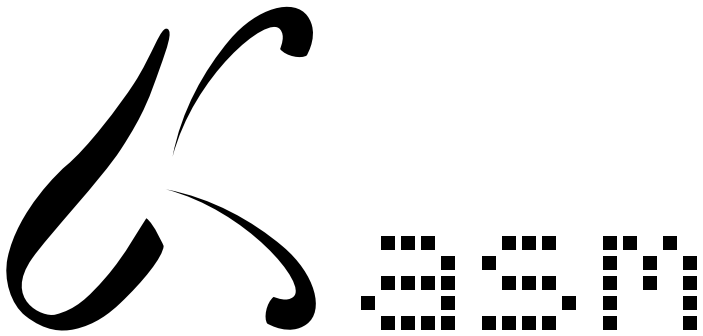

A PERL-BASED CROSS-ASSEMBLER

User's Guide v0.104
July 8, 2003

Contact. *The author may be reached at* `bret@ugcs.caltech.edu` *or visited at*
`http://www.ugcs.caltech.edu/~bret`

Acknowledgements. *The author is deeply indebted to Larry Wall, et al, for making the easy things easy and the hard things enjoyable; and Larry Wall, personally, for demonstrating how technical documentation is supposed be written.*

Colophon. `bkasm` *was written in GNU Emacs 21.2.1 and developed with ActiveState's distribution of* perl v5.8.0 *for Windows. This document was also written in GNU Emacs, and was typeset with the MiKTEX 2.2 distribution of LATEX 2ε.*

Contents

Chapter 1

Introduction

`bkasm` is a Perl-based cross-assembler designed for small embedded processors, particularly digital signal processors. It currently supports the AL3101 "1K" DSP from Alesis Semiconductor, and additional targets may be accommodated easily.

"Perl-based" can mean a couple of different things. In the literal sense, `bkasm` is written in the Perl programming language and makes no secret of that; in fact, its Perlishness is exposed to the user in several ways. If you know Perl, you can use it to craft complicated expressions, dynamically generate bits (or entireties) of assembly code, and even package up your dynamic code generators as modular, reusable macros. If you feel like looking under the hood, then you can use Perl to extend the assembler itself, adding your own directives, targets, and output formats, or building on the existing ones.

But you certainly don't *need* to know anything about Perl in order to use `bkasm`, any more than you need to be a mechanic to drive a car. The other meaning of "Perl-based" is found in `bkasm`'s design philosophy.

Like Perl, `bkasm` is one of the most powerful tools of its kind. Its hefty feature set would take hundreds of pages to describe in depth. Features can be good. But they are not as important as *you*. Your job is to write code; the job of the features is to help you do yours, not to get in the way. Thus, `bkasm` never forces to confront features that you don't care about. In fact, you don't need to know much of anything to get started with `bkasm`. Type in a few lines of bare assembly code, and `bkasm`

will give you back a few words of object code. Sure, `bkasm` knows powerful ways of dealing with sections, linking, precision, complex math, memory allocation, and all sorts of other fancy things. But it doesn't mind holding them in reserve until you ask for them. This is known as *scalable complexity*, or "what you don't know won't hurt you".

Like Perl, `bkasm` is familiar and eclectic. In the spirit of not reinventing the wheel for users who are stuck in a rut, much of `bkasm`'s syntax and vocabulary is borrowed from somewhere, although "somewhere" could be any of an variety of influences—ANSI C, GNU utilities, Perl, Matlab, UNIX shells, and of course, various assemblers of all sorts. If you have some amount of programming experience, you should be able to look at any `bkasm` expression or directive and have some idea of what it does.

Like Perl, `bkasm` provides for the general case but is tuned for the common case. Perl calls this "making the easy things easy and the hard things possible". For example, to allocate some data memory, you could say something like this:

```
foo  .data  bar, 16
```

This allocates sixteen words in the *bardata* chunk of the current section. `bkasm` allows you to carve up your memory space into as many chunks as you like, and even have overlapping chunks for unusual applications. That's a great feature, but it's not typically something you care about. You normally want to allocate straight from the main memory pool, without worrying about chunking it up. So, there's a shorthand:

```
foo  .ds  16
```

This happens to allocate sixteen words in the *sdata* chunk, but since the *sdata* chunk is mapped by default to the entire main memory pool, you can just think of it as "sixteen words of data". It also happens to be the mnemonic used by many other assemblers. But even this isn't good enough, because you typically only want to allocate one word at a time. So you can just say:

```
foo  .ds
```

This grabs you one word of memory, and it can't get much more brief than that.

Like Perl, `bkasm` was designed to evolve. `bkasm` did not descend from a mountain as inscriptions in stone tablets—it was written by a human, subject to fallacies and fashions. Ideas change; `bkasm` should be able to gracefully change with them. As a simple example, `bkasm` is uncharacteristically restrictive when it comes to certain parts of the syntax: a label *must* end with a colon, and a directive *must* start with

a dot. These are not crutches for the parser; they form an intentional partitioning of the namespace. They allow a target's vocabulary to expand and new directives to be invented without colliding with each other or with user-defined labels. `bkasm` tries to be forward as well as backward compatible.

Like Perl, `bkasm` is extensible. At the user level, `bkasm`'s pattern macros allow you extend the syntax almost arbitrarily. If you are in a reverse Polish mood, for example, and would like to write "$a = b + 7$" as "`(b)(7)+`", you can create a simple macro to do that. Under the hood, `bkasm`'s targets, linker, and output formats are conventional, object-oriented Perl modules, and you may write your own modules from scratch or subclass existing ones. Plugging in a module is a simple matter of naming it on the command line.

Like Perl, `bkasm` is open and free. Anybody may use `bkasm`, and anybody may modify it to suit their needs. Modifiers are encouraged to share their modifications, but they don't have to. `bkasm`'s license allows you to do pretty much anything you like with it, within reason.

`bkasm` may be "just" an assembler, but if your job is writing assembly code all day, the assembler is an important part of your life. `bkasm` was designed to be as powerful as you'll need, and as friendly as you'd like. So write some code, process some signals, do your job, and have some fun.

User's Automatic

Chapter 2

User's Automatic

You don't *really* want to the read the manual, do you?

This chapter will give you a tour of `bkasm` with as little prose and as many examples as possible. If you are already familiar with assemblers and the 1K, this may be all you need to get going. If you're not, then you might want to come back here *after* reading the manual, to let the examples guide you as you get started.

The set of features introduced here is by no means complete, and the emphasis is on specific, common cases instead of the general case. If you want completeness, that's what the manual is for.

2.1 Invocation

A typical command line for a small, single-file project:

```
% bkasm -WO -a= -Fc myfile.asm
```

`-WO` enables Warnings and Optimizations. `-a=` generates a listing file called `myfile.lst`. `-Fc` specifies the *c* output format. This generates two files, `myfile_obj.c` and `myfile_obj.h`, which represent the object code and may be compiled with your host software.

A typical command line for a larger project:

```
% bkasm -W0 -a= -l mylinkerscript.ld -o myobject *.asm
```

`*.asm` grabs and assembles all of the matching files in the directory. `-l` refers to a linker script. The output format and other options may be specified in this script, as is assumed here. If it specifies the *c* output format, `-o` will make it generate two files, `myobject.c` and `myobject.h`. The listing file will be `myobject.lst`.

2.2 Basic Syntax

```
;;; Comments start with a semi-colon, and extend to the end of the line.

;;; The ".include" directive will insert another source file.
.include "my_favorite_definitions.inc"

;;; Arithmetic is indicated by C-like statements that assign to "a".
a = 1.2*a + b         ; generates:  cab  1.2
a += 7                ; generates:  1ac  7    (same as:  a = a + 7)

;;; Each statement must algebraically represent a single instruction
a = (a + 2)*(a - 2)   ; generates:  aac  -4   (same as:  a = a^2 - 4)
a = a^2 + b           ; error:  No such instruction exists.

;;; Multiple statements on a single line are separated with commas
a = a^2,  a += b      ; generates:  aac  0,  cab  1

;;; Variables in memory are indicated by an address in [square brackets].
a = [0x30] + 2        ; generates:  1mc  2  0x30
a *= [myaddr + 2]     ; generates:  amc  0  myaddr+2  (myaddr is a symbol)

;;; Constants can use all C math operators and most trig functions
a = (1/2)^(1/2) + 1/sqrt(2) + sin(pi*7/4) + e^(pi*i - ln(2)/2)
```

2.3 Piggybacking

```
;;; Assignments to "b" or [memory] are combined with the subsequent
;;;  assignment to "a".
b = [myaddr]          ; piggybacks below
a += 7                ; generates:  l1ac  7  myaddr

;;; Multiple assignments can piggyback on a single instruction.
b = a                 ; piggybacks below
```

```
[myaddr] = a          ; also piggybacks below
a += 7                ; generates: sx1ac  7  myaddr

;;; If a piggyback is not possible, a dummy instruction is generated.
[myaddr] = a          ; generates:  s1ac  0  myaddr
a *= [otheraddr]      ; generates:   amc  0  otheraddr

;;; If optimization is on, "b = [memory]" piggybacks may be moved.
a += 7                ; generates:  l1ac  7  myaddr
b = [myaddr]          ; piggybacks above
a = a*[otheraddr] + b ; generates:   amb     otheraddr
```

2.4 Indirect Addressing

```
;;; Indirect addresses may be indexed by either
;;;   Ib, which equals int(b * 2^12), or
;;;   Fb, which equals int(b * 2^24)
a = [Ib]                ; generates: lindi  0x3ff  0
                        ; effective address:  int(b * 2^12) & 0x3ff
a = [Fb]                ; generates: lindf  0x3ff  0
                        ; effective address:  int(b * 2^24) & 0x3ff

;;; A fixed offset may be added to the index.
a = [myaddr + Ib]       ; generates: lindi  0x3ff  myaddr
                        ; effective address:  (myaddr + int(b * 2^12)) & 0x3ff

;;; A bitmask may be applied to the index as well.  (Use parentheses!)
a = [myaddr + (Fb & 7)] ; generates: lindf  0x007  myaddr
                        ; effective address:  (myaddr + (int(b * 2^24) & 7)) & 0x3ff

;;; Indirect storing is the same, except backwards.
[myaddr + (Ib & 7)] = a  ; generates: sindi  0x007  myaddr

;;; The I() and F() functions translate constants into the
;;;  I and F integer representations.
a = I(0x40)             ; same as:  a = 0x40 * 2^-12
b = a
a = [Ib]                ; effectively:  a = [0x40]

;;; Assigning to "Ia" represents modulo 2^16 integer arithmetic.
Ia += Ib                ; generates:  addb
                        ; effectively:  a = ((Ia + Ib) & 0xffff) * I(1)
```

2.5 Flow Control

```
 ;;; Labels are placed near the start of the line, and end with a colon.
mylabel:  a = a + b

 ;;; Local labels begin with a ~tilde.  Their scope is bounded by global labels.
globallabel:            ; start of ~locallabel's scope
~locallabel:            ; the name "locallabel" need be unique only within this scope
anothergloballabel:     ; end of ~locallabel's scope

 ;;; You may branch to a label unconditionally or conditionally.
 goto mylabel           ; unconditional branch
 if (a < 0) mylabel     ; branches only if "a" is negative

 ;;; Label-free flow control can be achieved with C-like blocks.
 if (a > 0) {           ; generates:  skip  N+Z  2
   a = [pos]            ; generates:  cm    1    pos
 }                      ; generates:  skip  T    4
 elsif (a < 0) {        ; generates:  skip  !N   2
   a = [neg]            ; generates:  c     1    neg
 }                      ; generates:  skip  T    1
 else {
   a = [zero]           ; generates:  cm    1    zero
 }

 ;;; The "allow-destructive-if" option allows for complicated conditions,
 ;;;  at the expense of implicitly clobbering "a".
 .option allow-destructive-if
 if ((a > 0) && (b < 2) || ([foo] == b && [bar]^2 != 1)) {
   a = 1            ; evaluating the above condition requires modifying "a"
   [myaddr] = a
 }

 ;;; "break" will exit a block, whether it is part of an "if" or not.
 {
   a = [foo],  b = [foo]    ; generates:  lcm  1  foo
   if (a < [bar]) break     ; generates:  cma -1  bar,  skip  N  2  ---
   a = b * [bar] + 1        ; generates:  bmc  1  bar                 |
   [foo] = a                ; generates: s1ac  0  foo                 |
 }                          ;   (skips to here)  <--------------------
```

2.6 Symbols

```
 ;;; Symbols are declared with the ".equ" directive.
MYSYMBOL  .equ  2 * sqrt(2)
```

```
 ;;; Statements may refer to symbols as if they were numbers.
 a = b/2 + sin(pi*MYSYMBOL)

 ;;; Symbols may be defined in terms of other symbols.
OTHERSYMBOL  .equ  MYSYMBOL^2 - 1/2

 ;;; A symbol may be redefined with the ".set" directive.
 a = MYSYMBOL              ; same as:  a = 2 * sqrt(2)
MYSYMBOL  .set  MYSYMBOL/4
 a = MYSYMBOL              ; same as:  a = 1/2 * sqrt(2)

 ;;; An array of symbols is declared with the ".table" directive.
MYTABLE  .table  sin(pi/5), sin(pi*2/5), sin(pi*3/5), sin(pi*4/5)
 a = MYTABLE(0)            ; same as:  a = sin(pi/5)
 a = MYTABLE(3)            ; same as:  a = sin(pi*4/5)

 ;;; A local symbol is declared with the ".lequ" directive.
 ;;; By default, its scope extends to the end of the innermost block.
 {
   FOO  .lequ  7
   a = FOO                 ; same as:  a = 7
 }                         ; FOO's scope ends here
 a = FOO                   ; error:  FOO no longer exists

 ;;; A local symbol's scope can extend to an explicit label instead.
 FOO  .lequ  7, mylabel
 a = FOO                   ; same as:  a = 7
mylabel:                   ; FOO's scope ends here
 a = FOO                   ; error:  FOO no longer exists

 ;;; The def() function tests whether a symbol is defined.  (Use quotes!)
 .if def("FOO") && def("BAR")
   a += FOO + BAR          ; only gets here if "FOO" and "BAR" both exist.
 .endif
```

2.7 Data

```
 ;;; A variable in sample memory (0 - 0x3ff) is allocated with the ".ds" directive.
myvariable  .ds         ; declare variable
 a = [myvariable] + 2   ; read from variable
 [myvariable] = a       ; write to variable

 ;;; Arrays of variables may be allocated at once.
myarray     .ds  20     ; declare array of length 20
 a = [myarray]          ; read from head of array.
 a = [myarray_TAIL]     ; read from tail of array.  Same as: a = [myarray + 19]
 a = [myarray + myarray_LENGTH - 1]  ; same as previous line.
```

```
 ;;; A variable in direct address memory (0x400 - 0x40f) is allocated with ".dv".
dirvariable  .dv
dirarray     .dv  4

 ;;; A local variable is allocated with ".ls" (or ".lv").
 ;;; By default, its scope extends to the end of the innermost block.
 ;;; When its scope is exited, its memory is deallocated and may be reused
 ;;; by other local variables.
 {
   foo  .ls             ; allocate local variable
   [foo] = a            ; use it for temporary storage
   a = [bar] + b
   a *= [foo]           ; effectively:  a *= [bar] + b
 }                      ; foo's scope ends here

 ;;; A local variable's scope can extend to an explicit label instead.
 foo  .ls  1, mylabel   ; allocate local variable
 [foo] = a              ; use it for temporary storage
 a = [bar] + b
 a *= [foo]
mylabel:                ; foo's scope ends here

 ;;; A variable's name is a symbol, and may be treated like any other symbol.
myvariable  .ds
 a = I(myvariable)      ; a = pointer to "myvariable", in I format.
 b = a                  ; transfer pointer to "b".
 a = [Ib]               ; read indirectly from pointer.  Effectively:  a = [myvariable]
```

2.8 Conditional and Iterative Assembly

```
 ;;; Code between the ".if" and ".endif" directives will only be assembled
 ;;;   if a given condition is true.
 .if  OFFSET != 0  &&  BLOCKSIZE != 0
   a = a + OFFSET*BLOCKSIZE   ; only use this statement if the addend is non-zero
 .endif

 ;;; ".else" and ".elsif" are available, and nested ".if"s are possible.
 .if    ENTREE == PASTA
   a = [linguini]
 .elsif ENTREE == PIZZA
   a = [deepdish]
   .if not def("HOLD_THE_ANCHOVIES")
     a += [catch_of_the_day]
   .endif
 .else
   a = [salad]
```

```
 .endif

 ;;; The ".repeat" directive repeats a block of code a given number of times.
 .repeat 4     ; Multiply by 2^12.  The 1K cannot multiply by more than
   a *= -8     ;   eight at a time, so we need to use four multiplications.
 .endloop      ;   (-2^3)^4 = 2^12

 ;;; The ".for" directive iterates a local symbol over a given range.
 .for OFFSET = 0 .. BUFFERLENGTH - 1
   a += [buffer + OFFSET]      ; add up the contents of an array in memory
 .endloop
 ;;; You can't name your index symbol "index" or "i".  "index" is a Perl
 ;;;  function, and "i" means sqrt(-1).

 ;;; The ".while" directive repeats a block of code while a condition is true.
 MULTIPLIER .equ 12345                        ; Multiply by 12345.
 {
   MULTIPLIER .lequ MULTIPLIER                ; Create a local copy of MULTIPLIER.
   .while MULTIPLIER < -8 || MULTIPLIER >= 8  ; Is it too big to multiply directly?
     a *= -8                                  ; Multiply by as much as we can.
     MULTIPLIER .set MULTIPLIER/(-8)          ; Divide down the local MULTIPLIER.
   .endloop                                   ; Repeat until it's small enough.
   a *= MULTIPLIER                            ; Multiply by whatever's left.
 }                                            ; Restore MULTIPLIER to 12345.
```

2.9 Macros

```
 ;;; Preprocessor-like macros may be defined as in C.
#define SCORE  20
#define YEARS_AGO(Pscore,Premainder) (Pscore*SCORE) + Premainder
 a = [YEARS_AGO(4,7)]     ; expands to:  a = [4*20 + 7]

 ;;; The ".macro" directive defines a statement-block macro.
mymacro .macro
    a = 2*a + 7
    .endm
 mymacro                  ; expands to:  a = 2*a + 7

 ;;; Arguments are listed after the directive, and must begin with '$'.
mymacro .macro $multiplier, $addend
    a = $multiplier*a + $addend
    .endm
 mymacro 2, 7             ; expands to:  a = 2*a + 7

 ;;; A quoted string after the directive indicates a pattern macro.
 ;;; The macro will be invoked if a statement matches the string.
mymacro .macro "Please multiply by two and add seven."
```

```
    a = 2*a + 7
    .endm
 Please multiply by two and add seven.        ; expands to:  a = 2*a + 7

 ;;; Pattern macros are lenient with whitespace.
 Please   multiply by two  and  add seven  .  ; expands to:  a = 2*a + 7

 ;;; The pattern may contain arguments beginning with '$'.
mymacro .macro "Please multiply by $multiplier and add $addend."
    a = $multiplier*a + $addend
    .endm
 Please multiply by 2 and add 7.   ; expands to:  a = 2*a + 7
 mymacro 2, 7                      ; This invocation still works, too.
```

2.10 Sections and Linking

2.11 Output Formats

User's Manual

Chapter 3

Invocation

`bkasm` is a command line utility. It provides no GUI or IDE; you simply offer it plain text files and it returns the same to you. This may be unfamiliar if you are used to programming Visually*, but the command line approach has a number of advantages: you can compose code using your favorite text editor and editing style (instead of the author's favorite), it allows for easy communication with other tools, it fits well into a `makefile`, it fits well into version control systems, and it's portable. The disadvantage is that you don't get a toolbar filled with rows of colorful and inscrutably tiny buttons.

The formal usage syntax, as you might find at the top of a `man` page, is something like:

```
% bkasm [OPTIONS] FILE [FILE...]
```

That's not especially helpful, of course. All it means is that running `bkasm` involves typing in the program name, followed by an optional list of a hyphen-prefixed options, followed by one or more filenames. What those `OPTIONS` and `FILE`s are supposed to be is the topic of this entire manual, but especially this chapter.

*Or being a Code Warlord.

3.1 How `bkasm` Eats Your Words

Before we get into the details of how to invoke `bkasm`, a quick overview of `bkasm`'s digestive process might be in order. It begins with you feeding in, via the command line, one or more *source files* that you have written using your aforementioned favorite text editor.As the *assembler* chews through your source, it assembles together one or more *sections*. A section is like an act or scene in a play, and represents a functional partition of your code. A simple program might only use a single default section, while a complex project may contain dozens of sections, subsections, and (sub)^{n}sections. Each section may contain a number of *chunks*. A chunk is like a character in a play, and represents a categorical partition of your code. There might be a *text* chunk with executable code, and/or one or more *data* chunks that represent different types of variable storage.* Once the assembler has digested your source into sections and chunks, it passes them to the *linker*. The linker coalesces the sections together and resolves all memory references. For a simple program, the linker's default behavior may be sufficient, but a complex project will typically feed `bkasm` a *linker script* to give the linker some guidance. Once the sections know where they and everything else are, they are passed on to the *postlinker*. The postlinker breaks down the text chunks from lists of machine instructions into the actual binary code that represents those instructions. In the final stage, this binary data is passed off to an *output format*, which decides how to give this data back to you.† Interestingly, you will typically ask the output format to package up the data as source code again. Unlike the source that you started with, however, this code is intended to be compiled in with *host* software, which will end up running on a processor that controls `bkasm`'s target processor. In addition to the output format's output, `bkasm` can also generate a *listing file*. The listing file contains your source code annotated with the assembled machine instructions, in case you are suspicious (or merely curious) about well `bkasm` chewed your source. In the case of indigestion, the listing file will also contain `bkasm`'s error messages and warnings, along with as much of the code as it could keep down. These messages will be printed to your terminal as well.

*Many compiler suites do not make this section/chunk distinction. But `bkasm` was designed with a Harvard architecture in mind (separate memory spaces for text and data), as well as for an extremely memory-constrained environment that makes complex overlays a frequent necessity. Conventional compiler suites don't handle either situation very well.

†The reader is free to extend the "digestive tract" analogy to its (scato)logical conclusion. The author will not.

3.2 Options

Minimalists will be pleased that the entire process described above can be set in motion by something as simple as:

```
% bkasm myfile.asm
```

Invoked in this way, `bkasm` reads and assembles `myfile.asm` as source code for the Alesis 1K DSP, and produces a file called `myfile.obj` which is, curiously enough, identical to what Alesis's own `1kasm` utility would produce.

However, `bkasm` is quite a bit more flexible than this example suggests, and its behavior can be customized in quite a number of ways. So many, in fact, that assigning single-letter command line switches to all of them, as is the custom, would turn your `makefile` into a bewildering, illegible mess.* Instead, all of `bkasm`'s options have descriptive names. On the command line, you can enable an option by giving its name, preceded by a double-dash:

```
% bkasm --fix-all-bugs myfile.asm
```

For some options, merely naming them is not enough. To set an option to a value, follow the option with an equals sign and the value:

```
% bkasm --bug-fixing-algorithm=achieve-sentience myfile.asm
```

If you specify the name without any value, the option is implicitly set to 1. Certain options are enabled by default, or tag along when other options are specified. In most cases, you may explicitly disable an option simply by setting it to zero:

```
% bkasm --taunt-user-when-bug-found=0 myfile.asm
```

Options aren't just the domain of the command line, however. Since options are `bkasm`'s general-purpose behavior-tweaking mechanism, it makes sense that you should be able to set and change them from within a source file as well. The `.option` assembler directive allows you to do just that. This, in turn, allows you to tweak `bkasm`'s behavior over specific ranges of your source file, or keep all of your favorite options in an external file which you `.include`. The `.option` directive will be described on page 63. In the meantime, keep in mind that when this manual refers to the `--so-and-so` option, you will often get at it with "`.option so-and-so`" instead.

*And would be impossible in any case, since there's more than 52 of them.

Since most options influence a very specific aspect of `bkasm`'s behavior, this manual will mention these options in context, wherever the tweakable behavior is discussed. But for reference purposes, you might want a complete list of options as well. That's what Appendix D is for.

3.3 Command Line Switches

Every option can be accessed through its full name, but a certain handful of options are important enough to warrant single-letter command line switches as well. Most of these options deal with basic I/O—where and what the input and output files are expected to be—with a few convenience switches that cover broad areas of `bkasm`'s behavior. All switches are optional.

`-a` *FILENAME* (`--listfile=`*FILENAME*)

If given, `bkasm` will generate a listing file and save it as the specified filename. There are a number of additional options for customizing your listing file; see Chapter 7. If the filename is given as a single hyphen ("-"), then the listing file will be written to the terminal's standard output, which can be useful if you'd like to pipe it elsewhere. If the filename is given as an equals sign ("="), then `bkasm` will figure out a good name for your listing file, based on the output or input filenames.

`-l` *FILENAME* (`--linkfile=`*FILENAME*)

If given, `bkasm` will read in the given file as a linker script, and the linker will do its best to make sense of it. If not given, a simple default linker script is used. The linker is covered in Chapter 6.

`-F` *FORMATNAME* (`--format=`*FORMATNAME*)

Specifies the output format that `bkasm` should use to generate the output file(s). If not given, the *hexdump* output format will be used, which for simple programs is compatible with the output generated by the `1kasm` utility. Output formats are covered in Chapter 8.

`-o` *FILENAME* (`--outfile=`*FILENAME*)

More or less specifies the file(s) that `bkasm` should create to represent your object code. What exactly `bkasm` does with this option is dependent on the output format, but for the *hexdump* format, it creates a file with the specified name, as you'd expect. If not given, the output format will typically figure out an appropriate output filename using the first source file on the command line.

`-I` *PATHNAME* (`--include-path=`*PATHNAME*)

This is a *list option*, which means that each time you specify it, you append a

new value to the list rather than overwriting the previous value. This option gives `bkasm` places to look when the `.include` directive is used with a filename in <pointy brackets>. The `.include` directive is covered on page 65.

`-S` *SECTIONNAME* (`--outsection=`*SECTIONNAME*)

This is also a list option, and specifies the names of sections that `bkasm` is expected to output. This can be used in addition to the `output` linker option, although the linker option is typically more convenient. The `output` linker option is covered on page ??.

`-D` *NAME=VALUE* (`--define-symbol=`*NAME=VALUE*)

This is a list option as well, and allows you to define symbols directly from the command line. You would typically use this to specify some sort of global behavior, rather than, say, redefine the value of `pi` in your `makefile`. If no value is given, it defaults to 1. Section 5.3 is all about symbols.

`-W` (`--warn`)

Enables the default set of warnings, which is highly recommended. You can gain finer control over warnings with the `--warn-*` family of options; see page 22.

`-O` (`--optimize`)

Enables the default set of optimizations, which is also highly recommended. To enable all optimizations, use the `--optimize-all` option. Specific optimizations can be enabled or disabled with the expected notation: `--optimize-company-cashflow` or `--optimize-competitor-cashflow=0`. This manual will cover each optimization in context.

`-k` (`--compatible`)

Enables a set of options to help compatibility with source written for the `1kasm` utility. Some of these options make it difficult to use some of `bkasm`'s advanced features, so it is recommended that you use this switch only when necessary. This option is equivalent to: `--case-insensitive --allow-integer-constants --allow-skip-count --allow-quote-mark-suffix`

`-n` (`--show-message-line`)

In the event of an error or warning, tells `bkasm` to print out the offending line of the source file after the error message.

`-T` *TARGETNAME* (`--target=`*TARGETNAME*)

Specifies the target that `bkasm` will assemble the source files for, which is usually the processor where the object code will be run. If not given, the default target is *1k*, which assembles for the Alesis 1K DSP. The 1K target is covered in Chapter 4;

targets in general are covered in Section 9.2.

`-L` *LINKERNAME* (`--linker=`*LINKERNAME*)

Specifies the linker that `bkasm` will use for linking. If not given, the default linker is the creatively-named *linker* linker. The default linker is covered in Chapter 6; linkers in general are covered in Section 9.3.

3.4 Messages

So far, we've been covering how you talk to `bkasm`. And although `bkasm` is generally a good listener, sometimes it has something it wants to say back to you. Messages from `bkasm` come in two flavors: *errors* and *warnings*.

`bkasm` generates an *error* when it comes across a phrase in your source file that it doesn't understand, or if you ask it to do something that it can't do. After telling you about the error, `bkasm` will usually try to pretend that the problematic phrase never existed, and continue making its way through your source. As with any compiler-like interpretation system, sometimes this works and sometimes it leads to a cascade of bogus, misleading error messages. In general, trust the first message, and only consider the subsequent messages if they appear to be unrelated. If `bkasm` finds itself generating too many error messages, it will simply give up and tell you to go fix something. You can control `bkasm`'s tolerance level with the `--max-errors=`*NUMBER* option.

`bkasm` generates a *warning* when it finds something that is suspicious, but not necessarily wrong. `bkasm` will not editorialize about your code unless you ask it to, but asking it to is highly recommended. The `--warn` option or `-W` switch will enable a default set of warnings. You may enable all possible warnings with `--warn-all`, although an overly paranoid `bkasm` can be an annoying `bkasm`. If you know the name of a warning you are interested in, you may enable or disable it individually: `--warn-useless-block-of-code` or `--warn-grammatical-errors-in-comments=0`. If there's a warning that you think is serious enough to be treated as error, you can ask for that by setting the option to "error": `--warn-rabid-doberman-behind-you=error`. If you prefer to treat all warnings as errors, you can use the aptly-named `--warnings-are-errors` option. Of course, it's hard to refer to warnings individually without knowing their names, so the `--show-error-name` option will report errors and warnings along with their internal names.

For either type of message, `bkasm` will tell you what source file inspired the message, followed by a line number in parentheses when applicable, followed by the type and meat of the message:

```
shoppinglist.asm(6): error: Apricots are out of season.  Perhaps you would
  like oranges instead?
```

This format is similar to that used by other compiler tools, which means that text editors which think they're smart enough to jump to error lines can probably parse `bkasm`'s messages without much trouble. By default, `bkasm` word-wraps messages around 78-column boundaries for easy reading. If you would like to specify your own window width, use the `--message-width=`*`NUMBER`* option. To prevent `bkasm` from doing any word-wrapping at all, set this option to zero.

If you are using included files or macros, `bkasm` will give you the entire stack of location information, so you can pinpoint exactly where the error occurred:

```
macro fruit(2) in macro produce(7) in supermarket.inc(15) in
  shoppinglist.asm(6): error: Apricots are out of season.  Perhaps you would
  like oranges instead?
```

Chapter 4

The 1K Target

The Alesis 1K is a DSP processor.* All this really means is:

- It's good at multiplying and adding numbers.
- It's not good at much else.

The 1K epitomizes the essence of a DSP processor. It can perform almost fifty million unpipelined fixed-point multiplies and additions per second, an impressive feat for a small, low-cost processor. But it knows only one trick, and that's it.

4.1 Processor Description

The 1K's instruction memory holds 1024 instructions, and the processor executes all 1024 of them, in order, once every 21 μs.† Because this architecture forbids looping constructs and subroutines, and makes time and space resource constraints explicit

*This stands for "Digital Signal Processing processor". You might use a DSP processor to calculate an FFT transform with a low SNR ratio, but not to verify a PIN number in an ATM machine.

†The 1K was designed for audio applications, and 21 μs is the period between samples of a 48 KHz signal.

and equivalent, the programming experience is a far cry from conventional software design.

The 1K contains 1040 words of data memory for variable storage, each 28 bits wide. To aid in creating delay lines, the processor provides the option to treat 1024 of these words as an implicitly-rotated circular buffer.

There are two 28-bit local registers which can be read and written more readily than data memory. The **a** register, or the *accumulator*, is where the results of most operations are stored. The **b** register functions as the accumulator's sidekick.*

All registers and data memory use a S3.24 fixed-point representation. This means that each 28-bit word is interpreted as a sign bit followed by three bits to the left of the decimal point and 24 bits to the right. This in turn means that, as far as the multiplier is concerned, values in the 1K are limited to the range $-8 \leq x < 8$, with 24-bit fractional precision. The operands of some instructions use different representations, such as S3.18 or S3.8. Although all numbers are in S3.24 format internally, the multiplier truncates one of its inputs to S3.18 before multiplying.

The 1K is typically used in a coprocessor position, babysat by a host microcontroller. Communication between the two generally consists of the host reading and writing directly into the 1K's address space using the 1K's microprocessor interface. The 1K also provides four input and four output serial ports for data transfer, but these are usually used for communicating with data converters and other DSP processors.

4.2 Machine Instruction Syntax

This section will describe the instruction syntax used in Alesis Semiconductor's processor documentation and understood by their `1kasm` utility. This syntax forms a very thin layer over the underlying machine language, and requires working at a level close to the internals of the processor. Of course, what is easy for a machine to read can be quite a bit harder for a human. Writing programs with machine instructions is difficult and error-prone, and reading them (especially programs written by others) is somewhat like trying to understand the intricacies of a hardware design by staring at the printed circuit board.

Although there is no need to learn how to program with this syntax, it may be worthwhile to gain a basic familiarity with the style in order to understand the 1K's capabilities and limitations. The syntax will also be used in this manual when

*There's another alphabetic operand known as **u**, although calling **u** a third register is a bit like referring to your car's engine compartment as a third passenger space.

describing how `bkasm` translates your source into machine instructions, and your listing files (as well as the occasional error message) will make reference to machine instructions in this way.

Most 1K instructions involve multiplying two numbers, adding a third, and writing the result to the accumulator. Each of the three participants on the right-hand side of this equation can be one of five things: **a**, **b**, **u**, a constant, or a location in data memory. In most cases, an instruction may reference no more than one constant and one memory location. This leads to a syntax where you spell out exactly what you want multiplied and added, and then specify the constant and/or memory location referenced by the instruction. For example:

```
amc  0.125  17
```

multiplies the Accumulator by a Memory location, and adds a Constant. The constant is 0.125, and the memory location is 17.

```
cb   0.125
```

multiplies a constant by the **b** register, and the adder doesn't get to play this round. The constant is 0.125, and no memory location is given because none is referenced.

Every instruction of this form writes its result to the **a** register. However, the 1K has the ability to do certain types of data transfers in parallel with these operations, and these are represented by a prefix on the instruction name. There are three of these prefix operations: Storing, Xferring, and Loading.

```
scab  0.125  17
```

will Store the accumulator into memory location 17. It then multiplies 0.125 by the accumulator, adds **b**, and writes the result back to the accumulator.

```
xcma  0.125  17
```

will transfer (Xfer?) the accumulator into the **b** register. It then multiplies 0.125 by the memory location 17, adds **a**, and writes the result back to the accumulator.

```
l1ac  0.125  17
```

will Load the value at memory location 17 into the **b** register. It then multiplies 1 by **a**, adds the constant 0.125, and writes the result back into **a**.

If this all sounds delightfully orthogonal, you're in for a disappointment. The 1K provides only fifty instructions of this form, which aren't nearly enough to cover all

of the permutations that this syntax implies. But to make up for this, the processor knows some additional useful tricks. These include integer arithmetic, bitwise masking, indirect addressing, conditional forward branching, and approximations of logarithmic and exponential functions. The machine instructions for these operations are covered in the processor documentation. You can also find a quick reference for all machine instructions in Appendix A.

`bkasm` fully supports this syntax, and programs written for the `1kasm` utility should assemble in `bkasm` without a hitch.* Furthermore, the operands of your instructions need not be simple constants or memory references, but may be full-blown expressions, as described in Section 5.2. For example:

```
amc  e^(sin(pi*2/3)-sqrt(2))+1.7  buffer+round(0,OFFSET/BLOCKSIZE)
```

The trick here, though, is that operands are delimited by whitespace. So you either have to cram each operand into a spaceless string of characters, as above, or use parentheses to keep the spaces from leaking out:

```
amc  e^( sin(pi * 2/3) - sqrt(2) )+1.7  ( buffer + round(0, OFFSET / BLOCKSIZE) )
```

For most instructions, the constant operand is assumed to be a real number in the range $-8 \leq C < 8$, and `bkasm` takes care of converting this number into the particular fixed-point representation that the instruction likes. If you'd like to be able to explicitly specify a literal integer constant to be used directly as the fixed-point representation, you will need to enable the `--allow-integer-constants` option.† With this option enabled, if you write a constant that uses a radix designator (for example, "`$f00`", "`0xfoo`", or "%01010"), the integer will be used directly as the fixed-point operand. This is emphatically not recommended because different instructions have different fixed-point representations, and `bkasm` is better at keeping track of them than you are.

```
.option allow-integer-constants
l1ac  0x100000  foo       ; adds 4 to accumulator
 1ac  0x100000            ; adds 1/16 to accumulator
```

This option does not apply to the indirect addressing instructions nor "`andc`", because they already assume an integer operand. (This is indicated by an "I" operand in Appendix A.) They also assume that you know what you're doing.

*The `--compatible` option or `-k` command line switch may occasionally be needed to ensure hitchlessness. But this option deals more with the idiosyncrasies of `1kasm` than with the 1K instruction syntax itself.

†This is one of the options that `--compatible` gives you.

In general, writing machine instructions in your source should be treated with the same reserve given to inlining assembly into a C file—do it only when you want to be perfectly explicit about what the processor should be doing. It is not uncommon to write 1K code that is intended to be dynamically modified, so it may be reasonable to write certain lines as machine instructions if you plan on fiddling with them at runtime.* But the readers and maintainers of your code will thank you if you keep the literal instructions to a minimum.

4.3 Assignment Statements

The purpose of a DSP processor, first and foremost, is to do math. If you are using a 1K, that almost certainly implies that your application has a set of mathematical equations that need to be solved. Thus, it makes sense that `bkasm`'s preferred instruction syntax closely follows the precedents set by high-level mathematically-capable computer languages, which in turn were inspired by modern algebraic notation.

Statements

Writing an equation in `bkasm` is very similar to writing it in C, or Matlab, or seventh-grade algebra class:

```
a = 2*a - 1.5
```

This is an example of a *statement*, and it does exactly what you'd expect:† it multiplies **a** by two, subtracts 1.5, and puts the result back in **a**. In order for it to do this, of course, `bkasm` must convert it to a more 1K-friendly form:

```
cad  2  -1.5
```

But you typically don't need to know this, any more than you typically need to know the assembly output of your C compiler. (And if you do want to know it, that's what the listing file is for. See Chapter 7.)

Unlike in C, you need no semi-colon to terminate a statement; a simple newline will suffice. If you wish to put multiple statements on a single line, you may delimit them with commas:

*But if you are interested in swapping around more than just an instruction or two, you probably should be using overlays. See Section 5.13.

†Provided that you have experience with C, or Matlab, or seventh-grade algebra.

```
a = b * 2,  a = a + 4,  b = a
```

You usually shouldn't, however. *"One statement per line"* is a universal guideline in the software industry because it makes things easy to read, especially if you are scanning code quickly. Nevertheless, there are a couple good reasons why you might want to put multiple statements on a line, and we'll get to those.

Elements

The purpose of every statement is to express a desired relationship among two or more *elements*. In the 1K universe, there are five basic elements: the accumulator, the **b** register, the **u** pseudo-register, memory locations, and constants. The first three are referred to simply by their names:

```
a = b + u
```

A memory reference is notated by writing its address, or any expression that resolves to an address, in square brackets. For example,

```
a = [sqrt(3*3 + 4*4)]    ; resolves to:  a = [5]
```

refers to the data stored at memory location 5. A constant is simply written as a real number, or any expression that resolves to a real number:

```
a = e^(pi*i)             ; resolves to:  a = -1
```

The details of expressions are covered in Section 5.2.

Except for the latter, `bkasm` allows you to assign to all of these elements, subject to the idiosyncrasies of the processor. The next few sections will discuss each of the elements in detail, and describe how to assign to them.

4.3.1 The Accumulator

Almost all of the 1K's instructions dump their results into the accumulator. Thus, anything more complicated than a simple data transfer must be accomplished through an assignment to the **a** register.

Forms

You do arithmetic by assigning an algebraic expression to **a**, as shown above. The full details of how expressions work must wait until Section 5.2, but for now, you can trust that the operators and their precedences are similar to those in C.* You get one bonus operator as well—Matlab's ^ exponentiation operator may be used in addition to C's ** operator.

As in C, you may combine an arithmetic operator with the assignment operator when the left operand is the assignee. This is encouraged when it enhances readability, since large sections of your code will typically consist of these sorts of transformations on the accumulator.

```
a += 1 + sqrt(2)           ; same as:  a = a + (1 + sqrt(2))
a -= 1 - sqrt(2)           ; same as:  a = a - (1 - sqrt(2))
a *= pi/2 + ANGLE_OFFSET   ; same as:  a = a * (pi/2 + ANGLE_OFFSET)
a /= 4                     ; same as:  a = a / (4),   or:  a = a * 0.25
a ^= 2                     ; same as:  a = a ^ (2),   or:  a = a * a
a  = a^2                   ; more readable way of writing previous line
```

The familiarity of this syntax may sometimes be misleading. ***bkasm*** *is not a compiler.* Each arithmetic statement you write must correspond directly to a single machine instruction. If you write a long, complicated expression, `bkasm` will not break it down for you into machine-sized chunks—it will merely complain that the 1K doesn't know how to do what you are asking of it. `bkasm` does not absolve you of the responsibility of learning your processor.

It does, however, make it much easier. Algebraic expressions tend to stick in your head better than acronyms because they correspond more closely to your mathematical thought process as you code. The types of expressions that the 1K can handle are referred to as *forms*. You will find the complete list of forms in Appendix B. You will most likely also find the process of learning and applying these forms to be an order of magnitude faster than learning machine instructions. Even veteran 1K programmers are encouraged to suspend their instruction-level knowledge so as to approach the forms with a clean slate.

Forms are semantic constructs. Syntactically, you may express a given form in almost any algebraically-equivalent manner. `bkasm`'s algebraic parser is, in general, smarter than you think it is. The following statements are equivalent:

```
a = a^2 - 4
a = (a + 2)*(a - 2)
```

*Actually, they are identical to those in Perl, since statements are evaluated directly as Perl expressions. Perl's operators, in turn, are similar to those in C.

```
a = (a + b + 1)^2 - 2*(a + 1)*(b + 1) - (b + 2*i)*(b - 2*i) + 1
aac  -4
```

All four refer to the "$a * a + C$" form, and generate the machine instruction that implements the assignment $a \leftarrow a^2 - 4$. The first statement is, of course, easiest to read, but the second and third will at least impress all your friends who think they know what an assembler is. The fourth is neither legible nor impressive, and will only make your friends pity you.*

Addition

All arithmetic instructions involve both a multiply and an addition. Thus, when you just ask for straight addition, `bkasm` will implicitly tell the 1K to multiply by 1.

```
a = b + [data]              ; generates:  cmb  1  data
```

When you write a statement that adds a constant, it may not be immediately obvious whether that constant gets 24 or 18 (or even 8) bits of fractional precision. Precision is given as part of the form in Appendix B, but different instructions provide different amounts of precision, and due to piggybacking and other concerns, you can't always be sure of which instruction you're going to get:

```
a =     a + 1/3             ; 1/3 gets 24 bits of precision
a =   2*a + 1/3             ; 1/3 gets 18 bits of precision
a = 0.2*a + 1/3             ; 1/3 gets 8 bits of precision
[data] = a
a =     a + 1/3             ; 1/3 gets 18 bits of precision
```

This is, on the whole, a good thing, since it's better to let `bkasm` manage your precision when you don't really care. When you do really care, `bkasm` provides ways of letting you tell it just how many bits you need for particular constants and symbols. Not only will `bkasm` actively work towards giving you that precision, but it will even generate an error if it can't. See Section 5.5.

Multiplication

The 1K's multiplier is probably the reason why you are using the 1K in the first place, so it might be worthwhile to take the time to get to know it. The multiplier's

*As written, there actually is a small semantic difference—the fourth line is not receptive to piggybacking by default. But that's getting ahead of ourselves. See page 38.

biggest quirk is its *asymmetry* with regards to the precision of its inputs. The 1K is not capable of doing a full S3.24 x S3.24 multiply; instead, one of the inputs must be truncated to S3.18 format. Those six bits can make a big difference, so when you request a multiply, it is important to know which element gets the 24-bit slot and which the 18-bit slot. This is determined by a fixed ordering called *precision precedence*, and you will never become an expert 1K coder until you commit this ordering to heart:

Precision Precedence: *u*, *[M]*, *a*, *b*, *C*

The element with lower precedence always gets truncated to 18-bit precision during a multiply:

```
a = a * u                   ; u      is S3.24, a   is S3.18
a = a * [data]              ; [data] is S3.24, a   is S3.18
a = a * a                   ; one copy of a is S3.24, the other is S3.18
a = a * b                   ; a      is S3.24, b   is S3.18
a = a * 1/3                 ; a      is S3.24, 1/3 is S3.18
```

One corollary of this is that you can never multiply by a S3.24 constant. Constants are lowest on the totem pole, so they always get the 18-bit slot. Another corollary is that memory references and **u** *always* get the 24-bit slot. (You cannot multiply a memory reference by **u**, so they actually have the same effective precedence.)

When you run into a situation where the 1K's precision precedence is the opposite of what you want, there are a few tricks you can use to turn it around. See page 43.

A second interesting aspect of the 1K's multiplier is that, despite what the official documentation claims, it does not produce a S3.24 result. The result is, in fact, in S4.24 format, and is not clipped to S3.24 until after the subsequent addition. For example:

```
a = 4
a = 3*a - 7                 ; results in:  a = 5
```

The final result is 5, which is correct, even though the intermediate result coming out the multiplier is 12, a number which is normally too much for the 1K to handle. Of course, if the final result *after* the addition is out of range, it will be clipped and the overflow flag will be set. Overflow will be covered on page 50.

Integer Arithmetic

The 1K is at home with saturating 28-bit fixed-point math. But sometimes you would rather it emulate the vintage arithmetic behavior of, say, the 8088. The 1K

is capable of performing classic 16-bit integer arithmetic, and this can occasionally come in handy for pointer manipulation, or with algorithms that rely on the modulo-2^{16} behavior of a standard integer ALU.

When the 1K is thinking in terms of integers, bit 12 of the 28-bit accumulator is treated as the least significant bit, and lower bits are ignored. This conveniently lines up the sign bit with the top of the accumulator, so sign tests still work.

Syntactically, integer arithmetic in `bkasm` looks similar to normal arithmetic, except with more "I"s. The pseudo-elements **Ia** and **Ib** refer to the **a** and **b** registers, but tell `bkasm` to think of the integer representation instead of fixed-point. The 1K supports three integer operations, which are written as:

```
Ia = Ia + Ib                    ; integer add:  a + b
Ia = Ia - Ib                    ; integer subtract:  a - b
Ia = Ia * Ib                    ; integer multiply:  a * b
```

In the case of multiplication, the accumulator is set to the lower 16 bits of the result. If you want the upper 16 bits instead, you can use plain old fixed-point multiplication.

As an aid in defining integer constants and otherwise massaging numbers into the 1K's integer representation, `bkasm` defines a function called `I()`:

```
I(x) = x * 2^-12
```

This function shifts an integer argument down to its proper place in the 1K's integer representation. That is, `I(1)` represents a 1 when the 1K is thinking in terms of integers.

As an example, here is how you might implement a *linear congruence generator*, which is a particularly efficient random number generator that relies upon modulo arithmetic:

```
; The formula for this linear congruence generator is:
;  next_random_number = ((previous_random_number * 25173) + 13849) mod 2^16

 b = [random_number]            ; b = previous random number
 a = I(25173)                   ; a = integer constant 25173
 Ia = Ia * Ib                   ; do the integer multiplication

 b = a                          ; b = result of multiplication
 a = I(13849)                   ; a = integer constant 13849
 Ia = Ia + Ib                   ; do the integer addition

 [random_number] = a            ; store new random number
```

The output of this random number generator goes through all 2^{16} integers, so it does not need to be seeded. Since the most significant bit is the same in both the 1K's integer and fixed-point representations, `[random_number]` appears from a fixed-point perspective to have a uniform distribution over the range $-8 \leq x < 8$. White noise enthusiasts, take note.

In case you need a conceptual anchor for this integer craziness, you can think of **I** as somewhat analogous to the prefix "milli". If you set **a** to "1 meter", then **Ia** means "**a**, as expressed in millimeters", or in this case, 1000. Correspondingly, `I(1000)` means "1000 millimeters". There are three equivalent ways to set **a** to this value:

```
a  = I(1000)              ; "a equals 1000 millimeters"
Ia = 1000                 ; "a, as expressed in millimeters, equals 1000"
a  = 1                    ; "a equals 1 meter"
```

Of course, "milli" means 10^{-3} while our **I** means 2^{-12}, but it's the same idea.

We're not through with **Ib** and `I()` yet; they will make a reappearance when we come to indirect addressing on page 44.

Masking

The 1K provides one bitwise logical operation, and that is AND. You may AND the accumulator with the **b** register:

```
a = a & b
```

or with a constant mask:

```
a = a & 3.75
```

In the latter case, the constant is interpreted as a fixed-point number, just as with any arithmetic operation. It might seem weird to write a mask as a real number rather than an integer. That's okay, because it *is* weird. But it keeps the notation consistent, and more importantly, it allows you to easily do the sorts of things you would want to do with masking:

```
a &= -0.25                  ; round down to multiple of 0.25
a +=  0.25/2,  a &= -0.25   ; round to nearest multiple of 0.25
a &= 8                      ; isolate sign bit
a &= I(0xffff)              ; isolate 1K's 16 "integer bits"
a &= 1 - F(1)               ; isolate fractional part of fixed-point number
```

The latter makes use of another one of `bkasm`'s handy predefined functions:

```
F(x) = x * 2^-24
```

This function shifts down an integer argument so its least significant bit lines up with the least significant bit of the 1K's 28-bit word. If you would rather forget about this fixed-point business and express your mask as a straightforward 28-bit integer, `F()` is what you need:

```
a &= F(0x00fff00)           ; isolate middle twelve bits
```

The one spot of awkwardness in writing masks as real numbers is that you can't directly write an expression that ANDs or ORs them with each other, because `bkasm`'s bitwise operators* only work on integers:

```
a &= (8 | F(1))             ; WRONG!  F(1) is truncated to zero.
```

You can, of course, add them together if that makes sense:

```
a &= (8 + F(1))             ; correct:  isolates highest and lowest bits
```

However, once you start composing fancy masks like this, you're probably better off writing them as integers anyway:

```
a &= F( 0x8000000 | 0x0000001 )       ; isolate highest and lowest bits
```

Incidentally, the 1K provides no OR operation. In many situations, the desired effect can be achieved through simple addition. If you want a true bitwise OR, your first inclination might be to try DeMorgan's Rule:

```
; bitwise OR using DeMorgan's Rule:  a | b = ~(~a & ~b)
 a = -[first],  a -= F(1)    ; a = one's complement of [first]
 b = a                       ; b = ~[first]
 a = -[second], a -= F(1)    ; a = one's complement of [second]
 a = a & b                   ; a = ~[first] & ~[second]
 a = -a,        a -= F(1)    ; a = [first] | [second]
```

But a more efficient implementation is possible by throwing in some addition:

```
; bitwise OR using logical/arithmetic hybrid:  a | b = (a & ~b) + b
 b = [first]                 ; b = [first]
 a = -[second],  a -= F(1)   ; a = one's complement of [second]
 a = a & b                   ; a = [first] & ~[second]
 a += [second]               ; a = [first] | [second]
```

*Or rather, Perl's bitwise operators.

Note that the one's complement technique shown above does not work for a value of -8, or `F(0x8000000)`, because the multiplier will saturate before performing a true two's complement negation. Bitwise logic isn't the 1K's strong point—handle with care.

4.3.2 Memory

The 1K has an 11-bit address space for data memory. The first half of that, from `0` to `0x3ff`, consists of *sample memory*, which is basically just 1024 places to store things. Above that, from `0x400` to `0x40f`, is *direct address memory*, which is basically 16 more places to store things. The difference is that, when the 1K is in *circular addressing mode*, sample memory is rotated up one word each sample period, while data written to direct address memory remains where you put it. When circular addressing is off, the only significant difference is that you can't get to direct address memory with indirect addressing. From `0x410` to `0x417` are the serial I/O ports, and above that are a few miscellaneous ports and status words that the 1K designers felt like memory-mapping.

It's nice to know your way around the memory map; however, you should rarely, if ever, refer to a memory address as a literal constant. Not only are literal addresses hard to read, but `bkasm` is simply better at memory allocation than you are. `bkasm` provides assembler directives to allocate memory* (see Section 5.7) and a linker to place that memory (see Chapter 6). These are good tools—use them. As for ports and status words, the standard 1K include file defines symbols for all of these (see Section 5.1). Using standard symbols allows you to share your code with others.

You have already seen examples of how to read from data memory:

```
a = [my_favorite_memory_location]        ; read from data memory
```

Given that, the syntax for writing to data memory is exactly what you'd expect:

```
[my_favorite_memory_location] = a        ; write to data memory
```

What you might not expect, however, is that the entirety of the 1K's memory-writing capability is represented by the above example. The only element that may be assigned to a memory location is the accumulator.

This seems easy enough. And if you are just learning the basics of 1K programming, it is easy enough, and that's all you need to know. But if you have some 1K experience, the line of code above might appear a little suspicious. The 1K does

*And it can even deallocate it when you're done with it.

not have a dedicated machine instruction for writing to memory. Instead, a store is done in parallel with an arithmetic operation. So a veteran 1K programmer might balk at seeing an isolated store—it cries out for an arithmetic instruction to ride on.

Piggybacking

`bkasm` doesn't balk when it comes to a memory write, but it doesn't immediately generate an instruction either. Instead, it waits for the next statement to see if it can *piggyback*. Any time you assign to an element other than the accumulator, `bkasm` tries to combine that assignment with a statement that does assign to the accumulator.

```
[sunken_treasure] = a
a += 4*b                    ; generates:  scba  4  sunken_treasure
```

If it cannot do this while maintaining the correct semantics of your program and respecting your precision preferences, it generates a dummy instruction to piggyback on.

```
[this_party_needs] = a      ; generates:  s1ac  0  this_party_needs
a = [rock_da_mic] + 2       ; generates:   1mc  2  rock_da_mic
```

`bkasm`'s automatic piggybacking has a number of advantages: it's intuitive and easy to read, it lets you write code quickly without worrying about manually combining operations, you can copy and paste code cleanly, and `bkasm` will sometimes notice combinations that you wouldn't. The disadvantage, especially from a traditional assembly perspective, is that one line of code no longer corresponds directly to one machine instruction. If you are tight on code space and need to be able to visualize the machine instructions for optimization, you might want to write the piggybacker and piggybackee on the same line to emphasize their oneness:

```
[wench] = a,  a = a*u + e   ; generates:  sauc  2.718  wench
```

If a store is followed by a literal machine instruction instead of an assignment statement, `bkasm` will not normally piggyback on it, since it assumes you know what you're doing. If you think that `bkasm` knows what you're doing better than you do, you can allow a piggyback by prefixing the instruction with an asterisk. (Think "wildcard".)

```
[picking_at] = a
*cab  4                     ; generates:  scab  4  picking_at
```

If you think that `bkasm` is responsible enough to piggyback on any and all machine instructions, you can tell it so with the `--piggyback-literal-instructions` option. With this option enabled, `bkasm` will effectively see stars in front of all of the instructions you write.

4.3.3 The b Register

The **b** register functions as a handy place to store data when you don't want to go through the hassle of writing to memory. It has a lower precision precedence than the accumulator, so it's useful in multiplications when the accumulator must keep its full 24-bit precision. The **b** register is also the main player in indirect addressing, as well as the input for the exponential and logarithmic functions.

There are two elements that you may assign to **b**: the accumulator and a memory location. The first is straightforward:

```
b = a
```

Again, there is no dedicated machine instruction for this operation. Instead, the statement requests a piggyback, much like the memory store statements above.

```
b = a
a -= 7                      ; generates:  x1ac  -7
```

In some cases, a memory store and an accumulator transfer can both piggyback on the same instruction:

```
b = a
[foo] = a
a = a^2 - 7                 ; generates:  sxaac  -7  foo
```

As with the store, a dummy instruction will be generated if no piggyback is possible.

```
b = a                       ; generates:  x1ac  0
a = b^2 + 1                 ; generates:   bbc  1
```

Syntactically, assigning a memory location to **b** is nothing surprising:

```
b = [box_o_chocolates]
a = a^2                     ; generates:  laac  0  box_o_chocolates
```

However, this particular operation is fundamentally different than the other two piggybacks mentioned so far. Because store and transfer piggybacks read from the accumulator, and every arithmetic instruction writes to the accumulator, these piggybacks must happen immediately. In order to maintain the correct semantics of the program, they can piggyback only on the subsequent statement. Anywhere else, they would grab the wrong value from the accumulator.

The load piggyback, however, does not care about the accumulator; its business is only between data memory and the **b** register. Neither of these elements are necessarily used on every instruction, which implies that `bkasm` might be able to slide your load request to somewhere else in the code while keeping your program functionally equivalent. If you let it, that's exactly what `bkasm` will do.

Optimized Load Placement

If the `--optimize-load-placement` option is not enabled, loads behave like stores and transfers; they either piggyback on the subsequent statement or generate a dummy instruction. But if you do enable this option (or the `-O` command line switch), `bkasm` will happily move the load piggyback up *or* down in the code in order to find the best possible home for it. As an example, here is a sine oscillator which you may remember from an earlier chapter:

```
; "modified coupled form" sine oscillator
 a = -0.1 * [sine]     ; generates:  lcm  -0.1  sine    <-----
 a += [cosine]         ; generates:  cma   1    cosine        |
 [cosine] = a          ; piggybacks below                     |
 b = [sine]            ; piggybacks above!  ------------------
 a = 0.1 * a + b       ; generates:  scab  0.1  cosine
 [sine] = a            ; generates:  s1ac  0    sine
```

`bkasm` doesn't just look for any piggybackable instruction; it takes into account the precision downgrade that some piggybacks cause, and tries to avoid that if it can.

```
 b = [call_a]
 a += p(24, 1/3)       ; generates:  1ac  0.3333
                       ;   (can't piggyback here; 24-bit precision required)
 a += 1/3              ; generates:  1ac  0.3333
                       ;   (can piggyback here, but would rather not)
 a += 1/4              ; generates: l1ac  0.25    call_a
                       ;   (best piggyback target; no sacrifice in precision)
 a = 2*a + b           ; generates:  cab  2
```

All else being equal, `bkasm` will go ahead and piggyback on the statement subsequent to the load statement, so you do get some measure of control as to where the load goes. If no piggyback is possible, a dummy instruction must be generated.

This optimization is done safely, of course; the load will never be moved past a statement that reads or writes the **b** register, or writes to the memory location that you are loading from, or otherwise performs an action that would affect the functionality of the code. You can't even fool `bkasm` with literal machine instructions, because `bkasm` knows what they mean too:

```
a += 1                ; can't move piggyback up here,
lindi  0x3ff  buffer  ;  because the "lindi" instruction uses the b register!
b = [buffest]         ; must generate dummy instruction:  l1ac  0  buffest
a += b
```

When in doubt, `bkasm` errs on the side of caution:

```
a += 1                ; generates:   1ac   1
if (a >= 0) positive  ; generates:  skip  !Z  1
a = 1 - a             ; generates:   cad  -1  1
positive:
b = [buffest]         ; generates:  l1ac   0  buffest
a += b                ; generates:   cab   1
```

Even though you can see that moving the load up to the "`a += 1`" statement is safe, `bkasm` isn't smart enough to do that* because it's too scared to look past the branch target.

This optimization is highly recommended. `bkasm` will frequently find load-friendly nooks and crannies that you simply wouldn't notice, and it can optimize across macro boundaries in ways that you couldn't possibly notice. More importantly, it allows you to write the statement that assigns to **b** right next to the statement that *uses* that value of **b**, which makes your code much easier to read and much less likely to break when you copy a section and paste it somewhere else. The disadvantage is that your optimized load may sometimes be difficult to spot in the listing file, since there's no telling where `bkasm` might have moved it to.

4.3.4 The u Register

The **u** pseudo-register is a funny animal. Funny enough that if you're a beginner, still struggling with the difference between S3.24 and S3.18 formats, you might want to consider skipping this section and pretending that **u** doesn't exist. Go on. We'll tell you about **u** when you're older.

Consider, for a moment, a microprocessor with a single-cycle multiplier. On every clock cycle, this multiplier latches in two inputs, feverishly multiplies them,

*Yet.

and produces an output. Now, suppose that on one particular clock cycle, the multiplier grows weary of life's burden and decides not to bother latching in one of its inputs. As a result, the other input ends up getting multiplied by whatever happened to be latched in on the previous clock cycle.

The designers of the 1K apparently thought that this sort of behavior would be a feature.* It appears in the programming model as a pseudo-register named **u**. Every arithmetic instruction implicitly sets **u** to the *multiplicand with the higher precision precedence*. This means, every time you assign to **a**, you are modifying **u** as well.

```
a = 2*b                 ; implicitly assigns:  u = b
a = a*b + 1             ; implicitly assigns:  u = a
a = a*[blorf] + 1       ; implicitly assigns:  u = [blorf]
```

u may then be multiplied by something else. Since **u** has the highest precision precedence, any statement that uses **u** ends up setting **u** to itself, effectively leaving it unmodified.

```
a = 2*b                 ; implicitly assigns:  u = b
a = a*u + 5             ; effectively:         a = a*b + 5
a = 5*u + b             ; effectively:         a = 5*b + b
```

`bkasm` allows you to assign to **u** directly. This functions as a strange sort of piggyback, and what exactly it means depends on the statement following the assignment.

After an assignment to **u**, if the subsequent statement actually *uses* **u**, then the effect is pretty much what you would expect from an assignment—an instruction is generated that copies the assigned value to **u** while leaving the other registers unmolested:

```
u = [blorf]             ; generates:   cma  0  blorf
                        ; u is now [blorf], but everything else is the same
a = a*u + 4             ; generates:  sauc  4  null_output
                        ; effectively computes:  a = a*[blorf] + 4
```

On the other hand, if the subsequent statement is an assignment to **a** that does *not* use **u**, then `bkasm` will assemble that statement in such a way that the *statement* effectively assigns the requested value to **u**. For example, this can resolve ambiguities with simple addition statements, where it isn't clear which addend is which:

```
        a = a + b       ; what is u?  who knows!
u = a,  a = a + b       ; sets u to a:   cab  1
u = b,  a = a + b       ; sets u to b:  scba  1  null_output
```

*Or at least, cheap and easy to implement.

Furthermore, if it is not possible to assemble the statement so that it gives you the **u** you want, `bkasm` generates an error. It is recommended that whenever you are planning to use the **u** output of a statement, you always prefix it with an explicit assignment to **u**:

```
u = a,  a = a^2 + 1     ; explicitly indicate that u is set to a
a = u*a + 1             ; effectively does a = (a*a + 1)*a + 1
```

This makes your code much easier to follow, and should you accidentally change the statement so it produces a different **u**, you'll get an error message instead of a bug:

```
u = a,  a = b^2 + 1     ; error: the statement cannot set u to a
a = u*a + 1
```

If the statement after the **u** assignment is not an arithmetic assignment at all, `bkasm` takes the first meaning, and generates an instruction that implements the **u** assignment.

Technically, the "dual" behavior of **u** assignments is no different than any other piggyback, which can either generate a instruction or modify the subsequent statement depending on what that statement looks like. But in practice, you use the two behaviors in different situations, so it helps to make a conceptual distinction. In fact, when you are aiming for the second type of behavior, it is actually encouraged to place the **u** assignment and its target statement on the same line of code, as shown above. This prevents the statement from getting detached from its predicate when you copy and paste, and makes it perfectly clear to your readers (and you) that the two phrases are part of the same sentence.

There aren't a whole lot of things you can assign to **u**, because there aren't a whole lot of things that the 1K uses as multiplicands. Here's what you get:

```
u = 0
u = a
u = b
u = [M]
```

u can be **u**seful. One trick takes advantage of its position as king of precision precedence. **u** always gets the 24-bit slot of the multiplier, so in order to reverse the normal precedence order, you can assign the lower-precedence element to **u**.

For example, a common scenario involves processing a signal and then scaling it by a "level" control. After you perform the processing, the signal will typically be sitting in the accumulator, while the level parameter will be stored in memory somewhere. But you might not want to simply multiply the accumulator by the memory location,

because memory has the higher precision precedence and your carefully calculated signal will get chopped down to S3.18. You can use a **u** assignment to shuffle around the multiplicands so they get the proper precisions:

```
; accumulator contains high-precision signal
 u = a,  a = [level]           ; u = signal, a = level
 a = a * u                     ; a = signal.24 * level.18
```

Similarly, some situations may call for multiplying the accumulator by **b**, but giving **b** the precision advantage. **u** makes that easy as well:

```
; b contains high-precision signal
 u = b                         ; u = signal
 a = a * u                     ; a = signal.24 * a.18
```

Another use for **u** is efficiently storing an array of S3.18 constants to memory, perhaps for later use as a lookup table:

```
              u = 0            ; generates:  andc  0xffffffff
              a = u*a + 1      ; generates:  sauc  1  null_output
[mem1] = a,   a = u*a + 2      ; generates:  sauc  2  mem1
[mem2] = a,   a = u*a + 3      ; generates:  sauc  3  mem2
[mem3] = a,   a = u*a + 4      ; generates:  sauc  4  mem3
[mem4] = a                     ; generates:  s1ac  0  mem4
```

There are some situations in which **u** can function somewhat as an actual register. After execution of an assignment that's prefixed with `u = a`, **u** will contain a value that is no longer available anywhere else—the contents of **a** *before* the assignment. With some clever coding, this can allow you to hang on to a value in the accumulator throughout a series of arithmetic operations, without using any additional storage:

```
a = [delta]                      ; put counter increment in a
u = a,  a = [count_up] + a       ; increment [count_up], ping delta to u
[count_up] = a,  a = u           ; store new [count_up], pong delta to a
u = a,  a = [count_down] - a     ; decrement [count_down], ping delta to u
[count_down] = a,  a = u         ; store new [count_down], pong delta to a
u = a,  a = [count_twice] + 2*a  ; increment [count_twice], ping delta to u
[count_twice] = a,  a = u        ; store new [count_twice], pong delta to a
```

What can **you** do with **u**?

4.4 Indirect Addressing

Sometimes you just don't know where you're going until you get there. And sometimes a program won't know which variable to use until it needs to use it. Indirect

addressing lets you read or write from a memory location whose address is determined at runtime.

All indirect addresses are indexed through the **b** register. This might lead you to guess that the syntax looks something like this:

```
a = [b]                         ; WRONG!
```

But that's not quite right. To get pedantic, "`b`", as we've been using it, refers to "the contents of the **b** register, interpreted as a S3.24 fixed-point number". It is unlikely that this would make a good index; if your indirect address could only vary over the range −8 to 8, it wouldn't be especially useful.

Instead, the 1K uses a couple of its integer representations to index the address. You were already introduced to one of them back on page 34:

```
a = [Ib]                        ; loads the value pointed to by Ib
```

The memory address that the above line refers to is an integer whose least significant bit is at bit 12 of the 28-bit **b** register. Everything that was said about the "I" representation in Section 4.3.1 still applies here. In particular, the `I()` function is handy for declaring constants in this representation:

```
a = I(0x123)                    ; " Ia = 0x123 " would have also worked
b = a                           ; move pointer to b register
a = [Ib]                        ; same as:  a = [0x123]
a = b + I(1)                    ; a = incremented pointer
b = a                           ; move pointer to b register
a = [Ib]                        ; same as:  a = [0x124]
```

The second integer representation has so far merely been hinted at, but now it gets its playing time:

```
a = [Fb]                        ; loads the value pointed to by Fb
```

The effective address above is an integer whose least significant bit is at bit 0 of the 28-bit **b** register.* The example above works just as well if you substitute "F"s for "I"s:

```
a = F(0x123)                    ; " Fa = 0x123 " would have also worked
b = a                           ; move pointer to b register
a = [Fb]                        ; same as:  a = [0x123]
a = b + F(1)                    ; a = incremented pointer
b = a                           ; move pointer to b register
a = [Fb]                        ; same as:  a = [0x124]
```

*In case you missed the duality: **Fb**'s low bit is aligned with the register's low bit. **Ib**'s high bit is aligned with the register's high bit (if you think of **Ib** as 16-bit).

If you bought the "millimeters" analogy proposed earlier for `I`, then you might like to think of `F` as "microns". The same idea, but smaller. Of course, the actual units for `I` and `F` are 2^{-12} and 2^{-24} respectively. But as long as you properly use the `I()` and `F()` functions to deal with the integer representations, you usually will not need to remember this.

Whether you use `Ib` or `Fb`, the syntax is the same. The discussion below will use `Ib` for the examples, but everything applies equally well to `Fb`.

Being able to dereference a pointer is nice, but usually you're more interested in indexing into an array. You can do this by adding an fixed offset to your index. Or if you want to think of it the other way around, you're adding an index to your fixed address. Whichever makes sense.

```
a = [buffer]                ; load from fixed address
a = [Ib]                    ; load from pointer
a = [buffer + Ib]           ; load by indexing into array
```

If you like to buffer circularly, you may apply a bitmask to your index. The index will be ANDed with the mask before being added to the fixed offset.

```
a = [ buffer + (Ib & 0x1f) ]  ; index is now modulo-32
```

The tricky part here is that the `&` operator has lower precedence than the `+` operator, so you need the parentheses to keep things straight. If you forget the parentheses (and you probably will), `bkasm` may gently remind you.

Indirect addressing isn't just for loading, of course—you may store indirectly as well. The syntax is exactly the same, except backwards:

```
[Ib] = a                    ; store to pointer
[buffer + Ib] = a           ; store into array
[buffer + (Ib & 0x1f)] = a  ; store into circular buffer
```

Only "sample memory", from addresses `0` to `0x3ff`, may be indirectly addressed. Thus, all indirect addressing is inherently modulo-1024. If you try to point past `0x3ff`, you'll wrap around.

Sometimes when people use `Ib` for indexing, it bothers them that there's twelve bits "underneath" the integer which are just going to waste. One way to put these bits to work is to use them for interpolating between values in the array. This might make sense if, for instance, you had a buffer of audio samples and wanted to connect the dots to approximate a continuous signal. In the example below, `ptr` can be interpreted as either an array index in `I` format which happens to have fractional bits, or a S3.24 number where 0.25 covers 1024 buffer samples. The important thing is that `ptr` has significant bits below `I(1)`.

```
; simple interpolation between two points in a buffer
 a = [ptr]                  ; a = pointer in I format
 a &= I(1) - F(1)           ; isolate fractional bits
 a *= 1/I(1)                ; scale up to S3.24 format (requires bigmult macro)
 [interp] = a               ; store interpolation fraction

 b = [ptr]                  ; b = pointer in I format
 a = [buf + Ib]             ; a = lower buffer data
 [lower] = a                ; store lower data
 a = [buf + 1 + Ib]         ; a = higher buffer data
 a -= [lower]               ; a = difference between higher and lower data
 b = [lower]                ; b = lower buffer data
 a = a * [interp] + b       ; a = lower + fraction*(higher - lower)
```

4.5 Conditionals

When you are processing a high-fidelity audio signal, it's great to have 24 bits past the decimal point to capture the subtleties. But sometimes you don't want all of these shades of grey; you want to ask the 1K a question and have it respond with "yes" or "no". That's what conditionals are all about.

4.5.1 1K-style Conditionals

At the machine instruction level, conditionals are evaluated with the 1K's `c` instruction. This instruction sets the accumulator to a particular value if a particular condition is true, or sets **a** to zero if not. For example,

```
c  Z  3.75
```

will test if the accumulator is Zero. If it is, it is set to 3.75; if not, it is set to zero. There are sixteen different conditions that 1K knows about:

```
c   T      3.75      ; always true  (effectively: a = 3.75)

c   Z      3.75      ; true if a == 0
c  !Z      3.75      ; true if a != 0
c   N      3.75      ; true if a <  0
c   N+Z    3.75      ; true if a <= 0
c  !N      3.75      ; true if a >= 0
c  !N!Z    3.75      ; true if a >  0

c  !V      3.75      ; true if no overflow
c  !VN     3.75      ; true if no overflow and a <  0
```

```
c  !V(N+Z)  3.75      ; true if no overflow and a <= 0
c  !V!N     3.75      ; true if no overflow and a >= 0
c  !V!N!Z   3.75      ; true if no overflow and a >  0

c   V       3.75      ; true if overflow
c   VN      3.75      ; true if overflow and a <  0
c   V!N     3.75      ; true if overflow and a >= 0
c   VZ      3.75      ; true if overflow and a == 0 (!)
```

You typically won't need to learn these condition codes, however, since `bkasm` provides a friendlier and more familiar syntax for specifying conditionals.

4.5.2 C-style Conditionals

The following line means the same to `bkasm` as it does to a C compiler:

```
a = (a < 0)
```

It tests whether the **a** register is negative. If it is, **a** is set to one; otherwise, it is set to zero. `bkasm` lets you compare **a** to zero in every way imaginable:*

```
a = (a == 0)
a = (a != 0)
a = (a <  0)
a = (a <= 0)
a = (a >= 0)
a = (a >  0)
```

The following line also means the same to `bkasm` as it does to a C compiler, although it's possibly more common an idiom in BASIC than in C:

```
a = 3.75 * (a < 0)
```

If **a** is negative, a is set to 3.75; otherwise, it is set to zero. Any conditional phrase may be multiplied by a constant.

That's about all that the 1K can do in a single instruction. But if you are willing to spend an extra instruction, `bkasm` will let you compare just about anything with anything else:

*As you might guess from standard operator precedence, the parentheses are not strictly necessary. But just because `bkasm` can read these statements without parentheses doesn't mean that you can. While we're at it, don't try merging the comparison operators with the assignment operator. "`a = a < 0`" is a valid statement, but "`a <= 0`", of course, means something else entirely. And don't even think of writing "`a = (a == 0)`" as "`a === 0`".

```
a = ( a != b )
a = ( a > 3*[monkey] )
a = ( b == [lemur]/2 )
a = ( [orangutan] <= 3.75 )
```

Well, not exactly anything. The 1K must have an instruction that can subtract the two terms in the comparison, although `bkasm` is smart enough to try doing the comparison backwards if forwards doesn't work. Of course, you may multiply these conditions by constants as well:

```
a = 7 * ( [gorilla] == 1/2 )
```

Compound Conditions

The following line (you guessed it) means the same to `bkasm` as it does to a C compiler:

```
a = (a > 2) && (b < 4)
```

If both clauses of the condition are true, **a** is set to one; otherwise, it is set to zero. You can make your conditions as complicated as you like:

```
a = ( ((a > 2) && (b < 4)) || ([monkey] < 3.75) ||   \
                              (b == [lemur]/2)     ) && ([gorilla] == 1/2)
```

But there's a catch. Since the accumulator is used for evaluating these conditions, the original value of the **a** register is clobbered as soon as you get past the first comparison. The syntax makes it tempting to write statements such as:

```
a = (a < -2) || (a > 2)        ; WRONG!
```

But there's no way to evaluate such a thing without using another register. Thus, you are only allowed to refer to the **a** register in the very first clause. Anywhere after that, `bkasm` will complain.

The syntax is even more general than this. The following line will make C programmers' eyes glaze over, although Perl coders will have no problem with it:

```
a = 3.75 * ([gibbon] > 0.7)  ||  4.75 * ([ape] > 0.8)
```

In C, you would write this as:

```
if      (gibbon > 0.7)  { a = 3.75; }
else if (ape    > 0.8)  { a = 4.75; }
else                    { a = 0;    }
```

That's because C's logical operators only know how to return zero or one. Perl's logical operators, which inspired this particular bit of syntax, return the result of the last expression they evaluate. The OR operator doesn't bother evaluating its right side if its left side is already true, so a chain of ORed expressions effectively returns the first one to come up non-zero. With `bkasm`, this can be very useful for mapping conditions to numbers:

```
a = 1 * ([monkey]  > 0) ||    \
    2 * ([lemur]   > 0) ||    \
    3 * ([gorilla] > 0) ||    \
    4 * ([gibbon]  > 0) ||    \
    5 * ([ape]     > 0)
```

Multiplication distributes, as it probably should, but it looks pretty weird in doing so. The following two lines are equivalent:

```
a = 1.5 * ( 2 * (a > 0)  ||  3   * (b > 0) )
a =         3 * (a > 0)  ||  4.5 * (b > 0)
```

Overflow

The S3.24 format can only represent numbers in the range $-8 \leq x < 8$. If the result of an arithmetic instruction tries to escape from this box, it gets squashed against the floor or the ceiling, and ends up as -8 or $8 - F(1)$ respectively. But the 1K doesn't leave you in the dark about this; the squashing sets the processor's *overflow flag*, which may be examined on the subsequent instruction.

You tell `bkasm` to check for overflow by comparing to ± 8. Essentially, your conditional verifies that a result is in the proper range:

```
a = [rain] + SPAIN              ; arithmetic instruction
a = (a <  -8) || (a >= 8)       ; a = 1 if overflow occurred

a = [main] + PLAIN              ; arithmetic instruction
a = (a >= -8) && (a <  8)       ; a = 1 if overflow did not occur
```

If you are a notational purist, this syntax may horrify you, since it involves testing for something that *cannot possibly be true*. But that's really just a matter of perception. One way to think about it is, in the inequalities above, **a** represents the result of the

previous arithmetic operation *before saturation*. You pretend that clipping doesn't actually take place until after the comparison. This sort of makes sense, if you let it.

You may also check specific ranges between zero and the boundaries, or verify the direction of your overflow. Here are the overflow-related conditions that you get:

```
a = (a >= -8) && (a <  8)   ; true if no overflow

a = (a >= -8) && (a <  0)   ; true if no overflow and a <  0
a = (a >= -8) && (a <= 0)   ; true if no overflow and a <= 0
a = (a >=  0) && (a <  8)   ; true if no overflow and a >= 0
a = (a >   0) && (a <  8)   ; true if no overflow and a >  0

a = (a <  -8) || (a >= 8)   ; true if either overflow
a = (a <  -8)               ; true if negative overflow
a =              (a >= 8)   ; true if positive overflow
```

The last two are the ones you are most likely to use, and they also happen to be simple one-clausers:

```
a = [ptr] + I(1)            ; test whether we can increment the pointer
a = (a >= 8)                ; would it overflow?
[reached_end_of_buffer] = a ; if so, make a note of that
```

Unfortunately, there is no inverse for those two; you cannot ask if a result "did *not* positively overflow". In practice, you rarely need to actually know this, because a simple "did not overflow" will suffice. However, in these situations, it can be convenient to simply be able to write `(a < 8)` instead of cluttering up your conditional with a full "did not overflow" test. The `--allow-one-sided-overflow` option allows you to do this:

```
.option allow-one-sided-overflow
a = [ptr] + I(1)            ; test whether we can increment the pointer
a = (a < 8)                 ; is it still in valid range?
[not_at_end_of_buffer] = a  ; if so, make a note of that
```

This is merely a syntactic shortcut for when you're only expecting overflow from a particular direction. It really is a full `(a >= 8) && (a < 8)` in disguise.*

*Conditional branches, on the other hand, can actually handle a genuine ($a < 8$). See Section 4.6.

Efficiency

When it comes to implementing compound conditions, there are various tricks that can save an instruction or two in special cases. `bkasm` does not (currently) know any of these—it treats every compound condition as a general case and doesn't try to optimize. Thus, if *you* try to optimize, you can often come up with a clever solution that beats `bkasm`'s.

For example, consider a positive-going zero-crossing detector, such as you might use in a simple frequency-detection algorithm. The most natural way to write this is:

```
a = ( [previous_sample] < 0 )  &&  ( [this_sample] >= 0 )
```

`bkasm`'s implementation takes five instructions:

```
cm    1  previous_sample
c     N  1
skip  Z  2
cm    1  this_sample
c    !N  1
```

But a more efficient solution can be found by rearranging things and letting the multiplier come out and play:

```
a = ( [this_sample] >= 0 )
a = ( a*[previous_sample] < 0 )
```

This only takes four instructions:

```
cm    1  this_sample
c    !N  1
amc   0  previous_sample
c     N  1
```

But is that one instruction really worth the time it took to design the clever solution, or the time it will take to reevaluate and redesign it when it needs to change later on? If you individually optimize your compound conditions, you might write faster code, but you'll never write code faster. Letting `bkasm` handle your compound conditions results in code that is easy to write, easy to read, and guaranteed to be correct. Especially during the initial stages of implementation, these properties are often more valuable than squeezing out an extra instruction or two. You can always go back and optimize when you hit an code size crunch.

4.6 Flow Control

In order for a computational system to be Turing complete, it must offer some means of executing code conditionally. Although the 1K clearly makes no claims to being complete in any sense, it does in fact provide a conditional branch operation.

This doesn't mean that you get loops or subroutines, though. The 1K's program counter moves forward inexorably. You can branch to the future, but you can't come back.

4.6.1 1K-style Flow Control

At the machine instruction level, branches are implemented with the `skip` instruction. This allows you to conditionally skip over a sequence of instructions that you don't want executed. The skipping is done in space, not time; the 1K always spends one clock cycle on each of its 1024 instructions, whether the instruction is skipped or not.

The `skip` instruction takes two operands. The first is the condition, and it uses exactly the same condition codes that you saw on page 47. The second operand indicates where to skip to.

```
skip  !N!Z  2          ; skip two instructions if (a > 0)
```

The first operand may be omitted for an unconditional branch:

```
skip        2          ; skip two instructions, no matter what
```

For backwards compatibility, the second operand of the `skip` instruction can actually mean two different things. If the operand is a simple number, as above, then it represents how many subsequent instructions should be skipped. `skip 0` is effectively a "no-op" and simply continues on to the next instruction; you start counting from there.

If the second operand is not a simple number, then it is interpreted as the address in instruction memory to skip to. This is normally a label, although it could be any symbol or expression that resolves to a valid address. Section 5.6 is all about labels.

```
 skip  !Z    wall       ; if a is non-zero, skip to ---
 cm     1    humpty     ;                              |
 s1ac   0    dumpty     ;                              |
wall:                   ; here <-----------------------
```

Except in extraordinary circumstances, you should *always* skip to labels. Skipping to an expression or over a literal number of instructions implies that you know exactly how many instructions `bkasm` is going to generate in a given location, and due to piggybacking and load propagation, that's a prediction you're better off not making. More importantly, `bkasm` likes to know where all of the branch targets are so it can avoid piggybacking or optimizing across them. If you provide a literal skip count, `bkasm` can't know exactly which statement you're skipping to, so it may generate incorrect code. Unless you are writing entirely in machine instructions, *always skip to labels.*

As with conditionals, `bkasm` provides a friendlier syntax than the raw machine instruction, so you're probably better off without `skip` in any case.

4.6.2 Asm-style Flow Control

If you come from an assembly language background, you may be most comfortable with `bkasm`'s assembly-style branching statements. These are similar to instructions offered by traditional microprocessors.

```
jmp  label   ; "jump".                              Same as: skip  T   label
bra  label   ; "branch always".                     Same as: skip  T   label
goto label   ; "go to".                             Same as: skip  T   label
beq  label   ; "branch if equal to zero".           Same as: skip  Z   label
bne  label   ; "branch if not equal to zero".       Same as: skip !Z   label
bpl  label   ; "branch if plus".                    Same as: skip !N   label
bmi  label   ; "branch if minus".                   Same as: skip  N   label
blt  label   ; "branch if less than".               Same as: skip  N   label
ble  label   ; "branch if less than or equal".      Same as: skip  N+Z label
bge  label   ; "branch if greater than or equal".   Same as: skip !N   label
bgt  label   ; "branch if greater than".            Same as: skip !N!Z label
bvc  label   ; "branch if overflow clear".          Same as: skip !V   label
bvs  label   ; "branch if overflow set".            Same as: skip  V   label
```

The ones that imply comparison can be used after a subtraction like so:

```
a = b - [acon]
blt  sandwich      ; branch to "sandwich" if (b < [acon])
```

4.6.3 C-style Flow Control

The preferred syntax for flow control follows that used by many popular high-level languages, such as C, Perl, BASIC, shell, and English. It's simply the familiar *if-then-else* construct. There are actually two variations on this syntax, one for

those who like labels and one that is label-free and more "structured". If you are inclined toward traditional assembly coding, then you'll probably feel more comfortable sticking with labels. But labels are considered somewhat passé in modern high-level programming, so if you come from that background, structured flow control is for you. Use whichever suits your style, or mix and match to taste.

If labels are your thing, this is how you get around:

```
if (a != 0) wall       ; if a is non-zero, skip to ---
a = [humpty]           ;                             |
[dumpty] = a           ;                             |
wall:                  ; here <-----------------------
```

Following the `if` keyword is an algebraic condition in parentheses, followed by a label to branch to if the condition is true. It's very similar to C, if you pretend there's a `goto` between the condition and the label.

If you prefer the more structured approach, then follow your condition with an opening curly brace instead of a label:

```
if (a == 0) {          ; if a is non-zero, skip to ---
  a = [humpty]         ;                             |
  [dumpty] = a         ;                             |
}                      ; here <-----------------------
```

The section of code between the opening and closing braces above is called a *block*. Blocks have a number of wonderful uses, such as defining the scope for local variables, symbols, and options (see Section 5.8), but here, they are being used to delimit conditionally-executable sequences of statements. Assembly is not a freeform language, so brace placement is less flexible than in C. The opening brace must be on the same line as the `if` keyword, and the closing brace must be near the start of a line. (The closing brace may be placed after labels, but not after statements.)

You get `else` as well. Like Perl, `bkasm` spells "else if" as `elsif`, and the syntax is the same as for the initial `if`:

```
if (a < 0) {
  a = [kumquat]
}
elsif (a > 0) {
  a = [lytchee]
}
else {
  a = [kiwi]
}
```

You may place the keywords on the same line as the closing braces, if that's more your style:

```
if (a < 0) {
  a = [kumquat]
} elsif (a > 0) {
  a = [lytchee]
} else {
  a = [kiwi]
}
```

The labeled syntax works with `elsif` as well. You can even combine the two syntax variations within a single structure, if you come up with a good reason for doing so:

```
if (a < 0) {
  a = [kumquat]
}
elsif (a > 0) getLytcheeFromChina
else {
  a = [kiwi]
}
```

You may nest blocks as deeply as you like. If you do, of course, be sure to use the proper indentation so you don't get lost:

```
if (a < 0) {
  a = [kumquat]
  if (a == 0) {
    a = [rhubarb]
  }
  else {
    a = [date]
  }
}
```

The conditions of your `if` and `elsif` statements may be any of the comparisons to zero on page 48 or overflow conditions on page 51:

```
if (a >= 8) {               ; positive overflow
  a = [mango]
}
elsif (a < -8) {            ; negative overflow
  a = [pomegranate]
}
elsif (a >= -8 && a < 0) {  ; negative, but no overflow
  a = [papaya]
}
```

Frankly, however, comparing to zero and testing for overflow are boring. But that's all that the 1K can do without clobbering the accumulator. And unlike conditional evaluation in the previous section, an `if` statement certainly does not *look* like an assignment to anything, and it would be unusual and unexpected if it went around messing with your registers. So, `bkasm` doesn't let it.

Destructive Conditions

Unless you want it to, of course. If you think you can deal with `if`s that modify the **a** register, then the `--allow-destructive-if` option is your ticket to conditionals that are limited only by your imagination.*

They're also limited by the restrictions in the previous section, although there is a bit more leniency. A clause in a compound condition still can't refer to the **a** register after it has been clobbered by a previous clause, but unlike before, comparing to zero or testing for overflow doesn't actually clobber anything. So `bkasm` will actually let you refer to the accumulator multiple times, as long as you put the "free" ones first:

```
.option allow-destructive-if
if ( (a < 0 || a >= 8) && (a != [guava]) && (b != [persimmon]) ) {
  a = [gooseberry]
}
```

In the example above, the accumulator is not actually clobbered until the `(a != [guava])` clause.

Destructive `if`s are highly recommended. Evaluating complex conditions and maintaining structured flow control are two of the most notoriously difficult tasks that assembly programmers have to deal with, which is why `bkasm` lets you step things up a level and handle them almost as if you were using C. But be careful! *You should never use the value of the accumulator immediately after a destructive if.* The familiarity of the syntax makes this sort of mistake all too easy:

```
a = [fig]
if (b == [apple]) {
  a = a + NEWTON            ; WRONG!  a is not what it appears to be!
}
```

Even if you think you know what the conditional is going to leave in the accumulator, you still should not rely on it, because `bkasm` might decide to evaluate a comparison

*As with all options that affect the semantics of the language, you're better off specifying this as a `.option` in your source file rather than on the command line. That way, future borrowers of your code won't need to know some specific command line incantation in order to make it assemble.

backwards. If you want to use the value of the accumulator after a comparison, you are much better off writing out the subtraction explicitly and following it with a non-destructive condition. This is both more reliable and more readable:

```
a = [currant] - MAX_AMPERAGE
if (a > 0) {                ; effectively, test if [currant] > MAX_AMPERAGE
  [currant_overflow] = a    ; but be able to use the difference here
  goto tripCitrisBreaker
}
```

Another pitfall to watch out for is using a clobbered accumulator in an `elsif` condition. This is especially insidious because the `if` and `elsif` can be textually spaced far apart, which makes it easy to overlook their connection:

```
a = [berry]
if (a == STRAWBERRY) {
  ...
}
elsif (a == RASPBERRY) {    ; WRONG!  a is not what it appears to be!
  ...
}
```

`bkasm` is (currently) not smart enough to catch this mistake, so hopefully you'll be smart enough not to make it.

Break

`bkasm` offers one more flow control mechanism, for when you've started out structured but need to bend the rules. `break` is similar to its namesake in C, although it is more closely related to the `last` statement in Perl.

`bkasm` lets you use `break` either as a symbol or a keyword. As a symbol, it refers to the end of the innermost enclosing block. This is especially useful as the label after an `if`, where it sort of looks like a statement:

```
if (a == 0) {
  a = [nuts] + WALNUT + CHESTNUT + HAZELNUT
  [nuts] = a
  if ([fruit] == 0) break     ; if [fruit] == 0, skips to ---
  a = [nuts] + COCONUT        ;                              |
  [nuts] = a                  ;                              |
}                             ; here <-----------------------
```

The broken block does not necessarily have to be part of an `if` statement. It's perfectly acceptable to lay down a bare block for the express purpose of breaking out of it:

```
{                              ; start a bare block
  a = [nuts] + WALNUT + CHESTNUT + HAZELNUT
  [nuts] = a
  if ([fruit] == 0) break      ; if [fruit] == 0, skips to ---
  a = [nuts] + COCONUT         ;                              |
  [nuts] = a                   ;                              |
}                              ; here <-----------------------
```

This is a convenient way to jump around without having to resort to labels. It also tends to be more readable as long as you pay attention to indentation, and it's easy to copy and paste without worrying about the uniqueness or locality of your labels.

When you write `break` by itself, as a statement, `bkasm` pretends that you said `skip break`, which has the effect of jumping you out of the innermost enclosing block. This can be used in conjunction with labels to emulate a switch-like construct:

```
{
 summer:  a = APRICOT + PEACH + MELON,  break
 autumn:  a = GRAPE + KIWI + FIG,       break
 winter:  a = APPLE + PEAR,             break
 spring:  a = LOQUAT
}
```

The above example assumes that other parts of the code jump to the seasonal labels. The `break` statement allows you to exit the block without having to explicitly label the end.

As described in Section 5.6, `bkasm` offers some fairly sophisticated labeling features. And if you come from a "braces and blocks" background, you might never use them at all. `if`, `elsif`, and `break` can get you almost everywhere you need to go, and you can probably get by without ever laying down a single label. On the other hand, if you prefer jumping to labels, there's no need to use blocks for anything. Go with whichever style you're most comfortable with.

Chapter 5

The Assembler

`bkasm` is a retargetable assembler. This means that `bkasm` would be happy to assemble code for any processor that you care to describe to it. All processor-specific functionality is boxed up into a self-contained module which `bkasm` loads in when it sees the `-T` command line switch (or when it doesn't). This module is called a *target module*, or more informally, a *target*.

This basically means that the previous chapter was one big lie. Almost nothing in the last chapter involved `bkasm` proper; instead, you were reading about how to write for the *1k target*. Without this target module, `bkasm` wouldn't know what **a** meant, let alone how to assign to it.

The target may be the brains of the assembler, but `bkasm` provides some hefty brawn to back it up. The target relies on `bkasm` to deal with expressions, symbols, and labels, as well as memory allocation for data. The target doesn't even know about conditional and iterative assembly or macros; these all happen behind the target's back. This chapter will cover all of these target-independent amenities, and much more.

Learn your target, but learn your assembler as well! `bkasm` provides quite a few features to make the code-writing process quicker, cleaner, and more enjoyable, but you have to ask for them.

5.1 Basics

Comments

Despite the algebraic syntax of the *1k* target, `bkasm` thinks of itself as an assembler, and therefore recognizes the traditional assembler comment character—the semi-colon. `bkasm` ignores everything from a semi-colon to the end of the line.*

```
; bkasm will never see this.  Hey, bkasm!  You suck!
```

There are no multi-line comments, as such, although an `.if 0` / `.endif` pair will usually work in a pinch. However, a line of semi-colons down the left column of the file has a certain visual distinction that makes it clear that you're reading something that's not intended to run. If your text editor cannot comment and uncomment regions automatically, you need a better text editor.

Directives

Much of what you write is intended to represent executable code, and these statements get passed on the target. But sometimes you want to talk to `bkasm` directly, and you do this through assembler *directives*.

When it comes to directive syntax, `bkasm` is a bit more schizophrenic than it is with comments. For various compatibility and convenience reasons, `bkasm` recognizes two different directive styles. Traditionally, assembly directives begin with a dot:

```
.eat OATS               ; assembly code
```

The C preprocessor, however, looks for directives that begin with a pound sign:

```
#eat OATS               /* C code */
```

Some traditional assembly directives take a subject *before* the directive itself:

```
MARES  .eat  OATS       ; assembly code
```

whereas directives in C must be at the start of the line, so the subject and object both follow the directive:

*Usually. The exceptions are within `.perl` and `.perlmacro` blocks, which follow Perl's rules. See Section 5.14.

```
#eat  MARES  OATS       /* C code */
```

Any of `bkasm`'s directives can be written in either style, and then some. `bkasm` never cares about leading whitespace on a line, so you may indent your directives however you like. With assembly-style directives, you can put the first argument either before or after the directive. These lines all mean the same thing:

```
#eat   MARES  OATES            ; c-style
       #eat   MARES  OATES      ; c-style, indented
MARES  .eat   OATS             ; asm-style, subject first
       .eat   MARES  OATS      ; asm-style, directive first
```

Assembly-style directives are recommended, since `bkasm` is, after all, an assembler, and writing your directives as *subject, verb, object* when the directive is a verb is recommended as well, since that's how we structure sentences in English. But if your fingers naturally reach for the pound sign when you think about directives, that's okay with `bkasm`.

One syntactic construct supported by many assemblers that is *not* okay with `bkasm` is the *pseudo-op*, which is a directive that looks like an assembly instruction:

```
SYMBOL  set  VALUE       ; WRONG in bkasm!
DATA    ds   1           ; WRONG in bkasm!
```

`bkasm` requires all directives to be marked with a dot or a pound sign.* Enforcing separate namespaces for directives and instructions is essential for forward compatibility, especially with a retargetable assembler, and it also allows you to visually distinguish the code from the metadata. A directive will never generate executable code—it can only indicate what you'd like done with the code you do write. It's good to keep the concepts distinct.

`bkasm` provides a lot of interesting directives, most of which will be described throughout this chapter. You may also use Appendix C as a quick reference. Targets are allowed to define additional directives as well. The *1k* target doesn't define anything worth mentioning, but the linker knows about plenty of extra directives, and they will be covered in Chapter 6.

5.1.1 The `.option` Directive

Directive: `.option` *NAME=VALUE*

*For compatibility with `1kasm`, there actually are a few directives which can go dotless. But don't do that.

The `.option` directive is one of the most important ones, which is why it was briefly introduced back on page 19. This directive lets you set named parameters that influence `bkasm`'s behavior. If you do not provide a value, the parameter is implicitly set to 1, which enables the more Boolean-oriented options. Setting an option to 0 will typically disable it.

```
.option  fix-all-bugs                    ; same as:  .option fix-all-bugs=1
.option  bug-fixing-algorithm=achieve-sentience
.option  taunt-user-when-bug-found=0     ; disable option
```

Specifying an option in the source is, in most respects, similar to specifying it on the command line:

```
% bkasm --fix-all-bugs myfile.asm
```

The main difference is that options given on the command line are active throughout all source files, whereas an `.option` directive is forgotten about when its file ends. This prevents one file's personal configuration from leaking over into another file that was expecting a default setup.

If you actually want a particular file's options to be remembered across all source files, you can use the strange but occasionally handy `--remember-options` option. When given on the command line, no options are ever forgotten—each source file inherits the options set by all previous source files.

```
% bkasm --remember-options onefish.asm twofish.asm redfish.asm
```

In the above example, any options set in `onefish.asm` would stay in effect all the way past `redfish.asm` and even into the linker script,* unless changed somewhere along the way.

When this option is given not on the command line, but as an `.option` directive, all other `.option` directives in the file will be remembered *except* for the `.option remember-options` option itself. This prevents subsequent source files from inadvertently making their own options memorable.

```
;;; onefish.asm:
 .option  fix-all-bugs
 .option  remember-options

% bkasm onefish.asm twofish.asm redfish.asm
```

In the above example, `--fix-all-bugs` would be in effect through all source files, but any options set by `twofish.asm` would be forgotten as soon as `twofish.asm` ended.

*Presumably named `bluefish.ld`.

5.1.2 The `.include` Directive

Directive: `.include` *FILENAME*

The `.include` directive tells `bkasm` to put the current source file on the back burner and start assembling another file in its place. The effect is almost as if the contents of the included file had been grafted into the main source at the point of the directive. The primary functional difference is that individual files must tidy up their control directives (Sections 5.9 and 5.10); the included file can't open an `.if` that is answered by an `.endif` in the main file, or vice versa. Such a situation usually indicates a mistake, so `bkasm` catches it.

If you specify an unqualified filename, bare or within quotes, `bkasm` will look for it in the "current directory", wherever that is. It typically is the same place that the assembly file itself is located.

```
.include  my_favorite_definitions.inc
.include "my_favorite_definitions.inc"
```

If the filename is given in pointy brackets, `bkasm` will expend a little more effort in trying to find it. If it's not in the current directory, `bkasm` will look through each directory in the *include path*. Directories are added to the include path through the `--include-path=`*`DIRECTORY`* list option or the `-I` command line switch (see page 20). `bkasm` tries these directories in reverse order, so more recently added directories get priority.

```
.option include-path=/usr/include/
.option include-path=/usr/local/include    ; trailing slash doesn't matter
.include <my_favorite_definitions.inc>
   ;  looks for:  ./my_favorite_definitions.inc
   ;       then:  /usr/local/include/my_favorite_definitions.inc
   ;       then:  /usr/include/my_favorite_definitions.inc
```

By default, included files show up in your listing file. However, you typically use includes just for symbol and macro definitions, and you may not want these cluttering up your listing. Setting `--list-included-files=0` will clean up this situation. More information about listing files and related options can be found in Chapter 7.

In higher-level languages, it is somewhat of an unwritten law* that an include file should never contain bare executable code, variable or table declarations, or anything else that takes up space in an object file. Except in unusual situations, every source file in a project should be able to include every include file without

*Well, unwritten except for here. And in dozens of books on good programming practice.

any ill effects. Historically, assembly-level projects have sometimes needed to bend this rule as a way around a toolset's poor (or non) support for linking and macros.* `bkasm` can link and macro with the best of them, so it's probably a good idea to reserve include files for definitions, and keep the instantiations in the main source.

5.1.3 The `.error` and `.warn` Directives

Directive: `.error` *MESSAGE*

The `.error` directive generates an error that looks and behaves a lot like one of `bkasm`'s internal errors. You would typically use it in a macro to indicate that a parameter of some sort is incorrect, although it can also be used for basic sanity-checking:

```
BUFFER_LENGTH  .equ  someComplicatedFormula()
.if BUFFER_LENGTH > MAX_LENGTH
   .error Buffer is too big!
.endif
```

Directive: `.warn` *MESSAGE*

The `.warn` directive generates a warning that looks and behaves a lot like one of `bkasm`'s internal warnings. This also is useful for general debugging, sanity-checking, and catching inefficiencies:

```
DELTA_DENTAL  .equ  someComplicatedFormula()
a = [dental] + DELTA_DENTAL        ; increment [dental] by a delta value
[dental] = a                       ; and store it back
.if DELTA_DENTAL == 0              ; warn if those were wasted instructions
   .warn This code has no effect.
.endif
```

5.2 Expressions

When you take numbers, symbols, operators, and functions, and mix them all together, you get an expression. Expressions are used throughout `bkasm`. The right side of a *1k* assignment statement is an expression, as are the two operands following a *1k* machine instruction. Many of the directives that you will learn about in this chapter will expect expressions as arguments. Most expressions are expected to (eventually) resolve to a simple number, although some expressions, like *1k* assignments, merely need to resolve to a form.

*Especially in the embedded systems field, where the only tools available for many specialized processors are abominable vendor-supplied assemblers written by hardware engineers.

5.2.1 Operators and Functions

`bkasm` uses Perl to evaluate all expressions, which means that you have unfettered access to all of the operators and functions that you know and love from Perl. If you are unfamiliar with Perl, then you can simply stick with the operators that you know and tolerate from C. Here are some of the more relevant of these operators, ordered by precedence:

`**  ^`	Exponentiation
`!`	Logical NOT
`*  /  %`	Multiplication, division, and remainder
`+  -`	Addition and subtraction
`<  >  <=  >=`	Inequality tests
`==  !=`	Equality tests
`&`	Bitwise AND
`|`	Bitwise OR
`&&`	Logical AND
`||`	Logical OR

The precedences and associativities are the same as in C. As expected, you may use parentheses to subvert the default precedence levels:

```
a = 2 * 3 + 4        ; same as:  a = 6 + 4
a = 2 *(3 + 4)       ; same as:  a = 2 * 7
```

`bkasm` also defines an expansive set of trig and other math functions. You can find these listed in Appendix E, and you'll probably recognize most of them from Matlab's `elfun` package, if that's where your background lies.* In general, they are spelled just like you'd expect, and there are synonyms for those cases in which you're not sure what to expect:

```
a = sin(pi/4)        ; same as:  a = 1/sqrt(2)
a = cot(pi/4)        ; same as:  a = 1
a = cotan(pi/4)      ; synonymous with previous line
```

Because all expressions are evaluated as Perl code, you have unrestricted access to Perl's entire vocabulary. Anything that Perl can do, an expression can do, even if you might not want it to. For example:

```
oopsies  .equ  3 + unlink <*>    ; DON'T TRY THIS AT HOME
```

*If your background lies with Perl, you'll recognize most of them from the standard `Math::Complex` module.

The preceding line will set the `oopsies` symbol to three plus the number of files that used to be in your directory before `bkasm` went and deleted all of them. Although it makes for great April Fool's pranks, this sort of thing is not exactly encouraged. If you want to invoke a Perl function for its side effects rather than its return value, you should probably use one of the directives in Section 5.14, such as `.eval` or `.perl`.

5.2.2 Complex Math

All of the mathematical operators and functions work with double-precision floating point numbers, which is mentioned only because most assemblers only evaluate integer expressions.* Perhaps more surprisingly, all math operators and functions are fully capable of accepting and returning complex numbers. `sqrt(-7)` and `acos(2)` are not errors—they return a perfectly usable, if not quite real, result.

```
a = Re( acos(2) )    ; same as:  a = 0
a = Im( acos(2) )    ; same as:  a = log(2 + sqrt(3))
```

If you want to start off with a complex number, you can use either of the synonymous `i` or `j` symbols:

```
a = i * i            ; same as:  a = -1
a = e^(pi * j)       ; same as:  a = -1
```

The general rule is that an expression is allowed to bounce around the complex plane as much as it likes during evaluation, but it ought to resolve to a real number by the time it is used as a constant in a machine instruction.† If `bkasm` is expecting a real number and finds something that isn't real enough, it will issue a warning and use the real part.‡

This may all sound like overkill. It's not. You can't open any textbook on digital signal processing without encountering a pageful of js (or is, depending on the persuasions of the authors). At some point long ago, a mathematician noticed that the cyclicity of exponentiated imaginary numbers could be used to represent periodic signals, and once that model caught on, there was no turning back. Within the modern framework, the design of any significant signal processing system involves complex manipulation.

*Most assemblers can't take a hyperbolic arc cosecant, either.

†At least until Alesis invents the (1+i)K DSP.

‡This questionable behavior is borrowed from Matlab. Questionable because in Matlab, taking the modulus would typically make a lot more sense than the real part. But `bkasm` tries to follow precedents whenever possible (except when it doesn't).

The *implementation* of such a system, on the other hand, can often be completely real, which leads to the prevalent DSP programming paradigm of designing a system in Matlab, getting back a list of coefficients, and then plugging them into a template of real-valued fixed-point DSP code. This is awkward. It requires writing code in two languages and running back and forth between two applications. It is easy to lose track of which version of Matlab code corresponds to which DSP iteration. Frequently, the Matlab script gets lost when its job is done, leaving future borrowers and maintainers of the DSP code with an opaque and untweakable list of coefficients.

The purpose of `bkasm`'s mathematical sophistication is not to show off Perl's seamless support for complex numbers,* but to tear down the artificial barrier between design and implementation languages, and give you the power to design a DSP system entirely within the assembler. If the Matlab detour can be reduced or eliminated, then not only will design code never get separated from implementation code, but iterations of the *tweak-compile-test* cycle can be made overwhelmingly faster. Instead of opening a Matlab script, changing a parameter, running the script, copying a coefficient table, pasting it into a DSP file, and assembling it, you simply change the parameter in the assembly file, reassemble, and you're good to go. When taken to its logical conclusion and combined with some of `bkasm`'s other features, this concept can completely change the playing field. Programming and experimentation can be a much different experience if you happen to have, at your fingertips, a library of generalized, parameterizable macros capable of transparently turning out ready-to-run DSP code for any occasion. `bkasm` makes it possible, and you'll see some examples in later chapters.

5.3 Symbols

Symbols have no direct analog among the lower-level high-level languages such as C, which is unfortunate because they are extremely handy. In C, their ecological niche tends to be filled (poorly) by constant preprocessor macros, but because these macros are blind text substitutions and they are not syntax-checked at definition time, they represent a notorious source of programming errors. Symbols are a much cleaner approach to named constants and compile-time variables, and it is strange that they have historically remained the domain of assembly languages.

5.3.1 The `.equ`, `.set`, and `.unequ` Directives

Directive: *`SYMBOL`* `.equ` *`EXPRESSION`*

*Well... maybe a little.

The `.equ` directive is the typical way to create a symbol:

```
ANSWER  .equ  42
```

After the above statement, you may use the word `ANSWER` to represent the number 42 in any expression:

```
a = sqrt(ANSWER)            ; same as:  a = sqrt(42)
```

This is not mere text substitution, however. The expression is evaluated when the symbol is defined, and `ANSWER` represents the number 42, not the string `"42"`. This is much different than a C-style macro:

```
ANSWER_SYMBOL  .equ  6*9          ; define a symbol
a = 1 / ANSWER_SYMBOL             ; same as:  a = 1 / (6*9)

#define  ANSWER_MACRO  6*9        ; define a macro
a = 1 / ANSWER_MACRO              ; same as:  a = (1/6) * 9

SINGULARITY_SYMBOL   .equ   1/0   ; error at definition time
#define  SINGULARITY_MACRO  1/0   ; no error until used, maybe not even then
```

Directive: *SYMBOL* `.set` *EXPRESSION*

The `.set` directive behaves almost identically to `.equ`. The only difference is that `.equ` will complain if you try to define a symbol that already exists, whereas `.set` will simply go ahead and redefine it. This makes `.equ` appropriate for constants, and `.set` handy for assembler variables:

```
INT_TO_FIXED  .equ  1/I(1)        ; define INT_TO_FIXED = 2^12

multiplier  .set  INT_TO_FIXED
.while  multiplier < -8  ||  multiplier >= 8
  a *= -8
  multiplier  .set  multiplier/(-8)
.endloop
a *= multiplier
```

Directive: *SYMBOL* `.unequ`

The `.unequ` directive makes `bkasm` forget about a symbol. If the symbol had been used previously in an expression, that expression is not affected, but subsequent expressions will not be able to use the symbol. The symbol may be brought back to life with either `.equ` or `.set`. It is not an error to `.unequ` a symbol that doesn't actually exist yet.

```
GOLD  .equ  (1 + sqrt(5))/2
a = GOLD                        ; same as:  a = (1 + sqrt(5))/2
GOLD  .unequ
a = GOLD                        ; error:  GOLD doesn't exist
```

The `def()` Function

The `def()` function may be used in an expression to test whether a symbol exists. The argument is a potential symbol's name, in quotes:

```
a = def("SILVER")               ; same as:  a = 0
SILVER  .equ  (1 - sqrt(5))/2
a = def("SILVER")               ; same as:  a = 1
```

This function is especially useful with `.if` directives, to assemble a block of code only if some flag symbol has been set.

```
a = a * b + DELTA                               ; do some math
.if def("CHECK_FOR_OVERFLOW")                   ; are we debugging?
  if (a < -8 || a >= 8) handleOverflowError     ; if so, check for overflow
.endif                                          ; if not, full speed ahead
```

You will often see these sorts of flag symbols defined right from the command line. This makes it easy to turn them on or off simply by specifying a different `makefile` target.

```
% bkasm -D CHECK_FOR_OVERFLOW myfile.asm
```

5.3.2 Advanced Symbolism

The expression in a `.equ` or `.set` directive is evaluated when the symbol is defined, but it doesn't necessarily have to resolve at that time. `bkasm` represents its symbols, well, symbolically. Any unrecognized words in the expression are assumed to be symbols that will be defined later, usually as externals, labels, or data variables, but possibly later in the same file. These symbols may be manipulated algebraically before they resolve, or even if they never resolve.

```
GENIUS  .equ  0.01*INSPIRATION + 0.99*PERSPIRATION  ; define unresolved symbol
a = 100*GENIUS - 99*PERSPIRATION                    ; same as:  a = 1.234
INSPIRATION .equ 1.234
 ; works even if PERSPIRATION is never defined
```

In the above example, the assignment statement is reduced down to `a = INSPIRATION` when it is assembled, but it doesn't get all the way to `a = 1.234` until the

postlinker revisits the file and notices that `INSPIRATION` has finally been defined. All expressions involving labels and data variables are late resolvers like this, because those symbols aren't defined until the linker's had its say. You don't usually need to know this, but there are a couple subtleties that may confuse you if you let yourself care about them:

```
[glarch] = a      ; piggybacks below
a = 0             ; generates:  sca  0  glarch

[glarch] = a      ; generates dummy instruction:     s1ac  0  glarch
a = ZERO          ; generates unpiggybackable inst:  c     T  0
ZERO  .equ  0
```

If the above example, `a = ZERO` had to be implemented with a less efficient instruction because `bkasm` didn't actually know it would be zero until postlink.

5.3.3 The `.table` Directive

Directive: *SYMBOL* `.table` *EXPRESSION, EXPRESSION, EXPRESSION, ...*

The `.table` directive is basically the plural form of `.equ`. It takes a comma-separated list of expressions, and defines an array of symbols at once.

```
SQUARE  .table  0^2, 1^2, 2^2, 3^2, 4^2, 5^2
ROUND   .table  0, pi/2, pi, 3*pi/2, 2*pi
```

Expressions may refer to a value from the table by giving the table's name followed by a zero-based index in parentheses. If it helps, think of the table lookup as a function.*

```
FIBBY  .table  0, 1, 1, 2, 3, 8, 13, 21, 34, 55, 89
a = FIBBY(0)      ; same as:  a = 0
a = FIBBY(1)      ; same as:  a = 1
a = FIBBY(10)     ; same as:  a = 89
```

`bkasm` inherits Perl's semantics when it comes to negative indices. A negative index will start counting backwards from the end of the table, with -1 referring to the very last entry.

```
FIBBY  .table  0, 1, 1, 2, 3, 8, 13, 21, 34, 55, 89
a = FIBBY(-1)     ; same as:  a = 89
a = FIBBY(-2)     ; same as:  a = 55
a = FIBBY(-11)    ; same as:  a = 0
```

*That's actually what it is.

Because table lookup is a function, the index itself may be a symbol, or any expression.

```
FIBBY  .table  0, 1, 1, 2, 3, 8, 13, 21, 34, 55, 89
NACHOS .equ  3
a = FIBBY(NACHOS*2) ; same as:  a = 13
```

5.3.4 The `.export` and `.x*` Directives

Each source file that you assemble is given its own private symbol table to play with. This means that every one of your source files can define the symbol `foo`, the label `bar`, and the variable `foobar` without fear of confusion.* However, sometimes you want one of your symbols to transcend the boundaries of its file and be usable everywhere. That's what exporting is for. If you want to jump to labels in other files or refer to external variables, those symbols will have to be exported.

Directive: *SYMBOL* `.export`

The `.export` directive indicates that a given symbol should be placed in the global symbol table. There is no corresponding `.import` directive; any exported symbol can be seen from anywhere. The importing happens during postlink, which means that the order of the files on the command line doesn't matter—an exported symbol can be used by any file, even previous ones.

The exporting happens during postlink too, which means that exported symbols can refer to symbols that are defined by the linker, such as variables and labels.† It also means that a symbol is exported with whatever value it had at the end of its file, which is something to remember if you were busy `.set`ting it to various things.

```
NUMBER    .equ  17
NUMBER    .export      ; export an equate

variable  .ds   2
variable  .export      ; export address of a data variable

Here:
Here      .export      ; export address of a label

There     .equ  Here + NUMBER
There     .export      ; export a symbol that depends on a label
```

*`bkasm` won't get confused. You might.

†This implies that exported symbols might be able to refer to other exported symbols. Yes, but this depends on command line ordering, so only do this if you know what you're doing, and maybe not even then.

When it is said that an exported symbol is visible from anywhere, anywhere might be bigger than you'd expect. In Chapter 8, you'll see that the *c* and *asm* output formats, which generate code to be compiled in with the host software, conveniently produce header files that define everything in the global symbol table. This means that when you `.export` a symbol, it shows up not just in your other assembly files, but in the host software as well. Since the primary means of communicating with the 1K is by directly poking around in its memory space, and exported labels and variables presumably describe good places to poke, you might find this to be to be `bkasm`'s most useful feature of all.

It is not an error to export the same symbol from multiple files, so long as both files agree on what the symbol's value should be. Exporting different values for the same symbol is an error. For example, this is okay:

```
;;; snowball.asm:
 LEGS  .equ  4
 LEGS  .export

;;; napoleon.asm:
 LEGS  .equ  4
 LEGS  .export
```

but this is an error:

```
;;; snowball.asm:
 LEGS  .equ  4
 LEGS  .export

;;; napoleon.asm:
 LEGS  .equ  2    ; Error at postlink:
 LEGS  .export    ; Four legs good, two legs bad.
```

You may not export tables. Sorry.

In most cases, you may place the `.export` either before or after the symbol's declaration. The exception is with labels. For optimization and piggybacking reasons, the *1k* target likes to know which labels are actually used, so you must `.export` a label *before* you declare the label itself. If you try to `.export` a label that's already been defined, it will complain.

```
 .export Here     ; correct:  export label before the label is defined
Here:
There:
 .export There    ; error:  cannot export label that's already defined
```

Directive: *SYMBOL* `.x* ...`

Not only can you export non-label symbols before or after their declaration, you can even export them at the same time. If any directive is prefixed with an "x", the first argument of that directive will be implicitly exported. This is most useful with equates and variable declarations:

```
NUMBER    .xequ  17  ; same as:  NUMBER    .export
                     ;           NUMBER    .equ  17
variable  .xds   2   ; same as:  variable  .export
                     ;           variable  .ds   2
```

5.4 Rounding

`bkasm` does all of its computation with so-called *double-precision floating point* math, which provides plenty of precision for most purposes. When you say "`pi`", `bkasm` knows what you are talking about to fourteen digits after the decimal point, which is probably twelve more than you normally remember yourself.

The precision inside a fixed-point DSP processor, on the other hand, is typically more limited; the poor 1K will never know more than seven or so digits of `pi`. Therefore, preparing a number calculated by `bkasm` for consumption by the processor requires chopping off some of those extra digits in a manner that is appropriate for the situation. `bkasm` provides a number of rounding functions and modes that let you describe how and where this chopping should occur.

The `round_*()` Functions

`bkasm` defines six functions for rounding off numbers. Each of them takes two arguments: `BITS` indicates the number of bits after the decimal point to round to, and `NUMBER` is the value to be rounded. A `BITS` of zero effectively rounds to an integer. `BITS` can even be negative to round to a power of two, if you are into that sort of thing.

```
RESULT = round_nearest(BITS, NUMBER)
```

Rounds to the nearest multiple of $2^{-\mathrm{BITS}}$. This is what you normally have in mind when you think about "rounding off". Matlab calls this function `round()`.

```
RESULT = round_in(BITS, NUMBER)
```

Rounds toward zero, producing a result that is equal or smaller in magnitude. You may need to use this rounding mode when calculating certain critical filter coefficients in order to prevent instabilities. Matlab calls this function `fix()`.

```
RESULT = round_out(BITS, NUMBER)
```

Rounds away from zero, producing a result that is equal or larger in magnitude.

```
RESULT = round_down(BITS, NUMBER)
```

Rounds down (or as some people would say, "towards $-\infty$"), producing a result that is less than or equal to `NUMBER`. This is what you get when truncate a number by ANDing a bitmask. If your DSP code truncates numbers in this way, this rounding mode may be useful for producing numbers that correspond. Matlab calls this function `floor()`.

```
RESULT = round_up(BITS, NUMBER)
```

Rounds up (or, towards $+\infty$), producing a result that is greater than or equal to `NUMBER`. Matlab calls this function `ceil()`.

```
RESULT = round_error(BITS, NUMBER)
```

Refuses to round at all. Instead, if there are significant bits past `BITS`, an error is generated. In certain situations, you will come across a coefficient that has no tolerance whatsoever. When you want to be certain that the number you've calculated is exactly the number that is being used, down to the last bit, this can come in handy.

Examples:

```
a = round_nearest(2, -0.51)    ; same as:  a = -0.5
a = round_in(2, -0.51)         ; same as:  a = -0.5
a = round_out(2, -0.51)        ; same as:  a = -0.75
a = round_down(2, -0.51)       ; same as:  a = -0.75
a = round_up(2, -0.51)         ; same as:  a = -0.5
a = round_error(2, -0.51)      ; error:  -0.51 has more than 2 significant bits
a = round_error(2, -0.5)       ; same as:  a = -0.5  (no error)
```

5.4.1 The `.round` Directive and `round()` Function

The functions in the previous section are useful if you have some number that you want explicitly rounded to a particular precision. But in a typical program, most rounding is going on behind the scenes. Whenever `bkasm` has to stuff a constant into S3.24, or any other fixed-point representation, it has to round off.

Directive: `.round` *MODE*

The `.round` directive tells `bkasm` how it should round off all numbers from that point on. There are six possible `MODE`s, corresponding to the six functions described above. The default mode is "nearest".*

```
;;; reminder:  F(x) = x * 2^-24
.round nearest
  a = F(-1.1)         ; same as:  a = F(-1)
.round in
  a = F(-1.1)         ; same as:  a = F(-1)
.round out
  a = F(-1.1)         ; same as:  a = F(-2)
.round down
  a = F(-1.1)         ; same as:  a = F(-2)
.round up
  a = F(-1.1)         ; same as:  a = F(-1)
.round error
  a = F(-1.1)         ; error:  F(-1.1) has more than 24 significant bits
  a = F(-1)           ; no error
```

The `round()` Function

The `round()` function has the same syntax as the specialized rounding functions above, but it rounds according to the whatever the current rounding mode is.

```
.round nearest
  a = round(2, 0.51)  ; same as:  a = 0.5
.round down
  a = round(2, 0.51)  ; same as:  a = 0.5
.round up
  a = round(2, 0.51)  ; same as:  a = 0.75
```

5.5 Precision

`bkasm`'s rounding features are for making the best of the bits you're given. `bkasm`'s precision management is about making sure you're given the bits you need.

Consider the following implementation of a free-running timer intended to wrap around once a second:

```
DELTA_PHASE .equ 1/48000   ; increment counter by 1/48000 each sample period
MAX_PHASE   .equ 1         ; after 48000 samples, or one second, we wrap around
```

*In case you are curious, constants that don't resolve immediately will still be properly rounded. Even if a constant doesn't resolve until postlink, it will be rounded using whatever mode was in effect when you wrote its instruction.

```
a = [phase] + DELTA_PHASE  ; add increment to current phase
b = a                      ; stash incremented phase in b
a = a - MAX_PHASE          ; check if we've gone past the maximum phase
if (a < 0) {               ; if so, leave wrapped phase in a
  a = b                    ; if not, restore phase from b
}
[phase] = a                ; store new phase
```

This appears simple and innocent enough. Which is why, if you were to run it, you might be surprised to find that it's off by almost 10%—the timer actually wraps around once every 0.91 seconds.

The problem is with this line:

```
a = [phase] + DELTA_PHASE  ; generates:  1mc  DELTA_PHASE  phase
```

The `1mc` instruction only gives 18 bits of precision to its constant operand. `DELTA_PHASE` is so small that most of those bits are zero, leaving only *three* significant bits. It is, in fact, equivalent to this:

```
.option allow-integer-constants
1mc  0x000005  phase
```

The purpose of `bkasm`'s precision management is to help you avoid this situation. You tell `bkasm` how much precision you need for certain symbols and constants, and `bkasm` will make sure you get it.

5.5.1 The `p()` Function and `.pequ` Directive

The `p()` function can be used to associate a precision with a term in an expression. The first argument specifies the number of bits of fractional precision required, and the second argument is the value itself:

```
a = a + p(24, 1/3)          ; similar to:  a = a + 1/3
                            ;  but 1/3 must be represented in at least S3.24
```

This function can be used in a symbol definition as well, to associate a precision with a symbol:

```
ONE_THIRD  .equ  p(24, 1/3)  ; similar to:  ONE_THIRD .equ 1/3
a = a + ONE_THIRD            ;  but ONE_THIRD must be represented in at least S3.24
```

This precision will be associated with any expression that uses this symbol, which means that it can propagate to other symbol definitions:

```
ONE_THIRD  .equ  p(24, 1/3)       ; ONE_THIRD has a precision of 24
TWO_THIRDS .equ  ONE_THIRD + 1/3  ; TWO_THIRDS also gets a precision of 24
```

Directive: *NAME* `.pequ` *VALUE*

If you have several symbols that should be set to some particular precision, you can set the `--precision` option to the number of bits that you want, and then define your symbols with the `.pequ` directive:

```
.option precision=24         ; tell .pequ to use a precision of 24
ONE_THIRD  .pequ  1/3        ; same as:  ONE_THIRD .equ p(24, 1/3)
```

There is a corresponding `.pset` directive that behaves the same precision-wise, but allows symbols to be redefined like `.set`. The `--precision` option is only used by `.pequ` and `.pset`; it doesn't affect anything else.

For convenience, `.pequ` and `.pset` can take an optional argument which specifies an explicit precision to use instead of `--precision`. The following two lines are equivalent:

```
ONE_THIRD  .equ   p(24, 1/3)
ONE_THIRD  .pequ  1/3, 24
```

Use whichever style makes sense to you.

5.5.2 The `ep()` Function and `.epequ` Directive

It's nice to be able to specify a minimum number of bits for constants and symbols, but typically, that's not what you really care about. In the timer example, it's perfectly possible for even an experienced coder to notice `DELTA_PHASE` only getting 18 bits of precision, and still not raise an eyebrow. The shock doesn't come from the 18 bits—it comes from the 10% deviation from the intended value.

Looking at it from the other direction, when you design a system, you normally aren't directly concerned with the number of bits you're throwing around. What you *are* concerned with (or should be) is relative error—the difference, percentage-wise, between the ideal coefficient and the coefficient you implement. When you design, you decide how much error the system can tolerate and still meet the specifications. `bkasm` refers to this tolerance as *epsilon*.

Specifying a precision in terms of epsilon is very similar to specifying it in bits. The `ep()` function is just like `p()`, but the first argument is a fractional tolerance:

```
a = a + ep(0.001, 1/3)       ; similar to:  a = a + 1/3
                             ;  but the addend must be 1/3 +/- 1/3000
```

In the above example, the precision associated with the constant is such that it cannot deviate from 1/3 by more than a factor of 0.001, or 0.1%. In this case, that translates to 12 bits. `ep()` works in a symbol definition as well:

```
ONE_THIRD  .equ  ep(0.001, 1/3)  ; similar to:  ONE_THIRD .equ 1/3
a = a + ONE_THIRD                ;  but ONE_THIRD must get at least S3.12
```

Directive: *NAME* `.epequ` *VALUE*

The epsilon analogue to `--precision` is, unsurprisingly enough, `--epsilon`. This option is used by the `.epequ` and `.epset` directives to define a symbol with an implicit precision:

```
.option epsilon=0.001        ; tell .epequ to use an epsilon of 0.001
ONE_THIRD  .epequ  1/3       ; same as:  ONE_THIRD .equ ep(0.001, 1/3)
.option epsilon=0.01         ; switch to a more tolerant tolerance
ONE_THIRD  .epset  1/3       ; same as:  ONE_THIRD .set ep(0.01, 1/3)
```

Like the directives in the last section, the epsilon directives can take an optional second argument specifying the epsilon to use. The following lines are equivalent:

```
ONE_THIRD  .equ    ep(0.001, 1/3)
ONE_THIRD  .epequ  1/3, 0.001
```

One caveat with `ep()`, `.epequ`, and friends is that the value you're epsilonning must resolve immediately, because `bkasm` needs to translate your epsilon into a plain old bit precision. In practice, this isn't much of a problem, because such values don't tend to be very dynamic.

5.5.3 Using Precision

By now, it should be clear that the first step in fixing the timer example is to assign `DELTA_PHASE` a precision:

```
DELTA_PHASE .equ p(24, 1/48000)  ; DELTA_PHASE must be represented as S3.24
```

Now, any expression that uses `DELTA_PHASE` must be given at least 24 bits of fractional precision.* If `bkasm` cannot dig up this much precision, it will generate an error:

*This is not strictly true, but believe it anyway. The rules by which precision information propagates through operators in an expression are too boring to describe here, but you can trust that they make sense.

```
a = [phase] + DELTA_PHASE  ; error: no instruction implements this form
                           ;        with the required precision
```

There's more to precision than just error messages, though. Suppose you had already noticed that DELTA_PHASE would need all the bits it could get, and wrote the code to reflect that:

```
a = [phase]                ; get current phase
a += DELTA_PHASE           ; add increment, in a separate instruction
```

This looks like it will use all 24 bits of DELTA_PHASE, even without you telling bkasm about DELTA_PHASE's precision. But it might not. Suppose that you decided to make a copy of phase before you incremented it:

```
a = [phase]                ; get current phase
[oldphase] = a             ; remember current phase for later
a += DELTA_PHASE           ; add increment
```

If you don't specify any precision, bkasm will see that the store can piggyback on the increment, and DELTA_PHASE will be cut back down to S3.18:

```
a = [phase]                ; generates:  cm    1  phase
[oldphase] = a             ; piggybacks below
a += DELTA_PHASE           ; generates:  s1ac  DELTA_PHASE  oldphase
```

If bkasm is informed that DELTA_PHASE needs extra precision, it will gladly avoid the piggyback in order to keep it at S3.24:

```
DELTA_PHASE  .equ  p(24, 1/48000)  ; DELTA_PHASE gets a precision of 24

a = [phase]                        ; generates:  cm    1  phase
[oldphase] = a                     ; generates:  s1ac  0  oldphase
a += DELTA_PHASE                   ; generates:  1ac   DELTA_PHASE
```

The above solution could also be written as:

```
DELTA_PHASE  .equ  1/48000       ; DELTA_PHASE has no precision
a = [phase]
[oldphase] = a
a += p(24, DELTA_PHASE)          ; but this expression does
```

Incidentally, with the corrections above, the timer wraps around every 1.001 seconds. The error has dropped from almost 10% down to about 0.1%. What a difference six bits make!

5.5.4 Natural Precision

On some occasions, you may find `bkasm` blatantly ignoring your request for precision. For example:

```
.option precision=24        ; set .pequ precision to 24
DELTA_PHASE  .pequ  1/256   ; DELTA_PHASE gets a precision of 24
a = [phase] + DELTA_PHASE   ; but this works without error!
```

`bkasm` isn't simply being rude to you. What happened here was that `bkasm` noticed that 1/256 looks exactly the same whether you represent it in S3.24 or S3.18, and knew it could get away with shoving it into a S3.18 operand without affecting the functionality.

1/256 is said to have a *natural precision* of eight, which means that once you look eight bits past the radix point, all you'll see are zeros. That is, this number can be represented *exactly* in S3.8 format, and any more is overkill. So even though you assigned the symbol a precision of 24, `bkasm` is smart enough to see that the value's natural precision is lower, and go with that instead.

Natural precision is also used for other things than overriding your specified precision. For example, the 1K offers two instructions that multiply the accumulator by a constant and add another constant. One of them represents the multiplier in S3.18 and the addend in S3.8, and the other does the opposite:

```
a = a * p(18, 1/3) + p( 8, 1/7)  ; generates:  cad  1/3  1/7
a = a * p( 8, 1/3) + p(18, 1/7)  ; generates:  dac  1/3  1/7
```

In the absence of explicit precisions like those above, `bkasm` normally prefers to give the extra bits to the multiplier. But if the multiplier has a natural precision of eight or less, S3.18 would obviously be wasted on it, so the addend gets the high precision instead:

```
a = a * 1/3 + 1/7               ; generates:  cad  1/3  1/7
a = a * 1/4 + 1/7               ; generates:  dac  1/4  1/7
```

5.6 Labels

A label is a symbol which points to an address in code memory. You declare a label simply by writing its name, followed by a colon:

```
DefJam:
 a = [hip] + HOP
```

In the above example, the symbol `DefJam` is defined as the address of the subsequent instruction. You may put instructions on the same line as the label, if you prefer:

```
BlueNote:  a = 7/4
```

The label declaration should be placed near the start of a line, although it can be preceded by whitespace or curly braces. You may also stack multiple labels on the same line, if you think that's readable:

```
if (b == [mg]) {
     Arista:  [rc] = a                  ; labels may be indented
} Columbia: Atlantic:  u = [niversal]   ; labels may be stacked
```

The primary purpose of labeling your code is to help you jump around.

```
 if ([house] == PAIN) TommyBoy   ; if condition is true, jump to ----
 a = UP + [down]                 ;                                    |
TommyBoy:                        ; here <-----------------------------
```

See Section 4.6 for the details of 1K flow control.

As described on page 74, an exported label is also useful for telling the host software where it should load in a particular routine:

```
 .export StartOfOscillator
StartOfOscillator:
 ;;; oscillator code goes here
```

If you are using the *c* or *asm* output formats, your host software will be compiled with a constant that says where `StartOfOscillator` is, so it knows where to load in its oscillators. Don't forget that you must export a label *before* it is defined.

Labels can also be useful for defining a local scope. See Section 5.8 for the details of locality.

5.6.1 Local Labels

5.6.2 Macro Labels

5.7 Data

5.8 Locals

5.9 Conditional Assembly

5.10 Iterative Assembly

5.11 C-style Macros

5.12 Asm-style Macros

5.13 Sections

5.14 Embedded Perl

5.15 Legacy Directives

Chapter 6

The Linker

Chapter 7

The Listing File

Chapter 8

Output Formats

8.1 hexdump

8.2 c

8.3 asm

Chapter 9

Extending `bkasm`

9.1 Output Formats

9.2 Targets

9.3 Linkers

Appendix A

1K Machine Instructions

`1mc C M`	$a = [M] + C_{18}$	$u = [M]$	
`l1mc C M`	$a = [M] + C_{18}$	$u = [M]$	$b = [M]$
`mmc C M`	$a = [M] * [M] + C_{18}$	$u = [M]$	
`xmmc C M`	$a = [M] * [M] + C_{18}$	$u = [M]$	$b = a$
`lmmc C M`	$a = [M] * [M] + C_{18}$	$u = [M]$	$b = [M]$
`amb M`	$a = [M] * a + b$	$u = [M]$	
`amc C M`	$a = [M] * a + C_{18}$	$u = [M]$	
`lamc C M`	$a = [M] * a + C_{18}$	$u = [M]$	$b = [M]$
`bmc C M`	$a = [M] * b + C_{18}$	$u = [M]$	
`cm C M`	$a = [M] * C_{18}$	$u = [M]$	
`xcm C M`	$a = [M] * C_{18}$	$u = [M]$	$b = a$
`lcm C M`	$a = [M] * C_{18}$	$u = [M]$	$b = [M]$
`cma C M`	$a = [M] * C_{18} + a$	$u = [M]$	
`xcma C M`	$a = [M] * C_{18} + a$	$u = [M]$	$b = a$
`lcma C M`	$a = [M] * C_{18} + a$	$u = [M]$	$b = [M]$
`cmb C M`	$a = [M] * C_{18} + b$	$u = [M]$	

`1ac C`	$a = a + C_{24}$	$u = a$		
`s1ac C M`	$a = a + C_{18}$	$u = a$	$[M] = a$	
`x1ac C`	$a = a + C_{24}$	$u = a$		$b = a$
`sx1ac C M`	$a = a + C_{18}$	$u = a$	$[M] = a$	$b = a$
`l1ac C M`	$a = a + C_{18}$	$u = a$		$b = [M]$
`aac C`	$a = a * a + C_{24}$	$u = a$		
`saac C M`	$a = a * a + C_{18}$	$u = a$	$[M] = a$	
`sxaac C M`	$a = a * a + C_{18}$	$u = a$	$[M] = a$	$b = a$
`laac C M`	$a = a * a + C_{18}$	$u = a$		$b = [M]$
`bac C`	$a = a * b + C_{24}$	$u = a$		
`sbac C M`	$a = a * b + C_{18}$	$u = a$	$[M] = a$	
`lca C M`	$a = a * C_{18}$	$u = a$		$b = [M]$
`sca C M`	$a = a * C_{18}$	$u = a$	$[M] = a$	$b = [M]$
`sxca C M`	$a = a * C_{18}$	$u = a$	$[M] = a$	$b = a$
`cam C M`	$a = a * C_{18} + [M]$	$u = a$		
`cab C`	$a = a * C_{18} + b$	$u = a$		
`scab C M`	$a = a * C_{18} + b$	$u = a$	$[M] = a$	
`cad C D`	$a = a * C_{18} + D_8$	$u = a$		
`dac D C`	$a = a * D_8 + C_{18}$	$u = a$		
`1bc C`	$a = b + C_{24}$	$u = b$		
`s1bc C M`	$a = b + C_{18}$	$u = b$	$[M] = a$	
`bbc C`	$a = b * b + C_{24}$	$u = b$		
`sbbc C M`	$a = b * b + C_{18}$	$u = b$	$[M] = a$	
`cb C`	$a = b * C_{18}$	$u = b$		
`scb C M`	$a = b * C_{18}$	$u = b$	$[M] = a$	
`cbm C M`	$a = b * C_{18} + [M]$	$u = b$		
`scba C M`	$a = b * C_{18} + a$	$u = b$	$[M] = a$	
`sauc C M`	$a = u * a + C_{18}$		$[M] = a$	
`sbua M`	$a = u * b + a$		$[M] = a$	
`sbuc C M`	$a = u * b + C_{18}$		$[M] = a$	
`scu C M`	$a = u * C_{18}$		$[M] = a$	
`scua C M`	$a = u * C_{18} + a$		$[M] = a$	
`scub C M`	$a = u * C_{18} + b$		$[M] = a$	

Appendix B

1K Forms

form	*available piggybacks*			*instruction*
Transfers				
$a = u$	$[M] = a$			`scu`
$a = [M]$		$b = a$	$b = [M]$	`*cm`
$a = a$	$[M] = a$	$b = a$	$b = [M]$	`*1ac`
$a = b$	$[M] = a$			`*1bc`
$a = C_{24}$				`c`
$a = 0$	$[M] = a$	$b = a$	$b = [M]$	`*ca`
Addition				
$a = u + a$	$[M] = a$			`scua`
$a = u + b$	$[M] = a$			`scub`
$a = [M] + a$		$b = a$	$b = [M]$	`*cma`
$a = [M] + b$				`cmb`
$a = [M] + C_{24^L}$			$b = [M]$	`*1mc`
$a = a + C_{24^{SL}}$	$[M] = a$	$b = a$	$b = [M]$	`*1ac`
$a = a + b$	$[M] = a$			`*cab`
$a = b + C_{24^S}$	$[M] = a$			`*1bc`

form	*available piggybacks*			*instruction*
Subtraction				
$a = [M] - a$				`cam`
$a = [M] - b$				`cbm`
$a = a - u$	$[M] = a$			`scua`
$a = a - [M]$		$b = a$	$b = [M]$	`*cma`
$a = a - b$	$[M] = a$			`*cb`
$a = b - u$	$[M] = a$			`scub`
$a = b - [M]$				`cmb`
$a = b - a$	$[M] = a$			`*cab`
$a = C_{18} - a$				`dac`
Multiplication				
$a = u * a$	$[M] = a$			`sauc`
$a = u * b$	$[M] = a$			`sbuc`
$a = u * C_{18}$	$[M] = a$			`scu`
$a = [M] * [M]$		$b = a$	$b = [M]$	`*mmc`
$a = [M] * a$			$b = [M]$	`*amc`
$a = [M] * b$				`bmc`
$a = [M] * C_{18}$		$b = a$	$b = [M]$	`*cm`
$a = a * a$	$[M] = a$	$b = a$	$b = [M]$	`*aac`
$a = a * b$	$[M] = a$			`*bac`
$a = a * C_{18}$	$[M] = a$	$b = a$	$b = [M]$	`*ca`
$a = b * b$	$[M] = a$			`*bbc`
$a = b * C_{18}$	$[M] = a$			`*cb`

form	*available piggybacks*			*instruction*
Multiplication and Addition				
$a = u * a + C_{18}$	$[M] = a$			`sauc`
$a = u * b + a$	$[M] = a$			`sbua`
$a = u * b + C_{18}$	$[M] = a$			`sbuc`
$a = u * C_{18} + a$	$[M] = a$			`scua`
$a = u * C_{18} + b$	$[M] = a$			`scub`
$a = [M] * [M] + C_{18}$		$b = a$	$b = [M]$	`*mmc`
$a = [M] * a + b$				`amb`
$a = [M] * a + C_{18}$			$b = [M]$	`*amc`
$a = [M] * b + C_{18}$				`bmc`
$a = [M] * C_{18} + a$		$b = a$	$b = [M]$	`*cma`
$a = [M] * C_{18} + b$				`cmb`
$a = a * a + C_{24^{SXL}}$	$[M] = a$	$b = a$	$b = [M]$	`*aac`
$a = a * b + C_{24^{S}}$	$[M] = a$			`*bac`
$a = a * C_{18} + [M]$				`cam`
$a = a * C_{18} + b$	$[M] = a$			`*cab`
$a = a * C_{18} + C_{8}$				`cad`
$a = a * C_{8} + C_{18}$				`dac`
$a = b * b + C_{24^{S}}$	$[M] = a$			`*bbc`
$a = b * C_{18} + [M]$				`cbm`
$a = b * C_{18} + a$	$[M] = a$			`*cb`
Masking				
$a = a\&b$				`andb`
$a = a\&C_{24}$				`andc`

Notes

The precisions of some constants are affected by certain piggybacks. If a form refers to C_{24^S}, this means that the constant gets 24-bit precision in the absence of piggybacking, but drops to 18-bit precision with a store piggyback ($[M] = a$). C_{24^X} and C_{24^L} refer similarly to the xfer ($b = a$) and load ($b = [M]$) piggybacks.

The instruction given in the last column is the typical one used for the given form, but `bkasm` may use other instructions in some instances. Do not rely on a form always being mapped to a particular instruction.

Appendix C

Directives

The following are terse descriptions of `bkasm`'s directives, for use as a quick reference. A full discussion of a directive, with examples, can usually be found on the page number indicated on the right side.

As explained on page 62, the first argument of any directive may go either before or after the directive itself. Any directive may also be written "C-style" with a pound sign (#) instead of a dot, in which case all arguments must come after the directive.

`.abs` **Synopsis:** *SYMBOL* `.abs` *[EXPRESSION]* p. **??**
Example: `myvariable abs 0x400`

Legacy directive for `1kasm` compatibility. Creates a symbol `SYMBOL` which equals `EXPRESSION`. If `EXPRESSION` is not given, uses the next available address. The dot is optional. This directive is strongly deprecated; use `.d*` and related directives for memory allocation instead.

`.d*` **Synopsis:** *SYMBOL* `.d*` *[LENGTH]* p. **??**
Example: `mybuffer .ds 256`

The `.ds` directive allocates `LENGTH` words of data in the *sdata* chunk of the current section, and defines a symbol which points to them. "`s`" may be replaced with any letter in order to allocate data in that particular chunk. If `LENGTH` is not given, it defaults to one.

`.data` **Synopsis:** *SYMBOL* `.data` *CHUNK [, LENGTH]* p. **??**
Example: `mybuffer .data s, 256`

Allocates `LENGTH` words of data in the *CHUNK*`data` chunk of the current section, and defines a symbol which points to them. If `LENGTH` is not given, it defaults to one.

`.define` **Synopsis:** *CMACRO* `.define` *DEFINITION* p. **??**
Example: `.define DEG2RAD(deg) (180*pi/deg)`

Defines a C-style text-substitution macro. If the macro takes arguments, `CMACRO` must be placed *after* `.define`, as shown in the example.

`.else` **Synopsis:** `.else` p. **??**

Begins a block of code that is assembled only if the condition in the preceding `.if` or `.elsif` directive is false.

`.elsif` **Synopsis:** `.elsif` *EXPRESSION* p. **??**
Example: `.elsif BANANAS != 0`

Begins a block of code that is assembled only if `EXPRESSION` is true *and* the condition in the preceding `.if` or `.elsif` directive is false.

`.endif` **Synopsis:** `.endif` p. **??**

Ends a conditional block of code started with `.if`.

`.endloop` **Synopsis:** `.endloop` p. **??**

Ends an iterative block of code started with `.for`, `.while`, or `.repeat`.

`.endm` **Synopsis:** `.endm` p. **??**

Ends a macro definition started with `.macro` or `.perlmacro`.

`.endperl` **Synopsis:** `.endperl` p. **??**

Ends a block of Perl code started with `.perl`.

`.epequ` **Synopsis:** *SYMBOL* `.epequ` *EXPRESSION [, EPSILON]* p. 80
Example: `FREQUENCY .epequ 1000/48000, 0.001`

Defines a symbol and associates a particular precision with it. If `EPSILON` is not given, it defaults to the value of the `--epsilon` option.

`.epset` **Synopsis:** *SYMBOL* `.epset` *EXPRESSION [, EPSILON]* p. 80
Example: `FREQUENCY .epset 1000/48000, 0.001`

Defines a symbol, redefining it if it already exists, and associates a particular precision with it. If `EPSILON` is not given, it defaults to the value of the `--epsilon` option.

`.equ` **Synopsis:** *SYMBOL* `.equ` *EXPRESSION* p. 69
Example: `PHI .equ (1 + sqrt(5) ) / 2`

Evaluates `EXPRESSION` and creates the symbol `SYMBOL`. The dot is optional, but recommended.

`.error` **Synopsis:** `.error` *MESSAGE* p. 66
Example: `.error Everything is wrong!`

Generates an assembler error.

`.eval` **Synopsis:** `.eval` *PERLCODE* p. **??**
Example: `.eval "a += " . (ADD_B ? "b" : "1.7")`

Evaluates `PERLCODE` as a line of Perl code, and then assembles the result as `bkasm` code. `bkasm`'s preprocessing places some restrictions on `PERLCODE`; the `.perl` directive is more robust.

`.exitm` **Synopsis:** `.exitm` p. **??**

During evaluation of a macro, immediately terminates the macro.

`.export` **Synopsis:** *SYMBOL* `.export` p. 73

Indicates that `SYMBOL` should be placed in the global symbol table, where it can be seen by all source files and possibly included in the output file.

`.for` **Synopsis:** `.for` *SYMBOL* `=` *EXPR* `..` *EXPR* p. **??**
Example: `.for COUNT = 1 .. 17`

Begins a block of code, ending with `.endloop`, which is assembled multiple times as `SYMBOL` is iterated over the indicated range. `SYMBOL` is a local symbol which disappears after the `.endloop`.

`.if` **Synopsis:** `.if` *EXPRESSION* p. **??**
Example: `.if BANANAS == 4`

Begins a block of code that is assembled only if `EXPRESSION` is true.

`.include` **Synopsis:** `.include` *FILENAME* p. 65
Example: `.include "definitions.inc"`

Reads and assembles the given file.

`.l*` **Synopsis:** *SYMBOL* `.l*` *[LENGTH [, ENDSCOPE]]* p. **??**
Example: `localbuffer .ls 16`

Behaves like `.d*`, except that the allocated data is locally scoped. See `.local` for a description of `ENDSCOPE`.

`.lequ` **Synopsis:** *SYMBOL* `.lequ` *EXPRESSION [, ENDSCOPE]* p. **??**
Example: `ANGLE .lequ pi/8`

Evaluates `EXPRESSION` and creates the locally-scoped symbol `SYMBOL`. If `ENDSCOPE` is not given, the symbol disappears (or reverts back to whatever it was before) at the end of the innermost block. `ENDSCOPE` can be a label or string of tildes in order to indicate a named scope or local label scope respectively.

`.local` **Synopsis:** *SYMBOL* `.local` *CHUNK [,LENGTH [,ENDSCOPE]]* p. **??**
Example: `localbuffer .local s, 16`

Allocates `LENGTH` words of locally-scoped data in the *CHUNKdata* chunk of the current section, and defines a symbol which points to them. If `ENDSCOPE` is not given, the data is deallocated at the end of the innermost block. `ENDSCOPE` can be a label or string of tildes in order to indicate a named scope or local label scope respectively.

`.loption` **Synopsis:** `.loption` *OPTION[=VALUE]* p. **??**
Example: `.loption precision=24`

This directive has not been implemented yet.

`.macro` **Synopsis:** *MACRO* `.macro` *[ARGUMENTS/PATTERN]* p. **??**
Example: `getdata .macro $pointer, $length`

Begins a macro definition, ending with `.endm`. `.macro` may be followed by either a comma-separated list of macro arguments, as shown above, or a quoted string indicating a pattern to match. Macro arguments must begin with a dollar sign.

`.mem` **Synopsis:** *SYMBOL* `.mem` *LENGTH* p. **??**
Example: `mybuffer mem 256`

Legacy directive for `1kasm` compatibility. Creates a symbol `SYMBOL` which points to a buffer of length `LENGTH`. The dot is optional. This directive is strongly deprecated; use `.d*` and related directives for memory allocation instead.

`.option` **Synopsis:** `.option` *`OPTION[=VALUE]`* p. 63
Example: `.option list-number-format=hex`

Sets an assembler option. If `VALUE` is not given, it defaults to one.

`.pequ` **Synopsis:** *`SYMBOL`* `.pequ` *`EXPRESSION [, PRECISION]`* p. 79
Example: `FREQUENCY .epequ 1000/48000, 24`

Defines a symbol and associates a particular precision with it. If `PRECISION` is not given, it defaults to the value of the `--precision` option.

`.perl` **Synopsis:** `.perl` p. **??**

Begins a block of Perl code, ending with `.endperl`. The code is evaluated, and the result is then assembled as `bkasm` code.

`.perlmacro` **Synopsis:** *`MACRO`* `.perlmacro` *`[ARGUMENTS/PATTERN]`* p. **??**
Example: `getdata .perlmacro $pointer, $length`

Begins a perlmacro definition, ending with `.endm`. The directive syntax is the same as with `.macro`, but the body of the macro definition is compiled as a Perl subroutine. When the macro is invoked, the subroutine is called with the macro arguments in lexical variables, and the return value is assembled as `bkasm` code.

`.popsection` **Synopsis:** `.popsection` p. **??**

Sets the current section to whatever it was at the time of the last `.pushsection` directive.

`.pset` **Synopsis:** *`SYMBOL`* `.pset` *`EXPRESSION [, PRECISION]`* p. 79
Example: `FREQUENCY .epequ 1000/48000, 24`

Defines a symbol, redefining it if it already exists, and associates a particular precision with it. If `PRECISION` is not given, it defaults to the value of the `--precision` option.

`.pushsection` **Synopsis:** `.pushsection` p. **??**

Makes a note of the current section, for later use by `.popsection`.

`.repeat` **Synopsis:** `.repeat` *EXPRESSION* p. **??**
Example: `.repeat 4`

Begins a block of code, ending with `.endloop`, which is assembled `EXPRESSION` times.

`.round` **Synopsis:** `.round` *MODE* p. 76
Example: `.round nearest`

Sets the current rounding mode, which affects the `round()` function and all internal fixed-point rounding.

`.section` **Synopsis:** `.section` *SECTION* p. **??**
Example: `.section Equalizer/FilterBank`

Sets the section into which subsequent instructions and data allocation will be assembled.

`.seg` **Synopsis:** *SYMBOL* `.seg` *[EXPRESSION]* p. **??**
Example: `seg 0x100`

Legacy directive for `1kasm` compatibility. Sets the address for subsequent `.abs` and `.mem` directives. The dot is optional. This directive is strongly deprecated; use `.d*` and related directives for memory allocation instead.

`.set` **Synopsis:** *SYMBOL* `.set` *EXPRESSION* p. 70
Example: `COUNTER .set COUNTER + 1`

Evaluates `EXPRESSION` and defines the symbol `SYMBOL`, redefining it if it already exists.

`.setdef` **Synopsis:** *CMACRO* `.setdef` *DEFINITION* p. **??**
Example: `.setdef CURRENTMASK(x) (x & 0xff00)`

Defines a C-style text-substitution macro, redefining it if it already exists. If the macro takes arguments, `CMACRO` must be placed *after* `.setdef`, as shown in the example.

`.table` **Synopsis:** *TABLE* `.table` *EXPR [, EXPR, ...]* p. 72
Example: `PRIMES .table 2, 3, 5, 7, 11, 13, 17, 19`

Evaluates all of the arguments, and creates a function `TABLE(x)` which returns the xth value in the list. If x is negative, it counts backwards from the end of the list.

`.undef` **Synopsis:** *`CMACRO`* `.undef` p. ??

Removes a C-style macro created with `.define` or `.setdef`, if one exists. `CMACRO` should just be the macro's name, without any argument list.

`.unequ` **Synopsis:** *`SYMBOL`* `.unequ` p. 70

Removes the `SYMBOL` symbol, if it exists.

`.warn` **Synopsis:** `.warn` *`MESSAGE`* p. 66
Example: `.warn Things are strange...`

Generates an assembler warning.

`.while` **Synopsis:** `.while` *`EXPRESSION`* p. ??
Example: `.while COUNT < 5`

Begins a block of code, ending with `.endloop`, which is repeatedly assembled as long as `EXPRESSION` evaluates true. If `EXPRESSION` is false to begin with, the block is never assembled.

`.x*` **Synopsis:** *`SYMBOL`* `.x* ...` p. 73
Example: `publicbuffer .xds 256`

If any directive is prefixed with an "x", the first argument of that directive is interpreted as the name of a symbol to export. The directive is then evaluated normally.

Appendix D

Options

The following are terse descriptions of `bkasm`'s options, for use as a quick reference. The page numbers on the right indicate where the option is introduced in context; a more in-depth discussion sometimes awaits there.

As explained on page 19, any option may be set from either the command line or in a source file:

```
# command line:
% bkasm --foo-bar=3 myfile.asm

; source file:
.option foo-bar=3
```

If the option is simply named, without any equals sign, it is implicitly set to 1. Most options may be disabled by setting them to 0.

`--allow-alternative-radix-designators` p. ??

Assembler: Allows hexadecimal numbers such as `0x123` to be expressed as `$123` or `123H`. Binary numbers such as `0b0101` may be given as `%0101` or `0101B` and octal numbers such as `0123` may be given as `123Q`. Enabled by default.

`--allow-destructive-if` p. 57

1k target: Allows you to write `if` statements that implicitly clobber the accumulator, such as `if (a > 4)`.

`--allow-integer-constants` p. 28
1k target: Compatibility option, enabled by `--compatible`. With this option enabled, if the constant operand of a *1k* machine instruction begins with a radix designator, it will not be interpreted as a fixed-point number; instead, the bit pattern will be used directly in the constant field of the instruction word.

`--allow-one-sided-overflow` p. 51
1k target: With this option enabled, a lone `(a >= -8)` or `(a < 8)` will actually refer to `(a >= -8) && (a < 8)`.

`--allow-quote-mark-suffix` p. ??
1k target: Compatibility option, enabled by `--compatible`. With this option enabled, `foo"` and `foo'` refer to `foo_TAIL` and `foo_CENTER` respectively. Incompatible with pattern macros, among other things.

`--allow-skip-count` p. ??
1k target: Compatibility option, enabled by `--compatible`. Quells the warning when the `skip` instruction is given a skip count instead of a label.

`--case-insensitive` p. ??
Global: Compatibility option, enabled by `--compatible`. Converts source files to lowercase before assembling.

`--cmacro-depth=`*`NUMBER`* p. ??
Assembler: Specifies how deeply C-style macros may recurse before `bkasm` gets suspicious. Defaults to 32.

`--compatible` (`-k`) p. 21
Global: Enables some options to help compatibility with code written for the `1kasm` tool.

`--define-symbol=`*`NAME=VALUE`* (`-D` *`NAME=VALUE`*) p. 21
Assembler: Defines a symbol that will be visible from all source files. Only useful from the command line; in a source file, use `.equ` and friends. If `VALUE` is omitted, it defaults to 1.

`--epsilon=`*`NUMBER`* p. 80
Assembler: Specifies the default epsilon value used by `.epequ` and `.epset`. Defaults to 1.

`--format=`*`FORMATNAME`* (`-F` *`FORMATNAME`*) p. 20

Global: Specifies the output format to use for generating the output files. This option need not go on the command line—it may be specified in any source file, including the linker script. Defaults to *hexdump*.

`--include-path=`*`PATHNAME`* (`-I` *`PATHNAME`*) p. 20

Assembler: Specifying this option adds to the list of pathnames which are searched when the `.include` directive is used with a filename in <pointy brackets>.

`--linker=`*`LINKERNAME`* (`-L` *`LINKERNAME`*) p. 22

Global: Specifies the linker to use. Defaults to *linker*.

`--linkfile=`*`FILENAME`* (`-l` *`FILENAME`*) p. 20

Global: Specifies the linker script to use. If not given, a simple default script is used.

`--list-eval-definitions` p. ??

Listing file: If enabled, includes the Perl code from `.perl` blocks in the listing file. Enabled by default, but you may want to disable it to avoid cluttering up your listing.

`--list-evals` p. ??

Listing file: If enabled, includes the assembly output from `.perl` blocks in the listing file. Enabled by default.

`--list-included-files` p. ??

Listing file: If enabled, includes the contents of files included with the `.include` directive in the listing file. Enabled by default, but you may want to disable it to avoid cluttering up your listing.

`--list-macro-definitions` p. ??

Listing file: If enabled, includes the macro definitions in the listing file. Enabled by default, but you may want to disable it to avoid cluttering up your listing.

`--list-macros` p. ??

Listing file: If enabled, includes the output of invoked macros in the listing file. Enabled by default.

`--list-margin=`*`NUMBER`* p. ??

Listing file: Specifies the width of the left column of the listing file, where the machine instructions are shown. Defaults to a target-dependent value.

`--list-messages` p. ??
Listing file: If enabled, error and warning messages are included at the top of the listing file as well as written to the terminal. Enabled by default.

`--list-number-format=`*NUMBERFORMAT* p. ??
1k target: Specifies the number format to use for fractional constants in the listing file. Possibilities include *frac*, *dec*, *hex*, and *hexfrac*. Defaults to *frac*.

`--listfile=`*FILENAME* (`-a` *FILENAME*) p. 20
Global: Specifies the filename of the listing file to create. If `FILENAME` is `-`, the listing file is written to the terminal. If `FILENAME` is `=`, an appropriate filename is chosen for you.

`--loop-iterations=`*NUMBER* p. ??
Assembler: Specifies how many times a `.for`, `.while`, or `.repeat` block may loop before `bkasm` thinks that you're stuck. Defaults to 100.

`--macro-depth=`*NUMBER* p. ??
Assembler: Specifies how deeply macros may recurse before `bkasm` gets suspicious. Defaults to 64.

`--max-errors=`*NUMBER* p. 22
Global: Specifies how many error messages `bkasm` will generate before it bails out. Defaults to 20.

`--message-width=`*NUMBER* p. 23
Global: Specifies the line-wrapping width in characters for error and warning messages. If set to zero, no line-wrapping will be performed. Defaults to 78.

`--optimize` (`-O`) p. 21
Global: Enables the standard set of optimizations.

`--optimize-all` p. 21
Global: Enables all optimizations.

`--optimize-load-placement` p. 40
1k target: Allows assignments to the **b** register to be moved to an optimal location.

`--outfile=`*`FILENAME`* (`-o` *`FILENAME`*) p. 20
Global: Specifies the output filename(s) to create. The exact behavior depends on the output format used. If not given, an appropriate filename will be chosen for you.

`--output-asm-define-byte-instruction=`*`INSTRUCTION`* p. ??
asm output format: Specifies the pseudo-op that the host assembler uses for declaring a table of literal byte values. Defaults to "`db`".

`--output-length-little-endian` p. ??
c/asm output formats: If `--output-length-word` is enabled, the length word is written bytewise little-endian instead of big-endian.

`--output-length-word=`*`SIZE`* p. ??
c/asm output formats: Prefixes the object code for each section with a word indicating the number of instructions in the section. `SIZE` may be one of: *none*, *byte*, *short*, *long*, *longlong*, *quad*, or an explicit number of bits.

`--output-little-endian` p. ??
c/asm output formats: If enabled, the object code is written bytewise little-endian instead of big-endian.

`--output-prefix` p. ??
c/asm output formats: Specifies a common prefix to use for the names of the arrays for each section. If not given, an appropriate prefix is chosen for you.

`--output-top-section` p. ??
Global: Forces `bkasm` to output an object code array representing all sections together.

`--outsection=`*`SECTION`* (`-S` *`SECTION`*) p. 21
Global: Adds to the list of sections that will be output.

`--piggyback-literal-instructions` p. 39
1k target: If enabled, machine instructions written directly in the source file are susceptible to piggybacking.

`--precision=`*`NUMBER`* p. 79
Assembler: Specifies the default precision value used by `.pequ` and `.pset`. Defaults to 0.

`--remember-options` p. 64

Global: If enabled, all other options are carried over into subsequent source files.

`--show-error-name` p. 22
Global: If enabled, errors and warnings messages will be prefixed with their internal names.

`--show-message-line` (`-n`) p. 21
Global: If enabled, errors and warnings are followed with the line of source code responsible for the message.

`--target=`*`TARGETNAME`* (`-T` *`TARGETNAME`*) p. 21
Global: Specifies the target to use. Defaults to *1k*.

`--warn` (`-W`) p. 21
Global: Enables the standard set of warnings.

`--warn-all` p. 22
Global: Enables all warnings.

`--warnings-are-errors` p. 22
Global: Treats warnings as if they were error messages.

Appendix E

Mathematical Functions

Any Perl function may be used in a `bkasm` expression. The following are some of the more useful functions for mathematical manipulation. All of these functions accept complex arguments, and may return complex results when appropriate. All trig operations are in radians.

`Im(x)`	Imaginary part of x
`Re(x)`	Real part of x
`abs(x)`	Absolute value of x
`acos(x)`	Arc cosine of x
`acosec(x)`	Arc cosecant of x
`acosech(x)`	Hyperbolic arc cosecant of x
`acosh(x)`	Hyperbolic arc cosine of x
`acot(x)`	Arc cotangent of x
`acotan(x)`	Arc cotangent of x
`acotanh(x)`	Hyperbolic cotangent of x
`acoth(x)`	Hyperbolic cotangent of x
`acsc(x)`	Arc cosecant of x
`acsch(x)`	Hyperbolic arc cosecant of x
`arg(x)`	Polar argument of x ($-\pi$ to π)
`asec(x)`	Arc secant of x
`asech(x)`	Hyperbolic secant of x

`asin(x)`	Arc sine of x
`asinh(x)`	Hyperbolic arc sine of x
`atan(x)`	Two-quadrant arc tangent of x ($-\pi/2$ to $\pi/2$)
`atan2(y,x)`	Four-quadrant arc tangent of y/x ($-\pi$ to π)
`atanh(x)`	Hyperbolic arc tangent of x
`cbrt(x)`	Cube root of x
`cos(x)`	Cosine of x
`cosec(x)`	Cosecant of x
`cosech(x)`	Hyperbolic cosecant of x
`cosh(x)`	Hyperbolic cosine of x
`cot(x)`	Cotangent of x
`cotan(x)`	Cotangent of x
`cotanh(x)`	Hyperbolic cotangent of x
`coth(x)`	Hyperbolic cotangent of x
`csc(x)`	Cosecant of x
`csch(x)`	Hyperbolic cosecant of x
`exp(x)`	e^x
`ln(x)`	Natural logarithm of x
`log(x)`	Natural logarithm of x
`log10(x)`	Base-10 logarithm of x
`logn(x,n)`	Base-n logarithm of x
`rho(x)`	Absolute value of x
`rootk(x,n,k)`	kth n-root of x
`sec(x)`	Secant of x
`sech(x)`	Hyperbolic secant of x
`sin(x)`	Sine of x
`sinh(x)`	Hyperbolic sine of x
`sqrt(x)`	Square root of x
`sqrt(x)`	Square root of x
`tan(x)`	Tangent of x
`tanh(x)`	Hyperbolic tangent of x
`theta(x)`	Polar argument of x ($-\pi$ to π)