UNIVERSITY OF HOUSTON

UNIVERSITY COMPUTING CENTER

HOUSTON, TEXAS 77004

B Reference Manual

Table of Contents

User's Reference to B for Honeywell Series 6000/66

By R.P. Gurd

University of Waterloo,

Waterloo, Ontario, Canada.

0. Introduction.

This manual describes the programming language B accepted by the compiler written at the University of Waterloo by R. Braga. It also introduces the runtime package written at the University of Waterloo, largely by T.J. Thompson.

A derivative of BCPL, B was designed and first implemented by D.M. Ritchie and K.L. Thompson, of Bell Telephone Laboratories, Inc., Murray Hill, N.J. The original implementation of the runtime package is due to S.C. Johnson, also of Bell Labs.

The present version of the compiler differs from the original by incorporating an expanded SWITCH statement, adding floating point operators, adding proper logical operators, and altering the order of evaluation of operators.

The runtime package works in both TSS and batch and will read almost any "media code" found in the GCOS environment. It uses EIS instructions whenever it is appropriate. Note that the language itself has no constructs for input/output; all i/o is done by function calls.

B is a *typeless* language, in which the compiler always assumes the type of a variable is suitable to the *operator* acting upon it. B has a large set of operators, providing integer, bitwise, logical and floating point operations.

The machine word is the basic unit of computation. The word size on the Honeywell Series 6000/66 machines is 36 bits.

A B program consists of procedures called *functions*. Any function may call another function or, since local variables are allocated on a stack, call itself recursively. All functions may selectively access a global pool of *externals*.

A function is composed of a set of one or more statements. A statement is composed of permissible combinations of keywords and expressions.

Although this is a reference manual, and definitely not a tutorial, it is organized such that you will often find examples which involve material covered later on. This is intentional. Such examples should be ignored at first reading, but will hopefully be beneficial when you refer to the manual again.

You should try to keep your first efforts at B programming as simple as possible, in order to minimize the difficulties you may encounter. If you are ever unsure of some feature of the language or run-time package, try it out by writing and running a simple little program which exercizes only that feature. The time you take to do so may save you a lot of trouble in the long run.

Before starting on your first major B program, you would be well advised to have a close look at the source code of a well-written B program or two in order to get some idea of exactly how things are done.

1. Basic symbols.

B is very much oriented towards use of the ASCII character set in which each character occupies nine bits (four characters per word). There is support for representing character constants in the BCD character set, in which one character occupies six bits (six characters per word).

Certain characters, such as '{' or '}', do not appear on some terminal keyboards. Escape sequences for these are defined in Appendix A.

The compiler treats its input as an unbroken stream of characters. Any form feed, tab or newline character is converted to the space character, except when it occurs inside a string or character constant. Newlines are counted, so the compiler can tell you on what line it detected an error. Line length may be arbitrary. The compiler knows nothing about any "sequence field", such as is supported by certain card-oriented compilers.

Line-numbered source files are permitted. If the first character of a line is a digit, the compiler assumes the program being compiled has line numbers. If so, on each line of the file, it attempts to form a line number by collecting numeric characters until a non-numeric character is found. The number is used in error messages pertaining to that line.

1.1. Identifiers.

An identifier or name is formed from the characters a-z, A-Z, 0-9, underscore ('_') or dot ('.'), and must begin with a non-digit.

Names may be arbitrarily long, but only the first eight characters are significant. For external names or external references, only the first six characters are significant, due to the restriction imposed by the GCOS and TSS loaders.

Normally, the compiler ignores case distinctions, so that the identifiers "SUM", "sum" and "Sum" would be considered to be the same thing. You may specify an option to the compile command which forces the compiler to respect case distinctions, but then all keywords **must** appear in lower case.

1.2. Comments.

The beginning of a comment is signalled by the appearance of a "/*" in the input stream. It is ended with the first occurrence of a "*/" any number of characters or lines later. For example,

```
/*
 *  this is a comment
 */
```

Comments may **not** be nested.

1.3. Keywords.

B uses 15 keywords, which may be categorized as follows:

1) identifier scope keywords:
   AUTO EXTRN
2) execution flow control keywords:
   IF ELSE FOR WHILE REPEAT SWITCH DO
3) transfer keywords:
   RETURN BREAK GOTO NEXT
4) switch statement keywords:
   CASE DEFAULT

The compiler does not allow you to use any keyword as an identifier. In particular, beware of inadvertently using "next" as an identifier.

1.4. Constants.

You may define octal, decimal, floating point, ASCII character, BCD character, or string constants in your program. The form of a constant is well-defined, in that it is possible to unambiguously differentiate between the various types of constants.

1.4.1. Decimal constants.

A decimal constant consists of an integer number, which may not contain leading zeroes. For example,

```
25   4737  981   32
```

1.4.2. Octal constants.

An octal constant consists of an integer number, preceded by a zero and formed only from the digits zero through seven. For example:

```
01   077   026   0400000 0777777777777
```

1.4.3. Floating point decimal constants.

A floating-point constant is any number containing a decimal point. It must not begin with a decimal point, but may have leading zeroes and may be followed by the letter 'e' and a possibly signed integer exponent. Examples:

```
3.2  1.    0.5   1.e5  3.e5 4.987e-2
```

1.4.4. ASCII character constants.

An ASCII character constant consists of from one to four characters inside single quotes. The result is a word which contains the internal form of the ASCII characters, right-adjusted and left-padded with zero bits. Some examples:

```
'a'  'abc' 'abcd'
```

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

1.4.5. BCD character constants.

A BCD constant consists of from one to six characters enclosed by grave accent characters. The result is a word containing the characters transliterated to BCD, right justified, and left-padded with zero bits. Characters which do not have an exact equivalent in the BCD set are converted to BCD blanks. Here are three examples of BCD constants:

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

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

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

Note that the runtime package provides functions to transliterate between ASCII and BCD and that the i/o function PRINTF will take a BCD format specification.

1.4.6. String constants.

A string constant is any string of characters enclosed in double quotes. For example:

```
"this is a string"
""
"the above is the null string"
```

When processing a string, the compiler packs the characters of the string four per word and always appends one extra character, an ASCII null (000), to mark the end of the string.

The *value* of a string is quite different from the other types of constants. The value of a floating-point, octal, decimal or character constant is a word containing the internal representation of the given constant. The value of a string constant is a word containing a *pointer* to the string in the lower 18 bits.

In constructing the string, the compiler gobbles all characters it sees, translating escape sequences if necessary, until it finds a closing, unescaped double quote. The rule which says tabs and form feeds are ignored does not, of course, apply in this case, but real newlines (as opposed to escaped newlines) are treated specially. If a real newline is preceded by a '*', both '*' and newline are thrown away, so you can enter a very long line. If the newline is not preceded by a '*', it is kept, but a warning message is issued, on the grounds that you probably forgot the closing string quote. To get a newline in the string without drawing a warning, use the escape '*n'.

1.4.7. Escape sequences.

Escape sequences, beginning with the character '*', are defined to allow you to use, in a string or character constant, characters which would otherwise be inconvenient or impossible to enter. For example, if you wanted to place a double quote inside a string constant, you would use the escape *". The special end-of-string character (ASCII null) is escaped as '*0'. The newline character is escaped as '*n'. A newline is taken as a carriage return and a line feed when output to a terminal. Arbitrary nine-bit characters can be generated with "*#nnn", where nnn is one to three octal digits. The complete set of escape sequences is given in Appendix A.

1.5. Source file inclusion.

If the compiler encounters in the source program a line of the form

```
%filename
```

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 next 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 file name 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
etc.
```

This allows you to keep parts of a large module broken up into easily manageable files, while retaining the ability to compile the files together. File inclusion is also often used to bring in a file of manifest definitions, such as TSS Derail equivalences, which a variety of possibly unrelated programs might find useful.

1.6. 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>
```

will place "<text>" in the comments field of any $ OBJECT card written from the time the directive is encountered.

```
#lbl <text>
```

will place "<text>", truncated to eight characters if necessary, in the "deck name" field of any $ OBJECT or $ DKEND card written from the time the directive is encountered. If this directive has not been encountered when it comes time to generate an object deck, then the current file name is used instead.

```
#ttldat <text>
```

After being truncated to six characters if necessary, "<text>" is used to fill in the "ttl" date field of any $OBJECT card written from the time the directive is encountered.

```
#copyright <text>
```

is taken as comments.

Here is an example of the use of all four directives:

```
#title tss login subsystem - .tslog
#lbl tlga
#ttldat 771209
#copyright (c) by the University of Waterloo, 1977.
```

2. The building blocks of a B program.

A complete B program consists of at least one, but usually many, modules. A "module" is any of the following:

1) a manifest constant identifier definition.

2) an external (global) variable definition.

3) a function body definition.

Modules may appear, and the different types interspersed, in any order at all, with the sole proviso that the definition of a manifest identifier must appear before the identifier is first used.

Manifest definitions are used to associate a name with a compile time constant.

External definitions are used to create a global pool of possibly initialized identifiers. This pool might be used to declare large blocks of memory, or to declare identifiers that must be accessible to more than one B function. Since an external is global in scope, any B function may use it, but only after declaring its intention to do so in an EXTRN statement.

A function definition is used to declare a component of the executable code of the program. The definition includes the name the function will be called by, the arguments it will be called with, and the statements which define what it will reference and what it will do.

2.1. Manifest constants.

A manifest constant has the general form

```
name = text;
```

where "name" is any valid identifier and "text" is the collection of characters between the equals sign and the semi-colon.

When a manifest is defined, the compiler enters the identifier in a symbol table associating it with the "text", which it keeps in an internal buffer. Absolutely no processing of the "text" is done at the time of definition.

When the compiler reads an identifier, it first checks to see if the identifier is a manifest. If so, the action taken is to substitute the text of the manifest for the identifier. For this reason, it is not possible to speak of redefining a manifest and any inadvertent attempt to do so will usually result in a syntax error. Substitution effectively takes place before the syntax analyzer scans the line. Manifests may be used anywhere, including inside later manifest definitions.

Because the compiler does not analyze the text of the manifest until substitution takes place, it is possible for the text to refer to a manifest which is defined after it, as long as the definitions of both appear before the first use.

The use of manifests has no effect on the order of expression evaluation. For example, look at the definitions

```
A = 1;
B = A+A;
C = B*B;
```

When "C" is used somewhere else in the program, the compiler will actually get "1+1*1+1", which will be evaluated as three and not four, as one might mistakenly assume.

The compiler permits nesting up to 10 levels deep of manifests inside other manifests.

Normally, however, you will find manifests quite natural to use. Here are some examples of manifests:

```
VECSIZE = 63;
vec [ VECSIZE ] ;
...
for( i = 0; i <= VECSIZE; ++i ) sum += vec[i];
```

The manifest "vecsize" is used to establish the size of the vector "vec" at compile time, and later used to control iteration in a FOR statement.

```
/* binary list structure */
NULL = -1;     EMPTY = 0;
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 */
```

Here, manifests are being used to make the code involved in traversing a binary tree more meaningful. Manifests are often used in this way to give names to the elements of a structure, which may be an array of fixed size, or a dynamically allocated block of storage.

A common convention, used in program examples, is to differentiate manifest identifiers from other identifiers by always showing the manifest identifier in upper case, and to show all others in lower case. It is also considered good practice to group all manifests for a program together at the beginning of the source code.

2.2. External definitions.

We will first look at the possible forms of external definitions, then look at several examples.

If an external is defined and possibly initialized in more than one place, the first one encountered during loading is the one which is used.

These are the possible forms of external definitions:

`name;`

A single word is allocated and initialized to zero.

`name { ival };`

Name is defined as a single word and initialized with the single value ival.

Ival may be any legal constant expression, in which case name has the value of its result. A constant expression may be either a string constant or an expression formulated with any legal combination of numeric or character constants, binary operators, unary operators and parentheses, following the rules in the chapter on expressions. Alternatively, ival may be a name, in which case the value of the ival is a word containing, in the lower 18 bits, the address of the name given as the ival. A function name may be used.

`name { ival, ival, ... };`

Allocates space for as many words as there are ivals.

This is in effect a vector which does not have a word set aside as a pointer to it; its address is "&name", rather than just "name", which in this case refers to the first ival. This is the way a vector is set up in FORTRAN, but is not the same as a B vector.

name [const-expr] ;
This is the first of several forms of B vector declarations. Name is defined as a pointer to a vector whose length is a number of words which is the value of the constant expression plus one (since all B vectors start subscripting at zero). The zeroth element of the vector is initialized to zero; the initial contents of the remaining cells are undefined.

The const-expr in brackets may be any expression which is a legal combination of numeric or character constants, unary operators, binary operators and parentheses. It is up to you to make sure the value of the expression is reasonable, since the compiler's grammar lets it accept things like floating point constants and negative numbers, which give absurd results. For all practical purposes, "const-expr" must be such that it gives an integer result.

name [] { ival, ival, ... } ;
Name is defined as a pointer to a vector whose length is the number of initial values.

name [const-expr] { ival, ival, ... } ;
Name is defined as a pointer to a vector whose length is the maximum of the result of the constant expression plus one and the number of initial values. The contents of those elements of the vector which do not have corresponding initial values are undefined.

Vectors declared as externals (or as AUTO variables) are always allocated so that the zeroth word of the vector immediately follows the word containing the vector pointer.

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 a numeric constant may be prefixed by an integer unary minus sign.

Here are some examples of external definitions:

```
a { 10 };
```
One word of storage is allocated, initialized to the decimal constant 10 and associated with the name "a".

```
b [ ]  { 'ab', 'cde', 'fghi' };
```
One word is associated with the name "b" and initialized with a pointer to a vector of three words. The first element of the vector, referred to as b[0], is initialized with the character constant 'ab'. The other two elements, b[1] and b[2], are initialized with 'cde' and 'fghi', respectively.

```
c { 'ab', 'abc' };
```
"c" is defined and associated with a word containing 'ab'. The word immediately following is initialized to the constant 'abc', but is not associated with any name. This is in effect a vector which does not have a word set aside as a pointer to it.

```
d[63];
```
Defines "d" and associates it with a word containing a pointer to a vector of 64 words, each of which is initialized to zero.

```
e[10] { a, b, c, d };
```
Declares "e" and associates it with 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 and initialized with a pointer to the storage occupied by the string constant "a string".

```
g[] { "pascal/library", "pascal/compiler", -1 };
```
This sets up a vector of strings with an end marker. Defines "g" and associates it with a word containing a pointer to a vector of three words. The cell "g[0]" is initialized with a pointer to the storage occupied by the string constant "pascal/library". The cell "g[1]" is initialized in a similar manner, while "g[2]", the last element of the three-word vector, is initialized to the decimal constant -1.

B does not allow you to explicitly declare arrays with more then one dimension. Usually if you need more than one dimension, you build it at run time by calling the library function GETMATRIX, which will obtain storage, construct the necessary edge vectors and return a pointer to the array. However, in spite of the fact that you cannot declare such an array, it is possible to construct one as an 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. The maximum nesting depth is seven. For example,

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

In this case, "x" ends up being 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.

2.3. Function definition.

B functions serve a purpose similar to the subroutine in FORTRAN or the procedure in ALGOL. The mechanism of the function call involves very little cost in overhead and permits recursion.

A working B program always contains at least one function, called MAIN, and usually others, since the language is designed to encourage modular or structured programming.

The general form of a function definition is

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

The name must be a valid identifier and is automatically defined as an external by the compiler.

The formal arguments consist of a possibly empty list of identifers separated by commas. Each argument is implicitly declared as an automatic (local) variable and storage for it is allocated on the runtime stack frame. Note that, although you may not declare a vector as a formal argument, you can always use an argument in a subscripting operation, as if it were a vector pointer.

"Statement" defines what actions the function will take. Most often, it is a compound statement, consisting of a set of statements enclosed by braces. The rules for formulating statements are presented in the next chapter.

When writing the code for a function, there are a few things you should keep in mind.

The caller will always pass its arguments strictly by value. This means that altering an argument has no effect on the state of the caller. However, if an argument is a pointer, you can change the state of the caller by indirecting through the pointer using either the unary indirection operator or subscripting.

The function may at any time return a one word value

using the RETURN statement. The caller and the callee do not have to agree on whether or not a value is returned. If a value is returned, but not expected, the value is ignored. If a value is expected, but not returned, the value is undefined.

A function can determine the actual number of arguments it is called with by invoking the library function NARGS. For instance, the statement

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

would assign to the variable "x" the number of arguments supplied to the current invocation of the function.

The availability of NARGS lets you write functions which may take a variable number of arguments. Most of the time, this means that such a function is called with fewer arguments than are defined for it, in which case one of the first things such a function does is to establish default values for the arguments it does not have.

The other case, in which a function is called with more arguments than are defined for it, is somewhat trickier, and should be avoided, unless you know precisely what you are doing.

The function called MAIN is the entry point to your program from the B runtime initialization routine. If not present, the TSS loader prints the message:

```
<w> main   undefined
```

For details on how your main function is invoked, see the chapter on the run time library.

3. Statements.

Statements are used to define the actions taken by a B function. They may appear only in the body of a function definition. In certain cases, the definition of a statement is recursive, in that a statement may appear inside a statement. In this chapter, you will see that in many cases, one may use an expression in a statement. Since the rules for formulating expressions are discussed in the next chapter, we merely note here that an expression may be a statement, but a statement may not appear in 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.

In the formal definitions which follow, reserved words are underlined and parentheses, where shown, are required. Also, "statement" implies either a statement ended by a semicolon or else a compound statement surrounded by braces.

3.1. 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.

3.2. Expression statement.

```
expression;
```

Any valid B expression followed by a semicolon is acceptable as a statement. To be meaningful, the expression will usually involve an assignment operation or function call, as in

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

but the compiler will happily accept 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.

3.3. Storage declaration.

3.3.1. Storage types.

Before discussing the statements pertaining to storage declaration or reference, we will briefly look at how storage is allocated in a B program.

External storage consists of the global pool of externals declared in the manner described previously. For a function to use one of these externals, the name must be referenced in an EXTRN statement.

Automatic storage consists of local variables which are created anew on the runtime stack each time the function is called and which disappear when the function returns. Automatic storage is unique to each invocation of a function.

Internal (local static) storage is allocated within a function body and is common to all invocations of a function. The label, which is never explicitly declared, is the only permitted instance of internal storage.

Constants used inside functions are allocated as internal storage, but the compiler will not accept constructs that would result in directly changing a constant's value.

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. This free area is automatically grown as required up to the limits imposed by the operating system.

Any identifier used in a function body must be a formal argument, a label, or previously referenced in an EXTRN or AUTO statement. The only exception is a function name used in a function call, since the compiler automatically types as external any name immediately followed by a left parenthesis '('.

AUTO and EXTRN statements may appear anywhere in a function body, but you should group them at the beginning of the function.

3.3.2. Extrn.

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

This statement allows the function to begin using the names previously defined as externals (see chapter 2). Although an identifier may be externally declared as a vector, you should not indicate this in the EXTRN statement, since B lets you use any cell in a subscripting operation.

3.3.3. Auto.

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

The AUTO statement is used to declare local storage, which is unique to each invocation of the function. For example,

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

A vector declaration is legal in an AUTO statement, but the size of the vector must be a constant expression, since it is established at compile time.

Const-expr is any legal combination of numeric or character constants, unary operators, binary operators and parentheses. Make sure what you use is sensible, because the compiler accepts constructs like "auto x[-1]" which lead to undefined results. Normally, one would expect a constant expression which is not a simple numeric constant to involve a manifest constant, as in

```
MAX = 10;
 ...
auto x[MAX*2], y[MAX + 7];
```

If you need dynamic vector allocation, you must use the library function GETVEC to obtain it from the free storage area.

An AUTO statement which declares a vector is executable in the sense that, when it is encountered, it initializes the pointer to "n+1" words. The initial contents of an AUTO vector or other AUTO variables are always undefined.

Because AUTO variables are allocated on the stack, and because there is no check for stack violation, you should be cautious about declaring large vectors as auto variables. Although the compiler command will let you change the default stack size of 500 words, it may be preferable to either use an external, or else allocate it from free storage.

3.3.4. Labels.

Any unique identifier followed by a colon and preceding a statement is defined as a label. For example:

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

A statement may be preceded by as many labels as appear to be necessary, as in

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

3.4. Transfer of control.

The GOTO statement does the obvious thing. The RETURN statement is used to exit from a function. The NEXT and BREAK statements greatly simplify loop control.

3.4.1. Goto.

goto label;

will cause a function to transfer control to the statement which has the label "label". The compiler actually accepts "goto expression".

If "label" has not already appeared, it is defined as one. It is a fatal error if it is not used as a label in the function body.

It is legal to transfer to any location inside a function body, including into or out of a compound statement. It is almost never a good idea to transfer into a compound statement, because the action is difficult to follow and because it can lead to unpleasant surprises.

Because B is a typeless language, the compiler has no way of knowing whether the label or expression you supply in a GOTO statement really turns out to be a valid label at runtime, so it is perfectly legal, but probably erroneous, to say

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

Never try to pass a label as an argument to a function and use it to transfer to another function. The program will end up in one function, but with a different function's stack pointer, resulting in immediate or eventual disaster, unless you know exactly what you are doing.

3.4.2. Return.

```
return ;

return ( expression ) ;
```

The RETURN statement ends the execution of a function and results in return to the caller. Upon return, all temporary storage in use by the particular invocation of the function disappears.

The first form of the RETURN statement merely returns control. The second form causes a one word value to be returned also.

Note that the construct

```
return();
```

is not permitted by the compiler (it gives a syntax error).

A simple RETURN statement is supplied implicitly at the end of a B function body.

The library function EXIT is also available, should your program wish to terminate execution at a point other than after the last statement of MAIN.

3.4.3. Break.

```
break ;
```

The effect of BREAK is to drop out of the most recent 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.

3.4.4. Next.

```
next ;
```

NEXT is a directive to skip all further statements in the most recent enclosing WHILE, FOR, REPEAT, or DO-WHILE loop, and transfer to the test which determines whether looping should continue.

3.5. Conditional statement.

3.5.1. If.

```
if ( expression ) statement
```

If the result of the expression is non-zero, then the statement is executed. The parentheses around the expression are mandatory.

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

If the result of the expression is non-zero, the first statement executes, otherwise the second statement executes.

In the case of nested IF statements where there are fewer ELSEs than IFs, the compiler associates the ELSE with the closest IF at the same level of nesting.

```
if ( ... ) if ( ... ) s1 else s2
```

resolves to

```
if( ... ) if-statement
```

Think of IFs and ELSEs being placed on a pushdown stack as they appear. An ELSE which you pull off the stack always goes with the next IF pulled off.

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;
  }
```

3.6. Iterative statements.

A REPEAT iterates a statement until a BREAK statement is encountered or a GOTO causes control to pass outside the loop. WHILE iterates a statement as long as an expression is non-zero, testing at the top of the loop. DO-WHILE iterates a statement until an expression is non-zero, testing at the bottom of the loop. A FOR uses three expressions to initialize, test and modify in controlling a loop.

3.6.1. Repeat.

repeat statement

The REPEAT merely executes the statement forever. The statement is almost invariably compound. NEXT and BREAK statements are legal inside a REPEAT.

3.6.2. While.

while (expression) statement

If the result of the evaluation of the expression is non-zero, the statement associated with the WHILE is executed. After execution of the statement, the expression is re-evaluated again and, if the result 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.

BREAK and NEXT statements are legal in a WHILE statement.

3.6.3. Do-while.

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;
  }
```

BREAK and NEXT statements are legal in a DO-WHILE statement.

3.6.4. For.

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, which might initialize a controlling variable, is evaluated. Then, if and while the result of second expression (usually a test) is non-zero, the statement is executed. Before returning to re-evaluate the second expression, the third expression, which might increment a 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.

Any or all of the expressions may be null, and they need not necessarily involve the same controlling variable, if any. Note that the second expression is always treated as a logical expression. 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 )
          q[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;
```

3.7. Switch statement.

The SWITCH 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 special labels are allowed inside the statement to point to where to start processing for a given case, as in

```
switch ( expression )
  {
       case const-expr: statement
       case const-expr :: const-expr: statement
            break ;
       case <rel op> const-expr: statement
            /* rel 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, 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 a statement which alters the control flow is encountered.

Usually, a BREAK is used. It causes control to go to the statement following the SWITCH.

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 1) a single value, 2) a range of values which include the endpoints, or 3) an upper or lower bound. Overlapping bounds draw a fatal diagnostic from the compiler.

By const-expr we mean as usual 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.

Any attempt to SWITCH on floating-point values will not work, since the generated code performs integer comparisons.

The compiler will construct a jump table for the SWITCH statement if the ratio of 1) the maximum case value minus the minimum to 2) the number of case labels is between one and two.

As an example, here is a function which uses a SWITCH to determine if a character is legal for a B identifier.

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

4. Expressions.

Expressions in B are constructed according to rules which 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 tends to force 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 possibly erroneous 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 runtime monitoring of possible arithmetic overflows or faults. Overflow faults are masked out, but a divide error (like dividing by zero) will result in an immediate abort.

Expressions are evaluated according to an *order of binding* which includes both the hierarchy of evaluation, which determines the order of evaluating different types of operators, and *grouping*, which determines the order in which operators of the same type are evaluated. We will discuss the hierarchy from highest (evaluated first) to lowest and mention the grouping rule for each type. The results are summarized in Appendix A and the explain file "explain b binding".

4.1. Primary expressions.

The primary expression is the basic building block used to construct expressions. It is defined recursively as follows:

name
: A legal identifier is a primary expression.

constant
: Any legal constant constitutes a primary expression.

primary[expr]
: A subscripting operation, which is a primary expression followed by an expression in square brackets, is a primary expression.

primary(arglist)
: A function call operation, which consists of a primary expression followed by an open parenthesis, is a primary expression. The open parenthesis must be followed by a possibly empty set of arguments, consisting of comma-separated expressions, followed by a close parenthesis.

(expr)
Any expression enclosed by parentheses which 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 itself of another primary expression, grouping occurs from left to right. For example, look at

```
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, one of which is to be called. In both cases, "x[i]" is evaluated first, placed in a temporary, call it "y", and then the remainder of the expression is evaluated as "y[j]" or "y()", respectively.

4.1.1. Subscripting.

Subscripting is not restricted to use with variables originally declared as vectors. It is a completely general operation which may be applied using any two arbitrary expressions.

To help you understand how subscripting works in B, take a look at the primary expression

```
a[i]
```

One of the variables is supposed to be a pointer, while the other is supposed to be an offset, but it does not matter which! The reason for this is that B gets a pointer to the cell "a[i]" simply by adding "a" and "i" together. If the value of the cell is required, the compiler gets it by using the pointer. Therefore it is perfectly legal to alternatively say:

```
i[a]
```

anywhere you could have said "a[i]".

4.1.2. Function calls.

As you can see above, the general for of a function call is:

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

Most commonly, "primary" is just the name of the function to call, but the generality of expression is there to permit you construct and use vectors or lists which contain functions to be called.

A function call primary may always be assumed to return

a value. It is up to the programmer to make sure that a value is returned when one is wanted or that a value is only wanted when one is returned.

It is also up to the programmer to make sure that a function is called with as many arguments as it needs. It is safe to call a function with more or fewer arguments than are defined for it, assuming the called function is prepared for such contingencies.

Note that it is the parentheses surrounding the argument list which tell the compiler the operation is a function call, so they must always be present. For instance, say you have a function called PROC which requires no arguments. To call it, you say

```
proc()
```

But if you say only

```
proc
```

no function call will take place, because none is implied.

4.2. Rvalues and lvalues.

When we come to the assignment statement, or to operators which perform implicit assignment, it becomes necessary to distinguish between the address of a thing and its contents.

An rvalue is the contents of a word. Any expression in B may be evaluated for an rvalue. For example, the rvalue of a subscripting operation is the word addressed by the sum of pointer and offset.

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

An lvalue is the address of a word. 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 formed by the adding pointer and offset.

It is convenient to think of an lvalue as an expression which is legal to the left of an assignment operator and of an rvalue as an expression which is legal to the right, as long as you remember that both may also appear in other circumstances.

Context determines whether an expression is evaluated for its rvalue or its lvalue. For example, look at 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]"

must be able to, and does, yield an lvalue which is the address of the place to put the sum.

Conversely, it is illegal to say either of

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

because the expressions which are on the left of the assignment operator are not permitted to have an lvalue. If they could have an lvalue, you could then in the first case change the value of the constant, or in the second case make a meaningless assignment.

4.3. Unary operators.

A unary operator acts upon a unary expression to transform 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. Except for the unary indirection operator, the result of applying a unary operator is always an rvalue.

The following unary operators are defined:

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

##rvalue
: Converts single precision floating point to integer.

~rvalue
: One's complement. Converts all zero bits of its operand to ones and all one bits to zeros.

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

#-rvalue
: Results in the floating point negation of the operand word.

!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 but uses it as an lvalue. This is the only case in which a unary operation returns an lvalue. Thus any primary expression prefixed by a "*" may appear on the left hand side of an assignment. "*6 = x" stores the contents of x in location six. "y = *x" stores the contents of the word pointed at by x into y.

&lvalue
: The address operator. Forces the program to generate the lvalue of the expression, then use it as an rvalue. For instance, "&x" is an rvalue which contains the address of x in the lower 18 bits, while "&6" is illegal, because "6" may not have an lvalue.

++lvalue
    Adds one to the rvalue, before using it. Each of the auto increment/decrement operators require an lvalue as its operand. An lvalue is required, because of the implicit action of assignment, but the result is always an rvalue.
  lvalue++
    Adds one to the rvalue, after using it.
--lvalue
    Subtracts one from the rvalue, then uses it.
  lvalue--
    Subtracts one from the rvalue, after using it.
 @primary
    The at-sign operator is used to force the use of Honeywell 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. The following are exactly equivalent everywhere:

a[b] <=> *(a+b) <=> b[a]

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", picking up the word pointed at and then adding "c" to the contents. It is equivalent to "*(*(a + b) + c)". If this were part of a larger expression, the word pointed at would be loaded into a temporary. Then the contents of the addressed matrix element is incremented by one.
&a[i]
    Forms the address of the cell "a[i]" by adding the values of "a" and "i". That is, it is evaluated as "a + i".
y = *(&x)
    The verbose way of saying "y = x".

x = *p++
The word which word pointer "p" points at is copied into "x", then "p" is incremented by one to point at the next word. That is, "p" is used, then incremented.

x = ++*p;
The word pointed at by "p" is incremented by one and then copied into "x".

*++p = x
The contents of "x" are copied into the word pointed at by "p", but only after "p" has been incremented by one to point to the next word. That is, "p" is first incremented, then used.

*6 = 2
Places the value two in location six. "*6" is the same as "0[6]" or "6[0]". This kind of construct is used to access locations in the slave program prefix.

x[a[b] + 1]
Gets the contents of "a[b]" into a temporary and adds one 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.

4.4. Binary operators.

All other operators are binary operators, which means they require both a left and a right operand. Each operand must be an rvalued expression.

With one exception, the order in which the two operands are evaluated is undefined, so don't have the evaluation of one side depend on a side effect generated by the other side (in a function call, for instance). Logical operators are the only exception. Their operands are always taken strictly from left to right.

The code generated for floating point operations is correct, but not blindingly efficient, since the compiler generates a separate load and store for each use of a floating-point operand. However, it is there if you need it. For non-casual use of these operators, it is probably better idea to either call a FORTRAN routine to do the job, or else program in some other language.

4.5. Shift operators.

expr << expr
The left operand is taken as the one word bit pattern to be logically left shifted. The right operand supplies the number of bits to shift. If negative, or greater than 127, the result 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.

4.6. Bitwise and.

expr & expr
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.

4.7. Bitwise exclusive or.

expr ^ expr
Takes the bitwise "exclusive or" of its two 36 bit operands. If bit i is on in one, but not in the other, then bit i of the result is on.

4.8. Bitwise or.

expr | expr
This take the bitwise "or" of its two operands, such that if bit i is on in either of the two operands or both, bit i in the result is on also.

The following is a summary chart of the results of bitwise operations. The table shows the effect of each operation on one bit.

operands		results		
a	b	and	or	exor
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

4.9. Multiplicative operators.

expr / expr
Integer division of the first integer operand by the second. Will result 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.

expr % expr
    Results in the integer remainder of the integer division of the first operand by the second. If the remainder is non-zero, it has the same sign as the left operand.
expr * expr
    Integer multiplication.
expr #/ expr
    Single precision floating point divide.
expr #* expr
    Single precision floating point multiply. All floating point operators assume floating-point operands.

Multiplicative operators group from left to right.

4.10. 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.

4.11. Relational operators.

expr == expr	(equal)
expr != expr	(not equal)
expr < expr	(less than)
expr <= expr	(less than or equal)
expr > expr	(greater than)
expr >= expr	(greater than or equal)

The result is one if the given relation between two integer operands is true, and zero otherwise.

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

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

4.12. Logical and.

expr && expr
The result is an integer one if the result of both expressions is non-zero, and zero otherwise.

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

4.13. Logical or.

expr || expr
The result is an integer one if the result of either expression or both is non-zero, and zero otherwise.

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

4.14. "Query" operator.

expr1 ? expr2 : expr3
The first expression is evaluated. If the result is non-zero, the second expression is evaluated and returned, while the third expression is ignored. If the result of the first expression is zero, the third is evaluated and returned, while the second is ignored.

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 left to right, so that

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

is equivalent to

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

4.15. Assignment operators.

lvalue = expr
Takes the one word result of the evaluation of "expr" and stores it in the word addressed by the lvalue.

lvalue <op>= expr
Is equivalent to the assignment
lvalue = rvalue <op> (expr)
where <op> can be any one of
* / % + - << >> & ^ |
Note that neither floating point nor relational operators are included.

For example,

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

is the same as

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

In all cases, the expression 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);
```

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

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

Note that parentheses are used in this case to alter the order of precedence. These are required in this case because the assignment operators have the lowest precedence, which means that they are evaluated last.

5. Implementation-dependent information.

The information in this chapter is subject to change.

5.1. Linkage conventions.

The B compiler's mechanism of performing a function call is rather different from the standard Honeywell calling sequence. In this chapter, we will describe the calling conventions in detail, so you can attempt to write or understand functions written in GMAP for the B environment.

5.1.1. Function call.

The standard B function call looks like

```
        tsx1    sub,*
        tsx1    sub,*
        zero    s,n
```

where "s" is amount by which the stack pointer should be bumped and "n" is the number of arguments supplied. The compiler generates these numbers for B functions.

When a B function is running, its stack pointer points to a word of return information. Above that point are a fixed area for arguments and a fixed area for auto variables, all addressed relative to the stack pointer. Stack space above the auto variables is used to hold temporaries created during expression evaluation. There is no check for stack violation. When one function calls another, the stack pointer must be moved so the callee does not affect the state of the caller.

Before executing the TSX1 instruction to transfer indirect to the function, the caller must first set up the argument values. The first and second arguments are loaded into the A and Q registers, respectively. Other arguments, if present, must be placed in the stack in such a way that they are available to the caller once the stack gets bumped. This is shown in the next section.

5.1.2. Entry.

The general convention for a subroutine entry looks like this:

```
        symdef  sub
        symref  .10001
sub     tra     *+2         indirect pointer to function body
        zero    end,debug    end of function or debug table
        xed     .10001      (adx7 0,1 - advance sp by "s")
        rem                 (stx1 0,7 - store return address)
        staq    1,7         first two args passed in the AQ
```

The ZERO word is used by the debugger, PMD, and by the profiler, .PROFILE. It contains either a pointer to the last word used in the function in the upper half or a pointer to the debug symbol table in the lower half, but not both. If you are writing your own function, it is quite safe for both of these to be zero.

Index register seven is reserved for the stack pointer. Index register six is reserved for the coroutine package. Address registers four and five are reserved for use by the input/output package. B functions assume a called function may have used any other registers.

The GMAP routines in the runtime package follow the added convention that index registers five through seven may not be used and that index registers three and four must be restored if necessary upon exit.

By convention, the first two arguments are passed in the A and Q register. It is the responsibility of the called function to store them if necessary.

The hardware will only allow a "STAQ" instruction to work correctly if the address of the store is at an even word boundary. To enforce this, the stack pointer is always initially set to an odd address, and must always be incremented by an even amount.

Once the "STAQ" is done, the stack is organized as follows:

```
0,7     -> return address
1,7     -> first argument (initially in A register)
2,7     -> second argument (initially in Q register)
n,7     -> nth argument (placed in stack by caller)
n+1,7   -> start of auto variables and temporaries
```

5.1.3. Exit.

When a B function has done its job, it returns using the sequence:

```
        symref  .10000
        tsx0    .10000  (ldx1 0,7 - restore return address)
        rem             (sbx7 0,1 - restore stack pointer)
        rem             (qls 0 - set indicators for caller)
        rem             (tra 1,1 - return)
```

Prior to this, the function may load a one word value into the Q register, which becomes the value of the function call. The calling function assumes the indicators are set upon return.

A TSX0 instruction is used so that the interactive debugger, if in use, can determine the address at which the function returned, in order to find out the name of the function returning.

5.2. Internal representation of objects.

The material in this section is intended to help you understand how the code generated by B works internally. The value of a function name is an external word containing a transfer instruction with the address of the first word of the function body in the upper 18 bits. Transfer of control to a function body always occurs indirectly through a word, whose top 18 bits contain the address and whose tag field is expected to be zero.

The value of a label is a word containing, in the upper 18 bits, the address of the place to go to in the function body and zeros in the lower 18 bits. The transfer involved in a GOTO occurs indirectly through the label word.

The value of a pointer or address is a word, whose bottom 18 bits are taken as an address.

The value of string constant is a pointer to the text of the string.

6. The B library.

One of the big advantages in using B is the availability of a large library of useful functions which simplify your programming problems and also supply a reasonable interface to the GCOS/TSS environment.

Every B library function you could reasonably expect to use has an explain file under "explain b lib". There is also an index of all documented routines.

Only the routines you need to get started using B will be discussed here and even then not all options may be treated.

Some functions may return a value; this is indicated in this section by showing an assignment to indicate a value is returned. Also, some functions are called with a variable number of arguments. If you want to use an optional argument, you must usually also specify any preceding optional arguments also. Optional arguments are shown enclosed in square brackets. For example, if a function is shown as

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

and you want to use "arg3", then you must also use "arg2". Sometimes, as you will see, the first argument, usually a "unit", may be optional. In this case the called function craftily examines the first argument to see if it is a number valid for "unit"; if it is not, it adjusts its argument references accordingly.

6.1. .BSEI - redirection of i/o.

Before your main program is entered, the run time initialization routine 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
: .BSET will open the file for reading on B unit 0. Input for unit 0 will come from this file, rather than the terminal.

>filename
: The filename is opened for writing. Output to B unit 1 will go to this file, rather than the terminal.

>>filename
: Same as above, except that if the file already exists, output is appended to the file.

A quoted string 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.

.BSET collects the arguments which are not redirect requests and builds a vector of pointers to those strings. MAIN is later called by

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

where "argc" is the number of arguments collected 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 cell of which contains a character giving some indication of the type of argument in the corresponding ARGV string:

type	character
string	' '
string in single quotes	'*'
string in double quotes	'"'
-option	'-'
+option	'+'
possibly signed number	'0'
string with = in it	'='

For example, look at the command line

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

MAIN will be called with ARGC set to four, since ">b.out" is not included. All writes on unit 1 will 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 have .BSET, simply supply your own function definition of ".bset();" which will replace the library version.

Normally, however, you will want .BSET, because it greatly 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 "explain b lib .bset".

6.2. Introduction to input/output.

The largest class of functions in the B library are those concerned with input and output. Sequential input routines will read, and convert to ASCII if necessary, 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). Output is ASCII (media 6) unless special precautions are taken.

The i/o package will create output files if necessary and "grow" them as required up to their maximum size or to the limit of the file space quota for a userid.

6.2.1. 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", to which every i/o call implicitly or explicitly refers.

There are five units whose function is predefined and may not be altered by the user.

-5

This is an input unit whose origin is always the terminal in TSS or file code I* in batch. It may be used to force reading from the terminal or file code I*, even though the standard input may have been redirected to come from a file.

-4

This is an output unit whose destination is always the terminal in TSS or file code P* in batch. It is most often used to avoid possible redirection of i/o by forcing error messages to appear on a hard copy device.

-3

Used for console input in batch only.

-2

Used for console output in batch only.

-1

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

Unit zero is initialized as the standard input unit. In TSS, this is the terminal but input may come from a file if redirection of i/o is used. In batch, reads on unit zero come from file code I*, if it is defined and if input was not redirected. If I* is not present and there is no input redirection, unit zero is placed in the end-of-file condition.

Similarly, unit one is initialized as the standard output unit. In TSS, this is again the terminal and is subject to redirection of i/o. In batch, output to unit one goes to the printer, unless redirected.

Units two through 19 may be assigned by or to you, using the file opening call, usually to permanent or temporary disk files. It is permissible to open units zero or one, but the usual practice is to leave them alone, so they may be redirected.

6.2.2. Unit opening.

Before any i/o may be performed on a unit, it must be initialized by a call to OPEN, which is of the following form:

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

Normally, "unit" is not supplied, and OPEN finds a free unit, which it returns. If you do specify a unit, and that unit is already open, The state of the previous unit is saved on a stack; when the current use of the unit is closed, the previous state is restored 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 (e.g. "fbaggins/s/test.b"). An altname in quotes is used, if present. Permissions are effectively ignored.

"action" is also a pointer to a string containing characters which specify the access permissions required and the type of i/o to be done on the unit.

"ret" is the value returned by OPEN. If non-negative, the open succeeded and "ret" contains the number of the unit just opened. "ret" is negative and no unit is opened if there was a file access error or if there was an OPEN error.

Although not quite all of the various things accepted by OPEN are dealt with in this chapter, they are treated fully in the explain file for OPEN.

Mode actions: OPEN offers three ways to specify the mode of the unit being opened. They are as follows:

l

This forces the requirement that the unit to be opened be a linked (sequential) file.

b

The 'b', for binary, requires that the unit being opened be a random access file.

s

This is used for so-called "string i/o". The "filename" argument is taken as a pointer to the start of a block or words in memory The stream i/o functions will place characters into this block of memory, rather then transmitting them to a file.

If none of these is supplied, "lb" is assumed, indicating that a file is wanted and that it should be accessed according to its mode. It is then up to the program to determine, with the help of the function FILDES, whether the file is random or sequential and then make whatever i/o calls seem appropriate.

Error actions: Normally, OPEN never returns an error status, since the default action is to abort the program with a reasonably understandable error message.

The OPEN function lets you specify options which allow you to handle either file opening errors or 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 a negative status is returned on an OPEN or file access error.

m

Normally, when an error status is returned, no error message is printed. The inclusion of the 'm' option in the action string will cause the appropriate routine to display an error message before returning bad status to the caller. It has no effect if neither the 'e' or 'f' action is used.

For instance, if you said

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

and "/myfile" could not be opened with read permission, OPEN would print an error message, then return bad status to the caller.

In the case of OPEN errors, you will most likely get

back a number with is the negative of the file system error status. For example, OPEN would return -5 for "permissions denied". In addition, OPEN itself is liable to return any of the following special error statuses:

-64 - too few arguments
-65 - no free unit
-66 - open append error

File accessing in TSS is done by calling the .GET subsystem. In batch, the file accessing logic is bound with your program.

File access conventions: There are a number of file access/create conventions for TSS which you should be aware of:

1. Search rule: If you are opening a file and the filename does not contain any slashes or dollar signs (e.g. "b.out"), the file accessor first searches the AFT for a file of that name. If not found, the file accessor searches for a quick-access file of that name under the current userid. If the filename does contain at least one slash or dollar sign, it is assumed to be the name of a permanent file. If the first character of the name is a slash or if the name contains no slashes at all, the file is assumed to be under the current userid; otherwise it is taken as a complete name.

2. Create rule: If the search fails and the request is to write or append, OPEN will attempt to create the file. If the filename contained no slashes or dollar signs, OPEN will try to create it as temporary; otherwise it tries to create a permanent file.

3. AFT rule: If the file was already in the AFT when accessed, it is left there when the unit is closed; otherwise it is ruthlessly removed from the AFT. You may override this by including in the action string either the character 't' (for transient) to force deaccess, or the character 'k' (for keep) to force the file to be kept in the AFT.

6.2.3. Unit closing.

When you are through with a unit, you may want to close it explicitly by calling CLOSE:

```
close( unit );
```

For sequential stream output units, CLOSE flushes the output buffer if necessary, with an end-of-file mark written if appropriate. A unit associated with a disk file has the file deaccessed, if required. CLOSE releases the i/o vector after checking to see if there was a prior use of the unit which had been interrupted and saved. If there was, it is restored and i/o may then proceed on that unit as if there had been no interruption; otherwise, the unit is free for further allocation.

When your MAIN function terminates, or when you call

EXIT directly, all open units are closed automatically. Note that if you hit break, things are set up to call EXIT, unless your program has established its own break handling procedure.

Any attempt to read data from a unit which is not open results in the library routine called returning a value which indicates nothing happened. Similarly, output to a unit which is not open tends to vanish.

6.2.4. Unit switching.

Since some input/output calls may not specify a unit directly, the i/o package maintains a default input unit and a default output unit to which these calls implicitly refer.

Initially, the i/o package is set to read from the standard input (unit zero) and write on the standard output (unit one).

Any successful call to OPEN changes the default input or output unit.

A library function which takes a unit as one of its arguments will will change the reading/writing unit for the duration of the call and restore the previous value before returning to the caller.

User control over unit switching is supplied by

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

If "new.unit" is given, it becomes the current read unit. The number of the old read unit is returned.

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

Works the same way as .read, except that it applies to the default write unit.

6.3. Sequential stream i/o.

This section describes the body of routines oriented towards handling input/output on terminals or standard system format sequential files.

The i/o package reads almost any media and arranges things so that the using program sees only a stream of ASCII characters. For instance, 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 it 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. If writing to SYSOUT in batch, output is media three (BCD printer format).

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

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

```
unit = open( filename, "wl" );
```
will open a unit for writing, requesting write/concurrent permission.

```
unit = open( filename, "al" );
```
will open a unit for writing so that data written by the program 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.

6.3.1. Terminal vs. file.

There are a few special features of and differences between file and terminal i/o of which you should be aware.

A logical record consists of a string of characters followed by a "record terminator", which is one of '*n', '*r', '*v', or '*f'. Of these, '*n' is never present on an input file; but a '*n' never is; instead, the '*n' is automatically supplied by the i/o package to indicate the end of a record. On input from a terminal, you 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 usually only) character is an ASCII file separator (FS - octal 034) character, the same as is used by TSS GFRC. On most ASCII terminals, including the Teleray and Volker-Craig, a FS character is transmitted by typing the 'cntl' and '\' (backslash) keys simultaneusly, followed by a carriage return. On certain others, you get FS by typing 'cntl' and 'l'. Alternatively, an EOT character, usually typed as 'cntl'-'d', may be used instead. You can't signal end of file on a 2741.

Sequential file output is written in GFRC standard system format with 320 word blocks. An end of line ('*n') character written to a sequential disk file signals the end of a record but is not itself placed in the record; all other record terminators do get written out. A new logical record is start following the receipt of any record terminator. If more than 1272 characters are written in a logical record, the i/o package will generate 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 CLOSE is called.

Terminal output occurs whenever a record terminator is received, but is buffered by TSS. 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.

When writing to the batch console, a '*v' is translated to a '*n', but does not cause the current line to be flushed, in order to let you either write a couple of lines or else write a line and then read a line, without having to worry about some other process affecting the console between writes or between read and write.

6.3.2. Stream i/o functions.

char_got = getchar()
Returns the next character from the current read unit. Returns the character '*0' if the current reading unit is closed or at end-of-file. Most of the input routines described here behave as if they make repetitive calls to GETCHAR.

char = ungetc(char);
Sends a character back to the current read unit, so that the next call to GETCHAR will return the last character put back.

char = getc(unit);
Same as GETCHAR, except the reading unit is switched to "unit" for the duration of the call, then restored.

char_put = putchar(char);
Sends the character supplied as its argument to the current writing unit. PUTCHAR also returns its argument word as its value. The argument word may actually contain up to four non-zero characters. PUTCHAR will output as many characters as there are in the word.

char = putc(unit, char);
Same as PUTCHAR, except PUTC switches writing units for the duration of the call.

string = getstring([unit,] string [,maxl]);
GETSTRING gets the next line of input, or the remainder of the current line of input if GETCHAR has already been called. The newline at the end of the line is not returned; instead it is replaced by a '*0' to mark the end of the string. "string" is taken as a pointer to a vector long enough to hold the string. "unit" is used if supplied; otherwise the current read unit is used. If "maxl" is given, only the first "maxl" characters are returned, with a '*0' tacked on to the end. If the unit is closed or at end-of-file, GETSTRING returns zero; otherwise it returns "string". The string is the nullstring if a line contains only a newline. If GETCHAR was not called, GETSTRING has the effect of returning the next line of input from the terminal or the next logical record from a file.

```
string = getline( [unit,] string [, maxlen] );
```
Same as GETSTRING, except the line terminating '*n' is included in the string, just before the string ending '*0'.

```
printf( [unit,] format-string, arg1, arg2, ... );
```
PRINTF is the most frequently used means of doing output in the B library. If "unit" is not supplied, the default writing unit is used. If "unit" is given, PRINTF temporarily switches writing units, but restores the original state upon return. "format" is a string describing how the arguments are to be output. It may contain any combination of literal characters and formats. A format is of 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, a count of six is assumed if not supplied. Trailing blanks are stripped.

c - The corresponding argument is printed as an ASCII character. The count option is not applicable. The argument word may actually contain up to four non-zero ASCII characters, which will be printed.

d - The corresponding argument is taken as an decimal integer which is converted to a string and output.

f - The argument is taken as a floating-point number to be converted and output. If you program does not use at least one floating-point operator, you must include an "extrn .float;" to force the loading of the floating-point output routine.

o - The contents of the argument word are output in octal.

s - The corresponding argument is taken as a pointer to an ASCII string, which is transmitted to the output unit stripped of its trailing '*0'.

z - Same as 'd', except that if a field width is given and the converted number is too small for the field, it is padded on the left with zeros, rather than blanks.

If a character in the format string is not part of a 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 given here; for full details, see the explain file "explain b lib printf".

```
string = putstring( string [,maxl] );
```
Works in much the same way as GETSTRING. The '*0' which marks the end of the string is not output. If you want to write out a logical record and the string to be output does not end with a '*n', you should usually follow a call to PUTSTRING by a "putchar('*n');".

```
number = getnum();
```
GETNUM returns the next possibly signed integer number from the input stream. It calls GETCHAR until it has skipped over all blanks, tabs or newlines. If the character is not a digit or a sign, zero is returned, indicating no number was found. If a sign was found, and the next character is not a digit, zero is returned again. Otherwise it collects numeric characters until a non-digit is found, then converts the numeric string it has collected to binary and returns that number as its value. The external GETN.A has the value 1 if a valid number was found. The external GETN.L contains the last character read.

```
putnum( number );
```
PUTNUM converts the assumed binary integer which is its argument to a character string and directs the string to the current output unit.

```
reread();
```
Arranges things so that the next input starts at the beginning of either the line currently being processed or the line just read. When called immediately upon entry, the next GETSTRING will return the command line the program was invoked by, if you do not want to use the services of .BSET.

```
status = eof( [unit] );
```
Returns a non-zero value if "unit" is at end of file. If "unit" is not given, the current reading unit is used. Once a unit is open, end of file is set only after an attempted read results in the detection of that condition. If "unit" is given, and the unit is an output disk file, a logical end of file is written and a new block begun.

There are a few other routines which must be mentioned, but which will not be described in detail here. READF does formatted input, somewhat like PRINTF in reverse. It also supplies the only convenient means of reading in a floating point number. GETNUM and PUTOCT can read or write octal numbers.

GETREC and PUTREC allow you to obtain/transmit a logical record, including record control word, without any intervening processing by the i/o package. You must understand standard system format before you attempt to use GETREC or PUTREC.

6.4. 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. Any number of sectors may be read or written at one time.

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 character 'b' in the action string indicates that the file is to be accessed as random. The action 'b' stands for "binary" - we would have used 'r' for "random", but it is already taken for "read".

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

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

If you intend to both read and write the file in batch, you should use an action string of "rwb".

Reading is accomplished by

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

"buffer" is a pointer to a vector into which the data will be read, "sector" indicates at what sector in the file the read will start, and "nwds" indicates how many words will be transferred. 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 major (bad) status from the i/o.

Writing is accomplished by

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

The arguments have the same meaning as those for READ. The GCOS i/o system always writes a multiple of 64 words. If the number of words you transmit is not a multiple of 64, the unused fraction will be filled with zeros on writing.

6.5. String operations.

The B compiler recognizes the existence of strings only in that it handles string constants. 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 '*0'.

Functions are available to permit processing characters in a string in a "random" manner or character by character. We will also mention several useful string utilities.

6.5.1. "Random" string processing.

```
ch = char( string, i );
```
Returns the ith character in a string pointed to by "string". The count always starts at zero.

```
ch = lchar( string, i, char );
```
Replaces the ith character in the string pointed at by "string" with the character "char" and returns as its value the character supplied.

Where characters are in BCD format (6 characters per word), you should use the function CHARB instead of CHAR and LCHARB instead of LCHAR. The calling sequences are the same, but remember that they take and return BCD characters, not ASCII.

6.5.2. Sequential string access.

By using one of the following calls, it is possible to "open" a string in such a manner that calls to regular sequential i/o routines place characters in or return characters from a string. This method is faster than using CHAR/LCHAR, because the implementation uses hardware "tallies". The action 's' stands for "string".

```
unit = open( string, "rs" [,pos] );
```
Opens a string so that calls to GETCHAR will return characters in the string. A call to GETSTRING returns all characters up to but not including the next '*n' or else up to the terminating '*0'. 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 even if the unit is a string.

```
unit = open( string, "ws" [,pos] );
```
Opens a string so that calls to PUTCHAR place characters in the string. PRINTF, PUTSTRING, PUTNUM and other functions will also send characters to the string. The function of "pos" is the same as described above.

```
unit = open( string, "as" );
```
Locates the terminating '*0' 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 pointed at by "string" is large enough to hold whatever your program puts into it.

```
print( string, format, arg1, arg2, ... );
    Same as PRINTF, except it directs its output to a
    string, instead of an open output unit.
```

When you call CLOSE on a unit open for output to a string, a terminating '*0' is placed in the string.

6.5.3. String utilities.

```
string = concat( string, s1, s2, ... );
    Concatenates the strings "s1" through "sn" together
    and places them in string "string". The output string
    may be used as input, providing it appears first in
    the list of strings to be concatenated. When called
    with only two arguments, CONCAT efficiently copies
    one string into another. CONCAT returns its first
    argument as its value.

val = nullstring( string );
    Returns a non-zero value if the string contains only
    the end-of-string character '*0' 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 at by "string", not including the terminating
    '*0'.

newpos = getarg( arg, string, pos [,delim] );
    Starting at position "pos" in "string", GETARG
    obtains the next group of characters ending with a
    blank and places it in the string pointed to by
    "arg". It returns the position in the string where
    the scan stopped, so that it can be called repeatedly
    to obtain successive "arguments" from the string.
    If "delim" is supplied, it is taken as a pointer to a
    string containing the delimiters which will cause the
    scan to stop; the string must include a blank if you
    want the scan to stop on a blank. Leading blanks are
    ignored. GETARG is useful for scanning a command
    line.
```

There are a number of other functions which perform string operations, including ANY, which determines if a given character is in a given string; COMPARE, which determines if one string is lexically greater than, less than, or equal to another; NUMARG, which scans off numbers instead of character strings; READF, which can do formatted input from a string; and COPYCH, which moves substrings. All of these have explain files.

6.6. Storage allocation.

Library functions are supplied which allow you to dynamically obtain or release memory in the free storage pool.

```
addr = getvec( n );
```

Obtains from free core a vector of length "n" plus one words and returns a pointer to the vector. Once acquired in this manner, the block may referenced using subscripting, as in

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

```
rlsevec( addr, n );
```

Undoes a GETVEC by releasing the "n" plus one words pointed to by "addr" back to the free memory area.

All memory allocation is done by manipulating a free list. The free list initially includes the so-called "core hole". You can return via RLSEVEC any space which is not on the free list, as long as the address of the space 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 will immediately abort.

GETVEC obtains more storage from the operating system as 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 "OK" abort code if a request for memory is denied.

Finally, we will mention three useful routines, each described fully by an explain file, which use these calls: GETMATRIX will construct and return a pointer to a multidimensional matrix; ALLOCATE can be used to obtain a dynamic array which will automatically disappear when a function returns; and RELMEM will release any free memory back to the operating system, in order to reduce program size.

6.7. Media conversion.

Two functions are supplied to let you transliterate BCD into ASCII and vice versa. A BCD string consists of a vector of words containing the characters packed six to a word, left-adjusted and padded with blanks. There is no equivalent to the '*0' in a BCD string.

```
ptr = ascbcd( output, count, input );
```

Takes "count" characters from the ASCII string "input", transliterates them to BCD and places them in "output". If a '*0' is encountered before "count" is exhausted, blanks are supplied and also used to pad the BCD string to a word boundary. "input" and "output" must be pointers. "output" is returned.

ptr = bcdasc(output, input, count);
Takes "count" BCD characters from "input", transliterates them to ASCII and places them in the ASCII string "output". Any trailing blanks are deleted and the end of string delimiter '*0' is placed at the end of the ASCII string. "input" and "output" must be pointers. "output" is returned.

6.8. Call fortran.

The function CALLF provides the ability to call FORTRAN subroutines, or any routine which uses the GCOS CALL conventions. However, routines so called must not be called recursively and must not attempt to do input/output. The FORTRAN i/o package, and File and Record Control (GFRC), which FORTRAN i/o calls, are completely incompatible with B i/o.

intval = callf(&routine, &arg1, &arg2, ...);
Converts its arguments to 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. That is, 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, ...);
Works 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.

6.9. DRLs and MMEs.

There are two 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 which 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:

```
%b/manif/drls
...
opts[2] = opts[3] = 1;
acc.fil( "gcos3/gcos-hi-3ic", opts, ret );
buf = getvec(600);
ascbcd(ret+2,6,".mbrt3");
drl.drl(restor_, ret<<18 | 1, buf<<18 | 1,
     (tra&0777777000000)|(buf+512));
tra:
 printf( "%24b*n", buf+4+status*4 );
rlsevec(buf,600);
```

This code sequence, which obtains a batch error message by locating it in the batch error message module, uses the value of a label to supply a return address to DRL RESTOR.

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

Functions in exactly the same manner as DRL.DRL, except that it uses externals called MME.A and MME.Q.

7. Using B.

7.1. Compiling/running/debugging.

The B command is the main tool for preparing B programs. If given a source file, it will call the compiler to read the source and generate a set of object decks. It may call the random library editor RANEDIT to place or replace modules in a library. Unless there are fatal compilation errors, it always calls the TSS loader to prepare a load module. Only the most common use of the B command is discussed here. For full details, see the TSS explain file "explain B command".

To compile and load a source file, just say

```
B srcfile
```

where "srcfile" is the name of a sequential file containing B source statements. If there were no fatal errors, the load module 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", it is used instead. This file is "grown" automatically by the TSS loader as required.

You could have forced the B to initiate execution by saying

```
B  -go source-file
```

but, if your program plans to interpret a command line, it is preferable to use the command "go", like

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

which will run the load module in ".h" with the given command line.

Your program will probably not work correctly the first time. Usually, undebugged B programs are aborted by TSS for some reason such as "memory fault", "address out of range", etc. These errors automatically force a memory dump to be written to a temporary file called "abrt".

Once you have the dump, you can inspect it using the post-mortem debugger PMD, which will allow you to see a traceback of the calls in effect at the time of the fault, as well as examine the state of local and external variables.

It may be that your program does not perform correctly, but does terminate normally. You can call the B function

```
abort();
```

at strategic points to force a dump to be taken.

PMD by itself may not be sufficient to put the finger on your program's problems. It is always advisable to use frequent calls to PRINTF or DUMPA to supply debugging information.

For full details on PMD, see the TSS explain file "explain pmd".

There are a few other options in the B command which you may find useful. If you find you are allocating too many AUTO variables, so that your program violates the stack limit and overwrites your own code, you can specify a larger stack by using the "Stack=nnn" option, as in

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

Fully debugged production programs need not carry the debug tables with them when running. Use the "-Nodebug" option to turn off the loading of debug tables.

```
B -nodebug src.b
```

You can also ask for a smaller stack, to save memory. The default size is 500 words.

If you change one routine in a program, you usually have to recompile the whole program. It is sometimes more convenient to store routines in a random library. That way, you need only recompile one routine or one group of routines to make a change. The B command provides an interface with the RANEDIT subroutine library editor. To start with, if you say

```
B src.b ranelib=/lib -clear
```

the routine or routines in src.b will be edited into the library "lib", which will be created if necessary according to the usual B file accessing conventions and initialized (cleared) by RANEDIT. To add new routines or replace old

ones, you simply say

```
B src1.b ranelib=lib
```

Your program can load from a user library by specifying the "Library=" option, as in

```
B src.b library=mylib
```

When the loader is called, the library specified by the "r=" option is searched along with any other libraries given using the "l=" option. Libraries are always searched in the order given on the command line. To delete routines from a library, it is necessary to use the TSS command RANEDIT (see the TSS explain file).

The options to the B command are of the forms

```
keyword=string
-keyword
```

In both cases, the keyword may be abbreviated using the following rule: In the explain file, a keyword is shown with upper and lower case letters. A valid abbreviation must include those letters which are in upper case, along with any other letters in the order in which they appear. For example, valid abbreviations of the "Ranelib=filename" option include

```
rane=filename
rlib=filename
r=filename
```

Various options may, of course, be combined onto one command line and abbreviated, as in the following example:

```
B cmdlib/s/roff h=cmdlib/roff l=b/xlib s=250 -n
```

This command line uses the option "Hstar=filename", which allows you to designate the file into which the generated load module will be placed.

7.2. Compiler/loader interface.

When processing a source program, the compiler generates not one but a set of object decks and places them onto 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 it is called.

The compiler always generates an object deck containing the B stack area, which is either 500 words or the size specified in the "Stack=nnn" option. This object deck also contains the externals defined before the first function definition, if any.

A separate object deck is generated for each function.

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.

7.3. Using tabs for readability.

When you type in a B program, you will probably want to leave indentations in order to make clear the order of nesting of your source statements.

Spaces are the logical thing to use, but they are tedious to type and it is difficult to be consistent. At Waterloo, we usually suggest you use an ASCII tab character as one unit of indentation. This has the advantage that, when you use TLIST to get a listing of the source, the TLIST command automatically expands tabs into the right number of blanks, so your program comes out indented the way you want. Also, you can use the "OTD" directive inside the QED text editor so that it, too, expands tabs when displaying a line.

It is not possible to use a "tab character" which is not an ASCII tab.

7.4. Some pitfalls.

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

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

because the code generated checks to see if the contents of "floatval" is logically non-zero, rather than to see if the contents are equal to a floating-point zero. Since a floating point zero may not in fact be a word containing all zero bits (since the exponent may be non-zero and is part of the word) it is better to try

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

In general, floating point is tricky to use in B, since there can be no type checking. You must constantly watch out for erroneous constructs such as using "-3.0" instead of "#-3.0".

Also, here is a common pitfall in the use of string constants. If you say

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

The cell x is changed to point to the storage occupied by the string and 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 had wanted to initialize the vector with the string, you should have used the library function

CONCAT to copy in the string:

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

or else you could have defined "x" as an initialized external.

Finally, something should be said about the size of a vector and the length of a string.

When you declare a vector of size "n", you know that it will actually occupy "n + 1" words, because the vector is indexed starting at zero. Every library routine to which you must pass the size of a vector observes exactly the same convention.

In practice, if one needs a vector of size "n", then one declares it to be of size "n", and then ignores the zeroth or the nth 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 ) ...
```

Strings also can be indexed into, using library functions, using a zero origin, but at first it might 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 '*0' character, which you do not see. If you include the trailing '*0' in the count, then a string of length "n" actually contains "n + 1" characters.

All library functions which require the length of a string need the number of characters, not including the '*0'. The library function LENGTH returns just that number.

B - escape sequences.

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

1. Escape sequences are used in character constants and strings to obtain characters which for one reason or another are hard to represent directly. They consist of '*-', where '-' is as below:

*0	end of string (ASCII NUL = 000)
*e	end of string (ASCII NUL = 000)
*(	{ - left curly brace
*)	} - right curly brace
*<	[- left square bracket
*>	] - right square bracket
*t	tab
**	*
*'	'
*"	"
*n	newline
*r	carriage return (no line feed)
*f	ASCII formfeed
*b	backspace
*v	vertical tab
*x	rubout (octal 177)
*#nnn	nnn is 1-3 character octal number

2. The following are escapes used outside character and string constants on terminals (such as the 2741) which do not have on their keyboards some of the characters used by B. If you use QED, it is nicer to use the QED escapes for these characters, so that when you shift 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
$- (or	cent-sign) - up-arrow
$a	@ - at-sign
$'	` - grave accent

Appendix B

B - Binding strengths of operators.

Operators are listed from highest to lowest binding strength; there is no order within groups. Operators of equal strength bind left to right or right to left as indicated.

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


Appendix C

B - B compiler error messages.

This is a list of diagnostics known to be generated by the B compiler. There may be others.

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

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.

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 semicolon is missing or the number of open curly braces "{" does not match the number of close curly braces "}", in which case the line number will be the number of the last line in the last file being processed plus one. This may be due to neglecting to end a string constant, character constant or comment. You also get this message if you use a keyword in an inappropriate context, such as an AUTO statement, 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 be nested. After reading a "/*", the compiler skips all text until a "*/" is encountered; if there is a comment inside a comment, then the compiler will attempt to compile the remainder of the outside 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" newline 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, precede the newline with a '*'; the newline will be discarded. When the warning is issued, the newline is kept.

invalid octal constant

An integer beginning with the digit zero, which is thus assumed to be an octal constant, contains a character other than the digits zero through seven.

character constant too long

A character constant may not contain more than four characters, although each character may be a two character escape sequence.

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 right operand of a division or remainder operator was found 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

Will happen if you try to use a string constant in a 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 will happen if you declare an external vector like "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 bound 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 non-printing, 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 will occur when you have manifest constants whose evaluation involves other manifest constants.

This will occur if you have 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 will require 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>

The compiler has detected a CASE constant which is in the range of a range case or relational case, in a SWITCH statement. The numbers are given in decimal. If something conflicts with a relational case, then the bounds generated for the relation are shown. 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 detected an attempt to redefine a symbol which has already been defined to the current function body.

no space for symdef

There are too many external definitions; try dividing them into two groups by either compiling them seperately 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 a '#', which is taken to be a compiler directive, does not contain a recognizable directive. The line is ignored.

TSS loader warning messages:

<w> name undefined
This is a loader message, which indicates 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

Routines in the B library are listed below,
with a brief explanation. For greater detail, type
explain b lib <name>
where <name> is the routine name.

```
.abbrv      - check command abbreviation for .bset
.boff       - define a debugger breakpoint
.bset       - breaks up a command line into useful chunks
.profile    - generate execution profile of a B program
.read       - change the current reading unit
.write      - change the current writing unit
abort       - abort job, producing dump and returning status
abs         - absolute value of an integer
acc.file    - access file
acclib      - access system libraries
addvec      - add more space to a vector obtained from getvec
aft.name    - get aftname/filecode for a unit
allocate    - get storage, which may be automatically released
any         - check if a character appears in a string
apply       - apply callers args to another call
ars         - arithmetic right shift
ascbcd      - convert an ascii string to a bcd vector
attach      - associate file name with file code for DRL TASK
back.d      - pass a backdoor file to sysout from tss
backspace   - backspace an output unit by one character
bcdadd      - add two bcd numbers
bcdasc      - convert characters from bcd to ascii
bcdsub      - subtract two bcd numbers
binbcd      - convert a binary number to bcd
c.read      - make unit a reading unit
c.write     - make unit a writing unit
callf       - call Fortran program from B routine
```

```
callff      - call Fortran floating point function from B
catscaf     -  obsolete - use scaf
char        - extract a character from an ascii string
charb       - extract a bcd character from a bcd string
charp       - create a charp character pointer
chcksm      - compute a checksum.
close       - close currently open file
cmpc        - compare two strings via EIS CMPC instruction
cmplog      - compare two values logically (unsigned)
cmpvec      - compare one B vector to another
column      - find the current output column
compare     - compare two B strings
concat      - concatenate a series of strings
copy        - copy contents of one vector into another
copych      - replace a substring by another substring
date        - return current date in ascii
datejul     - obsolete - use datesi
datesi      - convert date in string to standard integer form
datev       - return date in vector of integers
daymon      - convert date to dd/mmm/yy format
div         - integer divide with uniform direction of truncation
drl.drl     - execute a given TSS drl (system call)
drljsts     - obtain status of batch job
dtoa        - obsolete - use print
dump        - dump vector in multiple formats
dumpa       - dump an array
ebcasc      - convert string to ascii from ebcdic
eof         - test or write end-of-file
equal       - compare two strings for equality
error       - print error message, then exit
exit        - end job and return status
extern      - possibly useful externals in the library
fildes      - get file descriptor
flush       - write out contents of current output buffer
fpinput     - convert string to floating point binary
fpo         - obsolete - use print
fsfile      - space input file forward one file
getarg      - extract (command) arguments from a string
getbin      - read vector of binary data from sequential file
getc        - read a character, temporarily switching units
getchar     - get a character from an input stream
getdate     - put date into the form mm/dd/yy
getline     - read a line, with terminating "*n", into a string
getmatrix   - dynamically allocate a matrix
getmedia    - find out the media code of a file
getnumb     - read a number from the current input unit
getrcp      - return pointer to next logical record
getrec      - return next logical record, with rcw
getstr      - read a line, less its trailing "*n", into a string
gettape     - ask for a tape from gcos
getumc      - get userid of current user
getvec      - dynamically allocate a vector
gnumber     - extract number from string
gotoss      - execute a TSS command, never to return
gtb         - execute a gtb instruction
hist        - overview of the histogram package
```

```
histdestroy - free space used by a histogram
histinit    - allocate and initialize a histogram
histogram   - add a point to a histogram
histprint   - print accumulated histogram
incrun      - tell if a user is in a CRUN and not in $*$TALK
intrequest  - handle breaks from the terminal
intss       - tells whether running in tss or not
ioerrors    - obsolete - use open
juldate     - obsolete - use sidate
lchar       - replace a character in an ascii string
lcharb      - replace a bcd character in a bcd string
length      - return the length of a string
lowercase   - turn alphabetics in a string to lower case
linumb      - return number of current line of a file
lstar       - replace a character using a charp pointer
main        - entry to program from the operating system
max         - maximum value of two integers
min         - minimum value of two integers
mme.mme     - execute a given batch mme (system call)
movelr      - perform EIS MLR instruction
moverl      - perform EIS MRL instruction
nargs       - return number of arguments to a function
nobrks      - count times break key hit
nullstring  - check for null string
numarg      - extract numeric argument from a character string
open        - open string or file for read, write or append
overflow    - test and reset overflow indicator
pasust      - execute a tss drl pasust
peek        - peek at memory
pnmatch     - perform simple pattern match (eg, on pathnames)
print       - write (with format) to string
printf      - formatted output
prompt      - printf only if current reading unit is a terminal
putasc      - direct ascii characters to output stream
putbcd      - direct bcd characters to output stream
putbin      - write vector of binary data to sequential file
putc        - write a character, temporarily switching units
putchar     - send a character to the writing unit
putnumb     - output decimal numbers
putoct      - output octal numbers
putrec      - output, unprocessed, a record to a sequential file
putstr      - output a string using eis
qsort       - obsolete - use shellsort
ran.rd      - obsolete - use read
rem         - remainder of division as per function div
reread      - back up to start of input line
rotate      - rotate a word n bits to the left or right
rand        - generate pseudo-random numbers
rd.ran      - obsolete - use read
read        - disc i/o routines (unit oriented)
readf       - formatted input
relmem      - release all free memory
remov       - remove file from AFT given pathname
reset       - non-local goto
retfil      - remove a file from the aft (used with acc.fil)
rewind      - rewind an open file
```

```
rscr        - read system controller clock
rstpsw      - turn off switch word bits by Exclusive-or
sbar        - find size of allocated memory
scaf        - parse pathname into Filsys format
scan        - extract delimited substring of a string
scm         - perform EIS SCM or SCMR instruction
setmedia    - change media code of output file
setpsw      - set switch word by ORing
shellsort   - a Shell sort
sidate      - standard integer date to string
sleep       - wait for specified interval
smc.hash    - calculate smc "hash" bucket for a given user ID
star        - pick up a character using a charp pointer
strings     - working with strings via i/o calls
strip       - start stripping line numbers from input
swapdescr   - change program descriptors
system      - execute a TSS command
t2741       - check if terminal is a 2741
tabset      - establish settings for tab expansion in i/o package
tally       - create tally to bcd string
tallyb      - create tally to ascii string
tape        - describes tape i/o support for batch b programs
task        - submit a batch job via DRL TASK
tick        - return cpu time for current user
time        - return time of day, or convert a time in pulses
tr9to9      - translate any 9-bit character code into another
trace       - how to invoke the call/return trace
trim        - remove trailing blanks from a string
trtest      - perform an EIS translate and test
ttyn        - determine if I/O is to the terminal
ungetc      - put a character back to a reading unit
uppercase   - convert lower case alphabetics to upper case
vector      - getvec and initialize a vector
wdleng      - return word length in bits
xlate       - perform EIS move with translate
zero        - initialise a B vector to some value
```