USERS' REFERENCE TO B ON MH-TSS

S. C. Johnson
Bell Laboratories
Murray Hill, New Jersey

1.0 Introduction

B is a computer language directly descendant from BCPL [1,2]. A tutorial introduction to B on the H6070 computer, under MH-TSS, is given in [3]. The reader is urged to read the examples in section 9 in parallel with the language description, to get the flavor of actual B applications.

B was designed and implemented by D. M. Ritchie and K. L. Thompson. The B runtime support for MH-TSS was designed and implemented by S. C. Johnson.

2.0 Syntax

The next several sections describe the syntax of B. Throughout, reserved words and required characters are underlined, while names not underlined represent syntactic classes. Thus, the if statement is syntactically described by:

`if ( ` *rvalue* `)` *statement1* `else` *statement2* `;`

2.1 Syntactic Overview

B is like most other programming languages in that it has programs, made up of statements, many of the statements contain expressions, and the expressions are made up of operators, names, and constants, with parentheses used in the usual way to alter the order of evaluation.

B differs from most languages in that it has no types: no type declarations are needed, and no type conversions are performed. A variable takes on a type (integer, function, vector, etc.) from its context in an expression or statement. Another difference is the rich operator set of B, including logical and shift operators. B is recursive, but pays only a small penalty for being so.

2.2 Comments and Character Sets

Comments are delimited as in PL/I by /* and */.

B programs consist of *tokens* which are names, constants, and operators. Normally tokens are separated by blanks, tabs, new-lines, or comments; in addition, the compiler treats the characters `(){}[],;?:` and maximal sequences of the characters `+-*/<>&|!=` as complete tokens.

The character set used in B is ascii.

The characters `A` through `Z`, `a` through `z`, `_`, `.`, and backspace are alphabetic characters and may be used in names. The characters 0 through 9 are digits and may be used in constants or names; however, a name may not begin with a digit.

There are certain members of the character set which are acceptable in string and character constants, but sometimes hard to represent. In these contexts, escape sequences are provided. The following escape sequences are currently defined:

`*0`	null
`*e`	end-of-file
`*(`	{
`*)`	}
`*t`	tab
`**`	*
`*'`	'
`*"`	"
`*n`	new line

All keywords in the language are recognized only in lower case. Keywords are reserved.

3.0 Rvalues and Lvalues

An *rvalue* is a binary bit pattern of a fixed length. On the H6070 it is 36 bits. rvalues may have different meanings: integers, labels, vectors, functions, etc.. The actual kind of thing represented is called the type of the rvalue.

A B expression can be evaluated to yield an rvalue, but its type is undefined until the rvalue is used in some context. It is then assumed to represent an object of the required type. For example, in the following expression

```
a(b/c+d[3])
```

is taken to be of type function, since it is followed by a left parenthesis; b and c are taken to be signed integers; and d is taken to be of type integer vector, since it is followed by a left bracket.

There is no check to insure that there are no type mismatches. Similarly, there are no type conversions.

An *lvalue* is a binary number ("address") representing a storage location; thus, for every lvalue there is an associated rvalue, the contents of the lvalue. A variable in B (as in most other computer languages) has both an lvalue (its address) and an rvalue (its contents). Writing:

```
x = y
```

causes the contents (rvalue) of y to be stored into the location (lvalue) of x. The Rvalue is on the Right side of the assignment, the Lvalue is on the Left.

Some things (such as constants, expressions, and function calls) are not permitted to have lvalues in B; thus:

```
3 = x
a+b = x
f(3) = x
```

are all illegal. There are unary operators, * and &, for converting between rvalues and lvalues. Unary *, the *indirection* operator, treats an rvalue as an lvalue; on the H6070, the rightmost 18 bits of the rvalue are taken as a machine address. Thus

```
*3 = x
```

is legal, and stores the contents of x into memory location 3.

Unary &, the *address* operator, does the inverse operation: if x has an lvalue, &x is the rvalue which has the lvalue of x as the rightmost 18 bits (with the other bits 0). Thus, &x can be thought of as a pointer to x. If x has an lvalue, *(&x) is the same as x; this is just the same as saying that x can be used by indirecting through a pointer to x. The opposite identity, that &(*x) is the same as x, is always formally true but less useful in practice.

Since addresses are stored in the right half of a word, adding 1 to a pointer gives a pointer to the next word in memory. For example, the contents of the word immediately following variable x can be obtained by writing *(&x+1).

4.0 Expression Evaluation

The evaluation of expressions (in particular, the binding of the operators) is done in the same order as the sub-sections of this section. Thus expressions referred to as operands of '+' (section 4.4) are expressions defined in sections 4.1 to 4.3. The binding of operators at the same level (left to right, right to left) is specified in each sub-section.

This section discusses the binding and meaning of expressions; the order of evaluation of expressions is undefined. Thus, writing:

```
f(2) + g(3)
```

one cannot assume that f will be called before g.

4.1 Primary Expressions

1. A name is an lvalue of one of three storage classes (automatic, external and internal).
2. A decimal constant is an rvalue. It consists of a digit between 1 and 9 followed by any number of digits between 0 and 9. The value of the constant should not exceed the maximum value that can be stored in a word.
3. An octal constant is the same as a decimal constant except that it begins with a zero. It is then interpreted in base 8.
4. A character constant is represented by `'` followed by one or more characters (possibly escaped) followed by another `'`. It has an rvalue equal to the value of the characters packed and right adjusted, with zero fill. Obviously, the number of characters in a character constant is a machine dependent quantity; on the H6070, up to four characters are allowed.
5. A string is any number of characters between characters. The characters are packed into adjacent words (lvalues sequential) and terminated with the character `'*e'` (ascii EOT character) . The rvalue of the string is the lvalue of the word containing the first character. See section 8.0 for library functions used to manipulate strings in a machine independent fashion. Escapes are available to allow special characters, such as tab and newline, to be used in string and character constants (See section 2.2).
6. Any expression in () parentheses is a primary expression. Parentheses are used to alter order of binding.
7. A vector reference v[n] is a primary expression v followed by any expression n in `[]` brackets. The two expressions are evaluated to rvalues, added and the result is used as an lvalue. v can be thought of as a pointer to the base of a vector, while n can be thought of as the offset in the vector. Thus, if x is a vector, x points to a block of locations whose names are x[0], x[1], ... x[0] is the same as *x, x[1] is the same as *(x+1), and so on. A string is a special case of a vector, where the contents of the words x[0], x[1], ... contain packed characters.
8. A function call f(a,b, . . .) is a primary expression f followed by any number of expressions in () parentheses separated by commas. a, b, . . . are evaluated (in an unspecified order) to rvalues and assigned to the function's parameters. f is evaluated to an rvalue (assumed to be type function). The function is then called. Each call is recursive at little cost in time or space. Thus f(), f(2,x+y,g(1)), and ff[2](e33) are all function references: in the last example, ff is assumed to be a vector containing functions.

Primary expressions are bound left to right.

4.2 Unary Operators

The unary operators are *, &, -, !, ++, --, and ~.

1. The rvalue (or indirection) prefix unary operator `*` is described in section 3.0.
2. The lvalue (or address) prefix unary operator

    & is also described in section 3.0.

3. The operand of the negation prefix unary operator − is interpreted as an integer. The result is an rvalue with opposite sign.
4. The NOT prefix unary operator ! takes an integer operand. The result is zero if the operand is non-zero. The result is one if the operand is zero. 5. The increment ++ and decrement −− unary operators may be used either in prefix or postfix form. Either form requires an lvalue operand. The contents (rvalue) stored in the lvalue is either incremented or decremented by one. The prefix operators return the newly altered value; the postfix forms return the value before it is incremented or decremented. Thus if x currently contains 5, then ++x and x++ both change x to 6. The value of ++x is 6 while x++ is 5. Similarly, --x and x-- change x to 4. The former has value 4, the latter 5.
6. The ones complement operator ~, when applied to an rvalue, turns all 1 bits to 0 and all 0 bits to 1.

Unary operators are bound right to left. Thus -!x++ is bound -(!(x++)).

4.3 Multiplicative Operators

The multiplicative binary operators *, /, and %, expect rvalue integer operands. The result is also an integer.

1. The operator * denotes multiplication.
2. The operator / denotes division. The result is correct if the first operand is divisible by the second. If both operands are positive, the result is truncated toward zero. Otherwise the rounding is undefined, but never greater than one.
3. The operator % denotes modulo. If both operands are positive, the result is correct. It is undefined otherwise.

The multiplicative operators bind left to right.

4.4 Additive Operators

The binary operators + and − are add and subtract. The additive operators bind left to right.

4.5. Shift Operators

The binary operators << and >> are left and right shift respectively. The left rvalue operand is taken as a bit pattern. The right operand is taken as an integer shift count. The result is the bit pattern shifted by the shift count. Vacated bits are filled with zeros. Thus, 8<<2 gives 32, while 8>>2 gives 2. The result is undefined if the shift count is negative or larger than 36. The shift operators bind left to right.

4.6 Relational Operators

The relational operators < (less than), <= (less than or equal to), > (greater than), and >= (greater than or equal to) take integer rvalue operands. The result is 1 if the operands are in the given relation to one another, and O otherwise.

4.7 Equality Operators

The equality operators == (equal to) and != (not equal to) perform similarly to the relational operators.

4.8 AND Operator

The AND operator & takes bit patterns as operands. The result is the bit pattern that is the bit-wise AND of the operands. The AND operator binds left to right.

4.9 Exclusive OR Operator

The Exclusive OR operator exactly^performs result is the bik-wise exclusive OR of the operands. The Exclusive OR operator binds left to right.

4.10 OR Operator

The OR operator | performs exactly as AND, but the result is the bit-wise inclusive OR of the operands. The OR operator also binds left to right.

4.11 Conditional Expression

Three rvalue expressions separated by `?` and `:` form a conditional expression. The first expression (to the left of the ?) is evaluated. If the result is non-zero, the second expression is evaluated and the third ignored. If the value is zero, the second expression is ignored and the third is evaluated. The result is either the evaluation of the second or third expression. Thus, a<b?a:b returns the value of a if a is strictly less than b, and b otherwise; i.e., it computes min(a,b).

Binding is right to left. Thus a?b:c?d:e is a?b:(c?d:e).

4.12 Assignment Operators

There are 17 assignment operators in B. All require an lvalue on the left and an rvalue on the right. The simple assignment operator evaluates=merely the lvalue. The assignment operators `=*`, `=/`, `=%`, `=+`, `=-`, `=<<`, `=>>`, `=<`, `=<=`, `=>`, `=>=`, `===`, `=!=`, `=&`, `=^`, and `=|` perform a binary operation (See sections 4.3 to 4.10) between the rvalue stored in the assignment's lvalue and the assignment's rvalue. The result is then stored in the lvalue. Thus the expression `x=*10` is identical to `x=x*10`. Note that this is not `x= *10.` The result of an assignment is the rvalue for simple assignment, and the result of the binary operation for the other assignment operators. Assignments bind right to left; thus x=y=0 assigns zero to y, then x, and returns the rvalue zero; similarly, if y contains the rvalue 2, then x = y =+ 3 sets both x and y to 5.

5.0 Statements

Statements define program execution. Each statement is executed in sequence. There are, of course, statements to conditionally or unconditionally alter normal sequencing.

Most statements end with a semicolon `;`.

5.1 Compound Statement

A sequence of statements in `{}` braces is syntactically a single statement. This mechanism is provided so that where a single statement is expected, any number of statements can be placed.

5.2 Conditional Statement

A conditional statement has the general form:

`if (` *rvalue* `)` *statement1* `else` *statement2* `;`

This evaluates *rvalue* and executes *statement1* if the rvalue is non-zero, and *statement2* if the rvalue is zero. Notice the analogy with conditional expressions, section 4.10.

The "else" clause is optional; thus:

`if (` *rvalue* `)` *statement* `;`

executes statement if the rvalue is nonzero, and skips it if the rvalue is zero.

As an example, the program segment:

```
if( a<b ) x=a;
else x=b;
```

sets x to the minimum of a and b.

5.3 While Statement

The while statement has the form:

`while (` *rvalue* `)` *statement1* `;`

The rvalue is evaluated; if zero, statement is skipped and control passes to the following statement in the program. If non-zero, statement is executed. The rvalue is then evaluated again and the process is repeated until the rvalue becomes zero.

The while statement can be used to write tight loops quite tersely. Thus, to call the function f 10 times, we may write:

```
i = 10;
while( i-- ) f();
```

We may also write searches:

```
while (v[++i]) ;
```

sets i to the index of the first zero element in the vector v.

When statement is a compound statement, it can be broken out of by a break statement (see 5.6).

5.4 Switch Statement

The switch statement is the most complicated statement in B. The switch has the form:

switch (*rvalue*) *statement1*

Virtually always, statement1 above is a compound statement. Each statement in statement1 may be preceded by one or more cases as follows:

case *constant* :

During execution, the rvalue is evaluated and compared to each case constant in undefined order. If a case constant is equal to the evaluated rvalue, control is passed to the statement following the case. Thus, the code fragment:

```
switch(x) {
case 'a':
   y = 1 ;
case 'b':
   z = 2;
   }
```

sets z to two if x has the value 'b', sets y to one and z to two if x has the value 'a', and does nothing otherwise.

If the rvalue matches none of the case constants, statement is normally skipped; this can be changed, however, by preceding a statement in statement by

```
default:
```

Control then passes to the following statement when none of the case constants are matched.

A compound statement following a switch block can be broken out of by a break statement (see 5.6).

5.5 Goto Statement

The goto statement is as follows:

goto *rvalue* ;

The rvalue is expected to be of type label. Control is then passed to the corresponding label. Transfers into and out of compound statements are legal, but use of labels as dummy arguments to transfer between functions or function invocations is almost certain to cause disaster.

5.6 Break Statement

The break statement has the syntax:

```
break ;
```

It is used to break out of a compound statement controlled by a while or switch statement; other compound statements are ignored by it. If a break statement is within several compound statements controlled by while or switch statements? the innermost compound statement is the one broken out of. Thus, in the program fragment:

```
while (i--) {
   j = f(i);
   switch (j) {
   case 1:
     x = 5;
     break;
   case 2:
   case 3:
     x = g(j);
     }
labl:
   if( j < 0 ){
      xy = 3;
      break;
      }
   xy = j/2;
   }
more:
    ...
```

the first break statement is equivalent to "goto labl;", and the second is equivalent to "goto more;".

5.7 Return Statement

The return statement is used in a function to return control to the caller of the function. The first form simply returns control.

```
return ;
```

The second form returns an rvalue for the execution of the function.

return (*rvalue*) ;

The caller of the function need not use the returned rvalue.

A return statement is automatically generated before the closing } of a function definition (See section 7.3).

5.8 Rvalue Statement

Any rvalue followed by a semicolon is a statement. The two most common rvalue statements are assignment and function call.

5.9 Null Statement

A semicolon is a null statement causing no execution. It is used mainly to carry a label after the last-executable statement in a compound statement. It sometimes is used to supply a null body to a while statement (See the second example in 5.3).

6.0 Declarations

Declarations in B specify storage class of variables, and also, in some circumstances, specify initialization. There are three storage classes in B. Automatic storage is allocated at each function invocation, and becomes undefined upon return from the function. External storage is allocated before execution of the program, and is available to any and all functions. Internal storage is also allocated before execution, but is available to only one function; labels are the only current use of internal storage. Automatic and internal declarations result in immediate allocation of storage for the variable; an external declaration allocates no storage. Storage is allocated for external variables by means of external definitions (See section 7).

External and automatic declarations should appear before the first use of the variables declared; the declaration remains in force until the end of the function in which it appears. Internal variables may be used anywhere in the function in which they are declared.

6.1 External Declaration

The external declaration has the form:

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

The external declaration specifies that each of the named variables is of the external storage class. Each of the variables must also be externally defined.

If the first use of a name is immediately followed by a left parenthesis '(', the name is typed external by default; thus the library functions need not normally be declared.

6.2 Automatic Declaration

The automatic declaration also constitutes a definition:

`auto` *def1* `,` *def2* ... `;`

where each def has either the form

name

or the form

name `[` *constant* `]`

The automatic declaration defines the variable to be of class automatic, and allocates storage for the variable. When a name is followed by a constant in brackets, the automatic variable is initialized to the base of an automatic vector with size equal to constant+1. The actual subscripts used to reference the vector range from zero to the value of the constant.

6.3 Internal Declaration

The first reference to a variable not declared as external or automatic constitutes an internal declaration. The major use of internal declarations is with labels; at the end of each program, internal names not defined as labels will cause an error message. A label is defined by writing

name `;`

preceding any statement.

7.0 External Definitions

A complete B program consists of a series of external definitions. Execution is started by the hidden sequence

```
main(); exit();
```

Thus, it is expected that one of the external definitions is a function definition of main. (Exit is a predefined library function. See section 8.0.)

7.1 Simple Definition

The simple external definition allocates storage for an external object and optionally initializes it; the forms are:

name `;`

or

name ival `;`

In the first form, the external name is defined and initialized with zero. In the second form, an ival (initial value) is a constant or a name; the external name is defined and initialized with the value of the constant, or the lvalue of the name, respectively.

7.2 Vector Definitions

An external vector definition has one of the following forms:

name [ ] ;
name [*constant*] ;
name [ ] *ival* , *ival* . . . ;
name [*constant*] *ival* , *ival* . . . ;

The name is initialized with the lvalue of the base of an external vector. If the vector size is missing, zero is assumed. In either case, the vector is initialized with the list of ivals (See 7.1). The actual size of the vector is the maximum of constant+1 and the number of initial values. Any vector elements which are not explicitly initialized have undefined values.

7.3 Function Definitions

Function definitions have the following form:

(name (*arguments*) *statement*

The name is initialized to the rvalue of the function. The arguments consist of a list of names separated by commas. Each name is defined as an automatic variable; the statement (usually compound) defines the execution of the function. When the function is invoked, each dummy argument is initialized to the value of the corresponding actual argument in the call; there are no side effects on the actual arguments in the function invocation. This form of argument passing is frequently called "call by value".

For example, a function which returns the minimum of its two arguments can be defined by:

```
min(a,b) {
    if( a<b ) return(a);
    else return(b);
    }
```

8.0 Runtime Support for B in MH-TSS

8.1 Input/Output Routines

By far the largest class of routines in the runtime library for B are the I/O routines. This is largely because of the nonuniform way in which I/O is done in MH-TSS, and also because of a desire to give maximum facility with a minimum cost in core utilization.

The underlying I/O for B is character oriented, with two routines, putchar and getchar, forming the basis for the system. By default, getchar picks up characters from the teletype, and putchar writes characters to the teletype. The I/O facilities also permit the use of I/O units, in a way similar to Fortran. The default attachment of putchar and getchar to the teletype can be easily changed to allow the reading and writing of one or more ascii disc files concurrently. I/O buffer management is handled by the system. A unique feature of the runtime system is the existence of an I/O unit which has the effect of sending command lines to system; i.e., lines written to this unit behave as it they had been written at SYSTEM? level, and cause immediate execution of the specified SYSTEM level commands. A detailed description of the I/O routines follows.

`getchar()`

-- this routine returns the next character from the current input unit, right justified with zero fill. The current input unit is given by the external variable rd.unit, whose default value is O. Unit O is always defined as the teletype input unit. If a unit u has never been defined, or is in end-of-file status, getchar will return the ascii EOT character, denoted

`putchar(c)`

-- this routine puts the four ascii characters in c onto the current output unit. The null character (ascii code 000) and the EOT character (ascii code 004) are deleted whenever encountered, and are not copied to the output unit. Thus character constants may be output with expected results, since the characters which are not explicitly present are filled with nulls; putchar('ab') would result in the output of only two characters, 'a' and 'b', to the output unit.

The user must be careful to indicate explicitly the newline character at the end of each line of output; in character constants and strings this has the escape sequence '*n'. If this is not done, all of the output comes in one long line or record, resulting in probable chaos!

The current output unit is given by the external variable wr.unit, whose default value is 1. Unit 1 is always defined as writing the teletype. Another predefine unit is unit -1, which is always defined definition as the system unit; a line written onto unit -1 acts as if it was typed at "SYSTEM?" level in MH-TSS . This gives users a simple and flexible way of using many MH-TSS features without requiring a lot of resident code, or fancy bookkeeping.

In general, a user can switch units by simply assigning an integer from -1 through 10 to either rd.unit or wr.unit. The smart user will make a copy of the current unit setting and restore it after his function is completed; this increases the generality of his program.

putchar(c) returns c as its value; thus the statement:

```
while( putchar(getchar()) != '*n') ;
```

copies a line from the input unit to the output unit.

We have discussed the default settings for units 0, 1, and -1. In order to read and write on ascii disc files, it is necessary to open (i.e., access) the file, and associate the file With a unit number: Thus, we have the two functions "openr" and "openw".

`openr(u,s)`

-- the string s is taken as a cat/file description, and the file is rewound and opened as a sequential ascii disc file with read permission. An empty string s is taken to mean the teletype. The file will now be known as unit number u; the external variable rd.unit is set to u, so that the next call to getchar, getstrt etc. will cause input from this file. If u is not between 2 and 10, openr does nothing. If unit u is open when this function is called, it is closed before the access (See close, below). For error handling, see the end of this section.

`openw(u,s)`

-- the string s is taken as a cat/file description and the file is rewound and opened as a sequential ascii disc file with read and write permission. The null string s is taken to mean the teletype. The file will now be known as unit number u; the external variable wr.unit is set to u, so that the next call to putchar, putstr, printf, etc. will cause output onto this file. If u is not between 2 and 10, openw does nothing. If unit u is open when this function is called, it is closed before the access (See close, below). For error handling, see the end of this section.

The next three I/O functions are string, rather than character, oriented. They are implemented by repeated calls to getchar and putchar to do the I/O, followed by packing or unpacking the strings.

`getstr(a)`

-- This function reads the next input line into the vector a. The newline character at the end of the input line is replaced by the string delimiter '*e'. The user is responsible for seeing that the vector a is big enough to hold the input string in all cases; this must include leaving room for the string delimiter. This function returns a as its value.

`putstr(a)`

-- This function copies the string a to the current output unit by repeated calls to putchar. The characters are output successively up to, but not including, the string delimiter '*e'. Thus,

```
putstr(getstr(a));
putchar('*n' );
```

copies a complete line from the input to the output unit.

`system(a)`

-- This function copies the string a to the system output unit, unit -1. This has the same effect as if the string a had been typed at "SYSTEM?" level. A newline is not needed at the end of the string a, since the system function adds one.

The next three I/O functions do important nonstandard operations.

`close(u)`

-- the unit number u is endfiled, and, if u was an ascii disc file, the space for its buffers is reclaimed. Files assigned to the teletype have the current line (if nonempty) written out onto the teletype with no following newline; thus this function can be used to force a prompting question before reading a response on the same line.

Upon normal termination, each output unit is closed. In some cases of abnormal termination, control is lost and no wrapup is possible; in this case, the units may not be closed.

`flush()`

-- the current output file is closed. This is the same as close(wr.unit).

`reread()`

-- this function causes the current line to be reread. It is currently undefined for ascii input files; on the teletype, it means that the next character to be read will be the first character of the current line. There is an important special meaning of reread: if reread is called before the first call to getchar, the first line read will be the "command"; i.e., the line which was typed to initiate execution.

The highest level output function is printf; it is also one of the most useful.

`printf(fmt,a1,a2,a3,...,a10)`

-- This function writes the arguments a1, ..., a10 on the current output unit under the control of the string fmt. From 0 to 10 of the arguments a may be present. The format conversion is controlled by two-letter sequences of the form '%x' inside the string fmt, where x stands for one of the following:

c -- character data (in ascii)
d -- a decimal number
o -- an octal number
s -- a character string

The characters in the string fmt which do not appear in one of these two-character sequences are copied without change to the output unit.

Thus, the call:

```
printf("%d + %o is %s or %c*n", 1,-1,"zero",'0');
```

causes the output line:

```
1 + 777777777777 is zero or 0
```

As another example, a permanent file whose name is in the string s can be created with size n blocks and general read permission by using printf together with the system output unit and the filsys subsystem:

```
wr.unit = -1;
printf("filsys cf %s,b/%d/,r*n", s, n);
```

There are several additional points to be noted about the I/O system.

The first is that if a file which is being written on is too small, it is automatically grown.

The second point involves the treatment of cat/file descriptions. There are two types of cat/file descriptions: complete descriptions and quick access descriptions: A complete description is a string containing at least one "/" character; a quick access description contains no "/" character.

A complete description is viewed as accessing a permanent file in the file system. Thus, if there is a file of the same name already accessed (in the AFT) the previous file is deaccessed and the access repeated. If the file being accessed does not exist, the access fails.

A quick access description is viewed as accessing a file which may or may not be permanent and may or may not be already accessed. Thus, if a file of the given name is already accessed, it is used. If not, the user's catalog is searched for a file of that name. If the file is not found, a temporary file of that name is created, provided that the file is being opened for output.

Currently, cat/file descriptions may not contain permissions, passwords, subcatalogs, or alternate names.

Finally, there are two modes of error handling in the I/O system. Under the default mode, failure to access a file will cause an error message and termination of the job. Under the anything goes mode, access failures are not directly reported to the user, and execution continues. In this case, the functions openr and openw return 0 if the access was successful, and nonzero otherwise. In all cases, failure to grow an output file causes termination of execution.

An attempt to read a character from a nonexistent file causes character '*e' to be returned. Writing on a nonexistent file causes output to disappear without a trace. The error handling mode may be changed by calling the routine ioerrors:

```
ioerrors( n );
```

will set the mode to the default if n is O, and to "anything goes if n is 1.

8.2 Useful Routines

This section describes a number of useful functions for simple character and string manipulation, and storage allocation.

`char(s,n)`

-- returns the value of the nth ascii character, in the string s, right justified with zero fill. Characters are numbered from left to right, starting at zero. Thus char("abc",1) returns 'b'.

`lchar(s,n,c)`

-- This function replaces the nth ascii character of string s by the rightmost nine bits of word c. c is returned as a value. Thus, if s has the value "abc", lchar(s,1,'x') returns the value 'x', and sets s to have the value "axc".

`charb(s,n)`
`lcharb(x,n,c)`

-- These functions are the same as char and lchar, except that they operate with six bit characters.

`ascbcd(v,n,s)`

-- This function converts the first n ascii characters of the string s to bed, and stores them packed into the vector v. If the string s has fewer than n characters, trailing blanks are supplied to fill out v to n characters.

`bcdasc(s,v,n)`

-- converts the first n bcd characters of the vector v into lower case ascii, and puts them into the string s, followed by a '*e' character. Trailing blanks are deleted. The user is responsible for seeing that the vector s has enough room to store the output string, which may be as long as n+1 ascii characters, including the string delimiter. This function returns s as a function value.

`concat( a, b1, b2, . . .. b10)`

-- This function concatenates the strings bl through b10 together (from O to 10 such arguments can be present) and copies the resulting string into a. The user is responsible for seeing that the vector a is large enough to hold the resulting string. a is returned as a function value.

`getarg(a,b,n)`

-- A rudimentary scanning function is provided by this routine. getarg puts into string a the first argument at or after character n in string b. It returns a new value of n which contains the index of the first character after the argument returned. An argument is defined as a string of characters separated by blanks or tabs.

For example, the code:

```
n = getarg(a1,b,O);
n = getarg(a2,b,n);
getarg(a3,b,n);
```

would set a1, a2, and a3 to the first three arguments of b. Thus, if b were the string:

```
"well done"
```

a1 would become "well", a2 would become "done", and a3 would contain the null string. b would be unchanged.

8.2 Other Functions

`getvec(n)`

-- this function returns a vector of size n (i.e. n+1 words) from the core hole, The memory size used by the job will grow if necessary to get the space.

`rlsevec(v,n)`

-- this function releases the n+1 words of the static vector v back to the system. Calling this routine with an automatic vector v as argument results in immediate or eventual disaster!

`nargs()`

-- returns the number of arguments with which the currently executing function was called.

`exit()`

-- terminates the run gracefully, closing all open output files. A return from the main procedure simulates a call to exit.

9.0 Examples

The examples appear exactly as given to B.

```
/* The following complete B program, if compiled and put on your
  file "hstar", will act as an ascii file copy routine; the command
  at "SYSTEM?" level:

   /hstar file1 file2

  will copy file1 to file2. */

main () {
   auto j,s[20],t[20];
   reread(); /* get command line */
   getstr(s); /* put into s */
   j = getarg(t,s,0); /* skip H* name */
   j = getarg(t,s,j); /* filel */
   openr( 5,t );
   getarg(t,s,j); /* file2 */
   openw( 6,t );
   while( putchar( getchar() ) != '*e' ) ; /* copy contents */
   }
```

```
/* This function is called with a string s of the form nnn, nnn,
   nnn, . . .  , where the nnn are integers.  The values are placed
   in successive locations in a vector v.  The number of integers
   converted is returned as a function value.  This program
   provides a simple illustration of the switch and case state-
   ments. */

convert(s,v) {

   auto m,i,j,c,sign;

   i = O; /* vector index '/
   j =-1; /* character index */

init: /* initialize to convert an integer */
   m = 0; /* the integer value '/
   sign = 0; /* sign = 1 if the integer is negative */

loop: /* convert an integer */

   switch (C = char(s,++j)){

   case '-':
      if(sign) goto syntax;
     s = 1;

   case ' ':
      goto loop;

   case  '*e':
   case ',':  /* delimiter . . . store converted value */

      v[i++] = sign?(-m):m;
      if( c == '*e' ) return(i);
      goto init;
      }

/* none of the above cases . . . if a digit, add to m */

   if ( '0' <= c & c <= '9' ){
      m = 10*m + c- '0';
      goto loop;
      }

/* syntax error . . . print message and return -1 */

syntax:
   printf("bad syntax*n");
   return(-1 );
   }
```

9.3

```
/* This function replaces each upper case character in the input
   string s by its lower case equivalent.   It uses the fact that
   the ascii alphabetic characters are contiguous. */

lower(s) {

   auto c,i;
   i = -1 ;
   while( (c=char(s,++i)) != '*e'  )
   if( c >= 'A' & c <= 'Z' ) lchar(s~i~c-'A'+'a');
   }
```

9.4

```
/* This function prints out an unsympathetic error message on the
terminal for each integer value of errno from 0 to 5 */

snide(errno) {
   extrn wr.unit, mess;
   auto u; /* temporary storage for the unit number */

   u = wr.unit ; wr.unit = 1;

   printf("error number %d, %s*n'*,errno,mess[errno]");

   wr.unit = u;
   }

mess [5] "too bad", "tough luck", "sorry, Charlie", "that's the breaks",
       "what a shame", "some days you can't win";
```

10.0 Compiling and executing programs in MH-TSS

B programs, can be run in MH-TSS through the use of the command ./bj (for "B job"). This command, written by B. W. Kernighan and s. C. Johnson, provides an interface between the B compiler and the operating system. The syntax of the command is:

```
./bj [(options)] [source [hstar] ] [spec] [spec] [spec] . . .
```

In the above command, brackets [] indicate optional arguments. Source is the cat/file description for the B source file, hstar is the cat/file description for the H* output file, and options stands for the jrun options. All cat/file descriptions in this command which do not contain a user master catalog name are taken to be part of the current user master catalog. The arguments denoted by spec above give other specifications for the running of the job; there are the following types of specification:

```
h=cat/file
r=cat/file
l=cat/file
s=nnn
```

The construction "h=cat/file" is used to specify an H* file when there is no source file being compiled; i.e., the H* file is being entirely constructed from libraries. The construction "r=cat/file" specifies that the random library cat/file should have the output of the B compilation placed on it by means of she RANEDIT automatic edit facility. The construction "l=cat/file" specifies that the library cat/file should be searched by the loading process as part of constructing the H* file. The "r=" construction also specifies that the associated cat/file should be searched; i.e., it causes an implied "l=" construction. The libraries specified by "l=" and "r=" constructions are searched in the order in which they appear on the command line. The specification "s=nnn" is used to specify an optional size for the stack in B; the default value is 500 words. Two words of stack are used for each function invocation, together with one word for each argument, and space for automatic variables and a few temporaries. In this construction, nnn is an integer.

The user should ideally possess an IDENT file, exactly as required by the QED utility programs such as ./list, etc. If no IDENT file is found, a temporary IDENT file is created by asking the user for his ident card image and his userid.

The "./bj" command works as follows: the first two passes, ./b1 and ./b2, of the B compiler are run in MH-TSS. The result of the second pass of the compiler is a GMAP program, which must be sent to the batch world, compiled, and loaded with the B I/O library (on file ./blib) to create an executable H* file. The ./bj command calls ./b1 and ./b2, and if no errors are detected, the GMAP deck is submitted to the batch world; currently, this is done using "jrun". The snumb of the job is printed out on the terminal. When the job is completed, it can be run by typing "/hstar" at system level, where hstar is the name of the H* file in the ./bj command line.

Several examples follow:

```
./bj name hstar
```

This is the simplest command; the B program on file "name" is compiled, and the resulting H* file put on file hstar.

```
./bj (w) name hstar
```

This is the same as the first example, except that the GMAP compilation is submitted with "wait" disposition by jrun.

```
./bj name hstar r=abc/blib
```

This is the same as the first example, except that the object deck is saved on the library abc/blib. (Note that a RANEDIT job will have to be run to clear this library before any decks are put on it).

```
./bj name hstar l=lib1 r=lib2 l=lib3
```

This is the same as the first example except that the object deck is placed on library lib2m and the libraries lib1, lib2, and lib3 are searched, in that order, to create the file hstar.

11.0 Advanced Topics

This section describes the execution environment of B programs, and tells how to write B-callable GMAP subroutines.

B programs are recursive; the stack pointer is kept in index register 7. Thus, during the execution of any function, the return address is kept in address 0,7 , and the values of the first, second, etc. arguments are kept in addresses 1,7 , 2,7 , 3,7 , etc. If there are n arguments, the nth has address n,7; the locations from n+l ,7 onward are used for automatic variables and temporaries.

When one B program calls another, index register 7 must be moved forward to an unused portion of the stack; by convention, this is done by the called program. A call to a routine "sub" causes the following GMAP code to be generated:

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

In this call, s is an integer telling how much index register 7 should be advanced to avoid conflicting with the calling program's use of the stack. n contains the number of arguments; this is never used by the calling sequence, but is available to the user by calling the library procedure nargs. Notice that the tsx1 is indirect through the word sub, instead of directly to it. Functions and labels in B, because they point to executable code, contain their addresses in the left half of the 36 bit word; this means they can be indirected through. In contrast, pointers which point to data (such as vectors) have their addresses in the right half of the word.

Upon entry to the routine sub, the stack frame is bumped and the return address is stored on the stack; the first three instructions are typically:

```
sub     zero    *+1     pointer to code
        adx7    0,1     advance stack pointer
        stx1    0,7     store return address on stack
```

The return from this function is the reverse process:

```
        ldx1    0,7     restore return address
        sbx7    0,1     restore stack pointer
        tra     1,1     return
```

Because the return sequence is identical for all functions, and a function may have several returns, this sequence of code is stored in the runtime library with the external name .10000 . Thus, the above lines are replaced by:

```
        symref  .10000
        tra     .10000
```

Arguments are passed by value, and never copied back to the calling routine. If there is one argument, it is passed in the A register; if two, they are passed in the A and Q registers. If there are more than two, the third and later arguments are computed and stored on the stack so that they become addresses 3,7 , 4,7 , and so on after the stack frame is advanced. B programs treat their arguments exactly as other automatic variables; it is necessary, thus, to move the first two arguments from the registers onto the stack. This is done by a

```
        sta     1,7
```

or

```
        staq    1,7
```

immediately following the entry sequence given above. The instruction staq demands that its effective address be even. Thus , we have two restrictions designed to assure that index 7 is always odd:

a. Index 7 is initially odd.

b. In calls, the stack is always bumped by an even amount.

Functions return a value in the Q register. The indicators are not assumed set on return, since the .10000 sequence destroys them.

From the above discussion, the reader can write general B-callable GMAP subroutines of arbitrary complexity. If the subroutine has at most two arguments, however, the stack need not be changed, since the A and Q registers contain the first two arguments. The general rules for this situation are:

a. The A and Q registers contain the first two arguments.

b. Index registers 2 through 7 must be restored to their previous values upon return.

c. Return is to location 1,1

As an example, the following rather useless subroutine returns the sum of its two arguments:

```
$       gmap
        symdef  add2
add2    zero    *+1
        sta     temp    store first argument
        adq     temp    add to second
        tra     1,1     return with result in Q
temp    zero
        end
```

Simple GMAP functions of this type are particularly useful as interfaces to MH-TSS system calls ("derails").

12.0 Nasties

This section describes the uglier features of B on MH-TSS.

1. The compiler makes sense of certain expressions with operators in ambiguous cases (e.g. a+++b) but not others even in unambiguous cases (e.g. a+++++b) .

2. The GMAP assembler flags instructions of the form:

   ```
   staq1,7
   ```

   with "A" flags; in fact, these instructions always work properly, and are no cause for alarm.

3. External names in GECOS may only be six characters in length, and are in bcd so that case distinctions are lost. Thus, the B names aaaaaa, AAAAAA, and aaaaaaxx all refer to the same external function.

4. The use of the operator & in conditional statements has been optimized to allow for conditional transfers and much more efficient object code; unfortunately, there is a conflict between this use and the use of this operator in its more usual sense as a bitwise logical operator. Thus, the statement :

   ```
   if( a&077 )goto label;
   ```

   is ambiguous; on the one hand, it could mean "if the last two octal digits of a are not 00, go to label" (& used as a bitwise operator), while on the other hand it could mean if both a and 077 are nonzero, go to label" (& as a logical operator). This ambiguity is broken by always assuming the logical operator in a case where a truth value is expected (if, while, and conditional expression). To force the bitwise interpretation in the above example, one must write

   ```
   if( (a&077) != O )goto label;
   ```

   which is in fact optimized well by the code generator.

5. Missing subroutines, perhaps caused by misspelling, cause the GECOS loader to insert a MME GEBORT at the references. This, when executed in MH-TSS, gives the message "Illegal opcode". Check the load map to make sure.

13.0 Diagnostics

Diagnostics consist of two letters, an optional name, and a source line number. Due to the free format of the source, the source line number might be slightly too high. The following is a list of the diagnostics.

[Table omitted; same as in Kernighan tutorial -- DMR]

14.0 Future Plans

There are a number of desirable features missing from B in MH-TSS. These include the ability to use full cat/file descriptions when accessing a file, the ability to read and write BCD, random, and binary files, and the ability to run in batch. A number of these features could be added at once if an I/O interface to GFRC were written; this seems like the most likely future extension to the I/O library. It is almost certain that getchar and putchar will remain functionally unchanged in this case. Users are urged to gather their file accessing and error handling routines in a small number of clearly marked spots, both as a matter of good programming practice and to facilitate any future changes that might be needed.

B works tolerably well on the H6070, which is a word addressable machine; when using a byte addressable machine such as the IBM 360/370 models or the PDP-11, B seems less attractive. A successor language, C, is being developed which allows most of the advantages of B on byte addressable machines, as well as a structure capability. While the case for C on the H6070 is not as strong as it is on byte addressable machines, the structure and character manipulation capabilities make it likely that C will eventually appear on the H6070.

A final area where progress is both possible and desirable is in the area of debugging aids. It would be fairly easy to add a symbol table to the compiler output, provided that programs were written to access these tables.

References

1. Richards, M. "The BCPL Reference Manual." Multics repository M0099.
2. Canaday, R. H. and Ritchie, D. M. "Bell Laboratories BCPL." (Bell Laboratories Memorandum)
3. Kernighan, B. W. "A Tutorial Introduction to the Language B." This document, first part.