LDREX",    armdpi, 0,      "(R%n),R%d",
    "STREX",    armdpi, 0,      "R%s,(R%n),R%d",
    "CLREX",    armunk, 0,      "",

/* 96 */
    "DSB",              armunk, 0,      "",
    "DMB",              armunk, 0,      "",
    "ISB",              armunk, 0,      "",

/* 99 */
    "RFEV7%P%a",        armbdt, 0,      "(R%n)",

/* 100 */
    "MLA%f%C",  armdps, 0,      "F%s,F%n,F%d",
```

```
    "MLS%f%C",  armdps, 0,      "F%s,F%n,F%d",
    "NMLS%f%C", armdps, 0,      "F%s,F%n,F%d",
    "NMLA%f%C", armdps, 0,      "F%s,F%n,F%d",
    "MUL%f%C",  armdps, 0,      "F%s,F%n,F%d",
    "NMUL%f%C", armdps, 0,      "F%s,F%n,F%d",
    "ADD%f%C",  armdps, 0,      "F%s,F%n,F%d",
    "SUB%f%C",  armdps, 0,      "F%s,F%n,F%d",
    "DIV%f%C",  armdps, 0,      "F%s,F%n,F%d",

/* 109 */
    "MOV%f%C",  armdps, 0,      "F%s,F%d",
    "ABS%f%C",  armdps, 0,      "F%s,F%d",
    "NEG%f%C",  armdps, 0,      "F%s,F%d",
    "SQRT%f%C", armdps, 0,      "F%s,F%d",
    "CMP%f%C",  armdps, 0,      "F%s,F%d",
    "CMPE%f%C", armdps, 0,      "F%s,F%d",
    "CMP%f%C",  armdps, 0,      "$0.0,F%d",
    "CMPE%f%C", armdps, 0,      "$0.0,F%d",

/* 117 */
    "MOV%F%R%C",        armdps, 0,      "F%s,F%d",

/* 118 */
    "MOVW%C",   armdps, 0,      "R%d,F%n",
    "MOVW%C",   armdps, 0,      "F%n,R%d",
    "MOVW%C",   armdps, 0,      "R%d,%x",
    "MOVW%C",   armdps, 0,      "%x,R%d",

/* 122 */
    "MOV%f%C",  armvstdi,       0,      "F%d,%I",
    "MOV%f%C",  armvstdi,       0,      "%I,F%d",

/* 124 */
    "BKPT%C",   armbpt, 0,              "$#%i",
    "BX%C",             armdps, armfbx, "(R%s)",
    "BXJ%C",    armdps, armfbx, "(R%s)",
    "BLX%C",    armdps, armfbx, "(R%s)",
};
```

Uses `armb()` 232a, `armbdt()` 231c, `armbpt()` 232b, `armco()` 232c, `armdpi()` 229b, `armdps()` 229a, `armfadd()` 234a, `armfbranch()` 235b, `armfbx()` 234b, `armfmov()` 235c, `armfmovm()` 234c, `armhwby()` 231a, `armsdti()` 230a, `armsdts()` 231b, `armunk()` 231f, and `armvstdi()` 230b.

D.3 Decoding pipeline

With the `Instr` struct and the `opcodes` table now in hand, we can look at the decoding pipeline that turns a 4-byte word at some `pc` into a filled `Instr` ready for printing. The pipeline is three layers deep: `decode()`[217] reads the raw word from the `Map`, `armclass()`[219] computes an opcode index by walking a decision table on the top bits, and the format handler pulled from `opcodes[i.op]` fills in the remaining per-class operand fields.

D.3.1 `decode()`

`decode()`[217] is the thin wrapper that reads a 4-byte instruction word from the `Map` at address `pc`, extracts the condition code from bits 31–28, and calls `armclass()`[219] to set the opcode index. After `decode` returns, the `Instr` is ready for the format handler.

⟨*function* `decode` *(arm)* 217⟩≡ (309b)

```
static int
decode(Map *map, uvlong pc, Instr *i)
{
    ulong w;

    if(get4(map, pc, &w) < 0) {
        werrstr("can't read instruction: %r");
        return -1;
    }
    i->w = w;
    i->addr = pc;
    i->cond = (w >> 28) & 0xF;
    i->op = armclass(w);
    i->map = map;
    return 1;
}
```

Uses armclass() 219 and get4() 350b.

D.3.2 armclass()

armclass()[219] is the core of the disassembler: given a 32-bit instruction word, it returns an index into the opcodes[214b] table. The first dispatch is on bits 27–25 (the "major opcode"), which partitions the ARM instruction set into 8 groups:

```
bits 27-25    group             opcodes[] range
----------    -----             ---------------
   000        data proc r,r,r   0..47 (3 variants x 16 ALU ops)
   001        data proc imm     48..63
   010        load/store imm    72..75
   011        load/store reg    76..79
   100        block transfer    80..81
   101        branch / BL       82..83
   110        VFP load/store    122..123
   111        coprocessor       84..91
```

Within each group, sub-fields further refine the index. For example, data processing instructions (groups 0–1) use bits 24–21 to select one of 16 ALU operations (AND, EOR, SUB, RSB, ADD, ..., MVN), each appearing three times in the table: once for the plain register form (indices 0–15), once with a constant shift (16–31), and once with a register shift (32–47). Let us trace a concrete example. The instruction ADD R1, R2, R3 encodes as 0xE0821003:

```
0xE0821003 = 1110 000 0100 0 0010 0001 00000000 0011
             cond maj  op  S  Rn   Rd           Rm
              AL  000 0100 0  R2   R1           R3

armclass(0xE0821003):
  bits 27-25 = 000 -> case 0 (data proc r,r,r)
  bits 7-4   = 0000 -> not 0x9 (not mul/swp)
                    -> not 0x9/0xB (not halfword)
  bits 24-21 = 0100 -> op = 4 (ADD)
  bit 4 = 0, bits 7-11 = 0, bit 5 = 0
    -> no shift, stays in base range
  return 4
```

```
opcodes[4] = { "ADD%C%S", armdps, armfadd, "R%s,R%n,R%d" }
-> armdps extracts Rn=2, Rd=1, Rs=3
-> format prints: ADD  R3,R2,R1
```

Note the Plan 9 syntax is destination-last (opposite of standard ARM syntax).

⟨*function* armclass*(arm)* 219⟩≡ (309b)

```
int
armclass(long w)
{
    int op, done, cp;

    op = (w >> 25) & 0x7;
    switch(op) {
    case 0:     /* data processing r,r,r */
        if((w & 0x0ff00080) == 0x01200000) {
            op = (w >> 4) & 0x7;
            if(op == 7)
                op = 124;       /* bkpt */
            else if (op > 0 && op < 4)
                op += 124;      /* bx, blx */
            else
                op = 92;        /* unk */
            break;
        }
        op = ((w >> 4) & 0xf);
        if(op == 0x9) {
            op = 48+16;         /* mul, swp or *rex */
            if((w & 0x0ff00fff) == 0x01900f9f) {
                op = 93;        /* ldrex */
                break;
            }
            if((w & 0x0ff00ff0) == 0x01800f90) {
                op = 94;        /* strex */
                break;
            }
            if(w & (1<<24)) {
                op += 2;
                if(w & (1<<22))
                    op++;       /* swpb */
                break;
            }
            if(w & (1<<23)) {   /* mullu */
                op = (48+24+4+4+2+2+4);
                if(w & (1<<22)) /* mull */
                    op += 2;
            }
            if(w & (1<<21))
                op++;           /* mla */
            break;
        }
        if((op & 0x9) == 0x9)           /* ld/st byte/half s/u */
        {
            op = (48+16+4) + ((w >> 22) & 0x1) + ((w >> 19) & 0x2);
            break;
        }
        op = (w >> 21) & 0xf;
        if(w & (1<<4))
            op += 32;
```

```
    else
    if((w & (31<<7)) || (w & (1<<5)))
        op += 16;
    break;
case 1:     /* data processing i,r,r */
    op = (48) + ((w >> 21) & 0xf);
    break;
case 2:     /* load/store byte/word i(r) */
    if ((w & 0xffffff8f) == 0xf57ff00f) {   /* barriers, clrex */
        done = 1;
        switch ((w >> 4) & 7) {
        case 1:
            op = 95;        /* clrex */
            break;
        case 4:
            op = 96;        /* dsb */
            break;
        case 5:
            op = 97;        /* dmb */
            break;
        case 6:
            op = 98;        /* isb */
            break;
        default:
            done = 0;
            break;
        }
        if (done)
            break;
    }
    op = (48+24) + ((w >> 22) & 0x1) + ((w >> 19) & 0x2);
    break;
case 3:     /* load/store byte/word (r)(r) */
    op = (48+24+4) + ((w >> 22) & 0x1) + ((w >> 19) & 0x2);
    break;
case 4:     /* block data transfer (r)(r) */
    if ((w & 0xfe50ffff) == 0xf8100a00) {   /* v7 RFE */
        op = 99;
        break;
    }
    op = (48+24+4+4) + ((w >> 20) & 0x1);
    break;
case 5:     /* branch / branch link */
    op = (48+24+4+4+2) + ((w >> 24) & 0x1);
    break;
case 7:     /* coprocessor crap */
    cp = (w >> 8) & 0xF;
    if(cp == 10 || cp == 11){       /* vfp */
        if((w >> 4) & 0x1){
            /* vfp register transfer */
            switch((w >> 21) & 0x7){
            case 0:
                op = 118 + ((w >> 20) & 0x1);
                break;
            case 7:
                op = 118+2 + ((w >> 20) & 0x1);
                break;
            default:
                op = (48+24+4+4+2+2+4+4);
                break;
```

```
                }
                break;
            }
            /* vfp data processing */
            if(((w >> 23) & 0x1) == 0){
                op = 100 + ((w >> 19) & 0x6) + ((w >> 6) & 0x1);
                break;
            }
            switch(((w >> 19) & 0x6) + ((w >> 6) & 0x1)){
            case 0:
                op = 108;
                break;
            case 7:
                if(((w >> 19) & 0x1) == 0){
                    if(((w >> 17) & 0x1) == 0)
                        op = 109 + ((w >> 16) & 0x4) +
                            ((w >> 15) & 0x2) +
                            ((w >> 7) & 0x1);
                    else if(((w >> 16) & 0x7) == 0x7)
                        op = 117;
                }else
                    switch((w >> 16) & 0x7){
                    case 0:
                    case 4:
                    case 5:
                        op = 117;
                        break;
                    }
                break;
            }
            if(op == 7)
                op = (48+24+4+4+2+2+4+4);
            break;
        }
        op = (48+24+4+4+2+2) + ((w >> 3) & 0x2) + ((w >> 20) & 0x1);
        break;
    case 6:     /* vfp load / store */
        if(((w >> 21) &0x9) == 0x8){
            op = 122 + ((w >> 20) & 0x1);
            break;
        }
        /* fall through */
    default:
        op = (48+24+4+4+2+2+4+4);
        break;
    }
    return op;
}
```

Uses `done()` 255b.

D.3.3 `format()` mini-printf

`format()`[222] is a mini-`printf` specialized for ARM disassembly. It processes both the mnemonic template (`o`) and the operand template (`a`) from the `Opcode` entry, expanding format specifiers into the output buffer. The main specifiers are:

```
%C  condition suffix      (EQ, NE, CS, ... from cond[])
%S  S-bit suffix          (.S if set-flags bit is on)
```

```
%d  destination register  (Rd, bits 15-12)
%n  first operand reg     (Rn, bits 19-16)
%s  second operand reg    (Rs, bits 3-0)
%i  immediate value       (i->imm, in hex)
%b  branch target         (resolved to symbol name)
%I  memory operand        (#offset(Rn), with plocal() for SP)
%r  register list         (for LDM/STM: R0-R3,R14)
%P  block mode            (IA, DB, IB from mode[])
%p  P/W bits              (P, PW, W for pre/post/writeback)
%h  shift type            (<<, >>, ->, @> from shtype[])
%m  shift amount          (5-bit immediate)
%f  VFP precision         (F or D)
```

The function calls itself recursively: first with the mnemonic (to print the opcode name), then with nil mnemonic and the operand template (to print the arguments, separated by a tab). For the `ADD R1,R2,R3` example from `armclass()`[219] above, the mnemonic template `"ADD\%C\%S"` expands to just `"ADD"` (condition 14 = always, S-bit = 0), and the operand template `"R\%s,R\%n,R\%d"` expands to `"R3,R2,R1"` (Plan 9 destination-last syntax).

The `curr`/`end` pair is a write cursor into the output buffer—`bprint()`[144d] and `format()` append text here.

⟨*function* `format`*(arm)* 222⟩≡ (309b)

```
static void
format(char *mnemonic, Instr *i, char *f)
{
    int j, k, m, n;
    int g;
    char *fmt;

    if(mnemonic)
        format(0, i, mnemonic);
    if(f == 0)
        return;
    if(mnemonic)
        if(i->curr < i->end)
            *i->curr++ = '\t';
    for ( ; *f && i->curr < i->end; f++) {
        if(*f != '%') {
            *i->curr++ = *f;
            continue;
        }
        switch (*++f) {

        case 'C':       /* .CONDITION */
            if(cond[i->cond])
                bprint(i, ".%s", cond[i->cond]);
            break;

        case 'S':       /* .STORE */
            if(i->store)
                bprint(i, ".S");
            break;

        case 'P':       /* P & U bits for block move */
            n = (i->w >>23) & 0x3;
            if (mode[n])
                bprint(i, ".%s", mode[n]);
            break;
```

```
case 'p':       /* P & W bits for single data xfer*/
    if (pw[i->store])
        bprint(i, ".%s", pw[i->store]);
    break;

case 'a':       /* S & W bits for single data xfer*/
    if (sw[i->store])
        bprint(i, ".%s", sw[i->store]);
    break;

case 's':
    bprint(i, "%d", i->rs & 0xf);
    break;

case 'M':
    bprint(i, "%lud", (i->w>>8) & 0xf);
    break;

case 'm':
    bprint(i, "%lud", (i->w>>7) & 0x1f);
    break;

case 'h':
    bprint(i, shtype[(i->w>>5) & 0x3]);
    break;

case 'u':               /* Signed/unsigned Byte/Halfword */
    bprint(i, hb[(i->w>>5) & 0x3]);
    break;

case 'I':
    if (i->rn == 13) {
        if (plocal(i))
            break;
    }
    g = 0;
    fmt = "#%lx(R%d)";
    if (i->rn == 15) {
        /* convert load of offset(PC) to a load immediate */
        if (get4(i->map, i->addr+i->imm+8, (ulong*)&i->imm) > 0)
        {
            g = 1;
            fmt = "";
        }
    }
    if (mach->sb)
    {
        if (i->rd == 11) {
            ulong nxti;

            if (get4(i->map, i->addr+4, &nxti) > 0) {
                if ((nxti & 0x0e0f0fff) == 0x060c000b) {
                    i->imm += mach->sb;
                    g = 1;
                    fmt = "-SB";
                }
            }
        }
        if (i->rn == 12)
        {
```

```
            i->imm += mach->sb;
            g = 1;
            fmt = "-SB(SB)";
        }
    }
    if (g)
    {
        gaddr(i);
        bprint(i, fmt, i->rn);
    }
    else
        bprint(i, fmt, i->imm, i->rn);
    break;
case 'U':               /* Add/subtract from base */
    bprint(i, addsub[(i->w >> 23) & 1]);
    break;

case 'n':
    bprint(i, "%d", i->rn);
    break;

case 'd':
    bprint(i, "%d", i->rd);
    break;

case 'i':
    bprint(i, "%lux", i->imm);
    break;

case 'b':
    i->curr += symoff(i->curr, i->end-i->curr,
        (ulong)i->imm, CTEXT);
    break;

case 'g':
    i->curr += gsymoff(i->curr, i->end-i->curr,
        i->imm, CANY);
    break;

case 'f':
    switch((i->w >> 8) & 0xF){
    case 10:
        bprint(i, "F");
        break;
    case 11:
        bprint(i, "D");
        break;
    }
    break;

case 'F':
    switch(((i->w >> 15) & 0xE) + ((i->w >> 8) & 0x1)){
    case 0x0:
        bprint(i, ((i->w >> 7) & 0x1)? "WF" : "WF.U");
        break;
    case 0x1:
        bprint(i, ((i->w >> 7) & 0x1)? "WD" : "WD.U");
        break;
    case 0x8:
        bprint(i, "FW.U");
```

```
            break;
        case 0x9:
            bprint(i, "DW.U");
            break;
        case 0xA:
            bprint(i, "FW");
            break;
        case 0xB:
            bprint(i, "DW");
            break;
        case 0xE:
            bprint(i, "FD");
            break;
        case 0xF:
            bprint(i, "DF");
            break;
        }
        break;

    case 'R':
        if(((i->w >> 7) & 0x1) == 0)
            bprint(i, "R");
        break;

    case 'x':
        switch(i->rn){
        case 0:
            bprint(i, "FPSID");
            break;
        case 1:
            bprint(i, "FPSCR");
            break;
        case 2:
            bprint(i, "FPEXC");
            break;
        default:
            bprint(i, "FPS(%d)", i->rn);
            break;
        }
        break;

    case 'r':
        n = i->imm&0xffff;
        j = 0;
        k = 0;
        while(n) {
            m = j;
            while(n&0x1) {
                j++;
                n >>= 1;
            }
            if(j != m) {
                if(k)
                    bprint(i, ",");
                if(j == m+1)
                    bprint(i, "R%d", m);
                else
                    bprint(i, "R%d-R%d", m, j-1);
                k = 1;
            }
```

```
                j++;
                n >>= 1;
            }
            break;

        case '\0':
            *i->curr++ = '%';
            return;

        default:
            bprint(i, "%%%c", *f);
            break;
        }
    }
    *i->curr = 0;
}
```

Uses addsub 227a, cond 226b, fmt() 147b, gaddr() 228a, get4() 350b, gsymoff() 228c, hb 226d, mach 51a, mode, pw, shtype 226c, and symoff() 356a.

⟨*function* bprint*(arm)* 226a⟩≡ (309b)

```
static void
bprint(Instr *i, char *fmt, ...)
{
    va_list arg;

    va_start(arg, fmt);
    i->curr = vseprint(i->curr, i->end, fmt, arg);
    va_end(arg);
}
```

D.3.4 Format helpers

The cond table maps the 4-bit condition field (bits 31–28) to the Plan 9 condition suffix. Entry 14 is nil because it means "always" (no suffix printed); entry 15 is "NV" (never execute, mostly unused). For example, condition code 0 is "EQ" (equal / zero flag set), so ADD.EQ means "add only if the Z flag is set."

⟨*global* cond*(arm)* 226b⟩≡ (309b)

```
static
char*   cond[16] =
{
    "EQ",       "NE",   "CS",   "CC",
    "MI",       "PL",   "VS",   "VC",
    "HI",       "LS",   "GE",   "LT",
    "GT",       "LE",   0,      "NV"
};
```

⟨*global* shtype*(arm)* 226c⟩≡ (309b)

```
static
char*   shtype[4] =
{
    "<<",       ">>",   "->",   "@>"
};
```

⟨*global* hb*(arm)* 226d⟩≡ (309b)

```
static
char *hb[4] =
{
    "???",      "HU", "B", "H"
};
```

⟨*global* addsub*(arm)* 227a⟩≡ (309b)
```
static
char*   addsub[2] =
{
    "-",        "+",
};
```

⟨*global* mode*(arm)* 227b⟩≡ (309b)
```
static  char *mode[] = { 0, "IA", "DB", "IB" };
```

⟨*global* pw*(arm)* 227c⟩≡ (309b)
```
static  char *pw[] = { "P", "PW", 0, "W" };
```

⟨*global* sw*(arm)* 227d⟩≡ (309b)
```
static  char *sw[] = { 0, "W", "S", "SW" };
```

`plocal()`[227f] tries to resolve a memory reference relative to the stack pointer (R13) into a symbolic local variable or parameter name. It looks up the current function via the PC, finds the `.frame` symbol to get the frame size, then determines whether the offset falls in the auto (local) or param region. If it finds a match, it prints `name-offset(SP)` or `name+offset(FP)` instead of a raw hex address—this is what makes `acid`'s disassembly output readable.

⟨*global* FRAMENAME*(arm)* 227e⟩≡ (309b)
```
static  char    FRAMENAME[] = ".frame";
```

⟨*function* plocal*(arm)* 227f⟩≡ (309b)
```
static int
plocal(Instr *i)
{
    char *reg;
    Symbol s;
    char *fn;
    int class;
    int offset;

    if(!findsym(i->addr, CTEXT, &s)) {
        if(debug)fprint(2,"fn not found @%llux: %r\n", i->addr);
        return 0;
    }
    fn = s.name;
    if (!findlocal(&s, FRAMENAME, &s)) {
        if(debug)fprint(2,"%s.%s not found @%s: %r\n", fn, FRAMENAME, s.name);
            return 0;
    }
    if(s.value > i->imm) {
        class = CAUTO;
        offset = s.value-i->imm;
        reg = "(SP)";
    } else {
        class = CPARAM;
        offset = i->imm-s.value-4;
        reg = "(FP)";
    }
    if(!getauto(&s, offset, class, &s)) {
        if(debug)fprint(2,"%s %s not found @%ux: %r\n", fn,
            class == CAUTO ? " auto" : "param", offset);
        return 0;
    }
    bprint(i, "%s%c%lld%s", s.name, class == CPARAM ? '+' : '-', s.value, reg);
    return 1;
}
```

Uses `debug` 228b, `findlocal()` 332c, `findsym()` 335b, `getauto()` 334a, and `reg`.

⟨*function* `gaddr` *(arm)* 228a⟩≡ (309b)
```
static void
gaddr(Instr *i)
{
    *i->curr++ = '$';
    i->curr += gsymoff(i->curr, i->end-i->curr, i->imm, CANY);
}
```
Uses `gsymoff()` 228c.

⟨*global debug (libmach/5db.c)(arm)* 228b⟩≡ (309b)
```
static bool debug = false;
```
Uses `debug` 228b.

`gsymoff()`[228c] resolves a raw address into `symbol+offset` form by looking up the nearest symbol below the address. If the distance exceeds 4096 bytes, or the address is in a `"\.string"` section, it falls back to a plain hex literal. This is what turns `0x11B0` into `"printf"` in the disassembly output.

⟨*function* `gsymoff` *(arm)* 228c⟩≡ (309b)
```
/*
 * Print value v as name[+offset]
 */
static int
gsymoff(char *buf, int n, ulong v, int space)
{
    Symbol s;
    int r;
    long delta;

    r = delta = 0;              /* to shut compiler up */
    if (v) {
        r = findsym(v, space, &s);
        if (r)
            delta = v-s.value;
        if (delta < 0)
            delta = -delta;
    }
    if (v == 0 || r == 0 || delta >= 4096)
        return snprint(buf, n, "#%lux", v);
    if (strcmp(s.name, ".string") == 0)
        return snprint(buf, n, "#%lux", v);
    if (!delta)
        return snprint(buf, n, "%s", s.name);
    if (s.type != 't' && s.type != 'T')
        return snprint(buf, n, "%s+%llux", s.name, v-s.value);
    else
        return snprint(buf, n, "#%lux", v);
}
```
Uses `findsym()` 335b.

D.3.5 Format handlers

The format handlers below each correspond to one instruction *encoding format*. ARM has several: data processing with register operand (`armdps`[229a]), data processing with immediate (`armdpi`[229b]), single data transfer with immediate offset (`armsdti`[230a]), single data transfer with register offset (`armsdts`[231b]), block data transfer (`armbdt`[231c]), branch (`armb`[232a]), and so on. Each handler extracts the operand fields specific to its format, then calls `format()`[222] to produce the output string.

`armdps()`[229a] handles data processing instructions with register operands (the first 48 entries in `opcodes`[214b]). It extracts the S bit, Rn, Rd, and Rs fields. Special cases detect MSR/MRS (moves to/from the status register), which share the same encoding space.

⟨*function* `armdps` *(arm)* 229a⟩≡ (309b)

```
static void
armdps(Opcode *o, Instr *i)
{
    i->store = (i->w >> 20) & 1;
    i->rn = (i->w >> 16) & 0xf;
    i->rd = (i->w >> 12) & 0xf;
    i->rs = (i->w >> 0) & 0xf;
    if(i->rn == 15 && i->rs == 0) {
        if(i->op == 8) {
            format("MOVW", i,"CPSR, R%d");
            return;
        } else
        if(i->op == 10) {
            format("MOVW", i,"SPSR, R%d");
            return;
        }
    } else
    if(i->rn == 9 && i->rd == 15) {
        if(i->op == 9) {
            format("MOVW", i, "R%s, CPSR");
            return;
        } else
        if(i->op == 11) {
            format("MOVW", i, "R%s, SPSR");
            return;
        }
    }
    if(i->rd == 15) {
        if(i->op == 120) {
            format("MOVW", i, "PSR, %x");
            return;
        } else
        if(i->op == 121) {
            format("MOVW", i, "%x, PSR");
            return;
        }
    }
    format(o->o, i, o->a);
}
```

`armdpi()`[229b] handles data processing with an immediate operand. The ARM encoding packs an 8-bit value and a 4-bit rotation into 12 bits: the immediate is the 8-bit value rotated right by 2× the 4-bit field. This allows encoding common constants like `0xFF`, `0xFF00`, and `0xFF000000` in a single instruction. The `while(c)` loop performs this rotation. Two special cases detect `RET` (encoded as `ADD \$0,R14,R15`—add zero to the link register and store in PC) and indirect branches through a register (`ADD \$0,Rn,R15`).

⟨*function* `armdpi` *(arm)* 229b⟩≡ (309b)

```
static void
armdpi(Opcode *o, Instr *i)
{
    ulong v;
    int c;

    v = (i->w >> 0) & 0xff;
    c = (i->w >> 8) & 0xf;
```

```
    while(c) {
        v = (v<<30) | (v>>2);
        c--;
    }
    i->imm = v;
    i->store = (i->w >> 20) & 1;
    i->rn = (i->w >> 16) & 0xf;
    i->rd = (i->w >> 12) & 0xf;
    i->rs = i->w&0x0f;

        /* RET is encoded as ADD #0,R14,R15 */
    if((i->w & 0x0fffffff) == 0x028ef000){
        format("RET%C", i, "");
        return;
    }
    if((i->w & 0x0ff0ffff) == 0x0280f000){
        format("B%C", i, "0(R%n)");
        return;
    }
    format(o->o, i, o->a);
}
```

⟨*function* armsdti*(arm)* 230a⟩≡ (309b)

```
static void
armsdti(Opcode *o, Instr *i)
{
    ulong v;

    v = i->w & 0xfff;
    if(!(i->w & (1<<23)))
        v = -v;
    i->store = ((i->w >> 23) & 0x2) | ((i->w >>21) & 0x1);
    i->imm = v;
    i->rn = (i->w >> 16) & 0xf;
    i->rd = (i->w >> 12) & 0xf;
        /* RET is encoded as LW.P x,R13,R15 */
    if ((i->w & 0x0ffff000) == 0x049df000)
    {
        format("RET%C%p", i, "%I");
        return;
    }
    format(o->o, i, o->a);
}
```

⟨*function* armvstdi*(arm)* 230b⟩≡ (309b)

```
static void
armvstdi(Opcode *o, Instr *i)
{
    ulong v;

    v = (i->w & 0xff) << 2;
    if(!(i->w & (1<<23)))
        v = -v;
    i->imm = v;
    i->rn = (i->w >> 16) & 0xf;
    i->rd = (i->w >> 12) & 0xf;
    format(o->o, i, o->a);
}
```

⟨*function* armhwby*(arm)* 231a⟩≡ (309b)

```
/* arm V4 ld/st halfword, signed byte */
static void
armhwby(Opcode *o, Instr *i)
{
    i->store = ((i->w >> 23) & 0x2) | ((i->w >>21) & 0x1);
    i->imm = (i->w & 0xf) | ((i->w >> 8) & 0xf);
    if (!(i->w & (1 << 23)))
        i->imm = - i->imm;
    i->rn = (i->w >> 16) & 0xf;
    i->rd = (i->w >> 12) & 0xf;
    i->rs = (i->w >> 0) & 0xf;
    format(o->o, i, o->a);
}
```

⟨*function* armsdts*(arm)* 231b⟩≡ (309b)

```
static void
armsdts(Opcode *o, Instr *i)
{
    i->store = ((i->w >> 23) & 0x2) | ((i->w >>21) & 0x1);
    i->rs = (i->w >> 0) & 0xf;
    i->rn = (i->w >> 16) & 0xf;
    i->rd = (i->w >> 12) & 0xf;
    format(o->o, i, o->a);
}
```

⟨*function* armbdt*(arm)* 231c⟩≡ (309b)

```
static void
armbdt(Opcode *o, Instr *i)
{
    i->store = (i->w >> 21) & 0x3;              /* S & W bits */
    i->rn = (i->w >> 16) & 0xf;
    i->imm = i->w & 0xffff;
    if(i->w == 0xe8fd8000)
        format("RFE", i, "");
    else
        format(o->o, i, o->a);
}
```

⟨*function* armund*(arm)* 231d⟩≡ (309b)

```
static void
armund(Opcode *o, Instr *i)
{
    format(o->o, i, o->a);
}
```

⟨*function* armcdt*(arm)* 231e⟩≡ (309b)

```
static void
armcdt(Opcode *o, Instr *i)
{
    format(o->o, i, o->a);
}
```

⟨*function* armunk*(arm)* 231f⟩≡ (309b)

```
static void
armunk(Opcode *o, Instr *i)
{
    format(o->o, i, o->a);
}
```

`armb()`[232a] handles B (branch) and BL (branch-and-link). The 24-bit offset is sign-extended and shifted left by 2 (since ARM instructions are 4-byte aligned, the bottom 2 bits are always zero and can be left out of the encoding). The +8 accounts for the ARM pipeline: the PC used for the offset calculation is 2 instructions ahead of the branch itself. Concrete example: `BL printf` at address `0x10A0` with encoding `0xEB000042`:

```
0xEB000042:
  cond = 0xE (AL = always)
  bits 27-25 = 101 -> case 5 (branch)
  bit 24 = 1 -> BL (not B)
  offset bits 23-0 = 0x000042

armb():
  v = 0x000042
  bit 23 = 0, so no sign extension
  imm = (0x42 << 2) + 0x10A0 + 8
      = 0x108 + 0x10A8
      = 0x11B0
  -> "BL  printf"  (if symtab has printf at 0x11B0)
```

⟨*function* `armb` *(arm)* 232a⟩≡ (309b)

```
static void
armb(Opcode *o, Instr *i)
{
    ulong v;

    v = i->w & 0xffffff;
    if(v & 0x800000)
        v |= ~0xffffff;
    i->imm = (v<<2) + i->addr + 8;
    format(o->o, i, o->a);
}
```

⟨*function* `armbpt` *(arm)* 232b⟩≡ (309b)

```
static void
armbpt(Opcode *o, Instr *i)
{
    i->imm = ((i->w >> 4) & 0xfff0) | (i->w &0xf);
    format(o->o, i, o->a);
}
```

⟨*function* `armco` *(arm)* 232c⟩≡ (309b)

```
static void
armco(Opcode *o, Instr *i)              /* coprocessor instructions */
{
    int op, p, cp;

    char buf[1024];

    i->rn = (i->w >> 16) & 0xf;
    i->rd = (i->w >> 12) & 0xf;
    i->rs = i->w&0xf;
    cp = (i->w >> 8) & 0xf;
    p = (i->w >> 5) & 0x7;
    if(i->w&(1<<4)) {
        op = (i->w >> 21) & 0x07;
        snprint(buf, sizeof(buf), "#%x, #%x, R%d, C(%d), C(%d), #%x", cp, op, i->rd, i->rn, i->rs, p);
    } else {
```

```
        op = (i->w >> 20) & 0x0f;
        snprint(buf, sizeof(buf), "#%x, #%x, C(%d), C(%d), C(%d), #%x", cp, op, i->rd, i->rn, i->rs, p);
    }
    format(o->o, i, buf);
}
```

D.4 ARM follow set

The last piece of the ARM backend is the follow set computation: given the current `pc`, return the set of possible next program counters. ARM has no hardware single-step flag (unlike x86's `TF`), so `acid` implements single-stepping in software by planting a temporary breakpoint at each follow-set address, resuming the process with `startstop`, and removing them after the next stop (see the `step()` function covered earlier in the `acid` builtins chapter). This section walks through `armfoll()`[233], the top-level dispatch, and the per-class helpers that handle branches, conditional branches, and load/store-multiple instructions that can write to `PC`.

D.4.1 `armfoll()`

⟨*function* `armfoll` *(arm)* 233⟩≡ (309b)

```
static int
armfoll(Map *map, uvlong pc, Rgetter rget, uvlong *foll)
{
    uvlong d;
    Instr i;

    if(decode(map, pc, &i) < 0)
        return -1;

    if(opcodes[i.op].foll) {
        d = (*opcodes[i.op].foll)(map, rget, &i, pc);
        if(d == -1)
            return -1;
    } else
        d = pc+4;

    foll[0] = d;
    return 1;
}
```

D.4.2 `armfxxx()`

The `armf*` functions compute the follow set—the address(es) where execution will go after a given instruction. `acid` calls them through `armfoll()`[233] to implement single-stepping. The key insight is that on ARM, *any* instruction that writes to R15 (the PC) is a control-flow transfer, not just branches. So there are five cases:

- `armfadd`[234a]: data processing instructions (ADD, SUB, etc.) whose destination is R15. Computes `Rn + shifter\_operand`.
- `armfbranch`[235b]: B and BL. The target is PC-relative with a 24-bit signed offset, plus the 8-byte pipeline advance.
- `armfbx`[234b]: BX/BLX register. Target is the register value.
- `armfmov`[235c]: MOV or LDR into R15. MOV returns the shifted operand; LDR reads the target address from memory.

- `armfmovm`[234c]: block load (LDM) that includes R15 in the register list. Computes the stack address where R15 would be loaded from.

All of them first call `armcondpass()`[236b] to check whether the condition passes. If not, the follow address is simply `pc+4` (the instruction is a no-op).

⟨*function* `armfadd`*(arm)* 234a⟩≡ (309b)
```
static uvlong
armfadd(Map *map, Rgetter rget, Instr *i, uvlong pc)
{
    char buf[8];
    int r;

    r = (i->w >> 12) & 0xf;
    if(r != 15 || !armcondpass(map, rget, (i->w >> 28) & 0xf))
        return pc+4;

    r = (i->w >> 16) & 0xf;
    sprint(buf, "R%d", r);

    return rget(map, buf) + armshiftval(map, rget, i);
}
```
Uses `armcondpass()` 236b and `armshiftval()` 237.

⟨*function* `armfbx`*(arm)* 234b⟩≡ (309b)
```
static uvlong
armfbx(Map *map, Rgetter rget, Instr *i, uvlong pc)
{
    char buf[8];
    int r;

    if(!armcondpass(map, rget, (i->w>>28)&0xf))
        return pc+4;
    r = (i->w >> 0) & 0xf;
    sprint(buf, "R%d", r);
    return rget(map, buf);
}
```
Uses `armcondpass()` 236b.

⟨*function* `armfmovm`*(arm)* 234c⟩≡ (309b)
```
static uvlong
armfmovm(Map *map, Rgetter rget, Instr *i, uvlong pc)
{
    ulong v;
    ulong addr;

    v = i->w & 1<<15;
    if(!v || !armcondpass(map, rget, (i->w>>28)&0xf))
        return pc+4;

    addr = armmaddr(map, rget, i) + nbits(i->w & BITS(0,15));
    if(get4(map, addr, &v) < 0) {
        werrstr("can't read addr: %r");
        return -1;
    }
    return v;
}
```
Uses `BITS-84` 212b, `armcondpass()` 236b, `armmaddr()` 235a, `get4()` 350b, and `nbits()` 212f.

⟨*function* `armmaddr`*(arm)* 235a⟩≡ (309b)

```
static ulong
armmaddr(Map *map, Rgetter rget, Instr *i)
{
    ulong v;
    ulong nb;
    char buf[8];
    ulong rn;

    rn = (i->w >> 16) & 0xf;
    sprint(buf,"R%ld", rn);

    v = rget(map, buf);
    nb = nbits(i->w & ((1 << 15) - 1));

    switch((i->w >> 23) & 3) {
    default:
    case 0: return (v - (nb*4)) + 4;
    case 1: return v;
    case 2: return v - (nb*4);
    case 3: return v + 4;
    }
}
```

Uses `nbits()` 212f.

⟨*function* `armfbranch`*(arm)* 235b⟩≡ (309b)

```
static uvlong
armfbranch(Map *map, Rgetter rget, Instr *i, uvlong pc)
{
    if(!armcondpass(map, rget, (i->w >> 28) & 0xf))
        return pc+4;

    return pc + (((signed long)i->w << 8) >> 6) + 8;
}
```

Uses `armcondpass()` 236b.

⟨*function* `armfmov`*(arm)* 235c⟩≡ (309b)

```
static uvlong
armfmov(Map *map, Rgetter rget, Instr *i, uvlong pc)
{
    ulong rd, v;

    rd = (i->w >> 12) & 0xf;
    if(rd != 15 || !armcondpass(map, rget, (i->w>>28)&0xf))
        return pc+4;

     /* LDR */
    /* BUG: Needs LDH/B, too */
    if(((i->w>>26)&0x3) == 1) {
        if(get4(map, armaddr(map, rget, i), &v) < 0) {
            werrstr("can't read instruction: %r");
            return pc+4;
        }
        return v;
    }

     /* MOV */
    v = armshiftval(map, rget, i);

    return v;
```

```
}
```

Uses armaddr() 236a, armcondpass() 236b, armshiftval() 237, and get4() 350b.

⟨*function* armaddr*(arm)* 236a⟩≡ (309b)

```
static uvlong
armaddr(Map *map, Rgetter rget, Instr *i)
{
    char buf[8];
    ulong rn;

    snprint(buf, sizeof(buf), "R%ld", (i->w >> 16) & 0xf);
    rn = rget(map, buf);

    if((i->w & (1<<24)) == 0)                   /* POSTIDX */
        return rn;

    if((i->w & (1<<25)) == 0) {                 /* OFFSET */
        if(i->w & (1U<<23))
            return rn + (i->w & BITS(0,11));
        return rn - (i->w & BITS(0,11));
    } else {                                    /* REGOFF */
        ulong index = 0;
        uchar c;
        uchar rm;

        sprint(buf, "R%ld", i->w & 0xf);
        rm = rget(map, buf);

        switch((i->w & BITS(5,6)) >> 5) {
        case 0: index = rm << ((i->w & BITS(7,11)) >> 7);       break;
        case 1: index = LSR(rm, ((i->w & BITS(7,11)) >> 7));    break;
        case 2: index = ASR(rm, ((i->w & BITS(7,11)) >> 7));    break;
        case 3:
            if((i->w & BITS(7,11)) == 0) {
                c = (rget(map, "PSR") >> 29) & 1;
                index = c << 31 | LSR(rm, 1);
            } else {
                index = ROR(rm, ((i->w & BITS(7,11)) >> 7));
            }
            break;
        }
        if(i->w & (1<<23))
            return rn + index;
        return rn - index;
    }
}
```

Uses ASR-86 212d, BITS-84 212b, LSR-85 212c, and ROR-87 212e.

D.4.3 armfxxx() helpers

armcondpass()[236b] evaluates whether a conditional instruction would actually execute, given the current CPSR flags. It reads the PSR register through the rget callback, extracts the N/Z/C/V flags, and evaluates the condition code. This is essential for the follow-set: a conditional branch that *would not be taken* should return pc+4 (fall through), not the branch target.

⟨*function* armcondpass*(arm)* 236b⟩≡ (309b)

```
static int
armcondpass(Map *map, Rgetter rget, uchar cond)
```

```
{
    uvlong psr;
    uchar n;
    uchar z;
    uchar c;
    uchar v;

    psr = rget(map, "PSR");
    n = (psr >> 31) & 1;
    z = (psr >> 30) & 1;
    c = (psr >> 29) & 1;
    v = (psr >> 28) & 1;

    switch(cond) {
    default:
    case 0:             return z;
    case 1:             return !z;
    case 2:             return c;
    case 3:             return !c;
    case 4:             return n;
    case 5:             return !n;
    case 6:             return v;
    case 7:             return !v;
    case 8:             return c && !z;
    case 9:             return !c || z;
    case 10:    return n == v;
    case 11:    return n != v;
    case 12:    return !z && (n == v);
    case 13:    return z || (n != v);
    case 14:    return 1;
    case 15:    return 0;
    }
}
```

`armshiftval()`[237] evaluates the ARM "flexible second operand" (the barrel shifter). For the follow-set functions, the disassembler needs the *actual computed value*, not just the encoding. If bit 25 is set, the operand is an 8-bit immediate rotated by a 4-bit field (same as `armdpi`[229b]). Otherwise, it is a register value shifted by one of four modes: `LSL`, `LSR`, `ASR`, or `ROR`, with the shift amount coming from either an immediate (bits 11–7) or another register (bits 11–8).

⟨*function* `armshiftval` *(arm)* 237⟩≡ (309b)

```
static ulong
armshiftval(Map *map, Rgetter rget, Instr *i)
{
    if(i->w & (1 << 25)) {                              /* immediate */
        ulong imm = i->w & BITS(0, 7);
        ulong s = (i->w & BITS(8, 11)) >> 7; /* this contains the *2 */
        return ROR(imm, s);
    } else {
        char buf[8];
        ulong v;
        ulong s = (i->w & BITS(7,11)) >> 7;

        sprint(buf, "R%ld", i->w & 0xf);
        v = rget(map, buf);

        switch((i->w & BITS(4, 6)) >> 4) {
        default:
        case 0:                                 /* LSLIMM */
            return v << s;
```

```
        case 1:                                 /* LSLREG */
            sprint(buf, "R%lud", s >> 1);
            s = rget(map, buf) & 0xFF;
            if(s >= 32) return 0;
            return v << s;
        case 2:                                 /* LSRIMM */
            return LSR(v, s);
        case 3:                                 /* LSRREG */
            sprint(buf, "R%ld", s >> 1);
            s = rget(map, buf) & 0xFF;
            if(s >= 32) return 0;
            return LSR(v, s);
        case 4:                                 /* ASRIMM */
            if(s == 0) {
                if((v & (1U<<31)) == 0)
                    return 0;
                return 0xFFFFFFFF;
            }
            return ASR(v, s);
        case 5:                                 /* ASRREG */
            sprint(buf, "R%ld", s >> 1);
            s = rget(map, buf) & 0xFF;
            if(s >= 32) {
                if((v & (1U<<31)) == 0)
                    return 0;
                return 0xFFFFFFFF;
            }
            return ASR(v, s);
        case 6:                                 /* RORIMM */
            if(s == 0) {
                ulong c = (rget(map, "PSR") >> 29) & 1;

                return (c << 31) | LSR(v, 1);
            }
            return ROR(v, s);
        case 7:                                 /* RORREG */
            sprint(buf, "R%ld", (s>>1)&0xF);
            s = rget(map, buf);
            if(s == 0 || (s & 0xF) == 0)
                return v;
            return ROR(v, s & 0xF);
        }
    }
}
```

Uses ASR-86 212d, BITS-84 212b, LSR-85 212c, and ROR-87 212e.

Appendix E

db

`db` is the traditional Plan 9 debugger, descended from Ken Thompson's original UNIX `db` and Steve Bourne's `adb`. Because `acid` is strictly more powerful than `db`, the main book focuses on `acid`. This appendix presents `db` for historical interest and because its compact code makes the shared debugging mechanisms (breakpoints, single-step, stack trace) easier to see without the interpreter overhead.

E.1 Overview

`db` provides a fixed set of single-character commands for inspecting and controlling a target process, in contrast to `acid`'s programmable language. Both debuggers use the same kernel interface (`/proc/<pid>/`) and the same `libmach` library for symbol tables, disassembly, and follow-set computation, so the underlying debugging *mechanisms* are identical—what differs is the user interface. The main architectural difference between `acid` and `db` are:

```
acid                                db
----                                --
yacc parser + interpreter           hand-written [addr][,count][cmd]
user-defined functions (defn)       fixed command set
type system (complex/aggr)          no type awareness
garbage-collected AST/values        no heap allocation for values
~5000 LOC (acid/ + /lib/acid/)      ~2000 LOC (db/)
```

E.1.1 `db` services

⟨`main()` *print usage and exit (db)* 239⟩≡ (245d)

```
fprint(STDERR, "Usage: db [-kw] [-m machine] [-I dir] ([symfile]|[pid])\n");
exits("usage");
```

The command grammar is inherited from `adb` and is extremely terse—every command is a single character from this set:

```
[address] [,count] [command]

Commands:
  ?    inspect text (symmap)       :r   run program
  /    inspect data (cormap)       :c   continue
  =    print expression value      :s   single step (asm)
  $    dump state ($r,$c,$b,...)   :S   single step (C line)
  >    set register                :b   set breakpoint
```

```
!    shell escape              :d   delete breakpoint
:    process control           :k   kill process
                               :h   halt (stop)
                               :x   unhalt (resume)
```

The key idea is that `?` reads from the text file (`symmap`), `/` reads from the live process memory (`cormap`), and `=` just prints the expression value without reading memory at all. `acid` retains the same `*` and `@` dereference operators (through `cormap` and `symmap` respectively), but in practice most users interact through higher-level library functions like `src()`, `stk()`, and `mem()` that hide the map choice.

⟨*constant* `CMD_VERBS` 240⟩≡ (426c)

```
#define CMD_VERBS "?/=>!$: \t"
```

E.1.2 Debugging `hello_bug.c`

Here is the same `hello_bug.c` debugging session from Section 2.5, but using `db` instead of `acid`. The same bug (divide by zero), the same program, the same executable—only the debugger interface differs:

```
acid session                    db session
============                    ==========
$ acid ./hello_bug              $ db ./hello_bug
acid: new()                     :r
acid: bpset(bar)                bar:b
acid: cont()                    :c
66: breakpoint bar ...          breakpoint
acid: stk()                     $c
bar(a=0x2a,b=0x0)               bar(a=2a,b=0) ...
main(argc=0x1) ...                main(argc=1) ...
acid: src(*PC)                  (no equivalent; use ?i)
acid: cont()                    :c
-- divide fault --              stopped at
                                  foo+0x5/ IDIVL AX,CX
acid: stk()                     $C
foo(x=0x2a,y=0x0) ...            foo(x=2a,y=0) ...
bar(a=0x2a,b=0x0) ...              bar(a=2a,b=0) ...
```

The key observations: `db` commands are single characters (`:r` vs `new()`, `bar:b` vs `bpset(bar)`, `$c` vs `stk()`), which makes them faster to type but harder to remember. `db` has no `src()` command—you inspect code with `?i` (disassembly) rather than source lines. And `db`'s `$C` includes local variables automatically (like `acid`'s `lstk()`).

E.2 Core data structures

`db` uses the same `libmach` types as `acid`: `Map` for memory access (`symmap` for the executable file, `cormap` for the live process), `Fhdr` for the parsed header, and the symbol table functions (`findsym()`, `fileline()`, `pc2line()`). The difference is that `db` manages process control directly through global file descriptors (`msgfd` for `/proc/<pid>/ctl`, `notefd` for `/proc/<pid>/note`) rather than through `acid`'s `ptab`[56b] array. Where `acid`'s `msg()`[56d] looks up a PID in `ptab` to find the right `ctl` fd, `db`'s `msgpcs()`[242a] just writes to the single global `msgfd`[241c]—because `db` can only debug one process at a time.

Here is the relationship between `db`'s main data structures and the files they point to:

```
db globals            /proc/<pid>/         executable file
----------            ---------------      ---------------
symmap  ----------->  (not /proc)  ------> a.out or 8.out
  .seg[0] "text"                           text segment
  .seg[1] "data"                           data segment

cormap  ----------->  /proc/<pid>/mem
  .seg[0] "regs"  ->  /proc/<pid>/regs
  .seg[1] "fpregs"->  /proc/<pid>/fpregs
  .seg[2] "text"  ->  /proc/<pid>/mem      (text range)
  .seg[3] "data"  ->  /proc/<pid>/mem      (data range)

msgfd   ----------->  /proc/<pid>/ctl      (write "start" etc.)
notefd  ----------->  /proc/<pid>/note     (read/write notes)
```

This is structurally identical to `acid`'s setup (compare with the `symmap`/`cormap` diagram in Chapter 6), just with fewer globals since there is no multi-process support.

E.2.1 /proc/<pid>/text and symmap

⟨*global* `symmap` 241a⟩≡ (429b)
```
// was in setup.c
Map *symmap;
```

E.2.2 /proc/<pid>/mem and cormap

⟨*global* `cormap` 241b⟩≡ (429b)
```
Map *cormap;
```

E.2.3 /proc/<pid>/ctl, msgfd, and msgpcs()

⟨*global* `msgfd` 241c⟩≡ (439b)
```
fdt msgfd = -1;
```
Uses `msgfd` 241c.

⟨*global* `pcspid` 241d⟩≡ (439b)
```
int pcspid = -1;
```
Uses `pcspid` 241d.

⟨*function* `setpcs` 241e⟩≡ (439b)
```
void
setpcs(void)
{
    char buf[128];

    if(pid && pid != pcspid){
        if(msgfd >= 0){
            close(msgfd);
            msgfd = -1;
        }
        ⟨setpcs() close previous notefd if changed process 244b⟩

        pcspid = -1;
```

```
            sprint(buf, "/proc/%d/ctl", pid);
            msgfd = open(buf, OWRITE);
            if(msgfd < 0)
                error("can't open control file");

            ⟨setpcs() open notefd for new process 244c⟩

            pcspid = pid;
        }
    }
```

Uses msgfd 241c, pcspid 241d, and pid 246f.

⟨*function* msgpcs 242a⟩≡ (439b)

```
void
msgpcs(char *msg)
{
    int ret;
    ⟨msgpcs() locals 242b⟩

    setpcs();
    //dprint("--> %d: %s\n", pcspid, msg);
    ret = write(msgfd, msg, strlen(msg));
    ⟨msgpcs() error managment 242c⟩
}
```

Uses msgfd 241c and setpcs() 241e.

⟨msgpcs() *locals* 242b⟩≡ (242a)

```
char err[ERRMAX];
```

⟨msgpcs() *error managment* 242c⟩≡ (242a)

```
if(ret < 0 && !ending){
    errstr(err, sizeof err);
    if(strcmp(err, "interrupted") != 0)
        endpcs();
    errors("can't write control file", err);
}
```

Uses ending 287c, endpcs() 287d, and errors() 303a.

E.2.4 Addr and dot

The dot variable is db's central concept—it is the "current address" that most commands operate on implicitly. After main ?i, dot is at main; after printing three instructions with 3i, dot advances by 12 bytes (3 × 4 for ARM). Pressing Enter repeats the last command at the new dot, so you can page through code or data just by hitting Enter. This implicit-current-address model is what makes adb terse: you rarely need to type an address twice. acid has no equivalent of dot. Every operation takes an explicit address argument: *main\i instead of main ?i then Enter.

⟨*type* ADDR 242d⟩≡ (426c)

```
typedef uvlong ADDR;
```

⟨*global* dot 242e⟩≡ (429b)

```
// was in command.c
ADDR dot;
```

E.2.5 cntval

⟨*type* WORD 242f⟩≡ (426c)

```
typedef ulong WORD;
```

⟨*global* cntval 243a⟩≡ (429b)
```
WORD cntval;
```

E.2.6 Bkpt and bkpthead

The Bkpt struct is db's version of acid's breakpoint list (bplistX). The key difference is that db stores breakpoints in a linked list of C structs with a flag field tracking their state, while acid stores them as an acid list of addresses and uses library functions to manage them. The save[4] field holds the original instruction bytes that were overwritten by the breakpoint trap—db saves and restores them through bkput()[291b], using exactly the same get1/put1 mechanism as acid. The flag field tracks a small state machine:

```
BKPTCLR  (0) ---[user :b]---> BKPTSET (1)
BKPTSET  (1) ---[bp hit]----> BKPTSKIP (2)  (skip next time)
BKPTSKIP (2) ---[stepped]---> BKPTSET (1)   (re-arm)
BKPTTMP  (3) ---[bp hit]----> BKPTCLR (0)   (one-shot, for :s)
```

The BKPTSKIP state exists to handle the "step over a breakpoint" problem: after a breakpoint fires, the debugger must single-step past the original instruction before re-inserting the trap. acid solves the same problem in its cont() library function.

⟨*struct* bkpt 243b⟩≡ (426c)
```
struct bkpt {
    // address to break on
    ADDR loc;
    // enum<breakpoint_kind>
    int flag;

    // original code at the breapoint
    byte save[4];

    ⟨Bkpt other fields 243i⟩

    // Extra
    ⟨Bkpt extra fields 243h⟩
};
```

⟨*constant* BKPTCLR 243c⟩≡ (426c)
```
#define BKPTCLR 0 /* not a real breakpoint */
```

⟨*constant* BKPTSET 243d⟩≡ (426c)
```
#define BKPTSET 1 /* real, ready to trap */
```

⟨*constant* BKPTSKIP 243e⟩≡ (426c)
```
#define BKPTSKIP 2 /* real, skip over it next time */
```

⟨*constant* BKPTTMP 243f⟩≡ (426c)
```
#define BKPTTMP 3 /* temporary; clear when it happens */
```

⟨*global* bkpthead 243g⟩≡ (429b)
```
// list<BKPT>, next = BKPT.next
BKPT *bkpthead;
```

⟨Bkpt *extra fields* 243h⟩≡ (243b)
```
// list<ref_own<BKPT>, head = bkpthead
BKPT *nxtbkpt;
```

⟨Bkpt *other fields* 243i⟩≡ (243b) 294c▷
```
int count;
int initcnt;
```

E.2.7 /proc/<pid>/note, notefd, and notes

⟨*global* `notefd` 244a⟩≡ (439b)
```
fdt notefd = -1;
```
Uses `notefd` 244a.

⟨`setpcs()` *close previous notefd if changed process* 244b⟩≡ (241e)
```
if(notefd >= 0){
    close(notefd);
    notefd = -1;
}
```
Uses `notefd` 244a.

⟨`setpcs()` *open notefd for new process* 244c⟩≡ (241e)
```
sprint(buf, "/proc/%d/note", pid);
notefd = open(buf, ORDWR);
if(notefd < 0)
    error("can't open note file");
```
Uses `notefd` 244a and `pid` 246f.

⟨*global* `note` 244d⟩≡ (429b)
```
// array<option<ref_own<string>>, actual number of elts used = nnote
char note[NNOTE][ERRMAX];
```
Uses `NNOTE` 244e.

⟨*constant* `NNOTE` 244e⟩≡ (426c)
```
#define NNOTE 10
```

⟨*global* `nnote` 244f⟩≡ (429b)
```
int nnote;
```

E.3 /bin/db

The overall structure of `db` is much simpler than `acid`'s because there is no interpreter:

```
main()
  |
  +-- setsym()          (parse executable, load symbols)
  +-- setcor()          (open /proc/<pid>/mem if attached)
  |
  +-- for(;;)           (REPL loop)
        |
        +-- clrinp()    (reset input)
        +-- rdc()       (skip whitespace)
        +-- command()   (parse and execute one command)
              |
              +-- expr()        (parse optional address)
              +-- expr()        (parse optional count)
              +-- switch(cmd)   (dispatch: ?, /, =, $, :, >, !)
```

Compare with `acid`'s flow: `main` → `yyparse` → `execute` → `expop` dispatch. `db` has no parse tree, no values, no garbage collector—it reads a command character, dispatches on it, and calls `libmach` functions directly.

E.3.1 main()

⟨*global* symfil 245a⟩≡ (432)
```
char *symfil = nil;
```
Uses symfil 245a.

⟨*global* corfil 245b⟩≡ (432)
```
char *corfil = nil;
```
Uses corfil 245b.

⟨*global* infile 245c⟩≡ (430)
```
int infile = STDIN;
```
Uses infile 245c.

⟨*function main (db/main.c)* 245d⟩≡ (444)
```
void
main(int argc, char **argv)
{
    ⟨main() locals (db) 246g⟩

    outputinit();

    ARGBEGIN{
    ⟨main() command line processing (db) 297c⟩
    }ARGEND

    if (argc > 0 && !alldigs(argv[0])) {
        symfil = argv[0];
        argv++;
        argc--;
    }
    ⟨main() if pid argument, attach to existing process 246h⟩
    else if (argc > 0) {
        ⟨main() print usage and exit (db) 239⟩
    }

   if (!symfil)
        symfil = "8.out";

    ⟨main() initialization before repl (db) 246b⟩

    // repl loop
    for (;;) {
        // clear output
        flushbuf();

        ⟨main() in loop, handle errmsg (db) 303c⟩
        ⟨main() in loop, handle mkfault (db) 301a⟩

        // clear input
        clrinp();

        // go to next non whitespace char
        rdc();
        reread();
        ⟨main() in loop, if eof (db) 255a⟩

        command(nil, 0);

        reread();
```

```
        if (rdc() != '\n')
            error("newline expected");
    }
}
```

⟨*function* `alldigs` 246a⟩≡ (444)
```
bool
alldigs(char *s)
{
    while(*s){
        if(*s<'0' || '9'<*s)
            return false;
        s++;
    }
    return true;
}
```

⟨`main()` *initialization before repl (db)* 246b⟩≡ (245d) 298d▷
```
⟨main() call notify 300a⟩
⟨main() call setsym 247b⟩
⟨main() set dotmap 268e⟩
⟨main() print binary architecture 246c⟩
⟨main() setjmp 246e⟩
⟨main() just before repl 279d⟩
```

⟨`main()` *print binary architecture* 246c⟩≡ (246b)
```
⟨main() if db -m and unknown machine 298g⟩
dprint("%s binary\n", mach->name);
```

⟨*global* `env` 246d⟩≡ (429b)
```
jmp_buf env;
```

⟨`main()` *setjmp* 246e⟩≡ (246b)
```
if(setjmp(env) == 0){
    ⟨main() if setjmp == 0 and corfil 249a⟩
}
setjmp(env);
```

E.3.2 Process attachment: `pid`

⟨*global* `pid` 246f⟩≡ (429b)
```
int pid;
```

⟨`main()` *locals (db)* 246g⟩≡ (245d) 297b▷
```
char b1[100];
char b2[100];
```

⟨`main()` *if pid argument, attach to existing process* 246h⟩≡ (245d)
```
if(argc==1 && alldigs(argv[0])){

    pid = atoi(argv[0]);
    pcsactive = false;
    if (!symfil) {
        ⟨main() when pid argument, if kflag 299d⟩
        else
            sprint(b1, "/proc/%s/text", argv[0]);
        symfil = b1;
    }
    sprint(b2, "/proc/%s/mem", argv[0]);
    corfil = b2;
}
```

⟨*global* `pcsactive` 247a⟩≡ (429b)
```
// was in trcrun.c
bool pcsactive = false;
```
Uses `pcsactive` 247a.

E.3.3 Text file: `setsym()`

⟨`main()` *call setsym* 247b⟩≡ (246b)
```
setsym();
```

⟨*global fhdr (db/setup.c)* 247c⟩≡ (432)
```
static Fhdr fhdr;
```

⟨*global* `fsym` 247d⟩≡ (432)
```
fdt fsym;
```

⟨*function setsym (db)* 247e⟩≡ (432)
```
void
setsym(void)
{
    int ret;
    ⟨setsym() locals 248c⟩

    fsym = getfile(symfil, 1, wtflag);
    ⟨setsym() error managment on fsym 247f⟩
    ret = crackhdr(fsym, &fhdr);
    if (ret) {
        machbytype(fhdr.type);
        symmap = loadmap(symmap, fsym, &fhdr);
        ⟨setsym() error managment on symmap 247g⟩
        ret = syminit(fsym, &fhdr);
        ⟨setsym() error managment on syminit 248a⟩

        ⟨setsym() if mach has sbreg 248d⟩
    }
    ⟨setsym() error managment on crackhdr 247h⟩
}
```
Uses `crackhdr()` 90, `fhdr-68` 247c, `fsym` 247d, `loadmap()` 92, `machbytype()` 91, `symfil` 245a, and `syminit()` 94.

⟨`setsym()` *error managment on fsym* 247f⟩≡ (247e)
```
if(fsym < 0) {
    symmap = dumbmap(-1);
    return;
}
```
Uses `dumbmap()` 248b and `fsym` 247d.

⟨`setsym()` *error managment on symmap* 247g⟩≡ (247e)
```
if (symmap == nil)
    symmap = dumbmap(fsym);
```
Uses `dumbmap()` 248b and `fsym` 247d.

⟨`setsym()` *error managment on crackhdr* 247h⟩≡ (247e)
```
else
    symmap = dumbmap(fsym);
```
Uses `dumbmap()` 248b and `fsym` 247d.

⟨setsym() *error managment on syminit* 248a⟩≡ (247e)
```
if (ret < 0)
    dprint("%r\n");
```
Uses dprint() 250e.

⟨*function* dumbmap 248b⟩≡ (432)
```
Map *
dumbmap(fdt fd)
{
    Map *dumb;


    dumb = newmap(0, 1);
    setmap(dumb, fd, 0, 0xffffffff, 0, "data");
    if (!mach)    /* default machine = 386 */
        mach = &mi386;
    if (!machdata)
        machdata = &i386mach;
    return dumb;
}
```
Uses i386mach, mach 51a, machdata 355e, mi386, newmap() 323a, and setmap() 323b.

⟨setsym() *locals* 248c⟩≡ (247e)
```
Symbol s;
```

⟨setsym() *if mach has sbreg* 248d⟩≡ (247e)
```
if (mach->sbreg && lookup(0, mach->sbreg, &s))
    mach->sb = s.value;
```
Uses lookup() 331 and mach 51a.

⟨*global* wtflag 248e⟩≡ (429b)
```
// was in main.c
int wtflag = OREAD;
```

⟨*function* getfile 248f⟩≡ (432)
```
static fdt
getfile(char *filnam, int cnt, int omode)
{
    fdt f;

    if (filnam == nil)
        return ERROR_NEG1;

    if (strcmp(filnam, "-") == 0)
        return STDIN;
    f = open(filnam, omode|OCEXEC);

    ⟨getfile() error managment 248g⟩
    return f;
}
```

⟨getfile() *error managment* 248g⟩≡ (248f)
```
if(f < 0 && omode == ORDWR){
    f = open(filnam, OREAD|OCEXEC);
    if(f >= 0)
        dprint("%s open read-only\n", filnam);
}
⟨getfile() if wtflag 298b⟩
if (f < 0) {
```

```
    dprint("cannot open ‘%s’: %r\n", filnam);
    return ERROR_NEG1;
}
```

Uses dprint() 250e.

E.3.4 Memory file: setcor()

⟨main() *if setjmp == 0 and corfil* 249a⟩≡ (246e)

```
if (corfil) {
    setcor(); /* could get error */
    dprint("%s\n", machdata->excep(cormap, rget));
    printpc();
}
```

⟨*global* fcor 249b⟩≡ (432)

```
fdt fcor;
```

⟨*function* setcor 249c⟩≡ (432)

```
void
setcor(void)
{
    int i;

    ⟨setcor() free previous cormap 249d⟩
    fcor = getfile(corfil, 2, ORDWR);
    ⟨setcor() error managment getfile 249e⟩
    if(pid > 0) { /* provide addressability to executing process */
        cormap = attachproc(pid, kflag, fcor, &fhdr);
        ⟨setcor() error managment cormap 250a⟩
    } else {
        cormap = newmap(cormap, 2);
        ⟨setcor() error managment cormap 250a⟩
        setmap(cormap, fcor, fhdr.txtaddr, fhdr.txtaddr+fhdr.txtsz, fhdr.txtaddr, "text");
        setmap(cormap, fcor, fhdr.dataddr, 0xffffffff, fhdr.dataddr, "data");
    }
    kmsys();
    return;
}
```

Uses attachproc() 96b, corfil 245b, fcor 249b, fhdr-68 247c, kflag 299a, kmsys() 299f, newmap() 323a, pid 246f, and setmap() 323b.

⟨setcor() *free previous cormap* 249d⟩≡ (249c)

```
if (cormap) {
    for (i = 0; i < cormap->nsegs; i++)
        if (cormap->seg[i].inuse)
            close(cormap->seg[i].fd);
}
```

⟨setcor() *error managment getfile* 249e⟩≡ (249c)

```
if (fcor <= 0) {
    if (cormap)
        free(cormap);
    cormap = dumbmap(-1);
    return;
}
```

Uses dumbmap() 248b and fcor 249b.

⟨setcor() *error managment cormap* 250a⟩≡ (249)
```
if (!cormap)
    cormap = dumbmap(-1);
```
Uses dumbmap() 248b.

E.3.5 Output: outputinit(), dprint() and stdout

⟨*global* stdout 250b⟩≡ (430)
```
Biobuf stdout;
```

⟨*function* outputinit 250c⟩≡ (430)
```
void
outputinit(void)
{
    Binit(&stdout, 1, OWRITE);
    fmtinstall('t', tconv);
}
```
Uses stdout 250b and tconv() 250d.

⟨*function* tconv 250d⟩≡ (430)
```
/* was move to next f1-sized tab stop; now just print a tab */
int
tconv(Fmt *f)
{
    return fmtstrcpy(f, "\t");
}
```

⟨*function* dprint 250e⟩≡ (430)
```
int
dprint(char *fmt, ...)
{
    char buf[4096];
    va_list arg;
    int n;
    ⟨dprint() locals 251c⟩

    ⟨dprint() return if mkfault 301b⟩

    va_start(arg, fmt);
    n = vseprint(buf, buf+sizeof buf, fmt, arg) - buf;
    va_end(arg);

    //Bprint(&stdout, "[%s]", fmt);
    Bwrite(&stdout, buf, n);

    ⟨dprint() maintain printcol 251d⟩
    return n;
}
```
Uses stdout 250b.

⟨*function* printc 250f⟩≡ (430)
```
void
printc(int c)
{
    dprint("%c", c);
}
```
Uses dprint() 250e.

⟨*function* `prints` 251a⟩≡ (430)
```
void
prints(char *s)
{
    dprint("%s",s);
}
```
Uses `dprint()` 250e.

⟨*global* `printcol` 251b⟩≡ (430)
```
int printcol = 0;
```
Uses `printcol` 251b.

⟨`dprint()` *locals* 251c⟩≡ (250e)
```
char *p;
Rune r;
int w;
```

⟨`dprint()` *maintain printcol* 251d⟩≡ (250e)
```
for(p=buf; *p; p+=w){
    w = chartorune(&r, p);
    if(r == '\n')
        printcol = 0;
    else
        printcol++;
}
```
Uses `printcol` 251b.

⟨*function* `flushbuf` 251e⟩≡ (430)
```
void
flushbuf(void)
{
  if (printcol != 0)
        printc(EOR);
}
```
Uses `EOR` 251g, `printc()` 250f, and `printcol` 251b.

⟨*function* `flush` 251f⟩≡ (430)
```
void
flush(void)
{
    Bflush(&stdout);
}
```
Uses `stdout` 250b.

E.3.6 Input: `clrinp()`, `rdc()`, `reread()`

⟨*constant* `EOR` 251g⟩≡ (426c)
```
#define EOR '\n'
```

⟨*constant* `SPC` 251h⟩≡ (426c)
```
#define SPC ' '
```

⟨*constant* `TB` 251i⟩≡ (426c)
```
#define TB '\t'
```

⟨*global* `lp` 251j⟩≡ (431b)
```
Rune *lp;
```

⟨*global* peekc 252a⟩≡ (431b)
```
int peekc;
```

⟨*global* lastc 252b⟩≡ (431b)
```
int lastc = EOR;
```
Uses EOR 251g and lastc 252b.

⟨*function* clrinp 252c⟩≡ (431b)
```
void
clrinp(void)
{
    flush();
    lp = nil;
    peekc = 0;
}
```
Uses flush() 251f, lp 251j, and peekc 252a.

⟨*function* rdc 252d⟩≡ (431b)
```
int
rdc(void)
{
    do {
        readchar();
    } while (lastc==SPC || lastc==TB);
    return lastc;
}
```
Uses SPC 251h, TB 251i, lastc 252b, and readchar() 255c.

⟨*function* reread 252e⟩≡ (431b)
```
void
reread(void)
{
    peekc = lastc;
}
```
Uses lastc 252b and peekc 252a.

E.3.7 Interpreter: `command()`

`command()`[253a] is `db`'s equivalent of `acid`'s `yyparse`+`execute`—it parses and executes one command line in the `adb` grammar (*address, count, command*). The parsing is entirely manual: `expr()` parses the optional address, `rdc()`[252d] checks for a comma and count, and a `switch` dispatches on the command character. A concrete example:

```
Input:  main+4,3?i

expr(0)  -> parses "main+4", sets dot = main+4
rdc()    -> sees ',', expr(0) -> parses "3", sets cntval = 3
rdc()    -> reads '?', sets lastcom = '?'
switch:  case '?' -> acommand('?')
                  -> getformat("i")
                  -> scanform(3, ...) -> exform() x3
                     (disassembles 3 instructions from main+4)
```

Note the `do ... while(rdc()==';')` loop: commands can be chained with semicolons on one line, e.g., `main:b; :r`.

⟨*function* `command` 253a⟩≡ (442c)
```
/* command decoding */
int
command(char *buf, int defcom)
{
    ⟨command() locals (db) 253b⟩

    ⟨command() initializations (db) 254c⟩

    do {
        ⟨command() parse possibly address, set dot 253f⟩
        ⟨command() parse possibly count, set cntval 254a⟩
        ⟨command() parse possibly command, set lastcom 254b⟩

        switch(lastcom) {
        ⟨command() switch lastcom cases 261b⟩
        default:
            error("bad command");
        }
        flushbuf();
    } while (rdc()==';');

    ⟨command() finalizations (db) 295c⟩

    ⟨command() return (db) 282a⟩
}
```
Uses `flushbuf()` 251e and `rdc()` 252d.

⟨`command()` *locals (db)* 253b⟩≡ (253a) 253c▷
```
static char lastcom = '=';
```

⟨`command()` *locals (db)* 253c⟩+≡ (253a) ◁253b 292b▷
```
static char savecom = '=';
```

Addresses

⟨*global* `adrflg` 253d⟩≡ (429b)
```
bool adrflg;
```

⟨*global* `adrval` 253e⟩≡ (429b)
```
// was in command.c
WORD adrval;
```

⟨`command()` *parse possibly address, set dot* 253f⟩≡ (253a)
```
adrflg=expr(0);  /* first address */
if (adrflg){
    dot=expv;
    ditto=expv;
}
adrval=dot;
```
Uses `adrflg` 253d, `adrval` 253e, `ditto` 261a, `dot` 242e, and `expv` 257a.

Counts

⟨*global* `cntflg` 253g⟩≡ (429b)
```
bool cntflg;
```

⟨command() *parse possibly count, set cntval* 254a⟩≡ (253a)
```
if (rdc()==',' && expr(0)) { /* count */
    cntflg=TRUE;
    cntval=expv;
} else {
    cntflg=FALSE;
    cntval=1;
    reread();
}
```
Uses cntflg 253g, cntval 243a, expv 257a, rdc() 252d, and reread() 252e.

Commands

⟨command() *parse possibly command, set lastcom* 254b⟩≡ (253a)
```
if (!eol(rdc()))
    lastcom=lastc;  /* command */
else {
    if (adrflg==false)
        dot=inkdot(dotinc);
    lastcom=defcom;
    reread();
}
```
Uses adrflg 253d, dot 242e, dotinc 271b, eol() 254d, inkdot() 254e, lastc 252b, rdc() 252d, and reread() 252e.

⟨command() *initializations (db)* 254c⟩≡ (253a) 295a▷
```
if (defcom == 0)
    defcom = lastcom;
```

⟨*function* eol 254d⟩≡ (431b)
```
bool
eol(int c)
{
    return(c==EOR || c==';');
}
```
Uses EOR 251g.

⟨*function* inkdot 254e⟩≡ (433a)
```
ADDR
inkdot(int incr)
{
    ADDR newdot;

    newdot = dot+incr;
    ⟨inkdot() error managment 254f⟩
    return newdot;
}
```
Uses dot 242e.

⟨inkdot() *error managment* 254f⟩≡ (254e)
```
if ((incr >= 0 && newdot < dot)
||  (incr < 0 && newdot > dot))
    error("address wraparound");
```
Uses dot 242e.

E.3.8 Exit: done()

⟨*global* eof 254g⟩≡ (431b)
```
bool eof;
```

⟨main() *in loop, if eof (db)* 255a⟩≡ (245d)
```
if (eof) {
    if (infile == STDIN)
        done(); // will exits()
    ⟨main() in loop, if eof, and if infile was not STDIN 297d⟩
}
```

⟨*function* done 255b⟩≡ (444)
```
void
done(void)
{
    if (pid)
        endpcs();
    exits(nil);
}
```
Uses endpcs() 287d and pid 246f.

E.4 Command Input

db's input handling is a small hand-written lexer made of three functions and four globals. readchar()[255c] produces the next input character, rdc()[252d] skips whitespace and returns the next significant character, and reread()[252e] pushes one character back into the input stream for one-character lookahead. The shared state is four globals: lp[251j] points into the current line buffer, peekc[252a] is the pushback slot used by reread[252e], lastc[252b] holds the last character readchar[255c] returned, and line[70b] is a 4096-rune buffer that gets refilled one line at a time. There is no separate token layer—every command handler consumes characters directly out of the stream, which is why the adb grammar can be single-character (?, ,, $) without any keyword framing.

⟨*function* readchar 255c⟩≡ (431b)
```
int
readchar(void)
{
    Rune *p;

    if (eof) {
        lastc='\0';
    } else if (peekc) {
        lastc = peekc;
        peekc = 0;
    } else {
        ⟨readchar() if lp is nil read a line and set lp 256a⟩
        lastc = *lp;
        if (lastc != '\0')
            lp++;
    }
    return lastc;
}
```
Uses eof 254g, lastc 252b, lp 251j, and peekc 252a.

⟨*global* line 255d⟩≡ (431b)
```
Rune line[LINSIZ];
```
Uses LINSIZ 255e.

⟨*constant* LINSIZ 255e⟩≡ (426c)
```
#define LINSIZ 4096
```

⟨readchar() *if lp is nil read a line and set lp* 256a⟩≡ (255c)

```
if (lp==nil) {
    for (p = line; p < &line[LINSIZ-1]; p++) {
        eof = (readrune(infile, p) <= 0);
        ⟨readchar() if mkfault 301c⟩
        if (eof) {
            p--;
            break;
        }
        if (*p == EOR) {
            if (p <= line)
                break;
            if (p[-1] != '\\')
                break;
            p -= 2;
        }
    }
    p[1] = '\0';
    lp = line;
}
```

Uses EOR 251g, LINSIZ 255e, eof 254g, infile 245c, lp 251j, and readrune() 256b.

⟨*function* readrune 256b⟩≡ (431b)

```
int
readrune(int fd, Rune *r)
{
    char buf[UTFmax+1];
    int i;

    for(i=0; i<UTFmax && !fullrune(buf, i); i++)
        if(read(fd, buf+i, 1) <= 0)
            return -1; // EOF
    buf[i] = '\0';
    chartorune(r, buf);
    return 1;
}
```

⟨*function* nextchar 256c⟩≡ (431b)

```
int
nextchar(void)
{
    if (eol(rdc())) {
        reread();
        return 0;
    }
    return lastc;
}
```

Uses eol() 254d, lastc 252b, rdc() 252d, and reread() 252e.

E.5 Command Parsing

The expression parser (expr → term[258] → item[259]) is a hand-written recursive descent parser for adb's address expressions. It supports arithmetic (+, -, *, etc.), symbol lookup (main, main.x), register access (<PC), and memory dereference (*addr for cormap, @addr for symmap). Both acid and db have the same * and @ operators with the same semantics: * reads through cormap (live process memory) and @ reads through symmap (the executable file). The real difference is in how most users interact with them: acid's library functions like stk()

and `src()` hide the map choice behind higher-level abstractions, while in `db` you choose `?` or `/` explicitly for every inspection command.

A worked example of address parsing:

```
Input: main.x+8

item():
  "main" -> lookup(0, "main", &s) -> s.value = 0x1020
  '.'    -> "x"
  localaddr(cormap, "main", "x", &e) -> e = 0x1020 + frame - 4
  expv = 0x103c (say)

expr():
  lhs = 0x103c
  '+' -> term() -> "8" -> expv = 8
  expv = 0x103c + 8 = 0x1044
```

⟨*global* `expv` 257a⟩≡ (429b)

```
// was in expr.c
uvlong expv;
```

E.5.1 `expr()`

⟨*function* `expr` 257b⟩≡ (439a)

```
bool
expr(int a)
{ /* term | term dyadic expr |  */
    bool rc;
    WORD lhs;

    rdc();
    reread();

    rc=term(a);
    while (rc) {
        lhs = expv;
        switch ((int)readchar()) {

        case '+':
            term(a|1);
            expv += lhs;
            break;

        case '-':
            term(a|1);
            expv = lhs - expv;
            break;

        case '#':
            term(a|1);
            expv = round(lhs,expv);
            break;

        case '*':
            term(a|1);
            expv *= lhs;
```

```
            break;

        case '%':
            term(a|1);
            if(expv != 0)
                expv = lhs/expv;
            else{
                if(lhs)
                    expv = 1;
                else
                    expv = 0;
            }
            break;

        case '&':
            term(a|1);
            expv &= lhs;
            break;

        case '|':
            term(a|1);
            expv |= lhs;
            break;

        case ')':
            if ((a&2)==0)
                error("unexpected `)'");

        default:
            reread();
            return rc;
        }
    }
    return rc;
}
```

Uses expv 257a, rdc() 252d, readchar() 255c, reread() 252e, round() 438d, and term() 258.

E.5.2 term()

⟨*function* term 258⟩≡ (439a)

```
bool
term(int a)
{ /* item | monadic item | (expr) | */
    ADDR e;

    switch ((int)readchar()) {

    case '*':
        term(a|1);
        if (geta(cormap, expv, &e) < 0)
            error("%r");
        expv = e;
        return true;

    case '@':
        term(a|1);
        if (geta(symmap, expv, &e) < 0)
            error("%r");
        expv = e;
```

```
        return true;

    case '-':
        term(a|1);
        expv = -expv;
        return true;

    case '~':
        term(a|1);
        expv = ~expv;
        return true;

    case '(':
        expr(2);
        if (readchar()!=')')
            error("syntax error: `)' expected");
        return true;

    default:
        reread();
        return item(a);
    }
}
```

Uses expv 257a, geta() 349, item() 259, readchar() 255c, reread() 252e, and term() 258.

E.5.3 item()

⟨*function* item 259⟩≡ (439a)

```
bool
item(int a)
{ /* name [ . local ] | number | . | ^  | <register | 'x | | */
    char *base;
    char savc;
    uvlong e;
    Symbol s;
    char gsym[MAXSYM], lsym[MAXSYM];

    readchar();

    if (isfileref()) {
        readfname(gsym);
        rdc();   /* skip white space */
        if (lastc == ':') { /* it better be */
            rdc();  /* skip white space */
            if (!getnum(readchar))
                error("bad number");
            if (expv == 0)
                expv = 1; /* file begins at line 1 */
            expv = file2pc(gsym, expv);
            if (expv == -1)
                error("%r");
            return true;
        }
        error("bad file location");
    } else if (symchar(0)) {
        readsym(gsym);
        if (lastc=='.') {
            readchar(); /* ugh */
            if (lastc == '.') {
```

```
                lsym[0] = '.';
                readchar();
                readsym(lsym+1);
            } else if (symchar(0)) {
                readsym(lsym);
            } else
                lsym[0] = 0;
            if (localaddr(cormap, gsym, lsym, &e, rget) < 0)
                error("%r");
            expv = e;
        }
        else {
            if (lookup(0, gsym, &s) == 0)
                error("symbol not found");
            expv = s.value;
        }
        reread();
    } else if (getnum(readchar)) {
        ;
    } else if (lastc=='.') {
        readchar();
        if (!symchar(0) && lastc != '.') {
            expv = dot;
        } else {
            if (findsym(rget(cormap, mach->pc), CTEXT, &s) == 0)
                error("no current function");
            if (lastc == '.') {
                lsym[0] = '.';
                readchar();
                readsym(lsym+1);
            } else
                readsym(lsym);
            if (localaddr(cormap, s.name, lsym, &e, rget) < 0)
                error("%r");
            expv = e;
        }
        reread();
    } else if (lastc=='"') {
        expv=ditto;
    } else if (lastc=='+') {
        expv=inkdot(dotinc);
    } else if (lastc=='^') {
        expv=inkdot(-dotinc);
    } else if (lastc=='<') {
        savc=rdc();
        base = regname(savc);
        expv = rget(cormap, base);
    }
    else if (lastc=='\'')
        expv = ascval();
    else if (a)
        error("address expected");
    else {
        reread();
        return false;
    }
    return true;
}
```

Uses MAXSYM 437a, ascval() 435b, ditto 261a, dot 242e, dotinc 271b, expv 257a, file2pc() 337b, findsym() 335b,

getnum() 436e, inkdot() 254e, isfileref() 431a, lastc 252b, localaddr() 355f, lookup() 331, mach 51a, rdc() 252d, readchar() 255c, readfname() 438a, readsym() 437b, regname() 442b, reread() 252e, and symchar() 438c.

⟨*global* ditto 261a⟩≡ (429b)
```
ADDR ditto;
```

E.6 Dumper commands: $

⟨command() *switch lastcom cases* 261b⟩≡ (253a) 267b▷
```
case '$':
    lastcom=savecom;
    printtrace(nextchar());
    break;
```
Uses nextchar() 256c and printtrace() 261c.

⟨*function* printtrace 261c⟩≡ (441b)
```
void
printtrace(int modif)
{
    ⟨printtrace() locals 261d⟩

    if (cntflg==FALSE)
        cntval = -1;

    switch (modif) {
    ⟨printtrace() switch modif cases 261e⟩
    default:
        error("bad '$' command");
    }

}
```
Uses cntflg 253g and cntval 243a.

⟨printtrace() *locals* 261d⟩≡ (261c) 263b▷
```
int i;
ulong w;
BKPT *bk;
Symbol s;
int stack;
char *fname;
char buf[512];
```

E.6.1 Current process: $?

⟨printtrace() *switch modif cases* 261e⟩≡ (261c) 262a▷
```
case 0:
case '?':
    if (pid)
        dprint("pid = %d\n",pid);
    else
        prints("no process\n");
    flushbuf();
```
Uses dprint() 250e, flushbuf() 251e, pid 246f, and prints() 251a.

E.6.2 Address maps: $m

⟨printtrace() *switch modif cases* 262a⟩+≡ (261c) ◁261e 262c▷
```
case 'm':
    printmap("? map", symmap);
    printmap("/ map", cormap);
    break;
```
Uses printmap() 262b.

⟨*function* printmap 262b⟩≡ (441b)
```
void
printmap(char *s, Map *map)
{
    int i;

    if (!map)
        return;
    if (map == symmap)
        dprint("%s%12t‘%s’\n", s, fsym < 0 ? "-" : symfil);
    else if (map == cormap)
        dprint("%s%12t‘%s’\n", s, fcor < 0 ? "-" : corfil);
    else
        dprint("%s\n", s);

    for (i = 0; i < map->nsegs; i++) {
        if (map->seg[i].inuse)
            dprint("%s%8t%-16#llux %-16#llux %-16#llux\n",
                map->seg[i].name,
                map->seg[i].b,
                map->seg[i].e,
                map->seg[i].f);
    }
}
```
Uses corfil 245b, dprint() 250e, fcor 249b, fsym 247d, and symfil 245a.

E.6.3 Symbols: $S and $e

⟨printtrace() *switch modif cases* 262c⟩+≡ (261c) ◁262a 263a▷
```
case 'S':
    printsym();
    break;
```
Uses printsym() 262d.

⟨*function* printsym 262d⟩≡ (441b)
```
/*
 * dump the raw symbol table
 */
void
printsym(void)
{
    int i;
    Sym *sp;

    for (i = 0; sp = getsym(i); i++) {
        switch(sp->type) {
        case 't':
        case 'l':
            dprint("%16#llux t %s\n", sp->value, sp->name);
            break;
```

```
            case 'T':
            case 'L':
                dprint("%16#llux T %s\n", sp->value, sp->name);
                break;
            case 'D':
            case 'd':
            case 'B':
            case 'b':
            case 'a':
            case 'p':
            case 'm':
                dprint("%16#llux %c %s\n", sp->value, sp->type, sp->name);
                break;
            default:
                break;
            }
        }
    }
```

Uses `dprint()` 250e and `getsym()` 328b.

⟨`printtrace()` *switch modif cases* 263a⟩+≡ (261c) ◁262c 263d▷

```
    /*print externals*/
    case 'e':
        for (i = 0; globalsym(&s, i); i++) {
            if (get4(cormap, s.value, &w) > 0)
                dprint("%s/%12t%#lux\n", s.name, w);
        }
        break;
```

Uses `dprint()` 250e, `get4()` 350b, and `globalsym()` 337a.

E.6.4 Stack traces: $c

⟨`printtrace()` *locals* 263b⟩+≡ (261c) ◁261d

```
    uvlong pc, sp, link;
```

⟨*global* `tracetype` 263c⟩≡ (441b)

```
    static int tracetype;
```

⟨`printtrace()` *switch modif cases* 263d⟩+≡ (261c) ◁263a 264b▷

```
    case 'c':
    case 'C':
        tracetype = modif;
        if (machdata->ctrace) {
            if (adrflg) {
                /*
                 * trace from jmpbuf for multi-threaded code.
                 * assume sp and pc are in adjacent locations
                 * and mach->szaddr in size.
                 */
                if (geta(cormap, adrval, &sp) < 0 ||
                    geta(cormap, adrval+mach->szaddr, &pc) < 0)
                        error("%r");
            } else {
                sp = rget(cormap, mach->sp);
                pc = rget(cormap, mach->pc);
            }
            if(mach->link)
                link = rget(cormap, mach->link);
            else
```

```
            link = 0;
        if (machdata->ctrace(cormap, pc, sp, link, ptrace) <= 0)
            error("no stack frame");
    }
    break;
```

Uses adrflg 253d, adrval 253e, geta() 349, mach 51a, machdata 355e, ptrace() 264a, and tracetype-71 263c.

⟨*function* ptrace 264a⟩≡ (441b)

```
/*
 * callback on stack trace
 */
static void
ptrace(Map *map, uvlong pc, uvlong sp, Symbol *sym)
{
    char buf[512];

    USED(map);
    dprint("%s(", sym->name);
    printparams(sym, sp);
    dprint(") ");
    printsource(sym->value);
    dprint(" called from ");
    symoff(buf, 512, pc, CTEXT);
    dprint("%s ", buf);
    printsource(pc);
    dprint("\n");

    if(tracetype == 'C')
        printlocals(sym, sp);
}
```

Uses dprint() 250e, printlocals() 294a, printparams() 293b, printsource() 281b, symoff() 356a, and tracetype-71 263c.

E.6.5 Registers: $r and $f

⟨printtrace() *switch modif cases* 264b⟩+≡ (261c) ◁263d 265a▷

```
case 'r':
case 'R':
    printregs(modif);
    return;
```

Uses printregs() 264c.

⟨*function* printregs 264c⟩≡ (435a)

```
/*
 * print the registers
 */
void
printregs(int c)
{
    Reglist *rp;
    int i;
    uvlong v;

    for (i = 1, rp = mach->reglist; rp->rname; rp++, i++) {
        if ((rp->rflags & RFLT)) {
            if (c != 'R')
                continue;
            if (rp->rformat == '8' || rp->rformat == '3')
                continue;
```

```
            }
            v = getreg(cormap, rp);
            if(rp->rformat == 'Y')
                dprint("%-8s %-20#llux", rp->rname, v);
            else
                dprint("%-8s %-12#lux", rp->rname, (ulong)v);
            if ((i % 3) == 0) {
                dprint("\n");
                i = 0;
            }
        }
        if (i != 1)
            dprint("\n");
        dprint ("%s\n", machdata->excep(cormap, rget));
        printpc();
    }
```

Uses dprint() 250e, getreg() 433c, mach 51a, machdata 355e, and printpc() 280b.

⟨printtrace() *switch modif cases* 265a⟩+≡ (261c) ◁264b 265c▷

```
    case 'f':
    case 'F':
        printfp(cormap, modif);
        return;
```

Uses printfp() 265b.

⟨*function* printfp 265b⟩≡ (441b)

```
    static void
    printfp(Map *map, int modif)
    {
        Reglist *rp;
        int i;
        int ret;
        char buf[512];

        for (i = 0, rp = mach->reglist; rp->rname; rp += ret) {
            ret = 1;
            if (!(rp->rflags & RFLT))
                continue;
            ret = fpformat(map, rp, buf, sizeof(buf), modif);
            if (ret < 0) {
                werrstr("Register %s: %r", rp->rname);
                error("%r");
            }
                /* double column print */
            if (i&0x01)
                dprint("%40t%-8s%-12s\n", rp->rname, buf);
            else
                dprint("\t%-8s%-12s", rp->rname, buf);
            i++;
        }
    }
```

Uses dprint() 250e, fpformat() 356b, and mach 51a.

E.6.6 Quitting: $q

⟨printtrace() *switch modif cases* 265c⟩+≡ (261c) ◁265a 266a▷

```
    case 'q':
    case 'Q':
```

```
    done();
```
Uses done() 255b.

E.6.7 Other commands: $a, $s, $m

⟨printtrace() *switch modif cases* 266a⟩+≡ (261c) ◁265c 266f▷
```
  case 'a':
      attachprocess();
      break;
```
Uses attachprocess() 266b.

⟨*function* attachprocess 266b⟩≡ (432)
```
  void
  attachprocess(void)
  {
      char buf[100];
      Dir *sym, *mem;
      int fd;

      if (!adrflg) {
          dprint("used pid$a\n");
          return;
      }
      sym = dirfstat(fsym);
      sprint(buf, "/proc/%lud/mem", adrval);
      corfil = buf;
      setcor();
      sprint(buf, "/proc/%lud/text", adrval);
      fd = open(buf, OREAD);
      ⟨attachprocess() error managment 266c⟩
      if (fd >= 0)
          close(fd);
  }
```
Uses adrflg 253d, adrval 253e, corfil 245b, dprint() 250e, fsym 247d, and setcor() 249c.

⟨attachprocess() *error managment* 266c⟩≡ (266b)
```
  mem = nil;
  if (sym==nil || fd < 0 || (mem=dirfstat(fd))==nil
                || sym->qid.path != mem->qid.path)
      dprint("warning: text images may be inconsistent\n");
  free(sym);
  free(mem);
```
Uses dprint() 250e.

⟨*constant* MAXOFF 266d⟩≡ (426c)
```
  #define MAXOFF 0x1000000
```

⟨*global* maxoff 266e⟩≡ (429b)
```
  ADDR maxoff = MAXOFF;
```
Uses MAXOFF 266d and maxoff 266e.

⟨printtrace() *switch modif cases* 266f⟩+≡ (261c) ◁266a 267a▷
```
  case 's':
      maxoff=(adrflg?adrval:MAXOFF);
      break;
```
Uses MAXOFF 266d, adrflg 253d, adrval 253e, and maxoff 266e.

⟨printtrace() *switch modif cases* 267a⟩+≡ (261c) ◁266f 270d▷

```
case 'M':
    fname = getfname();
    if (machbyname(fname) == 0)
        dprint("unknown name\n");;
    break;
```

Uses dprint() 250e, getfname() 440c, and machbyname() 370c.

E.7 Inspecting commands, ?/=

The inspection commands (?, /, =) are db's main output mechanism. Each reads memory (or an expression value) and formats it according to a format string—a sequence of single-character format specifiers like i (instruction), X (hex word), s (string), c (char). The default format is "zMi": print the source location (z), the hex dump of the instruction (M), and the disassembly (i). The format system is a mini-language of its own:

```
main,5?i        disassemble 5 instructions from main
main?X          print 4-byte hex word at main
main?s          print null-terminated string at main
main?3XnXn      print 3 hex words, newline, 1 more, newline
main=X          print main's ADDRESS in hex (no memory read)
```

acid has no equivalent of this format system. Instead, you use format suffixes on expressions: main\X, *main\i, or call library functions like asm() and mem().

⟨command() *switch lastcom cases* 267b⟩+≡ (253a) ◁261b 279b▷

```
case '?':
case '/':
case '=':
    savecom = lastcom;
    acommand(lastcom);
    break;
```

Uses acommand() 267c.

⟨*function* acommand 267c⟩≡ (442c)

```
/*
 * [/?][wml]
 */
void
acommand(int pc)
{
    bool eqcom;
    Map *map;
    char *fmt;
    char buf[512];

    if (pc == '=') {
        eqcom = true;
        fmt = eqformat;
        map = dotmap;
    } else {
        eqcom = false;
        fmt = stformat;
        if (pc == '/')
            map = cormap;
        else
            map = symmap;
```

```
    }
    if (!map) {
        snprint(buf, sizeof(buf), "no map for %c", pc);
        error(buf);
    }

    switch (rdc()) {
    ⟨acommand() switch optional command suffix character 276f⟩
    default:
        reread();
        getformat(fmt);
        scanform(cntval, !eqcom, fmt, map, eqcom);
    }
}
```

Uses cntval 243a, dotmap 268d, eqformat 268b, getformat() 268f, rdc() 252d, reread() 252e, scanform() 269a, and stformat 268c.

E.7.1 Formats

⟨*constant* ARB 268a⟩≡ (426c)
```
#define ARB 512
```

⟨*global* eqformat 268b⟩≡ (442c)
```
char eqformat[ARB] = "z";
```

Uses ARB 268a and eqformat 268b.

⟨*global* stformat 268c⟩≡ (442c)
```
char stformat[ARB] = "zMi";
```

Uses ARB 268a and stformat 268c.

⟨*global* dotmap 268d⟩≡ (432)
```
Map *dotmap;
```

⟨main() *set dotmap* 268e⟩≡ (246b)
```
dotmap = dumbmap(-1);
```

⟨*function* getformat 268f⟩≡ (431b)
```
void
getformat(char *deformat)
{
    char *fptr;
    bool quote;
    Rune r;

    fptr=deformat;
    quote=FALSE;
    while ((quote ? readchar()!=EOR : !eol(readchar()))){
        r = lastc;
        fptr += runetochar(fptr, &r);
        if (lastc == '"')
            quote = ~quote;
    }
    lp--;
    if (fptr!=deformat)
        *fptr = '\0';
}
```

Uses EOR 251g, eol() 254d, lastc 252b, lp 251j, and readchar() 255c.

⟨*function* scanform 269a⟩≡ (433a)

```
void
scanform(long icount, int prt, char *ifp, Map *map, int literal)
{
    char *fp;
    char c;
    int fcount;
    ADDR savdot;
    bool firstpass = true;

    while (icount) {
        fp=ifp;
        savdot=dot;

        /*now loop over format*/
        while (*fp) {
            if (!isdigit(*fp))
                fcount = 1;
            else {
                fcount = 0;
                while (isdigit(c = *fp++)) {
                    fcount *= 10;
                    fcount += c-'0';
                }
                fp--;
            }
            if (*fp==0)
                break;
            fp=exform(fcount,prt,fp,map,literal,firstpass);
            firstpass = false;
        }
        dotinc=dot-savdot;
        dot=savdot;
        if (--icount)
            dot=inkdot(dotinc);
    }
}
```

Uses `dot` 242e, `dotinc` 271b, `exform()` 269b, and `inkdot()` 254e.

⟨*function* exform 269b⟩≡ (433a)

```
char *
exform(int fcount, int prt, char *ifp, Map *map, int literal, bool firstpass)
{
    /* execute single format item 'fcount' times
     * sets 'dotinc' and moves 'dot'
     * returns address of next format item
     */
    uvlong v;
    ulong w;
    ADDR savdot;
    char *fp;
    char c, modifier;
    int i;
    ushort sh, *sp;
    uchar ch, *cp;
    Symbol s;
    char buf[512];
    extern int printcol;

    fp = 0;
```

```
    while (fcount > 0) {
        fp = ifp;
        c = *fp;
        modifier = *fp++;
        if (firstpass) {
            firstpass = false;
            if (!literal  && (c == 'i' || c == 'I' || c == 'M')
                    && (dot & (mach->pcquant-1))) {
                dprint("warning: instruction not aligned");
                printc('\n');
            }
            if (prt && modifier != 'a' && modifier != 'A') {
                symoff(buf, 512, dot, CANY);
                dprint("%s%c%16t", buf, map==symmap? '?':'/');
            }
        }
        if (printcol==0 && modifier != 'a' && modifier != 'A')
            dprint("\t\t");

        switch(modifier) {
        ⟨exform() switch modifier cases 271a⟩
        default:
            error("bad modifier");
        }

        if (map->seg[0].fd >= 0)
            dot=inkdot(dotinc);
        fcount--;
        endline();
    }

    return fp;
}
```

Uses dot 242e, dotinc 271b, dprint() 250e, endline() 270a, inkdot() 254e, mach 51a, printc() 250f, printcol 251b, and symoff() 356a.

⟨*function* endline 270a⟩≡ (430)

```
void
endline(void)
{

    if (printcol >= maxpos)
        newline();
}
```

Uses maxpos 270c, newline() 279a, and printcol 251b.

⟨*constant* MAXPOS 270b⟩≡ (426c)

```
#define MAXPOS 80
```

⟨*global* maxpos 270c⟩≡ (430)

```
int maxpos = MAXPOS;
```

Uses MAXPOS 270b and maxpos 270c.

⟨printtrace() *switch modif cases* 270d⟩+≡ (261c) ◁267a 289c▷

```
case 'w':
    maxpos=(adrflg?adrval:MAXPOS);
    break;
```

Uses MAXPOS 270b, adrflg 253d, adrval 253e, and maxpos 270c.

E.7.2 Instruction disassembling: ?i

⟨exform() *switch modifier cases* 271a⟩≡ (269b) 271c▷
```
case 'I':
case 'i':
    i = machdata->das(map, dot, modifier, buf, sizeof(buf));
    if (i < 0)
        error("%r");
    dotinc = i;
    dprint("%s\n", buf);
    break;
```
Uses dot 242e, dotinc 271b, dprint() 250e, and machdata 355e.

⟨*global* dotinc 271b⟩≡ (429b)
```
int dotinc;
```

E.7.3 Other formats: ?t, ?a, ?p, etc.

⟨exform() *switch modifier cases* 271c⟩+≡ (269b) ◁271a 271d▷
```
case SPC:
case TB:
    dotinc = 0;
    break;
```
Uses SPC 251h, TB 251i, and dotinc 271b.

⟨exform() *switch modifier cases* 271d⟩+≡ (269b) ◁271c 271e▷
```
case 't':
case 'T':
    dprint("%*t", fcount);
    dotinc = 0;
    return(fp);
```
Uses dotinc 271b and dprint() 250e.

⟨exform() *switch modifier cases* 271e⟩+≡ (269b) ◁271d 271f▷
```
case 'a':
    symoff(buf, sizeof(buf), dot, CANY);
    dprint("%s%c%16t", buf, map==symmap? '?':'/');
    dotinc = 0;
    break;
```
Uses dot 242e, dotinc 271b, dprint() 250e, and symoff() 356a.

⟨exform() *switch modifier cases* 271f⟩+≡ (269b) ◁271e 271g▷
```
case 'A':
    dprint("%#llux%10t", dot);
    dotinc = 0;
    break;
```
Uses dot 242e, dotinc 271b, and dprint() 250e.

⟨exform() *switch modifier cases* 271g⟩+≡ (269b) ◁271f 272a▷
```
case 'p':
    if (get4(map, dot, &w) < 0)
        error("%r");
    symoff(buf, sizeof(buf), w, CANY);
    dprint("%s%16t", buf);
    dotinc = mach->szaddr;
    break;
```
Uses dot 242e, dotinc 271b, dprint() 250e, get4() 350b, mach 51a, and symoff() 356a.

⟨exform() *switch modifier cases* 272a⟩+≡ (269b) ◁271g 272b▷
```
  case 'u':
  case 'd':
  case 'x':
  case 'o':
  case 'q':
      if (literal)
          sh = (ushort) dot;
      else if (get2(map, dot, &sh) < 0)
          error("%r");
      w = sh;
      dotinc = 2;
      if (c == 'u')
          dprint("%-8lud", w);
      else if (c == 'x')
          dprint("%-8#lux", w);
      else if (c == 'd')
          dprint("%-8ld", w);
      else if (c == 'o')
          dprint("%-8#luo", w);
      else if (c == 'q')
          dprint("%-8#lo", w);
      break;
```
Uses dot 242e, dotinc 271b, dprint() 250e, and get2() 350c.

⟨exform() *switch modifier cases* 272b⟩+≡ (269b) ◁272a 272c▷
```
  case 'U':
  case 'D':
  case 'X':
  case 'O':
  case 'Q':
      if (literal)
          w = (long) dot;
      else if (get4(map, dot, &w) < 0)
          error("%r");
      dotinc = 4;
      if (c == 'U')
          dprint("%-16lud", w);
      else if (c == 'X')
          dprint("%-16#lux", w);
      else if (c == 'D')
          dprint("%-16ld", w);
      else if (c == 'O')
          dprint("%-#16luo", w);
      else if (c == 'Q')
          dprint("%-#16lo", w);
      break;
```
Uses dot 242e, dotinc 271b, dprint() 250e, and get4() 350b.

⟨exform() *switch modifier cases* 272c⟩+≡ (269b) ◁272b 273a▷
```
  case 'Z':
  case 'V':
  case 'Y':
      if (literal)
          v = dot;
      else if (get8(map, dot, &v) < 0)
          error("%r");
      dotinc = 8;
      if (c == 'Y')
          dprint("%-20#llux", v);
```

```
    else if (c == 'V')
        dprint("%-20lld", v);
    else if (c == 'Z')
        dprint("%-20llud", v);
    break;
```

Uses dot 242e, dotinc 271b, dprint() 250e, and get8() 350a.

⟨exform() *switch modifier cases* 273a⟩+≡ (269b) ◁272c 273c▷

```
case 'B':
case 'b':
case 'c':
case 'C':
    if (literal)
        ch = (uchar) dot;
    else if (get1(map, dot, &ch, 1)  < 0)
        error("%r");
    if (modifier == 'C')
        printesc(ch);
    else if (modifier == 'B' || modifier == 'b')
        dprint("%-8#lux", (long) ch);
    else
        printc(ch);
    dotinc = 1;
    break;
```

Uses dot 242e, dotinc 271b, dprint() 250e, get1() 351a, printc() 250f, and printesc() 273b.

⟨*function* printesc 273b⟩≡ (433a)

```
void
printesc(int c)
{
    static char hex[] = "0123456789abcdef";

    if (c < SPC || c >= 0177)
        dprint("\\x%c%c", hex[(c&0xF0)>>4], hex[c&0xF]);
    else
        printc(c);
}
```

Uses SPC 251h, dprint() 250e, and printc() 250f.

⟨exform() *switch modifier cases* 273c⟩+≡ (269b) ◁273a 273d▷

```
case 'r':
    if (literal)
        sh = (ushort) dot;
    else if (get2(map, dot, &sh) < 0)
        error("%r");
    dprint("%C", sh);
    dotinc = 2;
    break;
```

Uses dot 242e, dotinc 271b, dprint() 250e, and get2() 350c.

⟨exform() *switch modifier cases* 273d⟩+≡ (269b) ◁273c 274a▷

```
case 'R':
    if (literal) {
        sp = (ushort*) &dot;
        dprint("%C%C", sp[0], sp[1]);
        endline();
        dotinc = 4;
        break;
    }
```

```
        savdot=dot;
        while ((i = get2(map, dot, &sh) > 0) && sh) {
            dot=inkdot(2);
            dprint("%C", sh);
            endline();
        }
        if (i < 0)
            error("%r");
        dotinc = dot-savdot+2;
        dot=savdot;
        break;
```

Uses dot 242e, dotinc 271b, dprint() 250e, endline() 270a, get2() 350c, and inkdot() 254e.

⟨exform() *switch modifier cases* 274a⟩+≡ (269b) ◁273d 274b▷

```
  case 's':
        if (literal) {
            cp = (uchar*) &dot;
            for (i = 0; i < 4; i++)
                buf[i] = cp[i];
            buf[i] = 0;
            dprint("%s", buf);
            endline();
            dotinc = 4;
            break;
        }
        savdot = dot;
        for(;;){
            i = 0;
            do{
                if (get1(map, dot, (uchar*)&buf[i], 1) < 0)
                    error("%r");
                dot = inkdot(1);
                i++;
            }while(!fullrune(buf, i));
            if(buf[0] == 0)
                break;
            buf[i] = 0;
            dprint("%s", buf);
            endline();
        }
        dotinc = dot-savdot+1;
        dot = savdot;
        break;
```

Uses dot 242e, dotinc 271b, dprint() 250e, endline() 270a, get1() 351a, and inkdot() 254e.

⟨exform() *switch modifier cases* 274b⟩+≡ (269b) ◁274a 275a▷

```
  case 'S':
        if (literal) {
            cp = (uchar*) &dot;
            for (i = 0; i < 4; i++)
                printesc(cp[i]);
            endline();
            dotinc = 4;
            break;
        }
        savdot=dot;
        while ((i = get1(map, dot, &ch, 1) > 0) && ch) {
            dot=inkdot(1);
            printesc(ch);
            endline();
```

```
    }
    if (i < 0)
        error("%r");
    dotinc = dot-savdot+1;
    dot=savdot;
    break;
```

Uses dot 242e, dotinc 271b, endline() 270a, get1() 351a, inkdot() 254e, and printesc() 273b.

⟨exform() *switch modifier cases* 275a⟩+≡ (269b) ◁274b 275b▷

```
case 'M':
    i = machdata->hexinst(map, dot, buf, sizeof(buf));
    if (i < 0)
        error("%r");
    dotinc = i;
    dprint("%s", buf);
    if (*fp) {
        dotinc = 0;
        dprint("%48t");
    } else
        dprint("\n");
    break;
```

Uses dot 242e, dotinc 271b, dprint() 250e, and machdata 355e.

⟨exform() *switch modifier cases* 275b⟩+≡ (269b) ◁275a 275c▷

```
case 'f':
    /* BUG: 'f' and 'F' assume szdouble is sizeof(vlong) in the literal case */
    if (literal) {
        v = machdata->swav(dot);
        memmove(buf, &v, mach->szfloat);
    }else if (get1(map, dot, (uchar*)buf, mach->szfloat) < 0)
        error("%r");
    machdata->sftos(buf, sizeof(buf), (void*) buf);
    dprint("%s\n", buf);
    dotinc = mach->szfloat;
    break;
```

Uses dot 242e, dotinc 271b, dprint() 250e, get1() 351a, mach 51a, and machdata 355e.

⟨exform() *switch modifier cases* 275c⟩+≡ (269b) ◁275b 275d▷

```
case 'F':
    /* BUG: 'f' and 'F' assume szdouble is sizeof(vlong) in the literal case */
    if (literal) {
        v = machdata->swav(dot);
        memmove(buf, &v, mach->szdouble);
    }else if (get1(map, dot, (uchar*)buf, mach->szdouble) < 0)
        error("%r");
    machdata->dftos(buf, sizeof(buf), (void*) buf);
    dprint("%s\n", buf);
    dotinc = mach->szdouble;
    break;
```

Uses dot 242e, dotinc 271b, dprint() 250e, get1() 351a, mach 51a, and machdata 355e.

⟨exform() *switch modifier cases* 275d⟩+≡ (269b) ◁275c 276a▷

```
case 'n':
case 'N':
    printc('\n');
    dotinc=0;
    break;
```

Uses dotinc 271b and printc() 250f.

⟨exform() *switch modifier cases* 276a⟩+≡ (269b) ◁275d 276b▷

```
case '"':
    dotinc=0;
    while (*fp != '"' && *fp)
        printc(*fp++);
    if (*fp)
        fp++;
    break;
```

Uses dotinc 271b and printc() 250f.

⟨exform() *switch modifier cases* 276b⟩+≡ (269b) ◁276a 276c▷

```
case '^':
    dot=inkdot(-dotinc*fcount);
    return(fp);
```

Uses dot 242e, dotinc 271b, and inkdot() 254e.

⟨exform() *switch modifier cases* 276c⟩+≡ (269b) ◁276b 276d▷

```
case '+':
    dot=inkdot((WORD)fcount);
    return(fp);
```

Uses dot 242e and inkdot() 254e.

⟨exform() *switch modifier cases* 276d⟩+≡ (269b) ◁276c 276e▷

```
case '-':
    dot=inkdot(-(WORD)fcount);
    return(fp);
```

Uses dot 242e and inkdot() 254e.

⟨exform() *switch modifier cases* 276e⟩+≡ (269b) ◁276d

```
case 'z':
    if (findsym(dot, CTEXT, &s))
        dprint("%s() ", s.name);
    printsource(dot);
    printc(EOR);
    return fp;
```

Uses EOR 251g, dot 242e, dprint() 250e, findsym() 335b, printc() 250f, and printsource() 281b.

⟨acommand() *switch optional command suffix character* 276f⟩≡ (267c) 277a▷

```
case 'm':
    if (eqcom)
        error(BADEQ);
    cmdmap(map);
    break;
```

Uses BADEQ 442a and cmdmap() 276g.

⟨*function* cmdmap 276g⟩≡ (432)

```
/*
 * set up maps for a direct process image (/proc)
 */

void
cmdmap(Map *map)
{
    int i;
    char name[MAXSYM];

    extern char lastc;
```

```
        rdc();
        readsym(name);
        i = findseg(map, name);
        if (i < 0) /* not found */
            error("Invalid map name");

        if (expr(0)) {
            if (strcmp(name, "text") == 0)
                textseg(expv, &fhdr);
            map->seg[i].b = expv;
        } else
            error("Invalid base address");
        if (expr(0))
            map->seg[i].e = expv;
        else
            error("Invalid end address");
        if (expr(0))
            map->seg[i].f = expv;
        else
            error("Invalid file offset");
        if (rdc()=='?' && map == cormap) {
            if (fcor)
                close(fcor);
            fcor=fsym;
            corfil=symfil;
            cormap = symmap;
        } else if (lastc == '/' && map == symmap) {
            if (fsym)
                close(fsym);
            fsym=fcor;
            symfil=corfil;
            symmap=cormap;
        } else
            reread();
    }
```

Uses MAXSYM 437a, corfil 245b, expv 257a, fcor 249b, fhdr-68 247c, findseg() 98, fsym 247d, lastc 252b, rdc() 252d, readsym() 437b, reread() 252e, symfil 245a, and textseg() 328a.

⟨acommand() *switch optional command suffix character* 277a⟩+≡ (267c) ◁276f 278a▷

```
    case 'L':
    case 'l':
        if (eqcom)
            error(BADEQ);
        cmdsrc(lastc, map);
        break;
```

Uses BADEQ 442a, cmdsrc() 277b, and lastc 252b.

⟨*function* cmdsrc 277b⟩≡ (442c)

```
    void
    cmdsrc(int c, Map *map)
    {
        ulong w;
        long locval, locmsk;
        ADDR savdot;
        ushort sh;
        char buf[512];
        int ret;

        if (c == 'L')
```

```
        dotinc = 4;
    else
        dotinc = 2;
    savdot=dot;
    expr(1);
    locval=expv;
    if (expr(0))
        locmsk=expv;
    else
        locmsk = ~0;
    if (c == 'L')
        while ((ret = get4(map, dot, &w)) > 0 &&  (w&locmsk) != locval)
            dot = inkdot(dotinc);
    else
        while ((ret = get2(map, dot, &sh)) > 0 && (sh&locmsk) != locval)
            dot = inkdot(dotinc);
    if (ret < 0) {
        dot=savdot;
        error("%r");
    }
    symoff(buf, 512, dot, CANY);
    dprint(buf);
}
```

Uses dot 242e, dotinc 271b, dprint() 250e, expv 257a, get2() 350c, get4() 350b, inkdot() 254e, and symoff() 356a.

⟨acommand() *switch optional command suffix character* 278a⟩+≡ (267c) ◁277a

```
case 'W':
case 'w':
    if (eqcom)
        error(BADEQ);
    cmdwrite(lastc, map);
    break;
```

Uses BADEQ 442a, cmdwrite() 278c, and lastc 252b.

⟨*global* badwrite 278b⟩≡ (442c)

```
static char badwrite[] = "can't write process memory or text image";
```

Uses badwrite-67 278b.

⟨*function* cmdwrite 278c⟩≡ (442c)

```
void
cmdwrite(int wcom, Map *map)
{
    ADDR savdot;
    char *format;
    int pass;

    if (wcom == 'w')
        format = "x";
    else
        format = "X";
    expr(1);
    pass = 0;
    do {
        pass++;
        savdot=dot;
        exform(1, 1, format, map, 0, pass);
        dot=savdot;
        if (wcom == 'W') {
            if (put4(map, dot, expv) <= 0)
```

```
                error(badwrite);
            } else {
                if (put2(map, dot, expv) <= 0)
                    error(badwrite);
            }
            savdot=dot;
            dprint("=%8t");
            exform(1, 0, format, map, 0, pass);
            newline();
        } while (expr(0));
        dot=savdot;
    }
```

Uses badwrite-67 278b, dot 242e, dprint() 250e, exform() 269b, expv 257a, newline() 279a, put2() 352b, and put4() 352a.

⟨*function* newline 279a⟩≡ (430)

```
    void
    newline(void)
    {
        printc(EOR);
    }
```

Uses EOR 251g and printc() 250f.

E.8 Sub process control commands, :

The colon commands (:r, :c, :s, :b, :d, :k, :h, :x) map directly to acid library functions: :r is new(), :c is cont(), :s is the single-step loop in cont(), :b is bpset(), and so on. The underlying mechanism is identical: they write start, stop, startstop, waitstop, hang, or kill to /proc/<pid>/ctl through msgpcs()[242a] (the equivalent of acid's msg()[56d]). The main difference is that acid exposes these as programmable functions the user can compose (e.g., "set a breakpoint, continue, print a variable, delete the breakpoint, repeat 100 times"), while db offers them only as single-keystroke commands with a count parameter.

⟨command() *switch lastcom cases* 279b⟩+≡ (253a) ◁267b 292c▷

```
    case ':':
        if (!executing) {
            executing=TRUE;
            subpcs(nextchar());
            executing=FALSE;
            lastcom=savecom;
        }
        break;
```

Uses executing 279c, nextchar() 256c, and subpcs() 279e.

⟨*global* executing 279c⟩≡ (442c)

```
    bool executing;
```

⟨main() *just before repl* 279d⟩≡ (246b)

```
    if (executing)
        delbp();
    executing = FALSE;
```

⟨*function* subpcs 279e⟩≡ (443b)

```
    /* sub process control */

    void
    subpcs(int modif)
    {
        // enum<runmode>
```

```
    int runmode = SINGLE;
    bool keepnote = false;
    int check;
    int n;
    int r = 0;
    long line, curr;
    BKPT *bk;
    char *comptr;

    loopcnt=cntval;

    switch (modif) {
    ⟨subpcs() switch modif cases 283a⟩
    default:
        error("bad ':' command");
    }

    if (loopcnt>0) {
        dprint("%s: running\n", symfil);
        flush();
        r = runpcs(runmode, keepnote);
    }
    if (r)
        dprint("breakpoint%16t");
    else
        dprint("stopped at%16t");

Return:
    delbp();
    printpc();
    notes();
}
```

Uses SINGLE 281c, cntval 243a, delbp() 291c, dprint() 250e, flush() 251f, loopcnt 280a, printpc() 280b, runpcs() 281e, and symfil 245a.

⟨*global* loopcnt 280a⟩≡ (442c)

```
WORD loopcnt;
```

printpc()

⟨*function* printpc 280b⟩≡ (441b)

```
void
printpc(void)
{
    char buf[512];

    dot = rget(cormap, mach->pc);
    if(dot){
        printsource((long)dot);
        printc(' ');
        symoff(buf, sizeof(buf), (long)dot, CTEXT);
        dprint("%s/", buf);
        if (machdata->das(cormap, dot, 'i', buf, sizeof(buf)) < 0)
            error("%r");
        dprint("%16t%s\n", buf);
    }
}
```

Uses dot 242e, dprint() 250e, mach 51a, machdata 355e, printc() 250f, printsource() 281b, and symoff() 356a.

⟨*constant* STRINGSZ 281a⟩≡ (441b)
```
#define STRINGSZ 128
```

⟨*function* printsource 281b⟩≡ (441b)
```
/*
 * print the value of dot as file:line
 */
void
printsource(ADDR dot)
{
    char str[STRINGSZ];

    if (fileline(str, STRINGSZ, dot))
        dprint("%s", str);
}
```
Uses STRINGSZ-72 281a, dprint() 250e, and fileline() 340b.

Runmodes

⟨*constant* SINGLE 281c⟩≡ (426c)
```
#define SINGLE 1
```

⟨*constant* CONTIN 281d⟩≡ (426c)
```
#define CONTIN 2
```

runpcs()

runpcs()[281e] is the main execution loop, called by both :c (continue, CONTIN mode) and :s (single-step, SINGLE mode). In CONTIN mode it installs all breakpoints, resumes with startstop, then handles whatever stopped the target: a breakpoint hit, a note (signal), or an unexpected stop. The BKPTSKIP state handles the "step over the breakpoint we just hit" problem: the breakpoint is temporarily removed, one instruction is stepped, then the breakpoint is re-inserted. This is the same logic as acid's cont() function in /lib/acid/port.acid, just written in C instead of in the acid language.

⟨*function* runpcs 281e⟩≡ (443a)
```
/* service routines for sub process control */
int
runpcs(int runmode, bool keepnote)
{
    int rc = 0; // runcount
    BKPT *bkpt;

    if (adrflg)
        rput(cormap, mach->pc, dot);
    dot = rget(cormap, mach->pc);
    flush();

    while (loopcnt-- > 0) {
        if(loopcnt != 0)
            printpc();
        if (runmode == SINGLE) {
            ⟨runpcs() in SINGLE mode, clean breakpoint if at dot 282b⟩
            runstep(dot, keepnote);
        } else {
            if ((bkpt = scanbkpt(rget(cormap, mach->pc))) != nil) {
                execbkpt(bkpt, keepnote);
                keepnote = false;
```

```
            }
            setbp();
            runrun(keepnote);
        }
        keepnote = false;
        delbp();
        dot = rget(cormap, mach->pc);

        /* real note? */
        if (nnote > 0) {
            keepnote = true;
            rc = 0;
            continue;
        }
        bkpt = scanbkpt(dot);
        if(bkpt == nil){
            keepnote = false;
            rc = 0;
            continue;
        }
        /* breakpoint */
        if (bkpt->flag == BKPTTMP)
            bkpt->flag = BKPTCLR;
        else if (bkpt->flag == BKPTSKIP) {
            execbkpt(bkpt, keepnote);
            keepnote = false;
            loopcnt++; /* we didn't really stop */
            continue;
        }
        else {
            bkpt->flag = BKPTSKIP;
            --bkpt->count;
            if ((bkpt->comm[0] == EOR || command(bkpt->comm, ':') != 0)
                && bkpt->count != 0) {
                execbkpt(bkpt, keepnote);
                keepnote = false;
                loopcnt++;
                continue;
            }
            bkpt->count = bkpt->initcnt;
        }
        rc = 1;
    }
    return rc;
}
```

Uses BKPTCLR 243c, BKPTSKIP 243e, BKPTTMP 243f, EOR 251g, SINGLE 281c, adrflg 253d, command() 253a, delbp() 291c, dot 242e, execbkpt() 292a, flush() 251f, loopcnt 280a, mach 51a, nnote 244f, printpc() 280b, rput() 434b, runrun() 285b, runstep() 286c, scanbkpt() 289b, and setbp() 291a.

⟨command() *return (db)* 282a⟩≡ (253a)

```
if(adrflg)
    return dot;
return 1;
```

Uses adrflg 253d and dot 242e.

⟨runpcs() *in SINGLE mode, clean breakpoint if at dot* 282b⟩≡ (281e)

```
bkpt = scanbkpt(dot);
if (bkpt) {
    switch(bkpt->flag){
    case BKPTTMP:
```

```
            bkpt->flag = BKPTCLR;
            break;
        case BKPTSKIP:
            bkpt->flag = BKPTSET;
            break;
        }
    }
```

Uses BKPTCLR 243c, BKPTSET 243d, BKPTSKIP 243e, BKPTTMP 243f, dot 242e, and scanbkpt() 289b.

Running: :r

⟨subpcs() *switch modif cases* 283a⟩≡ (279e) 286a▷

```
    /* run program */
    case 'r':
    case 'R':
        endpcs();
        setup();
        runmode = CONTIN;
        break;
```

Uses CONTIN 281d, endpcs() 287d, and setup() 283b.

setup() and startpcs()

⟨*function* setup 283b⟩≡ (443a)

```
    /*
     * start up the program to be debugged in a child
     */
    void
    setup(void)
    {

        nnote = 0;
        startpcs();
        pcsactive = true;
        bpin = FALSE;
    }
```

Uses bpin 288b, nnote 244f, pcsactive 247a, and startpcs() 283c.

startpcs()[283c] is db's version of acid's newproc()[81c]/nproc()[82b]. It uses the same fork/hang/exec pattern: the child writes hang to its own ctl before calling exec, so the parent can intercept it at the first instruction. Compare with the sequence diagram in Chapter 3—the protocol is identical.

⟨*function* startpcs 283c⟩≡ (439b)

```
    void
    startpcs(void)
    {
        pid = fork();
        // child
        if (pid == 0) {
            pid = getpid();
            msgpcs("hang");
            doexec();
            exits(nil); // reachable?
        }
        // parent
        if (pid == -1)
            error("can't fork");
        child++;
```

```
    sprint(procname, "/proc/%d/mem", pid);
    corfil = procname;
    msgpcs("waitstop");

    // will call setcor()
    bpwait();

    if (adrflg)
        rput(cormap, mach->pc, adrval);

    while (rdc() != EOR)
        ;
    reread();
}
```

Uses EOR 251g, adrflg 253d, adrval 253e, bpwait() 284b, child 284c, corfil 245b, doexec() 284e, mach 51a, msgpcs() 242a, pid 246f, procname-66 284a, rdc() 252d, reread() 252e, and rput() 434b.

⟨*global* procname 284a⟩≡ (439b)

```
static char procname[100];
```

⟨*function* bpwait 284b⟩≡ (439b)

```
void
bpwait(void)
{
    setcor();
    unloadnote();
}
```

Uses setcor() 249c and unloadnote() 302a.

⟨*global* child 284c⟩≡ (439b)

```
int child;
```

doexec()

⟨*constant* MAXARG 284d⟩≡ (426c)

```
#define MAXARG 32
```

⟨*function* doexec 284e⟩≡ (439b)

```
void
doexec(void)
{
    char *argl[MAXARG];
    char args[LINSIZ];
    char *p;
    char **ap;
    char *thisarg;

    ap = argl;
    p = args;
    // argv[0] is the command itself
    *ap++ = symfil;

    ⟨doexec() adjust argl if extra arguments 285a⟩
    *ap = '\0';

    exec(symfil, argl);
    perror(symfil);
}
```

Uses LINSIZ 255e, MAXARG 284d, and symfil 245a.

⟨doexec() *adjust argl if extra arguments* 285a⟩≡ (284e)

```
for (rdc(); lastc != EOR;) {
    thisarg = p;
    if (lastc == '<' || lastc == '>') {
        *p++ = lastc;
        rdc();
    }
    while (lastc != EOR && lastc != SPC && lastc != TB) {
        *p++ = lastc;
        readchar();
    }
    if (lastc == SPC || lastc == TB)
        rdc();
    *p++ = 0;
    if (*thisarg == '<') {
        close(0);
        if (open(&thisarg[1], OREAD) < 0) {
            print("%s: cannot open\n", &thisarg[1]);
            _exits(0);
        }
    }
    else if (*thisarg == '>') {
        close(1);
        if (create(&thisarg[1], OWRITE, 0666) < 0) {
            print("%s: cannot create\n", &thisarg[1]);
            _exits(0);
        }
    }
    else
        *ap++ = thisarg;
}
```

Uses EOR 251g, SPC 251h, TB 251i, lastc 252b, rdc() 252d, and readchar() 255c.

runrun()

⟨*function* runrun 285b⟩≡ (439b)

```
void
runrun(bool keepnote)
{
    ⟨runrun() notes managment 285c⟩

    flush();
    msgpcs("startstop");
    bpwait();
}
```

Uses bpwait() 284b, flush() 251f, and msgpcs() 242a.

⟨runrun() *notes managment* 285c⟩≡ (285b)

```
int on = nnote;

unloadnote();
if(on != nnote){
    notes();
    error("not running: new notes pending");
}
if(keepnote)
    loadnote();
else
    nnote = 0;
```

Uses loadnote() 301d, nnote 244f, and unloadnote() 302a.

E.8.1 Stepping: :s

⟨subpcs() *switch modif cases* 286a⟩+≡ (279e) ◁283a 287b▷

```
/* single step */
case 's':
    if (pid == 0) {
        setup();
        loopcnt--;
    }
    runmode=SINGLE;
    keepnote=defval(true);
    break;
```

Uses SINGLE 281c, defval() 286b, loopcnt 280a, pid 246f, and setup() 283b.

⟨*function* defval 286b⟩≡ (439a)

```
WORD
defval(WORD w)
{
    if (expr(0))
        return (expv);
    else
        return (w);
}
```

Uses expv 257a.

runstep()[286c] implements single-stepping using the same follow-set mechanism as acid: call machdata->foll() (from libmach/5db.c) to compute the possible next PCs, set temporary breakpoints at each, resume with startstop, then remove the temporary breakpoints. This is identical to how acid's cont() library function handles single-stepping—the code path through libmach is shared. db's version is slightly more explicit because it manages the temporary BKPT structs inline rather than going through an acid list.

⟨*function* runstep 286c⟩≡ (439b)

```
void
runstep(uvlong loc, bool keepnote)
{
    int nfoll;
    uvlong foll[3];
    BKPT bkpt[3];
    int i;

    if(machdata->foll == 0){
        dprint("stepping unimplemented; assuming not a branch\n");
        nfoll = 1;
        foll[0] = loc+mach->pcquant;
    }else {
        nfoll = machdata->foll(cormap, loc, rget, foll);
        if (nfoll < 0)
            error("%r");
    }
    memset(bkpt, 0, sizeof bkpt);
    for(i=0; i<nfoll; i++){
        if(foll[i] == loc)
            error("can't single step: next instruction is dot");
        bkpt[i].loc = foll[i];
        bkput(&bkpt[i], true);
    }
```

```
    runrun(keepnote);
    for(i=0; i<nfoll; i++)
        bkput(&bkpt[i], false);
}
```

Uses bkput() 291b, dprint() 250e, mach 51a, machdata 355e, and runrun() 285b.

E.8.2 Killing: :k

⟨*global* NOPCS 287a⟩≡ (443b)

```
char NOPCS[] = "no process";
```

Uses NOPCS 287a.

⟨subpcs() *switch modif cases* 287b⟩+≡ (279e) ◁286a 288c▷

```
/* exit */
case 'k' :
case 'K':
    if (pid == 0)
        error(NOPCS);
    dprint("%d: killed", pid);
    pcsactive = true; /* force 'kill' ctl */
    endpcs();
    return;
```

Uses NOPCS 287a, dprint() 250e, endpcs() 287d, pcsactive 247a, and pid 246f.

⟨*global* ending 287c⟩≡ (429b)

```
// was in runpcs.c
bool ending;
```

⟨*function* endpcs 287d⟩≡ (443a)

```
/*
 * finish the process off;
 * kill if still running
 */
void
endpcs(void)
{
    BKPT *bk;

    if(ending)
        return;
    ending = true;
    if (pid) {
        if(pcsactive){
            killpcs();
            pcsactive = false;
        }
        pid=0;
        nnote=0;
        for (bk=bkpthead; bk; bk = bk->nxtbkpt)
            if (bk->flag == BKPTTMP)
                bk->flag = BKPTCLR;
            else if (bk->flag != BKPTCLR)
                bk->flag = BKPTSET;
    }
    bpin = FALSE;
    ending = false;
}
```

Uses BKPTCLR 243c, BKPTSET 243d, BKPTTMP 243f, bkpthead 243g, bpin 288b, ending 287c, killpcs() 288a, nnote 244f, pcsactive 247a, and pid 246f.

⟨*function* `killpcs` 288a⟩≡ (439b)
```
void
killpcs(void)
{
    msgpcs("kill");
}
```
Uses `msgpcs()` 242a.

⟨*global* `bpin` 288b⟩≡ (443a)
```
bool bpin;
```

E.8.3 Halting: `:h`

`:h` and the symmetric `:x` are the "attach to a running process" commands: after `db 1234` you can stop and resume pid 1234 without going through breakpoints. `:h` writes `stop` to `/proc/1234/ctl` via `grab()`[288d], which transitions the target into the `Stopped` state; `:x` writes `start` via `ungrab()`[289a] to resume it. The `acid` equivalents are `stop(pid)` and `start(pid)`, built on the same `ctl` messages. These two commands are what make `db 1234` useful for inspecting a process that is misbehaving but not yet crashed—handy when a program is stuck in a tight loop and you want to see where.

⟨`subpcs()` *switch modif cases* 288c⟩+≡ (279e) ◁287b 288f▷
```
/* halt the current process */
case 'h':
    ⟨subpcs() halting case, if addr 0 specified 288e⟩
    else {
        grab();
        dprint("stopped at%16t");
        goto Return;
    }
    return;
```
Uses `dprint()` 250e and `grab()` 288d.

⟨*function* `grab` 288d⟩≡ (439b)
```
void
grab(void)
{
    flush();
    msgpcs("stop");
    bpwait();
}
```
Uses `bpwait()` 284b, `flush()` 251f, and `msgpcs()` 242a.

⟨`subpcs()` *halting case, if addr 0 specified* 288e⟩≡ (288c)
```
if (adrflg && adrval == 0) {
    if (pid == 0)
        error(NOPCS);
    ungrab();
}
```
Uses `NOPCS` 287a, `adrflg` 253d, `adrval` 253e, `pid` 246f, and `ungrab()` 289a.

E.8.4 Unhalting: `:x`

⟨`subpcs()` *switch modif cases* 288f⟩+≡ (279e) ◁288c 289d▷
```
/* continue executing the current process */
case 'x':
    if (pid == 0)
```

```
        error(NOPCS);
    ungrab();
    return;
```

Uses NOPCS 287a, pid 246f, and ungrab() 289a.

⟨*function* ungrab 289a⟩≡ (439b)

```
void
ungrab(void)
{
    msgpcs("start");
}
```

Uses msgpcs() 242a.

E.9 Breakpoints

⟨*function* scanbkpt 289b⟩≡ (443a)

```
/*
 * find the breakpoint at adr, if any
 */
BKPT*
scanbkpt(ADDR adr)
{
    BKPT *bk;

    for (bk = bkpthead; bk; bk = bk->nxtbkpt)
        if (bk->flag != BKPTCLR && bk->loc == adr)
            break;
    return bk;
}
```

Uses BKPTCLR 243c and bkpthead 243g.

E.9.1 Inspecting breakpoints: $b

⟨printtrace() *switch modif cases* 289c⟩+≡ (261c) ◁270d 296b▷

```
/*print breakpoints*/
case 'b':
case 'B':
    for (bk=bkpthead; bk; bk=bk->nxtbkpt)
        if (bk->flag) {
            symoff(buf, 512, (WORD)bk->loc, CTEXT);
            dprint(buf);
            if (bk->count != 1)
                dprint(",%d", bk->count);
            dprint(":%c %s",
                    bk->flag == BKPTTMP ? 'B' : 'b',
                    bk->comm);
        }
    break;
```

Uses BKPTTMP 243f, bkpthead 243g, dprint() 250e, and symoff() 356a.

E.9.2 Setting breakpoints: :b

⟨subpcs() *switch modif cases* 289d⟩+≡ (279e) ◁288f 290b▷

```
/* set breakpoint */
case 'b':
```

```
case 'B':
    if (bk=scanbkpt(dot))
        bk->flag=BKPTCLR;

    ⟨subpcs() breakpoint case, find unused breakpoint bk or allocate one 290a⟩

    bk->loc = dot;
    bk->flag = modif == 'b' ? BKPTSET : BKPTTMP;
    bk->initcnt = bk->count = cntval;

    ⟨subpcs() breakpoint case, set optional breakpoint command 294e⟩
```

Uses BKPTCLR 243c, BKPTSET 243d, BKPTTMP 243f, cntval 243a, dot 242e, and scanbkpt() 289b.

⟨subpcs() *breakpoint case, find unused breakpoint bk or allocate one* 290a⟩≡ (289d)

```
for (bk=bkpthead; bk; bk=bk->nxtbkpt)
    if (bk->flag == BKPTCLR)
        break;
if (bk==nil) {
    bk = (BKPT *)malloc(sizeof(*bk));
    if (bk == nil)
        error("too many breakpoints");
    bk->nxtbkpt=bkpthead;
    bkpthead=bk;
}
```

Uses BKPTCLR 243c and bkpthead 243g.

E.9.3 Deleting breakpoints: :d

⟨subpcs() *switch modif cases* 290b⟩+≡ (279e) ◁289d 290c▷

```
/* delete breakpoint */
case 'd':
case 'D':
    if ((bk=scanbkpt(dot)) == 0)
        error("no breakpoint set");
    bk->flag=BKPTCLR;
    return;
```

Uses BKPTCLR 243c, dot 242e, and scanbkpt() 289b.

E.9.4 Continuing execution: :c

⟨subpcs() *switch modif cases* 290c⟩+≡ (279e) ◁290b 293a▷

```
/* continue with optional note */
case 'c':
case 'C':
    if (pid==0)
        error(NOPCS);
    runmode=CONTIN;
    keepnote=defval(1);
    break;
```

Uses CONTIN 281d, NOPCS 287a, defval() 286b, and pid 246f.

E.9.5 Installing breakpoints

Breakpoint installation in db uses the same get1/put1 mechanism as acid: save the original bytes at the breakpoint address, then overwrite with machdata->bpinst (the trap instruction). The bpin flag tracks whether

breakpoints are currently "live" in the target's memory, to avoid double-installing or double-removing. `acid` does not need this flag because its `bpset()` function installs each breakpoint individually. The lazy install/remove pattern matters for correctness: breakpoints are installed just before resuming (`setbp`) and removed as soon as the target stops (`delbp`). This ensures the debugger never sees breakpoint instructions when disassembling or inspecting memory.

⟨*function* `setbp` 291a⟩≡ (443a)

```
/*
 * install all the breakpoints
 */

void
setbp(void)
{
    BKPT *bk;

    if (bpin == TRUE || pid == 0)
        return;
    for (bk = bkpthead; bk; bk = bk->nxtbkpt)
        if (bk->flag != BKPTCLR)
            bkput(bk, true);
    bpin = TRUE;
}
```

Uses `BKPTCLR` 243c, `bkpthead` 243g, `bkput()` 291b, `bpin` 288b, and `pid` 246f.

⟨*function* `bkput` 291b⟩≡ (439b)

```
void
bkput(BKPT *bp, bool install)
{
    char buf[256];
    ADDR loc;
    int ret;

    errstr(buf, sizeof buf);
    if(machdata->bpfix)
        loc = (*machdata->bpfix)(bp->loc);
    else
        loc = bp->loc;

    if(install){
        ret = get1(cormap, loc, bp->save, machdata->bpsize);
        if (ret > 0)
            ret = put1(cormap, loc, machdata->bpinst, machdata->bpsize);
    }else
        ret = put1(cormap, loc, bp->save, machdata->bpsize);

    if(ret < 0){
        sprint(buf, "can't set breakpoint at %#llux: %r", bp->loc);
        print(buf);
        read(0, buf, 100);
    }
}
```

Uses `get1()` 351a, `machdata` 355e, and `put1()` 352c.

E.9.6 Uninstalling breakpoints

⟨*function* `delbp` 291c⟩≡ (443a)

```
/*
```

```
 * remove all breakpoints from the process' address space
 */

void
delbp(void)
{
    BKPT *bk;

    if (bpin == FALSE || pid == 0)
        return;
    for (bk = bkpthead; bk; bk = bk->nxtbkpt)
        if (bk->flag != BKPTCLR)
            bkput(bk, false);
    bpin = FALSE;
}
```

Uses BKPTCLR 243c, bkpthead 243g, bkput() 291b, bpin 288b, and pid 246f.

E.9.7 Executing breakpoints

⟨*function* execbkpt 292a⟩≡ (443a)

```
/*
 * skip over a breakpoint:
 * remove breakpoints, then single step
 * so we can put it back
 */
void
execbkpt(BKPT *bk, int keepnote)
{
    runstep(bk->loc, keepnote);
    bk->flag = BKPTSET;
}
```

Uses BKPTSET 243d and runstep() 286c.

E.9.8 Breakpoints and notes

E.10 Other commands

⟨command() *locals (db)* 292b⟩+≡ (253a) ◁253c 295b▷

```
char *reg;
char savc;
```

⟨command() *switch lastcom cases* 292c⟩+≡ (253a) ◁279b 297e▷

```
case '>':
    lastcom = savecom;
    savc=rdc();
    if (reg=regname(savc))
        rput(cormap, reg, dot);
    else
        error("bad variable");
    break;
```

Uses dot 242e, rdc() 252d, regname() 442b, and rput() 434b.

E.11 Source level C Debugging

`db` provides two source-level features: `:S` (step one C line) and `$C` (stack trace with local variable values). Both rely on the line number and symbol table metadata from the compiler/linker (Chapter 5). `acid` offers the same capabilities through its `step()` and `stk()` library functions, but `acid`'s programmability lets users build higher-level commands on top (e.g., "step until variable `x` changes"), while `db` can only repeat `:S` with a count.

E.11.1 Stepping: `:S`

⟨`subpcs()` *switch modif cases* 293a⟩+≡ (279e) ◁290c 302b▷

```
case 'S':
    if (pid == 0) {
        setup();
        loopcnt--;
    }
    keepnote=defval(true);

    line = pc2line(rget(cormap, mach->pc));
    n = loopcnt;
    dprint("%s: running\n", symfil);
    flush();
    for (loopcnt = 1; n > 0; loopcnt = 1) {
        r = runpcs(SINGLE, keepnote);
        curr = pc2line(dot);
        if (line != curr) { /* on a new line of c */
            line = curr;
            n--;
        }
    }
    loopcnt = 0;
    break;
```

Uses `SINGLE` 281c, `defval()` 286b, `dot` 242e, `dprint()` 250e, `flush()` 251f, `loopcnt` 280a, `mach` 51a, `pc2line()` 344b, `pid` 246f, `runpcs()` 281e, `setup()` 283b, and `symfil` 245a.

E.11.2 Stack traces: `$S`

⟨*function* `printparams` 293b⟩≡ (441b)

```
void
printparams(Symbol *fn, ADDR fp)
{
    int i;
    Symbol s;
    ulong w;
    int first = 0;

    fp += mach->szaddr;   /* skip saved pc */
    s = *fn;
    for (i = 0; localsym(&s, i); i++) {
        if (s.class != CPARAM)
            continue;
        if (first++)
            dprint(", ");
        if (get4(cormap, fp+s.value, &w) > 0)
            dprint("%s=%#lux", s.name, w);
    }
}
```

Uses `dprint()` 250e, `get4()` 350b, `localsym()` 336b, and `mach` 51a.

⟨*function* `printlocals` 294a⟩≡ (441b)

```
void
printlocals(Symbol *fn, ADDR fp)
{
    int i;
    ulong w;
    Symbol s;

    s = *fn;
    for (i = 0; localsym(&s, i); i++) {
        if (s.class != CAUTO)
            continue;
        if (get4(cormap, fp-s.value, &w) > 0)
            dprint("%8t%s.%s/%10t%#lux\n", fn->name, s.name, w);
        else
            dprint("%8t%s.%s/%10t?\n", fn->name, s.name);
    }
}
```

Uses `dprint()` 250e, `get4()` 350b, and `localsym()` 336b.

E.12 Advanced features

E.12.1 Conditional breakpoints

`db`'s conditional breakpoints are a limited form of programmability: each breakpoint can have an attached command string (stored in `bk->comm`) that is executed when the breakpoint fires. If the command returns non-zero (i.e., it sets `dot` to a non-zero address), the breakpoint "counts" and `db` continues. This lets you write things like `main:b *x/X` ("at `main`, print `*x` as hex and continue"). But the command language is still the fixed `adb` grammar—you cannot define new functions or use control flow. This is exactly the limitation that motivated `acid`'s design: Phil Winterbottom wanted breakpoint actions to be written in a real programming language.

⟨*constant* `MAXCOM` 294b⟩≡ (426c)

```
#define MAXCOM 64
```

⟨`Bkpt` *other fields* 294c⟩+≡ (243b) ◁243i

```
char comm[MAXCOM];
```

Uses `MAXCOM` 294b.

⟨*constant HUGEINT (db)* 294d⟩≡ (426c)

```
#define HUGEINT 0x7fffffff /* enormous WORD */
```

⟨`subpcs()` *breakpoint case, set optional breakpoint command* 294e⟩≡ (289d)

```
check=MAXCOM-1;
comptr=bk->comm;

rdc();
reread();

do {
    *comptr++ = readchar();
} while (check-- && lastc!=EOR);
*comptr='\0';
if(bk->comm[0] != EOR && cntflg == FALSE)
    bk->initcnt = bk->count = HUGEINT;
reread();
if (check)
```

```
    return;
  error("bkpt command too long");
```

Uses EOR 251g, HUGEINT 294d, MAXCOM 294b, cntflg 253g, lastc 252b, rdc() 252d, readchar() 255c, and reread() 252e.

⟨command() *initializations (db)* 295a⟩+≡ (253a) ◁254c

```
  if (buf) {
      if (*buf==EOR)
          return FALSE;
      clrinp();
      lp=(Rune*)buf;
  }
```

Uses EOR 251g, clrinp() 252c, and lp 251j.

⟨command() *locals (db)* 295b⟩+≡ (253a) ◁292b

```
  Rune*   savlp = lp;
  char savlc = lastc;
  char savpc = peekc;
```

Uses lastc 252b, lp 251j, and peekc 252a.

⟨command() *finalizations (db)* 295c⟩≡ (253a)

```
  if (buf == nil)
      reread();
  else {
      clrinp();
      lp=savlp;
      lastc = savlc;
      peekc = savpc;
  }
```

Uses clrinp() 252c, lastc 252b, lp 251j, peekc 252a, and reread() 252e.

E.12.2 Input file debugging commands

⟨*constant* MAXIFD 295d⟩≡ (430)

```
  #define MAXIFD 5
```

⟨*global* istack 295e⟩≡ (430)

```
  struct {
      int fd;
      int r9;
  } istack[MAXIFD];
```

Uses MAXIFD-69 295d and __anon_struct_6 295e.

⟨*global* ifiledepth 295f⟩≡ (430)

```
  int ifiledepth;
```

⟨*function* iclose 295g⟩≡ (430)

```
  void
  iclose(int stack, int err)
  {
      if (err) {
          if (infile) {
              close(infile);
              infile=STDIN;
          }
          while (--ifiledepth >= 0)
              if (istack[ifiledepth].fd)
                  close(istack[ifiledepth].fd);
          ifiledepth = 0;
```

```
    } else if (stack == 0) {
        if (infile) {
            close(infile);
            infile=STDIN;
        }
    } else if (stack > 0) {
        if (ifiledepth >= MAXIFD)
            error("$<< nested too deeply");
        istack[ifiledepth].fd = infile;
        ifiledepth++;
        infile = STDIN;
    } else {
        if (infile) {
            close(infile);
            infile=STDIN;
        }
        if (ifiledepth > 0) {
            infile = istack[--ifiledepth].fd;
        }
    }
}
```

Uses MAXIFD-69 295d, ifiledepth 295f, infile 245c, and istack 295e.

⟨*function* redirout 296a⟩≡ (430)

```
void
redirout(char *file)
{
    int fd;

    if (file == 0){
        oclose();
        return;
    }
    flushbuf();
    if ((fd = open(file, 1)) >= 0)
        seek(fd, 0L, 2);
    else if ((fd = create(file, 1, 0666)) < 0)
        error("cannot create");
    Bterm(&stdout);
    Binit(&stdout, fd, OWRITE);
}
```

Uses flushbuf() 251e, oclose() 303d, and stdout 250b.

⟨printtrace() *switch modif cases* 296b⟩+≡ (261c) ◁289c 297a▷

```
case '<':
    if (cntval == 0) {
        while (readchar() != EOR)
            ;
        reread();
        break;
    }
    if (rdc() == '<')
        stack = 1;
    else {
        stack = 0;
        reread();
    }
    fname = getfname();
    redirin(stack, fname);
    break;
```

Uses EOR 251g, cntval 243a, getfname() 440c, rdc() 252d, readchar() 255c, redirin() 441a, and reread() 252e.

⟨printtrace() *switch modif cases* 297a⟩+≡ (261c) ◁296b 299e▷
```
case '>':
    fname = getfname();
    redirout(fname);
    break;
```
Uses getfname() 440c and redirout() 296a.

⟨main() *locals (db)* 297b⟩+≡ (245d) ◁246g 298e▷
```
char *s;
```

⟨main() *command line processing (db)* 297c⟩≡ (245d) 298a▷
```
case 'I':
    s = ARGF();
    if(s == 0)
        dprint("missing -I argument\n");
    else
        Ipath = s;
    break;
```

⟨main() *in loop, if eof, and if infile was not STDIN* 297d⟩≡ (255a)
```
iclose(-1, 0);
eof = false;
longjmp(env, 1);
```

E.12.3 Formatted output

⟨command() *switch lastcom cases* 297e⟩+≡ (253a) ◁292c 297g▷
```
case '\0':
    prints(DBNAME);
    break;
```
Uses DBNAME 297f and prints() 251a.

⟨*constant* DBNAME 297f⟩≡ (426c)
```
#define DBNAME "db\n"
```

E.12.4 Shell output

⟨command() *switch lastcom cases* 297g⟩+≡ (253a) ◁297e
```
case '!':
    lastcom=savecom;
    shell();
    break;
```
Uses shell() 297h.

⟨*function* shell 297h⟩≡ (442c)
```
/*
 * shell escape
 */

void
shell(void)
{
    int rc, unixpid;
    char *argp = (char*)lp;
```

```
        while (lastc!=EOR)
            rdc();
        if ((unixpid=fork())==0) {
            *lp=0;
            execl("/bin/rc", "rc", "-c", argp, nil);
            exits("execl");    /* botch */
        } else if (unixpid == -1) {
            error("cannot fork");
        } else {
            mkfault = 0;
            while ((rc = waitpid()) != unixpid){
                if(rc == -1 && mkfault){
                    mkfault = 0;
                    continue;
                }
                break;
            }
            prints("!");
            reread();
        }
    }
```

Uses EOR 251g, lastc 252b, lp 251j, mkfault 300c, prints() 251a, rdc() 252d, and reread() 252e.

E.12.5 Modifying code: db -w

⟨main() *command line processing (db)* 298a⟩+≡ (245d) ◁297c 298f▷

```
  case 'w':
      wtflag = ORDWR;  /* suitable for open() */
      break;
```

⟨getfile() *if wtflag* 298b⟩≡ (248g)

```
  if (f < 0 && xargc > cnt && wtflag)
       f = create(filnam, 1, 0666);
```

Uses xargc 298c.

⟨*global* xargc 298c⟩≡ (429b)

```
  int xargc;  /* bullshit */
```

⟨main() *initialization before repl (db)* 298d⟩+≡ (245d) ◁246b

```
  xargc = argc;
```

E.12.6 Cross architecture debugging: db -m

⟨main() *locals (db)* 298e⟩+≡ (245d) ◁297b 299c▷

```
  char *name = nil;
```

⟨main() *command line processing (db)* 298f⟩+≡ (245d) ◁298a 299b▷

```
  case 'm':
      name = ARGF();
      if(name == nil)
          dprint("missing -m argument\n");
      break;
```

⟨main() *if db -m and unknown machine* 298g⟩≡ (246c)

```
  if (name && machbyname(name) == 0)
          dprint ("unknown machine %s", name);
```

E.12.7 Kernel debugging

`db -k` is the kernel-debugging mode: it tells `db` the symbol file is a kernel image rather than a user-space binary, and adjusts `symmap` so that its segment addresses match the kernel's virtual address space at runtime. `kmsys()`[299f] does this by masking off the top bits of the text segment (because the kernel is mapped at a fixed high virtual address, so the on-disk addresses need the high-bit mask stripped) and OR-ing the data segment address with `mach->kbase` (the kernel data base). Without this translation, every `?i` on a kernel symbol would miss because the executable's addresses would not line up with what you see through `/proc/1/mem`. The analogous feature on Linux is `gdb` with a `vmlinux` image and a `/proc/kcore` target, or `kgdb` for live kernel debugging.

`db -k`

⟨*global* `kflag` 299a⟩≡ (429b)
```
bool kflag;
```

⟨`main()` *command line processing (db)* 299b⟩+≡ (245d) ◁298f
```
case 'k':
    kflag = true;
    break;
```

⟨`main()` *locals (db)* 299c⟩+≡ (245d) ◁298e
```
char *cpu, *p, *q;
```

⟨`main()` *when pid argument, if kflag* 299d⟩≡ (246h)
```
if(kflag){
    cpu = getenv("cputype");
    if(cpu == nil){
        cpu = "386";
        dprint("$cputype not set; assuming %s\n", cpu);
    }
    p = getenv("terminal");
    if(p==nil || (p=strchr(p, ' '))==0 || p[1]==' ' || p[1]==0){
        strcpy(b1, "/386/9pc");
        dprint("missing or bad $terminal; assuming %s\n", b1);
    }else{
        p++;
        q = strchr(p, ' ');
        if(q)
            *q = '\0';
        sprint(b1, "/%s/9%s", cpu, p);

    }
}
```

⟨`printtrace()` *switch modif cases* 299e⟩+≡ (261c) ◁297a
```
case 'k':
    kmsys();
    break;
```
Uses `kmsys()` 299f.

⟨*function* `kmsys` 299f⟩≡ (432)
```
void
kmsys(void)
{
    int i;

    i = findseg(symmap, "text");
    if (i >= 0) {
        symmap->seg[i].b = symmap->seg[i].b & ~mach->ktmask;
```

```
            symmap->seg[i].e = ~0;
        }

        i = findseg(symmap, "data");
        if (i >= 0) {
            symmap->seg[i].b |= mach->kbase;
            symmap->seg[i].e |= mach->kbase;
        }
    }
```

Uses `findseg()` 98 and `mach` 51a.

E.12.8 Signals and notes

Like any interactive REPL that drives a child process, `db` has to juggle two separate note streams: notes delivered to the debugger itself (the common case is Ctrl-C from the user) and notes the kernel generates for the debuggee (a segfault, a user-sent note, or a `sys: breakpoint` fault when one of `db`'s own trap instructions fires). The two are handled very differently. The debugger's own notes are caught by the `notify(fault)` registration in `main()`[69e]; the `fault()`[300b] handler sets the `mkfault` flag and returns, which causes `readchar()`[255c] and `dprint()`[250e] to abandon what they are doing and `longjmp` back to the REPL via `error()`[206b]. This is the same setjmp/longjmp REPL-abort pattern used by `ed`, `acid`, and the Plan 9 shell: a Ctrl-C anywhere in the program forces an immediate return to the command prompt without unwinding the C call stack by hand. The debuggee's notes are pulled from `/proc/<pid>/note` by `unloadnote()`[302a] whenever the target stops, and parked in the `note[]` array for the user to inspect with `:n` or re-deliver to the process with `loadnote()`[301d]. There is one subtlety worth calling out: `unloadnote()` silently *discards* any note matching `sys: breakpoint`, because those are not real signals but the trap instructions `db` itself planted firing, and the user does not care about them. Every other note is kept, which is why a real segfault in the target shows up in `$?` output while a breakpoint hit does not.

Debugger notes

⟨`main()` *call notify* 300a⟩≡ (246b)

```
notify(fault);
```

⟨*function* `fault` 300b⟩≡ (444)

```
/*
 * An interrupt occurred;
 * seek to the end of the current file
 * and remember that there was a fault.
 */
void
fault(void *a, char *s)
{
    USED(a);
    if(strncmp(s, "interrupt", 9) == 0){
        seek(infile, 0L, 2);
        mkfault++;
        noted(NCONT);
    }
    noted(NDFLT);
}
```

Uses `infile` 245c and `mkfault` 300c.

⟨*global* `mkfault` 300c⟩≡ (429b)

```
bool mkfault;
```

⟨main() *in loop, handle mkfault (db)* 301a⟩≡ (245d)
```
if (mkfault) {
    mkfault=0;
    printc('\n');
    prints(DBNAME);
}
```

⟨dprint() *return if mkfault* 301b⟩≡ (250e)
```
if(mkfault)
    return -1;
```
Uses mkfault 300c.

⟨readchar() *if mkfault* 301c⟩≡ (256a)
```
if (mkfault) {
    eof = 0;
    error(nil);
}
```
Uses eof 254g and mkfault 300c.

Debugged notes

⟨*function* loadnote 301d⟩≡ (439b)
```
/*
 * reload the note buffer
 */
void
loadnote(void)
{
    int i;
    char err[ERRMAX];

    setpcs();
    for(i=0; i<nnote; i++){
        if(write(notefd, note[i], strlen(note[i])) < 0){
            errstr(err, sizeof err);
            if(strcmp(err, "interrupted") != 0)
                endpcs();
            errors("can't write note file", err);
        }
    }
    nnote = 0;
}
```
Uses endpcs() 287d, errors() 303a, nnote 244f, note 244d, notefd 244a, and setpcs() 241e.

⟨*function* notes 301e⟩≡ (439b)
```
void
notes(void)
{
    int n;

    if(nnote == 0)
        return;
    dprint("notes:\n");
    for(n=0; n<nnote; n++)
        dprint("%d:\t%s\n", n, note[n]);
}
```
Uses dprint() 250e, nnote 244f, and note 244d.

⟨*function* `unloadnote` 302a⟩≡ (439b)

```
/*
 * empty the note buffer and toss pending breakpoint notes
 */
void
unloadnote(void)
{
    char err[ERRMAX];

    setpcs();
    for(; nnote<NNOTE; nnote++){
        switch(read(notefd, note[nnote], sizeof note[nnote])){
        case -1:
            errstr(err, sizeof err);
            if(strcmp(err, "interrupted") != 0)
                endpcs();
            errors("can't read note file", err);
        case 0:
            return;
        }

        note[nnote][ERRMAX-1] = '\0';
        if(strncmp(note[nnote], "sys: breakpoint", 15) == 0)
            --nnote;
    }
}
```

Uses `NNOTE` 244e, `endpcs()` 287d, `errors()` 303a, `nnote` 244f, `note` 244d, `notefd` 244a, and `setpcs()` 241e.

`:n`

⟨`subpcs()` *switch modif cases* 302b⟩+≡ (279e) ◁293a

```
/* deal with notes */
case 'n':
    if (pid==0)
        error(NOPCS);
    n=defval(-1);
    if(n>=0 && n<nnote){
        nnote--;
        memmove(note[n], note[n+1], (nnote-n)*sizeof(note[0]));
    }
    notes();
    return;
```

Uses `NOPCS` 287a, `defval()` 286b, `nnote` 244f, `note` 244d, and `pid` 246f.

⟨*function* `error` 302c⟩≡ (429a)

```
/*
 * An error occurred; save the message for later printing,
 * close open files, and reset to main command loop.
 */
void
error(char *n)
{
    errmsg = n;
    iclose(0, 1);
    oclose();
    flush();
    delbp();
    ending = 0;
```

```
    longjmp(env, 1);
}
```

Uses delbp() 291c, ending 287c, env 246d, errmsg 303b, flush() 251f, iclose() 295g, and oclose() 303d.

⟨*function* errors 303a⟩≡ (429a)

```
void
errors(char *m, char *n)
{
    static char buf[128];

    sprint(buf, "%s: %s", m, n);
    error(buf);
}
```

⟨*global* errmsg 303b⟩≡ (429b)

```
// was static in main.c
char *errmsg;
```

⟨main() *in loop, handle errmsg (db)* 303c⟩≡ (245d)

```
if (errmsg) {
    dprint(errmsg);
    printc('\n');
    errmsg = nil;
}
```

⟨*function* oclose 303d⟩≡ (430)

```
void
oclose(void)
{
    flushbuf();
    Bterm(&stdout);
    Binit(&stdout, 1, OWRITE);
}
```

Uses flushbuf() 251e and stdout 250b.

Appendix F

Extra Code

F.1 include/

F.1.1 include/debug/mach.h

⟨*enum* dissembler_type 304a⟩≡ (304c)
```
/* dissembler types */
enum dissembler_type
{
    ANONE = 0,

    AI386,
    AI8086,                     /* oh god */
    AARM,
    AMIPS,
    AMIPSCO,            /* native mips */
};
```

⟨*enum* object_file_type 304b⟩≡ (304c)
```
/* object file types */
enum object_file_type
{
    Obj386 = 0,                 /* .8 */
    ObjArm,                     /* .5 */
    ObjMips,            /* .v */

    Maxobjtype,
};
```

⟨include/debug/mach.h 304c⟩≡
```
/*
 *      Architecture-dependent application data
 */

// This define the Exec and Sym structures.
#include "a.out.h"
//TODO: include "elf.h" too? and macho.h?

#pragma src     "/sys/src/libmach"
#pragma lib     "libmach.a"

/*
 *      Supported architectures:
 *              i386,
 *              arm,
```

```
 *              mips
 */
```

⟨*enum* executable_type 50a⟩
⟨*enum* machine_type 51b⟩
⟨*enum* dissembler_type 304a⟩
⟨*enum* object_file_type 304b⟩
⟨*enum* symbol_type 54b⟩

```
typedef struct  Map     Map;
typedef struct  Symbol  Symbol;
typedef struct  Reglist Reglist;
typedef struct  Mach    Mach;
typedef struct  Machdata Machdata;
typedef struct  Fhdr Fhdr;
```

⟨*struct* Map 53⟩

⟨*struct* Symbol 54a⟩

⟨*struct* Reglist 51c⟩

⟨*enum* register_flag 51d⟩

```
/*
 *      Machine-dependent data is stored in two structures:
 *              Mach  - miscellaneous general parameters
 *              Machdata - jump vector of service functions used by debuggers
 *
 *      Mach is defined in ?.c and set in executable.c
 *
 *      Machdata is defined in ?db.c
 *              and set in the debugger startup.
 */
```

⟨*struct* Mach 50b⟩

```
extern  Mach    *mach;                  /* Current machine */

typedef uvlong  (*Rgetter)(Map*, char*);
typedef void    (*Tracer)(Map*, uvlong, uvlong, Symbol*);
```

⟨*struct* Machdata 51e⟩

⟨*struct* Fhdr 49⟩

```
extern  int     asstype;        /* dissembler type - machdata.c */
extern  Machdata *machdata;     /* jump vector - machdata.c */

Map*            attachproc(int /*pidt*/, int, int, Fhdr*);

int             beieee80ftos(char*, int, void*);
int             beieeesftos(char*, int, void*);
int             beieeedftos(char*, int, void*);
ushort          beswab(ushort);
ulong           beswal(ulong);
uvlong          beswav(uvlong);

uvlong          ciscframe(Map*, uvlong, uvlong, uvlong, uvlong);
int             cisctrace(Map*, uvlong, uvlong, uvlong, Tracer);
```

```
int             crackhdr(int fd, Fhdr*);

uvlong          file2pc(char*, long);
int             fileelem(Sym**, uchar *, char*, int);
long            fileline(char*, int, uvlong);
int             filesym(int, char*, int);
int             findlocal(Symbol*, char*, Symbol*);

int             findseg(Map*, char*);

int             findsym(uvlong, int, Symbol *);
int             fnbound(uvlong, uvlong*);
int             fpformat(Map*, Reglist*, char*, int, int);

int             get1(Map*, uvlong, uchar*, int);
int             get2(Map*, uvlong, ushort*);
int             get4(Map*, uvlong, ulong*);
int             get8(Map*, uvlong, uvlong*);
int             geta(Map*, uvlong, uvlong*);
int             getauto(Symbol*, int, int, Symbol*);
Sym*            getsym(int);
int             globalsym(Symbol *, int);
char*           _hexify(char*, ulong, int);

int             ieeesftos(char*, int, ulong);
int             ieeedftos(char*, int, ulong, ulong);

int             isar(Biobuf*);

int             leieee80ftos(char*, int, void*);
int             leieeesftos(char*, int, void*);
int             leieeedftos(char*, int, void*);

ushort          leswab(ushort);
ulong           leswal(ulong);
uvlong          leswav(uvlong);
uvlong          line2addr(long, uvlong, uvlong);

Map*            loadmap(Map*, int, Fhdr*);

int             localaddr(Map*, char*, char*, uvlong*, Rgetter);
int             localsym(Symbol*, int);
int             lookup(char*, char*, Symbol*);

void            machbytype(int);
int             machbyname(char*);

int             nextar(Biobuf*, int, char*);
Map*            newmap(Map*, int);
void            objtraverse(void(*)(Sym*, void*), void*);
int             objtype(Biobuf*, char**);
uvlong          pc2sp(uvlong);
long            pc2line(uvlong);

int             put1(Map*, uvlong, uchar*, int);
int             put2(Map*, uvlong, ushort);
int             put4(Map*, uvlong, ulong);
int             put8(Map*, uvlong, uvlong);
int             puta(Map*, uvlong, uvlong);
```

```
int             readar(Biobuf*, int, vlong, int);
int             readobj(Biobuf*, int);

uvlong          riscframe(Map*, uvlong, uvlong, uvlong, uvlong);
int             risctrace(Map*, uvlong, uvlong, uvlong, Tracer);

int             setmap(Map*, int, uvlong, uvlong, vlong, char*);

Sym*            symbase(long*);
int             syminit(int, Fhdr*);

int             symoff(char*, int, uvlong, int);
void            textseg(uvlong, Fhdr*);
int             textsym(Symbol*, int);
void            unusemap(Map*, int);
```

F.1.2 include/exec/bootexec.h

⟨*struct* coffsect 307a⟩≡ (308a)

```
struct coffsect
{
    char        name[8];
    ulong       phys;
    ulong       virt;
    ulong       size;
    ulong       fptr;
    ulong       fptrreloc;
    ulong       fptrlineno;
    ulong       nrelocnlineno;
    ulong       flags;
};
```

⟨*struct* i386exec 307b⟩≡ (308a)

```
struct i386exec
{
    struct      i386coff{
        ulong   isectmagic;
        ulong   itime;
        ulong   isyms;
        ulong   insyms;
        ulong   iflags;
    };
    struct      i386hdr{
        ulong   imagic;
        ulong   itextsize;
        ulong   idatasize;
        ulong   ibsssize;
        ulong   ientry;
        ulong   itextstart;
        ulong   idatastart;
    };
    struct coffsect itexts;
    struct coffsect idatas;
    struct coffsect ibsss;
    struct coffsect icomments;
};
```

⟨include/bootexec.h 308a⟩≡
  ⟨*struct* coffsect 307a⟩

  ⟨*struct* i386exec 307b⟩

F.2 libmach/

F.2.1 libmach/5.c

⟨*function* REGOFF*(arm)* 308b⟩≡ (309a)
```
#define REGOFF(x)       (ulong) (&((struct Ureg *) 0)->x)
```

⟨*constant* SP*(arm)* 308c⟩≡ (309a)
```
#define SP              REGOFF(r13)
```

⟨*constant* PC*(arm)* 308d⟩≡ (309a)
```
#define PC              REGOFF(pc)
```

⟨*constant* REGSIZE*(arm)* 308e⟩≡ (309a)
```
#define REGSIZE         sizeof(struct Ureg)
```

⟨*global* armreglist*(arm)* 308f⟩≡ (309a)
```
Reglist armreglist[] =
{
    {"TYPE",    REGOFF(type),           RINT|RRDONLY, 'X'},
    {"PSR",             REGOFF(psr),            RINT|RRDONLY, 'X'},
    {"PC",              PC,                     RINT, 'X'},
    {"SP",              SP,                     RINT, 'X'},
    {"R15",             PC,                     RINT, 'X'},
    {"R14",             REGOFF(r14),            RINT, 'X'},
    {"R13",             REGOFF(r13),            RINT, 'X'},
    {"R12",             REGOFF(r12),            RINT, 'X'},
    {"R11",             REGOFF(r11),            RINT, 'X'},
    {"R10",             REGOFF(r10),            RINT, 'X'},
    {"R9",              REGOFF(r9),             RINT, 'X'},
    {"R8",              REGOFF(r8),             RINT, 'X'},
    {"R7",              REGOFF(r7),             RINT, 'X'},
    {"R6",              REGOFF(r6),             RINT, 'X'},
    {"R5",              REGOFF(r5),             RINT, 'X'},
    {"R4",              REGOFF(r4),             RINT, 'X'},
    {"R3",              REGOFF(r3),             RINT, 'X'},
    {"R2",              REGOFF(r2),             RINT, 'X'},
    {"R1",              REGOFF(r1),             RINT, 'X'},
    {"R0",              REGOFF(r0),             RINT, 'X'},
    {  0 }
};
```
Uses PC-174 308d, REGOFF-172 308b, and SP-173 308c.

⟨*global* marm*(arm)* 308g⟩≡ (309a)
```
Mach marm =
{
    "arm",
    MARM,               /* machine type */
    armreglist, /* register set */
    REGSIZE,    /* register set size */
    0,          /* fp register set size */
    "PC",               /* name of PC */
    "SP",               /* name of SP */
```

```
    "R14",              /* name of link register */
    "setR12",   /* static base register name */
    0,          /* static base register value */
    0x1000,             /* page size */
    0xC0000000ULL,      /* kernel base */
    0xC0000000ULL,      /* kernel text mask */
    0x3FFFFFFFULL,      /* user stack top */
    4,          /* quantization of pc */
    4,          /* szaddr */
    4,          /* szreg */
    4,          /* szfloat */
    8,          /* szdouble */
};
```

Uses REGSIZE-175 308e and armreglist 308f.

⟨libmach/5.c 309a⟩≡

```
/*
 * arm definition
 */
#include <u.h>
#include <libc.h>
#include <bio.h>
#include "../../include/arch/arm/ureg.h"
#include <mach.h>

⟨function REGOFF(arm) 308b⟩

⟨constant SP(arm) 308c⟩
⟨constant PC(arm) 308d⟩
⟨constant REGSIZE(arm) 308e⟩
⟨global armreglist(arm) 308f⟩
⟨global marm(arm) 308g⟩
```

F.2.2 libmach/5db.c

⟨libmach/5db.c 309b⟩≡

```
#include <u.h>
#include <libc.h>
#include <bio.h>
#include <mach.h>

⟨global debug (libmach/5db.c)(arm) 228b⟩

⟨function BITS(arm) 212b⟩

⟨function LSR(arm) 212c⟩
⟨function ASR(arm) 212d⟩
⟨function ROR(arm) 212e⟩

typedef struct  Instr   Instr;
⟨struct Instr(arm) 213⟩

typedef struct Opcode Opcode;
⟨struct Opcode(arm) 214a⟩

static  void    format(char*, Instr*, char*);
⟨global FRAMENAME(arm) 227e⟩

/*
```

```
 * Arm-specific debugger interface
 */

static  char    *armexcep(Map*, Rgetter);
static  int     armfoll(Map*, uvlong, Rgetter, uvlong*);
static  int     arminst(Map*, uvlong, char, char*, int);
static  int     armdas(Map*, uvlong, char*, int);
static  int     arminstlen(Map*, uvlong);
```

⟨*global* armmach*(arm)* 52a⟩

⟨*function* armexcep*(arm)* 52b⟩

⟨*global* cond*(arm)* 226b⟩

⟨*global* shtype*(arm)* 226c⟩

⟨*global* hb*(arm)* 226d⟩

⟨*global* addsub*(arm)* 227a⟩

⟨*function* armclass*(arm)* 219⟩

⟨*function* decode*(arm)* 217⟩

```
#pragma varargck        argpos  bprint          2
```

⟨*function* bprint*(arm)* 226a⟩

⟨*function* plocal*(arm)* 227f⟩

⟨*function* gsymoff*(arm)* 228c⟩

⟨*function* armdps*(arm)* 229a⟩

⟨*function* armdpi*(arm)* 229b⟩

⟨*function* armsdti*(arm)* 230a⟩

⟨*function* armvstdi*(arm)* 230b⟩

⟨*function* armhwby*(arm)* 231a⟩

⟨*function* armsdts*(arm)* 231b⟩

⟨*function* armbdt*(arm)* 231c⟩

⟨*function* armund*(arm)* 231d⟩

⟨*function* armcdt*(arm)* 231e⟩

⟨*function* armunk*(arm)* 231f⟩

⟨*function* armb*(arm)* 232a⟩

⟨*function* armbpt*(arm)* 232b⟩

⟨*function* armco*(arm)* 232c⟩

⟨*function* armcondpass*(arm)* 236b⟩

⟨*function* armshiftval*(arm)* 237⟩

⟨*function* nbits*(arm)* 212f⟩

⟨*function* armmaddr*(arm)* 235a⟩

⟨*function* armaddr*(arm)* 236a⟩

⟨*function* armfadd*(arm)* 234a⟩

⟨*function* armfbx*(arm)* 234b⟩

⟨*function* armfmovm*(arm)* 234c⟩

⟨*function* armfbranch*(arm)* 235b⟩

⟨*function* armfmov*(arm)* 235c⟩

⟨*global* opcodes*(arm)* 214b⟩

⟨*function* gaddr*(arm)* 228a⟩

⟨*global* mode*(arm)* 227b⟩
⟨*global* pw*(arm)* 227c⟩
⟨*global* sw*(arm)* 227d⟩

⟨*function* format*(arm)* 222⟩

⟨*function* printins*(arm)* 211b⟩

⟨*function* arminst*(arm)* 211a⟩

⟨*function* armdas*(arm)* 211d⟩

⟨*function* arminstlen*(arm)* 211c⟩

⟨*function* armfoll*(arm)* 233⟩

F.2.3 libmach/5obj.c

⟨*struct* Addr*(arm)* 311a⟩≡ (313c)

```
struct Addr
{
    char        type;
    char        sym;
    char        name;
};
```

⟨*function* _is5*(arm)* 311b⟩≡ (313c)

```
int
_is5(char *s)
{
    return  s[0] == ANAME                   /* ANAME */
        && s[1] == N_FILE                   /* type */
        && s[2] == 1                        /* sym */
        && s[3] == '<';                     /* name of file */
}
```

⟨*function* `_read5` *(arm)* 312a⟩≡ (313c)

```
int
_read5(Biobuf *bp, Prog *p)
{
    int as, n;
    Addr a;

    as = Bgetc(bp);                     /* as */
    if(as < 0)
        return 0;
    p->kind = aNone;
    p->sig = 0;
    if(as == ANAME || as == ASIGNAME){
        if(as == ASIGNAME){
            Bread(bp, &p->sig, 4);
            p->sig = leswal(p->sig);
        }
        p->kind = aName;
        p->type = type2char(Bgetc(bp));         /* type */
        p->sym = Bgetc(bp);                     /* sym */
        n = 0;
        for(;;) {
            as = Bgetc(bp);
            if(as < 0)
                return 0;
            n++;
            if(as == 0)
                break;
        }
        p->id = malloc(n);
        if(p->id == 0)
            return 0;
        Bseek(bp, -n, 1);
        if(Bread(bp, p->id, n) != n)
            return 0;
        return 1;
    }
    if(as == ATEXT)
        p->kind = aText;
    else if(as == AGLOBL)
        p->kind = aData;
    skip(bp, 6);                /* scond(1), reg(1), lineno(4) */
    a = addr(bp);
    addr(bp);
    if(a.type != D_OREG || a.name != N_INTERN && a.name != N_EXTERN)
        p->kind = aNone;
    p->sym = a.sym;
    return 1;
}
```

Uses `aData` 314e, `aName` 314e, `aNone` 314e, `aText` 314e, and `leswal()` 316b.

⟨*function* `addr` *(arm)* 312b⟩≡ (313c)

```
static Addr
addr(Biobuf *bp)
{
    Addr a;
    long off;

    a.type = Bgetc(bp); /* a.type */
    skip(bp,1);         /* reg */
```

```
        a.sym = Bgetc(bp);  /* sym index */
        a.name = Bgetc(bp); /* sym type */
        switch(a.type){
        default:
        case D_NONE:
        case D_REG:
        case D_FREG:
        case D_PSR:
        case D_FPCR:
            break;
        case D_OREG:
        case D_CONST:
        case D_BRANCH:
        case D_SHIFT:
            off = Bgetc(bp);
            off |= Bgetc(bp) << 8;
            off |= Bgetc(bp) << 16;
            off |= Bgetc(bp) << 24;
            if(off < 0)
                off = -off;
            if(a.sym && (a.name==N_PARAM || a.name==N_LOCAL))
                _offset(a.sym, off);
            break;
        case D_SCONST:
            skip(bp, NSNAME);
            break;
        case D_FCONST:
            skip(bp, 8);
            break;
        }
        return a;
    }
```

Uses _offset() 367b.

⟨*function* type2char*(arm)* 313a⟩≡ (313c)

```
    static char
    type2char(int t)
    {
        switch(t){
        case N_EXTERN:              return 'U';
        case N_INTERN:              return 'b';
        case N_LOCAL:               return 'a';
        case N_PARAM:               return 'p';
        default:            return UNKNOWN;
        }
    }
```

Uses UNKNOWN 314d.

⟨*function* skip*(arm)* 313b⟩≡ (313c)

```
    static void
    skip(Biobuf *bp, int n)
    {
        while (n-- > 0)
            Bgetc(bp);
    }
```

⟨libmach/5obj.c 313c⟩≡

```
    /*
     * 5obj.c - identify and parse an arm object file
     */
```

```
#include <u.h>
#include <libc.h>
#include <bio.h>
#include <mach.h>

#include <5.out.h>
#include "obj.h"

typedef struct Addr     Addr;
⟨struct Addr (arm) 311a⟩
static Addr addr(Biobuf*);
static char type2char(int);
static void skip(Biobuf*, int);

⟨function _is5 (arm) 311b⟩

⟨function _read5 (arm) 312a⟩

⟨function addr (arm) 312b⟩

⟨function type2char (arm) 313a⟩

⟨function skip (arm) 313b⟩
```

F.2.4 libmach/elf.h

⟨constant LOAD 314a⟩≡ (314b)
```
//TODO: could remove this file?
#define LOAD PT_LOAD
```

⟨libmach/elf.h 314b⟩≡
```
// see include/elf.h
#include <elf.h>

⟨constant LOAD 314a⟩
```

F.2.5 libmach/obj.h

⟨struct Prog (libmach/obj.h) 314c⟩≡ (314e)
```
struct Prog             /* info from .$O files */
{
    Kind        kind;           /* what kind of symbol */
    char        type;           /* type of the symbol: ie, 'T', 'a', etc. */
    char        sym;            /* index of symbol's name */
    char        *id;            /* name for the symbol, if it introduces one */
    uint        sig;            /* type signature for symbol */
};
```

⟨constant UNKNOWN 314d⟩≡ (314e)
```
#define UNKNOWN '?'
```

⟨libmach/obj.h 314e⟩≡
```
/*
 * obj.h -- defs for dealing with object files
 */

typedef enum Kind               /* variable defs and references in obj */
{
```

```
    aNone,                      /* we don't care about this prog */
    aName,                      /* introduces a name */
    aText,                      /* starts a function */
    aData,                      /* references to a global object */
} Kind;

typedef struct  Prog    Prog;
```

⟨*struct Prog (libmach/obj.h)* 314c⟩

⟨*constant* UNKNOWN 314d⟩

```
void            _offset(int, vlong);
```

Uses Kind 314e and Prog 314c.

F.2.6 libmach/swap.c

⟨*function* beswab 315a⟩≡ (316d)

```
/*
 * big-endian short
 */
ushort
beswab(ushort s)
{
    uchar *p;

    p = (uchar*)&s;
    return (p[0]<<8) | p[1];
}
```

⟨*function* beswal 315b⟩≡ (316d)

```
/*
 * big-endian long
 */
ulong
beswal(ulong l)
{
    uchar *p;

    p = (uchar*)&l;
    return (p[0]<<24) | (p[1]<<16) | (p[2]<<8) | p[3];
}
```

⟨*function* beswav 315c⟩≡ (316d)

```
/*
 * big-endian vlong
 */
uvlong
beswav(uvlong v)
{
    uchar *p;

    p = (uchar*)&v;
    return ((uvlong)p[0]<<56) | ((uvlong)p[1]<<48) | ((uvlong)p[2]<<40)
                | ((uvlong)p[3]<<32) | ((uvlong)p[4]<<24)
                | ((uvlong)p[5]<<16) | ((uvlong)p[6]<<8)
                | (uvlong)p[7];
}
```

⟨*function* leswab 316a⟩≡ (316d)
```
/*
 * little-endian short
 */
ushort
leswab(ushort s)
{
    uchar *p;

    p = (uchar*)&s;
    return (p[1]<<8) | p[0];
}
```

⟨*function* leswal 316b⟩≡ (316d)
```
/*
 * little-endian long
 */
ulong
leswal(ulong l)
{
    uchar *p;

    p = (uchar*)&l;
    return (p[3]<<24) | (p[2]<<16) | (p[1]<<8) | p[0];
}
```

⟨*function* leswav 316c⟩≡ (316d)
```
/*
 * little-endian vlong
 */
uvlong
leswav(uvlong v)
{
    uchar *p;

    p = (uchar*)&v;
    return ((uvlong)p[7]<<56) | ((uvlong)p[6]<<48) | ((uvlong)p[5]<<40)
                | ((uvlong)p[4]<<32) | ((uvlong)p[3]<<24)
                | ((uvlong)p[2]<<16) | ((uvlong)p[1]<<8)
                | (uvlong)p[0];
}
```

⟨libmach/swap.c 316d⟩≡
```
#include <u.h>

⟨function beswab 315a⟩
⟨function beswal 315b⟩
⟨function beswav 315c⟩

⟨function leswab 316a⟩
⟨function leswal 316b⟩
⟨function leswav 316c⟩
```

F.2.7 libmach/executable.c

⟨*function* hswal 316e⟩≡ (321e)
```
/*
 * Convert header to canonical form
 */
```

```
static void
hswal(void *v, int n, ulong (*swap)(ulong))
{
    ulong *ulp;

    for(ulp = v; n--; ulp++)
        *ulp = (*swap)(*ulp);
}
```

⟨*function* adotout 317a⟩≡ (321e)

```
/*
 *      Crack a normal a.out-type header
 */
static int
adotout(int fd, Fhdr *fp, ExecHdr *hp)
{
    long pgsize;

    USED(fd);
    pgsize = mach->pgsize;
    settext(fp, hp->e.entry, pgsize+sizeof(Exec),
            hp->e.text, sizeof(Exec));
    setdata(fp, _round(pgsize+fp->txtsz+sizeof(Exec), pgsize),
        hp->e.data, fp->txtsz+sizeof(Exec), hp->e.bss);
    setsym(fp, hp->e.syms, hp->e._unused, hp->e.pcsz, fp->datoff+fp->datsz);
    return 1;
}
```

Uses _round() 321d, mach 51a, setdata() 321b, and settext() 321a.

⟨*function* commonboot 317b⟩≡ (321e)

```
static void
commonboot(Fhdr *fp)
{
    if (!(fp->entry & mach->ktmask))
        return;

    switch(fp->type) {                          /* boot image */
    case FI386:
        fp->type = FI386B;
        fp->txtaddr = (u32int)fp->entry;
        fp->name = "386 plan 9 boot image";
        fp->dataddr = _round(fp->txtaddr+fp->txtsz, mach->pgsize);
        break;
    case FARM:
        fp->type = FARMB;
        fp->txtaddr = (u32int)fp->entry;
        fp->name = "ARM plan 9 boot image";
        fp->dataddr = _round(fp->txtaddr+fp->txtsz, mach->pgsize);
        return;
    default:
        return;
    }
    fp->hdrsz = 0;                      /* header stripped */
}
```

Uses _round() 321d and mach 51a.

⟨*function* common 317c⟩≡ (321e)

```
/*
 *      _MAGIC() style headers and
 *      alpha plan9-style bootable images for axp "headerless" boot
```

```
 *
 */
static int
common(int fd, Fhdr *fp, ExecHdr *hp)
{
    adotout(fd, fp, hp);
    if(hp->e.magic & DYN_MAGIC) {
        fp->txtaddr = 0;
        fp->dataddr = fp->txtsz;
        return 1;
    }
    commonboot(fp);
    return 1;
}
```

Uses adotout() 317a and commonboot() 317b.

⟨*function* elf32dotout 318⟩≡ (321e)

```
/*
 * ELF32 binaries.
 */
static int
elf32dotout(int fd, Fhdr *fp, ExecHdr *hp)
{
    ulong (*swal)(ulong);
    ushort (*swab)(ushort);
    Ehdr *ep;
    Phdr *ph;
    int i, it, id, is, phsz;

    /* bitswap the header according to the DATA format */
    ep = &hp->e;
    if(ep->ident[DATA] == ELFDATA2LSB) {
        swab = leswab;
        swal = leswal;
    } else if(ep->ident[DATA] == ELFDATA2MSB) {
        swab = beswab;
        swal = beswal;
    } else {
        werrstr("bad ELF32 encoding - not big or little endian");
        return 0;
    }

    ep->type = swab(ep->type);
    ep->machine = swab(ep->machine);
    ep->version = swal(ep->version);
    ep->elfentry = swal(ep->elfentry);
    ep->phoff = swal(ep->phoff);
    ep->shoff = swal(ep->shoff);
    ep->flags = swal(ep->flags);
    ep->ehsize = swab(ep->ehsize);
    ep->phentsize = swab(ep->phentsize);
    ep->phnum = swab(ep->phnum);
    ep->shentsize = swab(ep->shentsize);
    ep->shnum = swab(ep->shnum);
    ep->shstrndx = swab(ep->shstrndx);
    if(ep->type != EXEC || ep->version != CURRENT)
        return 0;

    /* we could definitely support a lot more machines here */
    fp->magic = ELF_MAG;
```

```
fp->hdrsz = (ep->ehsize+ep->phnum*ep->phentsize+16)&~15;
switch(ep->machine) {
case I386:
    mach = &mi386;
    fp->type = FI386;
    fp->name = "386 ELF32 executable";
    break;
case ARM:
    mach = &marm;
    fp->type = FARM;
    fp->name = "arm ELF32 executable";
    break;
case MIPS:
    mach = &mmips;
    fp->type = FMIPS;
    fp->name = "mips ELF32 executable";
    break;
default:
    return 0;
}

if(ep->phentsize != sizeof(Phdr)) {
    werrstr("bad ELF32 header size");
    return 0;
}
phsz = sizeof(Phdr)*ep->phnum;
ph = malloc(phsz);
if(!ph)
    return 0;
seek(fd, ep->phoff, 0);
if(read(fd, ph, phsz) < 0) {
    free(ph);
    return 0;
}
hswal(ph, phsz/sizeof(ulong), swal);

/* find text, data and symbols and install them */
it = id = is = -1;
for(i = 0; i < ep->phnum; i++) {
    if(ph[i].type == LOAD
    && (ph[i].flags & (R|X)) == (R|X) && it == -1)
        it = i;
    else if(ph[i].type == LOAD
    && (ph[i].flags & (R|W)) == (R|W) && id == -1)
        id = i;
    else if(ph[i].type == NOPTYPE && is == -1)
        is = i;
}
if(it == -1 || id == -1) {
    /*
     * The SPARC64 boot image is something of an ELF hack.
     * Text+Data+BSS are represented by ph[0].  Symbols
     * are represented by ph[1]:
     *
     *              filesz, memsz, vaddr, paddr, off
     * ph[0] : txtsz+datsz, txtsz+datsz+bsssz, txtaddr-KZERO, datasize, txtoff
     * ph[1] : symsz, lcsz, 0, 0, symoff
     */
    if(ep->machine == SPARC64 && ep->phnum == 2) {
        ulong txtaddr, txtsz, dataddr, bsssz;
```

```
            txtaddr = ph[0].vaddr | 0x80000000;
            txtsz = ph[0].filesz - ph[0].paddr;
            dataddr = txtaddr + txtsz;
            bsssz = ph[0].memsz - ph[0].filesz;
            settext(fp, ep->elfentry | 0x80000000, txtaddr, txtsz, ph[0].offset);
            setdata(fp, dataddr, ph[0].paddr, ph[0].offset + txtsz, bsssz);
            setsym(fp, ph[1].filesz, 0, ph[1].memsz, ph[1].offset);
            free(ph);
            return 1;
        }

        werrstr("No ELF32 TEXT or DATA sections");
        free(ph);
        return 0;
    }

    settext(fp, ep->elfentry, ph[it].vaddr, ph[it].memsz, ph[it].offset);
    setdata(fp, ph[id].vaddr, ph[id].filesz, ph[id].offset, ph[id].memsz - ph[id].filesz);
    if(is != -1)
        setsym(fp, ph[is].filesz, 0, ph[is].memsz, ph[is].offset);
    free(ph);
    return 1;
}
```

Uses LOAD 314a, beswab() 315a, beswal() 315b, hswal() 316e, leswab() 316a, leswal() 316b, mach 51a, marm 308g, mi386, mmips, setdata() 321b, and settext() 321a.

⟨*function* elfdotout 320a⟩≡ (321e)

```
/*
 * Elf binaries.
 */
static int
elfdotout(int fd, Fhdr *fp, ExecHdr *hp)
{
    Ehdr *ep;

    /* bitswap the header according to the DATA format */
    ep = &hp->e;
    if(ep->ident[CLASS] == ELFCLASS32)
        return elf32dotout(fd, fp, hp);
    werrstr("bad ELF class - not 32 bit");
    return 0;
}
```

Uses elf32dotout() 318.

⟨*function* armdotout 320b⟩≡ (321e)

```
/*
 * (Free|Net)BSD ARM header.
 */
static int
armdotout(int fd, Fhdr *fp, ExecHdr *hp)
{
    uvlong kbase;

    USED(fd);
    settext(fp, hp->e.entry, sizeof(Exec), hp->e.text, sizeof(Exec));
    setdata(fp, fp->txtsz, hp->e.data, fp->txtsz, hp->e.bss);
    setsym(fp, hp->e.syms, hp->e._unused, hp->e.pcsz, fp->datoff+fp->datsz);

    kbase = 0xF0000000;
```

```
    if ((fp->entry & kbase) == kbase) {         /* Boot image */
        fp->txtaddr = kbase+sizeof(Exec);
        fp->name = "ARM *BSD boot image";
        fp->hdrsz = 0;          /* header stripped */
        fp->dataddr = kbase+fp->txtsz;
    }
    return 1;
}
```

Uses setdata() 321b and settext() 321a.

⟨*function* settext 321a⟩≡ (321e)

```
static void
settext(Fhdr *fp, uvlong e, uvlong a, long s, vlong off)
{
    fp->txtaddr = a;
    fp->entry = e;
    fp->txtsz = s;
    fp->txtoff = off;
}
```

⟨*function* setdata 321b⟩≡ (321e)

```
static void
setdata(Fhdr *fp, uvlong a, long s, vlong off, long bss)
{
    fp->dataddr = a;
    fp->datsz = s;
    fp->datoff = off;
    fp->bsssz = bss;
}
```

⟨*function* setsym 321c⟩≡ (321e)

```
static void
setsym(Fhdr *fp, long symsz, long sppcsz, long lnpcsz, vlong symoff)
{
    fp->symsz = symsz;
    fp->symoff = symoff;
    fp->sppcsz = sppcsz;
    fp->sppcoff = fp->symoff+fp->symsz;
    fp->lnpcsz = lnpcsz;
    fp->lnpcoff = fp->sppcoff+fp->sppcsz;
}
```

⟨*function* _round 321d⟩≡ (321e)

```
static uvlong
_round(uvlong a, ulong b)
{
    uvlong w;

    w = (a/b)*b;
    if (a!=w)
        w += b;
    return(w);
}
```

⟨libmach/executable.c 321e⟩≡

```
#include        <u.h>
#include        <libc.h>
#include        <bio.h>

#include        <bootexec.h>
```

```
#include        <mach.h>

#include        "elf.h"

/*
 *      All a.out header types.  The dummy entry allows canonical
 *      processing of the union as a sequence of longs
 */

typedef struct {
    union{
        struct {
            Exec;               /* a.out.h */
            uvlong hdr[1];
        };
        Ehdr;                   /* elf.h */
        //TODO: struct mipsexec;       /* bootexec.h */
    } e;
    long dummy;                 /* padding to ensure extra long */
} ExecHdr;

static  int     common(int, Fhdr*, ExecHdr*);

static  int     adotout(int, Fhdr*, ExecHdr*);
static  int     elfdotout(int, Fhdr*, ExecHdr*);
static  int     armdotout(int, Fhdr*, ExecHdr*);

static  void    setsym(Fhdr*, long, long, long, vlong);
static  void    setdata(Fhdr*, uvlong, long, vlong, long);
static  void    settext(Fhdr*, uvlong, uvlong, long, vlong);
static  void    hswal(void*, int, ulong(*)(ulong));
static  uvlong  _round(uvlong, ulong);
```

⟨*struct* Exectable 89a⟩

```
//PAD: removed many archi
extern  Mach    mi386;
extern  Mach    marm;
extern  Mach    mmips;
```

⟨*global* exectab 89b⟩

⟨*global* mach 51a⟩

⟨*function* crackhdr 90⟩

⟨*function* hswal 316e⟩

⟨*function* adotout 317a⟩

⟨*function* commonboot 317b⟩

⟨*function* common 317c⟩

⟨*function* elf32dotout 318⟩

⟨*function* elfdotout 320a⟩

⟨*function* armdotout 320b⟩

⟨*function* settext 321a⟩

⟨*function* setdata 321b⟩

⟨*function* setsym 321c⟩

⟨*function* _round 321d⟩

Uses __anon_struct_10 321e, __anon_struct_11 321e, and __anon_struct_9 321e.

F.2.8 libmach/map.c

⟨*function* newmap 323a⟩≡ (324b)

```
Map *
newmap(Map *map, int n)
{
    int size;

    size = sizeof(Map)+(n-1)*sizeof(struct segment);
    if (map == 0)
        map = malloc(size);
    else
        map = realloc(map, size);
    if (map == 0) {
        werrstr("out of memory: %r");
        return 0;
    }
    memset(map, 0, size);
    map->nsegs = n;
    return map;
}
```

⟨*function* setmap 323b⟩≡ (324b)

```
int
setmap(Map *map, int fd, uvlong b, uvlong e, vlong f, char *name)
{
    int i;

    if (map == 0)
        return 0;
    for (i = 0; i < map->nsegs; i++)
        if (!map->seg[i].inuse)
            break;
    if (i >= map->nsegs)
        return 0;
    map->seg[i].b = b;
    map->seg[i].e = e;
    map->seg[i].f = f;
    map->seg[i].inuse = 1;
    map->seg[i].name = name;
    map->seg[i].fd = fd;
    return 1;
}
```

⟨*function* stacktop 323c⟩≡ (324b)

```
static uvlong
stacktop(int pid)
{
    char buf[64];
```

```
    int fd;
    int n;
    char *cp;

    snprint(buf, sizeof(buf), "/proc/%d/segment", pid);
    fd = open(buf, 0);
    if (fd < 0)
        return 0;
    n = read(fd, buf, sizeof(buf)-1);
    close(fd);
    buf[n] = 0;
    if (strncmp(buf, "Stack", 5))
        return 0;
    for (cp = buf+5; *cp && *cp == ' '; cp++)
        ;
    if (!*cp)
        return 0;
    cp = strchr(cp, ' ');
    if (!cp)
        return 0;
    while (*cp && *cp == ' ')
        cp++;
    if (!*cp)
        return 0;
    return strtoull(cp, 0, 16);
}
```

⟨*function* unusemap 324a⟩≡ (324b)

```
void
unusemap(Map *map, int i)
{
    if (map != 0 && 0 <= i && i < map->nsegs)
        map->seg[i].inuse = 0;
}
```

⟨libmach/map.c 324b⟩≡

```
/*
 * file map routines
 */
#include <u.h>
#include <libc.h>
#include <bio.h>
#include <mach.h>

⟨function newmap 323a⟩

⟨function setmap 323b⟩

⟨function stacktop 323c⟩

⟨function attachproc 96b⟩

⟨function findseg 98⟩

⟨function unusemap 324a⟩

⟨function loadmap 92⟩
```

F.2.9 libmach/sym.c

⟨*constant* HUGEINT 325a⟩≡ (347)
```
#define HUGEINT 0x7fffffff
```

⟨*constant* NNAME 325b⟩≡ (347)
```
#define NNAME   20              /* a relic of the past */
```

⟨*struct* txtsym 325c⟩≡ (347)
```
struct txtsym {                         /* Text Symbol table */
    int         n;                      /* number of local vars */
    Sym **locals;               /* array of ptrs to autos */
    Sym *sym;                   /* function symbol entry */
};
```

⟨*struct* hist 325d⟩≡ (347)
```
struct hist {                           /* Stack of include files & #line directives */
    char        *name;                  /* Assumes names Null terminated in file */
    long        line;                   /* line # where it was included */
    long        offset;                 /* line # of #line directive */
};
```

⟨*struct* file 325e⟩≡ (347)
```
struct file {                           /* Per input file header to history stack */
    uvlong      addr;                   /* address of first text sym */
    union {
        Txtsym  *txt;           /* first text symbol */
        Sym     *sym;           /* only during initilization */
    };
    int n;                      /* size of history stack */
    Hist        *hist;                  /* history stack */
};
```
Uses __anon_struct_12 325e.

⟨*global debug (libmach/sym.c)* 325f⟩≡ (347)
```
static  int     debug = 0;
```
Uses debug-98 325f.

⟨*global* autos 325g⟩≡ (347)
```
static  Sym     **autos;                /* Base of auto variables */
```

⟨*global* files 325h⟩≡ (347)
```
static  File    *files;                 /* Base of file arena */
```

⟨*global* fmax 325i⟩≡ (347)
```
static  int     fmax;                   /* largest file path index */
```

⟨*global fnames (libmach/sym.c)* 325j⟩≡ (347)
```
static  Sym     **fnames;               /* file names path component table */
```

⟨*global* globals 325k⟩≡ (347)
```
static  Sym     **globals;              /* globals by addr table */
```

⟨*global* hist 325l⟩≡ (347)
```
static  Hist    *hist;                  /* base of history stack */
```

⟨*global* isbuilt 325m⟩≡ (347)
```
static  int     isbuilt;                /* internal table init flag */
```

⟨*global* nauto 325n⟩≡ (347)
```
static  long    nauto;                  /* number of automatics */
```

⟨*global* nfiles 326a⟩≡ (347)
```
static  long    nfiles;                 /* number of files */
```

⟨*global* nglob 326b⟩≡ (347)
```
static  long    nglob;                  /* number of globals */
```

⟨*global* nhist 326c⟩≡ (347)
```
static  long    nhist;                  /* number of history stack entries */
```

⟨*global* ntxt 326d⟩≡ (347)
```
static  int     ntxt;                   /* number of text symbols */
```

⟨*global* pcline 326e⟩≡ (347)
```
static  uchar   *pcline;                /* start of pc-line state table */
```

⟨*global* pclineend 326f⟩≡ (347)
```
static  uchar   *pclineend;             /* end of pc-line table */
```

⟨*global* spoff 326g⟩≡ (347)
```
static  uchar   *spoff;                 /* start of pc-sp state table */
```

⟨*global* spoffend 326h⟩≡ (347)
```
static  uchar   *spoffend;              /* end of pc-sp offset table */
```

⟨*global* txt 326i⟩≡ (347)
```
static  Txtsym  *txt;                   /* Base of text symbol table */
```

⟨*global* txtstart 326j⟩≡ (347)
```
static  uvlong  txtstart;               /* start of text segment */
```

⟨*global* txtend 326k⟩≡ (347)
```
static  uvlong  txtend;                 /* end of text segment */
```

⟨*function* symerrmsg 326l⟩≡ (347)
```
static int
symerrmsg(int n, char *table)
{
    werrstr("can't read %d bytes of %s table", n, table);
    return -1;
}
```

⟨*function* decodename 326m⟩≡ (347)
```
static long
decodename(Biobuf *bp, Sym *p)
{
    char *cp;
    int c1, c2;
    long n;
    vlong o;

    if((p->type & 0x80) == 0) {         /* old-style, fixed length names */
        p->name = malloc(NNAME);
        if(p->name == 0) {
            werrstr("can't malloc %d bytes", NNAME);
            return -1;
        }
        if(Bread(bp, p->name, NNAME) != NNAME)
            return symerrmsg(NNAME, "symbol");
        Bseek(bp, 3, 1);
        return NNAME+3;
    }
```

```
    p->type &= ~0x80;
    if(p->type == 'z' || p->type == 'Z') {
        o = Bseek(bp, 0, 1);
        if(Bgetc(bp) < 0) {
            werrstr("can't read symbol name");
            return -1;
        }
        for(;;) {
            c1 = Bgetc(bp);
            c2 = Bgetc(bp);
            if(c1 < 0 || c2 < 0) {
                werrstr("can't read symbol name");
                return -1;
            }
            if(c1 == 0 && c2 == 0)
                break;
        }
        n = Bseek(bp, 0, 1)-o;
        p->name = malloc(n);
        if(p->name == 0) {
            werrstr("can't malloc %ld bytes", n);
            return -1;
        }
        Bseek(bp, -n, 1);
        if(Bread(bp, p->name, n) != n) {
            werrstr("can't read %ld bytes of symbol name", n);
            return -1;
        }
    } else {
        cp = Brdline(bp, '\0');
        if(cp == 0) {
            werrstr("can't read symbol name");
            return -1;
        }
        n = Blinelen(bp);
        p->name = malloc(n);
        if(p->name == 0) {
            werrstr("can't malloc %ld bytes", n);
            return -1;
        }
        strcpy(p->name, cp);
    }
    return n;
}
```

Uses NNAME-97 325b and symerrmsg() 326l.

⟨*function* cleansyms 327⟩≡ (347)

```
/*
 *      free any previously loaded symbol tables
 */
static void
cleansyms(void)
{
    if(globals)
        free(globals);
    globals = 0;
    nglob = 0;
    if(txt)
        free(txt);
```

```
    txt = 0;
    ntxt = 0;
    if(fnames)
        free(fnames);
    fnames = 0;
    fmax = 0;

    if(files)
        free(files);
    files = 0;
    nfiles = 0;
    if(hist)
        free(hist);
    hist = 0;
    nhist = 0;
    if(autos)
        free(autos);
    autos = 0;
    nauto = 0;
    isbuilt = 0;
    if(symbols)
        free(symbols);
    symbols = 0;
    nsym = 0;
    if(spoff)
        free(spoff);
    spoff = 0;
    if(pcline)
        free(pcline);
    pcline = 0;
}
```

Uses autos-99 325g, files-100 325h, fmax-101 325i, fnames-102 325j, globals-103 325k, hist-104 325l, isbuilt-105 325m, nauto-106 325n, nfiles-107 326a, nglob-108 326b, nhist-109 326c, nsym-110 93a, ntxt-111 326d, pcline-112 326e, spoff-114 326g, symbols-116 93b, and txt-117 326i.

⟨*function* textseg 328a⟩≡ (347)

```
/*
 *      delimit the text segment
 */
void
textseg(uvlong base, Fhdr *fp)
{
    txtstart = base;
    txtend = base+fp->txtsz;
}
```

Uses txtend-119 326k and txtstart-118 326j.

⟨*function* getsym 328b⟩≡ (347)

```
/*
 *      Get the ith symbol table entry
 */
Sym *
getsym(int index)
{
    if(index >= 0 && index < nsym)
        return &symbols[index];
    return 0;
}
```

Uses nsym-110 93a and symbols-116 93b.

⟨*function* buildtbls 329⟩≡ (347)

```
/*
 *      initialize internal symbol tables
 */
static int
buildtbls(void)
{
    long i;
    int j, nh, ng, nt;
    File *f;
    Txtsym *tp;
    Hist *hp;
    Sym *p, **ap;

    if(isbuilt)
        return 1;
    isbuilt = 1;
            /* allocate the tables */
    if(nglob) {
        globals = malloc(nglob*sizeof(*globals));
        if(!globals) {
            werrstr("can't malloc global symbol table");
            return 0;
        }
    }
    if(ntxt) {
        txt = malloc(ntxt*sizeof(*txt));
        if (!txt) {
            werrstr("can't malloc text symbol table");
            return 0;
        }
    }
    fnames = malloc((fmax+1)*sizeof(*fnames));
    if (!fnames) {
        werrstr("can't malloc file name table");
        return 0;
    }
    memset(fnames, 0, (fmax+1)*sizeof(*fnames));
    files = malloc(nfiles*sizeof(*files));
    if(!files) {
        werrstr("can't malloc file table");
        return 0;
    }
    hist = malloc(nhist*sizeof(Hist));
    if(hist == 0) {
        werrstr("can't malloc history stack");
        return 0;
    }
    autos = malloc(nauto*sizeof(Sym*));
    if(autos == 0) {
        werrstr("can't malloc auto symbol table");
        return 0;
    }
        /* load the tables */
    ng = nt = nh = 0;
    f = 0;
    tp = 0;
    i = nsym;
    hp = hist;
    ap = autos;
```

```
for(p = symbols; i-- > 0; p++) {
    switch(p->type) {
    case 'D':
    case 'd':
    case 'B':
    case 'b':
        if(debug)
            print("Global: %s %llux\n", p->name, p->value);
        globals[ng++] = p;
        break;
    case 'z':
        if(p->value == 1) {         /* New file */
            if(f) {
                f->n = nh;
                f->hist[nh].name = 0;       /* one extra */
                hp += nh+1;
                f++;
            }
            else
                f = files;
            f->hist = hp;
            f->sym = 0;
            f->addr = 0;
            nh = 0;
        }
            /* alloc one slot extra as terminator */
        f->hist[nh].name = p->name;
        f->hist[nh].line = p->value;
        f->hist[nh].offset = 0;
        if(debug)
            printhist("-> ", &f->hist[nh], 1);
        nh++;
        break;
    case 'Z':
        if(f && nh > 0)
            f->hist[nh-1].offset = p->value;
        break;
    case 'T':
    case 't':       /* Text: terminate history if first in file */
    case 'L':
    case 'l':
        tp = &txt[nt++];
        tp->n = 0;
        tp->sym = p;
        tp->locals = ap;
        if(debug)
            print("TEXT: %s at %llux\n", p->name, p->value);
        if(f && !f->sym) {                  /* first  */
            f->sym = p;
            f->addr = p->value;
        }
        break;
    case 'a':
    case 'p':
    case 'm':               /* Local Vars */
        if(!tp)
            print("Warning: Free floating local var: %s\n",
                p->name);
        else {
            if(debug)
```

```
                print("Local: %s %llux\n", p->name, p->value);
            tp->locals[tp->n] = p;
            tp->n++;
            ap++;
        }
        break;
    case 'f':               /* File names */
        if(debug)
            print("Fname: %s\n", p->name);
        fnames[p->value] = p;
        break;
    default:
        break;
    }
}
    /* sort global and text tables into ascending address order */
qsort(globals, nglob, sizeof(Sym*), symcomp);
qsort(txt, ntxt, sizeof(Txtsym), txtcomp);
qsort(files, nfiles, sizeof(File), filecomp);
tp = txt;
for(i = 0, f = files; i < nfiles; i++, f++) {
    for(j = 0; j < ntxt; j++) {
        if(f->sym == tp->sym) {
            if(debug) {
                print("LINK: %s to at %llux", f->sym->name, f->addr);
                printhist("... ", f->hist, 1);
            }
            f->txt = tp++;
            break;
        }
        if(++tp >= txt+ntxt)        /* wrap around */
            tp = txt;
    }
}
    return 1;
}
```

Uses autos-99 325g, debug-98 325f, filecomp() 343b, files-100 325h, fmax-101 325i, fnames-102 325j, globals-103 325k, hist-104 325l, isbuilt-105 325m, nauto-106 325n, nfiles-107 326a, nglob-108 326b, nhist-109 326c, nsym-110 93a, ntxt-111 326d, printhist() 346a, symbols-116 93b, symcomp() 342b, txt-117 326i, and txtcomp() 343a.

⟨*function lookup (libmach/sym.c)* 331⟩≡ (347)

```
/*
 * find symbol function.var by name.
 *      fn != 0 && var != 0     => look for fn in text, var in data
 *      fn != 0 && var == 0     => look for fn in text
 *      fn == 0 && var != 0     => look for var first in text then in data space.
 */
int
lookup(char *fn, char *var, Symbol *s)
{
    int found;

    if(buildtbls() == 0)
        return 0;
    if(fn) {
        found = findtext(fn, s);
        if(var == 0)            /* case 2: fn not in text */
            return found;
        else if(!found)         /* case 1: fn not found */
            return 0;
```

```
    } else if(var) {
        found = findtext(var, s);
        if(found)
            return 1;   /* case 3: var found in text */
    } else return 0;            /* case 4: fn & var == zero */

    if(found)
        return findlocal(s, var, s);    /* case 1: fn found */
    return findglobal(var, s);          /* case 3: var not found */

}
```

Uses `buildtbls()` 329, `findglobal()` 332b, `findlocal()` 332c, and `findtext()` 332a.

⟨*function* `findtext` 332a⟩≡ (347)

```
/*
 * find a function by name
 */
static int
findtext(char *name, Symbol *s)
{
    int i;

    for(i = 0; i < ntxt; i++) {
        if(strcmp(txt[i].sym->name, name) == 0) {
            fillsym(txt[i].sym, s);
            s->handle = (void *) &txt[i];
            s->index = i;
            return 1;
        }
    }
    return 0;
}
```

Uses `fillsym()` 343c, `ntxt-111` 326d, and `txt-117` 326i.

⟨*function* `findglobal` 332b⟩≡ (347)

```
/*
 * find global variable by name
 */
static int
findglobal(char *name, Symbol *s)
{
    long i;

    for(i = 0; i < nglob; i++) {
        if(strcmp(globals[i]->name, name) == 0) {
            fillsym(globals[i], s);
            s->index = i;
            return 1;
        }
    }
    return 0;
}
```

Uses `fillsym()` 343c, `globals-103` 325k, and `nglob-108` 326b.

⟨*function* `findlocal` 332c⟩≡ (347)

```
/*
 *      find the local variable by name within a given function
 */
int
```

```
findlocal(Symbol *s1, char *name, Symbol *s2)
{
    if(s1 == 0)
        return 0;
    if(buildtbls() == 0)
        return 0;
    return findlocvar(s1, name, s2);
}
```

Uses buildtbls() 329 and findlocvar() 333a.

⟨*function* findlocvar 333a⟩≡ (347)

```
/*
 *      find the local variable by name within a given function
 *              (internal function - does no parameter validation)
 */
static int
findlocvar(Symbol *s1, char *name, Symbol *s2)
{
    Txtsym *tp;
    int i;

    tp = (Txtsym *)s1->handle;
    if(tp && tp->locals) {
        for(i = 0; i < tp->n; i++)
            if (strcmp(tp->locals[i]->name, name) == 0) {
                fillsym(tp->locals[i], s2);
                s2->handle = (void *)tp;
                s2->index = tp->n-1 - i;
                return 1;
            }
    }
    return 0;
}
```

Uses fillsym() 343c.

⟨*function* textsym 333b⟩≡ (347)

```
/*
 *      Get ith text symbol
 */
int
textsym(Symbol *s, int index)
{

    if(buildtbls() == 0)
        return 0;
    if(index < 0 || index >= ntxt)
        return 0;
    fillsym(txt[index].sym, s);
    s->handle = (void *)&txt[index];
    s->index = index;
    return 1;
}
```

Uses buildtbls() 329, fillsym() 343c, ntxt-111 326d, and txt-117 326i.

⟨*function* filesym 333c⟩≡ (347)

```
/*
 *      Get ith file name
 */
int
```

```
filesym(int index, char *buf, int n)
{
    Hist *hp;

    if(buildtbls() == 0)
        return 0;
    if(index < 0 || index >= nfiles)
        return 0;
    hp = files[index].hist;
    if(!hp || !hp->name)
        return 0;
    return fileelem(fnames, (uchar*)hp->name, buf, n);
}
```

Uses buildtbls() 329, files-100 325h, fnames-102 325j, and nfiles-107 326a.

⟨*function* getauto 334a⟩≡ (347)

```
/*
 *      Lookup name of local variable located at an offset into the frame.
 *      The type selects either a parameter or automatic.
 */
int
getauto(Symbol *s1, int off, int type, Symbol *s2)
{
    Txtsym *tp;
    Sym *p;
    int i, t;

    if(s1 == 0)
        return 0;
    if(type == CPARAM)
        t = 'p';
    else if(type == CAUTO)
        t = 'a';
    else
        return 0;
    if(buildtbls() == 0)
        return 0;
    tp = (Txtsym *)s1->handle;
    if(tp == 0)
        return 0;
    for(i = 0; i < tp->n; i++) {
        p = tp->locals[i];
        if(p->type == t && p->value == off) {
            fillsym(p, s2);
            s2->handle = s1->handle;
            s2->index = tp->n-1 - i;
            return 1;
        }
    }
    return 0;
}
```

Uses buildtbls() 329 and fillsym() 343c.

⟨*function* srchtext 334b⟩≡ (347)

```
/*
 * Find text symbol containing addr; binary search assumes text array is sorted by addr
 */
static int
srchtext(uvlong addr)
{
```

```
    uvlong val;
    int top, bot, mid;
    Sym *sp;

    val = addr;
    bot = 0;
    top = ntxt;
    for (mid = (bot+top)/2; mid < top; mid = (bot+top)/2) {
        sp = txt[mid].sym;
        if(sp == nil)
            return -1;
        if(val < sp->value)
            top = mid;
        else if(mid != ntxt-1 && val >= txt[mid+1].sym->value)
            bot = mid;
        else
            return mid;
    }
    return -1;
}
```

Uses ntxt-111 326d and txt-117 326i.

⟨*function* srchdata 335a⟩≡ (347)

```
/*
 * Find data symbol containing addr; binary search assumes data array is sorted by addr
 */
static int
srchdata(uvlong addr)
{
    uvlong val;
    int top, bot, mid;
    Sym *sp;

    bot = 0;
    top = nglob;
    val = addr;
    for(mid = (bot+top)/2; mid < top; mid = (bot+top)/2) {
        sp = globals[mid];
        if(sp == nil)
            return -1;
        if(val < sp->value)
            top = mid;
        else if(mid < nglob-1 && val >= globals[mid+1]->value)
            bot = mid;
        else
            return mid;
    }
    return -1;
}
```

Uses globals-103 325k and nglob-108 326b.

⟨*function* findsym 335b⟩≡ (347)

```
/*
 * Find symbol containing val in specified search space
 * There is a special case when a value falls beyond the end
 * of the text segment; if the search space is CTEXT, that value
 * (usually etext) is returned.  If the search space is CANY, symbols in the
 * data space are searched for a match.
 */
int
```

```
findsym(uvlong val, int type, Symbol *s)
{
    int i;

    if(buildtbls() == 0)
        return 0;

    if(type == CTEXT || type == CANY) {
        i = srchtext(val);
        if(i >= 0) {
            if(type == CTEXT || i != ntxt-1) {
                fillsym(txt[i].sym, s);
                s->handle = (void *) &txt[i];
                s->index = i;
                return 1;
            }
        }
    }
    if(type == CDATA || type == CANY) {
        i = srchdata(val);
        if(i >= 0) {
            fillsym(globals[i], s);
            s->index = i;
            return 1;
        }
    }
    return 0;
}
```

Uses buildtbls() 329, fillsym() 343c, globals-103 325k, ntxt-111 326d, srchdata() 335a, srchtext() 334b, and txt-117 326i.

⟨*function* fnbound 336a⟩≡ (347)

```
/*
 *      Find the start and end address of the function containing addr
 */
int
fnbound(uvlong addr, uvlong *bounds)
{
    int i;

    if(buildtbls() == 0)
        return 0;

    i = srchtext(addr);
    if(0 <= i && i < ntxt-1) {
        bounds[0] = txt[i].sym->value;
        bounds[1] = txt[i+1].sym->value;
        return 1;
    }
    return 0;
}
```

Uses buildtbls() 329, ntxt-111 326d, srchtext() 334b, and txt-117 326i.

⟨*function* localsym 336b⟩≡ (347)

```
/*
 * get the ith local symbol for a function
 * the input symbol table is reverse ordered, so we reverse
 * accesses here to maintain approx. parameter ordering in a stack trace.
 */
int
```

```
localsym(Symbol *s, int index)
{
    Txtsym *tp;

    if(s == 0 || index < 0)
        return 0;
    if(buildtbls() == 0)
        return 0;

    tp = (Txtsym *)s->handle;
    if(tp && tp->locals && index < tp->n) {
        fillsym(tp->locals[tp->n-index-1], s);  /* reverse */
        s->handle = (void *)tp;
        s->index = index;
        return 1;
    }
    return 0;
}
```

Uses `buildtbls()` 329 and `fillsym()` 343c.

⟨*function* `globalsym` 337a⟩≡ (347)

```
/*
 * get the ith global symbol
 */
int
globalsym(Symbol *s, int index)
{
    if(s == 0)
        return 0;
    if(buildtbls() == 0)
        return 0;

    if(index >=0 && index < nglob) {
        fillsym(globals[index], s);
        s->index = index;
        return 1;
    }
    return 0;
}
```

Uses `buildtbls()` 329, `fillsym()` 343c, `globals-103` 325k, and `nglob-108` 326b.

⟨*function* `file2pc` 337b⟩≡ (347)

```
/*
 *      find the pc given a file name and line offset into it.
 */
uvlong
file2pc(char *file, long line)
{
    File *fp;
    long i;
    uvlong pc, start, end;
    short *name;

    if(buildtbls() == 0 || files == 0)
        return ~0;
    name = encfname(file);
    if(name == 0) {                     /* encode the file name */
        werrstr("file %s not found", file);
        return ~0;
    }
```

```
        /* find this history stack */
    for(i = 0, fp = files; i < nfiles; i++, fp++)
        if (hline(fp, name, &line))
            break;
    free(name);
    if(i >= nfiles) {
        werrstr("line %ld in file %s not found", line, file);
        return ~0;
    }
    start = fp->addr;           /* first text addr this file */
    if(i < nfiles-1)
        end = (fp+1)->addr;     /* first text addr next file */
    else
        end = 0;                /* last file in load module */
    /*
     * At this point, line contains the offset into the file.
     * run the state machine to locate the pc closest to that value.
     */
    if(debug)
        print("find pc for %ld - between: %llux and %llux\n", line, start, end);
    pc = line2addr(line, start, end);
    if(pc == ~0) {
        werrstr("line %ld not in file %s", line, file);
        return ~0;
    }
    return pc;
}
```

Uses buildtbls() 329, debug-98 325f, encfname() 338b, files-100 325h, line2addr() 345, and nfiles-107 326a.

⟨*function* pathcomp 338a⟩≡ (347)

```
/*
 *      search for a path component index
 */
static int
pathcomp(char *s, int n)
{
    int i;

    for(i = 0; i <= fmax; i++)
        if(fnames[i] && strncmp(s, fnames[i]->name, n) == 0)
            return i;
    return -1;
}
```

Uses fmax-101 325i and fnames-102 325j.

⟨*function* encfname 338b⟩≡ (347)

```
/*
 *      Encode a char file name as a sequence of short indices
 *      into the file name dictionary.
 */
static short*
encfname(char *file)
{
    int i, j;
    char *cp, *cp2;
    short *dest;

    if(*file == '/')    /* always check first '/' */
        cp2 = file+1;
    else {
```

```
        cp2 = strchr(file, '/');
        if(!cp2)
            cp2 = strchr(file, 0);
    }
    cp = file;
    dest = 0;
    for(i = 0; *cp; i++) {
        j = pathcomp(cp, cp2-cp);
        if(j < 0)
            return 0;   /* not found */
        dest = realloc(dest, (i+1)*sizeof(short));
        dest[i] = j;
        cp = cp2;
        while(*cp == '/')       /* skip embedded '/'s */
            cp++;
        cp2 = strchr(cp, '/');
        if(!cp2)
            cp2 = strchr(cp, 0);
    }
    dest = realloc(dest, (i+1)*sizeof(short));
    dest[i] = 0;
    return dest;
}
```

Uses pathcomp() 338a.

⟨*function* hline 339⟩≡ (347)

```
/*
 *      Search a history stack for a matching file name accumulating
 *      the size of intervening files in the stack.
 */
static int
hline(File *fp, short *name, long *line)
{
    Hist *hp;
    int offset, depth;
    long ln;

    for(hp = fp->hist; hp->name; hp++)          /* find name in stack */
        if(hp->name[1] || hp->name[2]) {
            if(hcomp(hp, name))
                break;
        }
    if(!hp->name)               /* match not found */
        return 0;
    if(debug)
        printhist("hline found ... ", hp, 1);
    /*
     * unwind the stack until empty or we hit an entry beyond our line
     */
    ln = *line;
    offset = hp->line-1;
    depth = 1;
    for(hp++; depth && hp->name; hp++) {
        if(debug)
            printhist("hline inspect ... ", hp, 1);
        if(hp->name[1] || hp->name[2]) {
            if(hp->offset){                     /* Z record */
                offset = 0;
                if(hcomp(hp, name)) {
                    if(*line <= hp->offset)
```

```
                        break;
                    ln = *line+hp->line-hp->offset;
                    depth = 1;  /* implicit pop */
                } else
                    depth = 2;  /* implicit push */
            } else if(depth == 1 && ln < hp->line-offset)
                    break;              /* Beyond our line */
            else if(depth++ == 1)               /* push */
                offset -= hp->line;
        } else if(--depth == 1)         /* pop */
            offset += hp->line;
    }
    *line = ln+offset;
    return 1;
}
```

⟨*function* hcomp 340a⟩≡ (347)

```
/*
 *      compare two encoded file names
 */
static int
hcomp(Hist *hp, short *sp)
{
    uchar *cp;
    int i, j;
    short *s;

    cp = (uchar *)hp->name;
    s = sp;
    if (*s == 0)
        return 0;
    for (i = 1; j = (cp[i]<<8)|cp[i+1]; i += 2) {
        if(j == 0)
            break;
        if(*s == j)
            s++;
        else
            s = sp;
    }
    return *s == 0;
}
```

⟨*function* fileline 340b⟩≡ (347)

```
/*
 *      Convert a pc to a "file:line {file:line}" string.
 */
long
fileline(char *str, int n, uvlong dot)
{
    long line, top, bot, mid;
    File *f;

    *str = 0;
    if(buildtbls() == 0)
        return 0;
        /* binary search assumes file list is sorted by addr */
    bot = 0;
    top = nfiles;
    for (mid = (bot+top)/2; mid < top; mid = (bot+top)/2) {
        f = &files[mid];
```

```
        if(dot < f->addr)
            top = mid;
        else if(mid < nfiles-1 && dot >= (f+1)->addr)
            bot = mid;
        else {
            line = pc2line(dot);
            if(line > 0 && fline(str, n, line, f->hist, 0) >= 0)
                return 1;
            break;
        }
    }
    return 0;
}
```

Uses buildtbls() 329, files-100 325h, fline() 341, nfiles-107 326a, and pc2line() 344b.

⟨*function* fline 341⟩≡ (347)

```
/*
 *      Convert a line number within a composite file to relative line
 *      number in a source file.  A composite file is the source
 *      file with included files inserted in line.
 */
static int
fline(char *str, int n, long line, Hist *base, Hist **ret)
{
    Hist *start;                        /* start of current level */
    Hist *h;                    /* current entry */
    long delta;                 /* sum of size of files this level */
    int k;

    start = base;
    h = base;
    delta = h->line;
    while(h && h->name && line > h->line) {
        if(h->name[1] || h->name[2]) {
            if(h->offset != 0) {        /* #line Directive */
                delta = h->line-h->offset+1;
                start = h;
                base = h++;
            } else {            /* beginning of File */
                if(start == base)
                    start = h++;
                else {
                    k = fline(str, n, line, start, &h);
                    if(k <= 0)
                        return k;
                }
            }
        } else {
            if(start == base && ret) {  /* end of recursion level */
                *ret = h;
                return 1;
            } else {                    /* end of included file */
                delta += h->line-start->line;
                h++;
                start = base;
            }
        }
    }
    if(!h)
        return -1;
```

```
    if(start != base)
        line = line-start->line+1;
    else
        line = line-delta+1;
    if(!h->name)
        strncpy(str, "<eof>", n);
    else {
        k = fileelem(fnames, (uchar*)start->name, str, n);
        if(k+8 < n)
            sprint(str+k, ":%ld", line);
    }
/**********Remove comments for complete back-trace of include sequence
 *      if(start != base) {
 *              k = strlen(str);
 *              if(k+2 < n) {
 *                      str[k++] = ' ';
 *                      str[k++] = '{';
 *              }
 *              k += fileelem(fnames, (uchar*) base->name, str+k, n-k);
 *              if(k+10 < n)
 *                      sprint(str+k, ":%ld}", start->line-delta);
 *      }
 ******************/
    return 0;
}
```

Uses `fline()` 341 and `fnames-102` 325j.

⟨*function* `fileelem` 342a⟩≡ (347)

```
/*
 *      convert an encoded file name to a string.
 */
int
fileelem(Sym **fp, uchar *cp, char *buf, int n)
{
    int i, j;
    char *c, *bp, *end;
    Sym *sym;

    bp = buf;
    end = buf+n-1;
    for(i = 1; j = (cp[i]<<8)|cp[i+1]; i+=2){
        sym = fp[j];
        if (sym == nil)
            break;
        c = sym->name;
        if(bp != buf && bp[-1] != '/' && bp < end)
            *bp++ = '/';
        while(bp < end && *c)
            *bp++ = *c++;
    }
    *bp = 0;
    i =  bp-buf;
    if(i > 1) {
        cleanname(buf);
        i = strlen(buf);
    }
    return i;
}
```

⟨*function* `symcomp` 342b⟩≡ (347)

```
/*
 *      compare the values of two symbol table entries.
 */
static int
symcomp(void *a, void *b)
{
    int i;

    i = (*(Sym**)a)->value - (*(Sym**)b)->value;
    if (i)
        return i;
    return strcmp((*(Sym**)a)->name, (*(Sym**)b)->name);
}
```

⟨*function* txtcomp 343a⟩≡ (347)

```
/*
 *      compare the values of the symbols referenced by two text table entries
 */
static int
txtcomp(void *a, void *b)
{
    return ((Txtsym*)a)->sym->value - ((Txtsym*)b)->sym->value;
}
```

⟨*function* filecomp 343b⟩≡ (347)

```
/*
 *      compare the values of the symbols referenced by two file table entries
 */
static int
filecomp(void *a, void *b)
{
    return ((File*)a)->addr - ((File*)b)->addr;
}
```

⟨*function* fillsym 343c⟩≡ (347)

```
/*
 *      fill an interface Symbol structure from a symbol table entry
 */
static void
fillsym(Sym *sp, Symbol *s)
{
    s->type = sp->type;
    s->value = sp->value;
    s->name = sp->name;
    s->index = 0;
    switch(sp->type) {
    case 'b':
    case 'B':
    case 'd':
    case 'D':
        s->class = CDATA;
        break;
    case 't':
    case 'T':
    case 'l':
    case 'L':
        s->class = CTEXT;
        break;
    case 'a':
        s->class = CAUTO;
```

```
            break;
        case 'p':
            s->class = CPARAM;
            break;
        case 'm':
            s->class = CSTAB;
            break;
        default:
            s->class = CNONE;
            break;
        }
        s->handle = 0;
    }
```

⟨*function* pc2sp 344a⟩≡ (347)

```
    /*
     *      find the stack frame, given the pc
     */
    uvlong
    pc2sp(uvlong pc)
    {
        uchar *c, u;
        uvlong currpc, currsp;

        if(spoff == 0)
            return ~0;
        currsp = 0;
        currpc = txtstart - mach->pcquant;

        if(pc<currpc || pc>txtend)
            return ~0;
        for(c = spoff; c < spoffend; c++) {
            if (currpc >= pc)
                return currsp;
            u = *c;
            if (u == 0) {
                currsp += (c[1]<<24)|(c[2]<<16)|(c[3]<<8)|c[4];
                c += 4;
            }
            else if (u < 65)
                currsp += 4*u;
            else if (u < 129)
                currsp -= 4*(u-64);
            else
                currpc += mach->pcquant*(u-129);
            currpc += mach->pcquant;
        }
        return ~0;
    }
```

Uses `mach` 51a, `spoff-114` 326g, `spoffend-115` 326h, `txtend-119` 326k, and `txtstart-118` 326j.

⟨*function* pc2line 344b⟩≡ (347)

```
    /*
     *      find the source file line number for a given value of the pc
     */
    long
    pc2line(uvlong pc)
    {
        uchar *c, u;
        uvlong currpc;
```

```
    long currline;

    if(pcline == 0)
        return -1;
    currline = 0;
    currpc = txtstart-mach->pcquant;
    if(pc<currpc || pc>txtend)
        return ~0;

    for(c = pcline; c < pclineend; c++) {
        if(currpc >= pc)
            return currline;
        u = *c;
        if(u == 0) {
            currline += (c[1]<<24)|(c[2]<<16)|(c[3]<<8)|c[4];
            c += 4;
        }
        else if(u < 65)
            currline += u;
        else if(u < 129)
            currline -= (u-64);
        else
            currpc += mach->pcquant*(u-129);
        currpc += mach->pcquant;
    }
    return ~0;
}
```

Uses mach 51a, pcline-112 326e, pclineend-113 326f, txtend-119 326k, and txtstart-118 326j.

⟨*function* line2addr 345⟩≡ (347)

```
/*
 *      find the pc associated with a line number
 *      basepc and endpc are text addresses bounding the search.
 *      if endpc == 0, the end of the table is used (i.e., no upper bound).
 *      usually, basepc and endpc contain the first text address in
 *      a file and the first text address in the following file, respectively.
 */
uvlong
line2addr(long line, uvlong basepc, uvlong endpc)
{
    uchar *c,  u;
    uvlong currpc, pc;
    long currline;
    long delta, d;
    int found;

    if(pcline == 0 || line == 0)
        return ~0;

    currline = 0;
    currpc = txtstart-mach->pcquant;
    pc = ~0;
    found = 0;
    delta = HUGEINT;

    for(c = pcline; c < pclineend; c++) {
        if(endpc && currpc >= endpc)    /* end of file of interest */
            break;
        if(currpc >= basepc) {          /* proper file */
            if(currline >= line) {
```

```
                d = currline-line;
                found = 1;
            } else
                d = line-currline;
            if(d < delta) {
                delta = d;
                pc = currpc;
            }
        }
        u = *c;
        if(u == 0) {
            currline += (c[1]<<24)|(c[2]<<16)|(c[3]<<8)|c[4];
            c += 4;
        }
        else if(u < 65)
            currline += u;
        else if(u < 129)
            currline -= (u-64);
        else
            currpc += mach->pcquant*(u-129);
        currpc += mach->pcquant;
    }
    if(found)
        return pc;
    return ~0;
}
```

Uses HUGEINT-96 325a, mach 51a, pcline-112 326e, pclineend-113 326f, and txtstart-118 326j.

⟨*function* printhist 346a⟩≡ (347)

```
/*
 *      Print a history stack (debug). if count is 0, prints the whole stack
 */
static void
printhist(char *msg, Hist *hp, int count)
{
    int i;
    uchar *cp;
    char buf[128];

    i = 0;
    while(hp->name) {
        if(count && ++i > count)
            break;
        print("%s Line: %lx (%ld)  Offset: %lx (%ld)  Name: ", msg,
            hp->line, hp->line, hp->offset, hp->offset);
        for(cp = (uchar *)hp->name+1; (*cp<<8)|cp[1]; cp += 2) {
            if (cp != (uchar *)hp->name+1)
                print("/");
            print("%x", (*cp<<8)|cp[1]);
        }
        fileelem(fnames, (uchar *) hp->name, buf, sizeof(buf));
        print(" (%s)\n", buf);
        hp++;
    }
}
```

Uses fnames-102 325j.

⟨*function* dumphist 346b⟩≡ (347)

```
/*
 *      print the history stack for a file. (debug only)
```

```
 *      if (name == 0) => print all history stacks.
 */
void
dumphist(char *name)
{
    int i;
    File *f;
    short *fname;

    if(buildtbls() == 0)
        return;
    if(name)
        fname = encfname(name);
    for(i = 0, f = files; i < nfiles; i++, f++)
        if(fname == 0 || hcomp(f->hist, fname))
            printhist("> ", f->hist, f->n);

    if(fname)
        free(fname);
}
```

⟨libmach/sym.c 347⟩≡

```
#include <u.h>
#include <libc.h>
#include <bio.h>
#include <mach.h>

⟨constant HUGEINT 325a⟩
⟨constant NNAME 325b⟩

typedef struct txtsym Txtsym;
typedef struct file File;
typedef struct hist Hist;

⟨struct txtsym 325c⟩

⟨struct hist 325d⟩

⟨struct file 325e⟩

⟨global debug (libmach/sym.c) 325f⟩

⟨global autos 325g⟩
⟨global files 325h⟩
⟨global fmax 325i⟩
⟨global fnames (libmach/sym.c) 325j⟩
⟨global globals 325k⟩
⟨global hist 325l⟩
⟨global isbuilt 325m⟩
⟨global nauto 325n⟩
⟨global nfiles 326a⟩
⟨global nglob 326b⟩
⟨global nhist 326c⟩
⟨global nsym (libmach/sym.c) 93a⟩
⟨global ntxt 326d⟩
⟨global pcline 326e⟩
⟨global pclineend 326f⟩
⟨global spoff 326g⟩
⟨global spoffend 326h⟩
⟨global symbols 93b⟩
```

⟨*global* `txt` 326i⟩
⟨*global* `txtstart` 326j⟩
⟨*global* `txtend` 326k⟩

```
static void     cleansyms(void);
static long     decodename(Biobuf*, Sym*);
static short    *encfname(char*);
static int      fline(char*, int, long, Hist*, Hist**);
static void     fillsym(Sym*, Symbol*);
static int      findglobal(char*, Symbol*);
static int      findlocvar(Symbol*, char *, Symbol*);
static int      findtext(char*, Symbol*);
static int      hcomp(Hist*, short*);
static int      hline(File*, short*, long*);
static void     printhist(char*, Hist*, int);
static int      buildtbls(void);
static int      symcomp(void*, void*);
static int      symerrmsg(int, char*);
static int      txtcomp(void*, void*);
static int      filecomp(void*, void*);
```

⟨*function* `syminit` 94⟩

⟨*function* `symerrmsg` 326l⟩

⟨*function* `decodename` 326m⟩

⟨*function* `cleansyms` 327⟩

⟨*function* `textseg` 328a⟩

⟨*function* `symbase` 96a⟩

⟨*function* `getsym` 328b⟩

⟨*function* `buildtbls` 329⟩

⟨*function lookup (libmach/sym.c)* 331⟩

⟨*function* `findtext` 332a⟩
⟨*function* `findglobal` 332b⟩

⟨*function* `findlocal` 332c⟩

⟨*function* `findlocvar` 333a⟩

⟨*function* `textsym` 333b⟩

⟨*function* `filesym` 333c⟩

⟨*function* `getauto` 334a⟩

⟨*function* `srchtext` 334b⟩

⟨*function* `srchdata` 335a⟩

⟨*function* `findsym` 335b⟩

⟨*function* `fnbound` 336a⟩

⟨*function* `localsym` 336b⟩

⟨*function* `globalsym` 337a⟩

⟨*function* `file2pc` 337b⟩

⟨*function* `pathcomp` 338a⟩

⟨*function* `encfname` 338b⟩

⟨*function* `hline` 339⟩

⟨*function* `hcomp` 340a⟩

⟨*function* `fileline` 340b⟩

⟨*function* `fline` 341⟩

⟨*function* `fileelem` 342a⟩

⟨*function* `symcomp` 342b⟩

⟨*function* `txtcomp` 343a⟩

⟨*function* `filecomp` 343b⟩

⟨*function* `fillsym` 343c⟩

⟨*function* `pc2sp` 344a⟩

⟨*function* `pc2line` 344b⟩

⟨*function* `line2addr` 345⟩

⟨*function* `printhist` 346a⟩

```
#ifdef DEBUG
⟨function dumphist 346b⟩
#endif
```

Uses `file` 325e, `hist` 325d, and `txtsym` 325c.

F.2.10 `libmach/access.c`

⟨*function* `geta` 349⟩≡ (354b)

```
/*
 * routines to get/put various types
 */
int
geta(Map *map, uvlong addr, uvlong *x)
{
    ulong l;
    uvlong vl;

    if (mach->szaddr == 8){
        if (get8(map, addr, &vl) < 0)
            return -1;
        *x = vl;
        return 1;
    }
```

```
    if (get4(map, addr, &l) < 0)
        return -1;
    *x = l;

    return 1;
}
```

Uses get4() 350b, get8() 350a, and mach 51a.

⟨*function* get8 350a⟩≡ (354b)

```
int
get8(Map *map, uvlong addr, uvlong *x)
{
    if (!map) {
        werrstr("get8: invalid map");
        return -1;
    }

    if (map->nsegs == 1 && map->seg[0].fd < 0) {
        *x = addr;
        return 1;
    }
    if (mget(map, addr, x, 8) < 0)
        return -1;
    *x = machdata->swav(*x);
    return 1;
}
```

Uses machdata 355e and mget() 353a.

⟨*function* get4 350b⟩≡ (354b)

```
int
get4(Map *map, uvlong addr, ulong *x)
{
    if (!map) {
        werrstr("get4: invalid map");
        return -1;
    }

    if (map->nsegs == 1 && map->seg[0].fd < 0) {
        *x = addr;
        return 1;
    }
    if (mget(map, addr, x, 4) < 0)
        return -1;
    *x = machdata->swal(*x);
    return 1;
}
```

Uses machdata 355e and mget() 353a.

⟨*function* get2 350c⟩≡ (354b)

```
int
get2(Map *map, uvlong addr, ushort *x)
{
    if (!map) {
        werrstr("get2: invalid map");
        return -1;
    }

    if (map->nsegs == 1 && map->seg[0].fd < 0) {
```

```
        *x = addr;
        return 1;
    }
    if (mget(map, addr, x, 2) < 0)
        return -1;
    *x = machdata->swab(*x);
    return 1;
}
```

Uses machdata 355e and mget() 353a.

⟨*function* get1 351a⟩≡ (354b)

```
int
get1(Map *map, uvlong addr, uchar *x, int size)
{
    uchar *cp;

    if (!map) {
        werrstr("get1: invalid map");
        return -1;
    }

    if (map->nsegs == 1 && map->seg[0].fd < 0) {
        cp = (uchar*)&addr;
        while (cp < (uchar*)(&addr+1) && size-- > 0)
            *x++ = *cp++;
        while (size-- > 0)
            *x++ = 0;
    } else
        return mget(map, addr, x, size);
    return 1;
}
```

Uses mget() 353a.

⟨*function* puta 351b⟩≡ (354b)

```
int
puta(Map *map, uvlong addr, uvlong v)
{
    if (mach->szaddr == 8)
        return put8(map, addr, v);

    return put4(map, addr, v);
}
```

Uses mach 51a, put4() 352a, and put8() 351c.

⟨*function* put8 351c⟩≡ (354b)

```
int
put8(Map *map, uvlong addr, uvlong v)
{
    if (!map) {
        werrstr("put8: invalid map");
        return -1;
    }
    v = machdata->swav(v);
    return mput(map, addr, &v, 8);
}
```

Uses machdata 355e and mput() 353b.

⟨*function put4 (libmach/access.c)* 352a⟩≡ (354b)
```
int
put4(Map *map, uvlong addr, ulong v)
{
    if (!map) {
        werrstr("put4: invalid map");
        return -1;
    }
    v = machdata->swal(v);
    return mput(map, addr, &v, 4);
}
```
Uses machdata 355e and mput() 353b.

⟨*function* put2 352b⟩≡ (354b)
```
int
put2(Map *map, uvlong addr, ushort v)
{
    if (!map) {
        werrstr("put2: invalid map");
        return -1;
    }
    v = machdata->swab(v);
    return mput(map, addr, &v, 2);
}
```
Uses machdata 355e and mput() 353b.

⟨*function* put1 352c⟩≡ (354b)
```
int
put1(Map *map, uvlong addr, uchar *v, int size)
{
    if (!map) {
        werrstr("put1: invalid map");
        return -1;
    }
    return mput(map, addr, v, size);
}
```
Uses mput() 353b.

⟨*function* spread 352d⟩≡ (354b)
```
static int
spread(struct segment *s, void *buf, int n, uvlong off)
{
    uvlong base;

    static struct {
        struct segment *s;
        char a[8192];
        uvlong off;
    } cache;

    if(s->cache){
        base = off&~(sizeof cache.a-1);
        if(cache.s != s || cache.off != base){
            cache.off = ~0;
            if(seek(s->fd, base, 0) >= 0
            && readn(s->fd, cache.a, sizeof cache.a) == sizeof cache.a){
                cache.s = s;
                cache.off = base;
            }
```

```
        }
        if(cache.s == s && cache.off == base){
            off &= sizeof cache.a-1;
            if(off+n > sizeof cache.a)
                n = sizeof cache.a - off;
            memmove(buf, cache.a+off, n);
            return n;
        }
    }

    return pread(s->fd, buf, n, off);
}
```

⟨*function* `mget` 353a⟩≡ (354b)

```
static int
mget(Map *map, uvlong addr, void *buf, int size)
{
    uvlong off;
    int i, j, k;
    struct segment *s;

    s = reloc(map, addr, (vlong*)&off);
    if (!s)
        return -1;
    if (s->fd < 0) {
        werrstr("unreadable map");
        return -1;
    }
    for (i = j = 0; i < 2; i++) {       /* in case read crosses page */
        k = spread(s, (void*)((uchar *)buf+j), size-j, off+j);
        if (k < 0) {
            werrstr("can't read address %llux: %r", addr);
            return -1;
        }
        j += k;
        if (j == size)
            return j;
    }
    werrstr("partial read at address %llux (size %d j %d)", addr, size, j);
    return -1;
}
```

Uses `reloc()` 354a and `spread()` 352d.

⟨*function* `mput` 353b⟩≡ (354b)

```
static int
mput(Map *map, uvlong addr, void *buf, int size)
{
    vlong off;
    int i, j, k;
    struct segment *s;

    s = reloc(map, addr, &off);
    if (!s)
        return -1;
    if (s->fd < 0) {
        werrstr("unwritable map");
        return -1;
    }

    seek(s->fd, off, 0);
```

```
    for (i = j = 0; i < 2; i++) {       /* in case read crosses page */
        k = write(s->fd, buf, size-j);
        if (k < 0) {
            werrstr("can't write address %llux: %r", addr);
            return -1;
        }
        j += k;
        if (j == size)
            return j;
    }
    werrstr("partial write at address %llux", addr);
    return -1;
}
```

Uses reloc() 354a.

⟨*function* reloc 354a⟩≡ (354b)

```
/*
 *      convert address to file offset; returns nonzero if ok
 */
static struct segment*
reloc(Map *map, uvlong addr, vlong *offp)
{
    int i;

    for (i = 0; i < map->nsegs; i++) {
        if (map->seg[i].inuse)
        if (map->seg[i].b <= addr && addr < map->seg[i].e) {
            *offp = addr + map->seg[i].f - map->seg[i].b;
            return &map->seg[i];
        }
    }
    werrstr("can't translate address %llux", addr);
    return 0;
}
```

⟨libmach/access.c 354b⟩≡

```
/*
 * functions to read and write an executable or file image
 */

#include <u.h>
#include <libc.h>
#include <bio.h>
#include <mach.h>

static  int     mget(Map*, uvlong, void*, int);
static  int     mput(Map*, uvlong, void*, int);
static  struct  segment*        reloc(Map*, uvlong, vlong*);
```

⟨*function* geta 349⟩

⟨*function* get8 350a⟩
⟨*function* get4 350b⟩
⟨*function* get2 350c⟩
⟨*function* get1 351a⟩

⟨*function* puta 351b⟩

⟨*function* put8 351c⟩
⟨*function put4 (libmach/access.c)* 352a⟩

⟨*function* put2 352b⟩
⟨*function* put1 352c⟩

⟨*function* spread 352d⟩

⟨*function* mget 353a⟩
⟨*function* mput 353b⟩

⟨*function* reloc 354a⟩

F.2.11 libmach/machdata.c

⟨*constant* STARTSYM 355a⟩≡ (362c)
```
#define STARTSYM        "_main"
```

⟨*constant* PROFSYM 355b⟩≡ (362c)
```
#define PROFSYM         "_mainp"
```

⟨*constant* FRAMENAME 355c⟩≡ (362c)
```
#define FRAMENAME       ".frame"
```

⟨*global* asstype 355d⟩≡ (362c)
```
int     asstype = AARM;         /* disassembler type */
```
Uses asstype 355d.

⟨*global* machdata 355e⟩≡ (362c)
```
Machdata *machdata;             /* machine-dependent functions */
```

⟨*function* localaddr 355f⟩≡ (362c)
```
int
localaddr(Map *map, char *fn, char *var, uvlong *r, Rgetter rget)
{
    Symbol s;
    uvlong fp, pc, sp, link;

    if (!lookup(fn, 0, &s)) {
        werrstr("function not found");
        return -1;
    }
    pc = rget(map, mach->pc);
    sp = rget(map, mach->sp);
    if(mach->link)
        link = rget(map, mach->link);
    else
        link = 0;
    fp = machdata->findframe(map, s.value, pc, sp, link);
    if (fp == 0) {
        werrstr("stack frame not found");
        return -1;
    }

    if (!var || !var[0]) {
        *r = fp;
        return 1;
    }

    if (findlocal(&s, var, &s) == 0) {
        werrstr("local variable not found");
        return -1;
```

```
    }

    switch (s.class) {
    case CAUTO:
        *r = fp - s.value;
        break;
    case CPARAM:                /* assume address size is stack width */
        *r = fp + s.value + mach->szaddr;
        break;
    default:
        werrstr("local variable not found: %d", s.class);
        return -1;
    }
    return 1;
}
```

Uses `findlocal()` 332c, `lookup()` 331, `mach` 51a, and `machdata` 355e.

⟨*function* `symoff` 356a⟩≡ (362c)

```
/*
 * Print value v as s.name[+offset] if possible, or just v.
 */
int
symoff(char *buf, int n, uvlong v, int space)
{
    Symbol s;
    int r;
    long delta;

    r = delta = 0;              /* to shut compiler up */
    if (v) {
        r = findsym(v, space, &s);
        if (r)
            delta = v-s.value;
        if (delta < 0)
            delta = -delta;
    }
    if (v == 0 || r == 0)
        return snprint(buf, n, "%llux", v);
    if (s.type != 't' && s.type != 'T' && delta >= 4096)
        return snprint(buf, n, "%llux", v);
    else if (delta)
        return snprint(buf, n, "%s+%lux", s.name, delta);
    else
        return snprint(buf, n, "%s", s.name);
}
```

Uses `findsym()` 335b.

⟨*function* `fpformat` 356b⟩≡ (362c)

```
/*
 *      Format floating point registers
 *
 *      Register codes in format field:
 *      'X' - print as 32-bit hexadecimal value
 *      'F' - 64-bit double register when modif == 'F'; else 32-bit single reg
 *      'f' - 32-bit ieee float
 *      '8' - big endian 80-bit ieee extended float
 *      '3' - little endian 80-bit ieee extended float with hole in bytes 8&9
 */
int
fpformat(Map *map, Reglist *rp, char *buf, int n, int modif)
```

```
{
    char reg[12];
    ulong r;

    switch(rp->rformat)
    {
    case 'X':
        if (get4(map, rp->roffs, &r) < 0)
            return -1;
        snprint(buf, n, "%lux", r);
        break;
    case 'F':   /* first reg of double reg pair */
        if (modif == 'F')
        if ((rp->rformat=='F') || (((rp+1)->rflags&RFLT) && (rp+1)->rformat == 'f')) {
            if (get1(map, rp->roffs, (uchar *)reg, 8) < 0)
                return -1;
            machdata->dftos(buf, n, reg);
            if (rp->rformat == 'F')
                return 1;
            return 2;
        }
            /* treat it like 'f' */
        if (get1(map, rp->roffs, (uchar *)reg, 4) < 0)
            return -1;
        machdata->sftos(buf, n, reg);
        break;
    case 'f':   /* 32 bit float */
        if (get1(map, rp->roffs, (uchar *)reg, 4) < 0)
            return -1;
        machdata->sftos(buf, n, reg);
        break;
    case '3':   /* little endian ieee 80 with hole in bytes 8&9 */
        if (get1(map, rp->roffs, (uchar *)reg, 10) < 0)
            return -1;
        memmove(reg+10, reg+8, 2);      /* open hole */
        memset(reg+8, 0, 2);            /* fill it */
        leieee80ftos(buf, n, reg);
        break;
    case '8':   /* big-endian ieee 80 */
        if (get1(map, rp->roffs, (uchar *)reg, 10) < 0)
            return -1;
        beieee80ftos(buf, n, reg);
        break;
    default:    /* unknown */
        break;
    }
    return 1;
}
```

Uses `beieee80ftos()` 359e, `get1()` 351a, `get4()` 350b, `leieee80ftos()` 360a, and `machdata` 355e.

⟨*function* `ieeedftos` 357⟩≡ (362c)

```
/*
 * These routines assume that if the number is representable
 * in IEEE floating point, it will be representable in the native
 * double format.  Naive but workable, probably.
 */
int
ieeedftos(char *buf, int n, ulong h, ulong l)
{
    double fr;
```

```
    int exp;

    if (n <= 0)
        return 0;


    if(h & (1L<<31)){
        *buf++ = '-';
        h &= ~(1L<<31);
    }else
        *buf++ = ' ';
    n--;
    if(l == 0 && h == 0)
        return snprint(buf, n, "0.");
    exp = (h>>20) & ((1L<<11)-1L);
    if(exp == 0)
        return snprint(buf, n, "DeN(%.8lux%.8lux)", h, l);
    if(exp == ((1L<<11)-1L)){
        if(l==0 && (h&((1L<<20)-1L)) == 0)
            return snprint(buf, n, "Inf");
        else
            return snprint(buf, n, "NaN(%.8lux%.8lux)", h&((1L<<20)-1L), l);
    }
    exp -= (1L<<10) - 2L;
    fr = l & ((1L<<16)-1L);
    fr /= 1L<<16;
    fr += (l>>16) & ((1L<<16)-1L);
    fr /= 1L<<16;
    fr += (h & (1L<<20)-1L) | (1L<<20);
    fr /= 1L<<21;
    fr = ldexp(fr, exp);
    return snprint(buf, n, "%.18g", fr);
}
```

⟨*function* ieeesftos 358⟩≡ (362c)

```
int
ieeesftos(char *buf, int n, ulong h)
{
    double fr;
    int exp;

    if (n <= 0)
        return 0;

    if(h & (1L<<31)){
        *buf++ = '-';
        h &= ~(1L<<31);
    }else
        *buf++ = ' ';
    n--;
    if(h == 0)
        return snprint(buf, n, "0.");
    exp = (h>>23) & ((1L<<8)-1L);
    if(exp == 0)
        return snprint(buf, n, "DeN(%.8lux)", h);
    if(exp == ((1L<<8)-1L)){
        if((h&((1L<<23)-1L)) == 0)
            return snprint(buf, n, "Inf");
        else
            return snprint(buf, n, "NaN(%.8lux)", h&((1L<<23)-1L));
```

```
    }
    exp -= (1L<<7) - 2L;
    fr = (h & ((1L<<23)-1L)) | (1L<<23);
    fr /= 1L<<24;
    fr = ldexp(fr, exp);
    return snprint(buf, n, "%.9g", fr);
}
```

⟨*function* beieeesftos 359a⟩≡ (362c)

```
int
beieeesftos(char *buf, int n, void *s)
{
    return ieeesftos(buf, n, beswal(*(ulong*)s));
}
```

Uses beswal() 315b and ieeesftos() 358.

⟨*function* beieeedftos 359b⟩≡ (362c)

```
int
beieeedftos(char *buf, int n, void *s)
{
    return ieeedftos(buf, n, beswal(*(ulong*)s), beswal(((ulong*)(s))[1]));
}
```

Uses beswal() 315b and ieeedftos() 357.

⟨*function* leieeesftos 359c⟩≡ (362c)

```
int
leieeesftos(char *buf, int n, void *s)
{
    return ieeesftos(buf, n, leswal(*(ulong*)s));
}
```

Uses ieeesftos() 358 and leswal() 316b.

⟨*function* leieeedftos 359d⟩≡ (362c)

```
int
leieeedftos(char *buf, int n, void *s)
{
    return ieeedftos(buf, n, leswal(((ulong*)(s))[1]), leswal(*(ulong*)s));
}
```

Uses ieeedftos() 357 and leswal() 316b.

⟨*function* beieee80ftos 359e⟩≡ (362c)

```
/* packed in 12 bytes, with s[2]==s[3]==0; mantissa starts at s[4]*/
int
beieee80ftos(char *buf, int n, void *s)
{
    uchar *reg = (uchar*)s;
    int i;
    ulong x;
    uchar ieee[8+8];    /* room for slop */
    uchar *p, *q;

    memset(ieee, 0, sizeof(ieee));
    /* sign */
    if(reg[0] & 0x80)
        ieee[0] |= 0x80;

    /* exponent */
    x = ((reg[0]&0x7F)<<8) | reg[1];
    if(x == 0)          /* number is + or - 0 */
```

```
        goto done;
    if(x == 0x7FFF){
        if(memcmp(reg+4, ieee+1, 8) == 0){ /* infinity */
            x = 2047;
        }else{                          /* NaN */
            x = 2047;
            ieee[7] = 0x1;              /* make sure */
        }
        ieee[0] |= x>>4;
        ieee[1] |= (x&0xF)<<4;
        goto done;
    }
    x -= 0x3FFF;                /* exponent bias */
    x += 1023;
    if(x >= (1<<11) || ((reg[4]&0x80)==0 && x!=0))
        return snprint(buf, n, "not in range");
    ieee[0] |= x>>4;
    ieee[1] |= (x&0xF)<<4;

    /* mantissa */
    p = reg+4;
    q = ieee+1;
    for(i=0; i<56; i+=8, p++, q++){     /* move one byte */
        x = (p[0]&0x7F) << 1;
        if(p[1] & 0x80)
            x |= 1;
        q[0] |= x>>4;
        q[1] |= (x&0xF)<<4;
    }
    done:
    return beieeedftos(buf, n, (void*)ieee);
}
```

Uses beieeedftos() 359b.

⟨*function* leieee80ftos 360a⟩≡ (362c)

```
int
leieee80ftos(char *buf, int n, void *s)
{
    int i;
    char *cp;
    char b[12];

    cp = (char*) s;
    for(i=0; i<12; i++)
        b[11-i] = *cp++;
    return beieee80ftos(buf, n, b);
}
```

Uses beieee80ftos() 359e.

⟨*function* cisctrace 360b⟩≡ (362c)

```
int
cisctrace(Map *map, uvlong pc, uvlong sp, uvlong link, Tracer trace)
{
    Symbol s;
    int found, i;
    uvlong opc, moved;

    USED(link);
    i = 0;
    opc = 0;
```

```
    while(pc && opc != pc) {
        moved = pc2sp(pc);
        if (moved == ~0)
            break;
        found = findsym(pc, CTEXT, &s);
        if (!found)
            break;
        if(strcmp(STARTSYM, s.name) == 0 || strcmp(PROFSYM, s.name) == 0)
            break;

        sp += moved;
        opc = pc;
        if (geta(map, sp, &pc) < 0)
            break;
        (*trace)(map, pc, sp, &s);
        sp += mach->szaddr;     /*assumes address size = stack width*/
        if(++i > 40)
            break;
    }
    return i;
}
```

Uses PROFSYM-82 355b, STARTSYM-81 355a, findsym() 335b, geta() 349, mach 51a, and pc2sp() 344a.

⟨*function* risctrace 361⟩≡ (362c)

```
int
risctrace(Map *map, uvlong pc, uvlong sp, uvlong link, Tracer trace)
{
    int i;
    Symbol s, f;
    uvlong oldpc;

    i = 0;
    while(findsym(pc, CTEXT, &s)) {
        if(strcmp(STARTSYM, s.name) == 0 || strcmp(PROFSYM, s.name) == 0)
            break;

        if(pc == s.value)       /* at first instruction */
            f.value = 0;
        else if(findlocal(&s, FRAMENAME, &f) == 0)
            break;

        oldpc = pc;
        if(s.type == 'L' || s.type == 'l' || pc <= s.value+mach->pcquant)
            pc = link;
        else
            if (geta(map, sp, &pc) < 0)
                break;

        if(pc == 0 || (pc == oldpc && f.value == 0))
            break;

        sp += f.value;
        (*trace)(map, pc-8, sp, &s);

        if(++i > 40)
            break;
    }
    return i;
}
```

Uses FRAMENAME-83 355c, PROFSYM-82 355b, STARTSYM-81 355a, findlocal() 332c, findsym() 335b, geta() 349, and mach 51a.

⟨*function* `ciscframe` 362a⟩≡ (362c)

```
uvlong
ciscframe(Map *map, uvlong addr, uvlong pc, uvlong sp, uvlong link)
{
    Symbol s;
    uvlong moved;

    USED(link);
    for(;;) {
        moved = pc2sp(pc);
        if (moved  == ~0)
            break;
        sp += moved;
        findsym(pc, CTEXT, &s);
        if (addr == s.value)
            return sp;
        if (geta(map, sp, &pc) < 0)
            break;
        sp += mach->szaddr;     /*assumes sizeof(addr) = stack width*/
    }
    return 0;
}
```

Uses `findsym()` 335b, `geta()` 349, `mach` 51a, and `pc2sp()` 344a.

⟨*function* `riscframe` 362b⟩≡ (362c)

```
uvlong
riscframe(Map *map, uvlong addr, uvlong pc, uvlong sp, uvlong link)
{
    Symbol s, f;

    while (findsym(pc, CTEXT, &s)) {
        if(strcmp(STARTSYM, s.name) == 0 || strcmp(PROFSYM, s.name) == 0)
            break;

        if(pc == s.value)       /* at first instruction */
            f.value = 0;
        else
        if(findlocal(&s, FRAMENAME, &f) == 0)
            break;

        sp += f.value;
        if (s.value == addr)
            return sp;

        if (s.type == 'L' || s.type == 'l' || pc-s.value <= mach->szaddr*2)
            pc = link;
        else
        if (geta(map, sp-f.value, &pc) < 0)
            break;
    }
    return 0;
}
```

Uses `FRAMENAME-83` 355c, `PROFSYM-82` 355b, `STARTSYM-81` 355a, `findlocal()` 332c, `findsym()` 335b, `geta()` 349, and `mach` 51a.

⟨`libmach/machdata.c` 362c⟩≡

```
/*
 * Debugger utilities shared by at least two architectures
 */

#include <u.h>
```

```
#include <libc.h>
#include <bio.h>
#include <mach.h>

⟨constant STARTSYM 355a⟩
⟨constant PROFSYM 355b⟩
⟨constant FRAMENAME 355c⟩

extern  Machdata        mipsmach;

⟨global asstype 355d⟩
⟨global machdata 355e⟩

⟨function localaddr 355f⟩

⟨function symoff 356a⟩

⟨function fpformat 356b⟩

⟨function _hexify 212a⟩

⟨function ieeedftos 357⟩

⟨function ieeesftos 358⟩

⟨function beieeesftos 359a⟩

⟨function beieeedftos 359b⟩

⟨function leieeesftos 359c⟩

⟨function leieeedftos 359d⟩

⟨function beieee80ftos 359e⟩

⟨function leieee80ftos 360a⟩

⟨function cisctrace 360b⟩

⟨function risctrace 361⟩

⟨function ciscframe 362a⟩

⟨function riscframe 362b⟩
```

F.2.12 libmach/obj.c

```
⟨function islocal 363a⟩≡                                        (369)
  #define islocal(t)      ((t)=='a' || (t)=='p')

⟨enum _anon_ (libmach/obj.c) 363b⟩≡                             (369)
  enum
  {
      NNAMES      = 50,
      MAXIS       = 8,            /* max length to determine if a file is a .? file */
      MAXOFF      = 0x7fffffff,   /* larger than any possible local offset */
      NHASH       = 1024,         /* must be power of two */
      HASHMUL     = 79L,
  };
```

⟨*struct* `Obj` 364a⟩≡ (369)
```
struct  Obj             /* functions to handle each intermediate (.$O) file */
{
    char        *name;                          /* name of each $O file */
    int (*is)(char*);                   /* test for each type of $O file */
    int (*read)(Biobuf*, Prog*);        /* read for each type of $O file*/
};
```

⟨*global* `obj` 364b⟩≡ (369)
```
static Obj      obj[] =
{                       /* functions to identify and parse each type of obj */
    [ObjArm]    "arm .5",       _is5, _read5,
    [Obj386]    "386 .8",       _is8, _read8,
    [ObjMips]  "mips .v",      _isv, _readv,
    [Maxobjtype]        0, 0
};
```
Uses `_is5()` 311b, `_is8()`, `_isv()`, `_read5()` 312a, `_read8()`, and `_readv()`.

⟨*struct* `Symtab` 364c⟩≡ (369)
```
struct  Symtab
{
    struct      Sym     s;
    struct      Symtab  *next;
};
```

⟨*global hash (libmach/obj.c)* 364d⟩≡ (369)
```
static  Symtab *hash[NHASH];
```
Uses `NHASH-180` 363b.

⟨*global* `names` 364e⟩≡ (369)
```
static  Sym     *names[NNAMES]; /* working set of active names */
```
Uses `NNAMES-177` 363b.

⟨*function* `objtype` 364f⟩≡ (369)
```
int
objtype(Biobuf *bp, char **name)
{
    int i;
    char buf[MAXIS];

    if(Bread(bp, buf, MAXIS) < MAXIS)
        return -1;
    Bseek(bp, -MAXIS, 1);
    for (i = 0; i < Maxobjtype; i++) {
        if (obj[i].is && (*obj[i].is)(buf)) {
            if (name)
                *name = obj[i].name;
            return i;
        }
    }
    return -1;
}
```
Uses `MAXIS-178` 363b and `obj` 364b.

⟨*function* isar 365a⟩≡ (369)

```
int
isar(Biobuf *bp)
{
    int n;
    char magbuf[SARMAG];

    n = Bread(bp, magbuf, SARMAG);
    if(n == SARMAG && strncmp(magbuf, ARMAG, SARMAG) == 0)
        return 1;
    return 0;
}
```

⟨*function* readobj 365b⟩≡ (369)

```
/*
 * determine what kind of object file this is and process it.
 * return whether or not this was a recognized intermediate file.
 */
int
readobj(Biobuf *bp, int objtype)
{
    Prog p;

    if (objtype < 0 || objtype >= Maxobjtype || obj[objtype].is == 0)
        return 1;
    objreset();
    while ((*obj[objtype].read)(bp, &p))
        if (!processprog(&p, 1))
            return 0;
    return 1;
}
```

Uses obj 364b, objreset() 368c, and processprog() 365d.

⟨*function* readar 365c⟩≡ (369)

```
int
readar(Biobuf *bp, int objtype, vlong end, int doautos)
{
    Prog p;

    if (objtype < 0 || objtype >= Maxobjtype || obj[objtype].is == 0)
        return 1;
    objreset();
    while ((*obj[objtype].read)(bp, &p) && Boffset(bp) < end)
        if (!processprog(&p, doautos))
            return 0;
    return 1;
}
```

Uses obj 364b, objreset() 368c, and processprog() 365d.

⟨*function* processprog 365d⟩≡ (369)

```
/*
 *      decode a symbol reference or definition
 */
static  int
processprog(Prog *p, int doautos)
{
    if(p->kind == aNone)
        return 1;
    if(p->sym < 0 || p->sym >= NNAMES)
        return 0;
```

```
    switch(p->kind)
    {
    case aName:
        if (!doautos)
        if(p->type != 'U' && p->type != 'b')
            break;
        objlookup(p->sym, p->id, p->type, p->sig);
        break;
    case aText:
        objupdate(p->sym, 'T');
        break;
    case aData:
        objupdate(p->sym, 'D');
        break;
    default:
        break;
    }
    return 1;
}
```

Uses NNAMES-177 363b, aData 314e, aName 314e, aNone 314e, aText 314e, objlookup() 366, and objupdate() 368a.

⟨*function* objlookup 366⟩≡ (369)

```
/*
 * find the entry for s in the symbol array.
 * make a new entry if it is not already there.
 */
static void
objlookup(int id, char *name, int type, uint sig)
{
    long h;
    char *cp;
    Sym *s;
    Symtab *sp;

    s = names[id];
    if(s && strcmp(s->name, name) == 0) {
        s->type = type;
        s->sig = sig;
        return;
    }

    h = *name;
    for(cp = name+1; *cp; h += *cp++)
        h *= HASHMUL;
    if(h < 0)
        h = ~h;
    h &= (NHASH-1);
    if (type == 'U' || type == 'b' || islocal(type)) {
        for(sp = hash[h]; sp; sp = sp->next)
            if(strcmp(sp->s.name, name) == 0) {
                switch(sp->s.type) {
                case 'T':
                case 'D':
                case 'U':
                    if (type == 'U') {
                        names[id] = &sp->s;
                        return;
                    }
                    break;
                case 't':
```

```
                case 'd':
                case 'b':
                    if (type == 'b') {
                        names[id] = &sp->s;
                        return;
                    }
                    break;
                case 'a':
                case 'p':
                    if (islocal(type)) {
                        names[id] = &sp->s;
                        return;
                    }
                    break;
                default:
                    break;
                }
            }
        }
        sp = malloc(sizeof(Symtab));
        sp->s.name = name;
        sp->s.type = type;
        sp->s.sig = sig;
        sp->s.value = islocal(type) ? MAXOFF : 0;
        names[id] = &sp->s;
        sp->next = hash[h];
        hash[h] = sp;
        return;
    }
```

Uses HASHMUL-181 363b, MAXOFF-179 363b, NHASH-180 363b, islocal-176 363a, and names 364e.

⟨*function* objtraverse 367a⟩≡ (369)

```
    /*
     *      traverse the symbol lists
     */
    void
    objtraverse(void (*fn)(Sym*, void*), void *pointer)
    {
        int i;
        Symtab *s;

        for(i = 0; i < NHASH; i++)
            for(s = hash[i]; s; s = s->next)
                (*fn)(&s->s, pointer);
    }
```

Uses NHASH-180 363b.

⟨*function* _offset 367b⟩≡ (369)

```
    /*
     * update the offset information for a 'a' or 'p' symbol in an intermediate file
     */
    void
    _offset(int id, vlong off)
    {
        Sym *s;

        s = names[id];
        if (s && s->name[0] && islocal(s->type) && s->value > off)
            s->value = off;
    }
```

Uses islocal-176 363a and names 364e.

⟨*function* objupdate 368a⟩≡ (369)

```
/*
 * update the type of a global text or data symbol
 */
static void
objupdate(int id, int type)
{
    Sym *s;

    s = names[id];
    if (s && s->name[0])
        if (s->type == 'U')
            s->type = type;
        else if (s->type == 'b')
            s->type = tolower(type);
}
```

Uses names 364e.

⟨*function* nextar 368b⟩≡ (369)

```
/*
 * look for the next file in an archive
 */
int
nextar(Biobuf *bp, int offset, char *buf)
{
    struct ar_hdr a;
    int i, r;
    long arsize;

    if (offset&01)
        offset++;
    Bseek(bp, offset, 0);
    r = Bread(bp, &a, SAR_HDR);
    if(r != SAR_HDR)
        return 0;
    if(strncmp(a.fmag, ARFMAG, sizeof(a.fmag)))
        return -1;
    for(i=0; i<sizeof(a.name) && i<SARNAME && a.name[i] != ' '; i++)
        buf[i] = a.name[i];
    buf[i] = 0;
    arsize = strtol(a.size, 0, 0);
    if (arsize&1)
        arsize++;
    return arsize + SAR_HDR;
}
```

⟨*function* objreset 368c⟩≡ (369)

```
static void
objreset(void)
{
    int i;
    Symtab *s, *n;

    for(i = 0; i < NHASH; i++) {
        for(s = hash[i]; s; s = n) {
            n = s->next;
            free(s->s.name);
            free(s);
```

```
        }
        hash[i] = 0;
    }
    memset(names, 0, sizeof names);
}
```

Uses NHASH-180 363b and names 364e.

⟨libmach/obj.c 369⟩≡

```
/*
 * obj.c
 * routines universal to all object files
 */
#include <u.h>
#include <libc.h>
#include <bio.h>
#include <ar.h>
#include <mach.h>

#include "obj.h"
```

⟨*function* islocal 363a⟩

⟨*enum* _anon_ (libmach/obj.c) 363b⟩

```
int                     /* in [$OS].c */ //$
  _is5(char*),
  _is8(char*),
  _isv(char*),
  _read5(Biobuf*, Prog*),
  _read8(Biobuf*, Prog*);
  _readv(Biobuf*, Prog*);


typedef struct Obj      Obj;
typedef struct Symtab   Symtab;
```

⟨*struct* Obj 364a⟩

⟨*global* obj 364b⟩

⟨*struct* Symtab 364c⟩

⟨*global hash (libmach/obj.c)* 364d⟩
⟨*global* names 364e⟩

```
static  int     processprog(Prog*,int); /* decode each symbol reference */
static  void    objreset(void);
static  void    objlookup(int, char *, int, uint);
static  void    objupdate(int, int);
```

⟨*function* objtype 364f⟩

⟨*function* isar 365a⟩

⟨*function* readobj 365b⟩

⟨*function* readar 365c⟩

⟨*function* processprog 365d⟩

⟨*function* `objlookup` 366⟩
⟨*function* `objtraverse` 367a⟩

⟨*function* `_offset` 367b⟩

⟨*function* `objupdate` 368a⟩

⟨*function* `nextar` 368b⟩

⟨*function* `objreset` 368c⟩
Uses `Obj` 369 and `Symtab` 369.

F.2.13 libmach/setmach.c

⟨*struct* `machtab` 370a⟩≡ (371a)
```
struct machtab
{
    char        *name;                  /* machine name */
    short       type;                   /* executable type */
    short       boottype;               /* bootable type */
    int         asstype;                /* disassembler code */
    Mach        *mach;                  /* machine description */
    Machdata    *machdata;              /* machine functions */
};
```

⟨*global* `machines` 370b⟩≡ (371a)
```
/*
 *      machine selection table.  machines with native disassemblers should
 *      follow the plan 9 variant in the table; native modes are selectable
 *      only by name.
 */
Machtab machines[] =
{
    {   "386",                          /*plan 9 386*/
        FI386,
        FI386B,
        AI386,
        &mi386,
        &i386mach,      },
    {   "arm",                          /*ARM*/
        FARM,
        FARMB,
        AARM,
        &marm,
        &armmach,       },
    {   "mips",                         /*plan 9 mips*/
 FMIPS,
 FMIPSB,
 AMIPS,
 &mmips,
 &mipsmach,     },
    {   0               },              /*the terminator*/
};
```
Uses `armmach` 52a, `i386mach`, `marm` 308g, `mi386`, `mipsmach`, and `mmips`.

⟨*function* `machbyname` 370c⟩≡ (371a)
```
/*
 *      select a machine by name
```

```
 */
int
machbyname(char *name)
{
    Machtab *mp;

    // choose a default if no name given
    if (!name) {
        asstype = AARM;
        machdata = &armmach;
        mach = &marm;
        return 1;
    }
    for (mp = machines; mp->name; mp++){
        if (strcmp(mp->name, name) == 0) {
            asstype = mp->asstype;
            machdata = mp->machdata;
            mach = mp->mach;
            return 1;
        }
    }
    return 0;
}
```

Uses armmach 52a, asstype 355d, mach 51a, machdata 355e, machines 370b, and marm 308g.

⟨libmach/setmach.c 371a⟩≡

```
#include        <u.h>
#include        <libc.h>
#include        <bio.h>
#include        <mach.h>
        /* table for selecting machine-dependent parameters */

typedef struct machtab Machtab;

⟨struct machtab 370a⟩

extern  Mach            mi386, marm, mmips;
extern  Machdata        i386mach, armmach, mipsmach;

⟨global machines 370b⟩

⟨function machbytype 91⟩
⟨function machbyname 370c⟩
```

Uses machtab 370a.

F.3 tracers/

F.3.1 tracers/ratrace.c

⟨tracers/ratrace.c 371b⟩≡

```
// System calls tracer

#include <u.h>
#include <libc.h>

#include <bio.h>
#include <thread.h>
```

```
// was called ratrace, maybe in hommage to Ed Wood "Rat Race" movie

⟨enum _anon_ (ratrace) 38a⟩

⟨global out 38c⟩
⟨global quit 38e⟩
⟨global forkc 38d⟩
⟨global nread 38b⟩

typedef struct Str Str;
⟨struct Str 41a⟩

⟨function die 45a⟩

⟨function cwrite 44c⟩

⟨function newstr 45b⟩

⟨function reader 43⟩
⟨function writer 41b⟩

⟨function usage (tracers/ratrace.c) 17⟩

⟨function hang 40⟩

⟨function threadmain 37⟩
```

Uses Str 41a.

F.3.2 tracers/ktrace.c

⟨global fhdr 372a⟩≡ (377b)

```
static  Fhdr    fhdr;
```

⟨global interactive 372b⟩≡ (377b)

```
static  int     interactive = 0;
```

Uses interactive-6 372b.

⟨constant FRAMENAME (ktrace) 372c⟩≡ (377b)

```
#define FRAMENAME       ".frame"
```

⟨function usage 372d⟩≡ (377b)

```
static void
usage(void)
{
    fprint(2, "usage: ktrace [-i] kernel pc sp [link]\n");
    exits("usage");
}
```

⟨function printaddr 372e⟩≡ (377b)

```
static void
printaddr(char *addr, uvlong pc)
{
    int i;
    char *p;

    /*
     * reformat the following.
     *
     * foo+1a1 -> src(foo+0x1a1);
```

```
     * 10101010 -> src(0x10101010);
     */

    if(strlen(addr) == 8 && strchr(addr, '+') == nil){
        for(i=0; i<8; i++)
            if(!isxdigit(addr[i]))
                break;
        if(i == 8){
            print("src(%#.8llux); // 0x%s\n", pc, addr);
            return;
        }
    }

    if(p=strchr(addr, '+')){
        *p++ = 0;
        print("src(%#.8llux); // %s+0x%s\n", pc, addr, p);
    }else
        print("src(%#.8llux); // %s\n", pc, addr);
}
```

⟨*global* `fmt` 373a⟩≡ (377b)

```
static void (*fmt)(char*, uvlong) = printaddr;
```

Uses `fmt-8` 373a and `printaddr()` 372e.

⟨*function* `main`*(ktrace.c)* 373b⟩≡ (377b)

```
void
main(int argc, char *argv[])
{
    int (*t)(uvlong, uvlong, uvlong);
    uvlong pc, sp, link;
    int fd;

    ARGBEGIN{
    case 'i':
        interactive = 1;
        break;
    default:
        usage();
    }ARGEND

    link = 0;
    t = rtrace;
    switch(argc){
    case 4:
        t = rtrace;
        link = strtoull(argv[3], 0, 16);
        break;
    case 3:
        break;
    default:
        usage();
    }
    pc = strtoull(argv[1], 0, 16);
    sp = strtoull(argv[2], 0, 16);
    if(!interactive)
        readstack();

    fd = open(argv[0], OREAD);
    if(fd < 0)
        fatal("can't open %s: %r", argv[0]);
```

```
    inithdr(fd);
    switch(fhdr.magic){
    case I_MAGIC:       /* intel 386 */
        t = i386trace;
        break;
    case E_MAGIC:       /* arm 7-something */
        t = rtrace;
        break;
    default:
        fprint(2, "%s: warning: can't tell what type of stack %s uses; assuming it's %s\n",
            argv0, argv[0], argc == 4 ? "risc" : "cisc");
        break;
    }
    (*t)(pc, sp, link);
    exits(0);
}
```

⟨*function* `inithdr` 374a⟩≡ (377b)

```
static void
inithdr(int fd)
{
    seek(fd, 0, 0);
    if(!crackhdr(fd, &fhdr))
        fatal("read text header");

    if(syminit(fd, &fhdr) < 0)
        fatal("%r\n");
}
```

Uses `crackhdr()` 90, `fhdr-5` 372a, and `syminit()` 94.

⟨*function* `rtrace` 374b⟩≡ (377b)

```
// for MIPS and ARM
static int
rtrace(uvlong pc, uvlong sp, uvlong link)
{
    Symbol s, f;
    char buf[128];
    uvlong oldpc;
    int i;

    i = 0;
    while(findsym(pc, CTEXT, &s)) {
        if(pc == s.value)       /* at first instruction */
            f.value = 0;
        else if(findlocal(&s, FRAMENAME, &f) == 0)
            break;

        symoff(buf, sizeof buf, pc, CANY);
        fmt(buf, pc);

        oldpc = pc;
        if(s.type == 'L' || s.type == 'l' || pc <= s.value+mach->pcquant){
            if(link == 0)
                fprint(2, "%s: need to supply a valid link register\n", argv0);
            pc = link;
        }else{
            pc = getval(sp);
            if(pc == 0)
                break;
        }
```

```
        if(pc == 0 || (pc == oldpc && f.value == 0))
            break;

        sp += f.value;

        if(++i > 40)
            break;
    }
    return i;
}
```

Uses FRAMENAME-7 372c, findlocal() 332c, findsym() 335b, fmt-8 373a, getval() 376f, mach 51a, and symoff() 356a.

⟨*function i386trace (ktrace)* 375⟩≡ (377b)

```
static int
i386trace(uvlong pc, uvlong sp, uvlong link)
{
    int i;
    uvlong osp;
    Symbol s, f;
    char buf[128];

    USED(link);
    i = 0;
    osp = 0;
    while(findsym(pc, CTEXT, &s)) {

        symoff(buf, sizeof buf, pc, CANY);
        fmt(buf, pc);

        if(pc != s.value) {     /* not at first instruction */
            if(findlocal(&s, FRAMENAME, &f) == 0)
                break;
            sp += f.value-mach->szaddr;
        }else if(strcmp(s.name, "forkret") == 0){
            print("//passing interrupt frame; last pc found at sp=%#llux\n", osp);
            sp +=  15 * mach->szaddr;           /* pop interrupt frame */
        }

        pc = getval(sp);
        if(pc == 0 && strcmp(s.name, "forkret") == 0){
            sp += 3 * mach->szaddr;                     /* pop iret eip, cs, eflags */
            print("//guessing call through invalid pointer, try again at sp=%#llux\n", sp);
            s.name = "";
            pc = getval(sp);
        }
        if(pc == 0) {
            print("//didn't find pc at sp=%#llux, last pc found at sp=%#llux\n", sp, osp);
            break;
        }
        osp = sp;

        sp += mach->szaddr;
        if(strcmp(s.name, "forkret") == 0)
            sp += 2 * mach->szaddr;                     /* pop iret cs, eflags */

        if(++i > 40)
            break;
    }
    return i;
```

```
}
```
Uses FRAMENAME-7 372c, findlocal() 332c, findsym() 335b, fmt-8 373a, getval() 376f, mach 51a, and symoff() 356a.

⟨*global* naddr 376a⟩≡ (377b)
```
int naddr;
```

⟨*global* addr 376b⟩≡ (377b)
```
uvlong addr[1024];
```

⟨*global* val 376c⟩≡ (377b)
```
uvlong val[1024];
```

⟨*function* putval 376d⟩≡ (377b)
```
static void
putval(uvlong a, uvlong v)
{
    if(naddr < nelem(addr)){
        addr[naddr] = a;
        val[naddr] = v;
        naddr++;
    }
}
```
Uses naddr 376a and val 376c.

⟨*function* readstack 376e⟩≡ (377b)
```
static void
readstack(void)
{
    Biobuf b;
    char *p;
    char *f[64];
    int nf, i;

    Binit(&b, 0, OREAD);
    while(p=Brdline(&b, '\n')){
        p[Blinelen(&b)-1] = 0;
        nf = tokenize(p, f, nelem(f));
        for(i=0; i<nf; i++){
            if(p=strchr(f[i], '=')){
                *p++ = 0;
                putval(strtoull(f[i], 0, 16), strtoull(p, 0, 16));
            }
        }
    }
}
```
Uses putval() 376d.

⟨*function* getval 376f⟩≡ (377b)
```
static uvlong
getval(uvlong a)
{
    char buf[256];
    int i, n;
    uvlong r;

    if(interactive){
        print("// data at %#8.8llux? ", a);
        n = read(0, buf, sizeof(buf)-1);
        if(n <= 0)
```

```
            return 0;
        buf[n] = '\0';
        r = strtoull(buf, 0, 16);
    }else{
        r = 0;
        for(i=0; i<naddr; i++)
            if(addr[i] == a)
                r = val[i];
    }

    return r;
}
```

Uses interactive-6 372b, naddr 376a, and val 376c.

⟨*function* fatal 377a⟩≡ (377b)

```
static void
fatal(char *fmt, ...)
{
    char buf[4096];
    va_list arg;

    va_start(arg, fmt);
    vseprint(buf, buf+sizeof(buf), fmt, arg);
    va_end(arg);
    fprint(2, "ktrace: %s\n", buf);
    exits(buf);
}
```

⟨tracers/ktrace.c 377b⟩≡

```
#include <u.h>
#include <libc.h>
#include <bio.h>
#include <ctype.h>

#include <mach.h>

// ktrace - interpret kernel stack dumps

static  int     rtrace(uvlong, uvlong, uvlong);
static  int     i386trace(uvlong, uvlong, uvlong);

static  uvlong  getval(uvlong);
static  void    inithdr(int);
static  void    fatal(char*, ...);
static  void    readstack(void);

⟨global fhdr 372a⟩
⟨global interactive 372b⟩

⟨constant FRAMENAME (ktrace) 372c⟩

⟨function usage 372d⟩

⟨function printaddr 372e⟩

⟨global fmt 373a⟩

⟨function main(ktrace.c) 373b⟩

⟨function inithdr 374a⟩
```

⟨*function* `rtrace` 374b⟩

⟨*function i386trace (ktrace)* 375⟩

⟨*global* `naddr` 376a⟩
⟨*global* `addr` 376b⟩
⟨*global* `val` 376c⟩

⟨*function* `putval` 376d⟩

⟨*function* `readstack` 376e⟩

⟨*function* `getval` 376f⟩

⟨*function* `fatal` 377a⟩

F.4 acid/

F.4.1 acid/acid.h

⟨*enum* `_anon_ (acid/acid.h)` 378a⟩≡ (378b)

```
enum
{
    ⟨acid constants 56c⟩
};
```

⟨`acid/acid.h` 378b⟩≡

```
/* acid.h */
⟨enum _anon_ (acid/acid.h) 378a⟩

#pragma varargck type "L"       void

typedef struct Node     Node;
typedef struct String   String;
typedef struct Lsym     Lsym;
typedef struct List     List;
typedef struct Store    Store;
typedef struct Gc       Gc;
typedef struct Strc     Strc;
typedef struct Rplace   Rplace;
typedef struct Ptab     Ptab;
typedef struct Value    Value;
typedef struct Type     Type;
typedef struct Frtype   Frtype;

extern int      kernel;
extern int      remote;
extern int      text;
extern int      silent;
extern Fhdr     fhdr;
extern int      line;
extern Biobuf*  bout;
extern Biobuf*  io[32];
extern int      iop;
extern char     symbol[Strsize];
extern int      interactive;
extern int      na;
```

```
extern int      wtflag;
extern Map*     cormap;
extern Map*     symmap;
extern Lsym*    hash[Hashsize];
extern long     dogc;
extern Rplace*  ret;
extern char*    aout;
extern bool     gotint;
extern Gc*      gcl;
extern int      stacked;
extern jmp_buf  err;
extern Node*    prnt;
extern List*    tracelist;
extern int      initialising;
extern int      quiet;

extern void     (*expop[])(Node*, Node*);
```
⟨*macro* `expr` 121e⟩
```
extern int      fmtsize(Value *v) ;
```

⟨*enum* `_anon_ (acid/acid.h)2` 60a⟩

⟨*struct* `Type` 60b⟩

⟨*struct* `Frtype` 139b⟩

⟨*struct* `Ptab` 56a⟩

```
extern Ptab     ptab[Maxproc];
```

⟨*struct* `Rplace` 135a⟩

⟨*struct* `Gc` 64a⟩

⟨*struct* `Store` 62a⟩

⟨*struct* `List` 63b⟩

⟨*struct* `Value` 61⟩

⟨*struct* `Lsym` 65a⟩

⟨*struct* `Node` 58a⟩
⟨*constant* `ZN` 58b⟩

⟨*struct* `String` 62f⟩

```
List*   addlist(List*, List*);
List*   al(int);
Node*   an(int, Node*, Node*);
void    append(Node*, Node*, Node*);
int     fbool(Node*);
void    build(Node*);
void    call(char*, Node*, Node*, Node*, Node*);
void    catcher(void*, char*);
void    checkqid(int, int);
void    cmd(void);
Node*   con(vlong);
List*   construct(Node*);
void    ctrace(int);
```

```
void    decl(Node*);
void    defcomplex(Node*, Node*);
void    deinstall(int);
void    delete(List*, int n, Node*);
void    dostop(int);
Lsym*   enter(char*, int);
void    error(char*, ...);
void    execute(Node*);
void    fatal(char*, ...);
void    flatten(Node**, Node*);
void    gc(void);
char*   getstatus(int);
void*   gmalloc(long);
void    indir(Map*, uvlong, char, Node*);
void    installbuiltin(void);
void    kinit(void);
int     Lfmt(Fmt*);
int     listcmp(List*, List*);
int     listlen(List*);
List*   listvar(char*, vlong);
void    loadmodule(char*);
void    loadvars(void);
Lsym*   look(char*);
void    ltag(char*);
void    marklist(List*);
Lsym*   mkvar(char*);
void    msg(int, char*);
void    notes(int);
int     nproc(char**);
void    nthelem(List*, int, Node*);
int     numsym(char);
void    odot(Node*, Node*);
void    pcode(Node*, int);
void    pexpr(Node*);
int     popio(void);
void    pstr(String*);
void    pushfile(char*);
void    pushstr(Node*);
void    readtext(char*);
void    restartio(void);
uvlong  rget(Map*, char*);
String  *runenode(Rune*);
int     scmp(String*, String*);
void    sproc(int);
String* stradd(String*, String*);
String* straddrune(String*, Rune);
String* strnode(char*);
String* strnodlen(char*, int);
char*   system(void);
void    trlist(Map*, uvlong, uvlong, Symbol*);
void    unwind(void);
void    userinit(void);
void    varreg(void);
void    varsym(void);
Waitmsg*        waitfor(int);
void    whatis(Lsym*);
void    windir(Map*, Node*, Node*, Node*);
void    yyerror(char*, ...);
int     yylex(void);
int     yyparse(void);
```

⟨*enum* `_anon_ (acid/acid.h)`*3* 59⟩

Uses `Frtype` 378b, `Gc` 378b, `List` 378b, `Lsym` 378b, `Node` 378b, `Ptab` 378b, `Rplace` 378b, `Store` 378b, `String` 378b, `Type` 378b, and `Value` 378b.

F.4.2 acid/globals.c

⟨`acid/globals.c` 381a⟩≡

```
#include <u.h>
#include <libc.h>
#include <bio.h>
#include <mach.h>
#include "acid.h"

⟨global kernel 190c⟩
⟨global remote 191a⟩
⟨global text 73d⟩
⟨global silent 78b⟩
⟨global fhdr (acid/globals.c) 74c⟩
⟨global line (acid/globals.c) 70b⟩
⟨global bout 69d⟩
⟨global io 207b⟩
⟨global iop 207c⟩
⟨global symbol 104a⟩
⟨global interactive (acid/globals.c) 70a⟩
⟨global na 136a⟩
⟨global wtflag (acid/globals.c) 189⟩
⟨global cormap (acid/globals.c) 55b⟩
⟨global symmap (acid/globals.c) 55a⟩
⟨global hash 66a⟩
⟨global dogc 151a⟩
⟨global ret 134b⟩
⟨global aout 69a⟩
⟨global gotint 207e⟩
⟨global gcl 64b⟩
⟨global stacked 71a⟩
⟨global err 206a⟩
⟨global prnt 144b⟩
⟨global tracelist 397a⟩
⟨global initialising 71h⟩
⟨global quiet 69b⟩
⟨global ptab 56b⟩
```

F.4.3 acid/lex.c

⟨`acid/lex.c` 381b⟩≡

```
#include <u.h>
#include <libc.h>

#include <bio.h>
#include <ctype.h>
#include <mach.h>

#include "acid.h"
#include "y.tab.h"

⟨global keywds 111b⟩
```

⟨*global* `cmap` 107a⟩

⟨*function* `kinit` 112a⟩

```
typedef struct IOstack IOstack;
```
⟨*struct* `IOstack` 100a⟩
⟨*global* `lexio` 100b⟩

⟨*function* `pushfile` 100d⟩
⟨*function* `pushstr` 143b⟩

⟨*function* `restartio` 101d⟩
⟨*function* `popio` 101e⟩

⟨*function* `Lfmt` 204b⟩

⟨*function* `unlexc` 102b⟩
⟨*function* `lexc` 102a⟩

⟨*function* `escchar` 106b⟩

⟨*function* `eatstring` 107c⟩

⟨*function* `eatnl` 104e⟩

⟨*function* `yylex` 103a⟩

⟨*function* `numsym` 104b⟩

⟨*function* `enter` 67b⟩

⟨*function* `look` 66d⟩
⟨*function* `mkvar` 67a⟩

Uses `IOstack` 100a.

F.4.4 `acid/main.c`

⟨*function* `con` 382a⟩≡ (383c)

```
Node*
con(vlong v)
{
    Node *n;

    n = an(OCONST, ZN, ZN);
    n->ival = v;
    n->fmt = 'W';
    n->type = TINT;
    return n;
}
```

Uses `OCONST` 59, `TINT` 60a, `ZN` 58b, and `an()` 58c.

⟨*function* `system` 382b⟩≡ (383c)

```
char*
system(void)
{
    char *cpu, *p, *q;
    static char *kernel;
```

```
    cpu = getenv("cputype");
    if(cpu == 0) {
        cpu = "mips";
        print("$cputype not set; assuming %s\n", cpu);
    }
    p = getenv("terminal");
    if(p == 0 || (p=strchr(p, ' ')) == 0 || p[1] == ' ' || p[1] == 0) {
        p = "ch";
        print("missing or bad $terminal; assuming %s\n", p);
    }
    else{
        p++;
        q = strchr(p, ' ');
        if(q)
            *q = 0;
    }

    if(kernel != nil)
        free(kernel);
    kernel = smprint("/%s/9%s", cpu, p);

    return kernel;
}
```

⟨main() *(acid) format initializations* 383a⟩+≡ (69e) ◁204a

```
fmtinstall('x', xfmt);
```

⟨*function* xfmt 383b⟩≡ (383c)

```
int
xfmt(Fmt *f)
{
    f->flags ^= FmtSharp;
    return _ifmt(f);
}
```

⟨acid/main.c 383c⟩≡

```
#include <u.h>
#include <libc.h>
#include <bio.h>
#include <mach.h>
#include "acid.h"
#include "y.tab.h"

extern int _ifmt(Fmt*);
```

⟨*global* bioout 69c⟩
⟨*global* prog 88b⟩
⟨*global* lm 71e⟩
⟨*global* nlm 71d⟩
⟨*global* mtype 191e⟩

```
static  int attachfiles(char*, int);
int     xfmt(Fmt*);
int     isnumeric(char*);
void    die(void);
void    loadmoduleobjtype(void);
```

⟨*function usage (acid/main.c)* 24⟩

⟨*function* main 69e⟩

⟨*function* `attachfiles` 73e⟩

⟨*function die (acid/main.c)* 205a⟩

⟨*function* `loadmoduleobjtype` 77b⟩

⟨*function* `userinit` 78e⟩

⟨*function* `loadmodule` 78a⟩

⟨*function* `readtext` 74d⟩

⟨*function* `an` 58c⟩
⟨*function* `al` 63c⟩

⟨*function* `con` 382a⟩

⟨*function fatal (acid/main.c)* 205b⟩

⟨*function* `yyerror` 205c⟩

⟨*function* `marktree` 150a⟩
⟨*function* `marklist` 150b⟩

⟨*function* `gc` 149⟩

⟨*function* `gmalloc` 151b⟩

⟨*function* `checkqid` 87a⟩

⟨*function* `catcher` 208a⟩

⟨*function* `system` 382b⟩

⟨*function* `isnumeric` 209⟩

⟨*function* `xfmt` 383b⟩

F.4.5 `acid/util.c`

⟨*global* `syren` 384a⟩≡ (387a)
```
static int syren;
```

⟨*function* `unique` 384b⟩≡ (387a)
```
Lsym*
unique(char *buf, Sym *s)
{
    Lsym *l;
    int i, renamed;

    renamed = 0;
    strcpy(buf, s->name);
    for(;;) {
        l = look(buf);
        if(l == 0 || (l->lexval == Tid && l->v->set == 0))
            break;

        if(syren == 0 && !quiet) {
```

```
            print("Symbol renames:\n");
            syren = 1;
        }
        i = strlen(buf)+1;
        memmove(buf+1, buf, i);
        buf[0] = '$';
        renamed++;
        if(renamed > 5 && !quiet) {
            print("Too many renames; must be X source!\n");
            break;
        }
    }
    if(renamed && !quiet)
        print("\t%s=%s %c/%llux\n", s->name, buf, s->type, s->value);
    if(l == 0)
        l = enter(buf, Tid);
    return l;
}
```

Uses Tid, enter() 67b, look() 66d, quiet 69b, and syren-63 384a.

⟨*function rget (acid/util.c)* 385a⟩≡ (387a)

```
uvlong
rget(Map *map, char *reg)
{
    Lsym *s;
    ulong x;
    uvlong v;
    int ret;

    s = look(reg);
    if(s == 0)
        fatal("rget: %s\n", reg);

    switch(s->v->fmt){
    default:
        ret = get4(map, s->v->ival, &x);
        v = x;
        break;
    case 'V':
    case 'W':
    case 'Y':
    case 'Z':
        ret = get8(map, s->v->ival, &v);
        break;
    }
    if(ret < 0)
        error("can't get register %s: %r\n", reg);
    return v;
}
```

Uses get4() 350b, get8() 350a, and look() 66d.

⟨*function* runenode 385b⟩≡ (387a)

```
String*
runenode(Rune *name)
{
    int len;
    Rune *p;
    String *s;

    p = name;
```

```
    for(len = 0; *p; p++)
        len++;

    len++;
    len *= sizeof(Rune);
    s = gmalloc(sizeof(String)+len);
    s->string = (char*)s+sizeof(String);
    s->len = len;
    memmove(s->string, name, len);

    s->gclink = gcl;
    gcl = s;

    return s;
}
```

Uses gcl 64b and gmalloc() 151b.

⟨*function* stradd 386a⟩≡ (387a)

```
String*
stradd(String *l, String *r)
{
    int len;
    String *s;

    len = l->len+r->len;
    s = gmalloc(sizeof(String)+len+1);
    s->gclink = gcl;
    gcl = s;
    s->len = len;
    s->string = (char*)s+sizeof(String);
    memmove(s->string, l->string, l->len);
    memmove(s->string+l->len, r->string, r->len);
    s->string[s->len] = 0;
    return s;
}
```

Uses gcl 64b and gmalloc() 151b.

⟨*function* straddrune 386b⟩≡ (387a)

```
String*
straddrune(String *l, Rune r)
{
    int len;
    String *s;

    len = l->len+runelen(r);
    s = gmalloc(sizeof(String)+len+1);
    s->gclink = gcl;
    gcl = s;
    s->len = len;
    s->string = (char*)s+sizeof(String);
    memmove(s->string, l->string, l->len);
    runetochar(s->string+l->len, &r);
    s->string[s->len] = 0;
    return s;
}
```

Uses gcl 64b and gmalloc() 151b.

⟨*function* scmp 386c⟩≡ (387a)

```
int
```

```
scmp(String *sr, String *sl)
{
    if(sr->len != sl->len)
        return 0;

    if(memcmp(sr->string, sl->string, sl->len))
        return 0;

    return 1;
}
```

⟨acid/util.c 387a⟩≡

```
#include <u.h>
#include <libc.h>
#include <bio.h>
#include <ctype.h>
#include <mach.h>
#include "acid.h"
#include "y.tab.h"
```

⟨*global* syren 384a⟩

⟨*function* unique 384b⟩

⟨*function* varsym 79d⟩

⟨*function* varreg 76b⟩

⟨*function* loadvars 72b⟩

⟨*function rget (acid/util.c)* 385a⟩

⟨*function* strnodlen 63a⟩

⟨*function* strnode 62g⟩

⟨*function* runenode 385b⟩

⟨*function* stradd 386a⟩

⟨*function* straddrune 386b⟩

⟨*function* scmp 386c⟩

F.4.6 acid/exec.c

⟨*function* convflt 387b⟩≡ (391)

```
void
convflt(Node *r, char *flt)
{
    char c;

    c = flt[0];
    if(('a' <= c && c <= 'z') || ('A' <= c && c <= 'Z')) {
        r->type = TSTRING;
        r->fmt = 's';
        r->string = strnode(flt);
    }
    else {
```

```
        r->type = TFLOAT;
        r->fval = atof(flt);
    }
}
```

Uses TFLOAT 60a, TSTRING 60a, and strnode() 62g.

⟨*function* indir 388⟩≡ (391)

```
void
indir(Map *m, uvlong addr, char fmt, Node *r)
{
    int i;
    ulong lval;
    uvlong uvval;
    int ret;
    uchar cval;
    ushort sval;
    char buf[512], reg[12];

    r->op = OCONST;
    r->fmt = fmt;
    switch(fmt) {
    default:
        error("bad pointer format '%c' for *", fmt);
    case 'c':
    case 'C':
    case 'b':
        r->type = TINT;
        ret = get1(m, addr, &cval, 1);
        if (ret < 0)
            error("indir: %r");
        r->ival = cval;
        break;
    case 'x':
    case 'd':
    case 'u':
    case 'o':
    case 'q':
    case 'r':
        r->type = TINT;
        ret = get2(m, addr, &sval);
        if (ret < 0)
            error("indir: %r");
        r->ival = sval;
        break;
    case 'a':
    case 'A':
    case 'W':
        r->type = TINT;
        ret = geta(m, addr, &uvval);
        if (ret < 0)
            error("indir: %r");
        r->ival = uvval;
        break;
    case 'B':
    case 'X':
    case 'D':
    case 'U':
    case 'O':
    case 'Q':
        r->type = TINT;
```

```
    ret = get4(m, addr, &lval);
    if (ret < 0)
        error("indir: %r");
    r->ival = lval;
    break;
case 'V':
case 'Y':
case 'Z':
    r->type = TINT;
    ret = get8(m, addr, &uvval);
    if (ret < 0)
        error("indir: %r");
    r->ival = uvval;
    break;
case 's':
    r->type = TSTRING;
    for(i = 0; i < sizeof(buf)-1; i++) {
        ret = get1(m, addr, (uchar*)&buf[i], 1);
        if (ret < 0)
            error("indir: %r");
        addr++;
        if(buf[i] == '\0')
            break;
    }
    buf[i] = 0;
    if(i == 0)
        strcpy(buf, "(null)");
    r->string = strnode(buf);
    break;
case 'R':
    r->type = TSTRING;
    for(i = 0; i < sizeof(buf)-2; i += 2) {
        ret = get1(m, addr, (uchar*)&buf[i], 2);
        if (ret < 0)
            error("indir: %r");
        addr += 2;
        if(buf[i] == 0 && buf[i+1] == 0)
            break;
    }
    buf[i++] = 0;
    buf[i] = 0;
    r->string = runenode((Rune*)buf);
    break;
case 'i':
case 'I':
    if ((*machdata->das)(m, addr, fmt, buf, sizeof(buf)) < 0)
        error("indir: %r");
    r->type = TSTRING;
    r->fmt = 's';
    r->string = strnode(buf);
    break;
case 'f':
    ret = get1(m, addr, (uchar*)buf, mach->szfloat);
    if (ret < 0)
        error("indir: %r");
    machdata->sftos(buf, sizeof(buf), (void*) buf);
    convflt(r, buf);
    break;
case 'g':
    ret = get1(m, addr, (uchar*)buf, mach->szfloat);
```

```
        if (ret < 0)
            error("indir: %r");
        machdata->sftos(buf, sizeof(buf), (void*) buf);
        r->type = TSTRING;
        r->string = strnode(buf);
        break;
    case 'F':
        ret = get1(m, addr, (uchar*)buf, mach->szdouble);
        if (ret < 0)
            error("indir: %r");
        machdata->dftos(buf, sizeof(buf), (void*) buf);
        convflt(r, buf);
        break;
    case '3':   /* little endian ieee 80 with hole in bytes 8&9 */
        ret = get1(m, addr, (uchar*)reg, 10);
        if (ret < 0)
            error("indir: %r");
        memmove(reg+10, reg+8, 2);      /* open hole */
        memset(reg+8, 0, 2);            /* fill it */
        leieee80ftos(buf, sizeof(buf), reg);
        convflt(r, buf);
        break;
    case '8':   /* big-endian ieee 80 */
        ret = get1(m, addr, (uchar*)reg, 10);
        if (ret < 0)
            error("indir: %r");
        beieee80ftos(buf, sizeof(buf), reg);
        convflt(r, buf);
        break;
    case 'G':
        ret = get1(m, addr, (uchar*)buf, mach->szdouble);
        if (ret < 0)
            error("indir: %r");
        machdata->dftos(buf, sizeof(buf), (void*) buf);
        r->type = TSTRING;
        r->string = strnode(buf);
        break;
    }
}
```

Uses OCONST 59, TINT 60a, TSTRING 60a, beieee80ftos() 359e, convflt() 387b, get1() 351a, get2() 350c, get4() 350b, get8() 350a, geta() 349, leieee80ftos() 360a, mach 51a, machdata 355e, runenode() 385b, and strnode() 62g.

⟨*function* windir 390⟩≡ (391)

```
void
windir(Map *m, Node *addr, Node *rval, Node *r)
{
    uchar cval;
    ushort sval;
    long lval;
    Node res, aes;
    int ret;

    if(m == 0)
        error("no map for */@=");

    expr(rval, &res);
    expr(addr, &aes);

    if(aes.type != TINT)
        error("bad type lhs of @/*");
```

```
    if(m != cormap && wtflag == 0)
        error("not in write mode");

    r->type = res.type;
    r->fmt = res.fmt;
    r->Store = res.Store;

    switch(res.fmt) {
    default:
        error("bad pointer format '%c' for */@=", res.fmt);
    case 'c':
    case 'C':
    case 'b':
        cval = res.ival;
        ret = put1(m, aes.ival, &cval, 1);
        break;
    case 'r':
    case 'x':
    case 'd':
    case 'u':
    case 'o':
        sval = res.ival;
        ret = put2(m, aes.ival, sval);
        r->ival = sval;
        break;
    case 'a':
    case 'A':
    case 'W':
        ret = puta(m, aes.ival, res.ival);
        break;
    case 'B':
    case 'X':
    case 'D':
    case 'U':
    case 'O':
        lval = res.ival;
        ret = put4(m, aes.ival, lval);
        break;
    case 'V':
    case 'Y':
    case 'Z':
        ret = put8(m, aes.ival, res.ival);
        break;
    case 's':
    case 'R':
        ret = put1(m, aes.ival, (uchar*)res.string->string, res.string->len);
        break;
    }
    if (ret < 0)
        error("windir: %r");
}
```

Uses TINT 60a, put1() 352c, put2() 352b, put4() 352a, put8() 351c, and puta() 351b.

⟨acid/exec.c 391⟩≡

```
#include <u.h>
#include <libc.h>
#include <bio.h>
#include <ctype.h>
#include <mach.h>
```

```
#include "acid.h"

⟨function error (acid/exec.c) 206b⟩

⟨function unwind 207a⟩

⟨function execute 120a⟩

⟨function fbool 123a⟩

⟨function convflt 387b⟩

⟨function indir 388⟩

⟨function windir 390⟩

⟨function call 136b⟩
```

F.4.7 acid/proc.c

⟨acid/proc.c 392a⟩≡

```
#include <u.h>
#include <libc.h>
#include <bio.h>
#include <ctype.h>
#include <mach.h>
#include "acid.h"
#include "y.tab.h"

static void install(int);

⟨function nocore 87b⟩

⟨function sproc 85b⟩
⟨function nproc 82b⟩

⟨function notes (acid/proc.c) 85a⟩

⟨function dostop 88a⟩

⟨function install 83b⟩
⟨function deinstall 84⟩

⟨function msg 56d⟩

⟨function getstatus 164⟩

⟨function waitfor 413b⟩
```

F.4.8 acid/list.c

⟨global tail 392b⟩≡ (397c)

```
static List **tail;
```

⟨function construct 392c⟩≡ (397c)

```
List*
construct(Node *l)
{
```

```
    List *lh, **save;

    save = tail;
    lh = 0;
    tail = &lh;
    build(l);
    tail = save;

    return lh;
}
```

Uses build() 393b and tail-11 392b.

⟨*function* listlen 393a⟩≡ (397c)

```
int
listlen(List *l)
{
    int len;

    len = 0;
    while(l) {
        len++;
        l = l->next;
    }
    return len;
}
```

⟨*function* build 393b⟩≡ (397c)

```
void
build(Node *n)
{
    List *l;
    Node res;

    if(n == 0)
        return;

    switch(n->op) {
    case OLIST:
        build(n->left);
        build(n->right);
        return;
    default:
        expr(n, &res);
        l = al(res.type);
        l->Store = res.Store;
        *tail = l;
        tail = &l->next;
    }
}
```

Uses OLIST 59, al() 63c, build() 393b, and tail-11 392b.

⟨*function* addlist 393c⟩≡ (397c)

```
List*
addlist(List *l, List *r)
{
    List *f;

    if(l == 0)
        return r;
```

```
    for(f = l; f->next; f = f->next)
        ;
    f->next = r;

    return l;
}
```

⟨*function* append 394a⟩≡ (397c)

```
void
append(Node *r, Node *list, Node *val)
{
    List *l, *f;

    l = al(val->type);
    l->Store = val->Store;
    l->next = 0;

    r->op = OCONST;
    r->type = TLIST;

    if(list->l == 0) {
        list->l = l;
        r->l = l;
        return;
    }
    for(f = list->l; f->next; f = f->next)
        ;
    f->next = l;
    r->l = list->l;
}
```

Uses OCONST 59, TLIST 60a, and al() 63c.

⟨*function* listcmp 394b⟩≡ (397c)

```
int
listcmp(List *l, List *r)
{
    if(l == r)
        return 1;

    while(l) {
        if(r == 0)
            return 0;
        if(l->type != r->type)
            return 0;
        switch(l->type) {
        case TINT:
            if(l->ival != r->ival)
                return 0;
            break;
        case TFLOAT:
            if(l->fval != r->fval)
                return 0;
            break;
        case TSTRING:
            if(scmp(l->string, r->string) == 0)
                return 0;
            break;
        case TLIST:
            if(listcmp(l->l, r->l) == 0)
                return 0;
```

```
            break;
        }
        l = l->next;
        r = r->next;
    }
    if(l != r)
        return 0;
    return 1;
}
```

Uses TFLOAT 60a, TINT 60a, TLIST 60a, TSTRING 60a, listcmp() 394b, and scmp() 386c.

⟨*function* nthelem 395a⟩≡ (397c)

```
void
nthelem(List *l, int n, Node *res)
{
    if(n < 0)
        error("negative index in []");

    while(l && n--)
        l = l->next;

    res->op = OCONST;
    if(l == 0) {
        res->type = TLIST;
        res->l = 0;
        return;
    }
    res->type = l->type;
    res->Store = l->Store;
}
```

Uses OCONST 59 and TLIST 60a.

⟨*function* delete 395b⟩≡ (397c)

```
void
delete(List *l, int n, Node *res)
{
    List **tl;

    if(n < 0)
        error("negative index in delete");

    res->op = OCONST;
    res->type = TLIST;
    res->l = l;

    for(tl = &res->l; l && n--; l = l->next)
        tl = &l->next;

    if(l == 0)
        error("element beyond end of list");
    *tl = l->next;
}
```

⟨*function* listvar 395c⟩≡ (397c)

```
List*
listvar(char *s, vlong v)
{
    List *l, *tl;

    tl = al(TLIST);
```

```
    l = al(TSTRING);
    tl->l = l;
    l->fmt = 's';
    l->string = strnode(s);
    l->next = al(TINT);
    l = l->next;
    l->fmt = 'X';
    l->ival = v;

    return tl;
}
```

Uses TINT 60a, TLIST 60a, TSTRING 60a, al() 63c, and strnode() 62g.

⟨*function* listlocals 396a⟩≡ (397c)

```
static List*
listlocals(Map *map, Symbol *fn, uvlong fp)
{
    int i;
    uvlong val;
    Symbol s;
    List **tail, *l2;

    l2 = 0;
    tail = &l2;
    s = *fn;

    for(i = 0; localsym(&s, i); i++) {
        if(s.class != CAUTO)
            continue;
        if(s.name[0] == '.')
            continue;

        if(geta(map, fp-s.value, &val) > 0) {
            *tail = listvar(s.name, val);
            tail = &(*tail)->next;
        }
    }
    return l2;
}
```

Uses geta() 349, listvar() 395c, and localsym() 336b.

⟨*function* listparams 396b⟩≡ (397c)

```
static List*
listparams(Map *map, Symbol *fn, uvlong fp)
{
    int i;
    Symbol s;
    uvlong v;
    List **tail, *l2;

    l2 = 0;
    tail = &l2;
    fp += mach->szaddr;                 /* skip saved pc */
    s = *fn;
    for(i = 0; localsym(&s, i); i++) {
        if (s.class != CPARAM)
            continue;

        if(geta(map, fp+s.value, &v) > 0) {
```

```
            *tail = listvar(s.name, v);
            tail = &(*tail)->next;
        }
    }
    return l2;
}
```

Uses geta() 349, listvar() 395c, localsym() 336b, and mach 51a.

⟨*global* tracelist 397a⟩≡ (381a)

```
List*   tracelist;
```

⟨*function* trlist 397b⟩≡ (397c)

```
void
trlist(Map *map, uvlong pc, uvlong sp, Symbol *sym)
{
    List *q, *l;

    static List **tail;

    if (tracelist == 0) {               /* first time */
        tracelist = al(TLIST);
        tail = &tracelist;
    }

    q = al(TLIST);
    *tail = q;
    tail = &q->next;

    l = al(TINT);                       /* Function address */
    q->l = l;
    l->ival = sym->value;
    l->fmt = 'X';

    l->next = al(TINT);         /* called from address */
    l = l->next;
    l->ival = pc;
    l->fmt = 'Y';

    l->next = al(TLIST);                /* make list of params */
    l = l->next;
    l->l = listparams(map, sym, sp);

    l->next = al(TLIST);                /* make list of locals */
    l = l->next;
    l->l = listlocals(map, sym, sp);
}
```

Uses TINT 60a, TLIST 60a, al() 63c, listlocals() 396a, listparams() 396b, and tracelist 397a.

⟨acid/list.c 397c⟩≡

```
#include <u.h>
#include <libc.h>
#include <bio.h>
#include <ctype.h>
#include <mach.h>
#include "acid.h"

⟨global tail 392b⟩

⟨function construct 392c⟩
```

⟨*function* `listlen` 393a⟩

⟨*function* `build` 393b⟩

⟨*function* `addlist` 393c⟩

⟨*function* `append` 394a⟩

⟨*function* `listcmp` 394b⟩

⟨*function* `nthelem` 395a⟩

⟨*function* `delete` 395b⟩

⟨*function* `listvar` 395c⟩

⟨*function* `listlocals` 396a⟩

⟨*function* `listparams` 396b⟩

⟨*function* `trlist` 397b⟩

F.4.9 acid/dot.c

⟨*function* `srch` 398a⟩≡ (400b)

```
Type*
srch(Type *t, char *s)
{
    Type *f;

    f = 0;
    while(t) {
        if(strcmp(t->tag->name, s) == 0) {
            if(f == 0 || t->depth < f->depth)
                f = t;
        }
        t = t->next;
    }
    return f;
}
```

⟨*function* `odot` 398b⟩≡ (400b)

```
void
odot(Node *n, Node *r)
{
    char *s;
    Type *t;
    Node res;
    uvlong addr;

    s = n->sym->name;
    if(s == 0)
        fatal("dodot: no tag");

    expr(n->left, &res);
    if(res.comt == 0)
        error("no type specified for (expr).%s", s);

    if(res.type != TINT)
```

```
            error("pointer must be integer for (expr).%s", s);

        t = srch(res.comt, s);
        if(t == 0)
            error("no tag for (expr).%s", s);

        /* Propagate types */
        if(t->type)
            r->comt = t->type->lt;

        addr = res.ival+t->offset;
        if(t->fmt == 'a') {
            r->op = OCONST;
            r->fmt = 'a';
            r->type = TINT;
            r->ival = addr;
        }
        else
            indir(cormap, addr, t->fmt, r);

    }
```

Uses OCONST 59, TINT 60a, indir() 388, and srch() 398a.

⟨*global tail (acid/dot.c)* 399a⟩≡ (400b)

```
    static Type **tail;
```

⟨*global* base 399b⟩≡ (400b)

```
    static Lsym *base;
```

⟨*function* buildtype 399c⟩≡ (400b)

```
    void
    buildtype(Node *m, int d)
    {
        Type *t;

        if(m == ZN)
            return;

        switch(m->op) {
        case OLIST:
            buildtype(m->left, d);
            buildtype(m->right, d);
            break;

        case OCTRUCT:
            buildtype(m->left, d+1);
            break;
        default:
            t = malloc(sizeof(Type));
            t->next = 0;
            t->depth = d;
            t->tag = m->sym;
            t->base = base;
            t->offset = m->ival;
            if(m->left) {
                t->type = m->left->sym;
                t->fmt = 'a';
            }
            else {
                t->type = 0;
```

```
            if(m->right)
                t->type = m->right->sym;
            t->fmt = m->fmt;
        }

        *tail = t;
        tail = &t->next;
    }
}
```

Uses OCTRUCT 59, OLIST 59, ZN 58b, base-62 399b, buildtype() 399c, and tail-61 399a.

⟨*function* defcomplex 400a⟩≡ (400b)

```
void
defcomplex(Node *tn, Node *m)
{
    tail = &tn->sym->lt;
    base = tn->sym;
    buildtype(m, 0);
}
```

Uses base-62 399b, buildtype() 399c, and tail-61 399a.

⟨acid/dot.c 400b⟩≡

```
#include <u.h>
#include <libc.h>
#include <bio.h>
#include <ctype.h>
#include <mach.h>
#include "acid.h"

⟨function srch 398a⟩

⟨function odot 398b⟩

⟨global tail (acid/dot.c) 399a⟩
⟨global base 399b⟩

⟨function buildtype 399c⟩

⟨function defcomplex 400a⟩

⟨function decl 139d⟩
```

F.4.10 acid/print.c

⟨acid/print.c 400c⟩≡

```
#include <u.h>
#include <libc.h>
#include <bio.h>
#include <ctype.h>
#include <mach.h>
#include "acid.h"

⟨global binop 199b⟩

⟨global tabs 197b⟩
⟨global typenames 203b⟩

⟨function cmp 198a⟩
```

⟨*function* `fundefs` 197a⟩

⟨*function* `whatis` 195c⟩

⟨*function* `slist` 199a⟩

⟨*function* `pcode` 198b⟩

⟨*function* `pexpr` 200⟩

⟨*function* `pstr` 203a⟩

F.4.11 acid/expr.c

⟨*global* `fsize` 401a⟩≡ (407c)

```
static int fsize[] =
{
    ['A'] 4,
    ['B'] 4,
    ['C'] 1,
    ['D'] 4,
    ['F'] 8,
    ['G'] 8,
    ['O'] 4,
    ['Q'] 4,
    ['R'] 4,
    ['S'] 4,
    ['U'] 4,
    ['V'] 8,
    ['W'] 8,
    ['X'] 4,
    ['Y'] 8,
    ['Z'] 8,
    ['a'] 4,
    ['b'] 1,
    ['c'] 1,
    ['d'] 2,
    ['f'] 4,
    ['g'] 4,
    ['o'] 2,
    ['q'] 2,
    ['r'] 2,
    ['s'] 4,
    ['u'] 2,
    ['x'] 2,
    ['3'] 10,
    ['8'] 10,
};
```

⟨*function* `chklval` 401b⟩≡ (407c)

```
void
chklval(Node *lp)
{
    if(lp->op != ONAME)
        error("need l-value");
}
```

Uses `ONAME` 59.

⟨*function* `olist` 402a⟩≡ (407c)

```
void
olist(Node *n, Node *res)
{
    expr(n->left, res);
    expr(n->right, res);
}
```

⟨*function* `oeval` 402b⟩≡ (407c)

```
void
oeval(Node *n, Node *res)
{
    expr(n->left, res);
    if(res->type != TCODE)
        error("bad type for eval");
    expr(res->cc, res);
}
```

Uses `TCODE` 60a.

⟨*function* `ocast` 402c⟩≡ (407c)

```
void
ocast(Node *n, Node *res)
{
    if(n->sym->lt == 0)
        error("%s is not a complex type", n->sym->name);

    expr(n->left, res);
    res->comt = n->sym->lt;
    res->fmt = 'a';
}
```

⟨*function* `oindm` 402d⟩≡ (407c)

```
void
oindm(Node *n, Node *res)
{
    Map *m;
    Node l;

    m = cormap;
    if(m == 0)
        m = symmap;
    expr(n->left, &l);
    if(l.type != TINT)
        error("bad type for *");
    if(m == 0)
        error("no map for *");
    indir(m, l.ival, l.fmt, res);
    res->comt = l.comt;
}
```

Uses `TINT` 60a and `indir()` 388.

⟨*function* `oindc` 402e⟩≡ (407c)

```
void
oindc(Node *n, Node *res)
{
    Map *m;
    Node l;

    m = symmap;
    if(m == 0)
```

```
        m = cormap;
    expr(n->left, &l);
    if(l.type != TINT)
        error("bad type for @");
    if(m == 0)
        error("no map for @");
    indir(m, l.ival, l.fmt, res);
    res->comt = l.comt;
}
```

Uses TINT 60a and indir() 388.

⟨*function* oframe 403a⟩≡ (407c)

```
void
oframe(Node *n, Node *res)
{
    char *p;
    Node *lp;
    uvlong ival;
    Frtype *f;

    p = n->sym->name;
    while(*p && *p == '$')
        p++;
    lp = n->left;
    if(localaddr(cormap, p, lp->sym->name, &ival, rget) < 0)
        error("colon: %r");

    res->ival = ival;
    res->op = OCONST;
    res->fmt = 'X';
    res->type = TINT;

    /* Try and set comt */
    for(f = n->sym->local; f; f = f->next) {
        if(f->var == lp->sym) {
            res->comt = f->type;
            res->fmt = 'a';
            break;
        }
    }
}
```

⟨*function* oindex 403b⟩≡ (407c)

```
void
oindex(Node *n, Node *res)
{
    Node l, r;

    expr(n->left, &l);
    expr(n->right, &r);

    if(r.type != TINT)
        error("bad type for []");

    switch(l.type) {
    default:
        error("lhs[] has bad type");
    case TINT:
        indir(cormap, l.ival+(r.ival*fsize[l.fmt]), l.fmt, res);
        res->comt = l.comt;
```

```
        res->fmt = l.fmt;
        break;
    case TLIST:
        nthelem(l.l, r.ival, res);
        break;
    case TSTRING:
        res->ival = 0;
        if(r.ival >= 0 && r.ival < l.string->len) {
            int xx8;    /* to get around bug in vc */
            xx8 = r.ival;
            res->ival = l.string->string[xx8];
        }
        res->op = OCONST;
        res->type = TINT;
        res->fmt = 'c';
        break;
    }
}
```

Uses OCONST 59, TINT 60a, TLIST 60a, TSTRING 60a, fsize-17 401a, indir() 388, and nthelem() 395a.

⟨*function* oappend 404a⟩≡ (407c)

```
void
oappend(Node *n, Node *res)
{
    Value *v;
    Node r, l;
    int  empty;

    expr(n->left, &l);
    expr(n->right, &r);
    if(l.type != TLIST)
        error("must append to list");
    empty = (l.l == nil && (n->left->op == ONAME));
    append(res, &l, &r);
    if(empty) {
        v = n->left->sym->v;
        v->type = res->type;
        v->Store = res->Store;
        v->comt = res->comt;
    }
}
```

Uses ONAME 59, TLIST 60a, and append() 394a.

⟨*function* odelete 404b⟩≡ (407c)

```
void
odelete(Node *n, Node *res)
{
    Node l, r;

    expr(n->left, &l);
    expr(n->right, &r);
    if(l.type != TLIST)
        error("must delete from list");
    if(r.type != TINT)
        error("delete index must be integer");

    delete(l.l, r.ival, res);
}
```

Uses TINT 60a and TLIST 60a.

⟨*function* ohead 405a⟩≡ (407c)

```
void
ohead(Node *n, Node *res)
{
    Node l;

    expr(n->left, &l);
    if(l.type != TLIST)
        error("head needs list");
    res->op = OCONST;
    if(l.l) {
        res->type = l.l->type;
        res->Store = l.l->Store;
    }
    else {
        res->type = TLIST;
        res->l = 0;
    }
}
```

Uses OCONST 59 and TLIST 60a.

⟨*function* otail 405b⟩≡ (407c)

```
void
otail(Node *n, Node *res)
{
    Node l;

    expr(n->left, &l);
    if(l.type != TLIST)
        error("tail needs list");
    res->op = OCONST;
    res->type = TLIST;
    if(l.l)
        res->l = l.l->next;
    else
        res->l = 0;
}
```

Uses OCONST 59 and TLIST 60a.

⟨*function* oconst 405c⟩≡ (407c)

```
void
oconst(Node *n, Node *res)
{
    res->op = OCONST;
    res->type = n->type;
    res->Store = n->Store;
    res->comt = n->comt;
}
```

Uses OCONST 59.

⟨*function* oname 405d⟩≡ (407c)

```
void
oname(Node *n, Node *res)
{
    Value *v;

    v = n->sym->v;
    if(v->set == 0)
        error("%s used but not set", n->sym->name);
```

```
    res->op = OCONST;
    res->type = v->type;
    res->Store = v->Store;
    res->comt = v->comt;
}
```

Uses OCONST 59.

⟨*function* octruct 406a⟩≡ (407c)

```
void
octruct(Node *n, Node *res)
{
    res->op = OCONST;
    res->type = TLIST;
    res->l = construct(n->left);
}
```

Uses OCONST 59, TLIST 60a, and construct() 392c.

⟨*function* oasgn 406b⟩≡ (407c)

```
void
oasgn(Node *n, Node *res)
{
    Node *lp, r;
    Value *v;

    lp = n->left;
    switch(lp->op) {
    case OINDM:
        windir(cormap, lp->left, n->right, res);
        break;
    case OINDC:
        windir(symmap, lp->left, n->right, res);
        break;
    default:
        chklval(lp);
        v = lp->sym->v;
        expr(n->right, &r);
        v->set = 1;
        v->type = r.type;
        v->Store = r.Store;
        res->op = OCONST;
        res->type = v->type;
        res->Store = v->Store;
        res->comt = v->comt;
    }
}
```

Uses OCONST 59, OINDC 59, OINDM 59, chklval() 401b, and windir() 390.

⟨*function* oeinc 406c⟩≡ (407c)

```
void
oeinc(Node *n, Node *res)
{
    Value *v;

    chklval(n->left);
    v = n->left->sym->v;
    res->op = OCONST;
    res->type = v->type;
    switch(v->type) {
    case TINT:
```

```
        if(n->op == OEDEC)
            v->ival -= fmtsize(v);
        else
            v->ival += fmtsize(v);
        break;
    case TFLOAT:
        if(n->op == OEDEC)
            v->fval--;
        else
            v->fval++;
        break;
    default:
        error("bad type for pre --/++");
    }
    res->Store = v->Store;
}
```

Uses OCONST 59, OEDEC 59, TFLOAT 60a, TINT 60a, chklval() 401b, and fmtsize() 148c.

⟨*function* opinc 407a⟩≡ (407c)

```
void
opinc(Node *n, Node *res)
{
    Value *v;

    chklval(n->left);
    v = n->left->sym->v;
    res->op = OCONST;
    res->type = v->type;
    res->Store = v->Store;
    switch(v->type) {
    case TINT:
        if(n->op == OPDEC)
            v->ival -= fmtsize(v);
        else
            v->ival += fmtsize(v);
        break;
    case TFLOAT:
        if(n->op == OPDEC)
            v->fval--;
        else
            v->fval++;
        break;
    default:
        error("bad type for post --/++");
    }
}
```

Uses OCONST 59, OPDEC 59, TFLOAT 60a, TINT 60a, chklval() 401b, and fmtsize() 148c.

⟨*function* ofmt 407b⟩≡ (407c)

```
void
ofmt(Node *n, Node *res)
{
    expr(n->left, res);
    res->fmt = n->right->ival;
}
```

⟨acid/expr.c 407c⟩≡

```
#include <u.h>
#include <libc.h>
#include <bio.h>
```

```
#include <ctype.h>
#include <mach.h>
#include "acid.h"
```

⟨*global* `fsize` 401a⟩

⟨*function* `fmtsize` 148c⟩

⟨*function* `chklval` 401b⟩

⟨*function* `olist` 402a⟩

⟨*function* `oeval` 402b⟩

⟨*function* `ocast` 402c⟩

⟨*function* `oindm` 402d⟩
⟨*function* `oindc` 402e⟩

⟨*function* `oframe` 403a⟩

⟨*function* `oindex` 403b⟩

⟨*function* `oappend` 404a⟩

⟨*function* `odelete` 404b⟩

⟨*function* `ohead` 405a⟩
⟨*function* `otail` 405b⟩

⟨*function* `oconst` 405c⟩

⟨*function* `oname` 405d⟩

⟨*function* `octruct` 406a⟩

⟨*function* `oasgn` 406b⟩

⟨*function* `oadd` 129⟩
⟨*function* `osub` 130⟩
⟨*function* `omul` 131⟩
⟨*function* `odiv` 132a⟩
⟨*function* `omod` 132b⟩

⟨*function* `olsh` 133a⟩
⟨*function* `orsh` 133b⟩

⟨*function* `olt` 123b⟩
⟨*function* `ogt` 124⟩
⟨*function* `oleq` 125a⟩
⟨*function* `ogeq` 125b⟩
⟨*function* `oeq` 126⟩

⟨*function* `oland` 127a⟩
⟨*function* `oxor` 127b⟩
⟨*function* `olor` 127c⟩
⟨*function* `ocand` 128a⟩
⟨*function* `onot` 128b⟩
⟨*function* `ocor` 128c⟩

⟨*function* `oeinc` 406c⟩
⟨*function* `opinc` 407a⟩

⟨*function* `ocall` 135d⟩

⟨*function* `ofmt` 407b⟩

⟨*function* `owhat` 195b⟩

⟨*global* `expop` 121f⟩

F.4.12 acid/builtin.c

⟨*function* `mkprint` 409a⟩≡ (423b)
```
void
mkprint(Lsym *s)
{
    prnt = malloc(sizeof(Node));
    memset(prnt, 0, sizeof(Node));
    prnt->op = OCALL;
    prnt->left = malloc(sizeof(Node));
    memset(prnt->left, 0, sizeof(Node));
    prnt->left->sym = s;
}
```
Uses `OCALL` 59 and `prnt` 144b.

⟨*function* `dosysr1` 409b⟩≡ (423b)
```
void
dosysr1(Node *r, Node*)
{
    extern int nop(void);

    r->op = OCONST;
    r->type = TINT;
    r->fmt = 'D';
//    r->ival = sysr1();
//pad: now it's sysnop
    r->ival = nop();
}
```
Uses `OCONST` 59 and `TINT` 60a.

⟨*function* `match` 409c⟩≡ (423b)
```
void
match(Node *r, Node *args)
{
    int i;
    List *f;
    Node *av[Maxarg];
    Node resi, resl;

    na = 0;
    flatten(av, args);
    if(na != 2)
        error("match(obj, list): arg count");

    expr(av[1], &resl);
    if(resl.type != TLIST)
        error("match(obj, list): need list");
```

```
    expr(av[0], &resi);

    r->op = OCONST;
    r->type = TINT;
    r->fmt = 'D';
    r->ival = -1;

    i = 0;
    for(f = resl.l; f; f = f->next) {
        if(resi.type == f->type) {
            switch(resi.type) {
            case TINT:
                if(resi.ival == f->ival) {
                    r->ival = i;
                    return;
                }
                break;
            case TFLOAT:
                if(resi.fval == f->fval) {
                    r->ival = i;
                    return;
                }
                break;
            case TSTRING:
                if(scmp(resi.string, f->string)) {
                    r->ival = i;
                    return;
                }
                break;
            case TLIST:
                error("match(obj, list): not defined for list");
            }
        }
        i++;
    }
}
```

Uses Maxarg 82a, OCONST 59, TFLOAT 60a, TINT 60a, TLIST 60a, TSTRING 60a, flatten() 419, na 136a, and scmp() 386c.

⟨*function* kill 410a⟩≡ (423b)

```
void
kill(Node *r, Node *args)
{
    Node res;

    USED(r);
    if(args == 0)
        error("kill(pid): no pid");
    expr(args, &res);
    if(res.type != TINT)
        error("kill(pid): arg type");

    msg(res.ival, "kill");
    deinstall(res.ival);
}
```

Uses TINT 60a, deinstall() 84, and msg() 56d.

⟨*function* reason 410b⟩≡ (423b)

```
void
reason(Node *r, Node *args)
{
```

```
    Node res;

    if(args == 0)
        error("reason(cause): no cause");
    expr(args, &res);
    if(res.type != TINT)
        error("reason(cause): arg type");

    r->op = OCONST;
    r->type = TSTRING;
    r->fmt = 's';
    r->string = strnode((*machdata->excep)(cormap, rget));
}
```

Uses OCONST 59, TINT 60a, TSTRING 60a, machdata 355e, and strnode() 62g.

⟨*function* follow 411a⟩≡ (423b)

```
void
follow(Node *r, Node *args)
{
    int n, i;
    Node res;
    uvlong f[10];
    List **tail, *l;

    if(args == 0)
        error("follow(addr): no addr");
    expr(args, &res);
    if(res.type != TINT)
        error("follow(addr): arg type");

    n = (*machdata->foll)(cormap, res.ival, rget, f);
    if (n < 0)
        error("follow(addr): %r");
    tail = &r->l;
    for(i = 0; i < n; i++) {
        l = al(TINT);
        l->ival = f[i];
        l->fmt = 'X';
        *tail = l;
        tail = &l->next;
    }
}
```

Uses TINT 60a, al() 63c, and machdata 355e.

⟨*function* funcbound 411b⟩≡ (423b)

```
void
funcbound(Node *r, Node *args)
{
    int n;
    Node res;
    uvlong bounds[2];
    List *l;

    if(args == 0)
        error("fnbound(addr): no addr");
    expr(args, &res);
    if(res.type != TINT)
        error("fnbound(addr): arg type");

    n = fnbound(res.ival, bounds);
```

```
    if (n != 0) {
        r->l = al(TINT);
        l = r->l;
        l->ival = bounds[0];
        l->fmt = 'X';
        l->next = al(TINT);
        l = l->next;
        l->ival = bounds[1];
        l->fmt = 'X';
    }
}
```

Uses TINT 60a, al() 63c, and fnbound() 336a.

⟨*function* setproc 412a⟩≡ (423b)

```
void
setproc(Node *r, Node *args)
{
    Node res;

    USED(r);
    if(args == 0)
        error("setproc(pid): no pid");
    expr(args, &res);
    if(res.type != TINT)
        error("setproc(pid): arg type");

    sproc(res.ival);
}
```

Uses TINT 60a and sproc() 85b.

⟨*function* filepc 412b⟩≡ (423b)

```
void
filepc(Node *r, Node *args)
{
    Node res;
    char *p, c;

    if(args == 0)
        error("filepc(filename:line): arg count");
    expr(args, &res);
    if(res.type != TSTRING)
        error("filepc(filename:line): arg type");

    p = strchr(res.string->string, ':');
    if(p == 0)
        error("filepc(filename:line): bad arg format");

    c = *p;
    *p++ = '\0';
    r->ival = file2pc(res.string->string, strtol(p, 0, 0));
    p[-1] = c;
    if(r->ival == ~0)
        error("filepc(filename:line): can't find address");

    r->op = OCONST;
    r->type = TINT;
    r->fmt = 'V';
}
```

Uses OCONST 59, TINT 60a, TSTRING 60a, and file2pc() 337b.

⟨*function* `rc` 413a⟩≡ (423b)

```
void
rc(Node *r, Node *args)
{
    Node res;
    int pid;
    char *p, *q, *argv[4];
    Waitmsg *w;

    USED(r);
    if(args == 0)
        error("error(string): arg count");
    expr(args, &res);
    if(res.type != TSTRING)
        error("error(string): arg type");

    argv[0] = "/bin/rc";
    argv[1] = "-c";
    argv[2] = res.string->string;
    argv[3] = 0;

    pid = fork();
    switch(pid) {
    case -1:
        error("fork %r");
    case 0:
        exec("/bin/rc", argv);
        exits(0);
    default:
        w = waitfor(pid);
        break;
    }
    p = w->msg;
    q = strrchr(p, ':');
    if (q)
        p = q+1;

    r->op = OCONST;
    r->type = TSTRING;
    r->string = strnode(p);
    free(w);
    r->fmt = 's';
}
```

Uses `OCONST` 59, `TSTRING` 60a, `strnode()` 62g, and `waitfor()` 413b.

⟨*function* `waitfor` 413b⟩≡ (392a)

```
Waitmsg*
waitfor(int pid)
{
    Waitmsg *w;

    for(;;) {
        if((w = wait()) == nil)
            error("wait %r");
        if(w->pid == pid)
            return w;
        free(w);
    }
}
```

⟨*function* `doerror` 414a⟩≡ (423b)

```
void
doerror(Node *r, Node *args)
{
    Node res;

    USED(r);
    if(args == 0)
        error("error(string): arg count");
    expr(args, &res);
    if(res.type != TSTRING)
        error("error(string): arg type");

    error(res.string->string);
}
```

Uses `TSTRING` 60a.

⟨*function* `doaccess` 414b⟩≡ (423b)

```
void
doaccess(Node *r, Node *args)
{
    Node res;

    if(args == 0)
        error("access(filename): arg count");
    expr(args, &res);
    if(res.type != TSTRING)
        error("access(filename): arg type");

    r->op = OCONST;
    r->type = TINT;
    r->ival = 0;
    if(access(res.string->string, 4) == 0)
        r->ival = 1;
}
```

Uses `OCONST` 59, `TINT` 60a, and `TSTRING` 60a.

⟨*function* `readfile` 414c⟩≡ (423b)

```
void
readfile(Node *r, Node *args)
{
    Node res;
    int n, fd;
    char *buf;
    Dir *db;

    if(args == 0)
        error("readfile(filename): arg count");
    expr(args, &res);
    if(res.type != TSTRING)
        error("readfile(filename): arg type");

    fd = open(res.string->string, OREAD);
    if(fd < 0)
        return;

    db = dirfstat(fd);
    if(db == nil || db->length == 0)
        n = 8192;
    else
```

```
        n = db->length;
    free(db);

    buf = malloc(n);
    n = read(fd, buf, n);

    if(n > 0) {
        r->op = OCONST;
        r->type = TSTRING;
        r->string = strnodlen(buf, n);
        r->fmt = 's';
    }
    free(buf);
    close(fd);
}
```

Uses OCONST 59, TSTRING 60a, and strnodlen() 63a.

⟨*function getfile (acid/builtin.c)* 415⟩≡ (423b)

```
void
getfile(Node *r, Node *args)
{
    int n;
    char *p;
    Node res;
    String *s;
    Biobuf *bp;
    List **l, *new;

    if(args == 0)
        error("file(filename): arg count");
    expr(args, &res);
    if(res.type != TSTRING)
        error("file(filename): arg type");

    r->op = OCONST;
    r->type = TLIST;
    r->l = 0;

    p = res.string->string;
    bp = Bopen(p, OREAD);
    if(bp == 0)
        return;

    l = &r->l;
    for(;;) {
        p = Brdline(bp, '\n');
        n = Blinelen(bp);
        if(p == 0) {
            if(n == 0)
                break;
            s = strnodlen(0, n);
            Bread(bp, s->string, n);
        }
        else
            s = strnodlen(p, n-1);

        new = al(TSTRING);
        new->string = s;
        new->fmt = 's';
        *l = new;
```

```
            l = &new->next;
        }
        Bterm(bp);
    }
```

Uses OCONST 59, TLIST 60a, TSTRING 60a, al() 63c, and strnodlen() 63a.

⟨*function* cvtatof 416a⟩≡ (423b)

```
    void
    cvtatof(Node *r, Node *args)
    {
        Node res;

        if(args == 0)
            error("atof(string): arg count");
        expr(args, &res);
        if(res.type != TSTRING)
            error("atof(string): arg type");

        r->op = OCONST;
        r->type = TFLOAT;
        r->fval = atof(res.string->string);
        r->fmt = 'f';
    }
```

Uses OCONST 59, TFLOAT 60a, and TSTRING 60a.

⟨*function* cvtatoi 416b⟩≡ (423b)

```
    void
    cvtatoi(Node *r, Node *args)
    {
        Node res;

        if(args == 0)
            error("atoi(string): arg count");
        expr(args, &res);
        if(res.type != TSTRING)
            error("atoi(string): arg type");

        r->op = OCONST;
        r->type = TINT;
        r->ival = strtoull(res.string->string, 0, 0);
        r->fmt = 'V';
    }
```

Uses OCONST 59, TINT 60a, and TSTRING 60a.

⟨*global* fmtflags 416c⟩≡ (423b)

```
    static char *fmtflags = "-0123456789. #,u";
```

Uses fmtflags-9 416c.

⟨*global* fmtverbs 416d⟩≡ (423b)

```
    static char *fmtverbs = "bdox";
```

Uses fmtverbs-10 416d.

⟨*function* acidfmt 416e⟩≡ (423b)

```
    static int
    acidfmt(char *fmt, char *buf, int blen)
    {
        char *r, *w, *e;

        w = buf;
```

```
    e = buf+blen;
    for(r=fmt; *r; r++){
        if(w >= e)
            return -1;
        if(*r != '%'){
            *w++ = *r;
            continue;
        }
        if(*r == '%'){
            *w++ = *r++;
            if(*r == '%'){
                if(w >= e)
                    return -1;
                *w++ = *r;
                continue;
            }
            while(*r && strchr(fmtflags, *r)){
                if(w >= e)
                    return -1;
                *w++ = *r++;
            }
            if(*r == 0 || strchr(fmtverbs, *r) == nil)
                return -1;
            if(w+3 > e)
                return -1;
            *w++ = 'l';
            *w++ = 'l';
            *w++ = *r;
        }
    }
    if(w >= e)
        return -1;
    *w = 0;

    return 0;
}
```

Uses `fmtflags-9` 416c and `fmtverbs-10` 416d.

⟨*function* `cvtitoa` 417⟩≡ (423b)

```
void
cvtitoa(Node *r, Node *args)
{
    Node res;
    Node *av[Maxarg];
    vlong ival;
    char buf[128], fmt[32];

    if(args == 0)
err:
        error("itoa(number [, fmt]): arg count");
    na = 0;
    flatten(av, args);
    if(na == 0 || na > 2)
        goto err;
    expr(av[0], &res);
    if(res.type != TINT)
        error("itoa(number [, fmt]): arg type");
    ival = res.ival;
    strncpy(fmt, "%lld", sizeof(fmt));
    if(na == 2){
```

```
            expr(av[1], &res);
            if(res.type != TSTRING)
                error("itoa(number [, fmt]): fmt type");
            if(acidfmt(res.string->string, fmt, sizeof(buf)))
                error("itoa(number [, fmt]): malformed fmt");
        }

        snprint(buf, sizeof(buf), fmt, ival);
        r->op = OCONST;
        r->type = TSTRING;
        r->string = strnode(buf);
        r->fmt = 's';
    }
```

Uses Maxarg 82a, OCONST 59, TINT 60a, TSTRING 60a, acidfmt() 416e, flatten() 419, na 136a, and strnode() 62g.

⟨*function* mapent 418a⟩≡ (423b)

```
    List*
    mapent(Map *m)
    {
        int i;
        List *l, *n, **t, *h;

        h = 0;
        t = &h;
        for(i = 0; i < m->nsegs; i++) {
            if(m->seg[i].inuse == 0)
                continue;
            l = al(TSTRING);
            n = al(TLIST);
            n->l = l;
            *t = n;
            t = &n->next;
            l->string = strnode(m->seg[i].name);
            l->fmt = 's';
            l->next = al(TINT);
            l = l->next;
            l->ival = m->seg[i].b;
            l->fmt = 'W';
            l->next = al(TINT);
            l = l->next;
            l->ival = m->seg[i].e;
            l->fmt = 'W';
            l->next = al(TINT);
            l = l->next;
            l->ival = m->seg[i].f;
            l->fmt = 'W';
        }
        return h;
    }
```

Uses TINT 60a, TLIST 60a, TSTRING 60a, al() 63c, and strnode() 62g.

⟨*function* map 418b⟩≡ (423b)

```
    void
    map(Node *r, Node *args)
    {
        int i;
        Map *m;
        List *l;
        char *ent;
        Node *av[Maxarg], res;
```

```
    na = 0;
    flatten(av, args);

    if(na != 0) {
        expr(av[0], &res);
        if(res.type != TLIST)
            error("map(list): map needs a list");
        if(listlen(res.l) != 4)
            error("map(list): list must have 4 entries");

        l = res.l;
        if(l->type != TSTRING)
            error("map name must be a string");
        ent = l->string->string;
        m = symmap;
        i = findseg(m, ent);
        if(i < 0) {
            m = cormap;
            i = findseg(m, ent);
        }
        if(i < 0)
            error("%s is not a map entry", ent);
        l = l->next;
        if(l->type != TINT)
            error("map entry not int");
        m->seg[i].b = l->ival;
        if (strcmp(ent, "text") == 0)
            textseg(l->ival, &fhdr);
        l = l->next;
        if(l->type != TINT)
            error("map entry not int");
        m->seg[i].e = l->ival;
        l = l->next;
        if(l->type != TINT)
            error("map entry not int");
        m->seg[i].f = l->ival;
    }

    r->type = TLIST;
    r->l = 0;
    if(symmap)
        r->l = mapent(symmap);
    if(cormap) {
        if(r->l == 0)
            r->l = mapent(cormap);
        else {
            for(l = r->l; l->next; l = l->next)
                ;
            l->next = mapent(cormap);
        }
    }
}
```

Uses `Maxarg` 82a, `TINT` 60a, `TLIST` 60a, `TSTRING` 60a, `fhdr` 74c, `findseg()` 98, `flatten()` 419, `listlen()` 393a, `mapent()` 418a, `na` 136a, and `textseg()` 328a.

⟨*function* `flatten` 419⟩≡ (423b)

```
void
flatten(Node **av, Node *n)
{
```

```
    if(n == 0)
        return;

    switch(n->op) {
    case OLIST:
        flatten(av, n->left);
        flatten(av, n->right);
        break;
    default:
        av[na++] = n;
        if(na >= Maxarg)
            error("too many function arguments");
        break;
    }
}
```

Uses Maxarg 82a, OLIST 59, flatten() 419, and na 136a.

⟨*function* strace 420a⟩≡ (423b)

```
void
strace(Node *r, Node *args)
{
    Node *av[Maxarg], *n, res;
    uvlong pc, sp;

    na = 0;
    flatten(av, args);
    if(na != 3)
        error("strace(pc, sp, link): arg count");

    n = av[0];
    expr(n, &res);
    if(res.type != TINT)
        error("strace(pc, sp, link): pc bad type");
    pc = res.ival;

    n = av[1];
    expr(n, &res);
    if(res.type != TINT)
        error("strace(pc, sp, link): sp bad type");
    sp = res.ival;

    n = av[2];
    expr(n, &res);
    if(res.type != TINT)
        error("strace(pc, sp, link): link bad type");

    tracelist = 0;
    if ((*machdata->ctrace)(cormap, pc, sp, res.ival, trlist) <= 0)
        error("no stack frame: %r");
    r->type = TLIST;
    r->l = tracelist;
}
```

Uses Maxarg 82a, TINT 60a, TLIST 60a, flatten() 419, machdata 355e, na 136a, tracelist 397a, and trlist() 397b.

⟨*function* regerror 420b⟩≡ (423b)

```
void
regerror(char *msg)
{
    error(msg);
}
```

⟨*function* `regexp` 421a⟩≡ (423b)

```
void
regexp(Node *r, Node *args)
{
    Node res;
    Reprog *rp;
    Node *av[Maxarg];

    na = 0;
    flatten(av, args);
    if(na != 2)
        error("regexp(pattern, string): arg count");
    expr(av[0], &res);
    if(res.type != TSTRING)
        error("regexp(pattern, string): pattern must be string");
    rp = regcomp(res.string->string);
    if(rp == 0)
        return;

    expr(av[1], &res);
    if(res.type != TSTRING)
        error("regexp(pattern, string): bad string");

    r->fmt = 'D';
    r->type = TINT;
    r->ival = regexec(rp, res.string->string, 0, 0);
    free(rp);
}
```

Uses `Maxarg` 82a, `TINT` 60a, `TSTRING` 60a, `flatten()` 419, and `na` 136a.

⟨*function* `printto` 421b⟩≡ (423b)

```
void
printto(Node *r, Node *args)
{
    int fd;
    Biobuf *b;
    int i, nas;
    Node res, *av[Maxarg];

    USED(r);
    na = 0;
    flatten(av, args);
    nas = na;

    expr(av[0], &res);
    if(res.type != TSTRING)
        error("printto(string, ...): need string");

    fd = create(res.string->string, OWRITE, 0666);
    if(fd < 0)
        fd = open(res.string->string, OWRITE);
    if(fd < 0)
        error("printto: open %s: %r", res.string->string);

    b = gmalloc(sizeof(Biobuf));
    Binit(b, fd, OWRITE);

    Bflush(bout);
    io[iop++] = bout;
    bout = b;
```

```
    for(i = 1; i < nas; i++) {
        expr(av[i], &res);
        switch(res.type) {
        default:
            if(comx(res))
                break;
            patom(res.type, &res.Store);
            break;
        case TLIST:
            blprint(res.l);
            break;
        }
    }
    if(ret == 0)
        Bputc(bout, '\n');

    Bterm(b);
    close(fd);
    free(b);
    bout = io[--iop];
}
```

Uses `Maxarg` 82a, `TLIST` 60a, `TSTRING` 60a, `blprint()` 144e, `bout` 69d, `comx()` 145a, `flatten()` 419, `gmalloc()` 151b, `io` 207b, `iop` 207c, `na` 136a, `patom()` 145b, and `ret` 134b.

⟨*function* `pcfile` 422a⟩≡ (423b)

```
void
pcfile(Node *r, Node *args)
{
    Node res;
    char *p, buf[128];

    if(args == 0)
        error("pcfile(addr): arg count");
    expr(args, &res);
    if(res.type != TINT)
        error("pcfile(addr): arg type");

    r->type = TSTRING;
    r->fmt = 's';
    if(fileline(buf, sizeof(buf), res.ival) == 0) {
        r->string = strnode("?file?");
        return;
    }
    p = strrchr(buf, ':');
    if(p == 0)
        error("pcfile(addr): funny file %s", buf);
    *p = '\0';
    r->string = strnode(buf);
}
```

Uses `TINT` 60a, `TSTRING` 60a, `fileline()` 340b, and `strnode()` 62g.

⟨*function* `pcline` 422b⟩≡ (423b)

```
void
pcline(Node *r, Node *args)
{
    Node res;
    char *p, buf[128];

    if(args == 0)
```

```
        error("pcline(addr): arg count");
    expr(args, &res);
    if(res.type != TINT)
        error("pcline(addr): arg type");

    r->type = TINT;
    r->fmt = 'D';
    if(fileline(buf, sizeof(buf), res.ival) == 0) {
        r->ival = 0;
        return;
    }

    p = strrchr(buf, ':');
    if(p == 0)
        error("pcline(addr): funny file %s", buf);
    r->ival = strtol(p+1, 0, 0);
}
```

Uses TINT 60a and fileline() 340b.

⟨*function* fmtof 423a⟩≡ (423b)

```
void fmtof(Node *r, Node *args)
{
    Node *av[Maxarg];
    Node res;

    na = 0;
    flatten(av, args);
    if(na < 1)
        error("fmtof(obj): no argument");
    if(na > 1)
        error("fmtof(obj): too many arguments") ;
    expr(av[0], &res);

    r->op = OCONST;
    r->type = TINT ;
    r->ival = res.fmt ;
    r->fmt = 'c';
}
```

Uses Maxarg 82a, OCONST 59, TINT 60a, flatten() 419, and na 136a.

⟨acid/builtin.c 423b⟩≡

```
#include <u.h>
#include <libc.h>
#include <bio.h>
#include <ctype.h>
#include <mach.h>
#include <regexp.h>
#include "acid.h"
#include "y.tab.h"

void    cvtatof(Node*, Node*);
void    cvtatoi(Node*, Node*);
void    cvtitoa(Node*, Node*);
void    bprint(Node*, Node*);
void    funcbound(Node*, Node*);
void    printto(Node*, Node*);
void    getfile(Node*, Node*);
void    fmt(Node*, Node*);
void    pcfile(Node*, Node*);
void    pcline(Node*, Node*);
```

```
void    setproc(Node*, Node*);
void    strace(Node*, Node*);
void    follow(Node*, Node*);
void    reason(Node*, Node*);
void    newproc(Node*, Node*);
void    startstop(Node*, Node*);
void    match(Node*, Node*);
void    status(Node*, Node*);
void    kill(Node*,Node*);
void    waitstop(Node*, Node*);
void    stop(Node*, Node*);
void    start(Node*, Node*);
void    filepc(Node*, Node*);
void    doerror(Node*, Node*);
void    rc(Node*, Node*);
void    doaccess(Node*, Node*);
void    map(Node*, Node*);
void    readfile(Node*, Node*);
void    interpret(Node*, Node*);
void    include(Node*, Node*);
void    regexp(Node*, Node*);
void    dosysr1(Node*, Node*);
void    fmtof(Node*, Node*) ;
void    dofmtsize(Node*, Node*) ;

typedef struct Btab Btab;
```
⟨*global* `tab` 68a⟩

⟨*global* `vfmt` 110c⟩

⟨*function* `mkprint` 409a⟩

⟨*function* `installbuiltin` 73a⟩

⟨*function* `dosysr1` 409b⟩

⟨*function* `match` 409c⟩

⟨*function* `newproc` 81c⟩

⟨*function* `startstop` 163a⟩

⟨*function* `waitstop` 163b⟩

⟨*function* `start` 162c⟩

⟨*function* `stop` 162d⟩

⟨*function* `kill` 410a⟩

⟨*function* `status` 163c⟩

⟨*function* `reason` 410b⟩

⟨*function* `follow` 411a⟩

⟨*function* `funcbound` 411b⟩

⟨*function* `setproc` 412a⟩

⟨*function* `filepc` 412b⟩

⟨*function* `interpret` 142c⟩

⟨*function* `include` 142a⟩

⟨*function* `rc` 413a⟩

⟨*function* `doerror` 414a⟩

⟨*function* `doaccess` 414b⟩

⟨*function* `readfile` 414c⟩

⟨*function* *getfile (acid/builtin.c)* 415⟩

⟨*function* `cvtatof` 416a⟩

⟨*function* `cvtatoi` 416b⟩

⟨*global* `fmtflags` 416c⟩
⟨*global* `fmtverbs` 416d⟩

⟨*function* `acidfmt` 416e⟩

⟨*function* `cvtitoa` 417⟩

⟨*function* `mapent` 418a⟩

⟨*function* `map` 418b⟩

⟨*function* `flatten` 419⟩

⟨*function* `strace` 420a⟩

⟨*function* `regerror` 420b⟩

⟨*function* `regexp` 421a⟩

⟨*function* `fmt` 147b⟩

⟨*function* `patom` 145b⟩

⟨*function* `blprint` 144e⟩

⟨*function* `comx` 145a⟩

⟨*function* `bprint` 144d⟩

⟨*function* `printto` 421b⟩

⟨*function* `pcfile` 422a⟩

⟨*function* `pcline` 422b⟩

⟨*function* `fmtof` 423a⟩

⟨*function* `dofmtsize` 148b⟩

Uses `Btab` 68a.

F.5 lib/acid/

F.5.1 lib/acid/port.acid

F.5.2 lib/acid/arm.acid

F.5.3 lib/acid/leak.acid

F.5.4 lib/acid/coverage.acid

F.5.5 lib/acid/truss.acid

F.5.6 lib/acid/syscall.acid

F.6 db/

F.6.1 db/defs.h

⟨*constant* TRUE 426a⟩≡ (426c)
```
#define TRUE    (-1)
```

⟨*constant* FALSE 426b⟩≡ (426c)
```
#define FALSE   0
```

⟨db/defs.h 426c⟩≡
```
/*
 * adb - common definitions
 * something of a grab-bag
 */

#include <u.h>
#include <libc.h>
#include <bio.h>
#include <ctype.h>

#include <mach.h>

⟨type ADDR 242d⟩
⟨type WORD 242f⟩

typedef struct bkpt     BKPT;

⟨constant HUGEINT (db) 294d⟩

⟨constant MAXOFF 266d⟩
⟨constant INCDIR 440a⟩
⟨constant DBNAME 297f⟩
⟨constant CMD_VERBS 240⟩

⟨constant MAXPOS 270b⟩
⟨constant MAXLIN 436d⟩
⟨constant ARB 268a⟩
⟨constant MAXCOM 294b⟩
⟨constant MAXARG 284d⟩
⟨constant LINSIZ 255e⟩
⟨constant MAXSYM 437a⟩

⟨constant EOR 251g⟩
```

⟨*constant* SPC 251h⟩
⟨*constant* TB 251i⟩

⟨*constant* TRUE 426a⟩
⟨*constant* FALSE 426b⟩

```
/*
 * run modes
 */
```
⟨*constant* SINGLE 281c⟩
⟨*constant* CONTIN 281d⟩

```
/*
 * breakpoints
 */
```
⟨*constant* BKPTCLR 243c⟩
⟨*constant* BKPTSET 243d⟩
⟨*constant* BKPTSKIP 243e⟩
⟨*constant* BKPTTMP 243f⟩

⟨*struct* bkpt 243b⟩

```
/*
 * common globals
 */

extern  ADDR    dot;
extern  int     dotinc;
extern ADDR     ditto;

extern  int     adrflg;
extern  WORD    adrval;

extern  int     cntflg;
extern  WORD    cntval;
extern  WORD    loopcnt;

extern  uvlong  expv;

extern  ADDR    maxoff;

extern  int     pid;
extern  char    *corfil, *symfil;
extern  int     fcor, fsym;
extern  Map     *cormap, *symmap, *dotmap;
extern  int     pcsactive;
extern  int     ending;
extern  bool    mkfault;

extern  BKPT    *bkpthead;

extern  int     lastc, peekc;
```

⟨*constant* NNOTE 244e⟩
```
extern  int     nnote;
extern  char    note[NNOTE][ERRMAX];

extern  int     xargc;

extern  bool    wtflag;
```

```
extern  bool    kflag;

// new decl, was in main.c before
extern char *errmsg;
extern jmp_buf env;
```

Uses bkpt 243b.

F.6.2 db/fns.h

⟨db/fns.h 428⟩≡

```
void            acommand(int);
void            attachprocess(void);
void            bkput(BKPT*, bool);
void            bpwait(void);
int             charpos(void);
void            chkerr(void);
void            clrinp(void);
void            cmdmap(Map*);
void            cmdsrc(int, Map*);
void            cmdwrite(int, Map*);
int             command(char*, int);
int             convdig(int);
void            ctrace(int);
WORD            defval(WORD);
void            delbp(void);
void            done(void);
int             dprint(char*, ...);
Map*            dumbmap(int);
void            endline(void);
void            endpcs(void);
int             eol(int);
void            error(char*);
void            errors(char*, char*);
void            execbkpt(BKPT*, int);
char*           exform(int, int, char*, Map*, int, int);
int             expr(int);
void            flush(void);
void            flushbuf(void);
char*           getfname(void);
void            getformat(char*);
int             getnum(int (*)(void));
void            grab(void);
void            iclose(int, int);
ADDR            inkdot(int);
int             isfileref(void);
int             item(int);
void            killpcs(void);
void            kmsys(void);
void            main(int, char**);
int             mapimage(void);
void            newline(void);
int             nextchar(void);
void            notes(void);
void            oclose(void);
void            outputinit(void);
void            printc(int);
void            printesc(int);
void            printlocals(Symbol *, ADDR);
```

```
void            printmap(char*, Map*);
void            printparams(Symbol *, ADDR);
void            printpc(void);
void            printregs(int);
void            prints(char*);
void            printsource(ADDR);
void            printsym(void);
void            printsyscall(void);
void            printtrace(int);
int             quotchar(void);
int             rdc(void);
int             readchar(void);
void            readsym(char*);
void            redirin(int, char*);
void            redirout(char*);
void            readfname(char *);
void            reread(void);
char*           regname(int);
uvlong          rget(Map*, char*);
Reglist*        rname(char*);
void            rput(Map*, char*, vlong);
int             runpcs(int, int);
void            runrun(int);
void            runstep(uvlong, int);
BKPT*           scanbkpt(ADDR adr);
void            scanform(long, int, char*, Map*, int);
void            setbp(void);
void            setcor(void);
void            setsym(void);
void            setup(void);
void            setvec(void);
void            shell(void);
void            startpcs(void);
void            subpcs(int);
int             symchar(int);
int             term(int);
void            ungrab(void);
int             valpr(long, int);

#pragma varargck        argpos  dprint  1
#pragma varargck        type    "t"     void
```

F.6.3 db/utils.c

⟨db/utils.c 429a⟩≡

```
#include "defs.h"
#include "fns.h"
```

⟨*function* error 302c⟩

⟨*function* errors 303a⟩

F.6.4 db/globals.c

⟨db/globals.c 429b⟩≡

```
#include "defs.h"
```

⟨*global* wtflag 248e⟩

⟨*global* `kflag` 299a⟩

⟨*global* `mkfault` 300c⟩
⟨*global* `maxoff` 266e⟩

⟨*global* `errmsg` 303b⟩
⟨*global* `env` 246d⟩

⟨*global* `symmap` 241a⟩
⟨*global* `cormap` 241b⟩

⟨*global* `ending` 287c⟩
⟨*global* `pid` 246f⟩
⟨*global* `nnote` 244f⟩
⟨*global* `note` 244d⟩
⟨*global* `bkpthead` 243g⟩

⟨*global* `dot` 242e⟩
⟨*global* `dotinc` 271b⟩
⟨*global* `ditto` 261a⟩

⟨*global* `adrval` 253e⟩
⟨*global* `adrflg` 253d⟩

⟨*global* `cntval` 243a⟩
⟨*global* `cntflg` 253g⟩

⟨*global* `expv` 257a⟩

⟨*global* `pcsactive` 247a⟩

⟨*global* `xargc` 298c⟩

F.6.5 db/output.c

⟨`db/output.c` 430⟩≡

```
/*
 *
 *      debugger
 *
 */

#include "defs.h"
#include "fns.h"
```

⟨*global* `printcol` 251b⟩
⟨*global* `infile` 245c⟩
⟨*global* `maxpos` 270c⟩

⟨*global* `stdout` 250b⟩

⟨*function* `printc` 250f⟩

⟨*function* `tconv` 250d⟩

⟨*function* `flushbuf` 251e⟩

⟨*function* `prints` 251a⟩

⟨*function* `newline` 279a⟩

⟨*constant* `MAXIFD` 295d⟩
⟨*global* `istack` 295e⟩
⟨*global* `ifiledepth` 295f⟩

⟨*function* `iclose` 295g⟩

⟨*function* `oclose` 303d⟩

⟨*function* `redirout` 296a⟩

⟨*function* `endline` 270a⟩

⟨*function* `flush` 251f⟩

⟨*function* `dprint` 250e⟩

⟨*function* `outputinit` 250c⟩

F.6.6 db/input.c

⟨*function* `isfileref` 431a⟩≡ (431b)

```
/*
 *      check if the input line if of the form:
 *              <filename>:<digits><verb> ...
 *
 *      we handle this case specially because we have to look ahead
 *      at the token after the colon to decide if it is a file reference
 *      or a colon-command with a symbol name prefix.
 */

int
isfileref(void)
{
    Rune *cp;

    for (cp = lp-1; *cp && !strchr(CMD_VERBS, *cp); cp++)
        if (*cp == '\\' && cp[1])       /* escape next char */
            cp++;
    if (*cp && cp > lp-1) {
        while (*cp == ' ' || *cp == '\t')
            cp++;
        if (*cp++ == ':') {
            while (*cp == ' ' || *cp == '\t')
                cp++;
            if (isdigit(*cp))
                return 1;
        }
    }
    return 0;
}
```

Uses `CMD_VERBS` 240 and `lp` 251j.

⟨`db/input.c` 431b⟩≡

```
/*
 *
 *      debugger
 *
```

```
 */

#include "defs.h"
#include "fns.h"

extern  int     infile;

⟨global line 255d⟩
⟨global lp 251j⟩
⟨global peekc 252a⟩
⟨global lastc 252b⟩
⟨global eof 254g⟩

/* input routines */

⟨function eol 254d⟩

⟨function rdc 252d⟩

⟨function reread 252e⟩

⟨function clrinp 252c⟩

⟨function readrune 256b⟩

⟨function readchar 255c⟩

⟨function nextchar 256c⟩

⟨function quotchar 435c⟩

⟨function getformat 268f⟩

⟨function isfileref 431a⟩
```

F.6.7 db/setup.c

⟨db/setup.c 432⟩≡

```
/*
 * init routines
 */
#include "defs.h"
#include "fns.h"

⟨global symfil 245a⟩
⟨global corfil 245b⟩

⟨global dotmap 268d⟩

⟨global fsym 247d⟩
⟨global fcor 249b⟩
⟨global fhdr (db/setup.c) 247c⟩

static int getfile(char*, int, int);

⟨function setsym (db) 247e⟩

⟨function setcor 249c⟩
```

```
extern Mach mi386;
extern Machdata i386mach;
```

⟨*function* dumbmap 248b⟩

⟨*function* cmdmap 276g⟩

⟨*function* getfile 248f⟩

⟨*function* kmsys 299f⟩

⟨*function* attachprocess 266b⟩

F.6.8 db/format.c

⟨db/format.c 433a⟩≡

```
/*
 *
 *      debugger
 *
 */

#include "defs.h"
#include "fns.h"
```

⟨*function* scanform 269a⟩

⟨*function* exform 269b⟩

⟨*function* printesc 273b⟩

⟨*function* inkdot 254e⟩

F.6.9 db/regs.c

⟨*function* rname 433b⟩≡ (435a)

```
/*
 * translate a name to a magic register offset
 */
Reglist*
rname(char *name)
{
    Reglist *rp;

    for (rp = mach->reglist; rp->rname; rp++)
        if (strcmp(name, rp->rname) == 0)
            return rp;
    return 0;
}
```

Uses mach 51a.

⟨*function* getreg 433c⟩≡ (435a)

```
static uvlong
getreg(Map *map, Reglist *rp)
{
    uvlong v;
    ulong w;
```

```
    ushort s;
    int ret;

    v = 0;
    ret = 0;
    switch (rp->rformat)
    {
    case 'x':
        ret = get2(map, rp->roffs, &s);
        v = s;
        break;
    case 'f':
    case 'X':
        ret = get4(map, rp->roffs, &w);
        v = w;
        break;
    case 'F':
    case 'W':
    case 'Y':
        ret = get8(map, rp->roffs, &v);
        break;
    default:
        werrstr("can't retrieve register %s", rp->rname);
        error("%r");
    }
    if (ret < 0) {
        werrstr("Register %s: %r", rp->rname);
        error("%r");
    }
    return v;
}
```

Uses `get2()` 350c, `get4()` 350b, and `get8()` 350a.

⟨*function* `rget` 434a⟩≡ (435a)

```
uvlong
rget(Map *map, char *name)
{
    Reglist *rp;

    rp = rname(name);
    if (!rp)
        error("invalid register name");
    return getreg(map, rp);
}
```

Uses `getreg()` 433c and `rname()` 433b.

⟨*function* `rput` 434b⟩≡ (435a)

```
void
rput(Map *map, char *name, vlong v)
{
    Reglist *rp;
    int ret;

    rp = rname(name);
    if (!rp)
        error("invalid register name");
    if (rp->rflags & RRDONLY)
        error("register is read-only");
    switch (rp->rformat)
    {
```

```
        case 'x':
            ret = put2(map, rp->roffs, (ushort) v);
            break;
        case 'X':
        case 'f':
        case 'F':
            ret = put4(map, rp->roffs, (long) v);
            break;
        case 'Y':
            ret = put8(map, rp->roffs, v);
            break;
        default:
            ret = -1;
        }
        if (ret < 0)
            error("can't write register");
    }
```

Uses put2() 352b, put4() 352a, put8() 351c, and rname() 433b.

⟨db/regs.c 435a⟩≡

```
    /*
     * code to keep track of registers
     */

    #include "defs.h"
    #include "fns.h"

    ⟨function rname 433b⟩

    ⟨function getreg 433c⟩

    ⟨function rget 434a⟩

    ⟨function rput 434b⟩
    ⟨function printregs 264c⟩
```

F.6.10 db/expr.c

⟨function ascval 435b⟩≡ (439a)

```
    static WORD
    ascval(void)
    {
        Rune r;

        if (readchar() == 0)
            return (0);
        r = lastc;
        while(quotchar())   /*discard chars to ending quote */
            ;
        return((WORD) r);
    }
```

Uses lastc 252b, quotchar() 435c, and readchar() 255c.

⟨function quotchar 435c⟩≡ (431b)

```
    int
    quotchar(void)
    {
        if (readchar()=='\\')
```

```
        return(readchar());
    else if (lastc=='\'')
        return 0;
    else
        return lastc;
}
```
Uses lastc 252b and readchar() 255c.

⟨*struct* fpin_union 436a⟩≡ (439a)
```
union fpin_union {
        WORD w;
        float f;
};
```

⟨*function* fpin 436b⟩≡ (439a)
```
/*
 * read a floating point number
 * the result must fit in a WORD
 */

static WORD
fpin(char *buf)
{
  union fpin_union x;

    x.f = atof(buf);
    return (x.w);
}
```
Uses fpin_union 436a.

⟨*constant* MAXBASE 436c⟩≡ (439a)
```
#define MAXBASE 16
```

⟨*constant* MAXLIN 436d⟩≡ (426c)
```
#define MAXLIN  128
```

⟨*function* getnum 436e⟩≡ (439a)
```
/* service routines for expression reading */
int
getnum(int (*rdf)(void))
{
    char *cp;
    int base, d;
    bool fpnum;
    char num[MAXLIN];

    base = 0;
    fpnum = FALSE;
    if (lastc == '#') {
        base = 16;
        (*rdf)();
    }
    if (convdig(lastc) >= MAXBASE)
        return (0);
    if (lastc == '0')
        switch ((*rdf)()) {
        case 'x':
        case 'X':
            base = 16;
            (*rdf)();
```

```
            break;

        case 't':
        case 'T':
            base = 10;
            (*rdf)();
            break;

        case 'o':
        case 'O':
            base = 8;
            (*rdf)();
            break;
        default:
            if (base == 0)
                base = 8;
            break;
        }
    if (base == 0)
        base = 10;
    expv = 0;
    for (cp = num, *cp = lastc; ;(*rdf)()) {
        if ((d = convdig(lastc)) < base) {
            expv *= base;
            expv += d;
            *cp++ = lastc;
        }
        else if (lastc == '.') {
            fpnum = TRUE;
            *cp++ = lastc;
        } else {
            reread();
            break;
        }
    }
    if (fpnum)
        expv = fpin(num);
    return (1);
}
```

Uses MAXBASE-70 436c, MAXLIN 436d, convdig() 438b, expv 257a, fpin() 436b, lastc 252b, and reread() 252e.

⟨*constant* MAXSYM 437a⟩≡ (426c)

```
#define MAXSYM  255
```

⟨*function* readsym 437b⟩≡ (439a)

```
void
readsym(char *isymbol)
{
    char        *p;
    Rune r;

    p = isymbol;
    do {
        if (p < &isymbol[MAXSYM-UTFmax-1]){
            r = lastc;
            p += runetochar(p, &r);
        }
        readchar();
    } while (symchar(1));
    *p = 0;
```

```
}
```
Uses MAXSYM 437a, lastc 252b, readchar() 255c, and symchar() 438c.

⟨*function* readfname 438a⟩≡ (439a)
```
void
readfname(char *filename)
{
    char        *p;
    Rune        c;

    /* snarf chars until un-escaped char in terminal char set */
    p = filename;
    do {
        if ((c = lastc) != '\\' && p < &filename[MAXSYM-UTFmax-1])
            p += runetochar(p, &c);
        readchar();
    } while (c == '\\' || strchr(CMD_VERBS, lastc) == 0);
    *p = 0;
    reread();
}
```
Uses CMD_VERBS 240, MAXSYM 437a, lastc 252b, readchar() 255c, and reread() 252e.

⟨*function* convdig 438b⟩≡ (439a)
```
int
convdig(int c)
{
    if (isdigit(c))
        return(c-'0');
    else if (!isxdigit(c))
        return(MAXBASE);
    else if (isupper(c))
        return(c-'A'+10);
    else
        return(c-'a'+10);
}
```
Uses MAXBASE-70 436c.

⟨*function* symchar 438c⟩≡ (439a)
```
int
symchar(int dig)
{
    if (lastc=='\\') {
        readchar();
        return(TRUE);
    }
    return(isalpha(lastc) || lastc>0x80 || lastc=='_' || dig && isdigit(lastc));
}
```
Uses lastc 252b and readchar() 255c.

⟨*function* round 438d⟩≡ (439a)
```
static long
round(long a, long b)
{
    long w;

    w = (a/b)*b;
    if (a!=w)
        w += b;
    return(w);
}
```

⟨db/expr.c 439a⟩≡
```
/*
 *
 *      debugger
 *
 */

#include "defs.h"
#include "fns.h"

static long     round(long, long);

extern  ADDR    ditto;
```

⟨*function* ascval 435b⟩

⟨*struct* fpin_union 436a⟩

⟨*function* fpin 436b⟩

⟨*function* defval 286b⟩

⟨*function* expr 257b⟩

⟨*function* term 258⟩

⟨*function* item 259⟩

⟨*constant* MAXBASE 436c⟩

⟨*function* getnum 436e⟩

⟨*function* readsym 437b⟩

⟨*function* readfname 438a⟩

⟨*function* convdig 438b⟩

⟨*function* symchar 438c⟩

⟨*function* round 438d⟩

F.6.11 db/trcrun.c

⟨db/trcrun.c 439b⟩≡
```
/*
 * functions for running the debugged process
 */

#include "defs.h"
#include "fns.h"
```

⟨*global* child 284c⟩
⟨*global* msgfd 241c⟩
⟨*global* notefd 244a⟩
⟨*global* pcspid 241d⟩

⟨*function* `setpcs` 241e⟩

⟨*function* `msgpcs` 242a⟩

⟨*function* `unloadnote` 302a⟩

⟨*function* `loadnote` 301d⟩

⟨*function* `notes` 301e⟩

⟨*function* `killpcs` 288a⟩

⟨*function* `grab` 288d⟩

⟨*function* `ungrab` 289a⟩

⟨*function* `doexec` 284e⟩

⟨*global* `procname` 284a⟩

⟨*function* `startpcs` 283c⟩

⟨*function* `runstep` 286c⟩

⟨*function* `bpwait` 284b⟩

⟨*function* `runrun` 285b⟩

⟨*function* `bkput` 291b⟩

F.6.12 db/print.c

⟨*constant* `INCDIR` 440a⟩≡ (426c)

```
#define INCDIR  "/usr/lib/adb"
```

⟨*global* `Ipath` 440b⟩≡ (441b)

```
char    *Ipath = INCDIR;
```

Uses `INCDIR` 440a and `Ipath` 440b.

⟨*function* `getfname` 440c⟩≡ (441b)

```
char *
getfname(void)
{
    static char fname[ARB];
    char *p;

    if (rdc() == EOR) {
        reread();
        return (0);
    }
    p = fname;
    do {
        *p++ = lastc;
        if (p >= &fname[ARB-1])
            error("filename too long");
    } while (rdc() != EOR);
    *p = 0;
    reread();
    return (fname);
```

```
}
```
Uses ARB 268a, EOR 251g, lastc 252b, rdc() 252d, and reread() 252e.

⟨*function* redirin 441a⟩≡ (441b)
```
void
redirin(int stack, char *file)
{
    char *pfile;

    if (file == 0) {
        iclose(-1, 0);
        return;
    }
    iclose(stack, 0);
    if ((infile = open(file, 0)) < 0) {
        pfile = smprint("%s/%s", Ipath, file);
        infile = open(pfile, 0);
        free(pfile);
        if(infile < 0) {
            infile = STDIN;
            error("cannot open");
        }
    }
}
```
Uses Ipath 440b, iclose() 295g, and infile 245c.

⟨db/print.c 441b⟩≡
```
/*
 *
 *      debugger
 *
 */
#include "defs.h"
#include "fns.h"

extern  int     infile;
extern  int     outfile;
extern  int     maxpos;

/* general printing routines ($) */

⟨global Ipath 440b⟩
⟨global tracetype 263c⟩
static void     printfp(Map*, int);

⟨function ptrace 264a⟩

⟨function printtrace 261c⟩

⟨function getfname 440c⟩

⟨function printfp 265b⟩

⟨function redirin 441a⟩

⟨function printmap 262b⟩

⟨function printsym 262d⟩

⟨constant STRINGSZ 281a⟩
```

⟨*function* `printsource` 281b⟩

⟨*function* `printpc` 280b⟩

⟨*function* `printlocals` 294a⟩

⟨*function* `printparams` 293b⟩

F.6.13 db/command.c

⟨*global* `BADEQ` 442a⟩≡ (442c)
```
char    BADEQ[] = "unexpected `='";
```
Uses `BADEQ` 442a.

⟨*function* `regname` 442b⟩≡ (442c)
```
/*
 * collect a register name; return register offset
 * this is not what i'd call a good division of labour
 */

char *
regname(int regnam)
{
    static char buf[64];
    char *p;
    int c;

    p = buf;
    *p++ = regnam;
    while (isalnum(c = readchar())) {
        if (p >= buf+sizeof(buf)-1)
            error("register name too long");
        *p++ = c;
    }
    *p = 0;
    reread();
    return (buf);
}
```
Uses `readchar()` 255c and `reread()` 252e.

⟨`db/command.c` 442c⟩≡
```
/*
 *
 *      debugger
 *
 */

#include "defs.h"
#include "fns.h"

⟨global BADEQ 442a⟩

⟨global executing 279c⟩
extern  Rune    *lp;

⟨global eqformat 268b⟩
⟨global stformat 268c⟩
```

⟨*global* `loopcnt` 280a⟩

⟨*function* `command` 253a⟩

⟨*function* `acommand` 267c⟩

⟨*function* `cmdsrc` 277b⟩

⟨*global* `badwrite` 278b⟩

⟨*function* `cmdwrite` 278c⟩

⟨*function* `regname` 442b⟩

⟨*function* `shell` 297h⟩

F.6.14 db/runpcs.c

⟨`db/runpcs.c` 443a⟩≡

```
/*
 *
 *      debugger
 *
 */

#include "defs.h"
#include "fns.h"
```

⟨*global* `bpin` 288b⟩

⟨*function* `runpcs` 281e⟩

⟨*function* `endpcs` 287d⟩

⟨*function* `setup` 283b⟩

⟨*function* `execbkpt` 292a⟩

⟨*function* `scanbkpt` 289b⟩

⟨*function* `delbp` 291c⟩

⟨*function* `setbp` 291a⟩

F.6.15 db/pcs.c

⟨`db/pcs.c` 443b⟩≡

```
/*
 *
 *      debugger
 *
 */

#include "defs.h"
#include "fns.h"
```

⟨*global* NOPCS 287a⟩

⟨*function* subpcs 279e⟩

F.6.16 db/main.c

⟨db/main.c 444⟩≡

```
/*
 * db - main command loop and error/interrupt handling
 */
#include "defs.h"
#include "fns.h"


extern  bool    executing;
extern  int     infile;
extern  int     eof;

int     alldigs(char*);
void    fault(void*, char*);

extern  char    *Ipath;
extern jmp_buf env;
extern char *errmsg;
```

⟨*function main (db/main.c)* 245d⟩

⟨*function* alldigs 246a⟩

⟨*function* done 255b⟩

⟨*function* fault 300b⟩

Glossary

Indexes

Here is a list of the identifiers used, and where they appear. Underlined entries indicate the place of definition. This index is generated automatically.

Bibliography

[Knu92] Donald E. Knuth. *Literate Programming*. Center for the Study of Language and Information, 1992. cited page(s) 14

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988. cited page(s) 13

[Pad09] Yoann Padioleau. Syncweb, literate programming meets unison, 2009. https://github.com/aryx/syncweb. cited page(s) 14

[Pad14] Yoann Padioleau. *Principia Softwarica: The Plan 9 Kernel 9*. 2014. cited page(s) 13, 193

[Pad15a] Yoann Padioleau. *Principia Softwarica: The ARM Assembler 5a*. 2015. cited page(s) 22, 47, 99, 193, 210

[Pad15b] Yoann Padioleau. *Principia Softwarica: The ARM Emulator 5i*. 2015. cited page(s) 18

[Pad15c] Yoann Padioleau. *Principia Softwarica: The ARM Linker 5l*. 2015. cited page(s) 13, 22, 47, 193

[Pad16a] Yoann Padioleau. *Principia Softwarica: The Plan 9 C Compiler 5c*. 2016. cited page(s) 13, 22, 47, 99, 193

[Pad16b] Yoann Padioleau. *Principia Softwarica: The Plan 9 Core Libraries*. 2016. cited page(s) 37

[Pad18] Yoann Padioleau. *Principia Softwarica: The Plan 9 Shell rc*. 2018. cited page(s) 99

[Ros96] Jonathan B. Rosenberg. *How Debuggers Work: Algorithms, Data Structures, and Architecture*. John Wiley & Sons, 1996. cited page(s) 13