Principia Softwarica: The Plan 9 Debuggers and Tracers

version 0.1

Yoann Padioleau
yoann.padioleau@gmail.com

with code from
Phil Winterbottom

January 14, 2026

Contents

Chapter 1

Introduction

The goal of this book is to present in 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 some of the best tools to understand how things work.

1.2 Plan 9 `db` and `strace`

1.3 Other debuggers

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

- gdb
- lldb
- valgrind
- ocamldebug
- Linux strace
- Solaris dtrace

1.4 Getting started

1.5 Requirements

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

Chapter 2

Overview

2.1 Tracer principles

2.2 `strace` services

2.3 Debugger principles

2.4 `db` services

⟨`main()` *print usage and exit (db)* 9a⟩≡ (18d)
```
fprint(2, "Usage: db [-kw] [-m machine] [-I dir] ([symfile]|[pid])\n");
exits("usage");
```

2.5 A simple debugging session

⟨*constant* `CMD_VERBS` 9b⟩≡ (85c)
```
#define CMD_VERBS "?/=>!$: \t"
```

2.6 `acid` services

2.7 Code organization

2.8 Software architecture

2.9 Book structure

Chapter 3

Kernel Support

3.1 Memory access: `/proc/x/mem`

3.2 Process control: `/proc/x/ctl`

3.3 Breakpoint faulting instruction

3.4 User level fault handling: `/proc/x/note`

3.5 Core dumps and broken processes

Chapter 4

Core Data Structures

4.1 Executable format: Fhdr

⟨*struct* Fhdr 11a⟩≡ (114b)

```
/*
 * 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 11b⟩≡ (114b)

```
/* types of executables */
enum executable_type
{
```

```
    FNONE = 0,  /* unidentified */

    FI386,   /* 8.out */
    FI386B,   /* I386 bootable */
    FARM,   /* 5.out */
    FARMB,   /* ARM bootable */
};
```

4.2 Mach and mach

⟨*struct* Mach 12a⟩≡ (114b)

```
/*
 * 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{
    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) */
};
```

⟨*enum* machine_type 12b⟩≡ (114b)

```
/* machine types */
enum machine_type
{
    MI386,
    MARM,
};
```

⟨*struct* Reglist 13a⟩≡ (114b)
```
/*
 * 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' */
};
```

⟨*enum* register_flag 13b⟩≡ (114b)
```
enum {     /* bits in rflags field */
    RINT = (0<<0),
    RFLT = (1<<0),

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

⟨*global* mach 13c⟩≡ (154b)
```
Mach *mach = &mi386;   /* Global current machine table */
```

4.3 Machdata and machdata

⟨*struct* Machdata 13d⟩≡ (114b)
```
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 */
};
```

4.4 Map

⟨*struct* Map 13e⟩≡ (114b)
```
/*
 *  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 */
        int 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 */
};
```

4.5 Symbol

⟨*struct* Symbol 14a⟩≡ (114b)

```
/*
 * 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 14b⟩≡ (114b)

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

    CAUTO,
    CPARAM,
    CSTAB,
    CTEXT,
    CDATA,

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

4.6 /proc/x/text and symmap

⟨*global* symmap 14c⟩≡ (88b)

```
// was in setup.c
Map *symmap;
```

4.7 /proc/x/mem and cormap

⟨*global* cormap 15a⟩≡ (88b)
```
Map *cormap;
```

4.8 /proc/x/ctl, msgfd, and msgpcs()

⟨*global* msgfd 15b⟩≡ (98)
```
fdt msgfd = -1;
```

⟨*global* pcspid 15c⟩≡ (98)
```
int pcspid = -1;
```

⟨*function* setpcs 15d⟩≡ (98)
```
void
setpcs(void)
{
    char buf[128];

    if(pid && pid != pcspid){
        if(msgfd >= 0){
            close(msgfd);
            msgfd = -1;
        }
        ⟨setpcs() close previous notefd if changed process 17e⟩

        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 17f⟩

        pcspid = pid;
    }
}
```

⟨*function* msgpcs 15e⟩≡ (98)
```
void
msgpcs(char *msg)
{
    int ret;
    ⟨msgpcs() locals 15f⟩

    setpcs();
    //dprint("--> %d: %s\n", pcspid, msg);
    ret = write(msgfd, msg, strlen(msg));
    ⟨msgpcs() error managment 16a⟩
}
```

⟨msgpcs() *locals* 15f⟩≡ (15e)
```
char err[ERRMAX];
```

⟨msgpcs() *error managment* 16a⟩≡ (15e)
```
if(ret < 0 && !ending){
    errstr(err, sizeof err);
    if(strcmp(err, "interrupted") != 0)
        endpcs();
    errors("can't write control file", err);
}
```

4.9 Addr and dot

⟨*type* ADDR 16b⟩≡ (85c)
```
typedef uvlong ADDR;
```

⟨*global* dot 16c⟩≡ (88b)
```
// was in command.c
ADDR dot;
```

4.10 cntval

⟨*type* WORD 16d⟩≡ (85c)
```
typedef ulong WORD;
```

⟨*global* cntval 16e⟩≡ (88b)
```
WORD cntval;
```

4.11 Bkpt and bkpthead

⟨*struct* bkpt 16f⟩≡ (85c)
```
struct bkpt {
    // address to break on
    ADDR loc;
    // enum<breakpoint_kind>
    int flag;

    // original code at the breapoint
    byte save[4];

    ⟨Bkpt other fields 17c⟩

    // Extra
    ⟨Bkpt extra fields 17b⟩
};
```

⟨*constant* BKPTCLR 16g⟩≡ (85c)
```
#define BKPTCLR 0 /* not a real breakpoint */
```

⟨*constant* BKPTSET 16h⟩≡ (85c)
```
#define BKPTSET 1 /* real, ready to trap */
```

⟨*constant* BKPTSKIP 16i⟩≡ (85c)
```
#define BKPTSKIP 2 /* real, skip over it next time */
```

⟨*constant* BKPTTMP 16j⟩≡ (85c)
```
#define BKPTTMP 3 /* temporary; clear when it happens */
```

⟨*global* `bkpthead` 17a⟩≡ (88b)
```
// list<BKPT>, next = BKPT.next
BKPT *bkpthead;
```

⟨`Bkpt` *extra fields* 17b⟩≡ (16f)
```
// list<ref_own<BKPT>, head = bkpthead
BKPT *nxtbkpt;
```

⟨`Bkpt` *other fields* 17c⟩≡ (16f) 71b ▷
```
int count;
int initcnt;
```

4.12 /proc/x/note, notefd, and notes

⟨*global* `notefd` 17d⟩≡ (98)
```
fdt notefd = -1;
```

⟨`setpcs()` *close previous notefd if changed process* 17e⟩≡ (15d)
```
if(notefd >= 0){
    close(notefd);
    notefd = -1;
}
```

⟨`setpcs()` *open notefd for new process* 17f⟩≡ (15d)
```
sprint(buf, "/proc/%d/note", pid);
notefd = open(buf, ORDWR);
if(notefd < 0)
    error("can't open note file");
```

⟨*global* `note` 17g⟩≡ (88b)
```
// array<option<ref_own<string>>, actual number of elts used = nnote
char note[NNOTE][ERRMAX];
```

⟨*constant* `NNOTE` 17h⟩≡ (85c)
```
#define NNOTE 10
```

⟨*global* `nnote` 17i⟩≡ (88b)
```
int nnote;
```

Chapter 5

main()

⟨*global* symfil 18a⟩≡ (91)
```
char *symfil = nil;
```

⟨*global* corfil 18b⟩≡ (91)
```
char *corfil = nil;
```

⟨*global* infile 18c⟩≡ (89)
```
int infile = STDIN;
```

⟨*function main (db/main.c)* 18d⟩≡ (102c)
```
void
main(int argc, char **argv)
{
    ⟨main() locals (db) 19g⟩

    outputinit();

    ARGBEGIN{
    ⟨main() command line processing (db) 73e⟩
    }ARGEND

    if (argc > 0 && !alldigs(argv[0])) {
        symfil = argv[0];
        argv++;
        argc--;
    }
    ⟨main() if pid argument, attach to existing process 20a⟩
    else if (argc > 0) {
        ⟨main() print usage and exit (db) 9a⟩
    }

   if (!symfil)
        symfil = "8.out";

    ⟨main() initialization before repl (db) 19b⟩

    // repl loop
    for (;;) {
        // clear output
        flushbuf();

        ⟨main() in loop, handle errmsg (db) 82d⟩
        ⟨main() in loop, handle mkfault (db) 78b⟩

        // clear input
```

```
        clrinp();

        // go to next non whitespace char
        rdc();
        reread();
        ⟨main() in loop, if eof (db) 27c⟩

        command(nil, 0);

        reread();
        if (rdc() != '\n')
            error("newline expected");
    }
}
```

⟨*function* alldigs 19a⟩≡ (102c)
```
bool
alldigs(char *s)
{
    while(*s){
        if(*s<'0' || '9'<*s)
            return false;
        s++;
    }
    return true;
}
```

⟨main() *initialization before repl (db)* 19b⟩≡ (18d) 75c▷
```
⟨main() call notify 77d⟩
⟨main() call setsym 20c⟩
⟨main() set dotmap 42f⟩
⟨main() print binary architecture 19c⟩
⟨main() setjmp 19e⟩
⟨main() just before repl 53c⟩
```

⟨main() *print binary architecture* 19c⟩≡ (19b)
```
⟨main() if db -m and unknown machine 76c⟩
dprint("%s binary\n", mach->name);
```

⟨*global* env 19d⟩≡ (88b)
```
jmp_buf env;
```

⟨main() *setjmp* 19e⟩≡ (19b)
```
if(setjmp(env) == 0){
    ⟨main() if setjmp == 0 and corfil 22b⟩
}
setjmp(env);
```

5.1 Process attachment: pid

⟨*global* pid 19f⟩≡ (88b)
```
int pid;
```

⟨main() *locals (db)* 19g⟩≡ (18d) 73d▷
```
char b1[100];
char b2[100];
```

⟨main() *if pid argument, attach to existing process* 20a⟩≡ (18d)
```
if(argc==1 && alldigs(argv[0])){

    pid = atoi(argv[0]);
    pcsactive = false;
    if (!symfil) {
        ⟨main() when pid argument, if kflag 77a⟩
        else
            sprint(b1, "/proc/%s/text", argv[0]);
        symfil = b1;
    }
    sprint(b2, "/proc/%s/mem", argv[0]);
    corfil = b2;
}
```

⟨*global* pcsactive 20b⟩≡ (88b)
```
// was in trcrun.c
bool pcsactive = false;
```

5.2 Text file: setsym()

⟨main() *call setsym* 20c⟩≡ (19b)
```
setsym();
```

⟨*global fhdr (db/setup.c)* 20d⟩≡ (91)
```
static Fhdr fhdr;
```

⟨*global* fsym 20e⟩≡ (91)
```
fdt fsym;
```

⟨*function setsym (db)* 20f⟩≡ (91)
```
void
setsym(void)
{
    int ret;
    ⟨setsym() locals 21e⟩

    fsym = getfile(symfil, 1, wtflag);
    ⟨setsym() error managment on fsym 20g⟩
    ret = crackhdr(fsym, &fhdr);
    if (ret) {
        machbytype(fhdr.type);
        symmap = loadmap(symmap, fsym, &fhdr);
        ⟨setsym() error managment on symmap 21a⟩
        ret = syminit(fsym, &fhdr);
        ⟨setsym() error managment on syminit 21c⟩

        ⟨setsym() if mach has sbreg 21f⟩
    }
    ⟨setsym() error managment on crackhdr 21b⟩
}
```

⟨setsym() *error managment on fsym* 20g⟩≡ (20f)
```
if(fsym < 0) {
    symmap = dumbmap(-1);
    return;
}
```

⟨setsym() *error managment on symmap* 21a⟩≡ (20f)
```
if (symmap == nil)
    symmap = dumbmap(fsym);
```

⟨setsym() *error managment on crackhdr* 21b⟩≡ (20f)
```
else
    symmap = dumbmap(fsym);
```

⟨setsym() *error managment on syminit* 21c⟩≡ (20f)
```
if (ret < 0)
    dprint("%r\n");
```

⟨*function* dumbmap 21d⟩≡ (91)
```
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;
}
```

⟨setsym() *locals* 21e⟩≡ (20f)
```
Symbol s;
```

⟨setsym() *if mach has sbreg* 21f⟩≡ (20f)
```
if (mach->sbreg && lookup(0, mach->sbreg, &s))
    mach->sb = s.value;
```

⟨*global* wtflag 21g⟩≡ (88b)
```
// was in main.c
int wtflag = OREAD;
```

⟨*function* getfile 21h⟩≡ (91)
```
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 22a⟩
    return f;
}
```

⟨getfile() *error managment* 22a⟩≡ (21h)
```
if(f < 0 && omode == ORDWR){
    f = open(filnam, OREAD|OCEXEC);
    if(f >= 0)
        dprint("%s open read-only\n", filnam);
}
⟨getfile() if wtflag 75a⟩
if (f < 0) {
    dprint("cannot open ‘%s’: %r\n", filnam);
    return ERROR_NEG1;
}
```

5.3 Memory file: `setcor()`

⟨main() *if setjmp == 0 and corfil* 22b⟩≡ (19e)
```
if (corfil) {
    setcor(); /* could get error */
    dprint("%s\n", machdata->excep(cormap, rget));
    printpc();
}
```

⟨*global* fcor 22c⟩≡ (91)
```
fdt fcor;
```

⟨*function* setcor 22d⟩≡ (91)
```
void
setcor(void)
{
    int i;

    ⟨setcor() free previous cormap 22e⟩
    fcor = getfile(corfil, 2, ORDWR);
    ⟨setcor() error managment getfile 22f⟩
    if(pid > 0) { /* provide addressability to executing process */
        cormap = attachproc(pid, kflag, fcor, &fhdr);
        ⟨setcor() error managment cormap 23a⟩
    } else {
        cormap = newmap(cormap, 2);
        ⟨setcor() error managment cormap 23a⟩
        setmap(cormap, fcor, fhdr.txtaddr, fhdr.txtaddr+fhdr.txtsz, fhdr.txtaddr, "text");
        setmap(cormap, fcor, fhdr.dataddr, 0xffffffff, fhdr.dataddr, "data");
    }
    kmsys();
    return;
}
```

⟨setcor() *free previous cormap* 22e⟩≡ (22d)
```
if (cormap) {
    for (i = 0; i < cormap->nsegs; i++)
        if (cormap->seg[i].inuse)
            close(cormap->seg[i].fd);
}
```

⟨setcor() *error managment getfile* 22f⟩≡ (22d)
```
if (fcor <= 0) {
    if (cormap)
        free(cormap);
    cormap = dumbmap(-1);
    return;
}
```

⟨setcor() *error managment cormap* 23a⟩≡ (22)
```
if (!cormap)
    cormap = dumbmap(-1);
```

5.4 Output: outputinit(), dprint() and stdout

⟨*global* stdout 23b⟩≡ (89)
```
Biobuf stdout;
```

⟨*function* outputinit 23c⟩≡ (89)
```
void
outputinit(void)
{
    Binit(&stdout, 1, OWRITE);
    fmtinstall('t', tconv);
}
```

⟨*function* tconv 23d⟩≡ (89)
```
/* was move to next f1-sized tab stop; now just print a tab */
int
tconv(Fmt *f)
{
    return fmtstrcpy(f, "\t");
}
```

⟨*function* dprint 23e⟩≡ (89)
```
int
dprint(char *fmt, ...)
{
    char buf[4096];
    va_list arg;
    int n;
    ⟨dprint() locals 24b⟩

    ⟨dprint() return if mkfault 78c⟩

    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 24c⟩
    return n;
}
```

⟨*function* printc 23f⟩≡ (89)
```
void
printc(int c)
{
    dprint("%c", c);
}
```

⟨*function* prints 23g⟩≡ (89)
```
void
prints(char *s)
{
    dprint("%s",s);
}
```

⟨*global* printcol 24a⟩≡ (89)
```
int printcol = 0;
```

⟨dprint() *locals* 24b⟩≡ (23e)
```
char *p;
Rune r;
int w;
```

⟨dprint() *maintain printcol* 24c⟩≡ (23e)
```
for(p=buf; *p; p+=w){
    w = chartorune(&r, p);
    if(r == '\n')
        printcol = 0;
    else
        printcol++;
}
```

⟨*function* flushbuf 24d⟩≡ (89)
```
void
flushbuf(void)
{
  if (printcol != 0)
        printc(EOR);
}
```

⟨*function* flush 24e⟩≡ (89)
```
void
flush(void)
{
    Bflush(&stdout);
}
```

5.5 Input: clrinp(), rdc(), reread()

⟨*constant* EOR 24f⟩≡ (85c)
```
#define EOR '\n'
```

⟨*constant* SPC 24g⟩≡ (85c)
```
#define SPC ' '
```

⟨*constant* TB 24h⟩≡ (85c)
```
#define TB '\t'
```

⟨*global* lp 24i⟩≡ (90b)
```
Rune *lp;
```

⟨*global* peekc 24j⟩≡ (90b)
```
int peekc;
```

⟨*global* lastc 24k⟩≡ (90b)
```
int lastc = EOR;
```

⟨*function* clrinp 24l⟩≡ (90b)
```
void
clrinp(void)
{
    flush();
    lp = nil;
    peekc = 0;
}
```

⟨*function* rdc 25a⟩≡ (90b)
```
int
rdc(void)
{
    do {
        readchar();
    } while (lastc==SPC || lastc==TB);
    return lastc;
}
```

⟨*function* reread 25b⟩≡ (90b)
```
void
reread(void)
{
    peekc = lastc;
}
```

5.6 Interpreter: command()

⟨*function* command 25c⟩≡ (101b)
```
/* command decoding */
int
command(char *buf, int defcom)
{
    ⟨command() locals (db) 25d⟩

    ⟨command() initializations (db) 26f⟩

    do {
        ⟨command() parse possibly address, set dot 26b⟩
        ⟨command() parse possibly count, set cntval 26d⟩
        ⟨command() parse possibly command, set lastcom 26e⟩

        switch(lastcom) {
        ⟨command() switch lastcom cases 36a⟩
        default:
            error("bad command");
        }
        flushbuf();
    } while (rdc()==';');

    ⟨command() finalizations (db) 72a⟩

    ⟨command() return (db) 56a⟩
}
```

⟨command() locals (db) 25d⟩≡ (25c) 25e▷
```
static char lastcom = '=';
```

⟨command() locals (db) 25e⟩+≡ (25c) ◁25d 64c▷
```
static char savecom = '=';
```

5.6.1 Addresses

⟨*global* adrflg 25f⟩≡ (88b)
```
bool adrflg;
```

⟨*global* `adrval` 26a⟩≡ (88b)
```
// was in command.c
WORD adrval;
```

⟨`command()` *parse possibly address, set dot* 26b⟩≡ (25c)
```
adrflg=expr(0);  /* first address */
if (adrflg){
    dot=expv;
    ditto=expv;
}
adrval=dot;
```

5.6.2 Counts

⟨*global* `cntflg` 26c⟩≡ (88b)
```
bool cntflg;
```

⟨`command()` *parse possibly count, set cntval* 26d⟩≡ (25c)
```
if (rdc()==',' && expr(0)) { /* count */
    cntflg=TRUE;
    cntval=expv;
} else {
    cntflg=FALSE;
    cntval=1;
    reread();
}
```

5.6.3 Commands

⟨`command()` *parse possibly command, set lastcom* 26e⟩≡ (25c)
```
if (!eol(rdc()))
    lastcom=lastc;  /* command */
else {
    if (adrflg==false)
        dot=inkdot(dotinc);
    lastcom=defcom;
    reread();
}
```

⟨`command()` *initializations (db)* 26f⟩≡ (25c) 71e▷
```
if (defcom == 0)
    defcom = lastcom;
```

⟨*function* `eol` 26g⟩≡ (90b)
```
bool
eol(int c)
{
    return(c==EOR || c==';');
}
```

⟨*function* `inkdot` 26h⟩≡ (92a)
```
ADDR
inkdot(int incr)
{
    ADDR newdot;

    newdot = dot+incr;
    ⟨inkdot() error managment 27a⟩
    return newdot;
}
```

⟨inkdot() *error managment* 27a⟩≡ (26h)
```
if ((incr >= 0 && newdot < dot)
||  (incr < 0 && newdot > dot))
    error("address wraparound");
```

5.7 Exit: done()

⟨*global* eof 27b⟩≡ (90b)
```
bool eof;
```

⟨main() *in loop, if eof (db)* 27c⟩≡ (18d)
```
if (eof) {
    if (infile == STDIN)
        done(); // will exits()
    ⟨main() in loop, if eof, and if infile was not STDIN 73f⟩
}
```

⟨*function* done 27d⟩≡ (102c)
```
void
done(void)
{
    if (pid)
        endpcs();
    exits(nil);
}
```

Chapter 6

libmach/

6.1 Fhdr and crackhdr()

⟨*function* crackhdr 28⟩≡ (154b)

```
int
crackhdr(int 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;
}
```

6.2 ARM

6.3 Map, `syminit()`, and `loadmap()`

6.4 `machdata`

6.5 Symbol table

Chapter 7

Command Input

⟨*function* `readchar` 30a⟩≡ (90b)
```
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 30d⟩
        lastc = *lp;
        if (lastc != '\0')
            lp++;
    }
    return lastc;
}
```

⟨*global* `line` 30b⟩≡ (90b)
```
Rune line[LINSIZ];
```

⟨*constant* `LINSIZ` 30c⟩≡ (85c)
```
#define LINSIZ 4096
```

⟨readchar() *if lp is nil read a line and set lp* 30d⟩≡ (30a)
```
if (lp==nil) {
    for (p = line; p < &line[LINSIZ-1]; p++) {
        eof = (readrune(infile, p) <= 0);
        ⟨readchar() if mkfault 78d⟩
        if (eof) {
            p--;
            break;
        }
        if (*p == EOR) {
            if (p <= line)
                break;
            if (p[-1] != '\\')
                break;
            p -= 2;
        }
    }
    p[1] = '\0';
    lp = line;
}
```

⟨*function* `readrune` 31a⟩≡ (90b)

```
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` 31b⟩≡ (90b)

```
int
nextchar(void)
{
    if (eol(rdc())) {
        reread();
        return 0;
    }
    return lastc;
}
```

Chapter 8

Command Parsing

⟨*global* expv 32a⟩≡ (88b)
```
// was in expr.c
uvlong expv;
```

8.1 expr()

⟨*function* expr 32b⟩≡ (97d)
```
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;
}
```

8.2 term()

⟨*function* term 33⟩≡ (97d)

```
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);
    }
}
```

8.3 item()

⟨*function* item 34⟩≡ (97d)

```
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;
}
```

⟨*global* ditto 35⟩≡ (88b)

```
ADDR ditto;
```

Chapter 9

Commands

9.1 Dumper commands: $

⟨command() *switch lastcom cases* 36a⟩≡ (25c) 41g▷
```
case '$':
    lastcom=savecom;
    printtrace(nextchar());
    break;
```

⟨*function* printtrace 36b⟩≡ (100a)
```
void
printtrace(int modif)
{
    ⟨printtrace() locals 36c⟩

    if (cntflg==FALSE)
        cntval = -1;

    switch (modif) {
    ⟨printtrace() switch modif cases 36d⟩
    default:
        error("bad '$' command");
    }

}
```

⟨printtrace() *locals* 36c⟩≡ (36b) 38b▷
```
int i;
ulong w;
BKPT *bk;
Symbol s;
int stack;
char *fname;
char buf[512];
```

9.1.1 Current process: $?

⟨printtrace() *switch modif cases* 36d⟩≡ (36b) 37a▷
```
case 0:
case '?':
    if (pid)
        dprint("pid = %d\n",pid);
    else
        prints("no process\n");
    flushbuf();
```

9.1.2 Address maps: $m

⟨printtrace() *switch modif cases* 37a⟩+≡ (36b) ◁36d 37c▷
```
  case 'm':
      printmap("? map", symmap);
      printmap("/ map", cormap);
      break;
```

⟨*function* printmap 37b⟩≡ (100a)
```
  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);
      }
  }
```

9.1.3 Symbols: $S and $e

⟨printtrace() *switch modif cases* 37c⟩+≡ (36b) ◁37a 38a▷
```
  case 'S':
      printsym();
      break;
```

⟨*function* printsym 37d⟩≡ (100a)
```
  /*
   * 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;
        }
    }
}
```

⟨printtrace() *switch modif cases* 38a⟩+≡ (36b) ◁37c 38d▷

```
/*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;
```

9.1.4 Stack traces: $c

⟨printtrace() *locals* 38b⟩+≡ (36b) ◁36c

```
uvlong pc, sp, link;
```

⟨*global* tracetype 38c⟩≡ (100a)

```
static int tracetype;
```

⟨printtrace() *switch modif cases* 38d⟩+≡ (36b) ◁38a 39b▷

```
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;
```

⟨*function* ptrace 39a⟩≡ (100a)

```
/*
 * 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);
}
```

9.1.5 Registers: $r and $f

⟨printtrace() *switch modif cases* 39b⟩+≡ (36b) ◁38d 40a▷

```
case 'r':
case 'R':
    printregs(modif);
    return;
```

⟨*function* printregs 39c⟩≡ (94a)

```
/*
 * 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();
}
```

⟨printtrace() *switch modif cases* 40a⟩+≡ (36b) ◁39b 40c▷

```
case 'f':
case 'F':
    printfp(cormap, modif);
    return;
```

⟨*function* printfp 40b⟩≡ (100a)

```
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++;
    }
}
```

9.1.6 Quitting: $q

⟨printtrace() *switch modif cases* 40c⟩+≡ (36b) ◁40a 40d▷

```
case 'q':
case 'Q':
    done();
```

9.1.7 XXX

⟨printtrace() *switch modif cases* 40d⟩+≡ (36b) ◁40c 41e▷

```
case 'a':
    attachprocess();
    break;
```

⟨*function* attachprocess 41a⟩≡ (91)
```
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 41b⟩
    if (fd >= 0)
        close(fd);
}
```

⟨attachprocess() *error managment* 41b⟩≡ (41a)
```
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);
```

⟨*constant* MAXOFF 41c⟩≡ (85c)
```
#define MAXOFF 0x1000000
```

⟨*global* maxoff 41d⟩≡ (88b)
```
ADDR maxoff = MAXOFF;
```

⟨printtrace() *switch modif cases* 41e⟩+≡ (36b) ◁40d 41f▷
```
case 's':
    maxoff=(adrflg?adrval:MAXOFF);
    break;
```

⟨printtrace() *switch modif cases* 41f⟩+≡ (36b) ◁41e 45c▷
```
case 'M':
    fname = getfname();
    if (machbyname(fname) == 0)
        dprint("unknown name\n");;
    break;
```

9.2 Inspecting commands, ?/=

⟨command() *switch lastcom cases* 41g⟩+≡ (25c) ◁36a 53a▷
```
case '?':
case '/':
case '=':
    savecom = lastcom;
    acommand(lastcom);
    break;
```

⟨*function* acommand 42a⟩≡ (101b)

```
/*
 * [/?][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 50e⟩
    default:
        reread();
        getformat(fmt);
        scanform(cntval, !eqcom, fmt, map, eqcom);
    }
}
```

9.2.1 Formats

⟨*constant* ARB 42b⟩≡ (85c)

```
#define ARB 512
```

⟨*global* eqformat 42c⟩≡ (101b)

```
char eqformat[ARB] = "z";
```

⟨*global* stformat 42d⟩≡ (101b)

```
char stformat[ARB] = "zMi";
```

⟨*global* dotmap 42e⟩≡ (91)

```
Map *dotmap;
```

⟨main() *set dotmap* 42f⟩≡ (19b)

```
dotmap = dumbmap(-1);
```

⟨*function* getformat 43a⟩≡ (90b)

```
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';
}
```

⟨*function* scanform 43b⟩≡ (92a)

```
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);
    }
}
```

⟨*function* exform 43c⟩≡ (92a)

```
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 45d⟩
        default:
            error("bad modifier");
        }

        if (map->seg[0].fd >= 0)
            dot=inkdot(dotinc);
        fcount--;
        endline();
    }

    return fp;
}
```

⟨*function* endline 44⟩≡ (89)

```
void
endline(void)
{

    if (printcol >= maxpos)
```

```
        newline();
}
```

⟨*constant* MAXPOS 45a⟩≡ (85c)
```
#define MAXPOS 80
```

⟨*global* maxpos 45b⟩≡ (89)
```
int maxpos = MAXPOS;
```

⟨printtrace() *switch modif cases* 45c⟩+≡ (36b) ◁41f 62a▷
```
case 'w':
    maxpos=(adrflg?adrval:MAXPOS);
    break;
```

9.2.2 Instruction disassembling: ?i

⟨exform() *switch modifier cases* 45d⟩≡ (43c) 45f▷
```
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;
```

⟨*global* dotinc 45e⟩≡ (88b)
```
int dotinc;
```

9.2.3 Registers

9.2.4 XXX

⟨exform() *switch modifier cases* 45f⟩+≡ (43c) ◁45d 45g▷
```
case SPC:
case TB:
    dotinc = 0;
    break;
```

⟨exform() *switch modifier cases* 45g⟩+≡ (43c) ◁45f 45h▷
```
case 't':
case 'T':
    dprint("%*t", fcount);
    dotinc = 0;
    return(fp);
```

⟨exform() *switch modifier cases* 45h⟩+≡ (43c) ◁45g 45i▷
```
case 'a':
    symoff(buf, sizeof(buf), dot, CANY);
    dprint("%s%c%16t", buf, map==symmap? '?':'/');
    dotinc = 0;
    break;
```

⟨exform() *switch modifier cases* 45i⟩+≡ (43c) ◁45h 46a▷
```
case 'A':
    dprint("%#llux%10t", dot);
    dotinc = 0;
    break;
```

⟨exform() *switch modifier cases* 46a⟩+≡ (43c) ◁45i 46b▷

```
case 'p':
    if (get4(map, dot, &w) < 0)
        error("%r");
    symoff(buf, sizeof(buf), w, CANY);
    dprint("%s%16t", buf);
    dotinc = mach->szaddr;
    break;
```

⟨exform() *switch modifier cases* 46b⟩+≡ (43c) ◁46a 46c▷

```
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;
```

⟨exform() *switch modifier cases* 46c⟩+≡ (43c) ◁46b 46d▷

```
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;
```

⟨exform() *switch modifier cases* 46d⟩+≡ (43c) ◁46c 47a▷

```
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;
```

⟨exform() *switch modifier cases* 47a⟩+≡ (43c) ◁46d 47c▷

```
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;
```

⟨*function* printesc 47b⟩≡ (92a)

```
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);
}
```

⟨exform() *switch modifier cases* 47c⟩+≡ (43c) ◁47a 47d▷

```
case 'r':
    if (literal)
        sh = (ushort) dot;
    else if (get2(map, dot, &sh) < 0)
        error("%r");
    dprint("%C", sh);
    dotinc = 2;
    break;
```

⟨exform() *switch modifier cases* 47d⟩+≡ (43c) ◁47c 48a▷

```
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;
```

⟨exform() *switch modifier cases* 48a⟩+≡ (43c) ◁47d 48b▷

```
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;
```

⟨exform() *switch modifier cases* 48b⟩+≡ (43c) ◁48a 49a▷

```
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;
```

⟨exform() *switch modifier cases* 49a⟩+≡ (43c) ◁48b 49b▷

```
  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;
```

⟨exform() *switch modifier cases* 49b⟩+≡ (43c) ◁49a 49c▷

```
  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;
```

⟨exform() *switch modifier cases* 49c⟩+≡ (43c) ◁49b 49d▷

```
  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;
```

⟨exform() *switch modifier cases* 49d⟩+≡ (43c) ◁49c 49e▷

```
  case 'n':
  case 'N':
    printc('\n');
    dotinc=0;
    break;
```

⟨exform() *switch modifier cases* 49e⟩+≡ (43c) ◁49d 50a▷

```
  case '"':
    dotinc=0;
    while (*fp != '"' && *fp)
        printc(*fp++);
    if (*fp)
        fp++;
    break;
```

⟨exform() *switch modifier cases* 50a⟩+≡ (43c) ◁49e 50b▷
```
  case '^':
      dot=inkdot(-dotinc*fcount);
      return(fp);
```

⟨exform() *switch modifier cases* 50b⟩+≡ (43c) ◁50a 50c▷
```
  case '+':
      dot=inkdot((WORD)fcount);
      return(fp);
```

⟨exform() *switch modifier cases* 50c⟩+≡ (43c) ◁50b 50d▷
```
  case '-':
      dot=inkdot(-(WORD)fcount);
      return(fp);
```

⟨exform() *switch modifier cases* 50d⟩+≡ (43c) ◁50c
```
  case 'z':
      if (findsym(dot, CTEXT, &s))
          dprint("%s() ", s.name);
      printsource(dot);
      printc(EOR);
      return fp;
```

⟨acommand() *switch optional command suffix character* 50e⟩≡ (42a) 51a▷
```
  case 'm':
      if (eqcom)
          error(BADEQ);
      cmdmap(map);
      break;
```

⟨*function* cmdmap 50f⟩≡ (91)
```
  /*
   * 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();
}
```

⟨acommand() *switch optional command suffix character* 51a⟩+≡ (42a) ◁50e 52a▷

```
case 'L':
case 'l':
    if (eqcom)
        error(BADEQ);
    cmdsrc(lastc, map);
    break;
```

⟨*function* cmdsrc 51b⟩≡ (101b)

```
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);
}
```

⟨acommand() *switch optional command suffix character* 52a⟩+≡ (42a) ◁51a
```
case 'W':
case 'w':
    if (eqcom)
        error(BADEQ);
    cmdwrite(lastc, map);
    break;
```

⟨*global* badwrite 52b⟩≡ (101b)
```
static char badwrite[] = "can't write process memory or text image";
```

⟨*function* cmdwrite 52c⟩≡ (101b)
```
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;
}
```

⟨*function* newline 52d⟩≡ (89)
```
void
newline(void)
{
    printc(EOR);
}
```

9.3 Sub process control commands, :

⟨command() *switch lastcom cases* 53a⟩+≡ (25c) ◁41g 64d▷
```
case ':':
    if (!executing) {
        executing=TRUE;
        subpcs(nextchar());
        executing=FALSE;
        lastcom=savecom;
    }
    break;
```

⟨*global* executing 53b⟩≡ (101b)
```
bool executing;
```

⟨main() *just before repl* 53c⟩≡ (19b)
```
if (executing)
    delbp();
executing = FALSE;
```

⟨*function* subpcs 53d⟩≡ (102b)
```
/* 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 56c⟩
    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();
}
```

⟨*global* loopcnt 54a⟩≡ (101b)
```
WORD loopcnt;
```

9.3.1 printpc()

⟨*function* printpc 54b⟩≡ (100a)
```
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);
    }
}
```

⟨*constant* STRINGSZ 54c⟩≡ (100a)
```
#define STRINGSZ 128
```

⟨*function* printsource 54d⟩≡ (100a)
```
/*
 * print the value of dot as file:line
 */
void
printsource(ADDR dot)
{
    char str[STRINGSZ];

    if (fileline(str, STRINGSZ, dot))
        dprint("%s", str);
}
```

9.3.2 Runmodes

⟨*constant* SINGLE 54e⟩≡ (85c)
```
#define SINGLE 1
```

⟨*constant* CONTIN 54f⟩≡ (85c)
```
#define CONTIN 2
```

9.3.3 runpcs()

⟨*function* runpcs 54g⟩≡ (102a)
```
/* 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 56b⟩
            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;
}
```

⟨command() *return (db)* 56a⟩≡ (25c)
```
if(adrflg)
    return dot;
return 1;
```

⟨runpcs() *in SINGLE mode, clean breakpoint if at dot* 56b⟩≡ (54g)
```
bkpt = scanbkpt(dot);
if (bkpt) {
    switch(bkpt->flag){
    case BKPTTMP:
        bkpt->flag = BKPTCLR;
        break;
    case BKPTSKIP:
        bkpt->flag = BKPTSET;
        break;
    }
}
```

9.3.4 Running: :r

⟨subpcs() *switch modif cases* 56c⟩≡ (53d) 59a ▷
```
/* run program */
case 'r':
case 'R':
    endpcs();
    setup();
    runmode = CONTIN;
    break;
```

setup() and startpcs()

⟨*function* setup 56d⟩≡ (102a)
```
/*
 * start up the program to be debugged in a child
 */
void
setup(void)
{

    nnote = 0;
    startpcs();
    pcsactive = true;
    bpin = FALSE;
}
```

⟨*function* startpcs 56e⟩≡ (98)
```
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();
}
```

⟨*global* procname 57a⟩≡ (98)
```
static char procname[100];
```

⟨*function* bpwait 57b⟩≡ (98)
```
void
bpwait(void)
{
    setcor();
    unloadnote();
}
```

⟨*global* child 57c⟩≡ (98)
```
int child;
```

doexec()

⟨*constant* MAXARG 57d⟩≡ (85c)
```
#define MAXARG 32
```

⟨*function* doexec 57e⟩≡ (98)
```
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 58a⟩
    *ap = '\0';

    exec(symfil, argl);
    perror(symfil);
}
```

⟨doexec() *adjust argl if extra arguments* 58a⟩≡ (57e)

```
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;
}
```

runrun()

⟨*function* runrun 58b⟩≡ (98)

```
void
runrun(bool keepnote)
{
    ⟨runrun() notes managment 58c⟩

    flush();
    msgpcs("startstop");
    bpwait();
}
```

⟨runrun() *notes managment* 58c⟩≡ (58b)

```
int on = nnote;

unloadnote();
if(on != nnote){
    notes();
    error("not running: new notes pending");
}
if(keepnote)
    loadnote();
else
    nnote = 0;
```

9.3.5 Stepping: :s

⟨subpcs() *switch modif cases* 59a⟩+≡ (53d) ◁56c 60a▷
```
/* single step */
case 's':
    if (pid == 0) {
        setup();
        loopcnt--;
    }
    runmode=SINGLE;
    keepnote=defval(true);
    break;
```

⟨*function* defval 59b⟩≡ (97d)
```
WORD
defval(WORD w)
{
    if (expr(0))
        return (expv);
    else
        return (w);
}
```

⟨*function* runstep 59c⟩≡ (98)
```
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);
}
```

9.3.6 Killing: :k

⟨*global* NOPCS 59d⟩≡ (102b)
```
char NOPCS[] = "no process";
```

⟨subpcs() *switch modif cases* 60a⟩+≡ (53d) ◁59a 61a▷
```
/* exit */
case 'k' :
case 'K':
    if (pid == 0)
        error(NOPCS);
    dprint("%d: killed", pid);
    pcsactive = true; /* force 'kill' ctl */
    endpcs();
    return;
```

⟨*global* ending 60b⟩≡ (88b)
```
// was in runpcs.c
bool ending;
```

⟨*function* endpcs 60c⟩≡ (102a)
```
/*
 * 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;
}
```

⟨*function* killpcs 60d⟩≡ (98)
```
void
killpcs(void)
{
    msgpcs("kill");
}
```

⟨*global* bpin 60e⟩≡ (102a)
```
bool bpin;
```

9.3.7 Halting: :h

⟨subpcs() *switch modif cases* 61a⟩+≡ (53d) ◁60a 61d▷
```
/* halt the current process */
case 'h':
    ⟨subpcs() halting case, if addr 0 specified 61c⟩
    else {
        grab();
        dprint("stopped at%16t");
        goto Return;
    }
    return;
```

⟨*function* grab 61b⟩≡ (98)
```
void
grab(void)
{
    flush();
    msgpcs("stop");
    bpwait();
}
```

⟨subpcs() *halting case, if addr 0 specified* 61c⟩≡ (61a)
```
if (adrflg && adrval == 0) {
    if (pid == 0)
        error(NOPCS);
    ungrab();
}
```

9.3.8 Unhalting: :x

⟨subpcs() *switch modif cases* 61d⟩+≡ (53d) ◁61a 62b▷
```
/* continue executing the current process */
case 'x':
    if (pid == 0)
        error(NOPCS);
    ungrab();
    return;
```

⟨*function* ungrab 61e⟩≡ (98)
```
void
ungrab(void)
{
    msgpcs("start");
}
```

9.4 Breakpoints

⟨*function* scanbkpt 61f⟩≡ (102a)
```
/*
 * 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;
}
```

9.4.1 Inspecting breakpoints: $b

⟨printtrace() *switch modif cases* 62a⟩+≡ (36b) ◁45c 73b▷

```
/*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;
```

9.4.2 Setting breakpoints: :b

⟨subpcs() *switch modif cases* 62b⟩+≡ (53d) ◁61d 62d▷

```
/* set breakpoint */
case 'b':
case 'B':
    if (bk=scanbkpt(dot))
        bk->flag=BKPTCLR;

    ⟨subpcs() breakpoint case, find unused breakpoint bk or allocate one 62c⟩

    bk->loc = dot;
    bk->flag = modif == 'b' ? BKPTSET : BKPTTMP;
    bk->initcnt = bk->count = cntval;

    ⟨subpcs() breakpoint case, set optional breakpoint command 71d⟩
```

⟨subpcs() *breakpoint case, find unused breakpoint bk or allocate one* 62c⟩≡ (62b)

```
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;
}
```

9.4.3 Deleting breakpoints: :d

⟨subpcs() *switch modif cases* 62d⟩+≡ (53d) ◁62b 63a▷

```
/* delete breakpoint */
case 'd':
```

```
case 'D':
    if ((bk=scanbkpt(dot)) == 0)
        error("no breakpoint set");
    bk->flag=BKPTCLR;
    return;
```

9.4.4 Continuing execution: :c

⟨subpcs() *switch modif cases* 63a⟩+≡ (53d) ◁62d 67a▷

```
/* continue with optional note */
case 'c':
case 'C':
    if (pid==0)
        error(NOPCS);
    runmode=CONTIN;
    keepnote=defval(1);
    break;
```

9.4.5 Installing breakpoints

⟨*function* setbp 63b⟩≡ (102a)

```
/*
 * 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;
}
```

⟨*function* bkput 63c⟩≡ (98)

```
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);
    }
}
```

9.4.6 Uninstalling breakpoints

⟨*function* delbp 64a⟩≡ (102a)

```
/*
 * 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;
}
```

9.4.7 Executing breakpoints

⟨*function* execbkpt 64b⟩≡ (102a)

```
/*
 * 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;
}
```

9.4.8 Breakpoints and notes

9.5 Other commands

⟨command() *locals (db)* 64c⟩+≡ (25c) ◁25e 71f▷

```
char *reg;
char savc;
```

⟨command() *switch lastcom cases* 64d⟩+≡ (25c) ◁53a 74a▷

```
case '>':
    lastcom = savecom;
    savc=rdc();
    if (reg=regname(savc))
```

```
    rput(cormap, reg, dot);
else
    error("bad variable");
break;
```

Chapter 10

Metadata Generation

10.1 Assembler

10.2 Compiler

10.3 Linker

Chapter 11

Source level C Debugging

11.1 Stepping: :S

⟨subpcs() *switch modif cases* 67a⟩+≡ (53d) ◁63a 79c▷

```
  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;
```

11.2 Stack traces: $S

⟨*function* printparams 67b⟩≡ (100a)

```
  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);
    }
}
```

⟨*function* `printlocals` 68⟩≡ (100a)

```
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);
    }
}
```

Chapter 12

/bin/strace

Chapter 13

acid

Chapter 14

Advanced Features

14.1 Conditional breakpoints

⟨*constant* MAXCOM 71a⟩≡ (85c)
```
#define MAXCOM 64
```

⟨Bkpt *other fields* 71b⟩+≡ (16f) ◁17c
```
char comm[MAXCOM];
```

⟨*constant HUGEINT (db)* 71c⟩≡ (85c)
```
#define HUGEINT 0x7fffffff /* enormous WORD */
```

⟨subpcs() *breakpoint case, set optional breakpoint command* 71d⟩≡ (62b)
```
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");
```

⟨command() *initializations (db)* 71e⟩+≡ (25c) ◁26f
```
if (buf) {
    if (*buf==EOR)
        return FALSE;
    clrinp();
    lp=(Rune*)buf;
}
```

⟨command() *locals (db)* 71f⟩+≡ (25c) ◁64c
```
Rune*   savlp = lp;
char savlc = lastc;
char savpc = peekc;
```

⟨command() *finalizations (db)* 72a⟩≡ (25c)
```
if (buf == nil)
    reread();
else {
    clrinp();
    lp=savlp;
    lastc = savlc;
    peekc = savpc;
}
```

14.2 Input file debugging commands

⟨*constant* MAXIFD 72b⟩≡ (89)
```
#define MAXIFD 5
```

⟨*global* istack 72c⟩≡ (89)
```
struct {
    int fd;
    int r9;
} istack[MAXIFD];
```

⟨*global* ifiledepth 72d⟩≡ (89)
```
int ifiledepth;
```

⟨*function* iclose 72e⟩≡ (89)
```
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;
        }
    }
}
```

⟨*function* redirout 73a⟩≡ (89)

```
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);
}
```

⟨printtrace() *switch modif cases* 73b⟩+≡ (36b) ◁62a 73c▷

```
case '<':
    if (cntval == 0) {
        while (readchar() != EOR)
            ;
        reread();
        break;
    }
    if (rdc() == '<')
        stack = 1;
    else {
        stack = 0;
        reread();
    }
    fname = getfname();
    redirin(stack, fname);
    break;
```

⟨printtrace() *switch modif cases* 73c⟩+≡ (36b) ◁73b 77b▷

```
case '>':
    fname = getfname();
    redirout(fname);
    break;
```

⟨main() *locals (db)* 73d⟩+≡ (18d) ◁19g 76a▷

```
char *s;
```

⟨main() *command line processing (db)* 73e⟩≡ (18d) 74e▷

```
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* 73f⟩≡ (27c)

```
iclose(-1, 0);
eof = false;
longjmp(env, 1);
```

14.3 Formatted output

⟨command() *switch lastcom cases* 74a⟩+≡ (25c) ◁64d 74c▷
```
case '\0':
    prints(DBNAME);
    break;
```

⟨*constant* DBNAME 74b⟩≡ (85c)
```
#define DBNAME "db\n"
```

14.4 Shell output

⟨command() *switch lastcom cases* 74c⟩+≡ (25c) ◁74a
```
case '!':
    lastcom=savecom;
    shell();
    break;
```

⟨*function* shell 74d⟩≡ (101b)
```
/*
 * 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();
    }
}
```

14.5 db -w

⟨main() *command line processing (db)* 74e⟩+≡ (18d) ◁73e 76b▷
```
case 'w':
    wtflag = ORDWR;  /* suitable for open() */
    break;
```

⟨`getfile()` *if wtflag* 75a⟩≡ (22a)

```
if (f < 0 && xargc > cnt && wtflag)
     f = create(filnam, 1, 0666);
```

⟨*global* `xargc` 75b⟩≡ (88b)

```
int xargc;  /* bullshit */
```

⟨`main()` *initialization before repl (db)* 75c⟩+≡ (18d) ◁19b

```
xargc = argc;
```

Chapter 15

Advanced Topics

15.1 Time travel

15.2 Cross machine debugging

15.3 Cross architecture debugging: `db -m`

⟨`main()` *locals (db)* 76a⟩+≡ (18d) ◁73d 76f▷
```
char *name = nil;
```

⟨`main()` *command line processing (db)* 76b⟩+≡ (18d) ◁74e 76e▷
```
case 'm':
    name = ARGF();
    if(name == nil)
        dprint("missing -m argument\n");
    break;
```

⟨`main()` *if db -m and unknown machine* 76c⟩≡ (19c)
```
if (name && machbyname(name) == 0)
        dprint ("unknown machine %s", name);
```

15.4 Kernel debugging

15.4.1 `/bin/ktrace`

15.4.2 `db -k`

⟨*global* `kflag` 76d⟩≡ (88b)
```
bool kflag;
```

⟨`main()` *command line processing (db)* 76e⟩+≡ (18d) ◁76b
```
case 'k':
    kflag = true;
    break;
```

⟨`main()` *locals (db)* 76f⟩+≡ (18d) ◁76a
```
char *cpu, *p, *q;
```

⟨main() *when pid argument, if kflag* 77a⟩≡ (20a)

```
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* 77b⟩+≡ (36b) ◁73c

```
case 'k':
    kmsys();
    break;
```

⟨*function* kmsys 77c⟩≡ (91)

```
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;
    }
}
```

15.5 Signals and notes

15.5.1 Debugger notes

⟨main() *call notify* 77d⟩≡ (19b)

```
notify(fault);
```

⟨*function* fault 77e⟩≡ (102c)

```
/*
 * 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);
}
```

⟨*global* `mkfault` 78a⟩≡ (88b)
```
bool mkfault;
```

⟨`main()` *in loop, handle mkfault (db)* 78b⟩≡ (18d)
```
if (mkfault) {
    mkfault=0;
    printc('\n');
    prints(DBNAME);
}
```

⟨`dprint()` *return if mkfault* 78c⟩≡ (23e)
```
if(mkfault)
    return -1;
```

⟨`readchar()` *if mkfault* 78d⟩≡ (30d)
```
if (mkfault) {
    eof = 0;
    error(nil);
}
```

15.5.2 Debugged notes

⟨*function* `loadnote` 78e⟩≡ (98)
```
/*
 * 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;
}
```

⟨*function* notes 79a⟩≡ (98)

```
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]);
}
```

⟨*function* unloadnote 79b⟩≡ (98)

```
/*
 * 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;
    }
}
```

15.5.3 :n

⟨subpcs() *switch modif cases* 79c⟩+≡ (53d) ◁67a

```
/* 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;
```

Chapter 16

Conclusion

Appendix A

Debugging

Appendix B

Error Management

⟨*function* `error` 82a⟩≡ (88a)
```
/*
 * 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);
}
```

⟨*function* `errors` 82b⟩≡ (88a)
```
void
errors(char *m, char *n)
{
    static char buf[128];

    sprint(buf, "%s: %s", m, n);
    error(buf);
}
```

⟨*global* `errmsg` 82c⟩≡ (88b)
```
// was static in main.c
char *errmsg;
```

⟨`main()` *in loop, handle errmsg (db)* 82d⟩≡ (18d)
```
if (errmsg) {
    dprint(errmsg);
    printc('\n');
    errmsg = nil;
}
```

⟨*function* `oclose` 82e⟩≡ (89)
```
void
oclose(void)
{
    flushbuf();
    Bterm(&stdout);
```

```
    Binit(&stdout, 1, OWRITE);
}
```

Appendix C

Utilities

Appendix D

Extra Code

D.1 db/

D.1.1 db/defs.h

⟨*constant* TRUE 85a⟩≡ (85c)
```
#define TRUE (-1)
```

⟨*constant* FALSE 85b⟩≡ (85c)
```
#define FALSE 0
```

⟨db/defs.h 85c⟩≡
```
/*
 * 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 16b⟩
⟨type WORD 16d⟩

typedef struct bkpt BKPT;

⟨constant HUGEINT (db) 71c⟩

⟨constant MAXOFF 41c⟩
⟨constant INCDIR 99a⟩
⟨constant DBNAME 74b⟩
⟨constant CMD_VERBS 9b⟩

⟨constant MAXPOS 45a⟩
⟨constant MAXLIN 95d⟩
⟨constant ARB 42b⟩
⟨constant MAXCOM 71a⟩
⟨constant MAXARG 57d⟩
⟨constant LINSIZ 30c⟩
⟨constant MAXSYM 96a⟩

⟨constant EOR 24f⟩
```

⟨*constant* SPC 24g⟩
⟨*constant* TB 24h⟩

⟨*constant* TRUE 85a⟩
⟨*constant* FALSE 85b⟩

```
/*
 * run modes
 */
```
⟨*constant* SINGLE 54e⟩
⟨*constant* CONTIN 54f⟩

```
/*
 * breakpoints
 */
```
⟨*constant* BKPTCLR 16g⟩
⟨*constant* BKPTSET 16h⟩
⟨*constant* BKPTSKIP 16i⟩
⟨*constant* BKPTTMP 16j⟩

⟨*struct* bkpt 16f⟩

```
/*
 * 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 17h⟩

```
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;
```

D.1.2 db/fns.h

⟨db/fns.h 87⟩≡

```
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
```

D.1.3 db/utils.c

⟨db/utils.c 88a⟩≡

```
#include "defs.h"
#include "fns.h"

⟨function error 82a⟩

⟨function errors 82b⟩
```

D.1.4 db/globals.c

⟨db/globals.c 88b⟩≡

```
#include "defs.h"

⟨global wtflag 21g⟩
⟨global kflag 76d⟩
```

⟨*global* `mkfault` 78a⟩
⟨*global* `maxoff` 41d⟩

⟨*global* `errmsg` 82c⟩
⟨*global* `env` 19d⟩

⟨*global* `symmap` 14c⟩
⟨*global* `cormap` 15a⟩

⟨*global* `ending` 60b⟩
⟨*global* `pid` 19f⟩
⟨*global* `nnote` 17i⟩
⟨*global* `note` 17g⟩
⟨*global* `bkpthead` 17a⟩

⟨*global* `dot` 16c⟩
⟨*global* `dotinc` 45e⟩
⟨*global* `ditto` 35⟩

⟨*global* `adrval` 26a⟩
⟨*global* `adrflg` 25f⟩

⟨*global* `cntval` 16e⟩
⟨*global* `cntflg` 26c⟩

⟨*global* `expv` 32a⟩

⟨*global* `pcsactive` 20b⟩

⟨*global* `xargc` 75b⟩

D.1.5 `db/output.c`

⟨`db/output.c` 89⟩≡

```
/*
 *
 * debugger
 *
 */

#include "defs.h"
#include "fns.h"
```

⟨*global* `printcol` 24a⟩
⟨*global* `infile` 18c⟩
⟨*global* `maxpos` 45b⟩

⟨*global* `stdout` 23b⟩

⟨*function* `printc` 23f⟩

⟨*function* `tconv` 23d⟩

⟨*function* `flushbuf` 24d⟩

⟨*function* `prints` 23g⟩

⟨*function* `newline` 52d⟩

⟨*constant* `MAXIFD` 72b⟩
⟨*global* `istack` 72c⟩
⟨*global* `ifiledepth` 72d⟩

⟨*function* `iclose` 72e⟩

⟨*function* `oclose` 82e⟩

⟨*function* `redirout` 73a⟩

⟨*function* `endline` 44⟩

⟨*function* `flush` 24e⟩

⟨*function* `dprint` 23e⟩

⟨*function* `outputinit` 23c⟩

D.1.6 db/input.c

⟨*function* `isfileref` 90a⟩≡ (90b)

```
/*
 * 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;
}
```

⟨`db/input.c` 90b⟩≡

```
/*
 *
 * debugger
 *
```

```
 */

#include "defs.h"
#include "fns.h"

extern int infile;

⟨global line 30b⟩
⟨global lp 24i⟩
⟨global peekc 24j⟩
⟨global lastc 24k⟩
⟨global eof 27b⟩

/* input routines */

⟨function eol 26g⟩

⟨function rdc 25a⟩

⟨function reread 25b⟩

⟨function clrinp 24l⟩

⟨function readrune 31a⟩

⟨function readchar 30a⟩

⟨function nextchar 31b⟩

⟨function quotchar 94c⟩

⟨function getformat 43a⟩

⟨function isfileref 90a⟩
```

D.1.7 db/setup.c

⟨db/setup.c 91⟩≡

```
/*
 * init routines
 */
#include "defs.h"
#include "fns.h"

⟨global symfil 18a⟩
⟨global corfil 18b⟩

⟨global dotmap 42e⟩

⟨global fsym 20e⟩
⟨global fcor 22c⟩
⟨global fhdr (db/setup.c) 20d⟩

static int getfile(char*, int, int);

⟨function setsym (db) 20f⟩

⟨function setcor 22d⟩
```

```
extern Mach mi386;
extern Machdata i386mach;
```

⟨*function* dumbmap 21d⟩

⟨*function* cmdmap 50f⟩

⟨*function* getfile 21h⟩

⟨*function* kmsys 77c⟩

⟨*function* attachprocess 41a⟩

D.1.8 db/format.c

⟨db/format.c 92a⟩≡

```
/*
 *
 * debugger
 *
 */

#include "defs.h"
#include "fns.h"
```

⟨*function* scanform 43b⟩

⟨*function* exform 43c⟩

⟨*function* printesc 47b⟩

⟨*function* inkdot 26h⟩

D.1.9 db/regs.c

⟨*function* rname 92b⟩≡ (94a)

```
/*
 * 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;
}
```

⟨*function* getreg 92c⟩≡ (94a)

```
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;
}
```

⟨*function* rget 93a⟩≡ (94a)

```
uvlong
rget(Map *map, char *name)
{
    Reglist *rp;

    rp = rname(name);
    if (!rp)
        error("invalid register name");
    return getreg(map, rp);
}
```

⟨*function* rput 93b⟩≡ (94a)

```
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");
    }
```

⟨db/regs.c 94a⟩≡

```
    /*
     * code to keep track of registers
     */

    #include "defs.h"
    #include "fns.h"

    ⟨function rname 92b⟩

    ⟨function getreg 92c⟩

    ⟨function rget 93a⟩

    ⟨function rput 93b⟩
    ⟨function printregs 39c⟩
```

D.1.10 db/expr.c

⟨function ascval 94b⟩≡ (97d)

```
    static WORD
    ascval(void)
    {
        Rune r;

        if (readchar() == 0)
            return (0);
        r = lastc;
        while(quotchar()) /*discard chars to ending quote */
            ;
        return((WORD) r);
    }
```

⟨function quotchar 94c⟩≡ (90b)

```
    int
    quotchar(void)
    {
        if (readchar()=='\\')
            return(readchar());
        else if (lastc=='\'')
            return 0;
        else
            return lastc;
    }
```

⟨*struct* `fpin_union` 95a⟩≡ (97d)

```
union fpin_union {
        WORD w;
        float f;
};
```

⟨*function* `fpin` 95b⟩≡ (97d)

```
/*
 * 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);
}
```

⟨*constant* `MAXBASE` 95c⟩≡ (97d)

```
#define MAXBASE 16
```

⟨*constant* `MAXLIN` 95d⟩≡ (85c)

```
#define MAXLIN 128
```

⟨*function* `getnum` 95e⟩≡ (97d)

```
/* 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);
}
```

⟨*constant* MAXSYM 96a⟩≡ (85c)

```
#define MAXSYM 255
```

⟨*function* readsym 96b⟩≡ (97d)

```
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;
}
```

⟨*function* readfname 96c⟩≡ (97d)

```
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();
}
```

⟨*function* convdig 97a⟩≡ (97d)

```
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);
}
```

⟨*function* symchar 97b⟩≡ (97d)

```
int
symchar(int dig)
{
    if (lastc=='\\') {
        readchar();
        return(TRUE);
    }
    return(isalpha(lastc) || lastc>0x80 || lastc=='_' || dig && isdigit(lastc));
}
```

⟨*function* round 97c⟩≡ (97d)

```
static long
round(long a, long b)
{
    long w;

    w = (a/b)*b;
    if (a!=w)
        w += b;
    return(w);
}
```

⟨db/expr.c 97d⟩≡

```
/*
 *
 * debugger
 *
 */

#include "defs.h"
#include "fns.h"

static long round(long, long);

extern ADDR ditto;
```

⟨*function* `ascval` 94b⟩

⟨*struct* `fpin_union` 95a⟩

⟨*function* `fpin` 95b⟩

⟨*function* `defval` 59b⟩

⟨*function* `expr` 32b⟩

⟨*function* `term` 33⟩

⟨*function* `item` 34⟩

⟨*constant* `MAXBASE` 95c⟩

⟨*function* `getnum` 95e⟩

⟨*function* `readsym` 96b⟩

⟨*function* `readfname` 96c⟩

⟨*function* `convdig` 97a⟩

⟨*function* `symchar` 97b⟩

⟨*function* `round` 97c⟩

D.1.11 db/trcrun.c

⟨`db/trcrun.c` 98⟩≡

```
/*
 * functions for running the debugged process
 */

#include "defs.h"
#include "fns.h"
```

⟨*global* `child` 57c⟩
⟨*global* `msgfd` 15b⟩
⟨*global* `notefd` 17d⟩
⟨*global* `pcspid` 15c⟩

⟨*function* `setpcs` 15d⟩

⟨*function* `msgpcs` 15e⟩

⟨*function* `unloadnote` 79b⟩

⟨*function* `loadnote` 78e⟩

⟨*function* `notes` 79a⟩

⟨*function* `killpcs` 60d⟩

⟨*function* `grab` 61b⟩

⟨*function* ungrab 61e⟩

⟨*function* doexec 57e⟩

⟨*global* procname 57a⟩

⟨*function* startpcs 56e⟩

⟨*function* runstep 59c⟩

⟨*function* bpwait 57b⟩

⟨*function* runrun 58b⟩

⟨*function* bkput 63c⟩

D.1.12 db/print.c

⟨*constant* INCDIR 99a⟩≡ (85c)

```
#define INCDIR "/usr/lib/adb"
```

⟨*global* Ipath 99b⟩≡ (100a)

```
char *Ipath = INCDIR;
```

⟨*function* getfname 99c⟩≡ (100a)

```
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);
}
```

⟨*function* redirin 99d⟩≡ (100a)

```
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");
            }
        }
    }
```

⟨db/print.c 100a⟩≡

```
    /*
     *
     * debugger
     *
     */
    #include "defs.h"
    #include "fns.h"

    extern int infile;
    extern int outfile;
    extern int maxpos;

    /* general printing routines ($) */

    ⟨global Ipath 99b⟩
    ⟨global tracetype 38c⟩
    static void printfp(Map*, int);

    ⟨function ptrace 39a⟩

    ⟨function printtrace 36b⟩

    ⟨function getfname 99c⟩

    ⟨function printfp 40b⟩

    ⟨function redirin 99d⟩

    ⟨function printmap 37b⟩

    ⟨function printsym 37d⟩

    ⟨constant STRINGSZ 54c⟩

    ⟨function printsource 54d⟩

    ⟨function printpc 54b⟩

    ⟨function printlocals 68⟩

    ⟨function printparams 67b⟩
```

D.1.13 db/command.c

⟨global BADEQ 100b⟩≡ (101b)

```
    char BADEQ[] = "unexpected ‘=’";
```

⟨*function* regname 101a⟩≡ (101b)

```
/*
 * 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);
}
```

⟨db/command.c 101b⟩≡

```
/*
 *
 * debugger
 *
 */

#include "defs.h"
#include "fns.h"

⟨global BADEQ 100b⟩

⟨global executing 53b⟩
extern Rune *lp;

⟨global eqformat 42c⟩
⟨global stformat 42d⟩


⟨global loopcnt 54a⟩

⟨function command 25c⟩

⟨function acommand 42a⟩

⟨function cmdsrc 51b⟩

⟨global badwrite 52b⟩

⟨function cmdwrite 52c⟩

⟨function regname 101a⟩

⟨function shell 74d⟩
```

D.1.14 db/runpcs.c

⟨db/runpcs.c 102a⟩≡

```
/*
 *
 * debugger
 *
 */

#include "defs.h"
#include "fns.h"
```

⟨*global* bpin 60e⟩

⟨*function* runpcs 54g⟩

⟨*function* endpcs 60c⟩

⟨*function* setup 56d⟩

⟨*function* execbkpt 64b⟩

⟨*function* scanbkpt 61f⟩

⟨*function* delbp 64a⟩

⟨*function* setbp 63b⟩

D.1.15 db/pcs.c

⟨db/pcs.c 102b⟩≡

```
/*
 *
 * debugger
 *
 */

#include "defs.h"
#include "fns.h"
```

⟨*global* NOPCS 59d⟩

⟨*function* subpcs 53d⟩

D.1.16 db/main.c

⟨db/main.c 102c⟩≡

```
/*
 * 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)* 18d⟩

⟨*function* `alldigs` 19a⟩

⟨*function* `done` 27d⟩

⟨*function* `fault` 77e⟩

D.2 tracers/

D.2.1 tracers/ktrace.c

⟨*global* `fhdr` 103a⟩≡ (108)
```
static Fhdr fhdr;
```

⟨*global* `interactive` 103b⟩≡ (108)
```
static int interactive = 0;
```

⟨*constant FRAMENAME (ktrace)* 103c⟩≡ (108)
```
#define FRAMENAME ".frame"
```

⟨*function* `usage` 103d⟩≡ (108)
```
static void
usage(void)
{
    fprint(2, "usage: ktrace [-i] kernel pc sp [link]\n");
    exits("usage");
}
```

⟨*function* `printaddr` 103e⟩≡ (108)
```
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 104a⟩≡ (108)

```
static void (*fmt)(char*, uvlong) = printaddr;
```

⟨*function* main 104b⟩≡ (108)

```
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 105a⟩≡ (108)

```
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");
}
```

⟨*function* rtrace 105b⟩≡ (108)

```
// 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;
}
```

⟨*function i386trace (ktrace)* 106a⟩≡ (108)

```
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;
}
```

⟨*global* naddr 106b⟩≡ (108)

```
int naddr;
```

⟨*global* addr 106c⟩≡ (108)

```
uvlong addr[1024];
```

⟨*global* val 106d⟩≡ (108)

```
uvlong val[1024];
```

⟨*function* putval 107a⟩≡ (108)

```
static void
putval(uvlong a, uvlong v)
{
    if(naddr < nelem(addr)){
        addr[naddr] = a;
        val[naddr] = v;
        naddr++;
    }
}
```

⟨*function* readstack 107b⟩≡ (108)

```
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));
            }
        }
    }
}
```

⟨*function* getval 107c⟩≡ (108)

```
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;
}
```

⟨*function* fatal 107d⟩≡ (108)

```
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 108⟩≡

```
#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` 103a⟩
⟨*global* `interactive` 103b⟩

⟨*constant FRAMENAME (ktrace)* 103c⟩

⟨*function* `usage` 103d⟩

⟨*function* `printaddr` 103e⟩

⟨*global* `fmt` 104a⟩

⟨*function* `main` 104b⟩

⟨*function* `inithdr` 105a⟩

⟨*function* `rtrace` 105b⟩

⟨*function i386trace (ktrace)* 106a⟩

⟨*global* `naddr` 106b⟩
⟨*global* `addr` 106c⟩
⟨*global* `val` 106d⟩

⟨*function* `putval` 107a⟩

⟨*function* `readstack` 107b⟩

⟨*function* `getval` 107c⟩

⟨*function* `fatal` 107d⟩

D.2.2 tracers/strace.c

⟨*enum* _anon_ (strace) 109a⟩≡ (113a)
```
enum {
    Stacksize = 8*1024,
    Bufsize  = 8*1024,
};
```

⟨*global* out 109b⟩≡ (113a)
```
Channel *out;
```

⟨*global* quit 109c⟩≡ (113a)
```
Channel *quit;
```

⟨*global* forkc 109d⟩≡ (113a)
```
Channel *forkc;
```

⟨*global* nread 109e⟩≡ (113a)
```
int nread = 0;
```

⟨*struct* Str 109f⟩≡ (113a)
```
struct Str {
    char *buf;
    int len;
};
```

⟨*function* die 109g⟩≡ (113a)
```
void
die(char *s)
{
    fprint(2, "%s\n", s);
    exits(s);
}
```

⟨*function* cwrite 109h⟩≡ (113a)
```
void
cwrite(int fd, char *path, char *cmd, int len)
{
    werrstr("");
    if (write(fd, cmd, len) < len) {
        fprint(2, "cwrite: %s: failed writing %d bytes: %r\n",
            path, len);
        sendp(quit, nil);
        threadexits(nil);
    }
}
```

⟨*function* newstr 109i⟩≡ (113a)
```
Str *
newstr(void)
{
    Str *s;

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

⟨*function* reader 110⟩≡ (113a)

```
void
reader(void *v)
{
    int cfd, tfd, forking = 0, exiting, pid, newpid;
    char *ctl, *truss;
    Str *s;
    static char start[] = "start";
    static char waitstop[] = "waitstop";

    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!

    cwrite(cfd, ctl, "startsyscall", 12);
    s = newstr();
    exiting = 0;
    while((s->len = pread(tfd, s->buf, Bufsize - 1, 0)) >= 0){
        if (forking && s->buf[1] == '=' && s->buf[3] != '-') {
            forking = 0;
            newpid = strtol(&s->buf[3], 0, 0);
            sendp(forkc, (void*)newpid);
            procrfork(reader, (void*)newpid, Stacksize, 0);
        }

        /*
         * 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 = 1;
            free(rf);
        } else if (strstr(s->buf, " Exits") != nil)
            exiting = 1;

        sendp(out, s); /* print line from /proc/$child/syscall */
        if (exiting) {
            s = newstr();
            strcpy(s->buf, "\n");
            sendp(out, s);
            break;
        }
```

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

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

⟨*function* `writer` 111a⟩≡ (113a)

```
void
writer(void *)
{
    int newpid;
    Str *s;

    // TODO use better 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(2, "%s", s->buf);
            free(s);
            break;
        case 2:   /* forkc */
            // procrfork(reader, (void*)newpid, Stacksize, 0);
            nread++;
            break;
        }
done:
    exits(nil);
}
```

⟨*function usage (tracers/strace.c)* 111b⟩≡ (113a)

```
void
usage(void)
{
    fprint(2, "Usage: strace [-c cmd [arg...]] | [pid]\n");
```

```
    exits("usage");
}
```

⟨*function* hang 112a⟩≡ (113a)

```
void
hang(void)
{
    int 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);
}
```

⟨*function* threadmain 112b⟩≡ (113a)

```
void
threadmain(int argc, char **argv)
{
    int pid;
    char *cmd = nil;
    char **args = nil;

    /*
     * 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;
    }

    /* run a command? */
    if(cmd) {
        pid = fork();
        if (pid < 0)
            sysfatal("fork failed: %r");
        if(pid == 0) {
            hang();
            exec(cmd, args);
            if(cmd[0] != '/')
                exec(smprint("/bin/%s", cmd), args);
```

```
            sysfatal("exec %s failed: %r", cmd);
        }
    } 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);
}
```

⟨tracers/strace.c 113a⟩≡

```
// 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
// but most people knows better 'strace'
```

⟨*enum* _anon_ (strace) 109a⟩

⟨*global* out 109b⟩
⟨*global* quit 109c⟩
⟨*global* forkc 109d⟩
⟨*global* nread 109e⟩

```
typedef struct Str Str;
```

⟨*struct* Str 109f⟩

⟨*function* die 109g⟩

⟨*function* cwrite 109h⟩

⟨*function* newstr 109i⟩

⟨*function* reader 110⟩

⟨*function* writer 111a⟩

⟨*function usage (tracers/strace.c)* 111b⟩

⟨*function* hang 112a⟩

⟨*function* threadmain 112b⟩

D.3 include/

D.3.1 include/debug/mach.h

⟨*enum* dissembler_type 113b⟩≡ (114b)

```
/* dissembler types */
enum dissembler_type
{
    ANONE = 0,

    AI386,
    AI8086,   /* oh god */
    AARM,
};
```

⟨*enum* object_file_type 114a⟩≡ (114b)

```
/* object file types */
enum object_file_type
{
    Obj386 = 0,   /* .8 */
    ObjArm,   /* .5 */

    Maxobjtype,
};
```

⟨include/debug/mach.h 114b⟩≡

```
/*
 * Architecture-dependent application data
 */
#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
 */

⟨enum executable_type 11b⟩
⟨enum machine_type 12b⟩
⟨enum dissembler_type 113b⟩
⟨enum object_file_type 114a⟩
⟨enum symbol_type 14b⟩

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 13e⟩

⟨struct Symbol 14a⟩

⟨struct Reglist 13a⟩

⟨enum register_flag 13b⟩

⟨struct Mach 12a⟩

extern Mach *mach;   /* Current machine */
```

```
typedef uvlong (*Rgetter)(Map*, char*);
typedef void (*Tracer)(Map*, uvlong, uvlong, Symbol*);
```

⟨*struct* Machdata 13d⟩

⟨*struct* Fhdr 11a⟩

```
extern int asstype; /* dissembler type - machdata.c */
extern Machdata *machdata; /* jump vector - machdata.c */

Map*  attachproc(int, 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);
```

D.3.2 include/bootexec.h

⟨*struct* coffsect 116a⟩≡ (117a)

```
struct coffsect
{
    char name[8];
    ulong phys;
    ulong virt;
    ulong size;
    ulong fptr;
    ulong fptrreloc;
    ulong fptrlineno;
    ulong nrelocnlineno;
    ulong flags;
};
```

⟨*struct* i386exec 116b⟩≡ (117a)

```
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 117a⟩≡
  ⟨*struct* coffsect 116a⟩

  ⟨*struct* i386exec 116b⟩

D.4 libmach/

D.4.1 libmach/5.c

⟨*function* REGOFF*(arm)* 117b⟩≡ (118a)
```
#define REGOFF(x) (ulong) (&((struct Ureg *) 0)->x)
```

⟨*constant* SP*(arm)* 117c⟩≡ (118a)
```
#define SP  REGOFF(r13)
```

⟨*constant* PC*(arm)* 117d⟩≡ (118a)
```
#define PC  REGOFF(pc)
```

⟨*constant* REGSIZE*(arm)* 117e⟩≡ (118a)
```
#define REGSIZE  sizeof(struct Ureg)
```

⟨*global* armreglist*(arm)* 117f⟩≡ (118a)
```
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 }
};
```

⟨*global* marm*(arm)* 117g⟩≡ (118a)
```
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 */
};
```

⟨libmach/5.c 118a⟩≡

```
/*
 * arm definition
 */
#include <u.h>
#include <libc.h>
#include <bio.h>
#include "../../include/arch/arm/ureg.h"
#include <mach.h>

⟨function REGOFF(arm) 117b⟩

⟨constant SP(arm) 117c⟩
⟨constant PC(arm) 117d⟩
⟨constant REGSIZE(arm) 117e⟩
⟨global armreglist(arm) 117f⟩
⟨global marm(arm) 117g⟩
```

D.4.2 libmach/5db.c

⟨global debug (libmach/5db.c)(arm) 118b⟩≡ (141b)

```
static int debug = 0;
```

⟨function BITS(arm) 118c⟩≡ (141b)

```
#define BITS(a, b) ((1<<(b+1))-(1<<a))
```

⟨function LSR(arm) 118d⟩≡ (141b)

```
#define LSR(v, s) ((ulong)(v) >> (s))
```

⟨function ASR(arm) 118e⟩≡ (141b)

```
#define ASR(v, s) ((long)(v) >> (s))
```

⟨function ROR(arm) 118f⟩≡ (141b)

```
#define ROR(v, s) (LSR((v), (s)) | (((v) & ((1 << (s))-1)) << (32 - (s))))
```

⟨struct Instr(arm) 118g⟩≡ (141b)

```
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 */
};
```

⟨*struct* Opcode*(arm)* 119a⟩≡ (141b)

```
struct Opcode
{
    char* o;
    void (*fmt)(Opcode*, Instr*);
    uvlong (*foll)(Map*, Rgetter, Instr*, uvlong);
    char* a;
};
```

⟨*global* FRAMENAME*(arm)* 119b⟩≡ (141b)

```
static char FRAMENAME[] = ".frame";
```

⟨*global* armmach*(arm)* 119c⟩≡ (141b)

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

    leswab,   /* short to local byte order */
    leswal,   /* long to local byte order */
    leswav,   /* long to local byte order */
    risctrace,  /* C traceback */
    riscframe,  /* Frame finder */
    armexcep,   /* print exception */
    0,   /* breakpoint fixup */
    0,   /* single precision float printer */
    0,   /* double precision float printer */
    armfoll,  /* following addresses */
    arminst,  /* print instruction */
    armdas,   /* dissembler */
    arminstlen,  /* instruction size */
};
```

⟨*function* armexcep*(arm)* 119d⟩≡ (141b)

```
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";
    }
}
```

⟨*global* cond*(arm)* 120a⟩≡ (141b)

```
static
char* cond[16] =
{
    "EQ", "NE", "CS", "CC",
    "MI", "PL", "VS", "VC",
    "HI", "LS", "GE", "LT",
    "GT", "LE", 0, "NV"
};
```

⟨*global* shtype*(arm)* 120b⟩≡ (141b)

```
static
char* shtype[4] =
{
    "<<", ">>", "->", "@>"
};
```

⟨*global* hb*(arm)* 120c⟩≡ (141b)

```
static
char *hb[4] =
{
    "???", "HU", "B", "H"
};
```

⟨*global* addsub*(arm)* 120d⟩≡ (141b)

```
static
char* addsub[2] =
{
    "-", "+",
};
```

⟨*function* armclass*(arm)* 120e⟩≡ (141b)

```
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;
}
```

⟨*function* `decode` *(arm)* 123a⟩≡ (141b)

```
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;
}
```

⟨*function* `bprint` *(arm)* 123b⟩≡ (141b)

```
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);
}
```

⟨*function* `plocal` *(arm)* 123c⟩≡ (141b)

```
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;
}
```

⟨*function* gsymoff*(arm)* 124⟩≡ (141b)

```
/*
 * 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);
}
```

⟨*function* armdps*(arm)* 125a⟩≡ (141b)

```
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);
}
```

⟨*function* armdpi*(arm)* 125b⟩≡ (141b)

```
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)* 126a⟩≡ (141b)

```
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)* 126b⟩≡ (141b)

```
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)* 126c⟩≡ (141b)

```
/* 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)* 127a⟩≡ (141b)

```
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)* 127b⟩≡ (141b)

```
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)* 127c⟩≡ (141b)

```
static void
armund(Opcode *o, Instr *i)
{
    format(o->o, i, o->a);
}
```

⟨*function* armcdt*(arm)* 127d⟩≡ (141b)

```
static void
armcdt(Opcode *o, Instr *i)
{
    format(o->o, i, o->a);
}
```

⟨*function* armunk*(arm)* 127e⟩≡ (141b)

```
static void
armunk(Opcode *o, Instr *i)
{
    format(o->o, i, o->a);
}
```

⟨*function* armb*(arm)* 127f⟩≡ (141b)

```
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)* 128a⟩≡ (141b)

```
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)* 128b⟩≡ (141b)

```
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);
}
```

⟨*function* armcondpass*(arm)* 128c⟩≡ (141b)

```
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;
    }
}
```

⟨*function* armshiftval*(arm)* 129⟩≡ (141b)

```
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);
        }
    }
}
```

⟨*function* `nbits` *(arm)* 130a⟩≡ (141b)

```
static int
nbits(ulong v)
{
    int n = 0;
    int i;

    for(i=0; i < 32 ; i++) {
        if(v & 1) ++n;
        v >>= 1;
    }

    return n;
}
```

⟨*function* `armmaddr` *(arm)* 130b⟩≡ (141b)

```
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;
    }
}
```

⟨*function* `armaddr` *(arm)* 130c⟩≡ (141b)

```
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;
    }
}
```

⟨*function* armfadd*(arm)* 131a⟩≡ (141b)

```
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);
}
```

⟨*function* armfbx*(arm)* 131b⟩≡ (141b)

```
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);
}
```

⟨*function* armfmovm*(arm)* 132a⟩≡ (141b)

```
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;
}
```

⟨*function* armfbranch*(arm)* 132b⟩≡ (141b)

```
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;
}
```

⟨*function* armfmov*(arm)* 132c⟩≡ (141b)

```
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;
}
```

⟨*global* opcodes*(arm)* 133⟩≡ (141b)

```
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)",
};
```

⟨*function* gaddr*(arm)* 135⟩≡ (141b)

```
static void
gaddr(Instr *i)
{
    *i->curr++ = '$';
    i->curr += gsymoff(i->curr, i->end-i->curr, i->imm, CANY);
}
```

⟨*global* mode*(arm)* 136a⟩≡ (141b)
```
static char *mode[] = { 0, "IA", "DB", "IB" };
```

⟨*global* pw*(arm)* 136b⟩≡ (141b)
```
static char *pw[] = { "P", "PW", 0, "W" };
```

⟨*global* sw*(arm)* 136c⟩≡ (141b)
```
static char *sw[] = { 0, "W", "S", "SW" };
```

⟨*function* format*(arm)* 136d⟩≡ (141b)
```
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;
}
```

⟨*function* printins*(arm)* 140a⟩≡ (141b)

```
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* arminst*(arm)* 140b⟩≡ (141b)

```
static int
arminst(Map *map, uvlong pc, char modifier, char *buf, int n)
{
    USED(modifier);
    return printins(map, pc, buf, n);
}
```

⟨*function* armdas*(arm)* 140c⟩≡ (141b)

```
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;
}
```

⟨*function* arminstlen*(arm)* 140d⟩≡ (141b)

```
static int
arminstlen(Map *map, uvlong pc)
{
    Instr i;

    if(decode(map, pc, &i) < 0)
        return -1;
    return 4;
}
```

⟨*function* armfoll*(arm)* 141a⟩≡ (141b)

```
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;
}
```

⟨libmach/5db.c 141b⟩≡

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

⟨*global debug (libmach/5db.c)(arm)* 118b⟩

⟨*function* BITS*(arm)* 118c⟩

⟨*function* LSR*(arm)* 118d⟩
⟨*function* ASR*(arm)* 118e⟩
⟨*function* ROR*(arm)* 118f⟩

```
typedef struct Instr Instr;
```
⟨*struct* Instr*(arm)* 118g⟩

```
typedef struct Opcode Opcode;
```
⟨*struct* Opcode*(arm)* 119a⟩

```
static void format(char*, Instr*, char*);
```
⟨*global* FRAMENAME*(arm)* 119b⟩

```
/*
 * 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)* 119c⟩

⟨*function* armexcep*(arm)* 119d⟩

⟨*global* `cond`*(arm)* 120a⟩

⟨*global* `shtype`*(arm)* 120b⟩

⟨*global* `hb`*(arm)* 120c⟩

⟨*global* `addsub`*(arm)* 120d⟩

⟨*function* `armclass`*(arm)* 120e⟩

⟨*function* `decode`*(arm)* 123a⟩

```
#pragma varargck argpos bprint  2
```

⟨*function* `bprint`*(arm)* 123b⟩

⟨*function* `plocal`*(arm)* 123c⟩

⟨*function* `gsymoff`*(arm)* 124⟩

⟨*function* `armdps`*(arm)* 125a⟩

⟨*function* `armdpi`*(arm)* 125b⟩

⟨*function* `armsdti`*(arm)* 126a⟩

⟨*function* `armvstdi`*(arm)* 126b⟩

⟨*function* `armhwby`*(arm)* 126c⟩

⟨*function* `armsdts`*(arm)* 127a⟩

⟨*function* `armbdt`*(arm)* 127b⟩

⟨*function* `armund`*(arm)* 127c⟩

⟨*function* `armcdt`*(arm)* 127d⟩

⟨*function* `armunk`*(arm)* 127e⟩

⟨*function* `armb`*(arm)* 127f⟩

⟨*function* `armbpt`*(arm)* 128a⟩

⟨*function* `armco`*(arm)* 128b⟩

⟨*function* `armcondpass`*(arm)* 128c⟩

⟨*function* `armshiftval`*(arm)* 129⟩

⟨*function* `nbits`*(arm)* 130a⟩

⟨*function* `armmaddr`*(arm)* 130b⟩

⟨*function* `armaddr`*(arm)* 130c⟩

⟨*function* `armfadd`*(arm)* 131a⟩

⟨*function* `armfbx`*(arm)* 131b⟩

⟨*function* armfmovm*(arm)* 132a⟩

⟨*function* armfbranch*(arm)* 132b⟩

⟨*function* armfmov*(arm)* 132c⟩

⟨*global* opcodes*(arm)* 133⟩

⟨*function* gaddr*(arm)* 135⟩

⟨*global* mode*(arm)* 136a⟩
⟨*global* pw*(arm)* 136b⟩
⟨*global* sw*(arm)* 136c⟩

⟨*function* format*(arm)* 136d⟩

⟨*function* printins*(arm)* 140a⟩

⟨*function* arminst*(arm)* 140b⟩

⟨*function* armdas*(arm)* 140c⟩

⟨*function* arminstlen*(arm)* 140d⟩

⟨*function* armfoll*(arm)* 141a⟩

D.4.3 libmach/5obj.c

⟨*struct* Addr*(arm)* 143a⟩≡ (145c)
```
struct Addr
{
    char type;
    char sym;
    char name;
};
```

⟨*function* _is5*(arm)* 143b⟩≡ (145c)
```
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)* 143c⟩≡ (145c)
```
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;
}
```

⟨*function* addr*(arm)* 144⟩≡ (145c)

```
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;
}
```

⟨*function* type2char*(arm)* 145a⟩≡ (145c)

```
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;
    }
}
```

⟨*function* skip*(arm)* 145b⟩≡ (145c)

```
static void
skip(Biobuf *bp, int n)
{
    while (n-- > 0)
        Bgetc(bp);
}
```

⟨libmach/5obj.c 145c⟩≡

```
/*
 * 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) 143a⟩
static Addr addr(Biobuf*);
static char type2char(int);
static void skip(Biobuf*, int);

⟨function _is5(arm) 143b⟩
```

⟨*function* `_read5` *(arm)* 143c⟩

⟨*function* `addr` *(arm)* 144⟩

⟨*function* `type2char` *(arm)* 145a⟩

⟨*function* `skip` *(arm)* 145b⟩

D.4.4 libmach/elf.h

⟨*constant* `LOAD` 146a⟩≡ (146b)
```
//TODO: could remove this file?
#define LOAD PT_LOAD
```

⟨`libmach/elf.h` 146b⟩≡
```
// see include/elf.h
#include <elf.h>
```

⟨*constant* `LOAD` 146a⟩

D.4.5 libmach/obj.h

⟨*struct Prog (libmach/obj.h)* 146c⟩≡ (146e)
```
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` 146d⟩≡ (146e)
```
#define UNKNOWN '?'
```

⟨`libmach/obj.h` 146e⟩≡
```
/*
 * 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)* 146c⟩

⟨*constant* `UNKNOWN` 146d⟩

```
void  _offset(int, vlong);
```

D.4.6 libmach/swap.c

⟨*function* beswab 147a⟩≡ (148b)
```
/*
 * big-endian short
 */
ushort
beswab(ushort s)
{
    uchar *p;

    p = (uchar*)&s;
    return (p[0]<<8) | p[1];
}
```

⟨*function* beswal 147b⟩≡ (148b)
```
/*
 * 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 147c⟩≡ (148b)
```
/*
 * 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 147d⟩≡ (148b)
```
/*
 * little-endian short
 */
ushort
leswab(ushort s)
{
    uchar *p;

    p = (uchar*)&s;
    return (p[1]<<8) | p[0];
}
```

⟨*function* leswal 147e⟩≡ (148b)
```
/*
 * 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 148a⟩≡ (148b)

```
/*
 * 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 148b⟩≡

```
#include <u.h>

⟨function beswab 147a⟩

⟨function beswal 147b⟩

⟨function beswav 147c⟩

⟨function leswab 147d⟩

⟨function leswal 147e⟩

⟨function leswav 148a⟩
```

D.4.7 libmach/executable.c

⟨*struct* Exectable 148c⟩≡ (154b)

```
/*
 * 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;
```

⟨*global* `exectab` 149a⟩≡ (154b)

```
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 },

    { 0 },
};
```

⟨*function* `hswal` 149b⟩≡ (154b)

```
/*
 * 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` 149c⟩≡ (154b)

```
/*
 * 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;
}
```

⟨*function* commonboot 150a⟩≡ (154b)

```
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 */
}
```

⟨*function* common 150b⟩≡ (154b)

```
/*
 * _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;
}
```

⟨*function* elf32dotout 150c⟩≡ (154b)

```
/*
 * 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;
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;
}
```

⟨*function* elfdotout 152⟩≡ (154b)

```
/*
 * 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;
}
```

⟨*function* armdotout 153a⟩≡ (154b)

```
/*
 * (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;
}
```

⟨*function* settext 153b⟩≡ (154b)

```
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 153c⟩≡ (154b)

```
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 153d⟩≡ (154b)

```
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 154a⟩≡ (154b)

```
static uvlong
_round(uvlong a, ulong b)
{
    uvlong w;

    w = (a/b)*b;
    if (a!=w)
        w += b;
    return(w);
}
```

⟨libmach/executable.c 154b⟩≡

```
#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 */
    } 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 148c⟩
```

```
//PAD: removed many archi
extern Mach mi386;
extern Mach marm;
```

⟨*global* `exectab` 149a⟩

⟨*global* `mach` 13c⟩

⟨*function* `crackhdr` 28⟩

⟨*function* `hswal` 149b⟩

⟨*function* `adotout` 149c⟩

⟨*function* `commonboot` 150a⟩

⟨*function* `common` 150b⟩

⟨*function* `elf32dotout` 150c⟩

⟨*function* `elfdotout` 152⟩

⟨*function* `armdotout` 153a⟩

⟨*function* `settext` 153b⟩

⟨*function* `setdata` 153c⟩

⟨*function* `setsym` 153d⟩

⟨*function* `_round` 154a⟩

D.4.8 `libmach/map.c`

⟨*function* `newmap` 155a⟩≡ (158b)

```
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` 155b⟩≡ (158b)

```
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` 156a⟩≡ (158b)

```
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* `attachproc` 156b⟩≡ (158b)

```
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;
}
```

⟨*function* `findseg` 157a⟩≡ (158b)

```
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;
}
```

⟨*function* `unusemap` 157b⟩≡ (158b)

```
void
unusemap(Map *map, int i)
```

```
{
    if (map != 0 && 0 <= i && i < map->nsegs)
        map->seg[i].inuse = 0;
}
```

⟨*function* loadmap 158a⟩≡ (158b)

```
Map*
loadmap(Map *map, int fd, Fhdr *fp)
{
    map = newmap(map, 2);
    if (map == 0)
        return 0;

    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;
}
```

⟨libmach/map.c 158b⟩≡

```
/*
 * file map routines
 */
#include <u.h>
#include <libc.h>
#include <bio.h>
#include <mach.h>

⟨function newmap 155a⟩

⟨function setmap 155b⟩

⟨function stacktop 156a⟩

⟨function attachproc 156b⟩

⟨function findseg 157a⟩

⟨function unusemap 157b⟩

⟨function loadmap 158a⟩
```

D.4.9 libmach/sym.c

⟨*constant* HUGEINT 158c⟩≡ (182c)

```
#define HUGEINT 0x7fffffff
```

⟨*constant* NNAME 158d⟩≡ (182c)

```
#define NNAME 20  /* a relic of the past */
```

⟨*struct* `txtsym` 159a⟩≡ (182c)
```
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` 159b⟩≡ (182c)
```
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` 159c⟩≡ (182c)
```
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 */
};
```

⟨*global debug (libmach/sym.c)* 159d⟩≡ (182c)
```
static int debug = 0;
```

⟨*global* `autos` 159e⟩≡ (182c)
```
static Sym **autos;  /* Base of auto variables */
```

⟨*global* `files` 159f⟩≡ (182c)
```
static File *files;   /* Base of file arena */
```

⟨*global* `fmax` 159g⟩≡ (182c)
```
static int fmax;   /* largest file path index */
```

⟨*global fnames (libmach/sym.c)* 159h⟩≡ (182c)
```
static Sym **fnames;  /* file names path component table */
```

⟨*global* `globals` 159i⟩≡ (182c)
```
static Sym **globals;  /* globals by addr table */
```

⟨*global* `hist` 159j⟩≡ (182c)
```
static Hist *hist;   /* base of history stack */
```

⟨*global* `isbuilt` 159k⟩≡ (182c)
```
static int isbuilt;  /* internal table init flag */
```

⟨*global* `nauto` 159l⟩≡ (182c)
```
static long nauto;   /* number of automatics */
```

⟨*global* `nfiles` 159m⟩≡ (182c)
```
static long nfiles;   /* number of files */
```

⟨*global* `nglob` 159n⟩≡ (182c)
```
static long nglob;   /* number of globals */
```

⟨*global* `nhist` 159o⟩≡ (182c)
```
static long nhist;   /* number of history stack entries */
```

⟨*global nsym (libmach/sym.c)* 160a⟩≡ (182c)
```
static long nsym;   /* number of symbols */
```

⟨*global* ntxt 160b⟩≡ (182c)
```
static int ntxt;   /* number of text symbols */
```

⟨*global* pcline 160c⟩≡ (182c)
```
static uchar *pcline;  /* start of pc-line state table */
```

⟨*global* pclineend 160d⟩≡ (182c)
```
static uchar  *pclineend;  /* end of pc-line table */
```

⟨*global* spoff 160e⟩≡ (182c)
```
static uchar *spoff;   /* start of pc-sp state table */
```

⟨*global* spoffend 160f⟩≡ (182c)
```
static uchar *spoffend;  /* end of pc-sp offset table */
```

⟨*global* symbols 160g⟩≡ (182c)
```
static Sym *symbols;  /* symbol table */
```

⟨*global* txt 160h⟩≡ (182c)
```
static Txtsym *txt;   /* Base of text symbol table */
```

⟨*global* txtstart 160i⟩≡ (182c)
```
static uvlong txtstart;  /* start of text segment */
```

⟨*global* txtend 160j⟩≡ (182c)
```
static uvlong txtend;   /* end of text segment */
```

⟨*function* syminit 160k⟩≡ (182c)
```
/*
 * 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;
}
```

⟨*function* symerrmsg 162a⟩≡ (182c)

```
static int
symerrmsg(int n, char *table)
{
    werrstr("can't read %d bytes of %s table", n, table);
    return -1;
}
```

⟨*function* decodename 162b⟩≡ (182c)

```
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;
}
```

⟨*function* cleansyms 163⟩≡ (182c)

```
/*
 * 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;
}
```

⟨*function* textseg 164a⟩≡ (182c)

```
/*
 * delimit the text segment
 */
void
textseg(uvlong base, Fhdr *fp)
{
    txtstart = base;
    txtend = base+fp->txtsz;
}
```

⟨*function* symbase 164b⟩≡ (182c)

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

⟨*function* getsym 164c⟩≡ (182c)

```
/*
 * Get the ith symbol table entry
 */
Sym *
getsym(int index)
{
```

```
    if(index >= 0 && index < nsym)
        return &symbols[index];
    return 0;
}
```

⟨*function* buildtbls 165⟩≡ (182c)

```
/*
 * 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;
}
```

⟨*function lookup (libmach/sym.c)* 167⟩≡ (182c)

```
/*
 * 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 */

}
```

⟨*function* `findtext` 168a⟩≡ (182c)

```
/*
 * 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;
}
```

⟨*function* `findglobal` 168b⟩≡ (182c)

```
/*
 * 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;
}
```

⟨*function* `findlocal` 168c⟩≡ (182c)

```
/*
 * 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);
}
```

⟨*function* `findlocvar` 169a⟩≡ (182c)

```
/*
 * 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;
}
```

⟨*function* `textsym` 169b⟩≡ (182c)

```
/*
 * 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;
}
```

⟨*function* `filesym` 169c⟩≡ (182c)

```
/*
 * 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);
}
```

⟨*function* getauto 170a⟩≡ (182c)

```
/*
 * 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;
}
```

⟨*function* srchtext 170b⟩≡ (182c)

```
/*
 * 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;
}
```

⟨*function* srchdata 171a⟩≡ (182c)

```
/*
 * 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;
}
```

⟨*function* findsym 171b⟩≡ (182c)

```
/*
 * 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;
}
```

⟨*function* `fnbound` 172a⟩≡ (182c)

```
/*
 * 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;
}
```

⟨*function* `localsym` 172b⟩≡ (182c)

```
/*
 * 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;
}
```

⟨*function* globalsym 173a⟩≡ (182c)

```
/*
 * 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;
}
```

⟨*function* file2pc 173b⟩≡ (182c)

```
/*
 * 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;
}
```

⟨*function* pathcomp 174a⟩≡ (182c)

```
/*
 * 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;
}
```

⟨*function* encfname 174b⟩≡ (182c)

```
/*
 * 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;
}
```

⟨*function* hline 175a⟩≡ (182c)

```
/*
 * 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 175b⟩≡ (182c)

```
/*
```

```
 * 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` 176a⟩≡ (182c)

```
/*
 * 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;
}
```

⟨*function* `fline` 176b⟩≡ (182c)

```
/*
 * 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;
}
```

⟨*function* `fileelem` 178a⟩≡ (182c)

```
/*
 * 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` 178b⟩≡ (182c)

```
/*
 * 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` 178c⟩≡ (182c)

```
/*
 * 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` 179a⟩≡ (182c)

```
/*
 * 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` 179b⟩≡ (182c)

```
/*
 * 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` 179c⟩≡ (182c)

```
/*
 * 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;
}
```

⟨*function* pc2line 180⟩≡ (182c)

```
/*
 * 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;
}
```

⟨*function* line2addr 181⟩≡ (182c)

```
/*
 * 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;
}
```

⟨*function* printhist 182a⟩≡ (182c)

```
/*
 * 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++;
    }
}
```

⟨*function* dumphist 182b⟩≡ (182c)

```
/*
 * 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 182c⟩≡

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

```
#include <bio.h>
#include <mach.h>
```

⟨*constant* HUGEINT 158c⟩
⟨*constant* NNAME 158d⟩

```
typedef struct txtsym Txtsym;
typedef struct file File;
typedef struct hist Hist;
```

⟨*struct* txtsym 159a⟩

⟨*struct* hist 159b⟩

⟨*struct* file 159c⟩

⟨*global debug (libmach/sym.c)* 159d⟩

⟨*global* autos 159e⟩
⟨*global* files 159f⟩
⟨*global* fmax 159g⟩
⟨*global fnames (libmach/sym.c)* 159h⟩
⟨*global* globals 159i⟩
⟨*global* hist 159j⟩
⟨*global* isbuilt 159k⟩
⟨*global* nauto 159l⟩
⟨*global* nfiles 159m⟩
⟨*global* nglob 159n⟩
⟨*global* nhist 159o⟩
⟨*global nsym (libmach/sym.c)* 160a⟩
⟨*global* ntxt 160b⟩
⟨*global* pcline 160c⟩
⟨*global* pclineend 160d⟩
⟨*global* spoff 160e⟩
⟨*global* spoffend 160f⟩
⟨*global* symbols 160g⟩
⟨*global* txt 160h⟩
⟨*global* txtstart 160i⟩
⟨*global* txtend 160j⟩

```
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 160k⟩

⟨*function* symerrmsg 162a⟩

⟨*function* `decodename` 162b⟩

⟨*function* `cleansyms` 163⟩

⟨*function* `textseg` 164a⟩

⟨*function* `symbase` 164b⟩

⟨*function* `getsym` 164c⟩

⟨*function* `buildtbls` 165⟩

⟨*function* *lookup (libmach/sym.c)* 167⟩

⟨*function* `findtext` 168a⟩
⟨*function* `findglobal` 168b⟩

⟨*function* `findlocal` 168c⟩

⟨*function* `findlocvar` 169a⟩

⟨*function* `textsym` 169b⟩

⟨*function* `filesym` 169c⟩

⟨*function* `getauto` 170a⟩

⟨*function* `srchtext` 170b⟩

⟨*function* `srchdata` 171a⟩

⟨*function* `findsym` 171b⟩

⟨*function* `fnbound` 172a⟩

⟨*function* `localsym` 172b⟩

⟨*function* `globalsym` 173a⟩

⟨*function* `file2pc` 173b⟩

⟨*function* `pathcomp` 174a⟩

⟨*function* `encfname` 174b⟩

⟨*function* `hline` 175a⟩

⟨*function* `hcomp` 175b⟩

⟨*function* `fileline` 176a⟩

⟨*function* `fline` 176b⟩

⟨*function* `fileelem` 178a⟩

⟨*function* `symcomp` 178b⟩

⟨*function* `txtcomp` 178c⟩

⟨*function* `filecomp` 179a⟩

⟨*function* `fillsym` 179b⟩

⟨*function* `pc2sp` 179c⟩

⟨*function* `pc2line` 180⟩

⟨*function* `line2addr` 181⟩

⟨*function* `printhist` 182a⟩

```
#ifdef DEBUG
⟨function dumphist 182b⟩
#endif
```

D.4.10 `libmach/access.c`

⟨*function* `geta` 185a⟩≡ (189c)

```
/*
 * 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;
}
```

⟨*function* `get8` 185b⟩≡ (189c)

```
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;
```

```
}
```

⟨*function* get4 186a⟩≡ (189c)

```
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;
}
```

⟨*function* get2 186b⟩≡ (189c)

```
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;
}
```

⟨*function* get1 186c⟩≡ (189c)

```
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;
}
```

⟨*function* puta 187a⟩≡ (189c)

```
int
puta(Map *map, uvlong addr, uvlong v)
{
    if (mach->szaddr == 8)
        return put8(map, addr, v);

    return put4(map, addr, v);
}
```

⟨*function* put8 187b⟩≡ (189c)

```
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);
}
```

⟨*function put4 (libmach/access.c)* 187c⟩≡ (189c)

```
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);
}
```

⟨*function* put2 187d⟩≡ (189c)

```
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);
}
```

⟨*function* put1 187e⟩≡ (189c)

```
int
put1(Map *map, uvlong addr, uchar *v, int size)
{
    if (!map) {
        werrstr("put1: invalid map");
        return -1;
    }
    return mput(map, addr, v, size);
}
```

⟨*function* spread 188a⟩≡ (189c)

```
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 188b⟩≡ (189c)

```
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;
}
```

⟨*function* mput 189a⟩≡ (189c)

```
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;
}
```

⟨*function* reloc 189b⟩≡ (189c)

```
/*
 * 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 189c⟩≡

```
/*
 * 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` 185a⟩

⟨*function* `get8` 185b⟩

⟨*function* `get4` 186a⟩

⟨*function* `get2` 186b⟩

⟨*function* `get1` 186c⟩

⟨*function* `puta` 187a⟩

⟨*function* `put8` 187b⟩

⟨*function put4 (libmach/access.c)* 187c⟩

⟨*function* `put2` 187d⟩

⟨*function* `put1` 187e⟩

⟨*function* `spread` 188a⟩

⟨*function* `mget` 188b⟩

⟨*function* `mput` 189a⟩

⟨*function* `reloc` 189b⟩

D.4.11 `libmach/machdata.c`

⟨*constant* `STARTSYM` 190a⟩≡ (198)
```
#define STARTSYM "_main"
```

⟨*constant* `PROFSYM` 190b⟩≡ (198)
```
#define PROFSYM  "_mainp"
```

⟨*constant* `FRAMENAME` 190c⟩≡ (198)
```
#define FRAMENAME ".frame"
```

⟨*global* `asstype` 190d⟩≡ (198)
```
int asstype = AARM;  /* disassembler type */
```

⟨*global* `machdata` 190e⟩≡ (198)
```
Machdata *machdata;  /* machine-dependent functions */
```

⟨*function* `localaddr` 190f⟩≡ (198)
```
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;
}
```

⟨*function* symoff 191⟩≡ (198)

```
/*
 * 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);
}
```

⟨*function* fpformat 192⟩≡ (198)

```
/*
 * 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;
}
```

⟨*function* _hexify 193a⟩≡ (198)

```
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;
}
```

⟨*function* ieeedftos 193b⟩≡ (198)

```
/*
 * 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 194a⟩≡ (198)

```
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 194b⟩≡ (198)

```
int
beieeesftos(char *buf, int n, void *s)
{
    return ieeesftos(buf, n, beswal(*(ulong*)s));
}
```

⟨*function* beieeedftos 194c⟩≡ (198)

```
int
beieeedftos(char *buf, int n, void *s)
{
    return ieeedftos(buf, n, beswal(*(ulong*)s), beswal(((ulong*)(s))[1]));
}
```

⟨*function* leieeesftos 195a⟩≡ (198)

```
int
leieeesftos(char *buf, int n, void *s)
{
    return ieeesftos(buf, n, leswal(*(ulong*)s));
}
```

⟨*function* leieeedftos 195b⟩≡ (198)

```
int
leieeedftos(char *buf, int n, void *s)
{
    return ieeedftos(buf, n, leswal(((ulong*)(s))[1]), leswal(*(ulong*)s));
}
```

⟨*function* beieee80ftos 195c⟩≡ (198)

```
/* 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);
}
```

⟨*function* `leieee80ftos` 196a⟩≡ (198)

```
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);
}
```

⟨*function* `cisctrace` 196b⟩≡ (198)

```
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;
}
```

⟨*function* `risctrace` 196c⟩≡ (198)

```
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;
}
```

⟨*function* ciscframe 197a⟩≡ (198)

```
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;
}
```

⟨*function* riscframe 197b⟩≡ (198)

```
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;
}
```

⟨libmach/machdata.c 198⟩≡

```
/*
 * Debugger utilities shared by at least two architectures
 */

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

⟨*constant* STARTSYM 190a⟩
⟨*constant* PROFSYM 190b⟩
⟨*constant* FRAMENAME 190c⟩

```
extern Machdata mipsmach;
```

⟨*global* asstype 190d⟩
⟨*global* machdata 190e⟩

⟨*function* localaddr 190f⟩

⟨*function* symoff 191⟩

⟨*function* fpformat 192⟩

⟨*function* _hexify 193a⟩

⟨*function* ieeedftos 193b⟩

⟨*function* ieeesftos 194a⟩

⟨*function* beieeesftos 194b⟩

⟨*function* beieeedftos 194c⟩

⟨*function* leieeesftos 195a⟩

⟨*function* leieeedftos 195b⟩

⟨*function* `beieee80ftos` 195c⟩

⟨*function* `leieee80ftos` 196a⟩

⟨*function* `cisctrace` 196b⟩

⟨*function* `risctrace` 196c⟩

⟨*function* `ciscframe` 197a⟩

⟨*function* `riscframe` 197b⟩

D.4.12 `libmach/obj.c`

⟨*function* `islocal` 199a⟩≡ (204b)
```
#define islocal(t) ((t)=='a' || (t)=='p')
```

⟨*enum* `_anon_ (libmach/obj.c)` 199b⟩≡ (204b)
```
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` 199c⟩≡ (204b)
```
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` 199d⟩≡ (204b)
```
static Obj obj[] =
{   /* functions to identify and parse each type of obj */
    [ObjArm] "arm .5", _is5, _read5,
    [Obj386] "386 .8", _is8, _read8,
    [Maxobjtype] 0, 0
};
```

⟨*struct* `Symtab` 199e⟩≡ (204b)
```
struct Symtab
{
    struct Sym  s;
    struct Symtab *next;
};
```

⟨*global hash (libmach/obj.c)* 199f⟩≡ (204b)
```
static Symtab *hash[NHASH];
```

⟨*global* `names` 199g⟩≡ (204b)
```
static Sym *names[NNAMES]; /* working set of active names */
```

⟨*function* objtype 200a⟩≡ (204b)

```
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;
}
```

⟨*function* isar 200b⟩≡ (204b)

```
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 200c⟩≡ (204b)

```
/*
 * 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;
}
```

⟨*function* readar 200d⟩≡ (204b)

```
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;
}
```

⟨*function* processprog 201a⟩≡ (204b)

```
/*
 * 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;
}
```

⟨*function* objlookup 201b⟩≡ (204b)

```
/*
 * 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;
}
```

⟨*function* objtraverse 202⟩≡ (204b)

```
/*
 * 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);
```

```
}
```

⟨*function* `_offset` 203a⟩≡ (204b)

```
/*
 * 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;
}
```

⟨*function* `objupdate` 203b⟩≡ (204b)

```
/*
 * 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);
}
```

⟨*function* `nextar` 203c⟩≡ (204b)

```
/*
 * 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` 204a⟩≡ (204b)

```
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);
}
```

⟨`libmach/obj.c` 204b⟩≡

```
/*
 * 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` 199a⟩

⟨*enum* `_anon_ (libmach/obj.c)` 199b⟩

```
int   /* in [$OS].c */ //$
  _is5(char*),
  _is8(char*),
  _read5(Biobuf*, Prog*),
  _read8(Biobuf*, Prog*);


typedef struct Obj Obj;
typedef struct Symtab Symtab;
```

⟨*struct* `Obj` 199c⟩

⟨*global* `obj` 199d⟩

⟨*struct* `Symtab` 199e⟩

⟨*global hash (libmach/obj.c)* 199f⟩
⟨*global* `names` 199g⟩

```
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` 200a⟩

⟨*function* `isar` 200b⟩

⟨*function* `readobj` 200c⟩

⟨*function* `readar` 200d⟩

⟨*function* `processprog` 201a⟩

⟨*function* `objlookup` 201b⟩
⟨*function* `objtraverse` 202⟩

⟨*function* `_offset` 203a⟩

⟨*function* `objupdate` 203b⟩

⟨*function* `nextar` 203c⟩

⟨*function* `objreset` 204a⟩

D.4.13 `libmach/setmach.c`

⟨*struct* `machtab` 205a⟩≡ (206b)
```
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` 205b⟩≡ (206b)
```
/*
 * 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, },
    { 0  },  /*the terminator*/
};
```

⟨*function* `machbytype` 205c⟩≡ (206b)
```
/*
 * 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;
        }
    }
}
```

⟨*function* machbyname 206a⟩≡ (206b)

```
/*
 * select a machine by name
 */
int
machbyname(char *name)
{
    Machtab *mp;

    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;
}
```

⟨libmach/setmach.c 206b⟩≡

```
#include <u.h>
#include <libc.h>
#include <bio.h>
#include <mach.h>
        /* table for selecting machine-dependent parameters */

typedef struct machtab Machtab;

⟨struct machtab 205a⟩

extern Mach  mi386, marm;
extern Machdata i386mach, armmach;

⟨global machines 205b⟩

⟨function machbytype 205c⟩
⟨function machbyname 206a⟩
```

D.5 acid/

D.5.1 acid/acid.h

⟨*enum* _anon_ (acid/acid.h) 207a⟩≡ (210)
```
/* acid.h */
enum
{
    Eof  = -1,
    Strsize  = 4096,
    Hashsize = 128,
    Maxarg  = 512,
    NFD  = 100,
    Maxproc  = 50,
    Maxval  = 10,
    Mempergc = 1024*1024,
};
```

⟨*macro* expr 207b⟩≡ (210)
```
#define expr(n, r) do{(r)->comt=0; (*expop[(n)->op])(n, r);}while(0)
```

⟨*enum* _anon_ (acid/acid.h)*2* 207c⟩≡ (210)
```
enum
{
    TINT,
    TFLOAT,
    TSTRING,
    TLIST,
    TCODE,
};
```

⟨*struct* Type 207d⟩≡ (210)
```
struct Type
{
    Type* next;
    int offset;
    char fmt;
    char depth;
    Lsym* type;
    Lsym* tag;
    Lsym* base;
};
```

⟨*struct* Frtype 207e⟩≡ (210)
```
struct Frtype
{
    Lsym* var;
    Type* type;
    Frtype* next;
};
```

⟨*struct* Ptab 207f⟩≡ (210)
```
struct Ptab
{
    int pid;
    int ctl;
};
```

⟨*struct* Rplace 208a⟩≡ (210)

```
struct Rplace
{
    jmp_buf rlab;
    Node* stak;
    Node* val;
    Lsym* local;
    Lsym** tail;
};
```

⟨*struct* Gc 208b⟩≡ (210)

```
struct Gc
{
    char gcmark;
    Gc* gclink;
};
```

⟨*struct* Store 208c⟩≡ (210)

```
struct Store
{
    char fmt;
    Type* comt;
    union {
        vlong ival;
        double fval;
        String* string;
        List* l;
        Node* cc;
    };
};
```

⟨*struct* List 208d⟩≡ (210)

```
struct List
{
    Gc;
    List* next;
    char type;
    Store;
};
```

⟨*struct* Value 208e⟩≡ (210)

```
struct Value
{
    char set;
    char type;
    Store;
    Value* pop;
    Lsym* scope;
    Rplace* ret;
};
```

⟨*struct* Lsym 208f⟩≡ (210)

```
struct Lsym
{
    char* name;
    int lexval;
    Lsym* hash;
    Value* v;
    Type* lt;
    Node* proc;
```

```
    Frtype* local;
    void (*builtin)(Node*, Node*);
};
```

⟨*struct* Node 209a⟩≡ (210)
```
struct Node
{
    Gc;
    char op;
    char type;
    Node* left;
    Node* right;
    Lsym* sym;
    int builtin;
    Store;
};
```

⟨*constant* ZN 209b⟩≡ (210)
```
#define ZN (Node*)0
```

⟨*struct* StringAcid 209c⟩≡ (210)
```
struct StringAcid
{
    Gc;
    char *string;
    int len;
};
```

⟨*enum* _anon_ (acid/acid.h)*3* 209d⟩≡ (210)
```
enum
{
    ONAME,
    OCONST,
    OMUL,
    ODIV,
    OMOD,
    OADD,
    OSUB,
    ORSH,
    OLSH,
    OLT,
    OGT,
    OLEQ,
    OGEQ,
    OEQ,
    ONEQ,
    OLAND,
    OXOR,
    OLOR,
    OCAND,
    OCOR,
    OASGN,
    OINDM,
    OEDEC,
    OEINC,
    OPINC,
    OPDEC,
    ONOT,
    OIF,
    ODO,
```

```
    OLIST,
    OCALL,
    OCTRUCT,
    OWHILE,
    OELSE,
    OHEAD,
    OTAIL,
    OAPPEND,
    ORET,
    OINDEX,
    OINDC,
    ODOT,
    OLOCAL,
    OFRAME,
    OCOMPLEX,
    ODELETE,
    OCAST,
    OFMT,
    OEVAL,
    OWHAT,
};
```

⟨acid/acid.h 210⟩≡

⟨*enum* _anon_ (acid/acid.h) 207a⟩

```
#pragma varargck type "L" void

typedef struct Node Node;
typedef struct StringAcid 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 int 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 207b⟩
extern int fmtsize(Value *v) ;

⟨enum _anon_ (acid/acid.h)2 207c⟩

⟨struct Type 207d⟩

⟨struct Frtype 207e⟩

⟨struct Ptab 207f⟩

extern Ptab ptab[Maxproc];

⟨struct Rplace 208a⟩

⟨struct Gc 208b⟩

⟨struct Store 208c⟩

⟨struct List 208d⟩

⟨struct Value 208e⟩

⟨struct Lsym 208f⟩

⟨struct Node 209a⟩
⟨constant ZN 209b⟩

⟨struct StringAcid 209c⟩

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* 209d⟩

D.5.2 acid/globals.c

⟨*global* kernel 212⟩≡ (214h)

```
int kernel;
```

⟨*global* `remote` 213a⟩≡ (214h)
```
int remote;
```

⟨*global* `text` 213b⟩≡ (214h)
```
int text;
```

⟨*global* `silent` 213c⟩≡ (214h)
```
int silent;
```

⟨*global fhdr (acid/globals.c)* 213d⟩≡ (214h)
```
Fhdr fhdr;
```

⟨*global line (acid/globals.c)* 213e⟩≡ (214h)
```
int line;
```

⟨*global* `bout` 213f⟩≡ (214h)
```
Biobuf* bout;
```

⟨*global* `io` 213g⟩≡ (214h)
```
Biobuf* io[32];
```

⟨*global* `iop` 213h⟩≡ (214h)
```
int iop;
```

⟨*global* `symbol` 213i⟩≡ (214h)
```
char symbol[Strsize];
```

⟨*global interactive (acid/globals.c)* 213j⟩≡ (214h)
```
int interactive;
```

⟨*global* `na` 213k⟩≡ (214h)
```
int na;
```

⟨*global wtflag (acid/globals.c)* 213l⟩≡ (214h)
```
int wtflag;
```

⟨*global cormap (acid/globals.c)* 213m⟩≡ (214h)
```
Map* cormap;
```

⟨*global symmap (acid/globals.c)* 213n⟩≡ (214h)
```
Map* symmap;
```

⟨*global* `hash` 213o⟩≡ (214h)
```
Lsym* hash[Hashsize];
```

⟨*global* `dogc` 213p⟩≡ (214h)
```
long dogc;
```

⟨*global* `ret` 213q⟩≡ (214h)
```
Rplace* ret;
```

⟨*global* `aout` 213r⟩≡ (214h)
```
char* aout;
```

⟨*global* `gotint` 213s⟩≡ (214h)
```
int gotint;
```

⟨*global* `gcl` 213t⟩≡ (214h)
```
Gc* gcl;
```

⟨*global* `stacked` 214a⟩≡ (214h)
```
int stacked;
```

⟨*global* `err` 214b⟩≡ (214h)
```
jmp_buf err;
```

⟨*global* `prnt` 214c⟩≡ (214h)
```
Node* prnt;
```

⟨*global* `tracelist` 214d⟩≡ (214h)
```
List* tracelist;
```

⟨*global* `initialising` 214e⟩≡ (214h)
```
int initialising;
```

⟨*global* `quiet` 214f⟩≡ (214h)
```
int quiet;
```

⟨*global* `ptab` 214g⟩≡ (214h)
```
Ptab ptab[Maxproc];
```

⟨`acid/globals.c` 214h⟩≡
```
#include <u.h>
#include <libc.h>
#include <bio.h>
#include <mach.h>
#include "acid.h"

⟨global kernel 212⟩
⟨global remote 213a⟩
⟨global text 213b⟩
⟨global silent 213c⟩
⟨global fhdr (acid/globals.c) 213d⟩
⟨global line (acid/globals.c) 213e⟩
⟨global bout 213f⟩
⟨global io 213g⟩
⟨global iop 213h⟩
⟨global symbol 213i⟩
⟨global interactive (acid/globals.c) 213j⟩
⟨global na 213k⟩
⟨global wtflag (acid/globals.c) 213l⟩
⟨global cormap (acid/globals.c) 213m⟩
⟨global symmap (acid/globals.c) 213n⟩
⟨global hash 213o⟩
⟨global dogc 213p⟩
⟨global ret 213q⟩
⟨global aout 213r⟩
⟨global gotint 213s⟩
⟨global gcl 213t⟩
⟨global stacked 214a⟩
⟨global err 214b⟩
⟨global prnt 214c⟩
⟨global tracelist 214d⟩
⟨global initialising 214e⟩
⟨global quiet 214f⟩

⟨global ptab 214g⟩
```

D.5.3 acid/lex.c

⟨*global* keywds 215a⟩≡ (225b)
```
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
};
```

⟨*global* cmap 215b⟩≡ (225b)
```
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,
};
```

⟨*function* kinit 215c⟩≡ (225b)
```
void
kinit(void)
{
    int i;

    for(i = 0; keywds[i].name; i++)
        enter(keywds[i].name, keywds[i].terminal);
}
```

⟨*struct* IOstack 215d⟩≡ (225b)
```
struct IOstack
```

```
{
    char *name;
    int line;
    char *text;
    char *ip;
    Biobuf *fin;
    IOstack *prev;
};
```

⟨*global* lexio 216a⟩≡ (225b)
```
IOstack *lexio;
```

⟨*function* pushfile 216b⟩≡ (225b)
```
void
pushfile(char *file)
{
    Biobuf *b;
    IOstack *io;

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

    if(b == 0)
        error("pushfile: %s: %r", file);

    io = malloc(sizeof(IOstack));
    if(io == 0)
        fatal("no memory");
    io->name = strdup(file);
    if(io->name == 0)
        fatal("no memory");
    io->line = line;
    line = 1;
    io->text = 0;
    io->fin = b;
    io->prev = lexio;
    lexio = io;
}
```

⟨*function* pushstr 216c⟩≡ (225b)
```
void
pushstr(Node *s)
{
    IOstack *io;

    io = malloc(sizeof(IOstack));
    if(io == 0)
        fatal("no memory");
    io->line = line;
    line = 1;
    io->name = strdup("<string>");
    if(io->name == 0)
        fatal("no memory");
    io->line = line;
    line = 1;
    io->text = strdup(s->string->string);
```

```
    if(io->text == 0)
        fatal("no memory");
    io->ip = io->text;
    io->fin = 0;
    io->prev = lexio;
    lexio = io;
}
```

⟨*function* restartio 217a⟩≡ (225b)

```
void
restartio(void)
{
    Bflush(lexio->fin);
    Binit(lexio->fin, 0, OREAD);
}
```

⟨*function* popio 217b⟩≡ (225b)

```
int
popio(void)
{
    IOstack *s;

    if(lexio == 0)
        return 0;

    if(lexio->prev == 0){
        if(lexio->fin)
            restartio();
        return 0;
    }

    if(lexio->fin)
        Bterm(lexio->fin);
    else
        free(lexio->text);
    free(lexio->name);
    line = lexio->line;
    s = lexio;
    lexio = s->prev;
    free(s);
    return 1;
}
```

⟨*function* Lfmt 217c⟩≡ (225b)

```
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;
}
```

⟨*function* unlexc 218a⟩≡ (225b)

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

    if(lexio->fin)
        Bungetc(lexio->fin);
    else
        lexio->ip--;
}
```

⟨*function* lexc 218b⟩≡ (225b)

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

    if(lexio->fin) {
        c = Bgetc(lexio->fin);
        if(gotint)
            error("interrupt");
        return c;
    }

    c = *lexio->ip++;
    if(c == 0)
        return -1;
    return c;
}
```

⟨*function* escchar 218c⟩≡ (225b)

```
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;
}
```

⟨*function* `eatstring` 219a⟩≡ (225b)

```
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);
}
```

⟨*function* `eatnl` 219b⟩≡ (225b)

```
void
eatnl(void)
{
```

```
    int c;

    line++;
    for(;;) {
        c = lexc();
        if(c == Eof)
            error("eof in comment");
        if(c == '\n')
            return;
    }
}
```

⟨*function* yylex 220⟩≡ (225b)

```
int
yylex(void)
{
    int c;
    extern char vfmt[];

loop:
    Bflush(bout);
    c = lexc();
    switch(c) {
    case Eof:
        if(gotint) {
            gotint = 0;
            stacked = 0;
            Bprint(bout, "\nacid: ");
            goto loop;
        }
        return Eof;

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

    case ' ':
    case '\t':
        goto loop;

    case '\n':
        line++;
        if(interactive == 0)
            goto loop;
        if(stacked) {
            print("\t");
            goto loop;
        }
        return ';';

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

        return '.';

    case '(':
    case ')':
```

```
case '[':
case ']':
case ';':
case ':':
case ',':
case '~':
case '?':
case '*':
case '@':
case '^':
case '%':
    return c;
case '{':
    stacked++;
    return c;
case '}':
    stacked--;
    return c;

case '\\':
    c = lexc();
    if(strchr(vfmt, c) == 0) {
        unlexc(c);
        return '\\';
    }
    yylval.ival = c;
    return Tfmt;

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

case '+':
    c = lexc();
    if(c == '+')
        return Tinc;
    unlexc(c);
    return '+';

case '/':
    c = lexc();
    if(c == '/') {
        eatnl();
        goto loop;
    }
    unlexc(c);
    return '/';

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

```
            unlexc(c);
        }
        return Tconst;

    case '&':
        c = lexc();
        if(c == '&')
            return Tandand;
        unlexc(c);
        return '&';

    case '=':
        c = lexc();
        if(c == '=')
            return Teq;
        unlexc(c);
        return '=';

    case '|':
        c = lexc();
        if(c == '|')
            return Toror;
        unlexc(c);
        return '|';

    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 '>';

    case '-':
        c = lexc();

        if(c == '>')
            return Tindir;

        if(c == '-')
            return Tdec;
        unlexc(c);
        return '-';

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

⟨*function* numsym 222⟩≡ (225b)

```
int
numsym(char first)
{
    int c, isbin, isfloat, ishex;
    char *sel, *p;
    Lsym *s;

    symbol[0] = first;
    p = symbol;

    ishex = 0;
    isbin = 0;
    isfloat = 0;
    if(first == '.')
        isfloat = 1;

    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 = 1;
            if(!isbin && c == 'x')
                ishex = 1;
            if(!ishex && c == 'b')
                isbin = 1;
            *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;
    }

    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;
}
```

⟨*function* enter 224a⟩≡ (225b)

```
Lsym*
enter(char *name, int t)
{
    Lsym *s;
    uint h;
    char *p;
    Value *v;

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

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

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

    return s;
}
```

⟨*function* look 224b⟩≡ (225b)

```
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;
    return 0;
}
```

⟨*function* mkvar 225a⟩≡ (225b)

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

    l = look(s);
    if(l == 0)
        l = enter(s, Tid);
    return l;
}
```

⟨acid/lex.c 225b⟩≡

```
#include <u.h>
#include <libc.h>
#include <bio.h>
#include <ctype.h>
#include <mach.h>
#include "acid.h"
#include "y.tab.h"

⟨global keywds 215a⟩

⟨global cmap 215b⟩

⟨function kinit 215c⟩

typedef struct IOstack IOstack;
⟨struct IOstack 215d⟩
⟨global lexio 216a⟩

⟨function pushfile 216b⟩

⟨function pushstr 216c⟩

⟨function restartio 217a⟩

⟨function popio 217b⟩

⟨function Lfmt 217c⟩

⟨function unlexc 218a⟩

⟨function lexc 218b⟩

⟨function escchar 218c⟩

⟨function eatstring 219a⟩
```

⟨*function* `eatnl` 219b⟩

⟨*function* `yylex` 220⟩

⟨*function* `numsym` 222⟩

⟨*function* `enter` 224a⟩

⟨*function* `look` 224b⟩

⟨*function* `mkvar` 225a⟩

D.5.4 `acid/main.c`

⟨*global* `bioout` 226a⟩≡ (236c)
```
static Biobuf bioout;
```

⟨*global* `prog` 226b⟩≡ (236c)
```
static char prog[128];
```

⟨*global* `lm` 226c⟩≡ (236c)
```
static char* lm[16];
```

⟨*global* `nlm` 226d⟩≡ (236c)
```
static int nlm;
```

⟨*global* `mtype` 226e⟩≡ (236c)
```
static char* mtype;
```

⟨*function usage (acid/main.c)* 226f⟩≡ (236c)
```
void
usage(void)
{
    fprint(2, "usage: acid [-kqw] [-l library] [-m machine] [pid] [file]\n");
    exits("usage");
}
```

⟨*function main (acid/main.c)* 226g⟩≡ (236c)
```
void
main(int argc, char *argv[])
{
    Lsym *l;
    Node *n;
    char *s;
    int pid, i;

    argv0 = argv[0];
    pid = 0;
    aout = "8.out";
    quiet = 1;

    mtype = 0;
    ARGBEGIN{
    case 'm':
        mtype = ARGF();
        break;
    case 'w':
        wtflag = 1;
        break;
```

```
case 'l':
    s = ARGF();
    if(s == 0)
        usage();
    lm[nlm++] = s;
    break;
case 'k':
    kernel++;
    break;
case 'q':
    quiet = 0;
    break;
case 'r':
    pid = 1;
    remote++;
    kernel++;
    break;
default:
    usage();
}ARGEND

if(argc > 0) {
    if(remote)
        aout = argv[0];
    else
    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 if(kernel)
            aout = system();
    }
    else {
        if(kernel) {
            fprint(2, "acid: -k requires a pid\n");
            usage();
        }
        aout = argv[0];
    }
} else
if(remote)
    aout = "/mips/9ch";

fmtinstall('x', xfmt);
fmtinstall('L', Lfmt);
Binit(&bioout, 1, OWRITE);
bout = &bioout;

kinit();
initialising = 1;
pushfile(0);
loadvars();
installbuiltin();

if(mtype && machbyname(mtype) == 0)
    print("unknown machine %s", mtype);

if (attachfiles(aout, pid) < 0)
```

```
        varreg();  /* use default register set on error */

    loadmodule("/sys/lib/acid/port");
    loadmoduleobjtype();

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

    userinit();
    varsym();

    l = look("acidmap");
    if(l && l->proc) {
        n = an(ONAME, ZN, ZN);
        n->sym = l;
        n = an(OCALL, n, ZN);
        execute(n);
    }

    interactive = 1;
    initialising = 0;
    line = 1;

    notify(catcher);

    for(;;) {
        if(setjmp(err)) {
            Binit(&bioout, 1, OWRITE);
            unwind();
        }
        stacked = 0;

        Bprint(bout, "acid: ");

        if(yyparse() != 1)
            die();
        restartio();

        unwind();
    }
    /* not reached */
}
```

⟨*function* attachfiles 228⟩≡ (236c)

```
static int
attachfiles(char *aout, int pid)
{
    interactive = 0;
    if(setjmp(err))
        return -1;

    if(aout) {    /* executable given */
        if(wtflag)
```

```
            text = open(aout, ORDWR);
        else
            text = open(aout, OREAD);

        if(text < 0)
            error("%s: can't open %s: %r\n", argv0, aout);
        readtext(aout);
    }
    if(pid)     /* pid given */
        sproc(pid);
    return 0;
}
```

⟨*function die (acid/main.c)* 229a⟩≡ (236c)

```
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(0);
}
```

⟨*function* `loadmoduleobjtype` 229b⟩≡ (236c)

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

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

⟨*function* `userinit` 229c⟩≡ (236c)

```
void
userinit(void)
{
    Lsym *l;
    Node *n;
    char *buf, *p;

    p = getenv("home");
    if(p != 0) {
        buf = smprint("%s/lib/acid", p);
        silent = 1;
        loadmodule(buf);
        free(buf);
    }

    interactive = 0;
    if(setjmp(err)) {
        unwind();
```

```
        return;
    }
    l = look("acidinit");
    if(l && l->proc) {
        n = an(ONAME, ZN, ZN);
        n->sym = l;
        n = an(OCALL, n, ZN);
        execute(n);
    }
}
```

⟨*function* `loadmodule` 230a⟩≡ (236c)

```
void
loadmodule(char *s)
{
    interactive = 0;
    if(setjmp(err)) {
        unwind();
        return;
    }
    pushfile(s);
    silent = 0;
    yyparse();
    popio();
    return;
}
```

⟨*function* `readtext` 230b⟩≡ (236c)

```
void
readtext(char *s)
{
    Dir *d;
    Lsym *l;
    Value *v;
    uvlong length;
    Symbol sym;
    extern Machdata armmach;

    if(mtype != 0){
        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;
    }

    machdata = &armmach;

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

    symmap = loadmap(0, text, &fhdr);
```

```
    if(symmap == 0)
        print("%s: (error) loadmap: cannot make symbol map\n", argv0);

    if(syminit(text, &fhdr) < 0) {
        print("%s: (error) syminit: %r\n", argv0);
        return;
    }
    print("%s:%s\n", s, fhdr.name);

    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;
    }

    l = mkvar("objtype");
    v = l->v;
    v->fmt = 's';
    v->set = 1;
    v->string = strnode(mach->name);
    v->type = TSTRING;

    l = mkvar("textfile");
    v = l->v;
    v->fmt = 's';
    v->set = 1;
    v->string = strnode(s);
    v->type = TSTRING;

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

⟨*function* an 231a⟩≡ (236c)

```
Node*
an(int op, Node *l, Node *r)
{
    Node *n;

    n = gmalloc(sizeof(Node));
    memset(n, 0, sizeof(Node));
    n->gclink = gcl;
    gcl = n;
    n->op = op;
    n->left = l;
    n->right = r;
    return n;
}
```

⟨*function* al 231b⟩≡ (236c)

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

    l = gmalloc(sizeof(List));
    memset(l, 0, sizeof(List));
```

```
    l->type = t;
    l->gclink = gcl;
    gcl = l;
    return l;
}
```

⟨*function* con 232a⟩≡ (236c)

```
Node*
con(vlong v)
{
    Node *n;

    n = an(OCONST, ZN, ZN);
    n->ival = v;
    n->fmt = 'W';
    n->type = TINT;
    return n;
}
```

⟨*function fatal (acid/main.c)* 232b⟩≡ (236c)

```
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(2, "%s: %L (fatal problem) %s\n", argv0, buf);
    exits(buf);
}
```

⟨*function* yyerror 232c⟩≡ (236c)

```
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);
}
```

⟨*function* marktree 232d⟩≡ (236c)

```
void
marktree(Node *n)
{

    if(n == 0)
        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;
    }
}
```

⟨*function* marklist 233a⟩≡ (236c)

```
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;
    }
}
```

⟨*function* gc 233b⟩≡ (236c)

```
void
gc(void)
{
    int i;
    Lsym *f;
    Value *v;
    Gc *m, **p, *next;

    if(dogc < Mempergc)
        return;
    dogc = 0;

    /* 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;
    }
}
```

⟨*function* gmalloc 234a⟩≡ (236c)

```
void*
gmalloc(long l)
{
    void *p;

    dogc += l;
    p = malloc(l);
    if(p == 0)
        fatal("out of memory");
    return p;
}
```

⟨*function* checkqid 234b⟩≡ (236c)

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

    if(kernel)
        return;

    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);
}
```

⟨*function* catcher 235a⟩≡ (236c)

```
void
catcher(void *junk, char *s)
{
    USED(junk);

    if(strstr(s, "interrupt")) {
        gotint = 1;
        noted(NCONT);
    }
    noted(NDFLT);
}
```

⟨*function* system 235b⟩≡ (236c)

```
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;
}
```

⟨*function* isnumeric 236a⟩≡ (236c)
```
int
isnumeric(char *s)
{
    while(*s) {
        if(*s < '0' || *s > '9')
            return 0;
        s++;
    }
    return 1;
}
```

⟨*function* xfmt 236b⟩≡ (236c)
```
int
xfmt(Fmt *f)
{
    f->flags ^= FmtSharp;
    return _ifmt(f);
}
```

⟨acid/main.c 236c⟩≡
```
#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 226a⟩
⟨*global* prog 226b⟩
⟨*global* lm 226c⟩
⟨*global* nlm 226d⟩
⟨*global* mtype 226e⟩

```
static int attachfiles(char*, int);
int xfmt(Fmt*);
int isnumeric(char*);
void die(void);
void loadmoduleobjtype(void);
```

⟨*function usage (acid/main.c)* 226f⟩

⟨*function main (acid/main.c)* 226g⟩

⟨*function* attachfiles 228⟩

⟨*function die (acid/main.c)* 229a⟩

⟨*function* loadmoduleobjtype 229b⟩

⟨*function* userinit 229c⟩

⟨*function* `loadmodule` 230a⟩

⟨*function* `readtext` 230b⟩

⟨*function* `an` 231a⟩

⟨*function* `al` 231b⟩

⟨*function* `con` 232a⟩

⟨*function fatal (acid/main.c)* 232b⟩

⟨*function* `yyerror` 232c⟩

⟨*function* `marktree` 232d⟩

⟨*function* `marklist` 233a⟩

⟨*function* `gc` 233b⟩

⟨*function* `gmalloc` 234a⟩

⟨*function* `checkqid` 234b⟩

⟨*function* `catcher` 235a⟩

⟨*function* `system` 235b⟩

⟨*function* `isnumeric` 236a⟩

⟨*function* `xfmt` 236b⟩

D.5.5 `acid/util.c`

⟨*global* `syren` 237a⟩≡ (242c)
```
static int syren;
```

⟨*function* `unique` 237b⟩≡ (242c)
```
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;
}
```

⟨*function* varsym 238⟩≡ (242c)

```
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 == 0)
        print("no symbol information\n");
}
```

⟨*function* varreg 239⟩≡ (242c)

```
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 == 0)
        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);
}
```

⟨*function* `loadvars` 240a⟩≡ (242c)

```
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;
}
```

⟨*function rget (acid/util.c)* 240b⟩≡ (242c)

```
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;
}
```

⟨*function* strnodlen 241a⟩≡ (242c)

```
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';

    s->gclink = gcl;
    gcl = s;

    return s;
}
```

⟨*function* strnode 241b⟩≡ (242c)

```
String*
strnode(char *name)
{
    return strnodlen(name, strlen(name));
}
```

⟨*function* runenode 241c⟩≡ (242c)

```
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;
}
```

⟨*function* stradd 241d⟩≡ (242c)

```
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;
}
```

⟨*function* straddrune 242a⟩≡ (242c)

```
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;
}
```

⟨*function* scmp 242b⟩≡ (242c)

```
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 242c⟩≡

```
#include <u.h>
#include <libc.h>
#include <bio.h>
#include <ctype.h>
#include <mach.h>
#include "acid.h"
#include "y.tab.h"

⟨global syren 237a⟩

⟨function unique 237b⟩

⟨function varsym 238⟩

⟨function varreg 239⟩

⟨function loadvars 240a⟩
```

⟨*function rget (acid/util.c)* 240b⟩

⟨*function* `strnodlen` 241a⟩

⟨*function* `strnode` 241b⟩

⟨*function* `runenode` 241c⟩

⟨*function* `stradd` 241d⟩

⟨*function* `straddrune` 242a⟩

⟨*function* `scmp` 242b⟩

D.5.6 acid/exec.c

⟨*function error (acid/exec.c)* 243a⟩≡ (251)

```
void
error(char *fmt, ...)
{
    int i;
    char buf[2048];
    va_list arg;

    /* Unstack io channels */
    if(iop != 0) {
        for(i = 1; i < iop; i++)
            Bterm(io[i]);
        bout = io[0];
        iop = 0;
    }

    ret = 0;
    gotint = 0;
    Bflush(bout);
    if(silent)
        silent = 0;
    else {
        va_start(arg, fmt);
        vseprint(buf, buf+sizeof(buf), fmt, arg);
        va_end(arg);
        fprint(2, "%L: (error) %s\n", buf);
    }
    while(popio())
        ;
    interactive = 1;
    longjmp(err, 1);
}
```

⟨*function* `unwind` 243b⟩≡ (251)

```
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;
            }
        }
    }
}
```

⟨*function* execute 244⟩≡ (251)

```
void
execute(Node *n)
{
    Value *v;
    Lsym *sl;
    Node *l, *r;
    vlong i, s, e;
    Node res, xx;
    static int stmnt;

    gc();
    if(gotint)
        error("interrupted");

    if(n == 0)
        return;

    if(stmnt++ > 5000) {
        Bflush(bout);
        stmnt = 0;
    }

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

    switch(n->op) {
    default:
        expr(n, &res);
        if(ret || (res.type == TLIST && res.l == 0 && n->op != OADD))
            break;
        prnt->right = &res;
        expr(prnt, &xx);
        break;
    case OASGN:
    case OCALL:
        expr(n, &res);
        break;
    case OCOMPLEX:
        decl(n);
        break;
    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;
    case ORET:
        if(ret == 0)
            error("return not in function");
        expr(n->left, ret->val);
        longjmp(ret->rlab, 1);
    case OLIST:
        execute(n->left);
        execute(n->right);
        break;
    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;
    case OWHILE:
        for(;;) {
            expr(l, &res);
            if(!fbool(&res))
                break;
            execute(r);
        }
        break;
    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;
    }
}
```

⟨*function* fbool 245⟩≡ (251)

```
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;
}
```

⟨*function* convflt 246a⟩≡ (251)

```
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);
    }
}
```

⟨*function* indir 246b⟩≡ (251)

```
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;
    }
}
```

⟨*function* windir 249⟩≡ (251)

```
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");
}
```

⟨*function* call 250⟩≡ (251)

```
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);
    }
}
```

⟨acid/exec.c 251⟩≡

```
#include <u.h>
#include <libc.h>
#include <bio.h>
#include <ctype.h>
#include <mach.h>
#include "acid.h"

⟨function error (acid/exec.c) 243a⟩

⟨function unwind 243b⟩

⟨function execute 244⟩

⟨function fbool 245⟩

⟨function convflt 246a⟩

⟨function indir 246b⟩

⟨function windir 249⟩

⟨function call 250⟩
```

D.5.7 acid/proc.c

⟨*function* nocore 252a⟩≡ (256c)

```
void
nocore(void)
{
    int i;

    if(cormap == 0)
        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 = 0;
}
```

⟨*function* sproc 252b⟩≡ (256c)

```
void
sproc(int pid)
{
    Lsym *s;
    char buf[64];
    int i, fcor;

    if(symmap == 0)
        error("no map");

    snprint(buf, sizeof(buf), "/proc/%d/mem", pid);
    fcor = open(buf, ORDWR);
    if(fcor < 0)
        error("setproc: open %s: %r", buf);

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

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

    nocore();
    cormap = attachproc(pid, kernel, fcor, &fhdr);
    if (cormap == 0)
        error("setproc: can't make coremap: %r");
    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);
}
```

⟨*function* nproc 252c⟩≡ (256c)

```
int
nproc(char **argv)
{
    char buf[128];
    int pid, i, fd;

    pid = fork();
```

```
    switch(pid) {
    case -1:
        error("new: fork %r");
    case 0:
        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(0);
        close(1);
        close(2);
        for(i = 3; i < NFD; i++)
            close(i);

        open("/dev/cons", OREAD);
        open("/dev/cons", OWRITE);
        open("/dev/cons", OWRITE);
        exec(argv[0], argv);
        fatal("new: exec %s: %r");
    default:
        install(pid);
        msg(pid, "waitstop");
        notes(pid);
        sproc(pid);
        dostop(pid);
        break;
    }

    return pid;
}
```

⟨*function notes (acid/proc.c)* 253⟩≡ (256c)

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

    s = look("notes");
    if(s == 0)
        return;
    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);
}
```

⟨*function* dostop 254a⟩≡ (256c)

```
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);
    }
}
```

⟨*function* install 254b⟩≡ (256c)

```
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;
}
```

⟨*function* deinstall 255a⟩≡ (256c)

```
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;
        }
    }
}
```

⟨*function* msg 255b⟩≡ (256c)

```
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);
}
```

⟨*function* getstatus 256a⟩≡ (256c)

```
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];
}
```

⟨*function* waitfor 256b⟩≡ (256c)

```
Waitmsg*
waitfor(int pid)
{
    Waitmsg *w;

    for(;;) {
        if((w = wait()) == nil)
            error("wait %r");
        if(w->pid == pid)
            return w;
        free(w);
    }
}
```

⟨acid/proc.c 256c⟩≡

```
#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 252a⟩

⟨*function* sproc 252b⟩

⟨*function* nproc 252c⟩

⟨*function notes (acid/proc.c)* 253⟩

⟨*function* dostop 254a⟩

⟨*function* `install` 254b⟩

⟨*function* `deinstall` 255a⟩

⟨*function* `msg` 255b⟩

⟨*function* `getstatus` 256a⟩

⟨*function* `waitfor` 256b⟩

D.5.8 acid/list.c

⟨*global* `tail` 257a⟩≡ (261b)
```
static List **tail;
```

⟨*function* `construct` 257b⟩≡ (261b)
```
List*
construct(Node *l)
{
    List *lh, **save;

    save = tail;
    lh = 0;
    tail = &lh;
    build(l);
    tail = save;

    return lh;
}
```

⟨*function* `listlen` 257c⟩≡ (261b)
```
int
listlen(List *l)
{
    int len;

    len = 0;
    while(l) {
        len++;
        l = l->next;
    }
    return len;
}
```

⟨*function* `build` 257d⟩≡ (261b)
```
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;
    }
}
```

⟨*function* addlist 258a⟩≡ (261b)

```
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 258b⟩≡ (261b)

```
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;
}
```

⟨*function* listcmp 258c⟩≡ (261b)

```
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;
}
```

⟨*function* nthelem 259a⟩≡ (261b)

```
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;
}
```

⟨*function* delete 259b⟩≡ (261b)

```
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` 260a⟩≡ (261b)

```
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;
}
```

⟨*function* `listlocals` 260b⟩≡ (261b)

```
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;
}
```

⟨*function* `listparams` 260c⟩≡ (261b)

```
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;
}
```

⟨*function* `trlist` 261a⟩≡ (261b)

```
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);
}
```

⟨`acid/list.c` 261b⟩≡

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

```
#include <bio.h>
#include <ctype.h>
#include <mach.h>
#include "acid.h"
```

⟨*global* `tail` 257a⟩

⟨*function* `construct` 257b⟩

⟨*function* `listlen` 257c⟩

⟨*function* `build` 257d⟩

⟨*function* `addlist` 258a⟩

⟨*function* `append` 258b⟩

⟨*function* `listcmp` 258c⟩

⟨*function* `nthelem` 259a⟩

⟨*function* `delete` 259b⟩

⟨*function* `listvar` 260a⟩

⟨*function* `listlocals` 260b⟩

⟨*function* `listparams` 260c⟩

⟨*function* `trlist` 261a⟩

D.5.9 acid/dot.c

⟨*function* `srch` 262a⟩≡ (265a)

```
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` 262b⟩≡ (265a)

```
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);

}
```

⟨*global tail (acid/dot.c)* 263a⟩≡ (265a)

```
static Type **tail;
```

⟨*global* base 263b⟩≡ (265a)

```
static Lsym *base;
```

⟨*function* buildtype 263c⟩≡ (265a)

```
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;
    }
}
```

⟨*function* defcomplex 264a⟩≡ (265a)

```
void
defcomplex(Node *tn, Node *m)
{
    tail = &tn->sym->lt;
    base = tn->sym;
    buildtype(m, 0);
}
```

⟨*function* decl 264b⟩≡ (265a)

```
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;
    }
```

⟨acid/dot.c 265a⟩≡

```
    #include <u.h>
    #include <libc.h>
    #include <bio.h>
    #include <ctype.h>
    #include <mach.h>
    #include "acid.h"

    ⟨function srch 262a⟩

    ⟨function odot 262b⟩

    ⟨global tail (acid/dot.c) 263a⟩
    ⟨global base 263b⟩

    ⟨function buildtype 263c⟩

    ⟨function defcomplex 264a⟩

    ⟨function decl 264b⟩
```

D.5.10 acid/print.c

⟨global binop 265b⟩≡ (273a)

```
    static char *binop[] =
    {
        [OMUL] "*",
        [ODIV] "/",
        [OMOD] "%",
        [OADD] "+",
        [OSUB] "-",
        [ORSH] ">>",
        [OLSH] "<<",
        [OLT] "<",
        [OGT] ">",
        [OLEQ] "<=",
        [OGEQ] ">=",
        [OEQ] "==",
        [ONEQ] "!=",
        [OLAND] "&",
        [OXOR] "^",
        [OLOR] "|",
        [OCAND] "&&",
        [OCOR] "||",
        [OASGN] " = ",
    };
```

⟨global tabs 265c⟩≡ (273a)

```
    static char *tabs = "\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t";
```

⟨*global* typenames 266a⟩≡ (273a)
```
char *typenames[] =
{
    [TINT]  "integer",
    [TFLOAT] "float",
    [TSTRING] "string",
    [TLIST]  "list",
    [TCODE]  "code",
};
```

⟨*function* cmp 266b⟩≡ (273a)
```
int
cmp(void *va, void *vb)
{
    char **a = va;
    char **b = vb;

    return strcmp(*a, *b);
}
```

⟨*function* fundefs 266c⟩≡ (273a)
```
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");
```

```
    }
}
```

⟨*function* whatis 267⟩≡ (273a)

```
void
whatis(Lsym *l)
{
    int t;
    int def;
    Type *ti;

    if(l == 0) {
        fundefs();
        return;
    }

    def = 0;
    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 = 1;
    }
    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 = 1;
    }
    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 = 1;
    }
    if(l->builtin) {
        Bprint(bout, "builtin function\n");
        def = 1;
```

```
    }
    if(def == 0)
        Bprint(bout, "%s is undefined\n", l->name);
}
```

⟨*function* slist 268a⟩≡ (273a)

```
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);
}
```

⟨*function* pcode 268b⟩≡ (273a)

```
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);
    }
}
```

⟨*function* pexpr 269⟩≡ (273a)

```
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;
    }
}
```

⟨*function* pstr 272⟩≡ (273a)

```
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, '"');
```

```
}
```

⟨acid/print.c 273a⟩≡
```
#include <u.h>
#include <libc.h>
#include <bio.h>
#include <ctype.h>
#include <mach.h>
#include "acid.h"

⟨global binop 265b⟩

⟨global tabs 265c⟩
⟨global typenames 266a⟩

⟨function cmp 266b⟩

⟨function fundefs 266c⟩

⟨function whatis 267⟩

⟨function slist 268a⟩

⟨function pcode 268b⟩

⟨function pexpr 269⟩

⟨function pstr 272⟩
```

D.5.11 acid/expr.c

⟨global fsize 273b⟩≡ (291)
```
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* fmtsize 274a⟩≡ (291)

```
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;
    }
}
```

⟨*function* chklval 274b⟩≡ (291)

```
void
chklval(Node *lp)
{
    if(lp->op != ONAME)
        error("need l-value");
}
```

⟨*function* olist 274c⟩≡ (291)

```
void
olist(Node *n, Node *res)
{
    expr(n->left, res);
    expr(n->right, res);
}
```

⟨*function* oeval 274d⟩≡ (291)

```
void
oeval(Node *n, Node *res)
{
    expr(n->left, res);
    if(res->type != TCODE)
        error("bad type for eval");
    expr(res->cc, res);
}
```

⟨*function* ocast 274e⟩≡ (291)

```
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 275a⟩≡ (291)

```
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;
}
```

⟨*function* oindc 275b⟩≡ (291)

```
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;
}
```

⟨*function* oframe 275c⟩≡ (291)

```
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 276a⟩≡ (291)

```
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;
    }
}
```

⟨*function* oappend 276b⟩≡ (291)

```
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;
    }
}
```

⟨*function* odelete 277a⟩≡ (291)

```
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);
}
```

⟨*function* ohead 277b⟩≡ (291)

```
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;
    }
}
```

⟨*function* otail 277c⟩≡ (291)

```
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;
    }
```

⟨*function* oconst 278a⟩≡ (291)

```
    void
    oconst(Node *n, Node *res)
    {
        res->op = OCONST;
        res->type = n->type;
        res->Store = n->Store;
        res->comt = n->comt;
    }
```

⟨*function* oname 278b⟩≡ (291)

```
    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;
    }
```

⟨*function* octruct 278c⟩≡ (291)

```
    void
    octruct(Node *n, Node *res)
    {
        res->op = OCONST;
        res->type = TLIST;
        res->l = construct(n->left);
    }
```

⟨*function* oasgn 278d⟩≡ (291)

```
    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;
    }
}
```

⟨*function* oadd 279⟩≡ (291)

```
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;
        }
    }
}
```

⟨*function* osub 280⟩≡ (291)

```
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;
    }
}
```

⟨*function* omul 281a⟩≡ (291)

```
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;
    }
}
```

⟨*function* odiv 281b⟩≡ (291)

```
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;
    }
}
```

⟨*function* omod 282a⟩≡ (291)

```
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;
}
```

⟨*function* olsh 282b⟩≡ (291)

```
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;
}
```

⟨*function* orsh 283a⟩≡ (291)

```
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;
}
```

⟨*function* olt 283b⟩≡ (291)

```
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;
        }
    }
```

⟨*function* ogt 284a⟩≡ (291)

```
    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;
        }
    }
```

⟨*function* oleq 284b⟩≡ (291)

```
    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;
    }
}
```

⟨*function* ogeq 285⟩≡ (291)

```
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;
    }
}
```

⟨*function* oeq 286⟩≡ (291)

```
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;
}
```

⟨*function* oland 287a⟩≡ (291)

```
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;
}
```

⟨*function* oxor 287b⟩≡ (291)

```
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;
}
```

⟨*function* olor 287c⟩≡ (291)

```
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;
}
```

⟨*function* ocand 287d⟩≡ (291)

```
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;
}
```

⟨*function* onot 288a⟩≡ (291)

```
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;
}
```

⟨*function* ocor 288b⟩≡ (291)

```
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;
    }
}
```

⟨*function* oeinc 288c⟩≡ (291)

```
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;
}
```

⟨*function* opinc 289a⟩≡ (291)

```
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 --/++");
    }
}
```

⟨*function* ocall 289b⟩≡ (291)

```
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;
}
```

⟨*function* ofmt 290a⟩≡ (291)

```
void
ofmt(Node *n, Node *res)
{
    expr(n->left, res);
    res->fmt = n->right->ival;
}
```

⟨*function* owhat 290b⟩≡ (291)

```
void
owhat(Node *n, Node *res)
{
    res->op = OCONST;  /* Default return value */
    res->type = TLIST;
    res->l = 0;
    whatis(n->sym);
}
```

⟨*global* expop 290c⟩≡ (291)

```
void (*expop[])(Node*, Node*) =
{
    [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,
    [OWHAT]  owhat,
};
```

⟨acid/expr.c 291⟩≡

```
#include <u.h>
#include <libc.h>
#include <bio.h>
#include <ctype.h>
#include <mach.h>
#include "acid.h"

⟨global fsize 273b⟩

⟨function fmtsize 274a⟩

⟨function chklval 274b⟩

⟨function olist 274c⟩

⟨function oeval 274d⟩

⟨function ocast 274e⟩

⟨function oindm 275a⟩

⟨function oindc 275b⟩

⟨function oframe 275c⟩

⟨function oindex 276a⟩

⟨function oappend 276b⟩

⟨function odelete 277a⟩

⟨function ohead 277b⟩
```

⟨*function* `otail` 277c⟩

⟨*function* `oconst` 278a⟩

⟨*function* `oname` 278b⟩

⟨*function* `octruct` 278c⟩

⟨*function* `oasgn` 278d⟩

⟨*function* `oadd` 279⟩

⟨*function* `osub` 280⟩

⟨*function* `omul` 281a⟩

⟨*function* `odiv` 281b⟩

⟨*function* `omod` 282a⟩

⟨*function* `olsh` 282b⟩

⟨*function* `orsh` 283a⟩

⟨*function* `olt` 283b⟩

⟨*function* `ogt` 284a⟩

⟨*function* `oleq` 284b⟩

⟨*function* `ogeq` 285⟩

⟨*function* `oeq` 286⟩

⟨*function* `oland` 287a⟩

⟨*function* `oxor` 287b⟩

⟨*function* `olor` 287c⟩

⟨*function* `ocand` 287d⟩

⟨*function* `onot` 288a⟩

⟨*function* `ocor` 288b⟩

⟨*function* `oeinc` 288c⟩

⟨*function* `opinc` 289a⟩

⟨*function* `ocall` 289b⟩

⟨*function* `ofmt` 290a⟩

⟨*function* `owhat` 290b⟩

⟨*global* `expop` 290c⟩

D.5.12 acid/builtin.c

⟨*global* tab 293a⟩≡ (315)
```
struct Btab
{
    char *name;
    void (*fn)(Node*, Node*);
} tab[] =
{
    "atof",  cvtatof,
    "atoi",  cvtatoi,
    "error", doerror,
    "file",  getfile,
    "readfile", readfile,
    "access", doaccess,
    "filepc", filepc,
    "fnbound", funcbound,
    "fmt",  fmt,
    "follow", follow,
    "itoa",  cvtitoa,
    "kill",  kill,
    "match", match,
    "newproc", newproc,
    "pcfile", pcfile,
    "pcline", pcline,
    "print", bprint,
    "printto", printto,
    "rc",  rc,
    "reason", reason,
    "setproc", setproc,
    "start", start,
    "startstop", startstop,
    "status", status,
    "stop",  stop,
    "strace", strace,
    "sysr1", dosysr1,
    "waitstop", waitstop,
    "map",  map,
    "interpret", interpret,
    "include", include,
    "regexp", regexp,
    "fmtof", fmtof,
    "fmtsize", dofmtsize,
    0
};
```

⟨*global* vfmt 293b⟩≡ (315)
```
char vfmt[] = "aBbcCdDfFgGiIoOqQrRsSuUVWxXYZ38";
```

⟨*function* mkprint 293c⟩≡ (315)
```
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;
}
```

⟨*function* `installbuiltin` 294a⟩≡ (315)

```
void
installbuiltin(void)
{
    Btab *b;
    Lsym *s;

    b = tab;
    while(b->name) {
        s = look(b->name);
        if(s == 0)
            s = enter(b->name, Tid);

        s->builtin = b->fn;
        if(b->fn == bprint)
            mkprint(s);
        b++;
    }
}
```

⟨*function* `dosysr1` 294b⟩≡ (315)

```
void
dosysr1(Node *r, Node*)
{
    extern int sysnop(void);

    r->op = OCONST;
    r->type = TINT;
    r->fmt = 'D';
//    r->ival = sysr1();
//pad: now it's sysnop
    r->ival = nop();
}
```

⟨*function* `match` 294c⟩≡ (315)

```
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++;
    }
}
```

⟨*function* newproc 295⟩≡ (315)

```
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] = 0;
    r->op = OCONST;
    r->type = TINT;
    r->fmt = 'D';
    r->ival = nproc(argv);
}
```

⟨*function* `startstop` 296a⟩≡ (315)

```
void
startstop(Node *r, Node *args)
{
    Node res;

    USED(r);
    if(args == 0)
        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);
}
```

⟨*function* `waitstop` 296b⟩≡ (315)

```
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);
}
```

⟨*function* `start` 296c⟩≡ (315)

```
void
start(Node *r, Node *args)
{
    Node res;

    USED(r);
    if(args == 0)
        error("start(pid): no pid");
    expr(args, &res);
    if(res.type != TINT)
        error("start(pid): arg type");

    msg(res.ival, "start");
}
```

⟨*function* stop 297a⟩≡ (315)

```
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);
}
```

⟨*function* kill 297b⟩≡ (315)

```
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);
}
```

⟨*function* status 297c⟩≡ (315)

```
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;
}
```

⟨*function* reason 297d⟩≡ (315)

```
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));
}
```

⟨*function* follow 298a⟩≡ (315)

```
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;
    }
}
```

⟨*function* funcbound 298b⟩≡ (315)

```
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';
    }
}
```

⟨*function* setproc 299a⟩≡ (315)

```
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);
}
```

⟨*function* filepc 299b⟩≡ (315)

```
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';
}
```

⟨*function* interpret 299c⟩≡ (315)

```
void
interpret(Node *r, Node *args)
{
```

```
    Node res;
    int isave;

    if(args == 0)
        error("interpret(string): arg count");
    expr(args, &res);
    if(res.type != TSTRING)
        error("interpret(string): arg type");

    pushstr(&res);

    isave = interactive;
    interactive = 0;
    r->ival = yyparse();
    interactive = isave;
    popio();
    r->op = OCONST;
    r->type = TINT;
    r->fmt = 'D';
}
```

⟨*function* include 300a⟩≡ (315)

```
void
include(Node *r, Node *args)
{
    Node res;
    int 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 = 0;
    r->ival = yyparse();
    interactive = isave;
    popio();
    r->op = OCONST;
    r->type = TINT;
    r->fmt = 'D';
}
```

⟨*function* rc 300b⟩≡ (315)

```
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';
}
```

⟨*function* doerror 301a⟩≡ (315)

```
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);
}
```

⟨*function* doaccess 301b⟩≡ (315)

```
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;
}
```

⟨*function* `readfile` 302a⟩≡ (315)

```
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);
}
```

⟨*function getfile (acid/builtin.c)* 302b⟩≡ (315)

```
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);
}
```

⟨*function* cvtatof 303a⟩≡ (315)

```
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';
}
```

⟨*function* cvtatoi 303b⟩≡ (315)

```
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';
}
```

⟨*global* fmtflags 304a⟩≡ (315)
```
static char *fmtflags = "-0123456789. #,u";
```

⟨*global* fmtverbs 304b⟩≡ (315)
```
static char *fmtverbs = "bdox";
```

⟨*function* acidfmt 304c⟩≡ (315)
```
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;
}
```

⟨*function* cvtitoa 305a⟩≡ (315)

```
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';
}
```

⟨*function* mapent 305b⟩≡ (315)

```
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;
}
```

⟨*function* map 306⟩≡ (315)

```
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);
        }
    }
}
```

⟨*function* flatten 307a⟩≡ (315)

```
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;
    }
}
```

⟨*function* strace 307b⟩≡ (315)

```
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;
}
```

⟨*function* regerror 308a⟩≡ (315)

```
void
regerror(char *msg)
{
    error(msg);
}
```

⟨*function* regexp 308b⟩≡ (315)

```
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);
}
```

⟨*function* fmt 308c⟩≡ (315)

```
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;
}
```

⟨*function* patom 309⟩≡ (315)

```
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;
}
```

```
}
```

⟨*function* blprint 311a⟩≡ (315)

```
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, "}");
}
```

⟨*function* comx 311b⟩≡ (315)

```
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;
}
```

⟨*function* bprint 312a⟩≡ (315)

```
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');
}
```

⟨*function* printto 312b⟩≡ (315)

```
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];
}
```

⟨*function* pcfile 313a⟩≡ (315)

```
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);
}
```

⟨*function* pcline 313b⟩≡ (315)

```
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);
}
```

⟨*function* fmtof 314a⟩≡ (315)

```
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';
}
```

⟨*function* dofmtsize 314b⟩≡ (315)

```
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';
}
```

⟨acid/builtin.c 315⟩≡

```
#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 293a⟩

⟨global vfmt 293b⟩

⟨function mkprint 293c⟩

⟨function installbuiltin 294a⟩

⟨function dosysr1 294b⟩

⟨function match 294c⟩

⟨function newproc 295⟩
```

⟨*function* `startstop` 296a⟩

⟨*function* `waitstop` 296b⟩

⟨*function* `start` 296c⟩

⟨*function* `stop` 297a⟩

⟨*function* `kill` 297b⟩

⟨*function* `status` 297c⟩

⟨*function* `reason` 297d⟩

⟨*function* `follow` 298a⟩

⟨*function* `funcbound` 298b⟩

⟨*function* `setproc` 299a⟩

⟨*function* `filepc` 299b⟩

⟨*function* `interpret` 299c⟩

⟨*function* `include` 300a⟩

⟨*function* `rc` 300b⟩

⟨*function* `doerror` 301a⟩

⟨*function* `doaccess` 301b⟩

⟨*function* `readfile` 302a⟩

⟨*function getfile (acid/builtin.c)* 302b⟩

⟨*function* `cvtatof` 303a⟩

⟨*function* `cvtatoi` 303b⟩

⟨*global* `fmtflags` 304a⟩
⟨*global* `fmtverbs` 304b⟩

⟨*function* `acidfmt` 304c⟩

⟨*function* `cvtitoa` 305a⟩

⟨*function* `mapent` 305b⟩

⟨*function* `map` 306⟩

⟨*function* `flatten` 307a⟩

⟨*function* `strace` 307b⟩

⟨*function* `regerror` 308a⟩

⟨*function* `regexp` 308b⟩

⟨*function* `fmt` 308c⟩

⟨*function* `patom` 309⟩

⟨*function* `blprint` 311a⟩

⟨*function* `comx` 311b⟩

⟨*function* `bprint` 312a⟩

⟨*function* `printto` 312b⟩

⟨*function* `pcfile` 313a⟩

⟨*function* `pcline` 313b⟩

⟨*function* `fmtof` 314a⟩

⟨*function* `dofmtsize` 314b⟩

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. For an introduction see http://en.wikipedia.org/wiki/Literate_Program. cited page(s) 7

[Pad09] Yoann Padioleau. Syncweb, literate programming meets unison, 2009. https://github.com/aryx/syncweb. cited page(s) 7