# How to Use the Plan 9 C Compiler

*Rob Pike*
*rob@plan9.bell-labs.com*

## Introduction

The C compiler on Plan 9 is a wholly new program; in fact it was the first piece of software written for what would eventually become Plan 9 from Bell Labs. Programmers familiar with existing C compilers will find a number of differences in both the language the Plan 9 compiler accepts and in how the compiler is used.

The compiler is really a set of compilers, one for each architecture — MIPS, SPARC, Intel 386, Power PC, ARM, etc. — that accept a dialect of ANSI C and efficiently produce fairly good code for the target machine. There is a packaging of the compiler that accepts strict ANSI C for a POSIX environment, but this document focuses on the native Plan 9 environment, that in which all the system source and almost all the utilities are written.

## Source

The language accepted by the compilers is the core 1989 ANSI C language with some modest extensions, a greatly simplified preprocessor, a smaller library that includes system calls and related facilities, and a completely different structure for include files.

Official ANSI C accepts the old (K&R) style of declarations for functions; the Plan 9 compilers are more demanding. Without an explicit run-time flag (`-B`) whose use is discouraged, the compilers insist on new-style function declarations, that is, prototypes for function arguments. The function declarations in the libraries' include files are all in the new style so the interfaces are checked at compile time. For C programmers who have not yet switched to function prototypes the clumsy syntax may seem repellent but the payoff in stronger typing is substantial. Those who wish to import existing software to Plan 9 are urged to use the opportunity to update their code.

The compilers include an integrated preprocessor that accepts the familiar `#include`, `#define` for macros both with and without arguments, `#undef`, `#line`, `#ifdef`, `#ifndef`, and `#endif`. It supports neither `#if` nor `##`, although it does honor a few `#pragmas`. The `#if` directive was omitted because it greatly complicates the preprocessor, is never necessary, and is usually abused. Conditional compilation in general makes code hard to understand; the Plan 9 source uses it sparingly. Also, because the compilers remove dead code, regular `if` statements with constant conditions are more readable equivalents to many `#ifs`. To compile imported code ineluctably fouled by `#if` there is a separate command, `/bin/cpp`, that implements the complete ANSI C preprocessor specification.

Include files fall into two groups: machine-dependent and machine-independent. The machine-independent files occupy the directory `/sys/include`; the others are placed in a directory appropriate to the machine, such as `/mips/include`. The compiler searches for include files first in the machine-dependent directory and then in the machine-independent directory. At the time of writing there are thirty-one machine-independent include files and two (per machine) machine-dependent ones: `<ureg.h>`

and <u.h>. The first describes the layout of registers on the system stack, for use by the debugger. The second defines some architecture-dependent types such as jmp_buf for setjmp and the va_arg and va_list macros for handling arguments to variadic functions, as well as a set of typedef abbreviations for unsigned short and so on.

Here is an excerpt from /386/include/u.h:

```
#define nil             ((void*)0)
typedef unsigned short  ushort;
typedef unsigned char   uchar;
typedef unsigned long   ulong;
typedef unsigned int    uint;
typedef   signed char   schar;
typedef long long       vlong;

typedef long    jmp_buf[2];
#define JMPBUFSP        0
#define JMPBUFPC        1
#define JMPBUFDPC       0
```

Plan 9 programs use nil for the name of the zero-valued pointer. The type vlong is the largest integer type available; on most architectures it is a 64-bit value. A couple of other types in <u.h> are u32int, which is guaranteed to have exactly 32 bits (a possibility on all the supported architectures) and mpdigit, which is used by the multiprecision math package <mp.h>. The #define constants permit an architecture-independent (but compiler-dependent) implementation of stack-switching using setjmp and longjmp.

Every Plan 9 C program begins

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

because all the other installed header files use the typedefs declared in <u.h>.

In strict ANSI C, include files are grouped to collect related functions in a single file: one for string functions, one for memory functions, one for I/O, and none for system calls. Each include file is protected by an #ifdef to guarantee its contents are seen by the compiler only once. Plan 9 takes a different approach. Other than a few include files that define external formats such as archives, the files in /sys/include correspond to *libraries.* If a program is using a library, it includes the corresponding header. The default C library comprises string functions, memory functions, and so on, largely as in ANSI C, some formatted I/O routines, plus all the system calls and related functions. To use these functions, one must #include the file <libc.h>, which in turn must follow <u.h>, to define their prototypes for the compiler. Here is the complete source to the traditional first C program:

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

void
main(void)
{
        print("hello world\n");
        exits(0);
}
```

The print routine and its relatives fprint and sprint resemble the similarly-named functions in Standard I/O but are not attached to a specific I/O library. In Plan 9 main is not integer-valued; it should call exits, which takes a string argument (or null; here ANSI C promotes the 0 to a char*). All these functions are, of course,

documented in the Programmer's Manual.

To use `printf`, `<stdio.h>` must be included to define the function prototype for `printf`:

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

void
main(int argc, char *argv[])
{
        printf("%s: hello world; argc = %d\n", argv[0], argc);
        exits(0);
}
```

In practice, Standard I/O is not used much in Plan 9. I/O libraries are discussed in a later section of this document.

There are libraries for handling regular expressions, raster graphics, windows, and so on, and each has an associated include file. The manual for each library states which include files are needed. The files are not protected against multiple inclusion and themselves contain no nested `#includes`. Instead the programmer is expected to sort out the requirements and to `#include` the necessary files once at the top of each source file. In practice this is trivial: this way of handling include files is so straightforward that it is rare for a source file to contain more than half a dozen `#includes`.

The compilers do their own register allocation so the `register` keyword is ignored. For different reasons, `volatile` and `const` are also ignored.

To make it easier to share code with other systems, Plan 9 has a version of the compiler, `pcc`, that provides the standard ANSI C preprocessor, headers, and libraries with POSIX extensions. `Pcc` is recommended only when broad external portability is mandated. It compiles slower, produces slower code (it takes extra work to simulate POSIX on Plan 9), eliminates those parts of the Plan 9 interface not related to POSIX, and illustrates the clumsiness of an environment designed by committee. `Pcc` is described in more detail in *APE—The ANSI/POSIX Environment,* by Howard Trickey.

**Process**

Each CPU architecture supported by Plan 9 is identified by a single, arbitrary, alphanumeric character: `k` for SPARC, `q` for 32-bit Power PC, `v` for MIPS, `0` for little-endian MIPS, `5` for ARM v5 and later 32-bit architectures, `6` for AMD64, `8` for Intel 386, and `9` for 64-bit Power PC. The character labels the support tools and files for that architecture. For instance, for the 386 the compiler is `8c`, the assembler is `8a`, the link editor/loader is `8l`, the object files are suffixed `.8`, and the default name for an executable file is `8.out`. Before we can use the compiler we therefore need to know which machine we are compiling for. The next section explains how this decision is made; for the moment assume we are building 386 binaries and make the mental substitution for `8` appropriate to the machine you are actually using.

To convert source to an executable binary is a two-step process. First run the compiler, `8c`, on the source, say `file.c`, to generate an object file `file.8`. Then run the loader, `8l`, to generate an executable `8.out` that may be run (on a 386 machine):

```
8c file.c
8l file.8
8.out
```

The loader automatically links with whatever libraries the program needs, usually including the standard C library as defined by `<libc.h>`. Of course the compiler and loader

have lots of options, both familiar and new; see the manual for details. The compiler does not generate an executable automatically; the output of the compiler must be given to the loader. Since most compilation is done under the control of `mk` (see below), this is rarely an inconvenience.

The distribution of work between the compiler and loader is unusual. The compiler integrates preprocessing, parsing, register allocation, code generation and some assembly. Combining these tasks in a single program is part of the reason for the compiler's efficiency. The loader does instruction selection, branch folding, instruction scheduling, and writes the final executable. There is no separate C preprocessor and no assembler in the usual pipeline. Instead the intermediate object file (here a `.8` file) is a type of binary assembly language. The instructions in the intermediate format are not exactly those in the machine. For example, on the 68020 the object file may specify a MOVE instruction but the loader will decide just which variant of the MOVE instruction — MOVE immediate, MOVE quick, MOVE address, etc. — is most efficient.

The assembler, `8a`, is just a translator between the textual and binary representations of the object file format. It is not an assembler in the traditional sense. It has limited macro capabilities (the same as the integral C preprocessor in the compiler), clumsy syntax, and minimal error checking. For instance, the assembler will accept an instruction (such as memory-to-memory MOVE on the MIPS) that the machine does not actually support; only when the output of the assembler is passed to the loader will the error be discovered. The assembler is intended only for writing things that need access to instructions invisible from C, such as the machine-dependent part of an operating system; very little code in Plan 9 is in assembly language.

The compilers take an option `-S` that causes them to print on their standard output the generated code in a format acceptable as input to the assemblers. This is of course merely a formatting of the data in the object file; therefore the assembler is just an ASCII-to-binary converter for this format. Other than the specific instructions, the input to the assemblers is largely architecture-independent; see ''A Manual for the Plan 9 Assembler'', by Rob Pike, for more information.

The loader is an integral part of the compilation process. Each library header file contains a `#pragma` that tells the loader the name of the associated archive; it is not necessary to tell the loader which libraries a program uses. The C run-time startup is found, by default, in the C library. The loader starts with an undefined symbol, `_main`, that is resolved by pulling in the run-time startup code from the library. (The loader undefines `_mainp` when profiling is enabled, to force loading of the profiling start-up instead.)

Unlike its counterpart on other systems, the Plan 9 loader rearranges data to optimize access. This means the order of variables in the loaded program is unrelated to its order in the source. Most programs don't care, but some assume that, for example, the variables declared by

```
int a;
int b;
```

will appear at adjacent addresses in memory. On Plan 9, they won't.

**Heterogeneity**

When the system starts or a user logs in the environment is configured so the appropriate binaries are available in `/bin`. The configuration process is controlled by an environment variable, `$cputype`, with value such as `mips`, `386`, `arm`, or `sparc`. For each architecture there is a directory in the root, with the appropriate name, that holds the binary and library files for that architecture. Thus `/mips/lib` contains the object code libraries for MIPS programs, `/mips/include` holds MIPS-specific include files, and `/mips/bin` has the MIPS binaries. These binaries are attached to `/bin` at

boot time by binding /$cputype/bin to /bin, so /bin always contains the correct files.

The MIPS compiler, vc, by definition produces object files for the MIPS architecture, regardless of the architecture of the machine on which the compiler is running. There is a version of vc compiled for each architecture: /mips/bin/vc, /arm/bin/vc, /sparc/bin/vc, and so on, each capable of producing MIPS object files regardless of the native instruction set. If one is running on a SPARC, /sparc/bin/vc will compile programs for the MIPS; if one is running on machine $cputype, /$cputype/bin/vc will compile programs for the MIPS.

Because of the bindings that assemble /bin, the shell always looks for a command, say date, in /bin and automatically finds the file /$cputype/bin/date. Therefore the MIPS compiler is known as just vc; the shell will invoke /bin/vc and that is guaranteed to be the version of the MIPS compiler appropriate for the machine running the command. Regardless of the architecture of the compiling machine, /bin/vc is *always* the MIPS compiler.

Also, the output of vc and vl is completely independent of the machine type on which they are executed: .v files compiled (with vc) on a SPARC may be linked (with vl) on a 386. (The resulting v.out will run, of course, only on a MIPS.) Similarly, the MIPS libraries in /mips/lib are suitable for loading with vl on any machine; there is only one set of MIPS libraries, not one set for each architecture that supports the MIPS compiler.

**Heterogeneity and** mk

Most software on Plan 9 is compiled under the control of mk, a descendant of make that is documented in the Programmer's Manual. A convention used throughout the mkfiles makes it easy to compile the source into binary suitable for any architecture.

The variable $cputype is advisory: it reports the architecture of the current environment, and should not be modified. A second variable, $objtype, is used to set which architecture is being *compiled* for. The value of $objtype can be used by a mkfile to configure the compilation environment.

In each machine's root directory there is a short mkfile that defines a set of macros for the compiler, loader, etc. Here is /mips/mkfile:

```
</sys/src/mkfile.proto

CC=vc
LD=vl
O=v
AS=va
```

The line

```
</sys/src/mkfile.proto
```

causes mk to include the file /sys/src/mkfile.proto, which contains general definitions:

```
#
# common mkfile parameters shared by all architectures
#

OS=5689qv
CPUS=arm amd64 386 power mips
CFLAGS=-FTVw
LEX=lex
YACC=yacc
MK=/bin/mk
```

CC is obviously the compiler, AS the assembler, and LD the loader. O is the suffix for the object files and CPUS and OS are used in special rules described below.

Here is a `mkfile` to build the installed source for `sam`:

```
</$objtype/mkfile
OBJ=sam.$O address.$O buffer.$O cmd.$O disc.$O error.$O \
        file.$O io.$O list.$O mesg.$O moveto.$O multi.$O \
        plan9.$O rasp.$O regexp.$O string.$O sys.$O xec.$O

$O.out: $OBJ
        $LD $OBJ

install:        $O.out
        cp $O.out /$objtype/bin/sam

installall:
        for(objtype in $CPUS) mk install

%.$O:   %.c
        $CC $CFLAGS $stem.c

$OBJ:   sam.h errors.h mesg.h
address.$O cmd.$O parse.$O xec.$O unix.$O:      parse.h

clean:V:
        rm -f [$OS].out *.[$OS] y.tab.?
```

(The actual `mkfile` imports most of its rules from other secondary files, but this example works and is not misleading.) The first line causes `mk` to include the contents of `/$objtype/mkfile` in the current `mkfile`. If `$objtype` is `mips`, this inserts the MIPS macro definitions into the `mkfile`. In this case the rule for `$O.out` uses the MIPS tools to build `v.out`. The `%.$O` rule in the file uses `mk`'s pattern matching facilities to convert the source files to the object files through the compiler. (The text of the rules is passed directly to the shell, `rc`, without further translation. See the `mk` manual if any of this is unfamiliar.) Because the default rule builds `$O.out` rather than `sam`, it is possible to maintain binaries for multiple machines in the same source directory without conflict. This is also, of course, why the output files from the various compilers and loaders have distinct names.

The rest of the `mkfile` should be easy to follow; notice how the rules for `clean` and `installall` (that is, install versions for all architectures) use other macros defined in `/$objtype/mkfile`. In Plan 9, `mkfiles` for commands conventionally contain rules to `install` (compile and install the version for `$objtype`), `installall` (compile and install for all `$objtypes`), and `clean` (remove all object files, binaries, etc.).

The `mkfile` is easy to use. To build a MIPS binary, `v.out`:

```
% objtype=mips
% mk
```

To build and install a MIPS binary:

```
% objtype=mips
% mk install
```

To build and install all versions:

```
% mk installall
```

These conventions make cross-compilation as easy to manage as traditional native compilation. Plan 9 programs compile and run without change on machines from large multiprocessors to laptops. For more information about this process, see ''Plan 9 Mkfiles'', by Bob Flandrena.

**Portability**

Within Plan 9, it is painless to write portable programs, programs whose source is independent of the machine on which they execute. The operating system is fixed and the compiler, headers and libraries are constant so most of the stumbling blocks to portability are removed. Attention to a few details can avoid those that remain.

Plan 9 is a heterogeneous environment, so programs must *expect* that external files will be written by programs on machines of different architectures. The compilers, for instance, must handle without confusion object files written by other machines. The traditional approach to this problem is to pepper the source with `#ifdefs` to turn byte-swapping on and off. Plan 9 takes a different approach: of the handful of machine-dependent `#ifdefs` in all the source, almost all are deep in the libraries. Instead programs read and write files in a defined format, either (for low volume applications) as formatted text, or (for high volume applications) as binary in a known byte order. If the external data were written with the most significant byte first, the following code reads a 4-byte integer correctly regardless of the architecture of the executing machine (assuming an unsigned long holds 4 bytes):

```
ulong
getlong(void)
{
        ulong l;

        l = (getchar()&0xFF)<<24;
        l |= (getchar()&0xFF)<<16;
        l |= (getchar()&0xFF)<<8;
        l |= (getchar()&0xFF)<<0;
        return l;
}
```

Note that this code does not 'swap' the bytes; instead it just reads them in the correct order. Variations of this code will handle any binary format and also avoid problems involving how structures are padded, how words are aligned, and other impediments to portability. Be aware, though, that extra care is needed to handle floating point data.

Efficiency hounds will argue that this method is unnecessarily slow and clumsy when the executing machine has the same byte order (and padding and alignment) as the data. The CPU cost of I/O processing is rarely the bottleneck for an application, however, and the gain in simplicity of porting and maintaining the code greatly outweighs the minor speed loss from handling data in this general way. This method is how the Plan 9 compilers, the window system, and even the file servers transmit data between programs.

To port programs beyond Plan 9, where the system interface is more variable, it is probably necessary to use `pcc` and hope that the target machine supports ANSI C and POSIX.

**I/O**

The default C library, defined by the include file <libc.h>, contains no buffered I/O package. It does have several entry points for printing formatted text: `print` outputs text to the standard output, `fprint` outputs text to a specified integer file descriptor, and `sprint` places text in a character array. To access library routines for buffered I/O, a program must explicitly include the header file associated with an appropriate library.

The recommended I/O library, used by most Plan 9 utilities, is `bio` (buffered I/O), defined by <bio.h>. There also exists an implementation of ANSI Standard I/O, `stdio`.

`Bio` is small and efficient, particularly for buffer-at-a-time or line-at-a-time I/O. Even for character-at-a-time I/O, however, it is significantly faster than the Standard I/O library, `stdio`. Its interface is compact and regular, although it lacks a few conveniences. The most noticeable is that one must explicitly define buffers for standard input and output; `bio` does not predefine them. Here is a program to copy input to output a byte at a time using `bio`:

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

Biobuf  bin;
Biobuf  bout;

main(void)
{
        int c;

        Binit(&bin, 0, OREAD);
        Binit(&bout, 1, OWRITE);

        while((c=Bgetc(&bin)) != Beof)
                Bputc(&bout, c);
        exits(0);
}
```

For peak performance, we could replace `Bgetc` and `Bputc` by their equivalent in-line macros `BGETC` and `BPUTC` but the performance gain would be modest. For more information on `bio`, see the Programmer's Manual.

Perhaps the most dramatic difference in the I/O interface of Plan 9 from other systems' is that text is not ASCII. The format for text in Plan 9 is a byte-stream encoding of 16-bit characters. The character set is based on the Unicode Standard and is backward compatible with ASCII: characters with value 0 through 127 are the same in both sets. The 16-bit characters, called *runes* in Plan 9, are encoded using a representation called UTF, an encoding that is becoming accepted as a standard. (ISO calls it UTF-8; throughout Plan 9 it's just called UTF.) UTF defines multibyte sequences to represent character values from 0 to 65535. In UTF, character values up to 127 decimal, 7F hexadecimal, represent themselves, so straight ASCII files are also valid UTF. Also, UTF guarantees that bytes with values 0 to 127 (NUL to DEL, inclusive) will appear only when they represent themselves, so programs that read bytes looking for plain ASCII characters will continue to work. Any program that expects a one-to-one correspondence between bytes and characters will, however, need to be modified. An example is

parsing file names. File names, like all text, are in UTF, so it is incorrect to search for a character in a string by `strchr(filename, c)` because the character might have a multi-byte encoding. The correct method is to call `utfrune(filename, c)`, defined in *rune*(2), which interprets the file name as a sequence of encoded characters rather than bytes. In fact, even when you know the character is a single byte that can represent only itself, it is safer to use `utfrune` because that assumes nothing about the character set and its representation.

The library defines several symbols relevant to the representation of characters. Any byte with unsigned value less than `Runesync` will not appear in any multi-byte encoding of a character. `Utfrune` compares the character being searched against `Runesync` to see if it is sufficient to call `strchr` or if the byte stream must be interpreted. Any byte with unsigned value less than `Runeself` is represented by a single byte with the same value. Finally, when errors are encountered converting to runes from a byte stream, the library returns the rune value `Runeerror` and advances a single byte. This permits programs to find runes embedded in binary data.

`Bio` includes routines `Bgetrune` and `Bputrune` to transform the external byte stream UTF format to and from internal 16-bit runes. Also, the `%s` format to `print` accepts UTF; `%c` prints a character after narrowing it to 8 bits. The `%S` format prints a null-terminated sequence of runes; `%C` prints a character after narrowing it to 16 bits. For more information, see the Programmer's Manual, in particular *utf*(6) and *rune*(2), and the paper, ''Hello world, or Καλημέρα κόσμε, or こんにちは世界'', by Rob Pike and Ken Thompson; there is not room for the full story here.

These issues affect the compiler in several ways. First, the C source is in UTF. ANSI says C variables are formed from ASCII alphanumerics, but comments and literal strings may contain any characters encoded in the native encoding, here UTF. The declaration

```
char *cp = "abcÿ";
```

initializes the variable `cp` to point to an array of bytes holding the UTF representation of the characters `abcÿ`. The type `Rune` is defined in `<u.h>` to be `ushort`, which is also the 'wide character' type in the compiler. Therefore the declaration

```
Rune *rp = L"abcÿ";
```

initializes the variable `rp` to point to an array of unsigned short integers holding the 16-bit values of the characters `abcÿ`. Note that in both these declarations the characters in the source that represent `abcÿ` are the same; what changes is how those characters are represented in memory in the program. The following two lines:

```
print("%s\n", "abcÿ");
print("%S\n", L"abcÿ");
```

produce the same UTF string on their output, the first by copying the bytes, the second by converting from runes to bytes.

In C, character constants are integers but narrowed through the `char` type. The Unicode character ÿ has value 255, so if the `char` type is signed, the constant `'ÿ'` has value -1 (which is equal to EOF). On the other hand, `L'ÿ'` narrows through the wide character type, `ushort`, and therefore has value 255.

Finally, although it's not ANSI C, the Plan 9 C compilers assume any character with value above `Runeself` is an alphanumeric, so α is a legal, if non-portable, variable name.

### Arguments

Some macros are defined in <libc.h> for parsing the arguments to main(). They are described in *ARG*(2) but are fairly self-explanatory. There are four macros: ARGBEGIN and ARGEND are used to bracket a hidden switch statement within which ARGC returns the current option character (rune) being processed and ARGF returns the argument to the option, as in the loader option -o file. Here, for example, is the code at the beginning of main() in ramfs.c (see *ramfs*(1)) that cracks its arguments:

```
void
main(int argc, char *argv[])
{
        char *defmnt;
        int p[2];
        int mfd[2];
        int stdio = 0;

        defmnt = "/tmp";
        ARGBEGIN{
        case 'i':
                defmnt = 0;
                stdio = 1;
                mfd[0] = 0;
                mfd[1] = 1;
                break;
        case 's':
                defmnt = 0;
                break;
        case 'm':
                defmnt = ARGF();
                break;
        default:
                usage();
        }ARGEND
```

### Extensions

The compiler has several extensions to 1989 ANSI C, all of which are used extensively in the system source. Some of these have been adopted in later ANSI C standards. First, *structure displays* permit struct expressions to be formed dynamically. Given these declarations:

```
typedef struct Point Point;
typedef struct Rectangle Rectangle;

struct Point
{
        int x, y;
};

struct Rectangle
{
        Point min, max;
};

Point   p, q, add(Point, Point);
Rectangle r;
int     x, y;
```

this assignment may appear anywhere an assignment is legal:

```
r = (Rectangle){add(p, q), (Point){x, y+3}};
```

The syntax is the same as for initializing a structure but with a leading cast.

If an *anonymous structure* or *union* is declared within another structure or union, the members of the internal structure or union are addressable without prefix in the outer structure. This feature eliminates the clumsy naming of nested structures and, particularly, unions. For example, after these declarations,

```
struct Lock
{
        int     locked;
};

struct Node
{
        int     type;
        union{
                double  dval;
                double  fval;
                long    lval;
        };              /* anonymous union */
        struct Lock;    /* anonymous structure */
} *node;

void    lock(struct Lock*);
```

one may refer to `node->type`, `node->dval`, `node->fval`, `node->lval`, and `node->locked`. Moreover, the address of a `struct Node` may be used without a cast anywhere that the address of a `struct Lock` is used, such as in argument lists. The compiler automatically promotes the type and adjusts the address. Thus one may invoke `lock(node)`.

Anonymous structures and unions may be accessed by type name if (and only if) they are declared using a `typedef` name. For example, using the above declaration for `Point`, one may declare

```
struct
{
        int     type;
        Point;
} p;
```

and refer to `p.Point`.

In the initialization of arrays, a number in square brackets before an element sets the index for the initialization. For example, to initialize some elements in a table of function pointers indexed by ASCII character,

```
void    percent(void), slash(void);

void    (*func[128])(void) =
{
        ['%']   percent,
        ['/']   slash,
};
```

A similar syntax allows one to initialize structure elements:

```
Point p =
{
        .y 100,
        .x 200
};
```

These initialization syntaxes were later added to ANSI C, with the addition of an equals sign between the index or tag and the value. The Plan 9 compiler accepts either form.

Finally, the declaration

```
extern register reg;
```

(*this* appearance of the register keyword is not ignored) allocates a global register to hold the variable `reg`. External registers must be used carefully: they need to be declared in *all* source files and libraries in the program to guarantee the register is not allocated temporarily for other purposes. Especially on machines with few registers, such as the i386, it is easy to link accidentally with code that has already usurped the global registers and there is no diagnostic when this happens. Used wisely, though, external registers are powerful. The Plan 9 operating system uses them to access per-process and per-machine data structures on a multiprocessor. The storage class they provide is hard to create in other ways.

### The compile-time environment

The code generated by the compilers is 'optimized' by default: variables are placed in registers and peephole optimizations are performed. The compiler flag `-N` disables these optimizations. Registerization is done locally rather than throughout a function: whether a variable occupies a register or the memory location identified in the symbol table depends on the activity of the variable and may change throughout the life of the variable. The `-N` flag is rarely needed; its main use is to simplify debugging. There is no information in the symbol table to identify the registerization of a variable, so `-N` guarantees the variable is always where the symbol table says it is.

Another flag, `-w`, turns *on* warnings about portability and problems detected in flow analysis. Most code in Plan 9 is compiled with warnings enabled; these warnings plus the type checking offered by function prototypes provide most of the support of the Unix tool `lint` more accurately and with less chatter. Two of the warnings, 'used and not set' and 'set and not used', are almost always accurate but may be triggered spuriously by code with invisible control flow, such as in routines that call `longjmp`. The compiler statements

```
SET(v1);
USED(v2);
```

decorate the flow graph to silence the compiler. Either statement accepts a comma-separated list of variables. Use them carefully: they may silence real errors. For the common case of unused parameters to a function, leaving the name off the declaration silences the warnings. That is, listing the type of a parameter but giving it no associated variable name does the trick.

### Debugging

There are two debuggers available on Plan 9. The first, and older, is `db`, a revision of Unix `adb`. The other, `acid`, is a source-level debugger whose commands are statements in a true programming language. `Acid` is the preferred debugger, but since it borrows some elements of `db`, notably the formats for displaying values, it is worth knowing a little bit about `db`.

Both debuggers support multiple architectures in a single program; that is, the programs are `db` and `acid`, not for example `vdb` and `vacid`. They also support cross-architecture debugging comfortably: one may debug a 386 binary on a MIPS.

Imagine a program has crashed mysteriously:

```
% X11/X
Fatal server bug!
failed to create default stipple
X 106: suicide: sys: trap: fault read addr=0x0 pc=0x00105fb8
%
```

When a process dies on Plan 9 it hangs in the 'broken' state for debugging. Attach a debugger to the process by naming its process id:

```
% acid 106
/proc/106/text:mips plan 9 executable

/sys/lib/acid/port
/sys/lib/acid/mips
acid:
```

The `acid` function `stk()` reports the stack traceback:

```
acid: stk()
At pc:0x105fb8:abort+0x24 /sys/src/ape/lib/ap/stdio/abort.c:6
abort() /sys/src/ape/lib/ap/stdio/abort.c:4
        called from FatalError+#4e
                /sys/src/X/mit/server/dix/misc.c:421
FatalError(s9=#e02, s8=#4901d200, s7=#2, s6=#72701, s5=#1,
    s4=#7270d, s3=#6, s2=#12, s1=#ff37f1c, s0=#6, f=#7270f)
    /sys/src/X/mit/server/dix/misc.c:416
        called from gnotscreeninit+#4ce
                /sys/src/X/mit/server/ddx/gnot/gnot.c:792
gnotscreeninit(snum=#0, sc=#80db0)
    /sys/src/X/mit/server/ddx/gnot/gnot.c:766
        called from AddScreen+#16e
                /n/bootes/sys/src/X/mit/server/dix/main.c:610
AddScreen(pfnInit=0x0000129c,argc=0x00000001,argv=0x7fffffe4)
    /sys/src/X/mit/server/dix/main.c:530
        called from InitOutput+0x80
                /sys/src/X/mit/server/ddx/brazil/brddx.c:522
InitOutput(argc=0x00000001,argv=0x7fffffe4)
    /sys/src/X/mit/server/ddx/brazil/brddx.c:511
        called from main+0x294
                /sys/src/X/mit/server/dix/main.c:225
main(argc=0x00000001,argv=0x7fffffe4)
    /sys/src/X/mit/server/dix/main.c:136
        called from _main+0x24
                /sys/src/ape/lib/ap/mips/main9.s:8
```

The function `lstk()` is similar but also reports the values of local variables. Note that the traceback includes full file names; this is a boon to debugging, although it makes the output much noisier.

To use `acid` well you will need to learn its input language; see the ''Acid Manual'', by Phil Winterbottom, for details. For simple debugging, however, the information in the manual page is sufficient. In particular, it describes the most useful functions for examining a process.

The compiler does not place information describing the types of variables in the executable, but a compile-time flag provides crude support for symbolic debugging. The `-a` flag to the compiler suppresses code generation and instead emits source text

in the acid language to format and display data structure types defined in the program. The easiest way to use this feature is to put a rule in the mkfile:

```
syms:   main.$O
        $CC -a main.c > syms
```

Then from within acid,

```
acid: include("sourcedirectory/syms")
```

to read in the relevant definitions. (For multi-file source, you need to be a little fancier; see *8c*(1)). This text includes, for each defined compound type, a function with that name that may be called with the address of a structure of that type to display its contents. For example, if rect is a global variable of type Rectangle, one may execute

```
Rectangle(*rect)
```

to display it. The * (indirection) operator is necessary because of the way acid works: each global symbol in the program is defined as a variable by acid, with value equal to the *address* of the symbol.

Another common technique is to write by hand special acid code to define functions to aid debugging, initialize the debugger, and so on. Conventionally, this is placed in a file called acid in the source directory; it has a line

```
include("sourcedirectory/syms");
```

to load the compiler-produced symbols. One may edit the compiler output directly but it is wiser to keep the hand-generated acid separate from the machine-generated.

To make things simple, the default rules in the system mkfiles include entries to make foo.acid from foo.c, so one may use mk to automate the production of acid definitions for a given C source file.

There is much more to say here. See acid manual page, the reference manual, or the paper ‘‘Acid: A Debugger Built From A Language’’, also by Phil Winterbottom.

# Changes to the Programming Environment in the Fourth Release of Plan 9

*Rob Pike*

*rob@plan9.bell-labs.com*

## Introduction

The fourth release of Plan 9 includes changes at many levels of the system, with repercussions in the libraries and program interfaces. This document summarizes the changes and describes how existing programs must be modified to run in the new release. It is not exhaustive, of course; for further detail about any of the topics refer to the manual pages, as always.

Programmers new to Plan 9 may find valuable tidbits here, but the real audience for this paper is those with a need to update applications and servers written in C for earlier releases of the Plan 9 operating system.

## 9P, NAMELEN, and strings

The underlying file service protocol for Plan 9, 9P, retains its basic form but has had a number of adjustments to deal with longer file names and error strings, new authentication mechanisms, and to make it more efficient at evaluating file names. The change to file names affects a number of system interfaces; because file name elements are no longer of fixed size, they can no longer be stored as arrays.

9P used to be a fixed-format protocol with NAMELEN-sized byte arrays representing file name elements. Now, it is a variable-format protocol, as described in *intro*(5), in which strings are represented by a count followed by that many bytes. Thus, the string `ken` would previously have occupied 28 (NAMELEN) bytes in the message; now it occupies 5: a two-byte count followed by the three bytes of `ken` and no terminal zero. (And of course, a name could now be much longer.) A similar format change has been made to `stat` buffers: they are no longer DIRLEN bytes long but instead have variable size prefixed by a two-byte count. And in fact the entire 9P message syntax has changed: every message now begins with a message length field that makes it trivial to break the string into messages without parsing them, so `aux/fcall` is gone. A new library entry point, `read9pmsg`, makes it easy for user-level servers to break the client data stream into 9P messages. All servers should switch from using `read` (or the now gone `getS`) to using `read9pmsg`.

This change to 9P affects the way strings are handled by the kernel and throughout the system. The consequences are primarily that fixed-size arrays have been replaced by pointers and counts in a variety of system interfaces. Most programs will need at least some adjustment to the new style. In summary: NAMELEN is gone, except as a vestige in the authentication libraries, where it has been rechristened ANAMELEN. DIRLEN and ERRLEN are also gone. All programs that mention these constants will need to be fixed.

The simplest place to see this change is in the `errstr` system call, which no

longer assumes a buffer of length ERRLEN but now requires a byte-count argument:

```
char buf[...];

errstr(buf, sizeof buf);
```

The buffer can be any size you like. For convenience, the kernel stores error strings internally as 256-byte arrays, so if you like — but it's not required — you can use the defined constant ERRMAX=256 as a good buffer size. Unlike the old ERRLEN (which had value 64), ERRMAX is advisory, not mandatory, and is not part of the 9P specification.

With names, stat buffers, and directories, there isn't even an echo of a fixed-size array any more.

**Directories and wait messages**

With strings now variable-length, a number of system calls needed to change: errstr, stat, fstat, wstat, fwstat, and wait are all affected, as is read when applied to directories.

As far as directories are concerned, most programs don't use the system calls directly anyway, since they operate on the machine-independent form, but instead call the machine-dependent Dir routines dirstat, dirread, etc. These used to fill user-provided fixed-size buffers; now they return objects allocated by malloc (which must therefore be freed after use). To 'stat' a file:

```
Dir *d;

d = dirstat(filename);
if(d == nil){
        fprint(2, "can't stat %s: %r\n", filename);
        exits("stat");
}
use(d);
free(d);
```

A common new bug is to forget to free a Dir returned by dirstat.

Dirfstat and Dirfwstat work pretty much as before, but changes to 9P make it possible to exercise finer-grained control on what fields of the Dir are to be changed; see *stat*(2) and *stat*(5) for details.

Reading a directory works in a similar way to dirstat, with dirread allocating and filling in an array of Dir structures. The return value is the number of elements of the array. The arguments to dirread now include a pointer to a Dir* to be filled in with the address of the allocated array:

```
Dir *d;
int i, n;

while((n = dirread(fd, &d)) > 0){
        for(i=0; i<n; i++)
                use(&d[i]);
        free(d);
}
```

A new library function, dirreadall, has the same form as dirread but returns the entire directory in one call:

```
n = dirreadall(fd, &d)
for(i=0; i<n; i++)
        use(&d[i]);
free(d);
```

If your program insists on using the underlying `stat` system call or its relatives, or wants to operate directly on the machine-independent format returned by `stat` or `read`, it will need to be modified. Such programs are rare enough that we'll not discuss them here beyond referring to the man page *stat*(2) for details. Be aware, though, that it used to be possible to regard the buffer returned by `stat` as a byte array that began with the zero-terminated name of the file; this is no longer true. With very rare exceptions, programs that call `stat` would be better recast to use the `dir` routines or, if their goal is just to test the existence of a file, `access`.

Similar changes have affected the `wait` system call. In fact, `wait` is no longer a system call but a library routine that calls the new `await` system call and returns a newly allocated machine-dependent `Waitmsg` structure:

```
Waitmsg *w;

w = wait();
if(w == nil)
        error("wait: %r");
print("pid is %d; exit string %s\n", w->pid, w->msg);
free(w);
```

The exit string `w->msg` may be empty but it will never be a nil pointer. Again, don't forget to free the structure returned by `wait`. If all you need is the pid, you can call `waitpid`, which reports just the pid and doesn't return an allocated structure:

```
int pid;

pid = waitpid();
if(pid < 0)
        error("wait: %r");
print("pid is %d\n", pid);
```

### Quoted strings and tokenize

`Wait` gives us a good opportunity to describe how the system copes with all this free-format data. Consider the text returned by the `await` system call, which includes a set of integers (pids and times) and a string (the exit status). This information is formatted free-form; here is the statement in the kernel that generates the message:

```
n = snprint(a, n, "%d %lud %lud %lud %q",
        wq->w.pid,
        wq->w.time[TUser], wq->w.time[TSys], wq->w.time[TReal],
        wq->w.msg);
```

Note the use of %q to produce a quoted-string representation of the exit status. The %q format is like %s but will wrap `rc`-style single quotes around the string if it contains white space or is otherwise ambiguous. The library routine `tokenize` can be used to parse data formatted this way: it splits white-space-separated fields but understands the %q quoting conventions. Here is how the `wait` library routine builds its `Waitmsg` from the data returned by `await`:

```
Waitmsg*
wait(void)
{
        int n, l;
        char buf[512], *fld[5];
        Waitmsg *w;

        n = await(buf, sizeof buf-1);
        if(n < 0)
                return nil;
        buf[n] = ' ';
        if(tokenize(buf, fld, nelem(fld)) != nelem(fld)){
                werrstr("couldn't parse wait message");
                return nil;
        }
        l = strlen(fld[4])+1;
        w = malloc(sizeof(Waitmsg)+l);
        if(w == nil)
                return nil;
        w->pid = atoi(fld[0]);
        w->time[0] = atoi(fld[1]);
        w->time[1] = atoi(fld[2]);
        w->time[2] = atoi(fld[3]);
        w->msg = (char*)&w[1];
        memmove(w->msg, fld[4], l);
        return w;
}
```

This style of quoted-string and `tokenize` is used all through the system now. In particular, devices now `tokenize` the messages written to their `ctl` files, which means that you can send messages that contain white space, by quoting them, and that you no longer need to worry about whether or not the device accepts a newline. In other words, you can say

```
echo message > /dev/xx/ctl
```

instead of `echo -n` because `tokenize` treats the newline character as white space and discards it.

While we're on the subject of quotes and strings, note that the implementation of `await` used `snprint` rather than `sprint`. We now deprecate `sprint` because it has no protection against buffer overflow. We prefer `snprint` or `seprint`, to constrain the output. The %q format is cleverer than most in this regard: if the string is too long to be represented in full, %q is smart enough to produce a truncated but correctly quoted string within the available space.

**Mount**

Although strings in 9P are now variable-length and not zero-terminated, this has little direct effect in most of the system interfaces. File and user names are still zero-terminated strings as always; the kernel does the work of translating them as necessary for transport. And of course, they are now free to be as long as you might want; the only hard limit is that their length must be represented in 16 bits.

One example where this matters is that the file system specification in the `mount` system call can now be much longer. Programs like `rio` that used the specification string in creative ways were limited by the NAMELEN restriction; now they can use the string more freely. `Rio` now accepts a simple but less cryptic specification language for the window to be created by the `mount` call, e.g.:

```
% mount $wsys /mnt/wsys 'new -dx 250 -dy 250 -pid 1234'
```

In the old system, this sort of control was impossible through the `mount` interface.

While we're on the subject of `mount`, note that with the new security architecture (see *factotum*(4)), 9P has moved its authentication outside the protocol proper. (For a full description of this change to 9P, see *fauth*(2), *attach*(5), and the paper *Security in Plan 9*.) The most explicit effect of this change is that `mount` now takes another argument, `afd`, a file descriptor for the authentication file through which the authentication will be made. For most user-level file servers, which do not require authentication, it is sufficient to provide `-1` as the value of `afd`:

```
if(mount(fd, -1, "/mnt/wsys", MREPL,
   "new -dx 250 -dy 250 -pid 1234") < 0)
        error("mount failed: %r");
```

To connect to servers that require authentication, use the new `fauth` system call or the reimplemented `amount` (authenticated mount) library call. In fact, since `amount` handles both authenticating and non-authenticating servers, it is often easiest just to replace calls to `mount` by calls to `amount`; see *auth*(2) for details.

**Print**

The C library has been heavily reworked in places. Besides the changes mentioned above, it now has a much more complete set of routines for handling `Rune` strings (that is, zero-terminated arrays of 16-bit character values). The most sweeping changes, however, are in the way formatted I/O is performed.

The `print` routine and all its relatives have been reimplemented to offer a number of improvements:

(1) Better buffer management, including the provision of an internal flush routine, makes it unnecessary to provide large buffers. For example, `print` uses a much smaller buffer now (reducing stack load) while simultaneously removing the need to truncate the output string if it doesn't fit in the buffer.

(2) Global variables have been eliminated so no locking is necessary.

(3) The combination of (1) and (2) means that the standard implementation of `print` now works fine in threaded programs, and `threadprint` is gone.

(4) The new routine `smprint` prints into, and returns, storage allocated on demand by `malloc`.

(5) It is now possible to print into a `Rune` string; for instance, `runesmprint` is the `Rune` analog of `smprint`.

(6) There is improved support for custom print verbs and custom output routines such as error handlers. The routine `doprint` is gone, but `vseprint` can always be used instead. However, the new routines `fmtfdinit`, `fmtstrinit`, `fmtprint`, and friends are often a better replacement. The details are too long for exposition here; *fmtinstall*(2) explains the new interface and provides examples.

(7) Two new format flags, space and comma, close somewhat the gap between Plan 9 and ANSI C.

Despite these changes, most programs will be unaffected; `print` is still `print`. Don't forget, though, that you should eliminate calls to `sprint` and use the `%q` format when appropriate.

**Binary compatibility**

The discussion so far has been about changes at the source level. Existing binaries will probably run without change in the new environment, since the kernel provides backward-compatible system calls for `errstr`, `stat`, `wait`, etc. The only exceptions are programs that do either a `mount` system call, because of the security changes and because the file descriptor in `mount` must point to a new 9P connection; or a `read` system call on a directory, since the returned data will be in the new format. A moment's reflection will discover that this means old user-level file servers will need to be fixed to run on the new system.

**File servers**

A full description of what user-level servers must do to provide service with the new 9P is beyond the scope of this paper. Your best source of information is section 5 of the manual, combined with study of a few examples. `/sys/src/cmd/ramfs.c` is a simple example; it has a counterpart `/sys/src/lib9p/ramfs.c` that implements the same service using the new *9p*(2) library.

That said, it's worth summarizing what to watch for when converting a file server. The `session` message is gone, and there is a now a `version` message that is exchanged at the start of a connection to establish the version of the protocol to use (there's only one at the moment, identified by the string `9P2000`) and what the maximum message size will be. This negotiation makes it easier to handle 9P encapsulation, such as with `exportfs`, and also permits larger message sizes when appropriate.

If your server wants to authenticate, it will need to implement an authentication file and implement the `auth` message; otherwise it should return a helpful error string to the `Tauth` request to signal that authentication is not required.

The handling of `stat` and directory reads will require some changes but they should not be fundamental. Be aware that seeking on directories is forbidden, so it is fine if you disregard the file offset when implementing directory reads; this makes it a little easier to handle the variable-length entries. You should still never return a partial directory entry; if the I/O count is too small to return even one entry, you should return two bytes containing the byte count required to represent the next entry in the directory. User code can use this value to formulate a retry if it desires. See the DIAGNOSTICS section of *stat*(2) for a description of this process.

The trickiest part of updating a file server is that the `clone` and `walk` messages have been merged into a single message, a sort of 'clone-multiwalk'. The new message, still called `walk`, proposes a sequence of file name elements to be evaluated using a possibly cloned fid. The return message contains the qids of the files reached by walking to the sequential elements. If all the elements can be walked, the fid will be cloned if requested. If a non-zero number of elements are requested, but none can be walked, an error should be returned. If only some can be walked, the fid is not cloned, the original fid is left where it was, and the returned `Rwalk` message should contain the partial list of successfully reached qids. See *walk*(5) for a full description.

# APE — The ANSI/POSIX Environment

*Howard Trickey*
*howard@plan9.bell-labs.com*

## Introduction

When a large or frequently-updated program must be ported to or from Plan 9, the ANSI/POSIX environment known as APE can be useful. APE combines the set of headers and object code libraries specified by the ANSI C standard (ANSI X3.159-1989) with the POSIX operating system interface standard (IEEE 1003.1-1990, ISO 9945-1), the part of POSIX defining the basic operating system functions. Using APE will cause slower compilation and marginally slower execution speeds, so if the importing or exporting happens only infrequently, due consideration should be given to using the usual Plan 9 compilation environment instead. Another factor to consider is that the Plan 9 header organization is much simpler to remember and use.

There are some aspects of required POSIX behavior that are impossible or very hard to simulate in Plan 9. They are described below. Experience has shown, however, that the simulation is adequate for the vast majority of programs. A much more common problem is that many programs use functions or headers not defined by POSIX. APE has some extensions to POSIX to help in this regard. Extensions must be explicitly enabled with an appropriate `#define`, in order that the APE environment be a good aid for testing ANSI/POSIX compliance of programs.

## Pcc

The `pcc` command acts as a front end to the Plan 9 C compilers and loaders. It runs an ANSI C preprocessor over source files, using the APE headers to satisfy `#include` <*file*> directives; then it runs a Plan 9 C compiler; finally, it may load with APE libraries to produce an executable program. The document *How to Use the Plan 9 C Compiler* explains how environment variables are used by convention to handle compilation for differing architectures. The environment variable `$objtype` controls which Plan 9 compiler and loader are used by `pcc`, as well as the location of header and library files. For example, if `$objtype` is `mips`, then `pcc` has `cpp` look for headers in `/mips/include/ape` followed by `/sys/include/ape`; then `pcc` uses `vc` to create `.v` object files; finally, `vl` is used to create an executable using libraries in `/mips/lib/ape`.

## Psh and Cc

The `pcc` command is intended for uses where the source code is ANSI/POSIX, but the programs are built in the usual Plan 9 manner — with `mk` and producing object files with names ending in `.v`, etc. Sometimes it is best to use the standard POSIX `make` and `cc` (which produces object files with names ending in `.o`, and automatically calls the loader unless `-c` is specified). Under these circumstances, execute the command:

```
ape/psh
```

This starts a POSIX shell, with an environment that includes the POSIX commands `ar89`, `c89`, `cc`, `basename`, `dirname`, `expr`, `false`, `grep`, `kill`, `make`, `rmdir`, `sed`,

sh, stty, true, uname, and yacc. There are also a few placeholders for commands that cannot be implemented in Plan 9: chown, ln, and umask.

The cc command accepts the options mandated for the POSIX command c89, as specified in the C-Language Development Utilities Option annex of the POSIX Shell and Utilities standard. It also accepts the following nonstandard options: -v for echoing the commands for each pass to stdout; -A to turn on ANSI prototype warnings; -S to leave assembly language in *file*.s; -Wp,*args* to pass *args* to the cpp; -W0,*args* to pass *args* to 2c, etc.; and -Wl,*args* to pass *args* to 2l, etc.

The sh command is pdksh, a mostly POSIX-compliant public domain Korn Shell. The Plan 9 implementation does not include the emacs and vi editing modes.

The stty command only has effect if the ape/ptyfs command has been started to interpose a pseudo-tty interface between /dev/cons and the running command. None of the distributed commands do this automatically.

**Symbols**

The C and POSIX standards require that certain symbols be defined in headers. They also require that certain other classes of symbols not be defined in the headers, and specify certain other symbols that may be defined in headers at the discretion of the implementation. POSIX defines *feature test macros*, which are preprocessor symbols beginning with an underscore and then a capital letter; if the program #defines a feature test macro before the inclusion of any headers, then it is requesting that certain symbols be visible in the headers. The most important feature test macro is _POSIX_SOURCE: when it is defined, exactly the symbols required by POSIX are visible in the appropriate headers. Consider <signal.h> for example: ANSI defines some names that must be defined in <signal.h>, but POSIX defines others, such as sigset_t, which are not allowed according to ANSI. The solution is to make the additional symbols visible only when _POSIX_SOURCE is defined.

To export a program, it helps to know whether it fits in one of the following categories:

1. Strictly conforming ANSI C program. It only uses features of the language, libraries, and headers explicitly required by the C standard. It does not depend on unspecified, undefined, or implementation-dependent behavior, and does not exceed any minimum implementation limit.
2. Strictly conforming POSIX program. Similar, but for the POSIX standard as well.
3. Some superset of POSIX, with extensions. Each extension is selected by a feature test macro, so it is clear which extensions are being used.

With APE, if headers are always included to declare any library functions used, then the set of feature test macros defined by a program will show which of the above categories the program is in. To accomplish this, no symbol is defined in a header if it is not required by the C or POSIX standard, and those required by the POSIX standard are protected by #ifdef _POSIX_SOURCE. For example, <errno.h> defines EDOM, ERANGE, and errno, as required by the C standard. The C standard allows more names beginning with E, but our header defines only those unless _POSIX_SOURCE is defined, in which case the symbols required by POSIX are also defined. This means that a program that uses ENAMETOOLONG cannot masquerade as a strictly conforming ANSI C program.

Pcc and cc do not predefine any preprocessor symbols except those required by the ANSI C standard: __STDC__, __LINE__, __FILE__, __DATE__, and __TIME__. Any others must be defined in the program itself or by using -D on the command line.

**Extensions**

The discipline enforced by putting only required names in the headers is useful for exporting programs, but it gets in the way when importing programs. The compromise is to allow additional symbols in headers, additional headers, and additional library functions, but only under control of extension feature test macros. The following extensions are provided; unless otherwise specified, the additional library functions are in the default APE library.

- _LIBG_EXTENSION. This allows the use of the Plan 9 graphics library. The functions are as described in the Plan 9 manual (see *graphics*(2)) except that `div` had to be renamed `ptdiv`. Include the <libg.h> header to declare the needed types and functions.
- _LIMITS_EXTENSION. POSIX does not require that names such as PATH_MAX and OPEN_MAX be defined in <limits.h>, but many programs assume they are defined there. If _LIMITS_EXTENSION is defined, those names will all be defined when <limits.h> is included.
- _BSD_EXTENSION. This extension includes not only Berkeley Unix routines, but also a grab bag of other miscellaneous routines often found in Unix implementations. The extension allows the inclusion of any of: <bsd.h> for `bcopy()`, `bcmp()`, and similar Berkeley functions; <netdb.h> for `gethostbyname()`, etc., and associated structures; <select.h> for the Berkeley `select` function and associated types and macros for dealing with multiple input sources; <sys/ioctl.h> for the `ioctl` function (minimally implemented); <sys/param.h> for NOFILES_MAX; <sys/pty.h> for pseudo-tty support via the `ptsname(int)` and `ptmname(int)` functions; <sys/resource.h>; <sys/socket.h> for socket structures, constants, and functions; <sys/time.h> for definitions of the `timeval` and `timezone` structures; and <sys/uio.h> for the `iovec` structure and the `writev` and `readv` functions used for scatter/gather I/O. Defining _BSD_EXTENSION also enables various extra definitions in <ctype.h>, <signal.h>, <stdio.h>, <unistd.h>, <sys/stat.h>, and <sys/times.h>.
- _NET_EXTENSION. This extension allows inclusion of <libnet.h>, which defines the networking functions described in the Plan 9 manual page *dial*(2).
- _PLAN9_EXTENSION. This extension allows inclusion of <u.h>, <lock.h>, <qlock.h>, <utf.h>, <fmt.h>, and <draw.h>. These are pieces of Plan 9 source code ported into APE, mostly from <libc.h>.
- _REGEXP_EXTENSION. This extension allows inclusion of <regexp.h>, which defines the regular expression matching functions described in the Plan 9 manual page *regexp*(2).
- _RESEARCH_SOURCE. This extension enables a small library of functions from the Tenth Edition Unix Research System (V10). These functions and the types needed to use them are all defined in the <libv.h> header. The provided functions are: `srand`, `rand`, `nrand`, `lrand`, and `frand` (better random number generators); `getpass`, `tty_echoon`, `tty_echooff` (for dealing with the common needs for mucking with terminal characteristics); `min` and `max`; `nap`; and `setfields`, `getfields`, and `getmfields` (for parsing a line into fields). See the Research Unix System Programmer's Manual, Tenth Edition, for a description of these functions.
- _C99_SNPRINTF_EXTENSION. This extension permits the use of the return values of *snprintf* and *vsnprintf*. Before C99, the 1999 C standard, these functions usually returned the number of bytes, excluding terminating NUL, actually stored in the target string. (GNU, as usual, had to be different and returned -1 if the target string was too small.) C99 requires them to instead return the number of bytes,

excluding terminating NUL, that would have been written into the target string if it were infinitely large or a negative value if an 'encoding error' occurs, so old programs compiled under C99 rules will be prone to overrunning their buffers. This extension is a way for the programmer to declare that he or she understands the situation and has adjusted the code being compiled to compensate.

**Common Problems**

Some large systems, including X11, have been ported successfully to Plan 9 using APE (the X11 port is not included in the distribution, however, because supporting it properly is too big a job). The problems encountered fall into three categories: (1) non-ANSI C/POSIX features used; (2) inadequate simulation of POSIX functions; and (3) compiler/loader bugs. By far the majority of problems are in the first category.

POSIX is just starting to be a target for programmers. Most existing code is written to work with one or both of a BSD or a System V Unix. System V is fairly close to POSIX, but there are some differences. Also, many System V systems have imported some BSD features that are not part of POSIX. A good strategy for porting external programs is to first try using `CFLAGS=-D_POSIX_SOURCE`; if that doesn't work, try adding `_D_BSD_EXTENSION` and perhaps include `<bsd.h>` in source files. Here are some solutions to problems that might remain:

- Third (environment) argument to `main`. Use the `environ` global instead.
- `OPEN_MAX`, `PATH_MAX`, etc., assumed in `<limits.h>`. Rewrite to call `sysconf` or define `_LIMITS_EXTENSION`.
- `<varargs.h>`. Rewrite to use `<stdarg.h>`.

The second class of problems has to do with inadequacies in the Plan 9 simulation of POSIX functions. These shortcomings have rarely gotten in the way (except, perhaps, for the `link` problem).

- Functions for setting the userid, groupid, effective userid and effective groupid do not do anything useful. The concept is impossible to simulate in Plan 9. `Chown` also does nothing.
- `execlp` and the related functions do not look at the `PATH` environment variable. They just try the current directory and `/bin` if the pathname is not absolute.
- Advisory locking via `fcntl` is not implemented.
- `isatty` is hard to do correctly. The approximation used is only sometimes correct.
- `link` always fails.
- With `open`, the `O_NOCTTY` option has no effect. The concept of a controlling tty is foreign to Plan 9.
- `setsid` forks the name space and note group, which is only approximately the right behavior.
- The functions dealing with stacking signals, `sigpending`, `sigprocmask` and `sigsuspend`, do not work.
- `umask` has no effect, as there is no such concept in Plan 9.
- code that does `getenv("HOME")` should be changed to `getenv("home")` on Plan 9.

# Acid: A Debugger Built From A Language

*Phil Winterbottom*
*philw@plan9.bell-labs.com*

*ABSTRACT*

Acid is an unusual source-level symbolic debugger for Plan 9. It is implemented as a language interpreter with specialized primitives that provide debugger support. Programs written in the language manipulate one or more target processes; variables in the language represent the symbols, state, and resources of those processes. This structure allows complex interaction between the debugger and the target program and provides a convenient method of parameterizing differences between machine architectures. Although some effort is required to learn the debugging language, the richness and flexibility of the debugging environment encourages new ways of reasoning about the way programs run and the conditions under which they fail.

## 1. Introduction

The size and complexity of programs have increased in proportion to processor speed and memory but the interface between debugger and programmer has changed little. Graphical user interfaces have eased some of the tedious aspects of the interaction. A graphical interface is a convenient means for navigating through source and data structures but provides little benefit for process control. The introduction of a new concurrent language, Alef [Win93], emphasized the inadequacies of the existing Plan 9 [Pike90] debugger *db*, a distant relative of *adb*, and made it clear that a new debugger was required.

Current debuggers like *dbx*, *sdb*, and *gdb* are limited to answering only the questions their authors envisage. As a result, they supply a plethora of specialized commands, each attempting to anticipate a specific question a user may ask. When a debugging situation arises that is beyond the scope of the command set, the tool is useless. Further, it is often tedious or impossible to reproduce an anomalous state of the program, especially when the state is embedded in the program's data structures.

Acid applies some ideas found in CAD software used for hardware test and simulation. It is based on the notion that the state and resources of a program are best represented and manipulated by a language. The state and resources, such as memory, registers, variables, type information and source code are represented by variables in the language. Expressions provide a computation mechanism and control statements allow repetitive or selective interpretation based on the result of expression evaluation. The heart of the Acid debugger is an interpreter for a small typeless language whose operators mirror the operations of C and Alef, which in turn correspond well to the basic operations of the machine. The interpreter itself knows nothing of the underlying hardware; it deals with the program state and resources in the abstract. Fundamental routines to control processes, read files, and interface to the system are implemented as builtin

Originally appeared in *Proc. of the Winter 1994 USENIX Conf.,* pp. 211-222, San Francisco, CA

functions available to the interpreter. The actual debugger functionality is coded in Acid; commands are implemented as Acid functions.

This language-based approach has several advantages. Most importantly, programs written in Acid, including most of the debugger itself, are inherently portable. Furthermore, Acid avoids the limitations other debuggers impose when debugging parallel programs. Instead of embedding a fixed process model in the debugger, Acid allows the programmer to adapt the debugger to handle an arbitrary process partitioning or program structure. The ability to interact dynamically with an executing process provides clear advantages over debuggers constrained to probe a static image. Finally, the Acid language is a powerful vehicle for expressing assertions about logic, process state, and the contents of data structures. When combined with dynamic interaction it allows a limited form of automated program verification without requiring modification or recompilation of the source code. The language is also an excellent vehicle for preserving a test suite for later regression testing.

The debugger may be customized by its users; standard functions may be modified or extended to suit a particular application or preference. For example, the kernel developers in our group require a command set supporting assembler-level debugging while the application programmers prefer source-level functionality. Although the default library is biased toward assembler-level debugging, it is easily modified to provide a convenient source-level interface. The debugger itself does not change; the user combines primitives and existing Acid functions in different ways to implement the desired interface.

## 2. Related Work

DUEL [Gol93], an extension to *gdb* [Stal91], proposes using a high level expression evaluator to solve some of these problems. The evaluator provides iterators to loop over data structures and conditionals to control evaluation of expressions. The author shows that complex state queries can be formulated by combining concise expressions but this only addresses part of the problem. A program is a dynamic entity; questions asked when the program is in a static state are meaningful only after the program has been 'caught' in that state. The framework for manipulating the program is still as primitive as the underlying debugger. While DUEL provides a means to probe data structures it entirely neglects the most beneficial aspect of debugging languages: the ability to control processes. Acid is structured around a thread of control that passes between the interpreter and the target program.

The NeD debugger [May92] is a set of extensions to TCL [Ous90] that provide debugging primitives. The resulting language, NeDtcl, is used to implement a portable interface between a conventional debugger, pdb [May90], and a server that executes NeDtcl programs operating on the target program. Execution of the NeDtcl programs implements the debugging primitives that pdb expects. NeD is targeted at multi-process debugging across a network, and proves the flexibility of a language as a means of communication between debugging tools. Whereas NeD provides an interface between a conventional debugger and the process it debugs, Acid is the debugger itself. While NeD has some of the ideas found in Acid it is targeted toward a different purpose. Acid seeks to integrate the manipulation of a program's resources into the debugger while NeD provides a flexible interconnect between components of the debugging environment. The choice of TCL is appropriate for its use in NeD but is not suitable for Acid. Acid relies on the coupling of the type system with expression evaluation, which are the root of its design, to provide the debugging primitives.

Dalek [Ols90] is an event based language extension to gdb. State transitions in the target program cause events to be queued for processing by the debugging language.

Acid has many of the advantages of same process or *local agent* debuggers, like Parasight [Aral], without the need for dynamic linking or shared memory. Acid improves

on the ideas of these other systems by completely integrating all aspects of the debugging process into the language environment. Of particular importance is the relationship between Acid variables, program symbols, source code, registers and type information. This integration is made possible by the design of the Acid language.

Interpreted languages such as Lisp and Smalltalk are able to provide richer debugging environments through more complete information than their compiled counterparts. Acid is a means to gather and represent similar information about compiled programs through cooperation with the compilation tools and library implementers.

## 3. Acid the Language

Acid is a small interpreted language targeted to its debugging task. It focuses on representing program state and addressing data rather than expressing complex computations. Program state is *addressable* from an Acid program. In addition to parsing and executing expressions and providing an architecture-independent interface to the target process, the interpreter supplies a mark-and-scan garbage collector to manage storage.

Every Acid session begins with the loading of the Acid libraries. These libraries contain functions, written in Acid, that provide a standard debugging environment including breakpoint management, stepping by instruction or statement, stack tracing, and access to variables, memory, and registers. The library contains 600 lines of Acid code and provides functionality similar to *dbx*. Following the loading of the system library, Acid loads user-specified libraries; this load sequence allows the user to augment or override the standard commands to customize the debugging environment. When all libraries are loaded, Acid issues an interactive prompt and begins evaluating expressions entered by the user. The Acid 'commands' are actually invocations of builtin primitives or previously defined Acid functions. Acid evaluates each expression as it is entered and prints the result.

## 4. Types and Variables

Acid variables are of four basic types: *integer*, *string*, *float*, and *list*. The type of a variable is inferred by the type of the right-hand side of an assignment expression. Many of the operators can be applied to more than one type; for these operators the action of the operator is determined by the type of its operands. For example, the + operator adds *integer* and *float* operands, and concatenates *string* and *list* operands. Lists are the only complex type in Acid; there are no arrays, structures or pointers. Operators provide `head`, `tail`, `append` and `delete` operations. Lists can also be indexed like arrays.

Acid has two levels of scope: global and local. Function parameters and variables declared in a function body using the `local` keyword are created at entry to the function and exist for the lifetime of a function. Global variables are created by assignment and need not be declared. All variables and functions in the program being debugged are entered in the Acid symbol table as global variables during Acid initialization. Conflicting variable names are resolved by prefixing enough '$' characters to make them unique. Syntactically, Acid variables and target program symbols are referenced identically. However, the variables are managed differently in the Acid symbol table and the user must be aware of this distinction. The value of an Acid variable is stored in the symbol table; a reference returns the value. The symbol table entry for a variable or function in the target program contains the address of that symbol in the image of the program. Thus, the value of a program variable is accessed by indirect reference through the Acid variable that has the same name; the value of an Acid variable is the address of the corresponding program variable.

## 5. Control Flow

The `while` and `loop` statements implement looping. The former is similar to the same statement in C. The latter evaluates starting and ending expressions yielding integers and iterates while an incrementing loop index is within the bounds of those expressions.

```
acid: i = 0; loop 1,5 do print(i=i+1)
0x00000001
0x00000002
0x00000003
0x00000004
0x00000005
acid:
```

The traditional `if-then-else` statement implements conditional execution.

## 6. Addressing

Two indirection operators allow Acid to access values in the program being debugged. The * operator fetches a value from the memory image of an executing process; the @ operator fetches a value from the text file of the process. When either operator appears on the left side of an assignment, the value is written rather than read.

The indirection operator must know the size of the object referenced by a variable. The Plan 9 compilers neglect to include this information in the program symbol table, so Acid cannot derive this information implicitly. Instead Acid variables have formats. The format is a code letter specifying the printing style and the effect of some of the operators on that variable. The indirection operators look at the format code to determine the number of bytes to read or write. The format codes are derived from the format letters used by *db*. By default, symbol table variables and numeric constants are assigned the format code `'X'` which specifies 32-bit hexadecimal. Printing such a variable yields output of the form `0x00123456`. An indirect reference through the variable fetches 32 bits of data at the address indicated by the variable. Other formats specify various data types, for example `i` an instruction, `D` a signed 32 bit decimal, `s` a null-terminated string. The `fmt` function allows the user to change the format code of a variable to control the printing format and operator side effects. This function evaluates the expression supplied as the first argument, attaches the format code supplied as the second argument to the result and returns that value. If the result is assigned to a variable, the new format code applies to that variable. For convenience, Acid provides the \ operator as a shorthand infix form of `fmt`. For example:

```
acid: x=10
acid: x                          // print x in hex
0x0000000a
acid: x = fmt(x, 'D')            // make x type decimal
acid: print(x, fmt(x, 'X'), x\X) // print x in decimal & hex
10 0x0000000a 0x0000000a
acid: x                          // print x in decimal
10
acid: x\o                        // print x in octal
000000000012
```

The ++ and -- operators increment or decrement a variable by an amount determined by its format code. Some formats imply a non-fixed size. For example, the `i` format code disassembles an instruction into a string. On a 68020, which has variable length instructions:

```
acid: p=main\i                          // p=addr(main), type INST
acid: loop 1,5 do print(p\X, @p++) // disassemble 5 instr's
0x0000222e LEA  0xffffe948(A7),A7
0x00002232 MOVL s+0x4(A7),A2
0x00002236 PEA  0x2f($0)
0x0000223a MOVL A2,-(A7)
0x0000223c BSR  utfrrune
acid:
```

Here, `main` is the address of the function of the same name in the program under test. The loop retrieves the five instructions beginning at that address and then prints the address and the assembly language representation of each. Notice that the stride of the increment operator varies with the size of the instruction: the `MOVL` at `0x0000223a` is a two byte instruction while all others are four bytes long.

Registers are treated as normal program variables referenced by their symbolic assembler language names. When a process stops, the register set is saved by the kernel at a known virtual address in the process memory map. The Acid variables associated with the registers point to the saved values and the * indirection operator can then be used to read and write the register set. Since the registers are accessed via Acid variables they may be used in arbitrary expressions.

```
acid: PC                                // addr of saved PC
0xc0000f60
acid: *PC
0x0000623c                              // contents of PC
acid: *PC\a
main
acid: *R1=10                            // modify R1
acid: asm(*PC+4)                        // disassemble @ PC+4
main+0x4 0x00006240     MOVW    R31,0x0(R29)
main+0x8 0x00006244     MOVW    $setR30(SB),R30
main+0x10 0x0000624c    MOVW    R1,_clock(SB)
```

Here, the saved `PC` is stored at address `0xc0000f60`; its current content is `0x0000623c`. The 'a' format code converts this value to a string specifying the address as an offset beyond the nearest symbol. After setting the value of register `1`, the example uses the `asm` command to disassemble a short section of code beginning at four bytes beyond the current value of the `PC`.

### 7. Process Interface

A program executing under Acid is monitored through the *proc* file system interface provided by Plan 9. Textual messages written to the `ctl` file control the execution of the process. For example writing `waitstop` to the control file causes the write to block until the target process enters the kernel and is stopped. When the process is stopped the write completes. The `startstop` message starts the target process and then does a `waitstop` action. Synchronization between the debugger and the target process is determined by the actions of the various messages. Some operate asynchronously to the target process and always complete immediately, others block until the action completes. The asynchronous messages allow Acid to control several processes simultaneously.

The interpreter has builtin functions named after each of the control messages. The functions take a process id as argument. Any time a control message causes the program to execute instructions the interpreter performs two actions when the control operation has completed. The Acid variables pointing at the register set are fixed up to point at the saved registers, and then the user defined function `stopped` is executed. The `stopped` function may print the current address, line of source or instruction and return to interactive mode. Alternatively it may traverse a complex data structure, gather

statistics and then set the program running again.

Several Acid variables are maintained by the debugger rather than the programmer. These variables allow generic Acid code to deal with the current process, architecture specifics or the symbol table. The variable `pid` is the process id of the current process Acid is debugging. The variable `symbols` contains a list of lists where each sublist contains the symbol name, its type and the value of the symbol. The variable `registers` contains a list of the machine-specific register names. Global symbols in the target program can be referenced directly by name from Acid. Local variables are referenced using the colon operator as `function:variable`.

## 8. Source Level Debugging

Acid provides several builtin functions to manipulate source code. The `file` function reads a text file, inserting each line into a list. The `pcfile` and `pcline` functions each take an address as an argument. The first returns a string containing the name of the source file and the second returns an integer containing the line number of the source line containing the instruction at the address.

```
acid: pcfile(main)              // file containing main
main.c
acid: pcline(main)              // line # of main in source
11
acid: file(pcfile(main))[pcline(main)]  // print that line
main(int argc, char *argv[])
acid: src(*PC)                  // print statements nearby
 9
 10 void
>11 main(int argc, char *argv[])
 12 {
 13     int a;
```

In this example, the three primitives are combined in an expression to print a line of source code associated with an address. The `src` function prints a few lines of source around the address supplied as its argument. A companion routine, `Bsrc`, communicates with the external editor `sam`. Given an address, it loads the corresponding source file into the editor and highlights the line containing the address. This simple interface is easily extended to more complex functions. For example, the `step` function can select the current file and line in the editor each time the target program stops, giving the user a visual trace of the execution path of the program. A more complete interface allowing two way communication between Acid and the `acme` user interface [Pike93] is under construction. A filter between the debugger and the user interface provides interpretation of results from both sides of the interface. This allows the programming environment to interact with the debugger and vice-versa, a capability missing from the `sam` interface. The `src` and `Bsrc` functions are both written in Acid code using the file and line primitives. Acid provides library functions to step through source level statements and functions. Furthermore, addresses in Acid expressions can be specified by source file and line. Source code is manipulated in the Acid *list* data type.

## 9. The Acid Library

The following examples define some useful commands and illustrate the interaction of the debugger and the interpreter.

```
defn bpset(addr)                          // set breakpoint
{
        if match(addr, bplist) >= 0 then
                print("bkpoint already set:", addr\a, "\n");
        else {
                *fmt(addr, bpfmt) = bpinst;   // plant it
                bplist = append bplist, addr; // add to list
        }
}
```

The bpset function plants a break point in memory. The function starts by using the match builtin to search the breakpoint list to determine if a breakpoint is already set at the address. The indirection operator, controlled by the format code returned by the fmt primitive, is used to plant the breakpoint in memory. The variables bpfmt and bpinst are Acid global variables containing the format code specifying the size of the breakpoint instruction and the breakpoint instruction itself. These variables are set by architecture-dependent library code when the debugger first attaches to the executing image. Finally the address of the breakpoint is appended to the breakpoint list, bplist.

```
defn step()                               // single step
{
        local lst, lpl, addr, bput;

        bput = 0;                         // sitting on bkpoint
        if match(*PC, bplist) >= 0 then {
                bput = fmt(*PC, bpfmt); // save current addr
                *bput = @bput;          // replace it
        }

        lst = follow(*PC);                // get follow set

        lpl = lst;
        while lpl do {                    // place breakpoints
                *(head lpl) = bpinst;
                lpl = tail lpl;
        }

        startstop(pid);                   // do the step

        while lst do {                    // remove breakpoints
                addr = fmt(head lst, bpfmt);
                *addr = @addr;          // replace instr.
                lst = tail lst;
        }
        if bput != 0 then
                *bput = bpinst;         // restore breakpoint
}
```

The step function executes a single assembler instruction. If the PC is sitting on a breakpoint, the address and size of the breakpoint are saved. The breakpoint instruction is then removed using the @ operator to fetch bpfmt bytes from the text file and to place it into the memory of the executing process using the * operator. The follow function is an Acid builtin which returns a follow-set: a list of instruction addresses which could be executed next. If the instruction stored at the PC is a branch instruction, the list contains the addresses of the next instruction and the branch destination; otherwise, it contains only the address of the next instruction. The follow-set is then used to replace each possible following instruction with a breakpoint instruction. The original instructions need not be saved; they remain in their unaltered state in the

text file. The `startstop` builtin writes the 'startstop' message to the *proc* control file for the process named `pid`. The target process executes until some condition causes it to enter the kernel, in this case, the execution of a breakpoint. When the process blocks, the debugger regains control and invokes the Acid library function `stopped` which reports the address and cause of the blockage. The `startstop` function completes and returns to the `step` function where the follow-set is used to replace the breakpoints placed earlier. Finally, if the address of the original `PC` contained a breakpoint, it is replaced.

Notice that this approach to process control is inherently portable; the Acid code is shared by the debuggers for all architectures. Acid variables and builtin functions provide a transparent interface to architecture-dependent values and functions. Here the breakpoint value and format are referenced through Acid variables and the `follow` primitive masks the differences in the underlying instruction set.

The `next` function, similar to the *dbx* command of the same name, is a simpler example. This function steps through a single source statement but steps over function calls.

```
defn next()
{
        local sp, bound;

        sp = *SP;                           // save starting SP
        bound = fnbound(*PC);               // begin & end of fn.
        stmnt();                            // step 1 statement
        pc = *PC;
        if pc >= bound[0] && pc < bound[1] then
                return {};

        while (pc<bound[0] || pc>bound[1]) && sp>=*SP do {
                step();
                pc = *PC;
        }
        src(*PC);
}
```

The `next` function starts by saving the current stack pointer in a local variable. It then uses the Acid library function `fnbound` to return the addresses of the first and last instructions in the current function in a list. The `stmnt` function executes a single source statement and then uses `src` to print a few lines of source around the new `PC`. If the new value of the `PC` remains in the current function, `next` returns. When the executed statement is a function call or a return from a function, the new value of the `PC` is outside the bounds calculated by `fnbound` and the test of the `while` loop is evaluated. If the statement was a return, the new value of the stack pointer is greater than the original value and the loop completes without execution. Otherwise, the loop is entered and instructions are continually executed until the value of the `PC` is between the bounds calculated earlier. At that point, execution ceases and a few lines of source in the vicinity of the `PC` are printed.

Acid provides concise and elegant expression for control and manipulation of target programs. These examples demonstrate how a few well-chosen primitives can be combined to create a rich debugging environment.

## 10. Dealing With Multiple Architectures

A single binary of Acid may be used to debug a program running on any of the five processor architectures supported by Plan 9. For example, Plan 9 allows a user on a MIPS to import the *proc* file system from an i486-based PC and remotely debug a program executing on that processor.

Two levels of abstraction provide this architecture independence. On the lowest level, a Plan 9 library supplies functions to decode the file header of the program being debugged and select a table of system parameters and a jump vector of architecture-dependent functions based on the magic number. Among these functions are byte-order-independent access to memory and text files, stack manipulation, disassembly, and floating point number interpretation. The second level of abstraction is supplied by Acid. It consists of primitives and approximately 200 lines of architecture-dependent Acid library code that interface the interpreter to the architecture-dependent library. This layer performs functions such as mapping register names to memory locations, supplying breakpoint values and sizes, and converting processor specific data to Acid data types. An example of the latter is the stack trace function `strace`, which uses the stack traversal functions in the architecture-dependent library to construct a list of lists describing the context of a process. The first level of list selects each function in the trace; subordinate lists contain the names and values of parameters and local variables of the functions. Acid commands and library functions that manipulate and display process state information operate on the list representation and are independent of the underlying architecture.

## 11. Alef Runtime

Alef is a concurrent programming language, designed specifically for systems programming, which supports both shared variable and message passing paradigms. Alef borrows the C expression syntax but implements a substantially different type system. The language provides a rich set of exception handling, process management, and synchronization primitives, which rely on a runtime system. Alef program bugs are often deadlocks, synchronization failures, or non-termination caused by locks being held incorrectly. In such cases, a process stalls deep in the runtime code and it is clearly unreasonable to expect a programmer using the language to understand the detailed internal semantics of the runtime support functions.

Instead, there is an Alef support library, coded in Acid, that allows the programmer to interpret the program state in terms of Alef operations. Consider the example of a multi-process program stalling because of improper synchronization. A stack trace of the program indicates that it is waiting for an event in some obscure Alef runtime synchronization function. The function itself is irrelevant to the programmer; of greater importance is the identity of the unfulfilled event. Commands in the Alef support library decode the runtime data structures and program state to report the cause of the blockage in terms of the high-level operations available to the Alef programmer. Here, the Acid language acts as a communications medium between Alef implementer and Alef user.

## 12. Parallel Debugging

The central issue in parallel debugging is how the debugger is multiplexed between the processes comprising the program. Acid has no intrinsic model of process partitioning; it only assumes that parallel programs share a symbol table, though they need not share memory. The `setproc` primitive attaches the debugger to a running process associated with the process ID supplied as its argument and assigns that value to the global variable `pid`, thereby allowing simple rotation among a group of processes. Further, the stack trace primitive is driven by parameters specifying a unique process context, so it is possible to examine the state of cooperating processes without switching the debugger focus from the process of interest. Since Acid is inherently extensible and capable of dynamic interaction with subordinate processes, the programmer can define Acid commands to detect and control complex interactions between processes. In short, the programmer is free to specify how the debugger reacts to events generated in specific threads of the program.

The support for parallel debugging in Acid depends on a crucial kernel modification: when the text segment of a program is written (usually to place a breakpoint), the segment is cloned to prevent other threads from encountering the breakpoint. Although this incurs a slight performance penalty, it is of little importance while debugging.

## 13. Communication Between Tools

The Plan 9 Alef and C compilers do not embed detailed type information in the symbol table of an executable file. However, they do accept a command line option causing them to emit descriptions of complex data types (e.g., aggregates and abstract data types) to an auxiliary file. The vehicle for expressing this information is Acid source code. When an Acid debugging session is subsequently started, that file is loaded with the other Acid libraries.

For each complex object in the program the compiler generates three pieces of Acid code. The first is a table describing the size and offset of each member of the complex data type. Following is an Acid function, named the same as the object, that formats and prints each member. Finally, Acid declarations associate the Alef or C program variables of a type with the functions to print them. The three forms of declaration are shown in the following example:

```
struct Bitmap {
        Rectangle    0 r;
        Rectangle   16 clipr;
        'D'   32 ldepth;
        'D'   36 id;
        'X'   40 cache;
};

defn
Bitmap(addr) {
        complex Bitmap addr;
        print("Rectangle r {\n");
        Rectangle(addr.r);
        print("}\n");
        print("Rectangle clipr {\n");
        Rectangle(addr.clipr);
        print("}\n");
        print(" ldepth  ", addr.ldepth, "\n");
        print(" id      ", addr.id, "\n");
        print(" cache   ", addr.cache, "\n");
};

complex Bitmap darkgrey;
complex Bitmap Window_settag:b;
```

The `struct` declaration specifies decoding instructions for the complex type named `Bitmap`. Although the syntax is superficially similar to a C structure declaration, the semantics differ markedly: the C declaration specifies a layout, while the Acid declaration tells how to decode it. The declaration specifies a type, an offset, and name for each member of the complex object. The type is either the name of another complex declaration, for example, `Rectangle`, or a format code. The offset is the number of bytes from the start of the object to the member and the name is the member's name in the Alef or C declaration. This type description is a close match for C and Alef, but is simple enough to be language independent.

The `Bitmap` function expects the address of a `Bitmap` as its only argument. It uses the decoding information contained in the `Bitmap` structure declaration to extract, format, and print the value of each member of the complex object pointed to by the argument. The Alef compiler emits code to call other Acid functions where a

member is another complex type; here, `Bitmap` calls `Rectangle` to print its contents.

The `complex` declarations associate Alef variables with complex types. In the example, `darkgrey` is the name of a global variable of type `Bitmap` in the program being debugged. Whenever the name `darkgrey` is evaluated by Acid, it automatically calls the `Bitmap` function with the address of `darkgrey` as the argument. The second `complex` declaration associates a local variable or parameter named `b` in function `Window_settag` with the `Bitmap` complex data type.

Acid borrows the C operators `.` and `->` to access the decoding parameters of a member of a complex type. Although this representation is sufficiently general for describing the decoding of both C and Alef complex data types, it may prove too restrictive for target languages with more complicated type systems. Further, the assumption that the compiler can select the proper Acid format code for each basic type in the language is somewhat naive. For example, when a member of a complex type is a pointer, it is assigned a hexadecimal type code; integer members are always assigned a decimal type code. This heuristic proves inaccurate when an integer field is a bit mask or set of bit flags which are more appropriately displayed in hexadecimal or octal.

## 14. Code Verification

Acid's ability to interact dynamically with an executing program allows passive test and verification of the target program. For example, a common concern is leak detection in programs using `malloc`. Of interest are two items: finding memory that was allocated but never freed and detecting bad pointers passed to `free`. An auxiliary Acid library contains Acid functions to monitor the execution of a program and detect these faults, either as they happen or in the automated post-mortem analysis of the memory arena. In the following example, the `sort` command is run under the control of the Acid memory leak library.

```
helix% acid -l malloc /bin/sort
/bin/sort: mips plan 9 executable
/lib/acid/port
/lib/acid/mips
/lib/acid/malloc
acid: go()
now
is
the
time
<ctrl-d>
is
now
the
time
27680 : breakpoint      _exits+0x4      MOVW    $0x8,R1
acid:
```

The `go` command creates a process and plants breakpoints at the entry to `malloc` and `free`. The program is then started and continues until it exits or stops. If the reason for stopping is anything other than the breakpoints in `malloc` and `free`, Acid prints the usual status information and returns to the interactive prompt.

When the process stops on entering `malloc`, the debugger must capture and save the address that `malloc` will return. After saving a stack trace so the calling routine can be identified, it places a breakpoint at the return address and restarts the program. When `malloc` returns, the breakpoint stops the program, allowing the debugger to grab the address of the new memory block from the return register. The address and stack trace are added to the list of outstanding memory blocks, the breakpoint is

removed from the return point, and the process is restarted.

When the process stops at the beginning of `free`, the memory address supplied as the argument is compared to the list of outstanding memory blocks. If it is not found an error message and a stack trace of the call is reported; otherwise, the address is deleted from the list.

When the program exits, the list of outstanding memory blocks contains the addresses of all blocks that were allocated but never freed. The `leak` library function traverses the list producing a report describing the allocated blocks.

```
acid: leak()
Lost a total of 524288 bytes from:
    malloc() malloc.c:32 called from dofile+0xe8 sort.c:217
    dofile() sort.c:190 called from main+0xac sort.c:161
    main() sort.c:128 called from _main+0x20 main9.s:10
Lost a total of 64 bytes from:
    malloc() malloc.c:32 called from newline+0xfc sort.c:280
    newline() sort.c:248 called from dofile+0x110 sort.c:222
    dofile() sort.c:190 called from main+0xac sort.c:161
    main() sort.c:128 called from _main+0x20 main9.s:10
Lost a total of 64 bytes from:
    malloc() malloc.c:32 called from realloc+0x14 malloc.c:129
    realloc() malloc.c:123 called from bldkey+0x358 sort.c:1388
    buildkey() sort.c:1345 called from newline+0x150 sort.c:285
    newline() sort.c:248 called from dofile+0x110 sort.c:222
    dofile() sort.c:190 called from main+0xac sort.c:161
    main() sort.c:128 called from _main+0x20 main9.s:10
acid: refs()
data...bss...stack...
acid: leak()
acid:
```

The presence of a block in the allocation list does not imply it is there because of a leak; for instance, it may have been in use when the program terminated. The `refs()` library function scans the *data*, *bss*, and *stack* segments of the process looking for pointers into the allocated blocks. When one is found, the block is deleted from the outstanding block list. The `leak` function is used again to report the blocks remaining allocated and unreferenced. This strategy proves effective in detecting disconnected (but non-circular) data structures.

The leak detection process is entirely passive. The program is not specially compiled and the source code is not required. As with the Acid support functions for the Alef runtime environment, the author of the library routines has encapsulated the functionality of the library interface in Acid code. Any programmer may then check a program's use of the library routines without knowledge of either implementation. The performance impact of running leak detection is great (about 10 times slower), but it has not prevented interactive programs like `sam` and the 8½ window system from being tested.

### 15. Code Coverage

Another common component of software test uses *coverage* analysis. The purpose of the test is to determine which paths through the code have not been executed while running the test suite. This is usually performed by a combination of compiler support and a reporting tool run on the output generated by statements compiled into the program. The compiler emits code that logs the progress of the program as it executes basic blocks and writes the results to a file. The file is then processed by the reporting tool to determine which basic blocks have not been executed.

Acid can perform the same function in a language independent manner without

modifying the source, object or binary of the program. The following example shows `ls` being run under the control of the Acid coverage library.

```
philw-helix% acid -l coverage /bin/ls
/bin/ls: mips plan 9 executable
/lib/acid/port
/lib/acid/mips
/lib/acid/coverage
acid: coverage()
acid
newstime
profile
tel
wintool
2: (error) msg: pid=11419 startstop: process exited
acid: analyse(ls)
ls.c:102,105
        102:     return 1;
        103: }
        104: if(db[0].qid.path&CHDIR && dflag==0){
        105:     output();
ls.c:122,126
        122:     memmove(dirbuf+ndir, db, sizeof(Dir));
        123:     dirbuf[ndir].prefix = 0;
        124:     p = utfrrune(s, '/');
        125:     if(p){
        126:         dirbuf[ndir].prefix = s;
```

The `coverage` function begins by looping through the text segment placing breakpoints at the entry to each basic block. The start of each basic block is found using the Acid builtin function `follow`. If the list generated by `follow` contains more than one element, then the addresses mark the start of basic blocks. A breakpoint is placed at each address to detect entry into the block. If the result of `follow` is a single address then no action is taken, and the next address is considered. Acid maintains a list of breakpoints already in place and avoids placing duplicates (an address may be the destination of several branches).

After placing the breakpoints the program is set running. Each time a breakpoint is encountered Acid deletes the address from the breakpoint list, removes the breakpoint from memory and then restarts the program. At any instant the breakpoint list contains the addresses of basic blocks which have not been executed. The `analyse` function reports the lines of source code bounded by basic blocks whose addresses are have not been deleted from the breakpoint list. These are the basic blocks which have not been executed. Program performance is almost unaffected since each breakpoint is executed only once and then removed.

The library contains a total of 128 lines of Acid code. An obvious extension of this algorithm could be used to provide basic block profiling.

## 16. Conclusion

Acid has two areas of weakness. As with other language-based tools like *awk*, a programmer must learn yet another language to step beyond the normal debugging functions and use the full power of the debugger. Second, the command line interface supplied by the *yacc* parser is inordinately clumsy. Part of the problem relates directly to the use of *yacc* and could be circumvented with a custom parser. However, structural problems would remain: Acid often requires too much typing to execute a simple command. A debugger should prostitute itself to its users, doing whatever is wanted with a minimum of encouragement; commands should be concise and obvious. The language interface is more consistent than an ad hoc command interface but is clumsy to use.

Most of these problems are addressed by an Acme interface which is under construction. This should provide the best of both worlds: graphical debugging and access to the underlying acid language when required.

The name space clash between Acid variables, keywords, program variables, and functions is unavoidable. Although it rarely affects a debugging session, it is annoying when it happens and is sometimes difficult to circumvent. The current renaming scheme is too crude; the new names are too hard to remember.

Acid has proved to be a powerful tool whose applications have exceeded expectations. Of its strengths, portability, extensibility and parallel debugging support were by design and provide the expected utility. In retrospect, its use as a tool for code test and verification and as a medium for communicating type information and encapsulating interfaces has provided unanticipated benefits and altered our view of the debugging process.

## 17. Acknowledgments

Bob Flandrena was the first user and helped prepare the paper. Rob Pike endured three buggy Alef compilers and a new debugger in a single sitting.

## 18. References

[Pike90] R. Pike, D. Presotto, K. Thompson, H. Trickey, ''Plan 9 from Bell Labs'', *UKUUG Proc. of the Summer 1990 Conf.,* London, England, 1990, reprinted, in a different form, in this volume.

[Gol93] M. Golan, D. Hanson, ''DUEL -- A Very High-Level Debugging Language'', *USENIX Proc. of the Winter 1993 Conf.,* San Diego, CA, 1993.

[Lin90] M. A. Linton, ''The Evolution of DBX'', *USENIX Proc. of the Summer 1990 Conf.,* Anaheim, CA, 1990.

[Stal91] R. M. Stallman, R. H. Pesch, ''Using GDB: A guide to the GNU source level debugger'', Technical Report, Free Software Foundation, Cambridge, MA, 1991.

[Win93] P. Winterbottom, ''Alef reference Manual'', this volume.

[Pike93] Rob Pike, ''Acme: A User Interface for Programmers'', *USENIX Proc. of the Winter 1994 Conf.,* San Francisco, CA, reprinted in this volume.

[Ols90] Ronald A. Olsson, Richard H. Crawford, and W. Wilson Ho, ''Dalek: A GNU, improved programmable debugger'', *USENIX Proc. of the Summer 1990 Conf.,* Anaheim, CA.

[May92] Paul Maybee, ''NeD: The Network Extensible Debugger'' *USENIX Proc. of the Summer 1992 Conf.,* San Antonio, TX.

[Aral] Ziya Aral, Ilya Gertner, and Greg Schaffer, ''Efficient debugging primitives for multiprocessors'', *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems,* SIGPLAN notices Nr. 22, May 1989.

# Acid Manual

*Phil Winterbottom*
*philw@plan9.bell-labs.com*

## Introduction

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

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

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

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

## Input format and `whatis`

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

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

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

The `whatis` operator can be used to report the state of identifiers known to the interpreter. With no argument, `whatis` reports the name of all defined Acid functions;

when supplied with an identifier as an argument it reports any variable, function, or type definition associated with the identifier. Because of the way the interpreter handles semicolons, the result of a `whatis` statement can be returned directly to Acid without adding semicolons. A syntax error or interrupt returns Acid to the normal evaluation mode; any partially evaluated definitions are lost.

### Using the Library Functions

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

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

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

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

### Types

An Acid variable has one of four types: *integer*, *float*, *list*, or *string*. The type of a variable is inferred from the type of the right-hand side of the assignment expression which last set its value. Referencing a variable that has not yet been assigned draws a "used but not set" error. Many of the operators may be applied to more than one type;

for these operators the action of the operator is determined by the types of its operands. The action of each operator is defined in the *Expressions* section of this manual.

### Variables

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

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

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

### Formats

In addition to a type, variables have formats. The format is a code letter that determines the printing style and the effect of some of the operators on that variable. The format codes are derived from the format letters used by *db*(1). By default, symbol table variables and numeric constants are assigned the format code `X`, which specifies 32-bit hexadecimal. Printing a variable with this code yields the output `0x00123456`. The format code of a variable may be changed from the default by using the builtin function `fmt`. This function takes two arguments, an expression and a format code. After the expression is evaluated the new format code is attached to the result and forms the return value from `fmt`. The backslash operator is a short form of `fmt`. The format supplied by the backslash operator must be the format character rather than an expression. If the result is assigned to a variable the new format code is maintained in the variable. For example:

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

The supported format characters are:

- `o` Print two-byte integer in octal.
- `O` Print four-byte integer in octal.
- `q` Print two-byte integer in signed octal.
- `Q` Print four-byte integer in signed octal.
- `B` Print four-byte integer in binary.
- `d` Print two-byte integer in signed decimal.
- `D` Print four-byte integer in signed decimal.
- `V` Print eight-byte integer in signed decimal.
- `Z` Print eight-byte integer in unsigned decimal.
- `x` Print two-byte integer in hexadecimal.
- `X` Print four-byte integer in hexadecimal.
- `Y` Print eight-byte integer in hexadecimal.

`u` Print two-byte integer in unsigned decimal.

`U` Print four-byte integer in unsigned decimal.

`f` Print single-precision floating point number.

`F` Print double-precision floating point number.

`g` Print a single precision floating point number in string format.

`G` Print a double precision floating point number in string format.

`b` Print byte in hexadecimal.

`c` Print byte as an ASCII character.

`C` Like `c`, with printable ASCII characters represented normally and others printed in the form `\x`*nn*.

`s` Interpret the addressed bytes as UTF characters and print successive characters until a zero byte is reached.

`r` Print a two-byte integer as a rune.

`R` Print successive two-byte integers as runes until a zero rune is reached.

`i` Print as machine instructions.

`I` As `i` above, but print the machine instructions in an alternate form if possible: `sunsparc` and `mipsco` reproduce the manufacturers' syntax.

`a` Print the value in symbolic form.

**Complex types**

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

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

is described by the Acid statement

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

**Scope**

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

### Addressing

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

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

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

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

### Name Conflicts

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

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

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

### Expressions

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

### Boolean expressions

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

*primary−expression:*
*identifier*
*identifier* : *identifier*
*constant*
( *expression* )
{ *elist* }

*elist:*
*expression*
*elist* , *expression*

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

```
*main:argc
```

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

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

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

**Lists**

Several operators manipulate lists.

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

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

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

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

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

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

**Operators**

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

*argument−list:*
*expression*
*argument−list , expression*

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

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

*unary−expression:*
*postfix−expression*
++ *unary−expression*
−− *unary−expression*

*unary−operator: one of*
* @ + − ~ !

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

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

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

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

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

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

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

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

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

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

*multiplicative−expression:*
*cast−expression*
*multiplicative−expression* * *multiplicative−expression*
*multiplicative−expression* / *multiplicative−expression*
*multiplicative−expression* % *multiplicative−expression*

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

*additive−expression:*
*multiplicative−expression*
*additive−expression* + *multiplicative−expression*
*additive−expression* − *multiplicative−expression*

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

concatenation. Adding a string and an integer is treated as concatenation with the Unicode character corresponding to the integer. Subtraction is undefined for strings and lists.

*shift−expression:*
*additive−expression*
*shift−expression* << *additive−expression*
*shift−expression* >> *additive−expression*

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

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

*equality−expression:*
*relational−expression*
*relational−expression* == *equality−expression*
*relational−expression* != *equality−expression*

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

*AND−expression:*
*equality−expression*
*AND−expression* & *equality−expression*

*XOR−expression:*
*AND−expression*
*XOR−expression* ^ *AND−expression*

*OR−expression:*
*XOR−expression*
*OR−expression* | *XOR−expression*

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

*logical−AND−expression:*
*OR−expression*
*logical−AND−expression* && *OR−expression*

*logical−OR−expression:*
*logical−AND−expression*
*logical−OR−expression* || *logical−AND−expression*

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

**Statements**

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

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

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

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

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

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

`return` *expression*

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

`local` *variable*

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

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

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

*parameter-list:*
*variable*
*parameter-list* `,` *variable*

*body:*
`{` *statement* `}`

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

**Code variables**

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

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

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

### Source Code Management

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

### Builtin Functions

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

*fmt* `fmtof(`*item*`)` Get format

`fmtof` evaluates the expression *item* and returns the format of the result.

```
acid: +fmtof(33)
W
acid: +fmtof("string")
s
```

*integer* `fmtsize(`*item*`)` Get format size

`fmtsize` evaluates the expression *item* and returns the size in bytes of a single element of result's format.

```
acid: +fmtsize('c')
8
acid: +fmtsize('c'\c)
1
acid: +fmtsize(0\X)
4
acid: +fmtsize('c'\3)
10
```

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

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

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

{} follow(*integer*) Compute follow set

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

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

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

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

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

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

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

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

*string* itoa(*integer[,string]*) Convert integer to string

itoa takes an integer argument and converts it into an ASCII string in the D format. an alternate format string may be provided in the % style of *print*(2). This function is commonly used to build rc command lines.

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

{} kill(*integer*) Kill a process

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**Library Functions**

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

the whatis operator and then modified during debugging sessions.

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

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

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

{} Fpr() Display double precision floating registers

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

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

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

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

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

{} acidinit() Interpreter startup

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

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

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

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

{} asm(*integer*) Disassemble machine instructions

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

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

{} bpdel(*integer*) Delete breakpoint

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

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

{} bpset(*integer*) Set a breakpoint

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

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

{} bptab() List active breakpoints

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

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

{} casm() Continue disassembly

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

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

{} cont() Continue program execution

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

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

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

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

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

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

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

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

{} fpr() Display single precision floating registers

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

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

{} func() Step while in function

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

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

{} gpr() Display general purpose registers

gpr prints the values of the general purpose processor registers.

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

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

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

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

{} lstk() Stack trace with local variables

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

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

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

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

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

{} new() Create new process

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

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

{} next() Step through language statement

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

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

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

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

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

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

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

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

{} procs() Print active process list

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

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

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

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

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

{} regs() Print registers

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

{} source() Summarize source data base

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

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

{} spr() Print special purpose registers

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

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

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

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

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

{} step() Single step process

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

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

{} stk() Stack trace

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

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

{} stmnt() Execute a single statement

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

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

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

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

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

{} symbols(*string*) Search symbol table

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

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

{} win() Start new process in a window

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

```
acid: win()
```

# Maintaining Files on Plan 9 with Mk

*Andrew G. Hume*
*andrew@research.att.com*
*Bob Flandrena*
*bobf@plan9.bell-labs.com*

*ABSTRACT*

Mk is a tool for describing and maintaining dependencies between files. It is similar to the UNIX program make, but provides several extensions. Mk's flexible rule specifications, implied dependency derivation, and parallel execution of maintenance actions are well-suited to the Plan 9 environment. Almost all Plan 9 maintenance procedures are automated using mk.

## 1. Introduction

This document describes how mk, a program functionally similar to make [Feld79], is used to maintain dependencies between files in Plan 9. Mk provides several extensions to the capabilities of its predecessor that work well in Plan 9's distributed, multi-architecture environment. It exploits the power of multiprocessors by executing maintenance actions in parallel and interacts with the Plan 9 command interpreter rc to provide a powerful set of maintenance tools. It accepts pattern-based dependency specifications that are not limited to describing rules for program construction. The result is a tool that is flexible enough to perform many maintenance tasks including database maintenance, hardware design, and document production.

This document begins by discussing the syntax of the control file, the pattern matching capabilities, and the special rules for maintaining archives. A brief description of mk's algorithm for deriving dependencies is followed by a discussion of the conventions used to resolve ambiguous specifications. The final sections describe parallel execution and special features.

An earlier paper [Hume87] provides a detailed discussion of mk's design and an appendix summarizes the differences between mk and make.

## 2. The Mkfile

Mk reads a file describing relationships among files and executes commands to bring the files up to date. The specification file, called a mkfile, contains three types of statements: assignments, includes, and rules. Assignment and include statements are similar to those in C. Rules specify dependencies between a *target* and its *prerequisites*. When the target and prerequisites are files, their modification times determine if they are out of date. Rules often contain a *recipe*, an *rc*(1) script that produces the target from the prerequisites.

This simple mkfile produces an executable from a C source file:

```
CC=pcc
f1:     f1.c
        $CC -o f1 f1.c
```

The first line assigns the name of the portable ANSI/POSIX compiler to the mk variable CC; subsequent references of the form $CC select this compiler. The only rule specifies a dependence between the target file f1 and the prerequisite file f1.c. If the target does not exist or if the prerequisite has been modified more recently than the target, mk passes the recipe to rc for execution. Here, f1.c is compiled and loaded to produce f1.

The native Plan 9 environment requires executables for all architectures, not only the current one. The Plan 9 version of the same mkfile looks like:

```
</$objtype/mkfile

f1:     f1.$O
        $LD $LDFLAGS -o f1 f1.$O
f1.$O:  f1.c
        $CC $CFLAGS f1.c
```

The first line is an include statement that replaces itself with the contents of the file /$objtype/mkfile. The variable $objtype is inherited from the environment and contains the name of the target architecture. The prototype mkfile for that architecture defines architecture-specific variables: CC and LD are the names of the compiler and loader, O is the code character of the architecture. The rules compile the source file into an object file and invoke the loader to produce f1. Invoking mk from the command line as follows

```
% objtype=mips mk
vc -w f1.c
vl $LDFLAGS -o f1 f1.k
%
```

produces the mips executable of program f1 regardless of the current architecture type.

We can extend the mkfile to build two programs:

```
</$objtype/mkfile
ALL=f1 f2

all:V:  $ALL

f1:     f1.$O
        $LD $LDFLAGS -o f1 f1.$O
f1.$O:  f1.c
        $CC $CFLAGS f1.c
f2:     f2.$O
        $LD $LDFLAGS -o f2 f2.$O
f2.$O:  f2.c
        $CC $CFLAGS f2.c
```

The target all, modified by the *attribute* V, builds both programs. The attribute identifies all as a dummy target that is not related to a file of the same name; its precise effect is explained later. This example describes cascading dependencies: the first target depends on another which depends on a third and so on. Here, individual rules build each program; later we'll see how to do this with a general rule.

## 3. Variables and the environment

Mk does not distinguish between its internal variables and rc variables in the environment. When mk starts, it imports each environment variable into a mk variable of the same name. Before executing a recipe, mk exports all variables, including those inherited from the environment, to the environment in which rc executes the recipe.

There are several ways for a variable to take a value. It can be set with an assignment statement, inherited from the environment, or specified on the command line. Mk also maintains several special internal variables that are described in *mk*(1). Assignments have the following decreasing order of precedence:

1) Command line assignment
2) Assignment statement
3) Imported from the environment
4) Implicitly set by mk

For example, a command line assignment overrides a value imported from the environment.

All variable values are strings. They can be used for pattern matching and comparison but not for arithmetic. A *list* is a string containing several values separated by white space. Each member is handled individually during pattern matching, target selection, and prerequisite evaluation.

A *namelist* is a list produced by transforming the members of an existing list. The transform applies a pattern to each member, replacing each matched string with a new string, much as in the substitute command in *sam*(1) or *ed*(1). The syntax is

```
${var:A%B=C%D}
```

where *var* is a variable. The pattern A%B matches a member beginning with the string *A* and ending with the string *B* with any string in between; it behaves like the regular expression A.*B. When a member of the *var* list matches this pattern, the string *C* replaces *A*, *D* replaces *B*, and the matched string replaces itself. Any of *A*, *B*, *C*, or *D* may be the empty string. In effect, a namelist is generated by applying the *ed*(1) substitute command

```
s/A(.*)B/C\1D/
```

to each member of a variable list.

Namelists are useful for generating a list based on a predictable transformation. For example,

```
SRC=a.c b.c c.c
OBJ=${SRC:%.c=%.v}
```

assigns the list (a.v b.v c.v) to OBJ. A namelist may be used anywhere a variable is allowed except in a recipe.

Command output is assigned to a variable using the normal rc syntax:

```
var=‘{rc command}
```

The command executes in an environment populated with previously assigned variables, including those inherited from mk’s execution environment. The command may be arbitrarily complex; for example,

```
TARG=‘{ls -d *.[cy] | sed 's/..$//'}
```

assigns a list of the C and yacc source files in the current directory, stripped of their suffix, to the variable TARG.

## 4. The include statement

The include statement replaces itself with the contents of a file. It is functionally similar to the C `#include` statement but uses a different syntax:

```
<filename
```

The contents of the file are evaluated as they are read. An include statement may be used anywhere except in a recipe.

Unlike `make`, `mk` has no built-in rules. Instead, the include statement allows generic rules to be imported from a prototype `mkfile`; most Plan 9 `mkfiles` use this approach [Flan95].

## 5. Rules

A rule has four elements: targets, prerequisites, attributes, and a recipe. It has the form:

```
targets:attributes:prerequisites
        recipe
```

The first line, containing the targets, attributes, and prerequisites is the *rule header*; it must begin at the left margin. The recipe contains zero or more lines, each of which begins with white space. One or more targets must be specified but the attributes, prerequisites, and recipe are optional. A rule specifies a dependency between the target(s) and its prerequisite(s), the recipe brings the target(s) up to date with the prerequisite(s) and attributes modify `mk`'s evaluation of the dependency.

Normally the target is a file that depends on one or more prerequisite files. `Mk` compares the modification times of each target and each prerequisite; a target is considered out of date when it does not exist or when a prerequisite has been modified more recently. When a target is out of date, `mk` executes the recipe to bring it up to date. When the recipe completes, the modification time of the target is checked and used in later dependency evaluations. If the recipe does not update the target, evaluation continues with the out of date target.

A prerequisite of one rule may be the target of another. When this happens, the rules cascade to define a multi-step procedure. For example, an executable target depends on prerequisite object files, each of which is a target in a rule with a C source file as the prerequisite. `Mk` follows a chain of dependencies until it encounters a prerequisite that is not a target of another rule or it finds a target that is up to date. It then executes the recipes in reverse order to produce the desired target.

The rule header is evaluated when the rule is read. Variables are replaced by their values, namelists are generated, and commands are replaced by their output at this time.

Most attributes modify `mk`'s evaluation of a rule. An attribute is usually a single letter but some are more complicated. This paper only discusses commonly used attributes; see *mk*(1) for a complete list.

The `V` attribute identifies a *virtual* target; that is, a target that is not a file. For example,

```
clean:V:
        rm *.$O $O.out
```

removes executables and compiler intermediate files. The target is virtual because it does not refer to a file named `clean`. Without the attribute, the recipe would not be executed if a file named `clean` existed. The `Q` attribute silences the printing of a recipe before execution. It is useful when the output of a recipe is similar to the recipe:

```
default:QV:
        echo 'No default target; use mk all or mk install'
```

The recipe is an rc script. It is optional but when it is missing, the rule is handled specially, as described later. Unlike make, mk executes recipes without interpretation. After stripping the first white space character from each line it passes the entire recipe to rc on standard input. Since mk does not interpret a recipe, escape conventions are exactly those of rc. Scripts for awk and sed commands can be embedded exactly as they would be entered from the command line. Mk invokes rc with the −e flag, which causes rc to stop if any command in the recipe exits with a non-zero status; the E attribute overrides this behavior and allows rc to continue executing in the face of errors. Before a recipe is executed, variables are exported to the environment where they are available to rc. Commands in the recipe may not read from standard input because mk uses it internally.

References to a variable can yield different values depending on the location of the reference in the mkfile. Mk resolves variable references in assignment statements and rule headers when the statement is read. Variable references in recipes are evaluated by rc when the recipe is executed; this happens after the entire mkfile has been read. The value of a variable in a recipe is the last value assigned in the file. For example,

```
STRING=all

all:VQ:
        echo $STRING
STRING=none
```

produces the message none. A variable assignment in a recipe does not affect the value of the variable in the mkfile for two reasons. First, mk does not import values from the environment when a recipe completes; one recipe cannot pass a value through the environment to another recipe. Second, no recipe is executed until mk has completed its evaluation, so even if a variable were changed, it would not affect the dependency evaluation.

## 6. Metarules

A *metarule* is a rule based on a pattern. The pattern selects a class of target(s) and identifies related prerequisites. Mk metarules may select targets and prerequisites based on any criterion that can be described by a pattern, not just the suffix transformations associated with program construction.

Metarule patterns are either *intrinsic* or regular expressions conforming to the syntax of *regexp*(6). The intrinsic patterns are shorthand for common regular expressions. The intrinsic pattern % matches one or more of anything; it is equivalent to the regular expression '.+'. The other intrinsic pattern, &, matches one or more of any characters except '/' and '.'. It matches a portion of a path and is equivalent to the regular expression '[^./]+'. An intrinsic pattern in a prerequisite references the string matched by the same intrinsic pattern in the target. For example, the rule

```
        %.v:    %.c
```

says that a file ending in .v depends on a file of the same name with a .c suffix: foo.v depends on foo.c, bar.v depends on bar.c, and so on. The string matched by an intrinsic pattern in the target is supplied to the recipe in the variable $stem. Thus the rule

```
%.$O:   %.c
        $CC $CFLAGS $stem.c
```

creates an object file for the target architecture from a similarly named C source file. If

several object files are out of date, the rule is applied repeatedly and `$stem` refers to each file in turn. Since there is only one `stem` variable, there can only be one % or & pattern in a target; the pattern %–%.c is illegal.

Metarules simplify the `mkfile` for building programs `f1` and `f2`:

```
</$objtype/mkfile

ALL=f1 f2

all:V:  $ALL

%:      %.$O
        $LD -o $target $prereq
%.$O:   %.c
        $CC $CFLAGS $stem.c
clean:V:
        rm -f $ALL *.[$OS]
```

(The variable `$OS` is a list of code characters for all architectures.) Here, metarules specify compile and load steps for all C source files. The loader rule relies on two internal variables set by `mk` during evaluation of the rule: `$target` is the name of the target(s) and `$prereq` the name of all prerequisite(s). Metarules allow this `mkfile` to be easily extended; a new program is supported by adding its name to the third line.

A regular expression metarule must have an `R` attribute. Prerequisites may reference matching substrings in the target using the form `\`*n* where *n* is a digit from 1 to 9 specifying the *n*th parenthesized sub-expression. In a recipe, `$stem`*n* is the equivalent reference. For example, a compile rule could be specified using regular expressions:

```
(.+)\.$O:R:     \1.c
        $CC $CFLAGS $stem1.c
```

Here, `\1` and `$stem1` refer to the name of the target object file without the suffix. The variable `$stem` associated with an intrinsic pattern is undefined in a regular expression metarule.

## 7. Archives

`Mk` provides a special mechanism for maintaining an archive. An archive member is referenced using the form *lib*(*file*) where *lib* is the name of the archive and *file* is the name of the member. Two rules define the dependency between an object file and its membership in an archive:

```
$LIB(foo.8):N:  foo.8
$LIB:   $LIB(foo.8)
        ar rv $LIB foo.8
```

The first rule establishes a dependency between the archive member and the object file. Normally, `mk` detects an error when a target does not exist and the rule contains no recipe; the `N` attribute overrides this behavior because the subsequent rule updates the member. The second rule establishes the dependency between the member and the archive; its recipe inserts the member into the archive. This two-step specification allows the modification time of the archive to represent the state of its members. Other rules can then specify the archive as a prerequisite instead of listing each member.

A metarule generalizes library maintenance:

```
LIB=lib.a
OBJS=etoa.$O atoe.$O ebcdic.$O

$LIB(%):N:      %
$LIB:   ${OBJS:%=$LIB(%)}
        ar rv $LIB $OBJS
```

The namelist prerequisite of the `$LIB` target generates archive member names for each object file name; for example, `etoa.$O` becomes `lib.a(etoa.$O)`. This formulation always updates all members. This is acceptable for a small archive, but may be slow for a big one. The rule

```
$LIB:   ${OBJS:%=$LIB(%)}
        ar rv $LIB `{membername $newprereq}
```

only updates out of date object files. The internal variable `$newprereq` contains the names of the out of date prerequisites. The `rc` script `membername` transforms an archive member specification into a file name: it translates `lib.a(etoa.$O)` into `etoa.$O`.

The `mkfile`

```
</$objtype/mkfile
LIB=lib.a
OBJS=etoa.$O atoe.$O ebcdic.$O

prog:   main.$O $LIB
        $LD -o $target $prereq

$LIB(%):N:      %
$LIB:   ${OBJS:%=$LIB(%)}
        ar rv $LIB $OBJS
```

builds a program by loading it with a library.

## 8. Evaluation algorithm

For each target of interest, `mk` uses the rules in a `mkfile` to build a data structure called a dependency graph. The nodes of the graph represent targets and prerequisites; a directed arc from one node to another indicates that the file associated with the first node depends on the file associated with the second. When the `mkfile` has been completely read, the graph is analyzed. In the first step, implied dependencies are resolved by computing the *transitive closure* of the graph. This calculation extends the graph to include all targets that are potentially derivable from the rules in the `mkfile`. Next the graph is checked for cycles; `make` accepts cyclic dependencies, but `mk` does not allow them. Subsequent steps prune subgraphs that are irrelevant for producing the desired target and verify that there is only one way to build it. The recipes associated with the nodes on the longest path between the target and an out of date prerequisite are then executed in reverse order.

The transitive closure calculation is sensitive to metarules; the patterns often select many potential targets and cause the graph to grow rapidly. Fortunately, dependencies associated with the desired target usually form a small part of the graph, so, after pruning, analysis is tractable. For example, the rules

```
%:      x.%
        recipe1
x.%:    %.k
        recipe2
%.k:    %.f
        recipe3
```

produce a graph with four nodes for each file in the current directory. If the desired target is `foo`, `mk` detects the dependency between it and the original file `foo.f` through intermediate dependencies on `foo.k` and `x.foo`. Nodes associated with other files are deleted during pruning because they are irrelevant to the production of `foo`.

Mk avoids infinite cycles by evaluating each metarule once. Thus, the rule

```
%:      %.z
        cp $prereq $prereq.z
```

copies the prerequisite file once.

### 9. Conventions for evaluating rules

There must be only one way to build each target. However, during evaluation metarule patterns often select potential targets that conflict with the targets of other rules. Mk uses several conventions to resolve ambiguities and to select the proper dependencies.

When a target selects more than one rule, `mk` chooses a regular rule over a metarule. For example, the `mkfile`

```
</$objtype/mkfile

FILES=f1.$O f2.$O f3.$O

prog:   $FILES
        $LD -o $target $prereq

%.$O:   %.c
        $CC $CFLAGS $stem.c

f2.$O:  f2.c
        $CC f2.c
```

contains two rules that could build `f2.$O`. Mk selects the last rule because its target, `f2.$O`, is explicitly specified, while the `%.$O` rule is a metarule. In effect, the explicit rule for `f2.$O` overrides the general rule for building object files from C source files.

When a rule has a target and prerequisites but no recipe, those prerequisites are added to all other rules with recipes that have the same target. All prerequisites, regardless of where they were specified, are exported to the recipe in variable `$prereq`. For example, in

```
</$objtype/mkfile

FILES=f1.$O f2.$O f3.$O

prog:   $FILES
        $LD -o $target $prereq

%.$O:   hdr.h

%.$O:   %.c
        $CC $CFLAGS $stem.c
```

the second rule adds `hdr.h` as a prerequisite of the compile metarule; an object file produced from a C source file depends on `hdr.h` as well as the source file. Notice that the recipe of the compile rule uses `$stem.c` instead of `$prereq` because the latter specification would attempt to compile `hdr.h`.

When a target is virtual and there is no other rule with the same target, `mk` evaluates each prerequisite. For example, adding the rule

```
all:V:  prog
```

to the preceding example builds the executable when either `prog` or `all` is the specified target. In effect, the `all` target is an alias for `prog`.

When two rules have identical rule headers and both have recipes, the later rule replaces the former one. For example, if a file named `mkrules` contains

```
$O.out: $OFILES
        $LD $LFLAGS $OFILES
%.$O:   %.c
        $CC $CFLAGS $stem.c
```

the `mkfile`

```
OFILES=f1.$O f2.$O f3.$O

<mkrules

$O.out: $OFILES
        $LD $LFLAGS -l $OFILES -lbio -lc
```

overrides the general loader rule with a special rule using a non-standard library search sequence. A rule is neutralized by overriding it with a rule with a null recipe:

```
<mkrules

$O.out:Q:       $OFILES
        ;
```

The Q attribute suppresses the printing of the semicolon.

When a rule has no prerequisites, the recipe is executed only when the target does not exist. For example,

```
marker:
        touch $target
```

defines a rule to manage a marker file. If the file exists, it is considered up to date regardless of its modification time. When a virtual target has no prerequisites the recipe is always executed. The `clean` rule is of this type:

```
clean:V:
        rm -f [$OS].out *.[$OS]
```

When a rule without prerequisites has multiple targets, the extra targets are aliases for the rule. For example, in

```
clean tidy nuke:V:
        rm -f [$OS].out *.[$OS]
```

the rule can be invoked by any of three names. The first rule in a `mkfile` is handled specially: when `mk` is invoked without a command line target all targets of the first non-metarule are built. If that rule has multiple targets, the recipe is executed once for each target; normally, the recipe of a rule with multiple targets is only executed once.

A rule applies to a target only when its prerequisites exist or can be derived. More than one rule may have the same target as long as only one rule with a recipe remains applicable after the dependency evaluation completes. For example, consider a program built from C and assembler source files. Two rules produce object files:

```
%.$O:   %.c
        $CC $CFLAGS $stem.c
%.$O:   %.s
        $AS $AFLAGS $stem.s
```

As long as there are not two source files with names like *foo*.c and *foo*.s, mk can unambiguously select the proper rule. If both files exist, the rules are ambiguous and mk exits with an error message.

In Plan 9, many programs consist of portable code stored in one directory and architecture-specific source stored in another. For example, the mkfile

```
</$objtype/mkfile

FILES=f1.$O f2.$O f3.$O f3.$O

prog:   $FILES
        $LD -o $target $prereq

%.$O:   %.$c
        $CC $CFLAGS $stem.c

%.$O:   ../port/%.c
        $CC $CFLAGS ../port/$stem.c
```

builds the program named prog using portable code in directory ../port and architecture-specific code in the current directory. As long as the names of the C source files in ../port do not conflict with the names of files in the current directory, mk selects the appropriate rule to build the object file. If like-named files exist in both directories, the specification is ambiguous and an explicit target must be specified to resolve the ambiguity. For example, adding the rule

```
f2.$O:  f2.c
        $CC $CFLAGS $f2.c
```

to the previous mkfile uses the architecture-specific version of f2.c instead of the portable one. Here, the explicit rule unambiguously documents which of the like-named source files is used to build the program.

Mk's heuristics can produce unintended results when rules are not carefully specified. For example, the rules that build object files from C or assembler source files

```
%.$O:   %.c
        $CC $CFLAGS $stem.c
%.$O:   %.s
        $AS $AFLAGS $stem.s
```

illustrate a subtle pratfall. Adding a header file dependency to the compile rule

```
%.$O:   %.c hdr.h
        $CC $CFLAGS $stem.c
```

produces the error message

```
don't know how to make 'file.c'
```

when *file*.s is an assembler source file. This occurs because *file*.s satisfies the assemble rule and hdr.h satisfies the compile rule, so either rule can potentially produce the target. When a prerequisite exists or can be derived, all other prerequisites in that rule header must exist or be derivable; here, the existence of hdr.h forces the evaluation of a C source file. Specifying the dependencies in different rules avoids this interpretation:

```
%.$O:   hdr.h
%.$O:   %.c
        $CC $CFLAGS $stem.c
```

Although hdr.h is an additional prerequisite of the compile rule, the two rules are evaluated independently and the existence of the C source file is not linked to the existence

of the header file. However, this specification describes a different dependency. Originally, only object files derived from C files depended on `hdr.h`; now all object files, including those built from assembler source, depend on the header file.

Metarule patterns should be as restrictive as possible to prevent conflicts with other rules. Consider the `mkfile`

```
</$objtype/mkfile
BIN=/$objtype/bin
PROG=foo

install:V:      $BIN/$PROG

%:      %.c
        $CC $stem.c
        $LD -o $target $stem.$O

$BIN/%: %
        mv $stem $target
```

The first target builds an executable in the local directory; the second installs it in the directory of executables for the architecture. Invoking `mk` with the `install` target produces:

```
mk: ambiguous recipes for /mips/bin/foo:
/mips/bin/foo <-(mkfile:8)- /mips/bin/foo.c <-(mkfile:12)- foo.c
/mips/bin/foo <-(mkfile:12)- foo <-(mkfile:8)- foo.c
```

The prerequisite of the `install` rule, `$BIN/$PROG`, matches both metarules because the % pattern matches everything. The & pattern restricts the compile rule to files in the current directory and avoids the conflict:

```
&:      &.c
        $CC $stem.c
        $LD -o $target $stem.$O
```

### 10. Missing intermediates

`Mk` does not build a missing intermediate file if a target is up to date with the prerequisites of the intermediate. For example, when an executable is up to date with its source file, `mk` does not compile the source to create a missing object file. The evaluation only applies when a target is considered up to date by pretending that the intermediate exists. Thus, it does not apply when the intermediate is a command line target or when it has no prerequisites.

This capability is useful for maintaining archives. We can modify the archive update recipe to remove object files after they are archived:

```
$LIB(%):N:      %
$LIB:   ${OBJS:%=$LIB(%)}
        names='{membername $newprereq}
        ar rv $LIB $names
        rm -f $names
```

A subsequent `mk` does not remake the object files as long as the members of the archive remain up to date with the source files. The `-i` command line option overrides this behavior and causes all intermediates to be built.

## 11. Alternative out-of-date determination

Sometimes the modification time is not useful for deciding when a target and prerequisite are out of date. The P attribute replaces the default mechanism with the result of a command. The command immediately follows the attribute and is repeatedly executed with each target and each prerequisite as its arguments; if its exit status is non-zero, they are considered out of date and the recipe is executed. Consider the mkfile

```
foo.ref:Pcmp -s:        foo
        cp $prereq $target
```

The command

```
cmp -s foo.ref foo
```

is executed and if foo.ref differs from foo, the latter file is copied to the former.

## 12. Parallel processing

When possible, mk executes recipes in parallel. The variable $NPROC specifies the maximum number of simultaneously executing recipes. Normally it is imported from the environment, where the system has set it to the number of available processors. It can be decreased by assigning a new value and can be set to 1 to force single-threaded recipe execution. This is necessary when several targets access a common resource such as a status file or data base. When there is no dependency between targets, mk assumes the recipes can be executed concurrently. Normally, this allows multiple prerequisites to be built simultaneously; for example, the object file prerequisites of a load rule can be produced by compiling the source files in parallel. Mk does not define the order of execution of independent recipes. When the prerequisites of a rule are not independent, the dependencies between them should be specified in a rule or the mkfile should be single-threaded. For example, the archive update rules

```
$LIB(%):N:      %
$LIB:   ${OBJS:%=$LIB(%)}
        ar rv $LIB '{membername $newprereq}
```

compile source files in parallel but update all members of the archive at once. It is a mistake to merge the two rules

```
$LIB(%):        %
        ar rv $LIB $stem
```

because an ar command is executed for every member of the library. Not only is this inefficient, but the archive is updated in parallel, making interference likely.

The $nproc environment variable contains a number associated with the processor executing a recipe. It can be used to create unique names when the recipe may be executing simultaneously on several processors. Other maintenance tools provide mechanisms to control recipe scheduling explicitly [Cmel86], but mk's general rules are sufficient for all but the most unusual cases.

## 13. Deleting target files on errors

The D attribute causes mk to remove the target file when a recipe terminates prematurely. The error message describing the termination condition warns of the deletion. A partially built file is doubly dangerous: it is not only wrong, but is also considered to be up to date so a subsequent mk will not rebuild it. For example,

```
pic.out:D:      mk.ms
                pic $prereq | tbl | troff -ms > $target
```

produces the message

```
mk: pic mk.ms | ...  : exit status=rc 685: deleting 'pic.out'
```

if any program in the recipe exits with an error status.

## 14. Unspecified dependencies

The −w command line flag forces the files following the flag to be treated as if they were just modified. We can use this flag with a command that selects files to force a build based on the selection criterion. For example, if the declaration of a global variable named *var* is changed in a header file, all source files that reference it can be rebuilt with the command

```
$ mk −w'{grep −l var *.[cyl]}
```

## 15. Conclusion

There are many programs related to make, each choosing a different balance between specialization and generality. Mk emphasizes generality but allows customization through its pattern specifications and include facilities.

Plan 9 presents a difficult maintenance environment with its heterogeneous architectures and languages. Mk's flexible specification language and simple interaction with rc work well in this environment. As a result, Plan 9 relies on mk to automate almost all maintenance. Tasks as diverse as updating the network data base, producing the manual, or building a release are expressed as mk procedures.

## 16. References

[Cmel86] R. F. Cmelik, ''Concurrent Make: A Distributed Program in Concurrent C'', AT&T Bell Laboratories Technical Report, 1986.

[Feld79] S. I. Feldman, ''Make — a program for maintaining computer programs'', *Software Practice & Experience* , 1979 Vol 9 #4, pp. 255–266.

[Flan95] Bob Flandrena, ''Plan 9 Mkfiles'', this volume.

[Hume87] A. G. Hume, ''Mk: A Successor to Make'', *USENIX Summer Conf. Proc.,* Phoenix, Az.

## 17. Appendix: Differences between make and mk

The differences between mk and make are:

- Make builds targets when it needs them, allowing systematic use of side effects. Mk constructs the entire dependency graph before building any target.
- Make supports suffix rules and % metarules. Mk supports % and regular expression metarules. (Older versions of make support only suffix rules.)
- Mk performs transitive closure on metarules, make does not.
- Make supports cyclic dependencies, mk does not.
- Make evaluates recipes one line at a time, replacing variables by their values and executing some commands internally. Mk passes the entire recipe to the shell without interpretation or internal execution.
- Make supports parallel execution of single-line recipes when building the prerequisites for specified targets. Mk supports parallel execution of all recipes. (Older versions of make did not support parallel execution.)
- Make uses special targets (beginning with a period) to indicate special processing. Mk uses attributes to modify rule evaluation.
- Mk supports virtual targets that are independent of the file system.
- Mk allows non-standard out-of-date determination, make does not.

It is usually easy to convert a `makefile` to or from an equivalent `mkfile`.

# Plan 9 Mkfiles

*Bob Flandrena*
*bobf@plan9.bell-labs.com*

## Introduction

Every Plan 9 source directory contains a file, called `mkfile`, specifying the rules for building the executable or library that is the product of the directory. *Mk*(1) interprets the rules in the file, calculates the dependencies, and executes an *rc*(1) script to construct the product. If necessary components are supplied by neighboring directories or sub-directories, the mkfiles in those directories are first executed to build the components before the local construction proceeds.

Most application source directories produce one of four types of product: a single executable, several executables, a local library, or a system library. Four generic mkfiles define the normal rules for building each type of product. The simplest mkfiles need only list the components and include the appropriate generic mkfile to do the work. More complex mkfiles may supply additional rules to augment, modify, or override the generic rules.

## Using a Mkfile

To build a product, change to the directory containing its source and invoke *mk* with the appropriate target as an argument. All mkfiles provide the following standard targets:

| | |
|---|---|
| `all` | Build a local version of the product or products for the current architecture. If the product is a single program, the result is stored in file `$O.out`. If the directory produces multiple executables, they are stored in the files named `$O.`*progname,* where *progname* is the name of each executable. A product may be built for a different architecture by prefacing the `mk` command with `objtype=`*architecture*, where *architecture* is the name of the target architecture. Directories producing system libraries always operate directly on the installed version of the library; in this case the target `all` is equivalent to the target `install`. |
| `install` | Build and install the product or products for the current architecture. |
| `installall` | Build and install the product or products for all architectures. |
| `clean` | Rid the directory and its subdirectories of the by-products of the build process. Intermediate files that are easily reproduced (e.g., object files, `yacc` intermediates, target executables) are always removed. Complicated intermediates, such as local libraries, are usually preserved. |
| `nuke` | Remove all intermediates from the directory and any subdirectories. This target guarantees that a subsequent build for the architecture is performed from scratch. |

If no target is specified on the `mk` command line, the `all` target is built by default. In a directory producing multiple executables, there is no default target.

In addition to the five standard targets, additional targets may be supplied by each generic mkfile or by the directory's mkfile.

The environment variable NPROC is set by the system to the number of available processors. Setting this variable, either in the environment or in a mkfile, controls the amount of parallelism in the build. For example, the command

```
NPROC=1 mk
```

restricts a build to a single thread of execution.

### Creating a Mkfile

The easiest way to build a new mkfile is to copy and modify an existing mkfile of the same type. Failing that, it is usually possible to create a new mkfile with minimal effort, since the appropriate generic mkfile predefines the rules that do all the work. In the simplest and most common cases, the new mkfile need only define a couple of variables and include the appropriate architecture-specific and generic mkfiles.

There are four generic mkfiles containing commonly used rules for building a product: `mkone`, `mkmany`, `mklib`, and `mksyslib`. These rules perform such actions as compiling C source files, loading object files, archiving libraries, and installing executables in the `bin` directory of the appropriate architecture. The generic mkfiles are stored in directory `/sys/src/cmd`. Mkfile `mkone` builds a single executable, `mkmany` builds several executables from the source in a single directory, and `mklib` and `mksyslib`, maintain local and system libraries, respectively. The rules in the generic mkfiles are driven by the values of variables, some of which must be set by the product mkfile and some of which are supplied by the generic mkfile. Variables in the latter class include:

| *Variable* | *Default* | *Meaning* |
|---|---|---|
| CFLAGS | −FVw | C compiler flags |
| LDFLAGS | | Loader flags |
| YFLAGS | −d | Yacc flags |
| AFLAGS | | Assembler flags |

The following variables are set by the product mkfile and used by the generic mkfile. Any may be empty depending on the specific product being made.

| | |
|---|---|
| TARG | Name(s) of the executable(s) to be built |
| LIB | Library name(s) |
| OFILES | Object files |
| HFILES | Header files included by all source files |
| YFILES | `Yacc` input files |
| BIN | Directory where executables are installed |

### Mkfile Organization

All mkfiles share the following common structure:

```
</$objtype/mkfile          # architecture-dependent definitions

variable definitions       # TARG, OFILES, HFILES, etc.

</sys/src/cmd/generic      # mkone, mkmany, mklib, or mksyslib

variable overrides         # CFLAGS, objtype, etc.

extra rules                # overrides, augmented rules, additional targets
```

Note that the architecture-dependent mkfiles include file /sys/src/mkfile.proto for system-wide variables that are common to all architectures.

The variables driving the expansion of the generic mkfile may be specified in any order as long as they are defined before the inclusion of the generic mkfile. The value of a variable may be changed by assigning a new value following the inclusion of the generic mkfile, but the effects are sometimes counter-intuitive. Such variable assignments do not apply to the target and prerequisite portions of any previously defined rules; the new values only apply to the recipes of rules preceding the assignment statement and to all parts of any rules following it.

The rules supplied by the generic mkfile may be overridden or augmented. The new rules must be specified after the inclusion of the generic mkfile. If the target and prerequisite portion of the rule exactly match the target and prerequisite portion of a previously defined rule and the new rule contains a recipe, the new rule replaces the old one. If the target of a new rule exactly matches the target of a previous rule and one or more new prerequisites are specified and the new rule contains no recipe, the new prerequisites are added to the prerequisites of the old rule.

Following sections discuss each generic mkfile in detail.

**Mkone**

The mkone generic mkfile contains rules for building a single executable from one or more files in a directory. The variable TARG specifies the name of the executable and variables OFILES and YFILES specify the object files and yacc source files used to build it. HFILES contains the names of the local header files included in all source files. BIN is the name of the directory where the executable is installed. LIB contains the names of local libraries used by the linker. This variable is rarely needed as libraries referenced by a #pragma directive in an associated header file, including all system libraries, are automatically searched by the loader.

If mk is executed without a target, the all target is built; it produces an executable in $O.out. Variable HFILES identifies the header files that are included in all or most or the C source files. Occasionally, a program has other header files that are only used in some source files. A header can be added to the prerequisites for those object files by adding a rule of the following form following the inclusion of generic mkfile mkone:

```
file.$O:        header.h
```

The mkfile for a directory producing a single executable using the normal set of rules is trivial: a list of some files followed by the inclusion of *mkone.* For example, /sys/src/cmd/diff/mkfile contains:

```
< /$objtype/mkfile

TARG=diff
OFILES=\
        diffdir.$O\
        diffio.$O\
        diffreg.$O\
        main.$O\

HFILES=diff.h

BIN=/$objtype/bin
</sys/src/cmd/mkone
```

The more complex mkfile in /sys/src/cmd/awk overrides compiler and loader variables to select the ANSI/POSIX Computing Environment with appropriately defined command line variables. It also overrides the default yacc rule to place the output soure in

file awkgram.c and the clean and nuke rules, so it can remove the non-standard intermediate files. Finally, the last three rules build a version of maketab appropriate for the architecture where the mk is being run and then executes it to create source file proctab.c:

```
</$objtype/mkfile

TARG=awk
OFILES=re.$O\
        lex.$O\
        main.$O\
        parse.$O\
        proctab.$O\
        tran.$O\
        lib.$O\
        run.$O\
        awkgram.$O\

HFILES=awk.h\
        y.tab.h\
        proto.h\

YFILES=awkgram.y

BIN=/$objtype/bin
</sys/src/cmd/mkone
CFLAGS=-c -D_REGEXP_EXTENSION -D_RESEARCH_SOURCE \
        -D_BSD_EXTENSION -DUTF
YFLAGS=-S -d -v
CC=pcc
LD=pcc
cpuobjtype=‘{sed -n 's/^O=//p' /$cputype/mkfile}

y.tab.h awkgram.c:      $YFILES
        $YACC -o awkgram.c $YFLAGS $prereq

clean:V:
        rm -f *.[$OS] [$OS].out [$OS].maketab y.tab.? y.debug\
                y.output $TARG

nuke:V:
        rm -f *.[$OS] [$OS].out [$OS].maketab y.tab.? y.debug\
                y.output awkgram.c $TARG

proctab.c:      $cpuobjtype.maketab
        ./$cpuobjtype.maketab >proctab.c

$cpuobjtype.maketab:    y.tab.h maketab.c
        objtype=$cputype
        mk maketab.$cputype

maketab.$cputype:V:     y.tab.h maketab.$O
        $LD -o $O.maketab maketab.$O
```

**Mkmany**

The mkmany generic mkfile builds several executables from the files in a directory. It differs from the operation of mkone in three respects: TARG specifies the names of all executables, there is no default command-line target, and additional rules allow a single

executable to be built or installed.

The TARG variable specifies the names of all executables produced by the mkfile. The rules assume the name of each executable is also the name of the file containing its main function. OFILES specifies files containing common subroutines loaded with all executables. Consider the mkfile:

```
</$objtype/mkfile

TARG=alpha beta
OFILES=common.$O
BIN=/$objtype/bin
</sys/src/cmd/mkmany
```

It assumes the main functions for executables alpha and beta are in files alpha.$O and beta.$O and that both programs use the subroutines in file common.$O. The all target builds all executables, leaving each in a file with a name of the form $O.*progname* where *progname* is the name of the executable. In this example the all target produces executables $O.alpha and $O.beta.

The mkmany rules provide additional targets for building a single executable:

| | |
|---|---|
| $O.progname | Builds executable $O.*progname* in the current directory. When the target architecture is not the current architecture the mk command must be prefixed with the customary objtype=*architecture* assignment to select the proper compilers and loaders. |
| progname.install | Installs executable *progname* for the target architecture. |
| progname.installall | Installs executable *progname* for all architectures. |

**Mklib**

The mklib generic mkfile builds a local library. Since this form of mkfile constructs no executable, the TARG and BIN variables are not needed. Instead, the LIB variable specifies the library to be built or updated. Variable OFILES contains the names of the object files to be archived in the library. The use of variables YFILES and HFILES does not change. When possible, only the out-of-date members of the library are updated.

The variable LIBDIR contains the name of the directory where the library is installed; by default it selects the current directory. It can be overridden by assigning the new directory name after the point where mklib is included.

The clean target removes object files and yacc intermediate files but does not touch the library. The nuke target removes the library as well as the files removed by the clean target. The command

```
mk -s clean all
```

causes the existing library to be updated, or created if it doesn't already exist. The command

```
mk -s nuke all
```

forces the library to be rebuilt from scratch.

The mkfile from /sys/src/cmd/upas/libString contains the following specifications to build the local library libString.a$O for the object architecture referenced by $O:

```
</$objtype/mkfile

LIB=libString.a$O
OFILES= s_alloc.$O\
        s_append.$O\
        s_array.$O\
        s_copy.$O\
        s_getline.$O\
        s_grow.$O\
        s_nappend.$O\
        s_parse.$O\
        s_read.$O\
        s_read_line.$O\
        s_tolower.$O\

</sys/src/cmd/mklib

nuke:V:
        mk clean
        rm -f libString.a[$OS]
```

The override of the rule for target `nuke` removes the libraries for all architectures as opposed to the default recipe for this target which removes the library for the current architecture.

### Mksyslib

The `mksyslib` generic mkfile is similar to the `mklib` mkfile except that it operates on a system library instead of a local library. The `install` and `all` targets are the same; since there is no local copy of the library, all updates are performed on the installed library. The rule for the `nuke` target is identical to that of the `clean` target; unlike the `nuke` target for local libraries, the library is never removed.

No attempt is made to determine if individual library members are up-to-date; all members of a library are always updated. Special targets support manipulation of a single object file; the target `objfile` updates file `objfile.$O` in the library of the current architecture and the target `objfile.all` updates `objfile.$O` in the libraries of all architectures.

### Overrides

The rules provided by a generic mkfile or the variables used to control the evaluation of those rules may be overridden in most circumstances. Overrides must be specified in the product mkfile after the point where the generic mkfile is included; in general, variable and rule overrides occupy the end of a product mkfile.

The value of a variable is overridden by assigning a new value to the variable. Most variable overrides modify the values of flags or the names of commands executed in recipes. For example, the default value of CFLAGS is often overridden or augmented and the ANSI/POSIX Computing Environment is selected by setting the CC and LD variables to `pcc`.

Modifying rules is trickier than modifying variables. Additional constraints can be added to a rule by specifying the target and the new prerequisite. For example,

```
%.$O:   header.h
```

adds file `header.h` the set of prerequisites for all object files. There is no mechanism for adding additional commands to an existing recipe; if a recipe is unsatisfactory, the rule and its recipe must be completely overridden. A rule is overridden only when the replacement rule matches the target and prerequisite portions of the original rule

exactly. The recipe associated with the new rule then replaces the recipe of the original rule. For example, /sys/src/cmd/lex/mkfile overrides the default installall rule to perform the normal loop on all architectures and then copy a prototype file to the system library directory.

```
</$objtype/mkfile

TARG=lex
OFILES=lmain.$O\
        y.tab.$O\
        sub1.$O\
        sub2.$O\
        header.$O\

HFILES=ldefs.h\

YFILES=parser.y\

BIN=/$objtype/bin
</sys/src/cmd/mkone

installall:V:
        for(objtype in $CPUS)
                mk install
        cp ncform /sys/lib/lex
```

Another way to perform the same override is to add a dependency to the default installall rule that executes an additional rule to install the prototype file:

```
installall:V:   ncform.install

ncform.install:V:
        cp ncform /sys/lib/lex
```

**Special Tricks**

Two special cases require extra deviousness.

In the first, a file needed to build an executable is generated by a program that, in turn, is built from a source file that is not part of the product. In this case, the executable must be built for the target architecture, but the intermediate executable must be built for the architecture mk is executing on. The intermediate executable is built by recursively invoking mk with the appropriate target and the executing architecture as the target architecture. When that mk completes, the intermediate is executed to generate the source file to complete the build for the target architecture. The earlier example of /sys/src/cmd/awk/mkfile illustrates this technique.

Another awkward situation occurs when a directory contains source to build an executable as well as source for auxiliary executables that are not to be installed. In this case the mkmany generic rules are inappropriate, because all executables would be built and installed. Instead, use the mkone generic file to build the primary executable and provide extra targets to build the auxiliary files. This approach is also useful when the auxiliary files are not executables; /sys/src/cmd/spell/mkfile augments the default rules to build and install the spell executable with elaborate rules to generate and maintain the auxiliary spelling lists.

# A Manual for the Plan 9 assembler

*Rob Pike*
*rob@plan9.bell-labs.com*

## Machines

There is an assembler for each of the MIPS, SPARC, Intel 386, AMD64, Power PC, and ARM. The 68020 assembler, `2a`, (no longer distributed) is the oldest and in many ways the prototype. The assemblers are really just variations of a single program: they share many properties such as left-to-right assignment order for instruction operands and the synthesis of macro instructions such as `MOVE` to hide the peculiarities of the load and store structure of the machines. To keep things concrete, the first part of this manual is specifically about the 68020. At the end is a description of the differences among the other assemblers.

The document, ''How to Use the Plan 9 C Compiler'', by Rob Pike, is a prerequisite for this manual.

## Registers

All pre-defined symbols in the assembler are upper-case. Data registers are `R0` through `R7`; address registers are `A0` through `A7`; floating-point registers are `F0` through `F7`.

A pointer in `A6` is used by the C compiler to point to data, enabling short addresses to be used more often. The value of `A6` is constant and must be set during C program initialization to the address of the externally-defined symbol `a6base`.

The following hardware registers are defined in the assembler; their meaning should be obvious given a 68020 manual: `CAAR`, `CACR`, `CCR`, `DFC`, `ISP`, `MSP`, `SFC`, `SR`, `USP`, and `VBR`.

The assembler also defines several pseudo-registers that manipulate the stack: `FP`, `SP`, and `TOS`. `FP` is the frame pointer, so `0(FP)` is the first argument, `4(FP)` is the second, and so on. `SP` is the local stack pointer, where automatic variables are held (`SP` is a pseudo-register only on the 68020); `0(SP)` is the first automatic, and so on as with `FP`. Finally, `TOS` is the top-of-stack register, used for pushing parameters to procedures, saving temporary values, and so on.

The assembler and loader track these pseudo-registers so the above statements are true regardless of what has been pushed on the hardware stack, pointed to by `A7`. The name `A7` refers to the hardware stack pointer, but beware of mixed use of `A7` and the above stack-related pseudo-registers, which will cause trouble. Note, too, that the `PEA` instruction is observed by the loader to alter SP and thus will insert a corresponding pop before all returns. The assembler accepts a label-like name to be attached to `FP` and `SP` uses, such as `p+0(FP)`, to help document that `p` is the first argument to a routine. The name goes in the symbol table but has no significance to the result of the program.

### Referring to data

All external references must be made relative to some pseudo-register, either PC (the virtual program counter) or SB (the ''static base'' register). PC counts instructions, not bytes of data. For example, to branch to the second following instruction, that is, to skip one instruction, one may write

```
        BRA     2(PC)
```

Labels are also allowed, as in

```
        BRA     return
        NOP
return:
        RTS
```

When using labels, there is no (PC) annotation.

The pseudo-register SB refers to the beginning of the address space of the program. Thus, references to global data and procedures are written as offsets to SB, as in

```
        MOVL    $array(SB), TOS
```

to push the address of a global array on the stack, or

```
        MOVL    array+4(SB), TOS
```

to push the second (4-byte) element of the array. Note the use of an offset; the complete list of addressing modes is given below. Similarly, subroutine calls must use SB:

```
        BSR     exit(SB)
```

File-static variables have syntax

```
        local<>+4(SB)
```

The <> will be filled in at load time by a unique integer.

When a program starts, it must execute

```
        MOVL    $a6base(SB), A6
```

before accessing any global data. (On machines such as the MIPS and SPARC that cannot load a register in a single instruction, constants are loaded through the static base register. The loader recognizes code that initializes the static base register and treats it specially. You must be careful, however, not to load large constants on such machines when the static base register is not set up, such as early in interrupt routines.)

### Expressions

Expressions are mostly what one might expect. Where an offset or a constant is expected, a primary expression with unary operators is allowed. A general C constant expression is allowed in parentheses.

Source files are preprocessed exactly as in the C compiler, so #define and #include work.

### Addressing modes

The simple addressing modes are shared by all the assemblers. Here, for completeness, follows a table of all the 68020 addressing modes, since that machine has the richest set. In the table, o is an offset, which if zero may be elided, and d is a displacement, which is a constant between -128 and 127 inclusive. Many of the modes listed have the same name; scrutiny of the format will show what default is being applied. For instance, indexed mode with no address register supplied operates as though a zero-valued register were used. For "offset" read "displacement." For ".s" read one of .L, or .W followed by *1, *2, *4, or *8 to indicate the size and scaling of

the data.

| | |
|---|---|
| data register | R0 |
| address register | A0 |
| floating-point register | F0 |
| special names | CAAR, CACR, etc. |
| constant | $con |
| floating point constant | $fcon |
| external symbol | name+o(SB) |
| local symbol | name<>+o(SB) |
| automatic symbol | name+o(SP) |
| argument | name+o(FP) |
| address of external | $name+o(SB) |
| address of local | $name<>+o(SB) |
| indirect post-increment | (A0)+ |
| indirect pre-decrement | -(A0) |
| indirect with offset | o(A0) |
| indexed with offset | o()(R0.s) |
| indexed with offset | o(A0)(R0.s) |
| external indexed | name+o(SB)(R0.s) |
| local indexed | name<>+o(SB)(R0.s) |
| automatic indexed | name+o(SP)(R0.s) |
| parameter indexed | name+o(FP)(R0.s) |
| offset indirect post-indexed | d(o())(R0.s) |
| offset indirect post-indexed | d(o(A0))(R0.s) |
| external indirect post-indexed | d(name+o(SB))(R0.s) |
| local indirect post-indexed | d(name<>+o(SB))(R0.s) |
| automatic indirect post-indexed | d(name+o(SP))(R0.s) |
| parameter indirect post-indexed | d(name+o(FP))(R0.s) |
| offset indirect pre-indexed | d(o()(R0.s)) |
| offset indirect pre-indexed | d(o(A0)) |
| offset indirect pre-indexed | d(o(A0)(R0.s)) |
| external indirect pre-indexed | d(name+o(SB)) |
| external indirect pre-indexed | d(name+o(SB)(R0.s)) |
| local indirect pre-indexed | d(name<>+o(SB)) |
| local indirect pre-indexed | d(name<>+o(SB)(R0.s)) |
| automatic indirect pre-indexed | d(name+o(SP)) |
| automatic indirect pre-indexed | d(name+o(SP)(R0.s)) |
| parameter indirect pre-indexed | d(name+o(FP)) |
| parameter indirect pre-indexed | d(name+o(FP)(R0.s)) |

**Laying down data**

Placing data in the instruction stream, say for interrupt vectors, is easy: the pseudo-instructions LONG and WORD (but not BYTE) lay down the value of their single argument, of the appropriate size, as if it were an instruction:

```
LONG    $12345
```

places the long 12345 (base 10) in the instruction stream. (On most machines, the only such operator is WORD and it lays down 32-bit quantities. The 386 has all three: LONG, WORD, and BYTE. The AMD64 adds QUAD to that for 64-bit values. The 960 has only one, LONG.)

Placing information in the data section is more painful. The pseudo-instruction DATA does the work, given two arguments: an address at which to place the item, including its size, and the value to place there. For example, to define a character array

array containing the characters abc and a terminating null:

```
DATA    array+0(SB)/1, $'a'
DATA    array+1(SB)/1, $'b'
DATA    array+2(SB)/1, $'c'
GLOBL   array(SB), $4
```

or

```
DATA    array+0(SB)/4, $"abc\z"
GLOBL   array(SB), $4
```

The /1 defines the number of bytes to define, GLOBL makes the symbol global, and the $4 says how many bytes the symbol occupies. Uninitialized data is zeroed automatically. The character \z is equivalent to the C \0. The string in a DATA statement may contain a maximum of eight bytes; build larger strings piecewise. Two pseudo-instructions, DYNT and INIT, allow the (obsolete) Alef compilers to build dynamic type information during the load phase. The DYNT pseudo-instruction has two forms:

```
DYNT    , ALEF_SI_5+0(SB)
DYNT    ALEF_AS+0(SB), ALEF_SI_5+0(SB)
```

In the first form, DYNT defines the symbol to be a small unique integer constant, chosen by the loader, which is some multiple of the word size. In the second form, DYNT defines the second symbol in the same way, places the address of the most recently defined text symbol in the array specified by the first symbol at the index defined by the value of the second symbol, and then adjusts the size of the array accordingly.

The INIT pseudo-instruction takes the same parameters as a DATA statement. Its symbol is used as the base of an array and the data item is installed in the array at the offset specified by the most recent DYNT pseudo-instruction. The size of the array is adjusted accordingly. The DYNT and INIT pseudo-instructions are not implemented on the 68020.

**Defining a procedure**

Entry points are defined by the pseudo-operation TEXT, which takes as arguments the name of the procedure (including the ubiquitous (SB)) and the number of bytes of automatic storage to pre-allocate on the stack, which will usually be zero when writing assembly language programs. On machines with a link register, such as the MIPS and SPARC, the special value -4 instructs the loader to generate no PC save and restore instructions, even if the function is not a leaf. Here is a complete procedure that returns the sum of its two arguments:

```
TEXT    sum(SB), $0
        MOVL    arg1+0(FP), R0
        ADDL    arg2+4(FP), R0
        RTS
```

An optional middle argument to the TEXT pseudo-op is a bit field of options to the loader. Setting the 1 bit suspends profiling the function when profiling is enabled for the rest of the program. For example,

```
TEXT    sum(SB), 1, $0
        MOVL    arg1+0(FP), R0
        ADDL    arg2+4(FP), R0
        RTS
```

will not be profiled; the first version above would be. Subroutines with peculiar state, such as system call routines, should not be profiled.

Setting the 2 bit allows multiple definitions of the same TEXT symbol in a program; the loader will place only one such function in the image. It was emitted only by the Alef compilers.

Subroutines to be called from C should place their result in R0, even if it is an address. Floating point values are returned in F0. Functions that return a structure to a C program receive as their first argument the address of the location to store the result; R0 is unused in the calling protocol for such procedures. A subroutine is responsible for saving its own registers, and therefore is free to use any registers without saving them (''caller saves''). A6 and A7 are the exceptions as described above.

### When in doubt

If you get confused, try using the −S option to 2c and compiling a sample program. The standard output is valid input to the assembler.

### Instructions

The instruction set of the assembler is not identical to that of the machine. It is chosen to match what the compiler generates, augmented slightly by specific needs of the operating system. For example, 2a does not distinguish between the various forms of MOVE instruction: move quick, move address, etc. Instead the context does the job. For example,

```
MOVL    $1, R1
MOVL    A0, R2
MOVW    SR, R3
```

generates official MOVEQ, MOVEA, and MOVESR instructions. A number of instructions do not have the syntax necessary to specify their entire capabilities. Notable examples are the bitfield instructions, the multiply and divide instructions, etc. For a complete set of generated instruction names (in 2a notation, not Motorola's) see the file /sys/src/cmd/2c/2.out.h. Despite its name, this file contains an enumeration of the instructions that appear in the intermediate files generated by the compiler, which correspond exactly to lines of assembly language.

### Laying down instructions

The loader modifies the code produced by the assembler and compiler. It folds branches, copies short sequences of code to eliminate branches, and discards unreachable code. The first instruction of every function is assumed to be reachable. The pseudo-instruction NOP, which you may see in compiler output, means no instruction at all, rather than an instruction that does nothing. The loader discards all NOP's.

To generate a true NOP instruction, or any other instruction not known to the assembler, use a WORD pseudo-instruction. Such instructions on RISCs are not scheduled by the loader and must have their delay slots filled manually.

### MIPS

The registers are only addressed by number: R0 through R31. R29 is the stack pointer; R30 is used as the static base pointer, the analogue of A6 on the 68020. Its value is the address of the global symbol setR30(SB). The register holding returned values from subroutines is R1. When a function is called, space for the first argument is reserved at 0(FP) but in C (not Alef) the value is passed in R1 instead.

The loader uses R28 as a temporary. The system uses R26 and R27 as interrupt-time temporaries. Therefore none of these registers should be used in user code.

The control registers are not known to the assembler. Instead they are numbered registers M0, M1, etc. Use this trick to access, say, STATUS:

```
#define STATUS  12
        MOVW    M(STATUS), R1
```

Floating point registers are called F0 through F31. By convention, F24 must be initialized to the value 0.0, F26 to 0.5, F28 to 1.0, and F30 to 2.0; this is done by the operating system.

The instructions and their syntax are different from those of the manufacturer's manual. There are no lui and kin; instead there are MOVW (move word), MOVH (move halfword), and MOVB (move byte) pseudo-instructions. If the operand is unsigned, the instructions are MOVHU and MOVBU. The order of operands is from left to right in dataflow order, just as on the 68020 but not as in MIPS documentation. This means that the Bcond instructions are reversed with respect to the book; for example, a va BGTZ generates a MIPS bltz instruction.

The assembler is for the R2000, R3000, and most of the R4000 and R6000 architectures. It understands the 64-bit instructions MOVV, MOVVL, ADDV, ADDVU, SUBV, SUBVU, MULV, MULVU, DIVV, DIVVU, SLLV, SRLV, and SRAV. The assembler does not have any cache, load-linked, or store-conditional instructions.

Some assembler instructions are expanded into multiple instructions by the loader. For example the loader may convert the load of a 32 bit constant into an lui followed by an ori.

Assembler instructions should be laid out as if there were no load, branch, or floating point compare delay slots; the loader will rearrange—*schedule*—the instructions to guarantee correctness and improve performance. The only exception is that the correct scheduling of instructions that use control registers varies from model to model of machine (and is often undocumented) so you should schedule such instructions by hand to guarantee correct behavior. The loader generates

```
        NOR     R0, R0, R0
```

when it needs a true no-op instruction. Use exactly this instruction when scheduling code manually; the loader recognizes it and schedules the code before it and after it independently. Also, WORD pseudo-ops are scheduled like no-ops.

The NOSCHED pseudo-op disables instruction scheduling (scheduling is enabled by default); SCHED re-enables it. Branch folding, code copying, and dead code elimination are disabled for instructions that are not scheduled.

**SPARC**

Once you understand the Plan 9 model for the MIPS, the SPARC is familiar. Registers have numerical names only: R0 through R31. Forget about register windows: Plan 9 doesn't use them at all. The machine has 32 global registers, period. R1 [sic] is the stack pointer. R2 is the static base register, with value the address of setSB(SB). R7 is the return register and also the register holding the first argument to a C (not Alef) function, again with space reserved at 0(FP). R14 is the loader temporary.

Floating-point registers are exactly as on the MIPS.

The control registers are known by names such as FSR. The instructions to access these registers are MOVW instructions, for example

```
        MOVW    Y, R8
```

for the SPARC instruction

```
        rdy     %r8
```

Move instructions are similar to those on the MIPS: pseudo-operations that turn into appropriate sequences of `sethi` instructions, adds, etc. Instructions read from left to right. Because the arguments are flipped to SUBCC, the condition codes are not inverted as on the MIPS.

The syntax for the ASI stuff is, for example to move a word from ASI 2:

```
MOVW    (R7, 2), R8
```

The syntax for double indexing is

```
MOVW    (R7+R8), R9
```

The SPARC's instruction scheduling is similar to the MIPS's. The official no-op instruction is:

```
ORN     R0, R0, R0
```

### i960

Registers are numbered R0 through R31. Stack pointer is R29; return register is R4; static base is R28; it is initialized to the address of `setSB(SB)`. R3 must be zero; this should be done manually early in execution by

```
SUBO    R3, R3
```

R27 is the loader temporary.

There is no support for floating point.

The Intel calling convention is not supported and cannot be used; use BAL instead. Instructions are mostly as in the book. The major change is that LOAD and STORE are both called MOV. The extension character for MOV is as in the manual: O for ordinal, W for signed, etc.

### i386

The assembler assumes 32-bit protected mode. The register names are SP, AX, BX, CX, DX, BP, DI, and SI. The stack pointer (not a pseudo-register) is SP and the return register is AX. There is no physical frame pointer but, as for the MIPS, FP is a pseudo-register that acts as a frame pointer.

Opcode names are mostly the same as those listed in the Intel manual with an L, W, or B appended to identify 32-bit, 16-bit, and 8-bit operations. The exceptions are loads, stores, and conditionals. All load and store opcodes to and from general registers, special registers (such as CR0, CR3, GDTR, IDTR, SS, CS, DS, ES, FS, and GS) or memory are written as

```
MOVx    src,dst
```

where *x* is L, W, or B. Thus to get AL use a MOVB instruction. If you need to access AH, you must mention it explicitly in a MOVB:

```
MOVB    AH, BX
```

There are many examples of illegal moves, for example,

```
MOVB    BP, DI
```

that the loader actually implements as pseudo-operations.

The names of conditions in all conditional instructions (J, SET) follow the conventions of the 68020 instead of those of the Intel assembler: JOS, JOC, JCS, JCC, JEQ, JNE, JLS, JHI, JMI, JPL, JPS, JPC, JLT, JGE, JLE, and JGT instead of JO, JNO, JB, JNB, JZ, JNZ, JBE, JNBE, JS, JNS, JP, JNP, JL, JNL, JLE, and JNLE.

The addressing modes have syntax like AX, (AX), (AX)(BX*4), 10(AX), and 10(AX)(BX*4). The offsets from AX can be replaced by offsets from FP or SB to access names, for example extern+5(SB)(AX*2).

Other notes: Non-relative JMP and CALL have a * added to the syntax. Only LOOP, LOOPEQ, and LOOPNE are legal loop instructions. Only REP and REPN are recognized repeaters. These are not prefixes, but rather stand-alone opcodes that precede the strings, for example

```
CLD; REP; MOVSL
```

Segment override prefixes in MOD/RM fields are not supported.

**AMD64**

The assembler assumes 64-bit mode unless a MODE pseudo-operation is given:

```
MODE $32
```

to change to 32-bit mode. The effect is mainly to diagnose instructions that are illegal in the given mode, but the loader will also assume 32-bit operands and addresses, and 32-bit PC values for call and return. The assembler's conventions are similar to those for the 386, above. The architecture provides extra fixed-point registers R8 to R15. All registers are 64 bit, but instructions access low-order 8, 16 and 32 bits as described in the processor handbook. For example, MOVL to AX puts a value in the low-order 32 bits and clears the top 32 bits to zero. Literal operands are limited to signed 32 bit values, which are sign-extended to 64 bits in 64 bit operations; the exception is MOVQ, which allows 64-bit literals. The external registers in Plan 9's C are allocated from R15 down.

There are many new instructions, including the MMX and XMM media instructions, and conditional move instructions. MMX registers are M0 to M7, and XMM registers are X0 to X15. As with the 386 instruction names, all new 64-bit integer instructions, and the MMX and XMM instructions uniformly use L for 'long word' (32 bits) and Q for 'quad word' (64 bits). Some instructions use O ('octword') for 128-bit values, where the processor handbook variously uses O or DQ. The assembler also consistently uses PL for 'packed long' in XMM instructions, instead of Q, DQ or PI. Either MOVL or MOVQ can be used to move values to and from control registers, even when the registers might be 64 bits. The assembler often accepts the handbook's name to ease conversion of existing code (but remember that the operand order is uniformly source then destination).

C's long long type is 64 bits, but passed and returned by value, not by reference. More notably, C pointer values are 64 bits, and thus long long and unsigned long long are the only integer types wide enough to hold a pointer value. The C compiler and library use the XMM floating-point instructions, not the old 387 ones, although the latter are implemented by assembler and loader. Unlike the 386, the first integer or pointer argument is passed in a register, which is BP for an integer or pointer (it can be referred to in assembly code by the pseudonym RARG). AX holds the return value from subroutines as before. Floating-point results are returned in X0, although currently the first floating-point parameter is not passed in a register. All parameters less than 8 bytes in length have 8 byte slots reserved on the stack to preserve alignment and simplify variable-length argument list access, including the first parameter when passed in a register, even though bytes 4 to 7 are not initialized.

**Power PC**

The Power PC follows the Plan 9 model set by the MIPS and SPARC, not the elaborate ABIs. The 32-bit instructions of the 60x and 8xx PowerPC architectures are supported; there is no support for the older POWER instructions. Registers are R0 through R31. R0 is initialized to zero; this is done by C start up code and assumed by the

compiler and loader. R1 is the stack pointer. R2 is the static base register, with value the address of setSB(SB). R3 is the return register and also the register holding the first argument to a C function, with space reserved at 0(FP) as on the MIPS. R31 is the loader temporary. The external registers in Plan 9's C are allocated from R30 down.

Floating point registers are called F0 through F31. By convention, several registers are initialized to specific values; this is done by the operating system. F27 must be initialized to the value 0x4330000080000000 (used by float-to-int conversion), F28 to the value 0.0, F29 to 0.5, F30 to 1.0, and F31 to 2.0.

As on the MIPS and SPARC, the assembler accepts arbitrary literals as operands to MOVW, and also to ADD and others where 'immediate' variants exist, and the loader generates sequences of addi, addis, oris, etc. as required. The register indirect addressing modes use the same syntax as the SPARC, including double indexing when allowed.

The instruction names are generally derived from the Motorola ones, subject to slight transformation: the '.' marking the setting of condition codes is replaced by CC, and when the letter 'o' represents 'OE=1' it is replaced by V. Thus add, addo. and subfzeo. become ADD, ADDVCC and SUBFZEVCC. As well as the three-operand conditional branch instruction BC, the assembler provides pseudo-instructions for the common cases: BEQ, BNE, BGT, BGE, BLT, BLE, BVC, and BVS. The unconditional branch instruction is BR. Indirect branches use (CTR) or (LR) as target.

Load or store operations are replaced by MOV variants in the usual way: MOVW (move word), MOVH (move halfword with sign extension), and MOVB (move byte with sign extension, a pseudo-instruction), with unsigned variants MOVHZ and MOVBZ, and byte-reversing MOVWBR and MOVHBR. 'Load or store with update' versions are MOVWU, MOVHU, and MOVBZU. Load or store multiple is MOVMW. The exceptions are the string instructions, which are LSW and STSW, and the reservation instructions lwarx and stwcx., which are LWAR and STWCCC, all with operands in the usual data-flow order. Floating-point load or store instructions are FMOVD, FMOVDU, FMOVS, and FMOVSU. The register to register move instructions fmr and fmr. are written FMOVD and FMOVDCC.

The assembler knows the commonly used special purpose registers: CR, CTR, DEC, LR, MSR, and XER. The rest, which are often architecture-dependent, are referenced as SPR(n). The segment registers of the 60x series are similarly SEG(n), but *n* can also be a register name, as in SEG(R3). Moves between special purpose registers and general purpose ones, when allowed by the architecture, are written as MOVW, replacing mfcr, mtcr, mfmsr, mtmsr, mtspr, mfspr, mftb, and many others.

The fields of the condition register CR are referenced as CR(0) through CR(7). They are used by the MOVFL (move field) pseudo-instruction, which produces mcrf or mtcrf. For example:

```
MOVFL   CR(3), CR(0)
MOVFL   R3, CR(1)
MOVFL   R3, $7, CR
```

They are also accepted in the conditional branch instruction, for example

```
BEQ     CR(7), label
```

Fields of the FPSCR are accessed using MOVFL in a similar way:

```
MOVFL   FPSCR, F0
MOVFL   F0, FPSCR
MOVFL   F0, $7, FPSCR
MOVFL   $0, FPSCR(3)
```

producing mffs, mtfsf or mtfsfi, as appropriate.

**ARM**

The assembler provides access to R0 through R14 and the PC. The stack pointer is R13, the link register is R14, and the static base register is R12. R0 is the return register and also the register holding the first argument to a subroutine. The external registers in Plan 9's C are allocated from R10 down. R11 is used by the loader as a temporary register. The assembler supports the CPSR and SPSR registers. It also knows about coprocessor registers C0 through C15. Floating registers are F0 through F7, FPSR and FPCR.

As with the other architectures, loads and stores are called MOV, e.g. MOVW for load word or store word, and MOVM for load or store multiple, depending on the operands.

Addressing modes are supported by suffixes to the instructions: .IA (increment after), .IB (increment before), .DA (decrement after), and .DB (decrement before). These can only be used with the MOV instructions. The move multiple instruction, MOVM, defines a range of registers using brackets, e.g. [R0-R12]. The special MOVM addressing mode bits W, U, and P are written in the same manner, for example, MOVM.DB.W. A .S suffix allows a MOVM instruction to access user R13 and R14 when in another processor mode. Shifts and rotates in addressing modes are supported by binary operators << (logical left shift), >> (logical right shift), -> (arithmetic right shift), and @> (rotate right); for example R7>>R2or R2@>2. The assembler does not support indexing by a shifted expression; only names can be doubly indexed.

Any instruction can be followed by a suffix that makes the instruction conditional: .EQ, .NE, and so on, as in the ARM manual, with synonyms .HS (for .CS) and .LO (for .CC), for example ADD.NE. Arithmetic and logical instructions can have a .S suffix, as ARM allows, to set condition codes.

The syntax of the MCR and MRC coprocessor instructions is largely as in the manual, with the usual adjustments. The assembler directly supports only the ARM floating-point coprocessor operations used by the compiler: CMP, ADD, SUB, MUL, and DIV, all with F or D suffix selecting single or double precision. Floating-point load or store become MOVF and MOVD. Conversion instructions are also specified by moves: MOVWD, MOVWF, MOVDW, MOVWD, MOVFD, and MOVDF.