B Language Reference Manual

Thinkage Ltd.
85 McIntyre Drive
Kitchener, Ontario
Canada N2R 1H6
Copyright © 1998 by Thinkage Ltd.

Contents

1. Introduction

This manual describes the B programming language as accepted by the GCOS8 B compiler.

B is a descendant of the programming language BCPL. B was first designed and implemented by D.W.Ritchie and K.L.Thompson of Bell Telephone Laboratories, Inc., Murray Hill, N.J. The original implementation of the run-time package was done by S.C.Johnson, also of Bell Labs.

There are several differences between this version of B and Bell Laboratory versions. The **switch** statement has been extended. Floating point operators and proper logical operators have also been been added. Finally, the order in which operators are evaluated has been changed.

"media code" found in the GCOS8 environment. The run-time package runs under GCOS8.

1.1 Overview

Although this manual makes an effort to explain most of the concepts used here, we assume that the average B user is already familiar with some computer terminology and that the user knows at least one other computing language (Fortran, Cobol, PL/I, etc.). New B users are warned that this *is* a Reference Manual and not a Beginner's Guide. Beginners may wish to start with the B Tutorial Guide before reading this manual.

The manual begins with a discussion of the various types of constants used in B. From there it moves on to the use of variables and vectors in the language. Next is a chapter on functions, followed by chapters on the various types of statements accepted by B, and the wide variety of operators in the language.

The final chapters deal with the B Library routines, I/O, and a number of miscellaneous details about working with the compiler. You should read these last chapters as carefully as you read the chapters about the basics of the language. It is virtually impossible to use B without a working knowledge of the run-time library routines. For one thing, B itself has no constructs for input or output; all I/O is done by function calls to library routines.

Before you begin your first major program in B, you should try a few smaller programs just to get the feel of the language. It is also very helpful to have a look at programs written by a more experienced B programmer, to get an idea of how the features of the language can be put together effectively.

However carefully we explain things in this manual, it's inevitable that we'll leave some of your questions unanswered. One of the best ways to get the answers you want is simply by experimentation. If you don't know how a given feature works, write a small program and try things out. This may take a little time, but it can also save a lot of the headaches which arise from guessing about how the language behaves.

2. Constants

B lets you define octal, decimal, floating point, ASCII character, BCD character, and string constants. All but the last of these are stored internally in a single machine word. On GCOS8 systems, a machine word is 36 bits (which is equivalent to four 9-bit ASCII characters or six 6-bit BCD characters).

2.1 Escape Sequences

Because some terminals do not have all the characters recognized by B, the compiler lets you "manufacture" these missing characters using two characters which *do* appear on your keyboard. For example, B translates `"$("` into the single curly brace bracket `"{"`. Pairs of characters that can be translated into single characters are known as *escape sequences*.

Escape sequences may be used in character constants and string constants for characters which may be difficult or impossible to enter any other way. For example, `"*n"` may be used in character and string constants to represent the "new-line" character, `"*b"` is a backspace character, and so on. Remember that

Appendix A gives a complete list of escape sequences.

2.2 Decimal Constants

A decimal constant consists of an ordinary integer number with no leading zeroes, as in

```
25    4737    981    32
```

2.3 Octal Constants

An octal constant consists of an integer number formed only from the octal digits zero through seven. To distinguish an octal constant from a decimal constant, an octal constant must begin with at least one leading zero, as in

```
01    077    026    0004000    0777777777777
```

2.4 Floating Point Decimal Constants

A floating point constant is any number containing a decimal point. It must not begin with the decimal point, but it may have leading zeroes. It may also be followed by the letter `'e'` and a signed integer exponent. (If the sign of the exponent is positive, the `"+"` may be omitted.) Here are some sample floating point constants:

```
3.2    1.    0.5    1.e5    001.5    4.987e-2
```

2.5 ASCII Character Constants

An ASCII character constant consists of one to four ASCII characters inside single quotes. The result is a word that contains the internal form of the ASCII characters. If there are less than four characters specified, the characters are right-justified in the word and padded on the left with ASCII null characters (000, escape sequence `'*e'`). Thus `'a'` is the same as `'*e*e*ea'`. Here are some sample ASCII character constants:

```
'a'    'abc'    'abcd'    'ab*'*n'
```

In the last example above, the constant is `"ab"` followed by a single quote followed by a new-line character. This is because `"*'"` is the escape sequence for a single quote in character constants, and `"*n"` is the escape sequence for the new-line character. Thus the constant is only four characters long, even though it is typed with six characters.

The compiler counts the number of characters inside character constants and issues an error message if there are more than four.

2.6 BCD Character Constants

word containing the characters transliterated to BCD, right-justified, and left-padded with BCD zeroes (zero bits). Thus `a` is equivalent to `00000a`. Characters which do not have exact equivalents in the BCD set are converted to BCD blanks. Here are some sample BCD character constants:

```
`a`    `ot`    `123456`
```

Note that lowercase characters are equivalent to uppercase characters in BCD constants.

If your terminal does not have the grave accent character, you may write a BCD constant as an ASCII character constant preceded by a dollar sign, as in

```
$'a'    $'ot'    $'123456'
```

The run-time package provides functions to convert BCD to ASCII and vice versa.

2.7 String Constants

A string constant is any sequence of ASCII characters enclosed in double quotes, as in

```
"this is a string"
""
"the above is the null string"
"this is a line*nand another line*nand another"
```

The `"*n"` new-line characters break up the last string into three lines.

When processing a string, B packs the string four characters to a word. The compiler marks the end of the string by appending one extra character, an ASCII null (000, string escape sequence `'*e'`). Thus the string `"abc"` is stored internally as `"abc*e"`.

The *value* of all other constants (decimal, octal, floating point, ASCII and BCD characters) is a single word containing the internal representation of the given constant. Obviously, the same thing cannot be true of string constants since most strings are too long to fit in a single four-character word. For this reason, the *value* of a string constant is a single machine word containing the memory *address* of the actual string. This is very important. If you were to say

```
A = "this string";
```

in a B program, the variable `A` receives the address in memory where `"this string"` is stored. `A` does not get the string itself. Thus if you printed `A` without being careful, you would be printing an essentially meaningless address, not the string you wanted. Chapter 4 explains this in more detail.

When constructing a string, the compiler gobbles all the characters it sees, translating escape sequences if necessary, until it finds an unescaped double quote to end the string. If the compiler encounters any true new-line characters (as opposed to `'*n'` escape sequences) it does one of the following things.

- If the new-line is preceded by a `'*'`, both the `'*'` and the new-line are thrown away. This lets you enter a very long string by breaking it up over several lines of source.
- If the new-line is not preceded by a `'*'`, the new-line character is kept. However, a warning message is issued, since this situation usually arises when you have forgotten the double quote which closes the string. To get a new-line in a string without a warning, use the escape `'*n'`, as in

```
     long string without a new-line";
b = "this is a very*n*
     long string that contains a new-line";
c = "this will get
     a warning";
```

When a string is broken over several lines as in the examples above, the compiler skips over any blanks and tab characters which appear at the beginning of each new line. Thus `c` above is equal to `"this will get*na warning"`; the spaces before `"a warning"` are not included in the string being collected. If you want to include spaces at the beginning of one of these lines, use the `'* '` escape sequence for the first space in the sequence. For example,

```
d = "this string is two lines*n*
*    with spaces";
```

points `d` towards the string `"this string is two lines*n with spaces"`.

3. Data Objects

In a language like FORTRAN, variables are always of one data type or another and you can only do certain operations with certain types of variables. For example, FORTRAN won't let you add a floating point number to a character, or perform logical operations on integers.

B, on the other hand, is a *typeless* language. The compiler doesn't keep track of whether variables refer to integers, characters, octal numbers, and so on; that is left up to you. You can subtract the letter `'a'` from `1.5` without getting an error in B...of course, the answer won't mean much, but you can do the subtraction.

This typeless nature of B gives you a great deal of freedom. Naturally, it gives you a certain amount of responsibility too; it's up to you to make sure that the operations you are asking B to perform make sense. For example, when you are multiplying two variables with the floating point multiplication operator, you are responsible for ensuring that the variables really are floating point numbers. If they aren't, the multiplication still takes place, but the result is bound to be useless.

B really only has two types of data objects: words and vectors of words. The rest of this chapter describes these different objects, as well as a special kind of construct called a *manifest*. First though, we must talk about the names that data objects may have.

3.1 Identifiers

In B, an identifier or name is formed from the characters `a-z`, `A-Z`, `0- 9`, the underscore (_), and the dot or period (`.`). The first character of an identifier cannot be a digit. Since the run-time package uses names which contain one or more dots, you can prevent name conflicts by avoiding the use of dot in any names defined in your programs.

Names in B may be arbitrarily long, but only the first eight characters are significant. Thus the compiler believes the names `function1` and `function2` refer to the same thing. Furthermore, the GCOS and TSS loaders only pay attention to the first six characters of a name. For this reason, the first six letters of all function names and external variable names should be different from one another. (Chapter 4 discusses functions and external variables.)

etc. all refer to the same thing. When you can compile a program, you can tell B to pay attention to the case of letters by specifying the proper option in the command line which invokes compilation (see Section 9.1). Even with this option, however, B ignores case distinctions in external names (external names are discussed in Chapter 4).

3.1.1 Keywords

There are 15 keywords in the B language. These may not be used as identifiers in B programs. Be especially careful that you don't inadvertently use **next** as one of your variable names.

The keywords of B may be listed in groups as follows:

Identifier scope keywords:
    auto extrn
Execution flow control keywords
    if else for while repeat switch do
Transfer keywords:
    return break goto next
Switch statement keywords:
    case default

If you are using the option which tells B to pay attention to the case of letters, the keywords above must be written in lowercase. Otherwise, the case of the keywords makes no difference to the compiler.

3.2 Simple Variables

A simple variable is a single word of memory with an associated name. As we have pointed out before, B does not keep track of what kind of data you have stored in this word. Thus you can say

```
a = 'm';
b = 5;
c = a + b;
```

without getting an error. The above code places the value of the constant `'m'` in the memory location referred to by the name `a`, places the integer `5` in the memory location of `b`, and then adds the letter to the integer. The result is put into the memory location of `c` and can be thought of as the character `'r'`, the octal number `0162`, or the decimal integer `114`, depending on how you want to use it.

3.3 Vectors

If you have experience with other programming languages, you are probably familiar with the idea of a vector. However, the actual implementation of vectors in B is quite different from vectors in most other languages, and so we will go to some effort to explain the concepts involved.

At its simplest, a vector is just a collection of consecutive words in memory along with an associated name. The name of the vector does *not* refer directly to the vector's data; instead, the vector name refers to a single word of memory which in turn contains the address of the first word of the vector. (We sometimes call this address the *vector pointer*.) It is very important to keep this distinction clear in your

```
vec = 5;
```

does *not* set any of the words in the vector to the value `5`. Instead, the statement puts the integer `5` into the single word of storage which previously held the address of the vector. Unless you've saved this address somewhere else, the statement above has just written over the only way you had of finding your vector.

The simplest way to access an element of a vector is to use the name of the vector followed by an expression in square brackets. This is known as *subscripting*. For example,

```
vec[10]
```

refers to the tenth word of the vector `vec`. B calculates this address by adding the number `10` to the address of the vector contained in the vector pointer `vec`. Thus `vec[0]` refers to the start of the memory pointed to by `vec`, `vec[1]` refers to the next word in memory, and so on.

Whenever B sees a subscript in square brackets, it adds the value of the subscript to the value of the vector pointer and uses the result as the address of a vector element. If your subscript is in the variable `i` you can get the `i`*th* element of a vector `a` by writing

```
a[i]
```

Surprising though it may seem to those used to other languages, you can write exactly the same thing as

```
i[a]
```

since in either case B adds the contents of `i` and the contents of `a` to get an address. Because B is typeless, it makes no effort to check whether `a` or `i` is actually the name of a vector. As far as the compiler is concerned, they are the names of single words in memory.

B does *not* check to see if an index has gone past the end of a vector. For example, even if you have only defined vector `vec` as 20 words long, you can still talk about `vec[21]`, `vec[30]`, or `vec[100]` without getting an error message. Of course, this is a dangerous practice, since it's hard to tell what lies in memory beyond the end of a vector. You may be looking at garbage, you may be looking at storage for another variable or vector, or you may be looking at some of the internal workings of your program. If you go too far past the end of your vector, you may even try to look at memory that hasn't been allocated to your program and end up with a memory fault error from the system. In B, it is the programmer's responsibility to make sure that indices don't run off the end of a vector.

Generally speaking, B treats every word in a vector as a separate element. Sometimes though you want to simulate an array of records, each of which is several words long. It is usually very simple to perform this simulation. For example, suppose `xxx` points to a vector which is made up of "records" which are three words long. Then

```
xxx[3 * i]
```

accesses the first word of the `i`*th* record in the vector. B simply multiplies `i` by three and adds this to the address in memory location `xxx`. To access the second word of the `i`*th* record you could type

```
xxx[1 + 3 * i]
```

since B performs the multiplication before the addition.

subscripts with a vector name as you want. For example, if `x` is the name of a vector, the construct

```
x[i][j]
```

is interpreted as follows. The value of `i` is added to the vector pointer `x` to get the address of a word in memory. The contents of that word are then added to `j` to get a second address and that second address is the one to which `x[i][j]` refers. Note that in this way, B is assuming that the elements of the vector `x` are such that adding `j` to them gives a meaningful address. In Section 4.1, we describe a way in which the elements of a vector can be initialized as pointers to other vectors. Thus `x[i][j]` would be the `j`*th* element of the vector pointed to by `x[i]`. Using constructs like this, you can manipulate an array of any dimension.

3.4 Manifest Constants

A *manifest constant* (or more briefly, a *manifest*) is not really a data object at all. A manifest is just a symbol which can be used in the compilation process to stand for a string of characters.

A manifest constant is defined in a statement of the form

```
name = text;
```

where `name` is any valid identifier, and `text` is simply a collection of any characters except `';'`. Some examples of manifests are

```
SIX7 = 0777777;
VECSIZE = 10;
B = VECSIZE + VECSIZE;
TWICE = 2 * ;
```

As in all identifiers, the letters in manifest names may be upper or lowercase. However, the common convention is to write all manifest names in uppercase. This convention helps you to distinguish manifests from variables when you are looking at the source of your program.

When a manifest is defined, the compiler enters the name in its symbol table and stores the associated text in an internal buffer. Absolutely no processing is done on the text at the time of definition.

When the compiler reads an identifier during compilation, it checks to see if the identifier is a manifest. If so, the identifier is replaced by the text associated with it. This replacement does not take place if the manifest name occurs inside a string or character constant.

Because manifests are changed into text in this way, manifests cannot be redefined. If you were to attempt

```
VECSIZE = 10;
...
VECSIZE = 20;
```

the second statement would be changed into

```
10 = 20;
```

which is a syntax error. The compiler replaces manifest names with their corresponding text values *before* analyzing the syntax of the line.

a manifest must be defined before it is used in any other statement. The exception is when a manifest occurs in another manifest definition. Since B stores the text associated with a manifest name without analyzing the text, other manifests appearing in the text need not be defined at the time. However, if one manifest contains a second manifest in its text, the second manifest must be defined by the time the first manifest is used in a B statement (otherwise B doesn't know what to replace the second manifest with). The safest approach is to define all your manifests at the very beginning of your B programs. The B compiler lets you nest up to 10 levels of manifests inside other manifests.

Manifests are just names for strings of text and don't really represent data objects which have memory space allocated to them. For example, consider

```
A = 10;
B = A + A;
C = B * B;
```

When `C` is used in a program, it is expanded to

```
C = A + A * A + A;
```

and substituting for `A`

```
C = 10 + 10 * 10 + 10;
```

Thus `C` is actually the number 120, not 400 as you might have thought if you said to yourself that `B` was equal to 20 and `C` equals `B * B`. Because of this kind of problem, it is usually a good idea to put parentheses around most manifest constants so that they are evaluated in the way you expect; thus you might say

```
B = (A + A);
```

Manifests are used as convenient shorthand symbols in B programs. For example, `SIX7` is easier to type than the octal constant `0777777` (and easier to read too). With `TWICE` standing for `2*`, the expression

```
TWICE i
```

stands for `2* i`. It is very common to use a manifest like `VECSIZE` for the length of a vector. In this way, you can change the length of a vector just by changing the manifest `VECSIZE`, rather than going through your source and changing every occurrence of your vector length. You can also use manifests to define structures in a vector. For example, if `vec` is a vector whose records are three words long, you can define

```
WORD1 = 3 * ;
WORD2 = 1 + 3 * ;
WORD3 = 2 + 3 * ;
```

In this way, `vec[WORD1 i]` is the first word of record `i`, `vec[WORD2 i]` is the second word, and `vec[WORD3 i]` is the third.

To end this section, we give an example where manifests are used in a program that prints a binary tree. The tree is represented by a vector of records which are each three words long. The first word of each element gives the contents of a node, the second word is the address of the node's left descendant, and the third word is the address of the node's right descendant. If a node doesn't have a left or right descendant, the descendant pointer is set to `-1`.

this manual. Still, you should be able to appreciate what this program does, even if you aren't familiar with all the details.)

```
/* binary list structure */
NULL = -1;
CONTENTS = 0;
LEFT_PTR = 1;
RIGHT_PTR = 2;
   ...
printree( ptr )
     if( ptr != NULL) {
          printree( LEFT_PTR[ptr] );
          print_contents( CONTENTS[ptr] );
          printree(RIGHT_PTR[ptr] );
     }
     /* end printree */
```

4. Functions

A program written in B can contain three kinds of components:

- Manifest constant definitions;
- External variable definitions;
- Function body definitions.

A program can contain any number of each kind of component, and components can occur in any order, provided that manifests are defined before they are used in other program components.

Chapter 3 discussed manifest constant definitions. An external variable definition defines a simple variable or vector which can be used by any function in your program, provided the function explicitly declares its intention to use the external variable with an **extrn** statement. (**extrn** is described in Section 5.4.1). An external definition automatically allocates sufficient static memory storage for the simple variable or vector.

A function definition defines a piece of the executable code of a program. Since all executable code must appear inside a function body, functions can be thought of as the building blocks of any B program. A function definition includes the name of the function, the arguments it accepts, and the statements which determine what the function actually does.

4.1 External Variables

External variables are the only form of "global" variables in B. We begin with the possible forms of external variable definitions, then look at several examples.

In the following external definition forms, *ival* stands for "initialization value". This may be any valid constant, constant expression, or manifest. (A *constant expression* may be a string constant or any valid combination of character or numeric constants, binary operators, unary operators, and parentheses. Chapter 6 gives the rules for forming constant expressions from constants and operators.)

The initialization value *ival* may also be the name of an external variable or function. In this case, the value of *ival* is taken to be the memory address of the variable or function. If the name given is the name

itself. Thus if a variable is initialized to `vec` and if `vec` is a vector, the variable holds the address of the single word `vec` which in turn holds the address of the actual vector.

Here are the possible forms of external definitions:

`name;`
: allocates a single word of memory for `name`.

`name { ival };`
: allocates a single word of memory for `name` and initializes that word to *ival*.

`name { ival, ival, ... };`
: allocates space for as many words as there are *ivals* and initializes the words with the *ivals*. Only the first word is associated with the identifier `name`; the other *ivals* are assigned to successive words of storage following `name`.

  In effect, this defines a vector which does not have a word set aside as a vector pointer. The symbol `&name` stands for the address of `name`. Consequently, `(&name)[0]` refers to the contents of the zero*th* word of the vector, `(&name)[1]` the contents of the first word, and so on. `name` on its own refers to the first *ival* given. This is the way a vector is set up in FORTRAN, but it is not the same as a B vector. (Note that the external definition form `name { ival };` is just a degenerate case of this kind of external definition.)

`name [ const-expr ];`
: is the first of several forms of B vector declarations. `name` refers to the vector pointer, the single word in memory which holds the address of the first word allocated to the vector. The total number of words allocated to the vector is equal to the value of the given constant expression plus one. Thus a declaration like

```
vec [10];
```

  allocates eleven words of storage for `vec`, so that your indices can run from zero to ten.

  The `const-expr` in brackets may be any valid combination of numeric or character constants, unary operators, binary operators, and parentheses. You must make sure the value of the expression is reasonable, since B accepts absurdities here like floating constants and negative numbers. For practical purposes, `const-expr` must give an integer result.

`name [] { ival, ival, ... };`
: `name` is defined as a one-word pointer to a vector. The length of the vector is the number of initial values, and the elements of the vector are initialized as indicated.

`name [ const-expr ] { ival, ival, ... };`
: `name` is defined as a one-word pointer to a vector. The length of the vector is either the value of the constant expression plus one or the number of initial values; the maximum is chosen. The contents of those vector elements which do not have corresponding initial values are left undefined.

For compatibility with a previous version of the compiler, B also accepts an *ival* or *ival* list which is not surrounded by braces. In this case, the compiler does not permit a constant expression to appear. Only a numeric, character, or string constant is acceptable, although an integer constant may be prefixed by a minus sign.

Here are some examples of external definitions:

One word of storage is allocated, initialized to the decimal constant 10, and associated with the name `a`. The statement `z=a;` gives `z` the value 10.

```
b [ ] { 'ab', 'cde', 'fghi' };
```

`b` is defined as a one-word pointer to a vector that is three words long. The first element of the vector, `b[0]`, is initialized to the character constant `'ab'`. The other two elements, `b[1]` and `b[2]`, are initialized to `'cde'` and `'fghi'`, respectively. The assignment statement `z=b;` gives `z` the address of the vector `b`; thus `z[0]` is `'ab'`, `z[1]` is `'cde'`, and `z[2]` is `'fghi'`.

```
c { 'ab', 'abc' };
```

`c` is defined as a single word and initialized to the value `'ab'`. The word immediately following `c` in memory is initialized to the constant `'abc'`, but is not associated with any name. In effect, this defines a vector which does not have a word pointing to it. The statement `z=c;` sets `z` to `'ab'`. The statement

```
z = (&c)[0];
```

also sets `z` to `'ab'` and

```
z = (&c)[1];
```

sets `z` to `'abc'`.

```
z = &c;
```

gives `z` the address of `c` and therefore makes `z` a pointer to the vector which begins at `c` (so that `z[0]` is `'ab'` and `z[1]` is `'abc'`).

```
d[63];
```

`d` is defined as a single word which contains a pointer to a vector of 64 words.

```
e[10] { a, b, c, d };
```

`e` is defined as a one-word pointer to a vector of 11 words. Words zero, one, two, and three are initialized with the addresses of the externals `a`, `b`, `c` and `d`, respectively. The contents of the remaining elements are undefined.

```
f { "a string" };
```

This is the usual way of defining an external string with an initial value. `f` is defined as a one-word a pointer to the place where the string constant `"a string"` is stored. The statement

```
z = f;
```

gives `z` the same pointer to `"a string"`.

```
g[] { "pascal/library", "pascal/compiler", -1 };
```

This sets up a vector of strings with an end marker. It defines `g` as a one-word pointer to a vector three words long. The word `g[0]` is initialized as a pointer to the storage occupied by the string constant `"pascal/library"`. The word `g[1]` is initialized as a pointer to the string `"pascal/compiler"`, and the final word `g[2]` is initialized to the decimal constant `-1`. This shows that the elements of a vector need not all have the same "type".

As mentioned in the last chapter, B has no explicit facilities for handling arrays of more than one dimension. Usually if you need more than one dimension, you build it during execution by calling the library function `GETMATRIX`. `GETMATRIX` obtains storage, constructs the necessary edge vectors, and returns a pointer to the array (see Chapter 7 for more on array-handling library functions).

initialized external. The secret is that any *ival* (inside braces) may be replaced by an *ival* or *ival* list surrounded by braces. The compiler then constructs the *ival* list and places a pointer to it in the original *ival* list. For example,

```
x[ ] {
       { 00, 01, 02 },
       { 10, 11, 12 },
       { 20, 21, 22 }
     };
```

puts this feature to use. In this case, `x` is initialized as a pointer to a vector containing three pointers. Each pointer points to a vector of three words. In an expression, the value of `x[0]` is a pointer to the first vector of three words, while the value of `x[1][2]` is 12.

The maximum depth to which these array-like initializations can be nested is seven levels.

4.2 Function Definition

B functions are similar in purpose to subroutines in FORTRAN and procedures in PL/I. Every working B program must contain one function called `main`: the function where execution of the program begins. Most B programs contain a number of other functions as well, since B is designed to encourage structured or "modular" programming.

The general form of a function definition is

```
name( arg1, arg2, ... ) statement
```

The `name` must be a valid identifier and is automatically defined by the compiler as an external (and therefore "global") symbol. Any function in a B program may call any other function; you can even call `main`. A function may also call itself recursively (discussed later in this chapter).

The argument list shown above need not contain any arguments at all. If it does contain arguments, they are given as a list of legal identifiers separated by commas. These arguments are "local" or **auto** variables; in other words, storage is allocated to these variables for the duration of the function's operation. (Section 5.4.2 discusses such variables in more detail). Once the function is finished, this storage is released and the local variables "disappear".

(To be precise, these variable values are *not* automatically wiped out when a function is finished. However, the storage they have been using is made available for other purposes so you can't really count on the storage remaining the way it was when the function was still executing.)

The `statement` in the function definition above defines what actions the function takes. Most of the time, this is a compound statement consisting of a number of statements enclosed by brace brackets. Chapter 5 gives the rules for forming statements.

4.3 Mechanics of Function Calls

When one function calls another, the arguments (if any) are always passed *by value*. This means that altering an argument inside a function has no effect on the value of the argument passed by the calling function. However, if one of the arguments passed is the *address* of a variable, the function which

caller's variable. For example, consider a call like

```
func (vec);
```

where `func` is a function and `vec` is a vector. `func` receives the value of `vec` which is the address of the beginning of the actual vector. Using this address, `func` can change the values of the vector elements. However, `func` cannot change `vec` itself, since `func` receives only the value of `vec`, not the location in memory where that value is stored.

A function may return a one-word value to its caller any time during its operation using the **return** statement (see Section 5.5.2). The caller and the callee do not have to agree on whether or not a function returns a value. If a value is returned but not expected, the value is ignored. If a value is expected but not returned, the value received by the caller is garbage.

A function can determine how many arguments it has been passed by its caller by invoking the library routine `nargs`. For example, the statement

```
x = nargs();
```

sets the variable `x` to the number of argument words used to call this invocation of the function. Because a function has this way to determine how many arguments it has been passed, B lets you write functions that take a variable number of arguments. Most of the time, this means that a function is called with fewer arguments than are defined for it in the function definition. In this case, the function has to determine the actual number of arguments it has received and establish default values for the arguments it does not have. If a caller specifies more arguments than a function needs, the surplus arguments are ignored.

4.3.1 Use of the Stack

Function calls in B are handled by use of an internal *stack*. This is a large block of memory which is used for storing various types of information.

When a function is called, internal information about the call is stored on the stack. Storage for the local or **auto** variables is also allocated on the stack. If a function calls itself recursively, the **auto** variables of the new invocation are again stored on the stack, so that the new function has its own set of local variables to work with. (Naturally there is only one copy kept of any external variable, and storage for external variables is *not* allocated on the stack.) Because B operates this way, you can nest functions (recursively or otherwise) to as many levels as your stack space allows. If you perform too many function calls, you fill up your stack area and run into memory used by your program for other purposes. Obviously, this can be a dangerous situation.

By default, the compiler allocates 500 words of memory as stack space. This is ample storage to handle a reasonable number of function calls. If this stack-size is not sufficient (or if you want to reduce this stack-size to reduce your memory requirements), you can specify a different stack- size in the command that compiles your program (see Chapter 9).

When a function returns to its caller, the stack space that was used by that function is released. The next time a function is called, it uses the stack space that belonged to the previous function (or at least a portion of it). For this reason, you cannot expect to be able to use the contents of an **auto** variable after the function that used the variable has finished operation: even if you save the variable's address, the

This section has been a very brief introduction to the mechanics of function calling, but it should be enough for the average user. Those who wish to know more about this process should read the B Environment Manual ("expl b environment manual"). This manual is of a highly technical nature; it requires a knowledge of GMAP and other GCOS8 features.

5. Statements

Statements are used to define the actions to be taken by a B function. Statements may only appear in the body of a function definition. In certain cases, a complete statement may occur inside another statement.

Many types of statements may contain expressions. Since the rules for formulating expressions are discussed in the next chapter, we will merely note here that an expression may be a statement, but a statement may not appear inside an expression.

In every case where a statement is permitted, it may be replaced by a *compound statement* consisting of one or more statements enclosed in curly braces, as in

```
{
    statement1
    statement2
        ...
}
```

The compiler does not permit a null compound statement like

```
{ }
```

All statements, except compound statements, must end with a semicolon.

When B is compiling statements (and everything else for that matter), it treats its input as an unbroken stream of characters. Formfeed, tab, and new-line characters are all converted to space characters, except where they occur inside string or character constants. New-line characters are counted so that the compiler can tell you the line number of a line in which an error occurred. Lines may be of any length. Unlike some other compilers, B does not recognize any part of a line as a "sequence field" (although it does accept line numbers as described in Section 9.3).

In the formal definitions which follow, keywords are printed in bold face. When parentheses are shown in a definition, they are required. By convention, the word "statement" stands for a B statement ended by a semi-colon, or else a compound statement enclosed in braces as described above.

5.1 Comments

Comments are not true statements: they can be included anywhere in a B program that a space could be used, inside or outside of function definitions. The beginning of a comment is signalled by `/*` in the input stream. The compiler ignores everything in the input stream until it encounters `*/` to end the comment. For example,

```
/*
 * This is a comment.
```

is a comment that takes up three lines of input.

Comments may *not* be nested. For example,

```
/* This comment /* has a comment inside it. */ */
```

causes a syntax error.

5.2 Null Statement

```
;
```

The null statement does absolutely nothing. It is typically used to supply a null body to a **while** statement, as in

```
while( putchar( getchar() ) );
```

or to provide a convenient place on which to hang a label as in

```
label: ;
```

5.3 Expression Statement

```
expression;
```

Any valid B expression followed by a semi-colon is acceptable as a statement. To be meaningful, the expression usually involves an assignment operation or a function call, as in

```
x = min(a,b) + x;
open( "/myfile", "r" );
++i;
```

However, the compiler happily accepts statements which do absolutely nothing, such as

```
a < b;
open;
i;
```

Remember that in B, assignment is an operator in an expression, not a statement.

5.4 Storage Declaration

Before discussing the statements which pertain to storage allocation or memory referencing, we must briefly review what we have said about the various types of data objects used in a B program.

External variables are stored in a global pool of memory which is accessible to any function in your B program. If a function intends to use a particular external variable, it must state its intentions by naming the variable in an **extrn** statement.

The local or **auto** variables of a function are allocated storage on the stack each time the function is called. When the function returns, the storage allocated for local variables is released for use in future

Constants used inside functions are allocated storage in the area of memory used by the executable code of the function. The compiler does not accept constructs that would result in directly changing a constant's value. Thus the storage used by constants should be considered "read-only".

Labels are also stored in the body of a function's executable code, and are not directly accessible to the user.

Finally, there is a pool of free storage which can be dynamically allocated by the library function GETVEC and dynamically released by the library function RLSEVEC (described in Section 7.3). This free storage area automatically grows as required up to the limits imposed by the operating system.

Identifiers used in the body of a function must be formal arguments of the function, labels, or variables which have been explicitly declared in **extrn** or **auto** statements within the function itself. The only exception is a function name used in a function call. The compiler automatically classifies any name immediately followed by a left parenthesis as external unless the name has already been classified in some other way.

auto and **extrn** statements may appear anywhere in a function body. However, it is highly recommended that you group both types of statement at the beginning of the function.

5.4.1 The extrn Statement

```
extrn name1, name2, ... ;
```

This statement declares the names of one or more external variables which the function intends to reference. Once the statement has been issued, the function may use any of the external variables named. Although an identifier may be externally declared as a vector pointer, you should not indicate this in the **extrn** statement. All the function needs to know is that the one-word vector pointer is an external variable. The actual elements of the vector can be obtained using subscripts with this single word, as in

```
extrn vec;
vec[1] = 1;
```

5.4.2 The auto Statement

```
auto name1, name2[const-expr], ... ;
```

The **auto** statement declares local storage unique to each invocation of a function, as in

```
auto x;
auto i, j, x[10];
```

To create a local vector, you must specify the vector's name and length in an **auto** statement as shown above. The length that you specify for the vector must be a constant expression since it is established at compile time. Because **auto** vectors are allocated storage on the stack, **auto** vectors should not be overly large; otherwise, you may use up all your stack space in just a few function calls. For vectors longer than 64 words, it is general practice to use GETVEC to obtain storage from the free storage area, rather than declaring the vectors as **auto** and taking up room on the stack.

The `const-expr` above is any valid combination of numeric or character constants, unary operators,

nonsense constructs like

```
auto x[-1];
```

which lead to unwanted results. Usually, array dimensions are either simple integer constants or expressions involving a manifest, as in

```
MAX = 10;
func() {
    auto x[MAX*2], y[MAX + 7];
```

All **auto** statements should appear at the beginning of a function because they actually do a certain amount of work. An **auto** statement points the variables towards the stack storage allocated for them. Thus if you have a loop of the form

```
loop: auto a[10];
      ...
      a = "string";
      ...
      goto loop;
```

the word `a` is first set as a pointer to a vector on the stack, and later as a pointer to the string `"string"`. When the function loops back to `loop`, the **auto** statement points `a` back to the vector. Putting all your **auto** statements at the beginning of a function avoids problems like this where you may be surprised by a sudden re-assignment.

The initial contents of an **auto** vector or **auto** variable are always undefined when a function begins execution. Thus you must make sure that your functions always initialize their local variables explicitly.

5.4.3 Labels

Any unique identifier followed by a colon and preceding a statement is defined as a label, as in

```
again: ;
nxt: x = getchar();
```

A statement may be preceded by as many labels as you find necessary, as in

```
lab1: lab2: lab3: printf("hi there");
```

5.5 Transfer of Control

There are four kinds of statements which transfer control from one part of a program to another. The **goto** statement jumps to a specified label. The **return** statement exits from a function. The **next** and **break** statements simplify loop control.

5.5.1 The goto Statement

```
goto identifier ;
```

transfers control to the statement which has the label `identifier`. If `identifier` has not already appeared in the function, B assumes it is a label.

however, it is almost always a bad idea to transfer into a compound statement. Not only is the action confusing to understand when reading code, it can lead to unpleasant surprises.

Because B is a typeless language, the compiler has no way of knowing whether the identifier you supply in a **goto** statement will really turn out to be a valid label. It is valid, but probably erroneous, to say

```
extrn b;
 ...
goto b;
```

Never try to pass a label as an argument to a function and then use that label to transfer to another function. The program will end up in the destination function, but with the previous function's stack pointer. This is bound to result in disaster eventually.

5.5.2 The return Statement

```
return;

return ( expression ) ;
```

The **return** statement ends the execution of a function, returning to the function's caller. Upon return, all temporary storage in use by the particular invocation of the function is released for future use by other functions.

The first form of the **return** statement merely returns control to the calling function. The second form passes back a one word value.

The construct

```
return();
```

is considered a syntax. If you do not want to return a value, omit the parentheses entirely.

If there is no **return** statement at the end of a function definition, B implicitly inserts one. This means that control returns to the caller at the end of a function, whether or not there is an explicit **return** statement there.

Normally, a B program terminates immediately after executing the last statement of `main`. The library function `EXIT` can terminate execution of your program at some other point (see "expl b lib exit"). Several other run-time functions can also terminate your program, including `ERROR` and `.ABORT`.

5.5.3 The break Statement

```
break;
```

break drops out of the innermost enclosing **while**, **for**, **switch**, **repeat**, or **do-while** statement. The compiler generates a fatal error if a **break** statement is not inside one of these.

5.5.4 The next Statement

```
next;
```

the test which determines whether looping should continue. Inside a **repeat** statement, **next** transfers to the top of the repeat block. Note that **next** is only valid in a **switch** statement when the **switch** itself lies inside one of these looping statements.

5.6 The if Statement

```
if ( expression ) statement
```

If the result of the `expression` is non-zero, **if** executes the `statement`. Note that the `expression` must be enclosed in parentheses.

A more complicated form of the **if** statement is

```
if ( expression ) statement else statement
```

If the result of the `expression` is non-zero, the first statement is executed; otherwise, the second statement is executed.

If a nested **if** statement has fewer **else**s than **if**s, the compiler associates each **else** with the closest **if** at the same level of nesting. For example,

```
if ( ex1 ) if ( ex2 ) stmt1 else stmt2 else stmt3
```

resolves to

```
if ( ex1 ) {
    if ( ex2 ) stmt1
    else stmt2
}
else stmt3
```

Think of **if**s and **else**s being placed on a pushdown stack as they appear. In this way, an **else** is paired with the **if** immediately preceding it, and both are popped off the stack together at the end of the **else**. If a new **else** then occurs, it is pushed onto the stack and paired with the next **if** that is still on the stack.

Here are some examples of **if** statements:

```
if( a ) y = x;

if( a < 2 ) y = a; else y = 0;

if( a != b )
    z = g( y );
else {
    a += x;
    b -= y;
}
```

5.7 Iterative Statements

Iterative statements repeat zero or more other statements until something stops the looping.

5.7.1 The repeat Statement

repeat merely executes the given statement forever unless a **break** statement is encountered, or a **goto** passes control to a statement outside the loop. The `statement` in a **repeat** statement is almost invariably compound. **next** and **break** statements are valid inside a **repeat**.

5.7.2 The while Statement

```
while ( expression ) statement
```

If the expression is non-zero, the `statement` associated with the **while** is executed. After execution of the `statement`, the expression is re-evaluated. If the expression is again non-zero, the `statement` is executed again. In other words, while the result of the expression is non-zero, the `statement` is executed. When the result of the expression is zero, control passes to the next statement following the **while** statement.

If the given expression is initially zero, the statement is not executed.

break and **next** statements are valid in a **while** statement.

5.7.3 The do-while Statement

```
do statement while ( expression ) ;
```

The **do-while** provides a loop with a test at the bottom of the loop. It is equivalent to

```
repeat {
    statement
    if( !expression ) break;
}
```

Thus, if the given expression is zero, the statement is still executed once.

break and **next** statements are valid in a **do- while** statement.

5.7.4 The for Statement

```
for ( expr1; expr2; expr3 ) statement
```

The **for** statement may be used to set, test, and increment a variable in order to control a loop. The **for** statement is equivalent to

```
expr1;
while ( expr2 ) {
    statement
    expr3;
}
```

The first expression (generally the initialization of a controlling variable) is evaluated. Then, while the result of the second expression (usually a test) is non-zero, the statement is executed. Before returning to re-evaluate the second expression, the third expression, (often incrementing the controlling variable) is evaluated.

Both **break** and **next** are legal in a **for** statement. The effect of **next** is to pass control to the evaluation of the third expression.

same controlling variable, if a controlling variable is even involved. Note that the second expression is always treated as a logical expression. Here are some examples:

```
for( i = 0; i < 10; ++i )
    x[i] = j[i];

for( i = 10; i <= x; i += 2 )
    for( j = 1; j < y; ++j )
        g[i][j] = f( i + j );

for( ; i < n; ++i )
    y[i] = z[n - i];

NULL = 0;
NEXT = 1;
DATA = 0;
   ...
for( p = startlist; p != NULL; p = p[NEXT]; )
    if( p[DATA] >= x ) break;
```

5.8 The switch Statement

The **switch** statement provides a conditional branch depending on the one-word result of an expression. The **switch** has the following formal syntax:

```
switch ( expression ) statement
```

The statement is always compound, and hence can never be null. Special labels are allowed inside the statement to indicate where processing starts for a given case, as in

```
switch ( expression ) {
    case const-expr: statement
    case const-expr :: const-expr: statement
    break;
    case op const-expr: statement
    /* op. is one of <, <=, >=, >  */
    default : statement
}
```

The **switch** evaluates the expression and compares the result with the constant or constant bounds in each **case** label. It selects a case, if there is one that matches the calculated result, and begins executing the compound statement at the statement immediately following the appropriate **case** label. If the expression result fits no case, execution continues at the label **default** (if supplied) or at the next statement following the **switch**, if **default** is not supplied.

Once a case is selected, execution always falls through into the next case, unless the program finds a statement that alters the control flow. **break** is often used to jump out of a **switch** statement after executing the code for a particular case.

A statement may have more than one label or **case** label, just as a label or **case** label may be followed by more than one statement.

As shown above, a **case** may be satisfied by

- a single value;
- a range of values which includes its endpoints; or

Overlapping bounds draw a fatal diagnostic from the compiler.

As usual, we use `const-expr` to denote any legal combination of numeric or character constants, unary operators, binary operators and parentheses which can be evaluated at compile time as some constant value. String constants are not permitted in this context.

Attempts to **switch** with floating point values may have unusual results, since the generated code performs integer comparisons. There is no problem with exact floating point comparisons, but if a range of floating point numbers is specified, the odds are that things won't work the way you want.

The compiler is capable of generating code for a **switch** statement in several different forms. It chooses which form to use on the basis of efficiency considerations.

As an example of **switch**, here is a function which determines whether a character is valid in a B identifier. The function also converts uppercase letters to lowercase.

```
alphnum( c )
    switch( c ) {
        case 'A' :: 'Z' :
            /* converts upper case to lower */
            /* and falls through to return */
            c += 'a' - 'A';
        case 'a' :: 'z' :
        case '0' :: '9':
        case '.':
        case '_':
            return( c );
        /* would use break if return not used */
        default:
            return( 0 );
}
/* end of alphnum */
```

6. Expressions

Expressions in B are constructed according to rules that govern combinations of operators, identifiers, square brackets, and parentheses. B has a large set of operators, which are described in this section.

Because B is typeless, the compiler always assumes a given operation on a word is appropriate. Although this forces you to do more checking yourself, it also gives you the scope to do almost anything you want. This typeless characteristic often causes trouble for beginning users of B, because the compiler happily accepts questionable operations, such as adding one to a function name, or using a pointer as a function call. Such is the price of freedom.

The compiler takes no responsibility for the validity of expressions. There is no run-time monitoring of possible arithmetic overflows or faults. In the B run-time environment, overflow faults are inhibited (unlike the Pascal environment where they are not). A divide error (such as dividing by zero) leads to an error message and program termination.

Expressions are evaluated according to a precise *order of binding*. This includes both the hierarchy of evaluation (which determines the order in which different types of operators are evaluated), and *grouping* (which determines the order in which operators of the same type are evaluated). This chapter lists the

The hierarchy of evaluation and the grouping rules are summarized in Appendix A.

Note: The rules of binding and grouping do not completely dictate the order in which expressions are evaluated. In particular, arguments to functions may be evaluated in any order that the compiler chooses. This may make a difference, as in

```
func(i,i++)
```

Suppose `i` begins with the value `1`. If the compiler chooses to evaluate the first argument before the second argument, the call turns into

```
func(1,1)
```

However, if the compiler chooses to evaluate the second argument before the first, the call turns into

```
func(2,1)
```

because the value of `i` has already been incremented by the time the compiler evaluates the first argument. Since there is no way to guarantee the order in which the compiler evaluates function arguments, you must avoid function calls that have this kind of ambiguity.

In the same way, consider an expression like

```
func1(a) + func2(b)
```

The compiler may evaluate either operand first; it may call `func1` then `func2`, or vice versa. Therefore, you should avoid code where the order of operation makes a difference. This applies to all operators where the order of operand evaluation is not stated specifically. (Do not confuse order of operand evaluation with grouping. Grouping states that

```
A + B + C
```

is evaluated as

```
(A + B) + C
```

In other words, the left addition takes place before the right. However, there is no guarantee of the order in which individual operands are evaluated.)

6.1 Primary Expressions

The primary expression is the basic building block used in constructing more complex expressions. Primary expressions are defined recursively as follows:

`name`
: Any valid identifier is a primary expression.

`constant`
: Any valid constant is also a primary expression.

`primary[ expr ]`
: Any primary expression followed by an expression in square brackets is also a primary expression. (This is the standard subscripting operation.)

`primary( arglist )`

form of a function call operation. The opening parenthesis must be followed by a (possibly empty) list of arguments consisting of expressions separated by commas. The list of arguments must be followed by a closing parenthesis.

`( expr )`
: Any expression which is enclosed by parentheses but is not a function argument list is a primary expression. This lets you use parentheses to alter the order of binding.

Here are some examples of simple primary expressions:

```
x   getchar()   (a + b)   6   x[i]   6[x]
```

In cases where a primary expression is composed of other primary expressions, grouping occurs from left to right. Consider, for example,

```
x[i][j]        x[i]()
```

In the first case, `x` is treated as a pointer to a vector of vectors. In the second case, `x` is treated as a pointer to a vector of functions, and the expression calls one of those functions. In both cases, `x[i]` is evaluated first and placed in a temporary storage location; for the purpose of illustration, we will call this temporary storage location `y`. The two expressions above are evaluated as `y[j]` and `y()`, respectively.

6.1.1 Function Calls

The general form of a function call is:

```
primary( expr1, expr2, ..., exprn )
```

Most commonly, `primary` is just the name of the function you wish to call, but the generality of the construction also lets you use vectors or lists of function addresses.

B gives no error if a caller expects a function to return a value but no value is returned. In this case, the caller receives an undefined value, which of course can lead to errors. It is up to you to make sure that functions return values when the caller expects such values.

You must make sure that a function is called with as many arguments as it needs. It is possible to write a function that can cope with receiving too many or too few arguments, but if the function is not ready to handle variable numbers of arguments, specifying the wrong number of arguments will lead to errors.

The compiler recognizes function calls by the parentheses around the argument list; thus these parentheses must always be present. For example, suppose you have a function named `proc` which requires no arguments. To call it, you must write

```
proc()
```

If you wrote

```
proc
```

the compiler would not recognize this as a function call.

6.2 Rvalues and Lvalues

word and the address of a word.

The term *Rvalue* refers to the contents of a word or the value of an expression. The term comes from the fact that Rvalues frequently appear on the Right hand side of an assignment (though they can certainly appear in other instances as well). Any expression in B may be evaluated for an Rvalue. For example, the Rvalue of a subscripting operation is the contents of the word addressed by the sum of the vector pointer and the index.

Everywhere in this manual where we say "expression", we mean an expression whose result is some Rvalue.

The term *Lvalue* refers to the address of a word. The term comes from the fact that Lvalues frequently appear on the Left hand side of an assignment (though they can also appear in other places as well). Only a name, a subscripting operation, or a primary expression prefixed by the unary indirection operator `'*'` may be evaluated for an Lvalue. The Lvalue of a subscripting operation is the *address* calculated by adding the vector pointer and the index.

Context determines whether an expression is evaluated for its Rvalue or its Lvalue. For example, consider the assignment

```
a[3] = 2 + x
```

The expression on the right yields an Rvalue which is the sum of the contents of `x` and the constant `2`. `a[3]` adds the index `3` to the contents of the vector pointer `a` and yields an Lvalue which is the address of a memory location to store the Rvalue `2+x`.

The expressions

```
6 = x
(a + b) = x
```

are both invalid because the expressions on the left of the assignment operator do not yield a proper Lvalue. If they did yield Lvalues, you could change the value of a constant in the first case, and make a meaningless assignment in the second.

6.3 Unary Operators

A unary operator acts upon a unary expression to change it in some manner. A *unary expression* is either a primary expression or a primary expression already modified by one or more unary operators. In the definitions below, "Rvalue" or "Lvalue" must be a unary expression. Unary operators are applied from left to right. The unary indirection operator `'*'` operates on an Rvalue to produce an Lvalue; all other unary operators produce Rvalues.

The unary operators recognized by B are:

`#Rvalue`
: Assumes the value of the expression to be integer and converts it to single precision floating point.

`##Rvalue`
: Assumes the value of the expression to be single precision floating point and converts it to integer.

`~Rvalue`

`-Rvalue`
: Results in the arithmetic negation (two's complement) of the operand.

`#-Rvalue`
: Results in the floating point negation of the operand word. (Note that `#-` is not the same as `-#`.)

`!Rvalue`
: Logical not. The result is zero if the operand is non-zero; otherwise, the result is one.

`*Rvalue`
: The indirection operator. Takes the Rvalue and allows it to be used as an Lvalue (an address). This is the only case in which a unary operation returns an Lvalue. Because of this, any primary expression prefixed by a `'*'` may appear on the left hand side of an assignment.

```
*6 = x
```

stores the contents of `x` in memory location six.

```
y = *x
```

sets y to the contents of the word pointed to by `x`.

`&Lvalue`
: The address operator. Forces the program to generate the address of the expression, and then allows that address to be used as an Rvalue. For instance, `&x` is an Rvalue which contains the address of `x` in the lower 18 bits. `&6` is invalid, because `6` may not be considered as an Lvalue. `&*6` is the same thing as `6` itself.

`++Lvalue`
: Auto-increment. Adds `1` to the contents of the word referred to by the address Lvalue, and then allows the Rvalue of that word to be used. The auto increment/decrement operators all require an Lvalue as their operand. The Lvalue is required because of the implicit action of assignment, but the result is always an Rvalue.

`Lvalue++`
: Uses the Rvalue of the word which Lvalue points to, and afterwards adds `1` to that word. For example, suppose `x` has the value `1`. Then the value of `x++` is `1` (since `x` is incremented after obtaining the value of the expression) while the value of `++x` would be `2` (since `x` is incremented before obtaining the value of the expression). In both cases, `x` ends up with a value of `2` after the expression has been evaluated.

`--Lvalue`
: Subtracts `1` from the Rvalue of the word which Lvalue points to, and then allows that Rvalue to be used.

`Lvalue--`
: Uses the Rvalue of the word which Lvalue points to, and afterwards subtracts `1` from that word.

`@primary`
: The at-sign operator forces the use of GCOS8 hardware indirection. Its effect is to OR the indirect bit into the last generated instruction for an expression (Rvalue or Lvalue). The instruction affected is usually a load or store. It was most commonly used to access characters using tallies, by indirecting to a word with tally modification and the address of a tally word. Normally, you should not use it.

There is a fundamental relationship between the `'*'` operator and subscripting which should help you understand how addressing works in B. Both seek to generate an address using one or more Lvalues. In

added to the Rvalue of the Lvalue first. The following are exactly equivalent everywhere:

```
a[b]        *(a+b)        b[a]
```

For example, `a[0]`, `*a`, and `0[a]` are completely equivalent in B. To be able to write or understand B programs, it is vital that you understand the validity of this relationship.

Here are some examples involving unary expressions:

`++i`
: adds one to the value of `i`. Frequently used shorthand for

```
i = i + 1
```

`a[b][c]`
: forms an address by adding together `a` and `b`, getting the word that the result points at, and then adding `c` to the contents. This expression is equivalent to

```
*(*(a + b) + c)
```

`&a[i]`
: forms the address of the word `a[i]` by adding the values of `a` and `i`. Thus, this expression is almost equivalent to

```
a + i
```

The equivalence can break down if one of `a`, `i`, or `a+i` is too big to be represented in 18 bits. Since addresses in B are only 18 bits long, treating longer numbers as addresses may lead to unexpected consequences. When one of `a`, `i` or the sum exceeds 18 bits, the value in the upper 18 bits is undefined.

`y = *(&x)`
: A verbose way of saying `y=x`. `&x` yields the address of `x`, and `*(&x)` refers to the contents of that address. Note that `&(*x)` is *not* the same thing as `*(&x)`.

`x = *p++`
: The word which to which pointer `p` points is copied into `x`, then `p` is incremented by `1` to point at the next word. In other words, `p` is used, then incremented.

`x = ++*p;`
: The word that `p` points at is incremented by `1` and then copied into `x`.

`*++p = x`
: The contents of `x` are copied into the word that `p` points to, but only after `p` has been incremented to point to the next word. In other words, `p` is incremented, then used.

`*6 = 2`
: places the value `2` in memory location six. `*6` is the same as `0[6]` or `6[0]`. This kind of construct is often used to access locations in the slave program prefix (memory locations 0 to 0144); for example, this is an easy way for a B program to address its fault vectors.

`x[a[b] + 1]`
: puts the contents of `a[b]` into a temporary storage location and adds `1` to the temporary to get the subscript. The contents of `x` and the subscript are added together, yielding the address of the element of `x` to be used.

6.4 Binary Operators

All other operators are *binary* operators: they require both a left and a right operand. Each operand must be an Rvalued expression. The result of any binary operation is also an Rvalue.

With one exception, the order in which the two operands are evaluated is arbitrary, so the evaluation of one side should not depend on a side effect of the evaluation of the other side (a function call, for instance). Logical operators are the only exception. Their operands are always evaluated strictly from left to right.

The code that B generates for floating point operations is correct, but not especially efficient, since the compiler generates a separate load and store each time a floating point operand is used. This means that floating point capability is available if you need it, but for intensive use of floating point, you should probably call a Fortran or Pascal routine to do the job.

The binary operators that follow are listed in the order in which they are evaluated: the operators described first are evaluated first.

6.4.1 Shift Operators

`expr << expr`
: Takes the left operand as a one-word bit pattern to be left shifted logically. The right operand supplies the number of bits to shift. If negative, or greater than 127, the result of this shift operation is undefined.

`expr >> expr`
: Logical right shift according to the same rules. No arithmetic right shift is defined in the language, but you may use the library function `ARS`.

Shift operators group from left to right.

6.4.2 Bitwise Operators

The chart summarizes the results of bitwise operations. These operators are described in more detail below. The table shows the effect of each operation on one bit.

Operands		Results		
A	B	A&B (and)	A|B (or)	A^B (exor)
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

The three bitwise operators group from left to right.

`expr & expr`
: Bitwise AND. The `&` operator takes the bitwise "and" of its two 36 bit operands. If bit `i` of both operands is one, then bit `i` of the result is one. Otherwise, bit `i` of the result is zero.

`expr ^ expr`
: Bitwise exclusive OR. If bit `i` is on in one, but not in the other, then bit `i` of the result is on (one). Otherwise, the bit is off (zero).

`expr | expr`

also. Otherwise, the bit is off.

6.4.3 Multiplicative Operators

These operators perform multiplication and division.

`expr * expr`
: Integer multiplication.

`expr / expr`
: Integer division of the first integer operand by the second. This results in a divide check abort if the right operand is zero. The result is zero if the left operand is less than the right. The result is truncated towards zero if the right operand does not divide evenly into the left. The result is positive if the operands are both positive or both negative; otherwise, it is negative. Thus `-(a/b)`, `(-a)/b`, and `a/-b` are all equivalent.

`expr % expr`
: Integer remainder from the integer division of the first operand by the second; in other words, the result is the first operand modulo the second. If the remainder of the integer division is non-zero, it has the same sign as the left operand.

`expr #* expr`
: Single precision floating point multiply. All floating point operators assume floating point operands.

`expr #/ expr`
: Single precision floating point divide.

Multiplicative operators group from left to right.

6.4.4 Additive Operators

These provide integer or floating point addition and subtraction.

`expr + expr`
: Integer add.

`expr - expr`
: Integer subtract.

`expr #+ expr`
: Single precision floating point add.

`expr #- expr`
: Single precision floating point subtract.

Additive operators group left to right.

6.4.5 Relational Operators

`expr == expr`
: Equal.

`expr != expr`
: Not equal.

`expr < expr`

`expr <= expr`
: Less than or equal.

`expr > expr`
: Greater than.

`expr >= expr`
: Greater than or equal.

The result is `1` if the given relation between two integer operands is true; otherwise, the result is zero.

The following operators perform the same function for floating point operands:

```
#==     #!=     #<     #<=     #>     #>=
```

6.4.6 Logical And

`expr && expr`
: The result is an integer `1` if the result of both expressions is non-zero; otherwise, the result is zero.

The left hand operand is always evaluated first. If its result is zero, the result of the expression is zero and the right hand operand is not evaluated.

6.4.7 Logical Or

`expr || expr`
: The result is an integer `1` if the result of either expression or both is non- zero; otherwise, the result is zero.

The left hand operand is always evaluated first. If the result is non-zero, the result of the total expression is non-zero and the right hand operand is not evaluated.

6.5 Query Operator

```
expr1 ? expr2 : expr3
```

The first operand is evaluated. If the result is non-zero, the second operand is evaluated and returned, while the third expression is ignored. If the result of the first operand is zero, the third operand is evaluated and returned, while the second is ignored. In both cases, the value that is returned is an Rvalue.

This is analagous to

if(expr1) expr2; else expr3 but has the advantage that it may be used in an expression. For example, a function to calculate the maximum of two numbers might be coded as

```
max( a, b ) return( a > b ? a : b );
```

Grouping is right to left, so that

```
a ? b : c ? d : e
```

is equivalent to

6.6 Assignment Operators

`Lvalue = expr`

Simple assignment. Takes the one word result of the evaluation of `expr` and stores it in the word addressed by the Lvalue.

`Lvalue` *op*`= expr`

Compound assignment. This is equivalent to the assignment

```
Lvalue = Rvalue op ( expr )
```

where *op* can be any one of

```
*  /  %  +  -  <<  >>  &  ^  |
```

and Rvalue refers to the contents of the word addressed by Lvalue. Note that neither floating point nor relational operators may be used in compound assignments.

As an example of compound assignment,

```
x *= a + b;
```

is the same as

```
x = x*(a + b);
```

In all cases, the right hand operand is evaluated first, even though the operator in the assignment may have higher binding strength than an operator in the expression.

Assignments group right to left.

```
x = y = 0;
```

is taken as

```
x = (y = 0);
```

This points out that the value of an assignment expression is the value that is being assigned.

Remember that assignment is an operation, not a statement, and so is valid almost anywhere, including conditional expressions, such as

```
if( (x = y[i++]) == z )...
```

Parentheses are used in this case to alter the order of precedence. These are required here because the assignment operators have the lowest evaluation precedence, which means they are evaluated last. If the parentheses were removed, `x` would be assigned either zero or one, depending on the outcome of the comparison

```
y[i++] == z
```

7. The B Library

These simplify your programming problems and also supply a reasonable interface with the GCOS/TSS environment.

Every B library function available for public use has an expl file under "expl b lib". There is also an index of all documented routines in "expl b lib index".

This chapter only discusses the routines you need to get started using B. Furthermore, we do not discuss all the features of the functions mentioned here. The "explain" files are more up to date and detailed than this manual can be.

The B library functions were written to run quickly in as small a space as possible. For this reason, most library functions do no error checking. It is up to the programmer to make sure that library functions are passed valid arguments. If invalid arguments are passed to a library function, the function could very well give meaningless results or cause your program to abort. In particular note that a large number of library routines may "lockup fault" if called with the wrong number of arguments.

For historical reasons, the library is known as the *B library*, but in fact, the library supports several languages. Many library routines are sensitive to the format of pointer arguments and must have the upper 18 bits zero in order for the routine to function correctly. If the upper 18 bits are non-zero, the function takes the lower 18 bits as a machine pointer and the upper 18 bits as the actual word address. In addition, non-zero values in the upper 18 bits of a pointer may cause the function to behave in slightly different ways. In particular, you must remember to mask off the upper 18 bits when dealing with values passed in the `argv` vector to the routine `MAIN`.

Some functions may return a value. This chapter indicates the nature of return values using sample assignments. Also, some functions are called with a variable number of arguments. If you want to use an optional argument, you must usually specify all the optional arguments which precede it in the function's argument list. Throughout this chapter, optional arguments are shown in square brackets. For example, if a function is shown as

```
return = func( arg1 [, arg2, arg3] );
```

and you want to specify a value for `arg3`, then you must also specify a value for `arg2`.

Sometimes, the first argument may be optional. This often occurs in I/O functions where the first argument is a *unit*. In this case, the function examines its first argument to see if it is a valid `unit` number; if the number is not valid, the function adjusts argument references accordingly.

7.1 Useful I/O Functions

Chapter 8 describes I/O in detail. However, since all I/O in B is performed through library functions, we will describe the most frequently used I/O functions in this chapter.

Chapter 8 also provides a detailed discussion of read and write units. For the time being, all you need to know is that `unit` is a way to specify the physical device used in a read or write operation.

7.1.1 Stream I/O Functions

```
char_got = getchar()
```

file, GETCHAR returns an '*e' character (which means that char_got is set to zero). Most of the input routines described below behave as if they make repetitive calls to GETCHAR.

`char = getc( unit );`
: is the same as GETCHAR, except that it reads from the specified unit instead of the current read unit.

`char = ungetc( char );`
: sends a character back to the current read unit, so that the next call to GETCHAR returns the last character put back.

`char_put = putchar( char );`
: outputs the specified character to the current write unit. PUTCHAR also returns its argument word as its value. Thus in the statement above, char_put is assigned the value of char.

`char = putc( unit, char );`
: is the same as PUTCHAR, except that PUTC outputs the character to the specified unit instead of the current write unit.

`str = getstring( [unit,] string [,maxl] );`
: gets the next line of input, or the remainder of the current line of input if GETCHAR has already read some of the line. The new-line at the end of the line is not returned; instead it is replaced by '*e' to mark the end of the string. string is taken as a pointer to a vector long enough to hold the string (it is your responsibility to make sure the vector *is* long enough). unit is used if supplied; otherwise GETSTRING reads from the current read unit. If maxl is given, only the first maxl characters are returned, with '*e' added to the end. If the unit is closed or at end-of-file, GETSTRING returns zero; otherwise it returns string. The string is the null string "" if the input line only contains a new-line. GETSTRING normally returns the next line of input from the terminal, or the next logical record from a file.

`string = getline( [unit,] string [, maxlen] );`
: is the same as GETSTRING, except that GETSTRING includes the '*n' terminating the line just before the '*e' that ends the string.

`printf( [unit,] format-string, arg1, arg2, ... );`
: is the most frequently used output function. If unit is not supplied, PRINTF writes to the default write unit. If unit is given, PRINTF writes to the specified unit instead. format-string is a string describing how the arguments are to be output. It may contain any combination of literal characters and format placeholders. A placeholder has the form "%nnx", where nn is an optional count, and x is one of the following characters:

  `b`
  : The corresponding argument is taken as a *pointer* to a string of BCD characters, which is to be translated to ASCII and printed. Since BCD strings do not have a string terminator, the count is assumed to be 6 if it is not specified explicitly. Trailing blanks in the BCD string are stripped.

  `c`
  : The corresponding argument is printed as an ASCII character. The count option is not applicable.

  `d`
  : The corresponding argument is taken as an integer which is converted to a string of decimal characters and output.

  `f`
  : The argument is taken as a floating point number to be converted to a string, then written out. If your program does not use at least one floating point operator, your program must include the statement

```
extrn .float;
```

to force the loading of the floating point output routines.

`o`
: The contents of the argument word are output in octal format.

`s`
: The corresponding argument is taken as a pointer to an ASCII string, which is stripped of its trailing `'*e'` and transmitted to the output unit.

Note that PRINTF strips off all null (`'*e'`) characters wherever they appear and does not print them. This is a common feature of most B output functions.

Below we give some examples of uses of the PRINTF routine.

```
printf( "%c", 'a');
```

prints the character constant 'a'.

```
printf( "The number is %d.", num );
```

prints the contents of the variable `num` as a decimal integer. If `num` contains the integer five, the print line has the form `"The number is 5."`

```
printf("Floating %f*nis octal %o.*n", x, x);
```

prints two lines because of the `'*n'` new-line escape in the format string. The first line contains the floating point value of `x` and the second line contains the octal equivalent of this value. Note that the second line ends in another new-line character.

If a character in the format string is not part of a recognized format, it is printed as it appears. If a format does not have a corresponding argument, it is printed as a literal string. To print out `'%'`, you must use `'%%'`.

There is more to `PRINTF` than is described here; for full details, see the explain file "expl b lib printf".

`number = getnum();`
: reads the next (possibly signed) integer number from the current read unit. `GETNUM` skips along the current input line until it finds a character which is not a blank, tab, or new-line. If the character found is not a digit or a sign, `GETNUM` returns a zero to indicate that a number couldn't be found. `GETNUM` also returns zero if it finds a sign but the next character is not a digit. Otherwise, `GETNUM` collects numeric characters until it finds a non-digit. It then converts the collected numeric string into a decimal integer and returns that number as its value. The external variable `GETN.A` has the value `1` if a valid number was found. The external variable `GETN.L` contains the last character read.

`status = eof( [unit] );`
: returns a non-zero value if `unit` is at end-of-file. If `unit` is not given, the current read unit is used. Once a unit is open, end-of-file is set only after an attempted read detects a true end-of-file condition. If `unit` is given, and the unit is an output disk file, `EOF` writes a logical end of file and begins a new block.

There are a few other routines which should be mentioned, but which will not be described in detail here:

- GETREC and PUTREC let you read/write a logical record (including a record control word) without any intervening processing by the I/O package. You must understand the standard system format before you attempt to use GETREC or PUTREC.

7.2 String Operations

The compiler itself does not provide any string-manipulation abilities; all operations on strings are handled by function calls. Recall that a string is a sequence of characters packed four to a word in a vector and terminated by the ASCII null `'*e'`.

7.2.1 Random String Processing

`ch = char( string, i );`
returns the i*th* character in a string pointed to by `string`. The count always starts at zero, so that

```
char("the string",2)
```

is `'e'`.

`ch = lchar( string, i, char );`
replaces the i*th* character in the string pointed to by `string` with the character `char`. The value LCHAR returns is the character `char` that was placed in the string.

For BCD strings, you should use the function CHARB instead of CHAR and LCHARB instead of LCHAR. The calling sequences are the same, but they take and return BCD characters, not ASCII.

7.2.2 String Utilities

`str1 = print( str2, format, arg1, arg2, ...);`
is like PRINTF except that the prepared print string is put into `str2` instead of being output. Thus,

```
print(x,"The answer is %d.",2)
```

assigns the string `"The answer is 2."` to `x`. The print string created by PRINT is always a string of ASCII characters. Because of this, PRINT can be used to convert from BCD to ASCII; for example,

```
print(x,"%b",y);
```

takes `y` as a pointer to a string of BCD characters, converts this string into ASCII characters, and places the ASCII string in the memory location indicated by `x`. PRINT returns a pointer to the created print string as its value.

`string = concat(string, s1, s2, ... );`
concatenates the strings `s1` through `sn` and places them in string `string`. The output string may be identical to the first input string `s1`. When called with only two arguments, as in

```
concat( xstr, ystr )
```

CONCAT copies one string into another. CONCAT returns its first argument as its value (i.e. the output string).

returns a non-zero value if the string contains only the end-of-string character `'*e'` and zero otherwise.

`val = equal( string1, string2 );`
: returns a non-zero value if the two strings supplied are identical, and zero otherwise.

`count = length( string );`
: returns the number of characters in the string pointed to by `string`, not including the terminating `'*e'`.

`result = compare( string1, string2 )`
: compares two strings. If they are equal, COMPARE returns zero. Otherwise, it returns a value one greater than the position in `string1` where the two strings differed. The returned value is positive if `string1` is greater than `string2`, or negative if `string1` is less than `string2`. Thus the value of

```
compare( "string1", "string2" )
```

is `-7`.

`pos = any( c, string [,i] );`
: checks whether the character `c` appears anywhere in the string `string`. If `i` is specified, the scan for `c` starts at character position `i` in `string`. Otherwise, the scan begins in position zero. If the desired character is found, ANY returns the position of the character in the string. If it is not found, ANY returns `-1`.

`newpos = scan( arg, string, pos [,skipstr, stopstr] );`
: obtains the next group of characters ending with a given delimiter, then places those characters in the string pointed to by `arg`. SCAN begins scanning at position `pos` in `string`.

`stopstr` is a string which contains a list of acceptable delimiters to end the string being collected, while `skipstr` is a string containing a list of characters to be skipped before beginning to collect characters for the string. For this reason,

```
scan(x,y,0," *n"," ;.")
```

begins at position zero in string `y`, skips over any spaces and new-line characters until it finds a different character, thens collect characters in `x` until SCAN finds a space, a semi-colon, or a period. SCAN implicitly assumes that `stopstr` contains the end-of-string character `'*e'`. If the `stopstr` argument is omitted, the default is the same string of characters as in `skipstr` (plus `'*e'`); if `skipstr` is omitted, the default is `" *t"` (spaces and tabs). The value `newpos` which SCAN returns is the position of the character which ended the scanning. In this way, SCAN can be called repeatedly to obtain arguments from a command line.

There are a number of other functions which perform string operations, including:

- NUMARG, which scans off numbers instead of character strings;
- ADDCHA, which appends a character to the end of a string; and
- READF, which can do formatted input from a string.

All these functions have explain files.

7.3 Dynamic Memory Allocation

memory is located in a free storage pool called the *heap*. Dynamic allocation from the heap is useful when you need storage for a vector, but won't be able to decide how much space you need until run-time. (If you did know the amount of space, you could just declare the correct length at compilation time.)

`addr = getvec( n );`
: obtains a vector of length `n+1` words from the heap and returns a pointer to this vector. Once a block of memory has been acquired in this manner, it may be referenced using subscripts, as in

```
x = getvec( 63 );
x[1] = 3;
```

`rlsevec( addr, n );`
: undoes a GETVEC by taking the block indicated by `addr` and marking it as free memory.

All memory allocation is done by manipulating a linked list of free memory called the *free list*. The free list initially includes the so-called "core hole". You can use RLSEVEC to return any area of memory which is not on the free list, as long as the address of the memory is greater than the load address of RLSEVEC. If you attempt to release memory which is already on the free list, in whole or in part, RLSEVEC immediately aborts your program.

GETVEC obtains more storage from the operating system if it is required. In TSS, a subsystem is aborted with the message "not enough core to run job" if a request for memory cannot be satisfied. In batch, GETVEC aborts with a "0K" abort code if the system denies a request for memory. Whether or not your program aborts, indiscriminate use of GETVEC without corresponding RLSEVECs can dramatically build up "garbage" storage and increase the size of your program unnecessarily.

Finally, here are three useful routines, each described fully by an explain file, which call GETVEC and RLSEVEC.

- GETMATRIX constructs a multi-dimensional matrix and returns a pointer to the object.
- ALLOCATE obtains a dynamic block of memory which automatically disappears when a function returns.
- RELMEM releases any free memory back to the operating system, in order to reduce program size.

8. Introduction to Input/Output

The largest class of functions in the B library are those concerned with input and output. Sequential input routines read any sequential file in standard system format, including media 0, 2 or 3 BCD, media 5, 6 or 7 ASCII and media 1 compressed source decks (comdks); in the process, the input routines convert the input to ASCII if necessary. Output is ASCII (media 6) unless specified otherwise.

The I/O package creates output files if necessary and "grows" them as required up to their maximum size or to the limit of the file space quota for a userid.

8.1 Units

A B program may have several files open for reading or writing at the same time. Each file is associated with a number called a `unit`, and every I/O function refers implicitly or explicitly to the I/O unit number.

`-5`
: is an input unit that is always associated with the terminal in TSS or filecode `I*` in batch. You can use unit `-5` to read from the terminal or filecode `I*`, even if the normal input has been redirected to come from a file. Normally, you should use unit `0` to read from the standard input instead of `-5`.

`-4`
: is an output unit whose destination is always the terminal in TSS or filecode `P*` in batch. It is most often used to avoid possible redirection of I/O by forcing error messages to appear on a terminal or hard copy device. Normally you should use unit `1` for output instead of `-4`.

`-3`
: is used for console input in batch only.

`-2`
: is used for console output in batch only.

`-1`
: In TSS, all output directed to this unit behaves as if it were typed at command level. In batch, output to `-1` goes to the execution report.

Unit `0` is initialized as the *standard input* unit. In TSS, this is the terminal but input may come from a file if you redirect the standard input on the command line that invokes the program. In batch, reading from unit `0` uses filecode `I*` if the filecode is defined and if input has not been redirected. If `I*` is not present and there is no input redirection, unit `0` is placed in the end-of-file condition.

Similarly, unit `1` is initialized as the *standard output* unit. In TSS, the default standard output is the terminal, but this may be redirected on the command line that invokes the program. In batch, output to unit `1` goes to the filecode `P*`, which is associated with the printer unless redirected.

Units `2` through `49` may be assigned by or to you, using various library functions. These units are usually associated with permanent or temporary disk files. It is permissible to open units `0` or `1`, but the usual practice is to leave them alone, so they may be redirected.

8.2 Unit Opening

Before any I/O may be performed on a unit, it must be initialized by a call to `OPEN`. `OPEN` has the form

```
ret = open( [unit,] filename, action );
```

where:

`unit`
: is the number of the unit you want to open. Normally, you omit the `unit` argument; this gives `OPEN` the freedom to choose any free unit. If you do specify a unit, and that unit is already open, `OPEN` saves the state of the open unit on a stack, then reassigns the unit number for the specified use. When you close this unit, the previous unit is restored from the stack and I/O may continue on that unit as if there had been no interruption.

`filename`
: is a pointer to a string containing the usual catalog/file string (for example, `"fbaggins/s/test.b"`). This string may also specify an altname in quotes. The string may not specify permissions.

`action`

the unit.

`ret`
: is the value returned by `OPEN`. If `ret` is not negative, the open succeeded and `ret` contains the number of the unit just opened. If there was an error of some kind, `ret` is given a negative value and no unit is opened. You only get this negative return value if you specify special access actions for `OPEN` in the event of an error. Usually if `OPEN` encounters an accessing error, it simply aborts your program.

Only a few of the various situations accepted by `OPEN` are dealt with in this section. For a full description of `OPEN`, see "expl b lib open".

8.2.1 Access Actions

Most of the time, the only thing you need to specify with `OPEN` is how you want to access the unit. Here are the alternatives:

`r`
: (read) means you want to read from the unit.

`w`
: (write) means you want to write to the unit.

`a`
: (append) means you want to write on the unit, but by appending to what is already there. If the unit is a file that doesn't contain anything, or if appending is not appropriate with this particular output device, the result is the same as if you had used the `'w'` action.

For ordinary sequential file processing, you usually only have to specify one of the above actions.

8.2.2 Mode Actions

`OPEN` offers three ways to specify the mode of the unit being opened in the action string. They are listed below.

`b`
: `"b"` stands for *block I/O*. When `"b"` is specified, the unit being opened must be a random access file.

`s`
: is used for so-called *string I/O*. The `filename` argument is a pointer to the start of a block in memory. The stream I/O functions place characters into this block of memory, rather then transmitting them to a file.

`i`
: instructs the function to open the file according to its mode (random or sequential). The program must then use the `FILDES` function to determine whether the file is random or sequential, then make whatever I/O calls seem appropriate.

If the `action` does not contain any of these modes, `OPEN` assumes that the I/O unit is a sequential file.

8.2.3 Error Actions

reasonably understandable error message.

The OPEN function lets you specify options which let you handle file opening errors, file I/O errors, or both. These options are in the form of characters which may appear in the `action` string:

`e`
: Arranges things so that a negative status is returned on an I/O error.

`f`
: Sets things up so that a negative status is returned on an OPEN or file access error, rather than aborting your program.

`m`
: Normally, when an error status is returned, OPEN does not print an error message. Including `"m"` in the `action` string tells OPEN to display an error message before returning an error status to the caller. `"m"` has no effect if you are not using `"e"` or `"f"`.

As an example, suppose you write

```
open( "/myfile", "rfm" );
```

and `/myfile` cannot be opened with read permission. OPEN prints an error message, then returns a negative value to indicate that the open operation failed.

For most errors, OPEN returns the negative of the file system error status. For example, OPEN returns `-5` for "permissions denied".

8.2.4 Default Units

At any given time, there is a default read unit and a default write unit, used by functions like GETCHAR and PUTCHAR. When you use OPEN to open a unit for reading, the unit which is opened is usually made the new default input unit. Similarly, using OPEN to open a unit for writing makes that unit the new default output unit. If you wish to prevent a unit from becoming the new default input/output unit, you can specify a `"u"` in the options for OPEN. Thus

```
open( "/myfile", "wu" );
```

opens `/myfile` for writing, but does not make it the new default output unit.

8.2.5 File Access Conventions

In TSS, OPEN follows a number of file access/create conventions.

1. Search rule:

A *quick access* name is one which contains no slashes or dollar signs and does not have an altname. It must also be less than nine characters long; if not, it is considered to be in error. If you are opening a file and the name is a quick access name (for example, `b.out`), the file accessor first searches the AFT for a file of that name. If such a file is not found, the file accessor searches for a file of that name in the current catalog. If the specified file name is not of the quick access form, it is assumed to be the name of a permanent file.

If the search fails and the file is being opened for write or append, OPEN tries to create the file. If the file name is of the quick access form, OPEN tries to create a temporary file; otherwise, it tries to create a permanent file.

3. AFT rule:

If the file was already in the AFT when accessed or if the file is a temporary file created during the access, the file is left in the AFT when the unit is closed. If the file was not in the AFT initially and the file is permanent, it is removed from the AFT when the unit is closed. You may override these defaults by including appropriate characters in the `action` string. The character `'t'` (for transient) forces a file to be removed from the AFT when it is closed, whether or not it was in the AFT to begin with; and the character `'k'` (for keep) keeps the file in the AFT even if it would normally be removed.

8.3 Unit Closing

When you are through with a unit, you should close it explicitly by calling the CLOSE function. This has the form:

```
close( unit );
```

For sequential stream output units, CLOSE flushes the output buffer if necessary, and writes an end-of-file mark if appropriate. When CLOSE closes a unit associated with a disk file, the file is deaccessed if necessary. CLOSE frees any memory used to store information about the given I/O unit. If there was an old I/O stream associated with the unit number (before the just-closed stream was opened), the old stream is restored and I/O may proceed on that unit as if there had been no interruption. Otherwise, the unit number is free for further allocation.

When your `main` function terminates, or when you call EXIT directly, all open units are closed automatically. If you press Break during the execution of a program, EXIT is called unless your program has established its own break handling procedure.

Even though B does this clean-up closing automatically, it is a good idea to close units once you have finished using them. B uses about 500 words of memory for each open unit. Closing an unneeded unit therefore releases quite a bit of memory which your program can use for other purposes.

If a library routine tries to read data from a unit that is not open, the routine returns a value which indicates nothing happened. Similarly, output to a unit which is not open simply vanishes.

8.4 Unit Switching

Initially, the default read unit is the standard input unit (unit `0`) and the default write unit is the standard output unit (unit `1`). Successful calls to OPEN change the default read or write unit (unless you specify `"u"` as part of the `action` string).

You can change the default read unit with the `.READ` function, as in

```
old.unit = .read( [new.unit] );
```

returned.

```
old.unit = .write( [new.unit] );
```

works the same way as .READ, except that it applies to the default write unit.

8.5 .BSET - Redirection of I/O

Before starting your main program, the run-time initialization routine which prepares your program for execution calls a function named .BSET which "predigests" the command line for the user program and also sets up any "redirection of I/O" requested on the command line.

In batch, .BSET looks for the command line on filecode CZ. In your JCL, you might have something like

```
$      data    cz
command arg1 arg2 ...
```

.BSET breaks the command line up into "arguments". An argument is either a string of non-blank characters, a quoted string, or a redirect request.

A redirect request has three forms:

`<filename`
: opens the file for reading on unit 0. Input for unit 0 comes from this file, rather than the terminal.

`>filename`
: opens the file for writing. Output to unit 1 goes to this file, rather than the terminal. The output overwrites any current contents of the file.

`>>filename`
: is like `>filename`, except that if the file already exists, output is appended to the end of the current contents of the file.

A quoted string on the command line is delimited by either single or double quotes. To get a quote inside a quoted string, either use the quote which is not the delimiter or else put in two of the delimiter characters, as in

```
"that's the way to do it"
'that''s the way to do it'
```

.BSET collects arguments which are not redirect requests and builds a vector of pointers to those strings. MAIN is then invoked with

```
main( argc, argv );
```

where argc is the number of arguments collected (excluding redirection arguments) and argv is a pointer to the vector of strings. argv[argc] always contains the constant -1.

In addition, .BSET builds an external vector called .ARGTYPE. Each word of .ARGTYPE contains a character giving some indication of the type of argument in the corresponding ARGV string:

```
Type                    Character
-------------------------------
string                      ' '
string in single quotes     '*''
```

```
-option                  '-'
+option                  '+'
possibly signed number   '0'
string with = in it      '='
```

For example, consider the command line

```
go -r /myfile >b.out "a string"
```

MAIN starts up with ARGC set to 4, since >b.out is not included. All writes on unit 1 go to the file b.out, which is created if necessary. The ARGV and .ARGTYPE vectors are set up as follows:

```
argv[0] = "go"          .argtyp[0] = ' '
argv[1] = "-r"          .argtyp[1] = '-'
argv[2] = "/myfile"     .argtyp[2] = ' '
argv[3] = "a string"    .argtyp[3] = '"'
argv[4] = -1
```

and the contents of the remaining elements of the ARGV vector are undefined.

If you do not want to use .BSET, simply supply your own function definition of .bset(); to replace the library version.

Normally, however, you will want .BSET, because it simplifies the task of handling command lines. In fact, there are even more powerful facilities built into .BSET for scanning command lines with arguments of specified types. As well, you may call .BSET to scan an arbitrary string. For full details, see the explain file "expl b lib .bset".

8.6 Sequential Stream I/O

This section describes routines that handle I/O on terminals or standard system format sequential files.

The I/O package reads files of almost any media and arranges things so that the calling program sees only a stream of ASCII characters. For instance, any BCD printer slews and strange escapes in Media 3 files are correctly detected and converted on input, as are ASCII slews in Media 7 print image files. Compressed source decks are handled correctly, but the way B handles object decks is probably not very useful. Media 0 is always taken as variable-length BCD.

On output, the I/O package writes Media 6 ASCII unless special action is taken as described in the explain file for OPEN. When writing to SYSOUT in batch, output is Media 3 (BCD printer format).

You have the option of opening a unit to read, to write, or to append as follows:

`unit = open( filename, "r" );`
: opens a unit for reading, requesting read/concurrent permission.

`unit = open( filename, "w" );`
: opens a unit for writing, requesting write/concurrent permission.

`unit = open( filename, "a" );`
: opens a unit for writing so that output data written is appended to the end of the file. If the file is null to start with, OPEN treats the situation just like a regular open for writing.

8.6.1 Terminal vs. File

'*n', '*r', or '*f'. Of these, '*n' is never present on an input file; instead, the '*n' is automatically supplied by the I/O package to indicate the end of a record. On input from a terminal, you may separate logical records (lines) by using either the "return" or "line-feed" key.

End-of-file on a file is signalled by the presence of a special record at the end of the file. End-of-file on a terminal is signalled by a line whose first and only character is an ASCII file separator character (FS, octal 034), the same as used by TSS GFRC. On most ASCII terminals, you can transmit an FS character by typing the CNTL and '\' (backslash) keys simultaneously, followed by a carriage return. On single-case terminals, you can get a backslash by typing an 'L' and the shift key; thus an FS character is CNTL and shift-'L', followed by a carriage return.

Sequential file output is written in GFRC standard system format with blocks of 320 words. An end of line ('*n') character written to a sequential disk file signals the end of a record but the character itself is not placed in the record. All other record terminators do get written out. A new logical record is started after any record terminator is received. If more than 1272 characters are written in a logical record, the I/O package generates partitioned records to permit the logical record to span more than one physical block. A 320-word buffer is written out only when it is necessary to start a new buffer or when you call CLOSE.

The library functions buffer terminal output. However, if your program issues a read from a terminal, the I/O package arranges that all output sent appears on the terminal before it is "unlocked" for input.

8.7 Random File I/O

When using random-access files, your program is responsible for all input or output done on the file. The basic unit of I/O is the sector of 64 words. You may read or write any number of sectors with one function call.

To open a random file for reading, use the call

```
unit = open( filename, "rb" );
```

The rules are the same as for the regular OPEN call, except that the 'b' in the action string indicates a random file. 'b' stands for *block*.

To open a file for writing or reading-and-writing, use

```
unit = open( filename, "wb" );
```

To read on such a unit, use

```
status = read(unit, buffer, sector, nwds);
```

where buffer is a pointer to a vector where input data should be stored, sector indicates the sector of the file where the read should start, and nwds indicates the number of words to be read. The first sector number in the file is zero. If the status returned is non-negative, it is a count of the number of words transmitted; otherwise, it is the negative of the major error status from the I/O operation.

To write on such a unit, use

```
status = write(unit, buffer, sector, nwds);
```

multiple of 64 words. If the number of words you transmit is not a multiple of 64, the unused fraction is filled with zeroes on writing.

8.8 String I/O

By using one of the following calls, you can "open" a string in a way that lets you use standard sequential I/O functions on the string. Reading/writing characters in the string with this method is faster than using CHAR/LCHAR.

`unit = open( string, "rs" [,pos] );`
: opens a string so that calls to GETCHAR read characters from the string (`'s'` stands for *string*). GETSTRING reads all characters up to but not including the next `'*n'` or else up to the terminating `'*e'`. When the string is exhausted, the unit is in EOF status. If you want to start getting characters at some point other than the first character position, use the optional starting character position `pos`. Any library function which obtains characters from an I/O unit will also work if the unit is a string.

`unit = open( string, "ws" [,pos] );`
: opens a string so that calls to PUTCHAR write characters in the string. PRINTF, PUTSTRING, PUTNUM and other functions also write characters into the string.

`unit = open( string, "as" );`
: locates the terminating `'*e'` of `string` and sets things up so you start writing into the string at that point. It is up to you to make sure that the vector indicated by `string` is large enough to hold whatever data your program puts into it.

When you use CLOSE to close an output unit associated with a string, CLOSE writes a terminating `'*e'` at the end of the string.

9. Using B

This chapter deals with the process of compiling B programs. It also says a few words on programming style and ways to make your B programs more readable.

9.1 Compiling and Running

The B command compiles B source code. If given a source file, the B command calls the compiler to read the source and generate a set of object modules. It may also call the random library editor RANEDIT to place or replace modules in a library.

Unless there are fatal compilation errors, the B command then calls the TSS loader to prepare a load module, except when the Update= option is specified in the B command line. The load module prepared by the loader is not executed unless you specify the +Go option in the command line.

Only the most common uses of the B command are discussed here. For full details, see the explain file "expl b command".

9.1.1 Using the B Commmand

```
b srcfile
```

where `srcfile` is the name of a sequential file containing B source statements. If there are no fatal errors in compilation, the load module which the compiler creates is left in a random file called `".h"`, which is created as temporary if necessary. If you have a quick access permanent file called `".h"`, the compiler uses that file instead. This file is "grown" automatically by the TSS loader as required.

You can force the `B` command to execute the program after it is compiled by specifying

```
b +go srcfile
```

However, if your program plans to interpret a command line, it's better to invoke your program with the `GO` command. Typing

```
go arg1 arg2 ....
```

runs the load module in `".h"`, passing it the given command line.

There are a few other options in the `B` command which you may find useful. If you have too many **auto** variables for the default amount of stack space, you can specify a larger stack by using the `Stack=nnn` option, as in

```
b src.b stack=700
```

You can also use this option to decrease your stack-size from the default of 500 words, in the interests of saving space. However, it is inadvisable to use a stack-size of less than 300 words, since pre-execution routines like `.BSET` require about 300 words of stack to work properly.

9.1.2 Debug Tables

By default, the B compiler stores a number of debug tables in your program. These can be used later to detect error conditions and generate useful diagnostic messages. Fully debugged production programs, however, need not carry these debug tables when running. The `-Tables` option may be used to turn off the loading of debug tables, as in

```
b -tables src.b
```

9.1.3 Random Libraries

If you change one routine in a program, you don't want to be forced to recompile the whole program. To avoid this, you can store routines in a random library. That way, you need only recompile one routine or one group of routines to make a change. The `Update=lib` and `Library=lib` options of the `B` command provide an interface with the `RANEDIT` subroutine library editor.

To begin with, the command

```
b src.b update=/lib +clear
```

stores the routine or routines from `src.b` into the library `lib`; this library is created if necessary according to the usual B file accessing conventions. The `+Clear` option tells `RANEDIT` to clear and

To add new routines or replace old ones, use

```
b src1.b update=/lib
```

With this option, the loader is not called to prepare a load module. The object decks prepared by the compiler are simply stored in the specified library for later use.

Your program can obtain compiled functions from this user library (and others) by specifying the `Library=` option, as in

```
b src.b library=/lib
```

When the `B` command calls the loader, any libraries specified by the `Library=` option are searched for any external names which do not appear in the source file being compiled. Libraries are searched in the order they appear on the command line. To delete routines from a library, use the TSS command `RANEDIT` (see "expl ranedit").

9.1.4 B Command Options

The options for the `B` command have three possible forms:

```
keyword=string
+keyword
-keyword
```

In all cases, the keyword may be *abbreviated* using the following rule: in the explain file, a keyword is shown with upper and lowercase letters. A valid abbreviation must include those letters which are in uppercase, along with any other letters in the order in which they appear.

For example, valid abbreviations of the `Update=filename` option include

```
upda=filename
upte=filename
u=filename
```

Options may be combined on one command line and abbreviated, as in

```
b cmdlib/s/roff h=cmdlib/roff l=b/xlib -t +r
```

This command line uses the option `Hstar=filename`, which lets you designate the file into which the generated load module will be placed. It also uses the `Library=lib` option, the `-Tables` option, and the `+Respectcase` option which tells the compiler to pay attention to the case of alphabetic characters in the source code.

9.2 Compiler/Loader Interface

When processing a source program, the compiler generates several object decks and places them in a temporary file called `b*` (the input file for the loader). This file is also used as input to the random library editor `RANEDIT`, if `RANEDIT` is called.

When loading, the compiler always generates an object deck containing the B stack area (either 500

defined before the first function definition, if any.

A separate object deck is generated for each function. The deck includes a SYMDEF for the function and a SYMREF for each name mentioned in an **extrn** statement or used in a function call.

An object deck is also generated for each group of externals between function bodies and one for the group of externals after the last function body, if any. These decks include a SYMDEF for each defined external and a SYMREF for any external name referenced in an initializer list.

9.3 Line Numbers

B accepts line-numbered source files. If the first character of the first file is a digit, the compiler assumes the source file has line numbers. If so, on each line of the file, the compiler tries to find a line number by collecting numeric characters until a non-numeric character is found. This number is used in error messages pertaining to that line.

9.4 Source File Inclusion

If the compiler encounters a line of the form

```
%filename
```

in the source program, it suspends processing of the current file and begins collecting input from the specified file. When end-of-file is encountered in the included file, processing in the original file resumes at the line following the file inclusion request. Such included files may themselves contain `%filename` requests pointing to other files.

The `'%'` character must be the first character on the line, not just the first non- blank character. If the line has a line number, the `'%'` must immediately follow the line number.

The `filename` given may be in any of the forms acceptable in the TSS environment, such as

```
%temp
%/main.b
%fbaggins/dif.b
%fbaggins/s/dif.b
010%fbaggins/s/dif.b
```

The file inclusion feature lets you keep parts of a large module broken up into easily manageable files, while retaining the ability to compile the files together. File inclusion is often used to bring in a file of manifest definitions, such as TSS DRL equivalences; this avoids a lot of re- typing of manifests which a variety of programs use. A variety of convenient manifest inclusion files are kept under the catalog `b/manif`.

9.5 Compiler Directives

Any line whose first character is a `'#'` is assumed to be one of the compiler directives shown below. In each case, `"text"` denotes a string of characters which begins with any non-blank character.

```
#title text
```

```
#lbl text
```

places "text", truncated to eight characters if necessary, in the "deck name" field of any $OBJECT or $DKEND card written after the directive is encountered. If the compiler has not found a #lbl directive by the time the compiler generates an object deck, the compiler uses the current file name.

```
#ttldat text
```

places "text", truncated to six characters if necessary, in the "ttl" date field of any $OBJECT card written after the directive is encountered.

```
#copyright text
```

is taken as a comment.

```
#alias bname symdef
```

associates a second name with the name of a B external. bname must be an external B object that has already been defined in the B program. symdef is another name that will be defined as a SYMDEF at the location of the B object; in other words, it is a second name for the object.

The following example demonstrates all these directives.

```
#title tss login subsystem - .tslog
#lbl tlga
#ttldat 800524
#copyright (c) by Thinkage Ltd., 1990.
null(a) return(a);
#alias null .null
```

9.6 Readability

The easier you can read your program, the easier you can debug and modify it. You can make your programs more readable by using tabs and a clear brace bracket style.

When you type in a B program, it is a good idea to indent lines in order to emphasize the order of nesting of your source statements.

You can indent lines with spaces, but typing a lot of spaces can be tedious, not to mention that it's difficult to be consistent. We suggest that you use the ASCII tab character as one unit of indentation. (If your terminal doesn't have a tab key, you can get a tab by typing CTRL and I simultaneously.) If you use the SLIST command to list your source files, SLIST automatically expands tabs into the right number of blanks so that your program comes out indented the way you want. If you are using the FRED text editor to type in your source, FRED's O+OT command expands tabs into blanks on output to your terminal so that your program is indented properly.

It is not possible to specify alternate tab characters; you must use the ASCII tab.

There are many opinions of what is the "best" style for using brace brackets in B, but all experienced B programmers agree that a consistent brace style not only makes programs easy to read but helps in avoiding certain types of syntax errors.

statement occurs in an **if**, **repeat**, **while**, **do-while**, **for**, or **switch**, the opening brace `'{'` is placed on the same line as the keyword, and the closing brace `'}'` is placed on a line all by itself, indented so that it is directly under the first letter of the keyword. All intervening statements making up the single compound statement are indented by a tab stop. Examples of this style can be found throughout this manual. The advantage of this style is that opening braces are on the same line as their associated keyword, and closing braces are lined up with the associated keyword. This makes it easy to remember to include opening braces; it also makes it easy to see if there are too many or too few closing braces.

9.7 Some Pitfalls

Floating Point:

If you have a floating point value, it is not a good idea to say

```
if( floatval ) ...
```

because the code generated by the compiler checks to see if `floatval` is logically non- zero, rather than to see if it is equal to a floating point zero. Since a floating point zero is not a word containing all 0-bits, it is better to try

```
if( floatval #!= 0.0 ) ...
```

In general, floating point is tricky to use in B, since there is no type-checking. You must constantly watch out for erroneous constructs such as using `-3.0` when you really want `#-3.0`.

Confusing Operators

One of the most common mistakes for B programmers to make is to type an = (assignment) when they want == (logical comparison). The statement

```
if (a = b) ++i;
```

assigns the value of `b` to `a` and then does the `++i` if the assigned value was non-zero. This is quite a bit different from

```
if (a == b) ++i;
```

which does the `++i` if the value of `a` equals the value of `b`. Very often, you want the second form but type the first. When you make this kind of mistake, your variables are suddenly assigned values they likely shouldn't have, and many unwanted things may happen.

A similar mistake comes from confusing `&` (the *bitwise* AND) and `&&` (the *logical* AND). Consider the situation of `a` equal to `1` and `b` equal to `2`.

```
a & b
```

does a bitwise AND, yielding a value of zero (since `a` and `b` have no bits on in the same bit positions).

```
a && b
```

does a logical AND, yielding a value of one (since `a` and `b` are both non-zero).

for mistakes of this nature, since they are easy to make and can lead to some very odd results.

String Constants

Here is a common pitfall in the use of string constants. If you say

```
auto x[20];
x = "a string";
```

the word `x` is changed from a pointer to a vector to a pointer to the storage occupied by the given string. In this way, you lose the ability to address the 21 words originally reserved for the vector. What you really want to say is

```
auto x;
x = "a string";
```

Alternatively, if you want to initialize the vector with the string, you should use the library function `CONCAT` to copy in the string:

```
auto x[20];
concat( x, "a string");
```

Another approach would be to define `x` as an initialized external in the declaration

```
x {"a string"};
```

Vector and String Size

When you declare a vector of size `n`, you know that it actually occupies `n+1` words, because the vector is indexed starting at zero. Most library routines that take the size of a vector as an argument observe exactly the same convention.

In practice, if you need a vector of size `n`, you declare it to be of size `n`, and then ignore the zeroth or the `n`th word. Thus a **for** loop indexing through the vector might run in either of two ways:

```
for( i = 0; i < n; ++i ) ...

for( i = 1; i <= n; ++i ) ...
```

You can also index into strings using library functions, and the origin is zero just as with vectors. However, at first it may appear to you that a string with `n` characters in it has length `n`, rather than `n+1`.

For example, the string `"abcdef"` has six characters and its length is six. But recall that, by definition, a string is terminated by a `'*e'` character, which you do not see. If you include the trailing `'*e'` in the count, then a string of length `n` actually contains `n+1` characters.

The trailing `'*e'` should also be taken into account when you are computing how many words of memory a string takes up. A string like `"abcd"` for example requires two words of memory; in the first word you have `'abcd'` and in the second you have the `'*e'`.

All library functions which require the length of a string need the number of characters, not including the `'*e'`. The library function `LENGTH` returns just that number.

9.8 Library Explain Files

This manual is only one part of the documentation for the B compiler. The explain files for B provide a good deal of information not contained here, particularly concerning library functions. People who are planning to do any significant amount of programming in B should learn how to use the explain files.

"expl b index" gives a short description of the major explain files for B. "expl b lib index" lists the library functions which have their own explain files. Because the library index is rather long, users may prefer to use the LOCATE TSS subsystem to look for pertinent functions rather than read through the entire index. For example,

```
locate "string" expl/b/lib/index
```

searches for occurrences of `"string"` in the library index file, and lists all the functions with the word `"string"` in their descriptions. The command

```
locate "line num" expl/b/lib/index
```

searches for routines which deal with line numbers, and so on. While this is not an infallible way to find the name of the function you want, it can save a lot of time.

9.8.1 BOFF

Newly written programs seldom work the first time they are run. If errors occur during compilation, the B compiler can issue a number of diagnostic messages, many of which appear in Appendix C. Once you have cleaned out compilation errors, it is still quite possible that there will be errors during execution of the program. When run, your program could do nothing, it could go into an infinite loop, or it could abort with a memory fault or some other hardware-detected error.

The BOFF subsystem is a useful tool for detecting and correcting errors in B programs. BOFF stands for "B Obscure Feature Finder". You can use BOFF to inspect and/or patch the core image file of your program as prepared by the loader, to monitor the progress of your program as it is running, or to inspect a TSS dump file called `abrt` after your program has failed.

For more information on BOFF debugging capabilities, see "expl boff" and "expl boff manual".

Appendix A: Escape Sequences

There are two sets of escape sequences, one for use inside string or character constants, and the other for use outside.

A.1 String and Character Constant Escapes

Escape sequences are used in character constants and strings to obtain characters which for one reason or another are hard to represent directly. Here are the escapes:

`*e`
: end of string (ASCII NUL = 000)

`*(`
: `{` - left curly brace

`*)`
: `}` - right curly brace

`*<`
: `[` - left square bracket

`*>`
: `]` - right square bracket

`*t`
: horizontal tab

`**`
: `*`

`*'`
: `'`

`*"`
: `"`

`*n`
: new-line

`*r`
: carriage return (no line feed)

`*f`
: ASCII formfeed

`*b`
: backspace

`*v`
: vertical tab

`*x`
: rubout (octal 177)

`*`*nnn*
: *nnn* is 1-3 digit octal number, standing for the ASCII character with that octal value

A.2 Source Code Escapes

The following are escapes used outside character and string constants. They are intended for use on terminals whose keyboards do not have some of the characters used by B. If you use the FRED text editor, it is better to use the FRED escapes for these characters, so that if you move to an ASCII terminal you can see the characters the way they ought to appear.

`$(`
: `{` - left curly brace

`$)`
: `}` - right curly brace

`$<`
: `[` - left square bracket

`$>`
: `]` - right square bracket

`$+`
: `|` - or-bar

`^` - up-arrow (or cent-sign)

`$a`

`@` - at-sign

`$'`

`` ` `` - grave accent

Appendix B: Binding Strength of Operators

Operators are listed from highest to lowest binding strength; there is no order within groups. Operators of equal strength bind as indicated, either left to right (denoted by [LR]) or right to left (denoted by [RL]).

```
[LR]  name const arr[expr] func(args) (expr)
[RL]  ++ -- * & - ! ~ #- # ## (unary)
[LR]  >> <<
[LR]  &
[LR]  ^
[LR]  |
[LR]  * / % #* #/ (binary)
[LR]  + - #- #+
[LR]  == != > < <= >= #== #!= #> #< #<= #>=
[LR]  &&
[LR]  ||
[RL]  ?:
[RL]  =  +=  -=  etc.  (all assignment operators)
```

Appendix C: B Compiler Error Messages

This is a list of the most common diagnostics generated by the B compiler.

In each description, `nn` means a line number, while `name` is some identifier name. The name of the source file is usually given as well.

Any message not preceded by `Warning:` is a fatal error. If there is a fatal error, neither the loader nor the random library editor will be called.

C.1 Diagnostics

`syntax error at line nn [in file <name>]`
: This is the most common diagnostic and it could mean almost any kind of error. Most often, it means a semi-colon is missing or the number of opening curly braces `'{'` does not match the number of closing curly braces `'}'`. In the second case, the line number is the number of the last line in the last file being processed plus one. This may also occur if you do not end a string constant, character constant, or comment. You also get this message if you use a keyword in an inappropriate context, if you neglect to define a manifest, or if you attempt to redefine a manifest.

`<identifier> undefined in function <name>`
: An identifier in the named function has not been referenced by an **extrn** or **auto** statement and has not been used as a label. The line number given is the last line of the function being compiled.

`warning: /* inside comment ...`
: This is a warning only, but there will probably be a syntax error later on, since comments may not

comment inside a comment, then the compiler tries to compile the remainder of the outer comment.

`end of file in comment`

This usually indicates that you forgot to end a comment with the terminating `*/`.

`warning: newline in constant not preceded by '*'`

The most probable cause is that you forgot to terminate a string or character constant with the appropriate delimiter. If this is the case, you will surely get a syntax error later. If you want a "real" new-line inside the constant, but no warning, use the escape sequence `'*n'`. If the constant is a string constant which is too long to fit on one line, put a `'*'` in front of the new-line, and the new-line will be discarded. If you actually see the warning, the new-line is kept in the source code.

`invalid octal constant`

An octal constant (an integer with a leading zero) contains a character other than the digits `0-7`.

`character constant too long`

A character constant may not contain more than four characters (but some of the characters may be represented by two-character escape sequences).

`bcd constant too long`

A BCD constant contains more than six characters.

`exponent too large in constant`

The exponent of a floating point constant is too large or too small to represent in the hardware.

`attempt zero division`

In evaluating the constant part of an expression, the compiler found the right operand of a division or remainder operator to be the constant zero.

`invalid & prefix`

The `&&` operator has been used in an invalid context, such as `&&x=y`.

`warning: found ++r-value`

`warning: found --r-value`

You get this if you say something like `++x++`.

`invalid $ escape sequence`

An escape sequence beginning with `$` is not known to the compiler.

`invalid unary operator`

The compiler discovered you trying to use a binary operator in a unary manner.

`invalid =`

`invalid *=`

`invalid >>=`

The expression on the left hand side of an assignment operator does not have an Lvalue.

`invalid ++`

`invalid --`

The expression operated upon by the `++` or `--` operator does not have an Lvalue.

`invalid label`

A name used as a label has previously been declared as **extrn** or **auto** in the current function.

`invalid break`

The compiler found a **break** statement which was not inside a **for**, **while**, **do-while**, **repeat** or **switch** statement.

`invalid next`

The compiler found a **next** statement which was not inside a **for**, **while**, **do-while** or **repeat** statement.

`invalid constant expression`

`invalid operator`

This is one of those "cannot happen" messages. If it does happen, please submit an error report.

`auto array too large`

You attempted to declare an **auto** vector with a dimension greater than 1000 words. It is better to use an external vector or else `GETVEC` the space, since **auto** variables are allocated on the stack and stack space is limited.

`extrn array too large`

This usually happens because you declared an external vector incorrectly, as in `x[3.0];`.

`invalid case`

A **case** label is not inside a **switch** statement.

`invalid default`

A **default** label is not inside a **switch** statement.

`default already supplied`

More than one **default** label in a **switch** statement.

`invalid case operator`

The only operators permitted in a **case** are <, >, >=, and <=.

`%filename ignored-too many open files`

This usually happens when you include a file which includes itself.

`bad input character: <ddd> (octal)`

A character encountered in the input stream outside of a string or character constant has no meaning for the compiler. This might be a backspace or some control character typed in by mistake. Since it may be a non-printing character, the value of the offending character is displayed in octal.

`rewrite this expression`

A subscripting expression is too involved for the code generator to handle. Try breaking up the expression into more than one statement.

`manifest nesting too deep`

This occurs when you have manifest constants whose evaluation involves other manifest constants. It may also happen if you have too long a series of manifest definitions, each of which is defined in terms of the previous manifest. This is ok in GMAP, but not in B.

`warning: program size > 32k`

One of the object decks generated requires more than 32K words to load. You may get this warning if you declare several very large external vectors. However, it might also mean the loader will be aborted by TSS due to "not enough core to run job".

`expression too complex`

`no tree space`

`no stack space`

An expression is too complex for the compiler to evaluate. Try simplifying it by breaking it up into two or more expressions.

`The constant <ddd> occurs in two case labels`

The same constant appears in more than one **case** label in a **switch** statement. The value of the offending constant is printed in decimal. The upper range <ddd> overlaps the lower range `<ddd>`

The compiler has detected overlapping bounds inside a **switch** statement. The values of the bounds are displayed in decimal.

`The constant <ddd> is in the range <ddd>::<ddd>`

In a **switch** statement, the compiler has detected a **case** constant which is in the range of a range

case, the message shows the bounds generated for the relation. For example, the bounds for `case>0:` would be `1::34359738367`.

`Initializers nested too deeply`
: An external declaration has initializers in braces nested to a depth greater than seven.

`external redefined`

`auto variable redefined`

`label redefined`

`auto array name redefined`
: The compiler has found an attempt to redefine a symbol which has already been defined in the current function body.

`no space for symdef`
: There are too many external definitions; try dividing them into two groups by either compiling them separately or placing a function in between. This error is almost never encountered.

`no space for symref`
: There are too many external references in a function definition; try simplification. This error is almost never encountered.

`warning: #<text> ignored`
: A line beginning with `'#'` does not contain a recognizable compiler directive. The line is ignored.

C.2 TSS Loader Warning Messages

`<w> name undefined`
: This is a loader message, indicating that an external variable referenced by one of your functions or a library function, remains undefined after all libraries have been searched. If your program references the named external, it will abort with a MME fault in TSS, or with a USER'S L1 MME GEBORT in batch.

`<w> name loaded previously`
: The loader has discovered a function or external with the same name as one already loaded. The most probable reason is that you have two or more different names which, when truncated to six characters, end up being the same. The loader ignores all but the first. Make sure all your externals and function names are unique in their first six characters.

Appendix D: Partial Index of B Library Routines

The following is a list of some of the routines currently in the B library. The list is by no means complete; we have simply chosen some of the more commonly used B library functions so that the reader can get an idea of what the B library can do.

`.ABBRV`
: check for valid abbreviations

`.ABORT`
: print an error message and abort

`.BSET`
: parse string into arguments

`.READ`
: reference or change the current read unit

set tabs for the current output unit

`.WRITE`
reference or change the current write unit

`ABS`
absolute value of an integer

`ALLOCATE`
simple garbage collecting storage allocator

`ANY`
check if a character appears in a string

`APPLY`
call arbitrary function with arbitrary arguments

`ASCBCD`
convert an ASCII string to a BCD vector

`CALLF`
call FORTRAN program from B routine

`CALLFF`
call FORTRAN function returning floating point value

`CLOSE`
close currently open unit

`COMPARE`
compare two B strings

`CONCAT`
concatenate a series of strings

`COPY`
copy contents of one vector into another

`DATE`
return current date in ASCII

`DATESI`
convert date in ASCII string to a standard form

`EOF`
test input end of file, or write output end of file

`EQUAL`
compare two strings for equality

`ERROR`
type an error message, then exit

`EXIT`
end job and return status

`GETCHAR/GETC`
read a character

`GETDATE`
turn ASCII date into the form mm/dd/yy

`GETLINE`
read a line from an input unit

`GETSTR`
read a string from an input unit

`GETUMC`
get userid of current user

dynamically allocate a vector

`HIST`
: the B histogram package

`LENGTH`
: return the length of a string

`LINUMB`
: return the current line number of a file

`LOWERCASE`
: turn alphabetics in a string to lower case

`MAX`
: maximum value of list of integers

`MIN`
: minimum of a list of integers

`NARGS`
: return number of arguments to a function

`NOBRKS`
: count number of times break key was hit

`NULLSTRING`
: check for null string

`OPEN`
: open a file or string for I/O

`PRINT`
: do a PRINTF into a string

`PRINTF`
: formatted print

`PROMPT`
: prompt for input at terminal

`PUTCHAR/PUTC`
: output a character

`RAND`
: generate pseudo-random numbers

`READ/WRITE`
: unit-oriented block I/O

`READF`
: formatted character stream input

`RESET/SETEXIT`
: non-local goto

`RLSEVEC`
: release memory obtained by `GETVEC`

`SCAF`
: convert ASCII pathname to BCD catfile stack

`SCAN`
: extract delimited substring of a string

`SHELLSORT`
: a Shell sort

`SIDATE`
: convert standard date to ASCII string

wait for specified interval

`STRIP`
: cause line numbers to be stripped on input

`SYSTEM`
: execute a TSS command

`TABSET`
: specify tab stops for the current output unit

`TIME`
: get time in pulses, or convert it to a string

`TRIM`
: trim trailing blanks off a string

`ZERO`
: initialize a B vector to some value

Appendix E: Interface with FORTRAN Subroutines

The `CALLF` function lets you call Fortran 66 subroutines, or any routine which uses the GCOS CALL conventions. However, routines called in this way must not be called recursively.

```
intval = callf( &routine, &arg1, &arg2, ... );
```
: `CALLF` converts its arguments into the form of a standard GCOS CALL macro and calls the named `routine`. `routine` must be referenced in an **extrn** statement and must be passed *by address* as shown. The arguments must be passed by address also. Thus if an argument is not a vector pointer, you must say `&arg`. Constant values must be assigned to a temporary before being given to `CALLF`, since you can't say something like `&2`. The value of a `CALLF` is the logical or integer value returned if the routine called is a function subroutine.

```
floatval = callff( &routine, &arg1, &arg2, ... );
```
: `CALLFF` works in exactly the same way as `CALLF`, except that it must be used for function subroutines which return a floating point result.

As usual, you are responsible for ensuring the correct number and type of arguments are passed.

Appendix F: DRLs and MMEs

For those rare occasions when you need to perform system calls directly, here are the functions provided to let your B program execute DRL or MME system calls in a reasonable manner:

```
drl.drl(number [,arg1, arg2, ...] )
```
: allows direct access to the DRL functions. `Number` is the DRL number to be executed, and any following arguments are the words to follow the DRL. The A and Q registers are set to the values of the externals DRL.A and DRL.Q respectively. After the DRL has been executed, these externals are set to the contents of the A and the Q. It is possible to use a DRL that requires an error exit or a place to go to, since the DRL is executed in the stack, using the stack pointer of the caller. For example, consider the following:

```
UPHALF = 0777777000000;
      ...
auto buf, vec[2];
      ...
buf = getvec(600);
open( 9, "gcos3/gcos-hi-use", "r" );
aft.name( 9, vec );
vec[2] = {grave}.mbrt3{grave};
drl.drl(RESTOR_, buf<<18 | 1, buf<<18 | 1,
      (tra&UPHALF)|(buf+512));        tra:
      printf( "%24b*n", buf+4+STATUS*4 );
rlsevec(buf,600);
```

This code sequence obtains a batch error message by locating it in the batch error message module. It uses the value of a label to supply a return address to DRL RESTOR. In this particular case, you could have called the library function `.RESTR`, replacing the call to `AFT.NAME` and `DRL.DRL` with

```
.restr(9,`.mbrt3`,1,buf);
```

Note that the above example uses the manifests `RESTOR_` and `STATUS` from `b/manif/drls` and the user manifest `UPHALF`.

`mme.mme( number [,arg1, arg2, ... ] );`

operates in exactly the same way as `DRL.DRL`, except that that A and Q register values are stored in externals called `MME.A` and `MME.Q`.