A TUTORIAL INTRODUCTION TO THE LANGUAGE B

B. W. Kernighan
Bell Laboratories
Murray Hill, New Jersey

1. Introduction

B is a new computer language designed and implemented at Murray Hill. It runs and is actively supported and documented on the H6070 TSS system at Murray Hill.

B is particularly suited for non-numeric computations, typified by system programming. These usually involve many complex logical decisions, computations on integers and fields of words, especially characters and bit strings, and no floating point. B programs for such operations are substantially easier to write and understand than GMAP programs. The generated code is quite good. Implementation of simple TSS subsystems is an especially good use for B.

B is reminiscent of BCPL [2] , for those who can remember. The original design and implementation are the work of K. L. Thompson and D. M. Ritchie; their original 6070 version has been substantially improved by S. C. Johnson, who also wrote the runtime library.

This memo is a tutorial to make learning B as painless as possible. Most of the features of the language are mentioned in passing, but only the most important are stressed. Users who would like the full story should consult A User's Reference to B on MH-TSS, by S. C. Johnson [1], which should be read for details anyway.

We will assume that the user is familiar with the mysteries of TSS, such as creating files, text editing, and the like, and has programmed in some language before.

Throughout, the symbolism (->n) implies that a topic will be expanded upon in section n of this manual. Appendix E is an index to the contents.

2. General Layout of Programs

```
main( ) {
-- statements --
}

newfunc(arg1, arg2) {
-- statements --
}

fun3(arg) {
-- more statements --
}
```

All B programs consist of one or more "functions", which are similar to the functions and subroutines of a Fortran program, or the procedures of PL/I. `main` is such a function, and in fact all B programs must have a `main`. Execution of the program begins at the first statement of `main`, and usually ends at the last. `main` will usually invoke other functions to perform its job, some coming from the same program, and others from libraries. As in Fortran, one method of communicating data between functions is by arguments. The parentheses following the function name surround the argument list; here `main` is a function of no arguments, indicated by (). The { } enclose the statements of the function.

Functions are totally independent as far as the compiler is concerned, and `main` need not be the first, although execution starts with it. This program has two other functions: `newfunc` has two arguments, and `fun3` has one. Each function consists of one or more statements which express operations on variables and transfer of control. Functions may be used recursively at little cost in time or space (->30).

Most statements contain expressions, which in turn are made up of operators, names, and constants, with parentheses to alter the order of evaluation. B has a particularly rich set of operators and expressions, and relatively few statement types.

The format of B programs is quite free; we will strive for a uniform style as a good programming practice. Statements can be broken into more than one line at any reasonable point, i.e., between names, operators, and constants. Conversely, more than one statement can occur on one line. Statements are separated by semicolons. A group of statements may be grouped together and treated as a single statement by enclosing them in {}; in this case, no semicolon is used after the '}' . This convention is used to lump together the statements of a function.

3. Compiling and Running B Programs

```
(a) SYSTEM? filsys cf bsource,b/1,100/
(b) SYSTEM? filsys cf bhs,b/6,6/,m/r/
(c) SYSTEM? [create source program with any text editor]
(d) SYSTEM? ./bj (w) bsource bhs
(e) SYSTEM? /bhs
```

(a) and (b) need only be done once; they create file space for the source program and the loaded program. (d) compiles the source program from `bsource`, and after many machinations, creates a program in `bhs`. (e) runs that program. ./bj is more fully described in [1].

You may from time to time get error messages from the B compilers These look like either of

```
ee n
ee name n
```

where ee is a two-character error message, and `n` is the approximate source line number of the error. (->Appendix A)

4. A Working B Program; Variables

```
main( ) {
  auto a, b, c, sum;

  a = 1; b = 2; c = 3;
  sum = a+b+c;
  putnumb(sum);
}
```

This simple program adds three numbers, and prints the sum. It illustrates the use of variables and constants, declarations, a bit of arithmetic, and a function call. Let us discuss them in reverse order.

`putnumb` is a library function of one argument, which will print a number on the terminal (unless some other destination is specified (->28)). The code for a version of `putnumb` can be found in (->30). Notice that a function is invoked by naming it, followed by the argument(s) in parentheses. There is no CALL statement as in Fortran.

The arithmetic and the assignment statements are much the same as in Fortran, except for the semicolons. Notice that we can put several on a line if we want. Conversely, we can split a statement among lines if it seems desirable - the split may be between any of the operators or variables, but not in the middle of a variable name.

One major difference between B and Fortran is that B is a `typeless` language: there is no analog in B of the Fortran IJKLMN convention. Thus a, b, c, and sum are all 36-bit quantities, and arithmetic operations are integer. This is discussed at length in the next section (->5).

Variable names have one to eight ascii characters, chosen from A-Z, a-z, ., _, 0-9, and start with a non-digit. Stylistically, it's much better to use only a single case (upper or lower) and give functions and external variables (->7) names that are unique in the first six characters. (Function and external variable names are used by batch GMAP, which is limited to six character single-case identifiers.)

The statement "auto ..." is a declaration, that is, it defines the variables to be used within the function. auto in particular is used to declare local variables, variables which exist only within their own function (`main` in this case). Variables may be distributed among auto declarations arbitrarily, but all declarations must precede executable statements. All variables must be declared, as one of auto, extrn (->7), or implicitly as function arguments (->8).

`auto` variables are different from Fortran local variables in one important respect - they appear when the function is entered, and disappear when it is left. They cannot be initialized at compile time, and they have no memory from one call to the next.

5. Arithmetic; Octal Numbers

All arithmetic in B is integer, unless special functions are written. There is no equivalent of the Fortran IJKLMN convention, no floating point, no data types, no type conversions, and no type checking. Users of double-precision complex will have to fend for themselves. (But see ->32 for how to call Fortran subroutines.)

The arithmetic operators are the usual '+', '-', '*', and '/' (integer division). In addition we have the remainder operator

```
x = a%b
```

sets x to the remainder after a is divided by b (both of which should be positive).

Since B is often used for system programming and bit-manipulation, octal numbers are an important part of the language. The syntax of B says that any number that begins with 0 is an octal number (and hence can't have any 8's or 9's in it). Thus 0777 is an octal constant, with decimal value 511.

6. Characters; putchar; Newline

```
main( ) {
  auto a;
  a= 'hi!';
  putchar(a);
  putchar('*n' );
}
```

This program prints "hi!" on the terminal it illustrates the use of character constants and variables. A "character" is one to four ascii characters, enclosed in single quotes. The characters are stored in a single machine word, right-justified and zero-filled. We used a variable to hold "hi!", but could have used a constant directly as well.

The sequence "*n" is B Jargon for "newline character", which, when printed, skips the terminal to the beginning of t:e next line. No output occurs until putchar encounters a "*n" or the program terminates. Omission of the "*n", a common error, causes all output to appear on one line, which is usually not the desired result. There are several other escapes like "*n" for representing hard-to-get characters (->Appendix B). Notice that "*n" is a single character: putchar('hi!*n') prints four characters (the maximum with a single call).

Since B is a typeless language, arithmetic on characters is quite legal, and even makes sense sometimes:

```
c = c+'A'-'a';
```

converts a single character stored in c to upper case (making use of the fact that corresponding ascii letters are a fixed distance apart).

7. External Variables

```
main( ) {
 extrn a, b, c;
 putchar(a); putchar(b); putchar(c); putchar('!*n');
}

a 'hell';
b 'o, w';
c 'orld';
```

This example illustrates `external variables`, variables which are rather like Fortran COMMON, in that they exist external to all functions, and are (potentially) available to all functions. Any function that wishes to access an external variable must contain an `extrn` declaration for it. Furthermore, we must define all external variables outside any function% For our example variables a, b, and c, this is done in the last three lines.

External storage is static, and always remains in existence, so it can be initialized. (Conversely, `auto` storage appears whenever the function is invoked, and disappears when the function is left. `auto` variables of the same name in different functions are unrelated.) Initialization is done as shown: the initial value appears after the name, as a constant. We have shown character constants; other forms are also possible. External variables are initialized to zero if not set explicitly.

8. Functions

```
main( ) {
  extrn a,b,c,d;
  put2char(a,b) ;
  put2char(c,d) ;
}

put2char(x,y) {
putchar(x);
putchar(y);
}

a 'hell'; b 'o, w'; c 'orld'; d '!*n';
```

This example does the same thing as the last, but uses a separate (and quite artificial) function of two arguments, `put2char`. The dummy names x and y refer to the arguments throughout putchar; they are not declared in an auto statement. We could equally well have made a,b,c,d `auto` variables, or have called `put2char` as

```
put2char( 'hell','o, w');
  etc.
```

Execution of `put2char` terminates when it runs into the closing '}', as in `main`; control returns to the next statement of the calling program.

9. Input; getchar

```
main( ) {
  auto c;
  c= getchar();
  putchar(c);
}
```

`getchar` is another library IO function. It fetches one character from the input line each time it is called, and returns that character, right adjusted and zero filled, as the value of the function. The actual implementation of `getchar` is to read an entire line, and then hand out the characters one at a time. After the '*n' at the end of the line has been returned, the next call to getchar will cause an entire new line to be read, and its first character returned. Input is generally from the terminal (->28).

10. Relational Operators

```
== equal to (".EQ." to Fortraners)
!= not equal to
> greater than
< less than
>= greater than or equal to
<= less than or equal to
```

These are the relational operators; we will use '!=' ("not equal to) in the next example. Don't forget that the equality test is

11 if; goto; Labels; Comments

```
main( ) {
  auto c;
read:
  c= getchar();
  putchar(c);
  if(c != '*n') goto read;
}               /* loop if not a newline */
```

This function reads and writes one line. It illustrates several new things. First and easiest is the comment, which is anything enclosed in /*...*/, just as in PL/I.

`read` is a label - a name followed by a colon ':', also just as in PL/I. A statement can have several labels if desired. (Good programming practice dictates using few labels, so later examples will strive to get rid of them.) The goto references a label in the same function.

The `if` statement is one of four conditionals in B. (while (->14), switch (->26), and "?:" (->16) are the others.) Its general form is

```
if (expression) statement
```

The condition to be tested is any expression enclosed in parentheses. It is followed by a statement. The expression is evaluated, and if its value is non-zero, the statement is executed. One of the nice things about B is that the statement can be made arbitrarily complex (although we've used very simple ones here) by enclosing simple statements in { } (->12)

12. Compound Statements

```
if(a<b) {
  t = a;
  a = b;
  b = t;
}
```

As a simple example of compound statements, suppose we want to ensure that a exceeds b, as part of a sort routine. The interchange of a and be takes three statements in B; they are grouped together by { } in the example above.

As a general rule in B, anywhere you can use a simple statement, you can use any compound statement, which is just a number of simple or compound ones enclosed in { }. Compound statements are not followed by semicolons.

The ability to replace single statements by complex ones at will is one thing that makes B much more pleasant to use than Fortran. Logic which requires several GOTO's and labels in Fortran can be done in a simple clear natural way using the compound statements of B.

13. Function Nesting

```
read:
   c = putchar(getchar());
   if (c != '*n') goto read;
```

Functions which return values (as all do, potentially ->24) can be used anywhere you might normally use an expression. Thus we can use `getchar` as the argument of `putchar`, nesting the functions. `getchar` returns exactly the character that `putchar` needs as an argument. `putchar` also passes its argument back as a value, so we assign this to c for later reference.

Even better, if c won't be needed later, we can say

```
read:
 if (putchar(getchar()) != '*n') goto read;
```

14. While statement; Assignment within an Expression

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

Avoiding is`goto`'s This`goto`'s. the previous one. (The difference? This one doesn't print the terminating '*n'.) The `while` statement is basically a loop statement, whose general form is

```
while (expression) statement
```

As in the `if` statement, the expression and the statement can both be arbitrarily complicated, although we haven't seen that yet. The action of while is

(a) evaluate the expression
(b) if its value is true (i. e., not zero), do the
   statement and return to (a)

Our example gets the character and assigns it to c, and tests to see if it's a '*n'. If it isn't, it does the statement part of the `while`, a `putchar`, and then repeats.

Notice that we used an assignment statement "c=getchar()'" within an expression. This is a handy notational shortcut, usually produces better code, and is sometimes necessary if a statement is to be compressed. This works because an assignment statement has a value, just as any other expression does. This also implies that we can successfully use multiple assignments like

```
x = y = f(a)+g(b)
```

As an aside, the extra pair of parentheses in the assignment statement within the conditional were really necessary: if we had said

```
c = getchar() != '*n'
```

c would be set to 0 or 1 depending on whether the character fetched was a newline or not. This is because the binding strength of the assignment operator '=' is lower than the relational operator '!='. (-> Appendix D)

15. More on Whiles; Null Statement; exit

```
main( ) {
  auto c;
  while (1) {
    while ( (c=getchar()) != ' ')
      if (putchar(c) == '*n') exit();
    putchar( '*n' );
    while ( (c=getchar()) == ' '); /* skip blanks */
    if (putchar(c)=='*n') exit(); /* done when newline */
    }
}
```

This terse example reads one line from the terminal, and prints each non-blank string of characters on a separate line: one or more blanks are converted into a single newline.

Line four fetches a character, and if it is non-blank, prints it, and uses the library function `exit` to terminate execution if it's a newline. Note the use of the assignment within an expression to save the value of c.

Line seven skips over multiple blanks. It fetches a character, and if it's blank, does the statement ';'. This is called a null statement - it really does nothing. Line seven is just a very tight loop, with all the work done in the expression part of the while clause.

The slightly odd-looking construction

```
while(1){
  ---
}
```

is a way to get an infinite loop. "1" is always non-zero; so the statement part (compound here!) will always be done. The only way to get out of the loop is to execute a goto (->11) or break (->26), or return from the function with a `return` statement (->24), or terminate execution by `exit`.

16. Else Clause; Conditional expressions

```
if (expression) statement else statement2
```

is the most general form of the `if` - the `else` part is sometimes useful. The canonical example sets x to the minimum of a and b:

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

If's and else's can be used to construct logic that branches one of several ways, and then rejoins, a common programming structure, in this way:

```
if(---)
  { -----
    ----- }
 else if(---)
  { -----
    ----- }
 else if(----)
  { -----
    ----- }
```

etc.

The conditions are tested in order, and exactly one block is executed - the first one whose if is satisfied. When this block is finished, the next statement executed is the one after the last "else if" block.

B provides an alternate form of conditional which is often more concise and efficient. It is called the "conditional expression" because it is a conditional which actually has a value and can be used anywhere an expression can. The last example can best be expressed as

```
x = a<b ? a : b;
```

The meaning is: evaluate the expression to the left of '?'. If it is true, evaluate the expression between '?'

and ':' and assign that value to x. If it's false, evaluate the expression to the right of ':' and assign its value to x.

17. Unary Operators

```
auto k;
k = 0;
while (getchar() != '*n') ++k;
```

B has several interesting unary operators in addition to the traditional '-'. The program above uses the increment operator is equivalent to "k=k+1" but generates better code. '++' man also be used after a variable, as in

```
k++
```

The only difference is this. suppose k is 5. Then

```
x = ++k
```

sets k to 6 and then sets x to k, i.e., to 6. But

```
x = k++
```

sets x to 5, and then k to 6.

Similarly, the unary decrement operator '--' can be used either before or after a variable to decrement by one.

18. Bit Operators

```
c = o;
i = 36;
while ((i = i-9) >= 0) c = c | getchar()<<i;
```

B has several operators for logical bit-manipulation. The example above illustrates two, the OR '|' , and the left shift <<. This code packs the next four 9-bit characters from the input into the single 36-bit word c, by successively left-shifting the character by i bit positions to the correct position and OR-ing it into the word.

The other logical operators are AND '&', logical right shift

```
x = ~y&0777
```

sets x to the last 9 bits of the 1's complement of y, and

```
x = x&077
```

replaces x by its last six bits.

The order of evaluation of unparenthesized expressions is determined by the relative binding strengths of the operators involved (-> Appendix D).

19. Assignment operators

An unusual feature of B is that the normal binary operators like to form new assignment operators. For example,

```
x =- 10
```

means "x=x-10". '=-' is an assignment operator. This convention is a useful notational shorthand, and usually makes it possible for B to generate better code. But the spaces around the operator are critical! For instance,

```
x = -10
```

sets x to -10, while

```
x =- 10
```

subtracts 10 from x. When no space is present,

```
x=-10
```

also decreases x by 10. This is quite contrary to the experience of most programmers.

Because assignment operators have lower binding strength than any other operator, the order of evaluation should be watched carefully:

```
x = x<<y|z
```

means "shift x left y places, then OR in z, and store in x". But

```
x =<< y|z
```

means "shift x left by y|z places", which is rather different.

The character-packing example can now be written better as

```
c= 0;
i = 36;
while ((i =- 9) >= 0) c =| getchar()<<i;
```

20. Vectors

A vector is a set of consecutive words, somewhat like a one-dimensional array in Fortran. The declaration

```
auto v[10];
```

allocates 11 consecutive words? v[0], v[1], ..., v[10] for v. Reference to the i-th element is made by v[i] ; the [] brackets indicate subscripting. (There is no Fortran-like ambiguity between subscripts and function call.)

```
sum = o;
i = 0;
while (i<=n) sum =+ V[i++];
```

This code sums the elements of the vector v. Notice the increment within the subscript - this is common usage in B. Similarly, to find the first zero element of v,

```
i = -1;
while (v[++i]);
```

21. External Vectors

Vectors can of course be external variables rather than `auto`. We declare v external within the function (don't mention a size) by

```
extrn v;
```

and then outside all functions write

```
v[10];
```

External vectors may be initialized just as external simple variables:

```
v[10]  'hi!', 1, 2, 3, 0777;
```

sets v[0] to the character constant 'hi!', and V[1] through v[4] to various numbers. v[5] through v[10] are not initialized.

22. Addressing

To understand all the ramifications of vectors, a digression into addressing is necessary. The actual implementation of a vector consists of a word v which contains in its right half the address of the 0-th word of the vector; i.e., v is really a pointer to the actual vector.

This implementation can lead to a fine trap for the unwary. If we say

```
auto u[10], v[10];
---
u = v;
v[0] = ...
v[1] = ...
---
```

the unsuspecting might believe that u[0] and u[1] contain the old values of v[0] and v[1]; in fact, since u is just a pointer to v[0] , u actually refers to the new content. The statement "u=v" does not cause any copy of information into the elements of u; they may indeed be lost because u no longer points to them.

The fact that vector elements are referenced by a pointer to the zeroeth entry makes subscripting trivial - the address of v[i] is simply v+i. Furthermore, constructs like v[i][j] are quite legal: v[i] is simply a pointer to a vector, and v[i][j] is its j-th entry. You have to set up your own storage, though.

To facilitate manipulation of addresses when it seems advisable, B provides two unary ad˜ress operators, '*' and '&'. '&' is the address operator so &x is the address of x, assuming it has one. '*' is the indirection operator; "*x" means "use the content of x as an address."

Subscripting operations can now be expressed differently:

```
x[0]  is equivalent to *x
x[1]  is equivalent to *(x+1)
&x[0] is equivalent to x
```

23. Strings

A string in B is in essence a vector of characters. It is written as

```
"this is a string"
```

and, in internal representation, is a vector with characters packed four to a word, left justified, and terminated by a mysterious character known as '*e' (ascii EOT). Thus the string

```
"hello"
```

has six characters, the last of which is '*e'. Characters are numbered from zero. Notice the difference between strings and character constants. Strings are double-quoted, and have zero or more characters; they are left justified and terminated by an explicit delimiter, '*e'. Character constants are single-quoted, have from one to four characters, and are right justified and zero-filled.

Some characters can only be put into strings by "escape sequences" (-> Appendix B). For instance, to get a double quote into a string, use '*"', as in

```
"He said, *"Let's go.*""
```

External variables may be initialized to string values simply by following their names by strings, exactly as for other constants. These definitions initialize a, b, and three elements of v:

```
 a "hello"; b "world'*;
 v[2] "now is the time", "for all good men",
     "to come to the aid of the party";
```

B provides several library functions for manipulating strings and accessing characters within them in a relatively machine-independent way. These are all discussed in more detail in section 8.2 of [1]. The two most commonly used are `char` and `lchar`. char(s,n) returns the n-th character of s, right justified and zero-filled. lchar(s,n,c) replaces the n-th character of s by c, and returns c. (The code for `char` is given in the next section.)

Because strings are vectors, writing "s1=s2" does not copy the characters in s2 into s1, but merely makes s1 point to s2. To copy s2 into s1, we can use char and lchar in a function like `strcopy`:

```
strcopy(sl ,s2){
auto i;
i = 0;
while (lchar(sl,i,char(s2,i)) != '*e') i++;
}
```

24. Return statement

How do we arrange that a function like char returns a value to its caller? Here is `char`:

```
char(s, n){
  auto y,sh,cpos;
  y = s[n/4];         /* word containing n-th char */
  cpos = n%4;         /* position of char in word */
  sh = 27-9*cpos;     /* bit positions to shift */
  y =  (y>>sh)&0777; /* shift and mask all but 9 bits */
  return(y);          /* return character to caller */
}
```

A `return` in a function causes an immediate return to the calling program. If the `return` is followed by an expression in parentheses, this expression is evaluated, and the value returned to the caller. The expression can be arbitrarily complex, of course: `char` is actually defined as

```
char(s,n) return((s[n/4]>>(27-9*(n%4)))&0777);
```

Notice that char is written without { }, because it can be expressed as a simple statement.

25. Other String Functions

concat(s,al,a2,...,a10) concatenates the zero or more strings ai into one big string s, and returns s (which means a pointer to S[0]). s must be declared big enough to hold the result (including the terminating '*e').

getstr(s) fetches the entire next line from the input unit into s, replacing the terminating '*n' by '*e'. In B, it is

```
getstr(s){
auto c,i;
while ((c=getchar()) != '*n') lchar(s,i++,c);
lchar(s,i,'*e');
return(s) ;
}
```

Similarly putstr(s) puts the string s onto the output unit, except for the final '*e'. Thus

```
auto s[20] ;
putstr(getstr(s)); putchar('*n');
```

copies an input line to the output.

26. Switch statement; break

```
loop:
  switch (c=getchar()){

   pr:
         case 'p':  /* print */
         case 'P':
           print();
           goto loop;

         case 's':  /* subs */
         case 'S':
           subs() ;
           goto pr;

         case 'd':  /* delete */
         case 'D':
           delete();
           goto loop;

         case '*n': /* quit on newline */
           break;

         default:
           error();  /* error if fall out */
           goto loop;

   }
```

This fragment of a text-editor program illustrates the `switch` statement, which is a multi-way branch. Here, the branch is controlled by a character read into c. If c is 'p' or 'P', function `print` is invoked. If c is an 's' or 'S', `subs` and `print` are both invoked. Similar actions take place for other values. If c does not mention any of the values mentioned in the case statements, the default statement (in this case, an error function) is invoked.

The general form of a switch is

```
switch (expression) statement
```

The statement is always complex, and contains statements of the form

```
case constant:
```

The constants in our example are characters, but they could also be numbers. The expression is evaluated, and its value matched against the possible cases. If a match is found, execution continues at that case. (Execution may of course fall through from one case to the next, and may transfer around within the switch, or transfer outside it.) Notice we put multiple cases on several statements.

If no match is found, execution continues at the statement labeled `default:` if it exists; if there is no match and no `default`, the entire statement part of the switch is skipped.

The `break` statement forces an immediate transfer to the next statement after the switch statement, i.e., the statement after the {} which enclose the statement part of the `switch`. `break` may also be used in while statements in an analogous manner. `break` is a useful way to avoid useless labels.

27. Formatted IO

In this section, we point to the function `printf`, which is used for producing formatted IO. This function is so useful that it is actually placed in Appendix C for ready reference.

28. Lots More on IO

File IO is described in much more detail in [1], section 8. This discussion does only the easy things.

Basically, all IO in B is character-oriented, and based on getchar and putchar. By default, these functions refer to the terminal, and IO goes there unless explicit steps are taken to divert it. The diversion is easy and systematic, so IO unit switching is quite feasible.

At any time there is one input unit and one output unit; `getchar` and `putchar` use these for their input and output by implication. The unit is a number between -1 and 10 (rather like Fortran file numbers). The current values are stored in the external variables `rd.unit` and `wr.unit` so they can be changed easily, although most programs need not use them directly. When execution of a B program begins, `rd.unit` is 0, which means "read from terminal", and `wr.unit` is -1, meaning "write on terminal".

To do input from a file, we set `rd.unit` to a value, and assign a file name to that unit. Then, until the assignment or the value is changed, `getchar` reads its characters from that file. Thus, to open for reading an ascii permanent file called "cat/file", as input unit 5,

```
openr(5, "cat/file")
```

The first argument is the unit number, and the second is a cat/file description. (An empty string means the terminal.) `rd.unit` is set to 5 by `openr` so successive calls to `getchar`, `getstr`, etc., will fetch characters from this file. If the file is non-existent or if we read to end of file, all successive calls produce '*e'.

Writing is much the same:

```
openw(u,s)
```

opens ascii file s as unit u for writing. Successive calls to `putchar`, `putstr`, `printf`, etc., will place output on file s.

If the IO unit mentioned in `openr` or `openw` is already open, it is closed before the next use. That is, any output for output files is flushed and end of file written; any input from input files is thrown away. Files are also closed this way upon normal termination.

Programs which have only one input and one output unit open simultaneously need not reference `rd.unit` and `wr.unit` explicitly, since the opening routines will do It automatically. Otherwise, don't forget to declare them in an `extrn` statement.

The function `flush` closes the current output unit without writing a terminating newline character. If this file is the terminal, this is the way to force a prompting question before reading a response on the same line:

```
putstr("old or new?");
flush();
getstr(s);
```

prints

```
        old or new?
```

and reads the response from the same line.

File names may only be "cat/file", "/file", or "file": no permissions, passwords subcatalogs or alternate names are allowed. File names containing a '/' are assumed to be permanent files; if you try to open a permanent file which is already accessed, it is released and reaccessed. If it can't be found, the access fails. A file name without '/' may or may not be a permanent file, so if a file of this name is already accessed, it is used without question. If it is not already accessed, a search of the user's catalog is made for it. If it can't be found there either, and writing is desired, a temporary file is made.

The function `reread` is used to re-read a line. Its primary function is that if it is called before the first call to `getchar`, it retrieves the command line with which the program was called.

```
reread();
getstr(s);
```

fetches the command line, and puts it into string s.

29. System Call from TSS

TSS users can easily execute other TSS subsystems from within a running B program, by using the library function `system`. For instance,

```
system("./rj (w) ident;file");
```

acts exactly as if we had typed

```
./rj (w) ident;file
```

in response to "SYSTEM?". The argument of `system` is a string, which is executed as if typed at "SYSTEM?" level. (No '*n' is needed at the end of the string.) Return is to the B program unless disaster strikes. This feature permits B programs to use TSS subsystems and other programs to do major functions without bookkeeping or core overhead.

We can also use `printf` (->Appendix C) to write on the "system" IO unit, which is predefined as -1. All output written on unit -1 acts as if typed at "SYSTEM?" level. Thus,

```
extrn wr.unit;
---
wr.unit = -1;
opts = "(w)";
fname = "file";
printf("./rj %s ident;%s*n", opts,fname');
```

will set the current output unit to -1, the system, and then print on it using `printf` to format the string. This example does the same thing as the last, but the RUNJOB options and the file names are variables, which can be changed as desired. Notice that this time a '*n' is necessary to terminate the line.

Because of serious design failings in TSS, there is generally no decent way to send more than the command line as input to a program called this way, and no way to retrieve output from it.

30. Recursion

```
putnumb(n) {
  auto a;
  if(a=n/10)   /* assignment, not equality test */
     putnumb(a);  /* recursive */
     putchar(n%10 + '0');
 }
```

This simplified version of the library function `putnum` illustrates the use of recursion. (It only works for n>0.) "a" is set to the quotient of n/10; if this is non-zero, a recursive call on putnumb is made to print the quotient. Eventually the high-order digit of n will be printed, and then as each invocation of putnumb is popped up, the next digit will be added on.

31. Argument Passing for Functions

There is a subtlety in function usage which can trap the unsuspecting Fortran programmer. Argument passing is call by value", which means that the called function is given the actual values of its arguments, and doesn't know their addresses. This makes it very hard to change the value of one of the input arguments!

For instance, consider a function flip(x,y) which is to interchange its arguments. We might naively write

```
flip(x,y){
   auto t;
   t = x;
   x = y;
   y = t;
 }
```

but this has no effect at all. (Non-believers: try it!) What we have to do is pass the arguments as explicit addresses, and use them as addresses. Thus we invoke `flip` as

```
flip(&a,&b);
```

and the definition of flip is

```
flip(x,y){
auto t;
 t = *y;
 *x = *y;
 *y = t;
 }
```

(And don't leave out the spaces on the right of the equal signs.) This problem doesn't arise if the arguments are vectors, since a vector is represented by its address.

32. Calling Fortran and GMAP Subroutines from B Programs

As we mentioned, B doesn't have floating point, multi-dimensional arrays, and similar things that Fortran does better. A B program, `callf`,
allows B programs to call Fortran and GMAP subroutines, like this:

```
callf(name, a1,a2, . . .,a10)
```

Here `name` is the Fortran function or subroutine to be used, and the a's are the zero or more arguments of the subroutine.

There are some restrictions on `callf`, related to the "call by value" problem mentioned above. First, variables that are not vectors or strings must be referenced by address: use "&x" instead of "x" as an argument. Second, no constants are allowed among the arguments, because the construction "&constant" is illegal in B. We can't say

```
tod = callf(itimez,&0)
```

to get the time of day; instead we have to use

```
t=0;
tod = callf(itimez,&t) ;
```

Third, it is not possible to return floating-point function values from Fortran programs, since the results are in the wrong register. They must be explicitly passed back as arguments. Thus we can't say

```
s  = callf(sin,&x)
```

but rather

```
callf(sin2,&x,&s)
```

where sin2 is defined as

```
subroutine sin2(x,y)
y = sin(x)
return
end
```

Fourth, `callf` isn't blindingly efficient, since there's a lot of overhead converting calling sequences. (Very short GMAP routines to interface with B programs can often be coded quite efficiently see [1], section 11.)

1273-bwk-pdp

B. W. KERNIGHAN

Attached
References
Appendix A,B,C,D

References

[1] Johnson, S. C. A User's Reference to B on MH-TSS. Bell Laboratories memorandum.

[2] Canaday, R. H., Ritchie, D. M. Bell Laboratories BCPL. Bell Laboratories memorandum.

Appendix A: B Compiler Diagnostics

Diagnostics consist of two letters, an optional name, and a source line number. Because of the free form input, the source line number may be too high.

$)	-	{ } imbalance
()	-	( ) imbalance
*/	-	/**/ imbalance
[]	-	[] imbalance
>c	-	case table overflow (fatal)
>e	-	expression stack overflow (fatal)
>i	-	label table overflow (fatal)
>s	-	symbol table overflow (fatal)
ex	-	expression syntax error
lv	-	address expected
rd	name	name redeclaration - multiple definition
sx	keyword	statement syntax error
un	name	undefined name - line number is last line of function where definition is missing
xx	-	syntax error in external definition

Appendix B: Escape Sequences and Typography

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	null (ascii NUL = 000)
*e	end of string (ascii EOT = 004)
*(	{
*)	}
*t	tab
**	*
*'	'
*"	"
*n	newline

Users of non-ascii devices like model 33/35 Teletypes or IBM 2741's should use a reasonably clever text-editor like QED to create special characters for B programs.

Appendix C: printf for Formatted IO

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

This function writes the arguments ai (from 0 to 10 of which may be present) onto the current output stream under control of the format string `fmt`. 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 (ascii)
d -- decimal number
o -- octal number
s -- character string

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

Thus the call

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

leads to 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 (->29) and the "filsys" subsystem:

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

Appendix D: Binding Strength 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]
* / % (binary)   [LR]
+ - [LR]
>> << [LR]
> < <= >=
== !=
&  [LR]
^  [LR]
|  [LR]
?: [RL]
= =+ =- etc. (all assignment operators) [RL]
```

Appendix E: Index

[omitted]