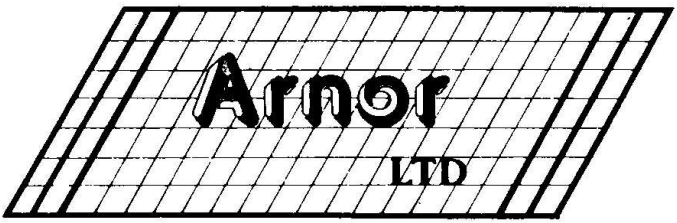
BCPL

THE SYSTEMS PROGRAMMING LANGUAGE

Amstrad PCW8256/8512

Amstrad CPC6128/664/464



BCPL
THE SYSTEMS PROGRAMMING LANGUAGE

Amstrad PCW8256/8512
Amstrad CPC6128/664/464

by Phillip Blenkinsopp

CONTENTS

Issue 1, 1986 (v1.00/2.00)

The BCPL compiler was developed using the MAXAM assembler ROM.
This manual was written on the PROTEXT word processor and checked using PROSPELL.

Arnor Ltd., 118 Whitehorse Road, Croydon, CR0 2JF.

1. ABOUT BCPL

BCPL is often described as a "systems programming language". This makes BCPL sound very grand and technical, but this is not the case. BCPL is certainly a very good language for writing programs such as operating systems, compilers and word processors but the features that make BCPL ideal for these applications also make it an excellent general purpose language.

What are these features? First, flexibility. BCPL programmers are not constrained by rules as to which operations can be performed on which type of variable. Indeed there are no different variable types - all variables are simple numbers and other types and structures are achieved by using variables differently. This makes BCPL very different from other languages such as BASIC and PASCAL. A string is treated as a variable pointing to a string, similarly a vector. Even procedures are simply variables containing the address of the procedure, and these may be assigned to or passed as parameters.

This flexibility allows the BCPL programmer to access any part of memory (indeed any individual bit) and so do things that can normally only be done in machine code. Of course freedom always carries a price - responsibility. In languages such as PASCAL the compiler detects an attempt to perform an illegal or meaningless operation. In BCPL it rarely does so and it is quite possible to crash the system by corrupting the system memory or jumping to an illegal address.

An essential book to read

This manual gives an introduction to certain aspects of BCPL, as well as a summary of the facilities implemented in this version. It does not attempt to be a complete guide of BCPL programming, and assumes some knowledge of programming (for example in BASIC). Note that examples of the use of most features of BCPL are to be found among the example programs contained on the disc.

The definitive book on BCPL, both as an introduction and a reference is "BCPL - the language and its compiler" by Martin Richards and Colin Whitby-Strevens.

This book is essential reading for the BCPL programmer. It is published by Cambridge University Press at a reasonable price in paperback. Martin Richards originally designed BCPL in Cambridge in 1967. This book will henceforth be referred to as "the BCPL book".

In particular the book contains an excellent section on "Advanced facilities" which covers, among other things, pointers, bit operations, recursion and scope rules.

For the purposes of this manual, the BCPL book is regarded as defining "standard" BCPL, and many minor variations or enhancements to this standard are pointed out.

2. USING ARNOR BCPL UNDER CP/M

CPC users may use BCPL under CP/M (either 2.2 or Plus). The supplied disc has the CP/M version of the program on one side, and the AMSDOS version on the other.

Note: PCW8256/8512 users should ignore the AMSDOS side of the disc and the EPROM, as well as any references to Amsdos or the EPROM/ROM version, throughout this manual. These are for CPC users only.

The newcomer to the language is advised to pass over chapters 2, 3 and 4 on the first reading as they describe the mechanics of using the compiler, rather than details of the language.

List of files contained on the disc - CP/M side

The main files are:

BCPL.COM	the main compiler program
CLIBHDR	the main I/O library
CLIBHDR1	the extra I/O library containing less used routines
DEBUG	a routine useful for debugging BCPL programs

The disc also contains the following example files:

BEDC.B	BCPL editor source code, CP/M version
BEDC.COM	BCPL editor object code, CP/M version
COPY.B	a simple example program which copies a file
KEYDEFS.PCW	a key definition file for use with BEDC.COM on the PCW8256/8512.
TIDY.B	A BCPL program to format BCPL source code tidily

Creating a working disc

NOTE:Under no circumstances should the original disc be used as a working disc for normal use. The required files should copied onto a system disc and the original retained in a safe place, as a backup.

A system disc should be created using DISCKIT(DISCKIT3,DISCKIT2 or FORMAT, as appropriate on the CPC range) to format a disc. The first four files in the above list should then be copied (using PIP) from the supplied BCPL disc onto the newly formatted disc, together with (if using CP/M Plus) the EMS file from the CP/M System Utilities.

This will create a disc suitable for use, but you might also decide to add further files, such as an editor or SETKEYS.COM (see below), to the disc.

Entering and editing BCPL programs under CP/M

Any editor or word processor may be used for typing in BCPL programs, as long as ASCII file can be produced.

The CP/M version of PROTEXT is specially suitable (using program mode), as is the Arnor program editor supplied with other Arnor CP/M programs. Alternatively CPC users can use the AMSDOS versions of PROTEXT and MAXAM.

If you do not have a suitable editor the program BEDC.COM can be used, and this should also be copied onto the system disc. This is a simple full screen editor which is compatible with PROTEXT editing commands.

Note: If using BEDC.COM with a PCW8256/8512, the keys must be defined using SETKEYS. Suitable definitions are provided in the file KEYDEFS.PCW and the syntax used is: SETKEYS KEYDEFS.PCW

This can be incorporated into a PROFILE.SUB file, in which case, both SETKEYS.COM and SUBMIT.COM must also be copied onto the working disc.

Compiling a BCPL program under CP/M

The CP/M version of the compiler takes a file containing BCPL source code and produces a COM file containing an executable CP/M object code program. There is no intermediate compilation stage. The command to compile a program is:

bcpl <source><destination>

where <source> and <destination> are the names of the files which code is taken and to which it is written. The destination filename is optional and if omitted will default to the source file name with a .COM extension. The extensions on both filenames are optional. If the source name has no extension then the compiler will first search for a file with no extension and then for a file with a .B extension. The same applies in the GET directive on both CP/M and AMSDOS versions. If the destination file has no extension then this will default to .COM.

e.g. bcpl tidy

will first look for the file TIDY and if it is not present will then look for the file TIDY.B. The compiler will read this file and then send the object code produced to a file TIDY.COM.

Under CP/M Plus the compiler returns an error return code if an error occurred. It can then be convenient to use SUBMIT to compile and run the program with a single command. The CP/M conditional command facility can be used so the program is only executed if no error occurred. For example a file COMPILE.SUB may contain the following:

```
bcpl program.b
<y
:program
```

The second line causes the previous version to be deleted. Note that this will only work under CP/M Plus (CPC6128 and PCW8256/8512) and not under CP/M 2.2.

Including other files

Other BCPL source files may be included at any point in a program by using the GET directive.

```
GET "file"
```

causes compilation to continue with the new file. at the end of the included file compilation resumes at the point after the GET directive in the original file.

Most programs will start by including the library routines:

```
GET "CLIBHDR"
GET "CLIBHDR1"          // often not needed
GET "DEBUG"             // used when debugging (see section 8k)
```

Note that the GET directives to include the library routines must be in the order shown because routines in CLIBHDR1 call other routines in CLIBHDR.

Compilation stops on the CP/M version when a full stop '.' is found in the initial source file which was specified in the original command line, or when the end of that file is reached.

3. USING ARNOR BCPL UNDER AMSDOS

NOTE: PCW8256/8512 users should ignore this chapter.

BCPL may be used from Amsdos on the CPC range, either in ROM form or from a disc. If the ROM version is being used, the files BCPL.BAS and BCPL.BIN are not required, though the library files will still be required.

NOTE: Under no circumstances should the original disc be used as a working disc for normal use. The required files should be copied onto a system disc and the original retained in a safe place, as a backup.

Creating a working disc

A new disc should be formatted and the first six files from the list below should be copied onto it, using either FILECOPY or PIP from CP/M, depending on the model being used, or the COPY command from the UTOPIA ROM if that is installed. This will produce a disc containing all the essential files. ROM users can omit the first two files called BCPL.BAS and BCPL.BIN.

Installing the ROM version

Fit the 16k EPROM into your ROM board following the instructions supplied with the rom board. The ROM selection number is not critical, though it must not clash with any existing ROMs, even if they are not installed on the ROM board. (The disc rom for example uses ROM select 7).

Suitable boards may be purchased from Arnor if you do not already have one.

The BCPL ROM provides an extremely convenient and fast environment for developing BCPL programs, particularly when used in conjunction with PROTEXT or MAXAM. Programs can be typed in using PROTEXT and MAXAM and compiled directly from memory.

The ROM version may only be used on the CPC464, CPC664, and CPC6128 computers.

List of files contained on the disc - AMSDOS side
The following files are most important:

BCPL.BAS	the loader for the compiler program
BCPL.BIN	the main compiler program
ALIBHDR	the main I/O library
ALIBHDR1	the extra I/O library containing less used routines
AMSDOS	a further I/O library containing graphics routines
DEBUG	a routine useful for debugging BCPL programs

The disc also contains the following example files:

BED.B	BCPL editor source code, AMSDOS version
BED.BIN	BCPL editor object code, AMSDOS version
BED1.B	a file containing directives to compile BED
DIS.B	a Z80 disassembler written in BCPL
INVADE.B	a space invaders game, written in BCPL

Load the AMSDOS disc version

To load the compiler type: RUN"BCPL"

Entering and editing BCPL programs under AMSDOS

Any editor or word processor may be used for typing in BCPL programs, as long as an ASCII file can be produced.

PROTEXT (using program mode) and MAXAM are especially suitable, in particular the ROM versions are recommended for convenience.

If you do not have a suitable editor the program BED.BIN can be used. This is a simple full screen editor which is compatible with PROTEXT editing commands.

Any source files created should preferably be saved to disc with a '.B' suffix to the filename. This is not essential, but does help identify the type of files on a disc.

Compiling a BCPL program under AMSDOS

Apart from having to load the disc version, the disc and ROM versions are used in exactly the same way. There are, however, a few differences when compared to the CP/M version. The major difference is that the CP/M version allows the inclusion of source files within other source files. The AMSDOS version does not allow this although it does allow the inclusion of stored PROTEXT /MAXAM text. Due to this limitation the AMSDOS version has a simple front end which allows you to type in source code which will then be submitted directly to the compiler.

To run the compiler type: |BCPL

The screen will clear and the BCPL signon message will be printed. You will then be prompted with "Output file name?". The name you type here is the name of the file into which the program will be compiled. It is also possible (unless the program is too large) to compile directly into memory. To do this just press RETURN (or ENTER) in response to the question.

You will then be presented with the BCPL prompt '->'. The compiler is now ready to accept BCPL code for compilation. This will almost always consist of one or more GET directives. There are two forms permitted:

(i) GET "file" to include another previously prepared file

(ii) GET to include the current text (PROTEXT/MAXAM)

Any number of GET directives may be typed. After each one the compiler will perform the first part of the compilation. To end the compilation and produce the object code type a full stop '.'.

Note: If the '.B' extension of a source filename is omitted from the filename in a GET command when the filename contains it, Amsdos will initially produce the message 'file not found' but will continue to search for a file with the .B extension and compile it.

Testing BCPL programs under AMSDOS

If a program is compiled to memory, as described above, the command '|GO' may be used to run the program.

Typing '|GO' will run the most recently compiled BCPL program. If no program has been compiled, or an error occurred on compilation, then |GO will simply print the message 'No program'.

WARNING - it is possible to corrupt a compiled program in memory by editing the PROTEXT or MAXAM text, so after editing the program should always be compiled again before using GO.

With large programs it is quite likely that there will be insufficient memory to compile to memory. There are three courses of action that may be taken here:

(i) if you are using the disc version, buy a ROM board and use the ROM version. You will gain an extra 16k of memory.

(ii) compile directly to disc.

(iii) use the CP/M version.

Extra facilities for ROM users

ROMON7

Some programs (particularly games) will not run with a rom installed. The command |ROMON7 is provided to enable the BCPL rom (and any others) to be turned off. All roms except the AMSDOS rom will be turned off. Note that ROMON7 will reset the computer and lose the contents of memory.

RUN"DISC

When the BCPL ROM is installed, pressing CTRL-ENTER will generate the command RUN"DISC. Thus a file 'DISC' will be run when CTRL-ENTER is pressed.

4. COMPILER OPTIONS

The OPTION directive controls various compiler facilities. It does not cause any code to be generated. There are 5 options on the AMSDOS versions of the compiler and 3 on the CP/M version. Multiple options may be selected with a single OPTION directive by separating the options with commas. The OPTION directive may appear anywhere in a program so options can be selected for any part of a program.

The three options common to all versions are as follows:

If the option is followed by a '+' sign the option will be switched on and if followed by a '-', it will be switched off.

L - switch listing on or off

e.g. OPTION L+

S - switch stack checking on or off. If checking is on then the compiler will check for stack overflow at the entry point of each procedure. If stack checking is off then the stack may overflow and crash the machine.

e.g. OPTION S-

B - switch break checking on or off. If checking is on then the object code produced will check whether the break key (ESC or STOP) has been pressed at the start of loops and at the entry points of procedures. A number may be specified to control how often the key is checked. The default is 10, which means it is checked once for every ten loops or procedures.

e.g. OPTION B-
OPTION B+25

The defaults for the above options are listing off, stack checking on, break checking on.

The remaining two options are only available on AMSDOS versions.

O - sets the code origin to the value of the number which follows it.
e.g. OPTION O #x172

will produce code which will run at address 172 (hex).

H - sets the highest byte usable by the compiled code. This is so that programs compiled on one machine can be run on another.
e.g. OPTION H 40000

The origin defaults to the first unused byte of memory at compile time (after any BASIC program or text) and the highest byte defaults to the last unused byte of memory at compile time (usually HIMEM).

The main use for the H and O options is when producing stand alone code to be run from a disc with no BASIC support etc. These options enable all of the memory to be used for a program. It should be noted however that if the Z80 stack pointer goes below #x4000 or above #xC000 then the machine will probably crash due to the ROM paging which uses these areas of memory.

5. SOME KEY FACTS ABOUT BCPL

(i) BCPL is a structured language

A BCPL program is made up of a number of named procedures, each of which is quite separate. There is one special procedure, called 'start', which must be present in every BCPL program. This is where execution will begin. This procedure will call other procedures, which will in turn call other procedures, and so on.

You will later come across terms such as blocks, compound commands and scope - terms that apply to structured languages in general. Examples of other structured programming languages are Pascal, Modula 2 and C. Locomotive BASIC and Mallard BASIC are not structured languages.

(ii) BCPL is a compiled language

BASIC is usually an interpreted language. When a program is run, the BASIC interpreter reads each line and executes the appropriate machine code instructions. This decoding is done every time a line is executed. The result is that BASIC programs run very slowly when compared with a machine code program doing the same thing.

A BCPL program must be submitted to the compiler before it can be run. The compiler translates the entire program into machine code instructions. The machine code program is saved and can then be run. Since each BCPL line is only translated once, a BCPL program runs a lot faster than a corresponding BASIC program.

The disadvantage of using a compiled language is that you have to wait for the program to be compiled each time before you can test it. This is not a problem with Arnor BCPL since it compiles extremely quickly.

(iii) BCPL is a typeless language

Most languages have a number of internal variable types. BASIC usually has 6 - integer, real, string, integer array, real array and string array. PASCAL and C have rather more, and have the ability for the programmer to define his own using "record" or "struct".

BCPL has just one internal variable type - a variable that can be assigned a single word value. This is referred to as bit-pattern indicating that no inherent meaning is attached to this value. The meaning of the value is determined by the way that the programmer uses it, and this meaning can change. In Arnor's implementation of BCPL each value is 16 bits long. There follow some examples of how some of the common data types are used in BCPL. Note that in each case a 'LET' declaration is required to define a variable before it can be used.

Simulating different variable types

Integer: LET number=?
number:=42
This assigns the value 42 (that is the bit pattern 0000000000101010) to the variable 'number'.

Character: LET char=?
char:='A'
This assigns the number 65 (the ASCII code representing the letter 'A') to the variable 'char'.

Vector: LET array=VEC 10
FOR i=0 TO 10 DO array!i:=i
This sets aside an 11 word (22 byte) area of memory, and assigns the address of the start os this area to the variable 'array'. The vector is accessed by the indirection operator '!', where 'array!i' means the contents of address (array+i). !array means the same as array!0.

Two dimensional array:
MANIFEST $(m=3,n=4$)
LET array=VEC(m+1)*(n+1)-1
LET getarray(i,j)=array!(m*i+j)
LET putarray(i,j,value) BE array!(m*i+j):=value
Multi-dimensional arrays can be implemented by defining a vector and calculating the required word from the array subscripts.

String: LET text="BCPL is a typeless language"
This stores the string "BCPL is a typeless language" somewhere in memory, preceded by a byte containing the length of the string. The value assigned to the variable 'text' is the address of this length byte. Note that this is only possible because the memory address is the same length as the BCPL word (this is not a coincidence!).

Note: There is a very important difference between the use of single and double quotes, and confusing these can lead to bugs that are very difficult to find, so be careful! The BCPL book contains a very useful section on the pitfalls of BCPL programming.

Single quotes enclose a character and define a constant whose value is the ASCII code of that character.

Double quotes enclose a string and define a constant whose value is the address of that string.

6. TWO BCPL PROGRAMS

(i) A very simple BCPL program

This simply displays a message on the screen.

```
GET "CLIBHDR"
LET start() BE
$(
        writes("Hello everyone!")
$)
```

Point to note:

1. Let and BE are BCPL keywords. BCPL keywords are printed in uppercase throughout this manual for clarity. It is not necessary to type them in uppercase as the compiler ignores the distinction.

2. start(). This is the procedure that must be present. The brackets indicate that 'start' is the name of a procedure. They will often contain parameters for the procedure.

3. $(and $). These are section brackets, marking the start and the end of the procedure 'start'.

4. writes. This is a BCPL procedure (note the brackets containing the parameter). It is not a BCPL keyword and so the procedure must be defined somewhere (see below).

5. GET "CLIBHDR". This is a directive (an instruction to the compiler) to compile the BCPL code contained in the file 'CLIBHDR'. This is a library of useful procedures, including one called 'writes'.

(ii) A slightly more complex BCPL program

```
GET "clibhdr"

/* This is a copy routine */

LET start() BE

$(
        LET inname=VEC20    ; LET inbuff=VEC 82
        LET outname=VEC20 ; LET outbuff=VEC 82
        LET c=0
```

```
        writes("Input file name:    ")
        reads(inname)
        writes("*nOutput file name:")
        reads(outname)
        TEST findinput(inname,inbuff) THEN
        $(
                selectinput(inbuff)
                TEST findoutput(outname,outbuff) THEN
                $(
                        selectoutput(outbuff)
                        $(
                                c:=rdch()
                                UNLESS c=endstreamch DO wrch(c)
                        $)
                        REPEATUNTIL c=endstreamch
                        endwrite()
                $)
                ELSE writes("Can't open output file")
                endread()
        $)
        ELSE writes("Can't open input file")
$)
```

Points to note:

1. The use of /* and */ to enclose comments.

2. The use of LET to declare the variables 'inname', 'inbuff', 'outname', 'outbuff' and 'c'. All variables must be declared before use, and the declarations must appear before any commands.

3. The use of VEC to define vectors as described above.

4. reads. Another library procedure, to read a string from the keyboard into the vector given as parameter.

5. findoutput, findinput, readch, writech. Library procedures to open files, and read and write characters.

6. TEST ... THEN ... ELSE. A conditional command. Section brackets are used to enclose the block of commands that will be executed if the test succeeds or fails.

7. REPEATUNTIL. The preceding block (enclosed in section brackets) is repeated until the condition becomes true (in this case when the ene of the file is reached).

7. SUMMARY OF THE BCPL LANGUAGE

The BCPL language is best considered in two parts. First the standard language comprising the commands and syntax for expressions, constants etc. Second the input/output library routines which are mostly procedures written in BCPL. There are a fairly standard set of these, but this implementation includes additional I/O routines.

Elements:

(i) Variable name. Must start with a letter and may also contain digits and full stops. All characters significant.

(ii) Number. Can be decimal, octal(prefix with # or #O), hexadecimal (prefix with #) or binary (prefix with #B).

(iii) String constant. Enclose in double quotes.

(iv) Character constant. Enclose in single quotes.
Certain characters are represented within a string or character constant in a special way:

*"	the " symbol
*'	the ' symbol
**	the * symbol
*B	the backspace character
*C	a carriage return (13)
*N	the newline character (in fact two characters, 13 and 10)
*P	the new page (form feed) character (12)
*S	a space
*T	the tab character
*Xnn	a hexadecimal number between 0 and FF, e.g. *XF1

(v) Truth values, TRUE and FALSE. Pre-defined constants whose values are the bit patterns representing true and false.

Operators:

Addressing operators

!	indirection (subscript),e.g. !a, a!b
@	address of, the inverse of !, so @(!a) = a

Arithmetic operators

+	addition
-	subtraction
*	multiplication
/	integer division
REM	integer remainder

Relational operators

=	equal to (or EQ)
~=	not equal to (or NE)
<	less than (or LT)
>	greater than (or GT)
<=	less than or equal to (or LE)
>=	greater than or equal to (or GE)

Logical operators

|	or (or LOGOR)
&	and (or LOGAND)
~	not (or NOT)
EQV	bitwise equivalence
NEQV	exclusive or

Shift operators

<<	logical shift left,e.g. a << 4 shifts left 4 bits
>>	logical shift right

Other components of expressions:

(i) Function call (see below)

(ii) Conditional expression,e.g. a->b,c
This means: evaluate a as a truth table. If true then the value of the expression is b, otherwise c.

(iii) TABLE,e.g. TABLE a,b,c,d
This is an expression whose value is the address in memory at which the table of values is stored. The values must be constants.

(iv) VALOF (see below)

Order of precedence of operators (highest first)

1. Bracketed expression
2. Function call
3. ! as subscript
4. @, ! as indirection
5. %
6. * / REM ABS
7. + -
8. = ~= < > <= =>
9. << >>
10. ~
11. &
12. |
13. EQV NEQV
14. -> (conditional expression)
15. TABLE
16. VALOF

Note: PCW users will not find the | symbol shown on their keyboard. It is obtained by pressing the EXTRA and the full stop key.

Section brackets and compound commands:

Many of the commands described below apply to a single command. For example REPEAT will repeat a command until the required condition is true. Section brackets are used to group several commands as a 'compound command', which is considered as a single command for these purposes. The section brackets are $(and $). The commands to be grouped should be enclosed between these symbols.

Commands:

Assignment

:= is used to assign a value to a variable. There are three basic forms of the assignment statement:

variable := expression
variable!subscript := expression
!variable := expression

Conditional commands

IF <expression> THEN <command>
UNLESS <expression> do <command>
TEST <expression> THEN <command> ELSE <command>

Note that IF does not take an ELSE clause, TEST must be used. The ELSE clause must be present in a TEST command.

REPEAT and WHILE

<command> REPEAT
<command> REPEATUNTIL <expression>
<command> REPEATWHILE <expression>
WHILE <expression> DO <command>
UNTIL <expression> DO <command>

FOR loop

FOR i = a TO b DO <command>
FOR i = a TO b BY c DO <command>

In the first case i takes values starting at a, incrementing by 1 until greater than b.
In the second case i is incremented by c.c must be a constant expression.

SWTICHON

SWITCHON <expression> INTO <compound command>

The compound command consists of a sequence of commands with case labels attached to commands. A case label takes the form 'CASE <constant>:', and causes execution to start at the following instruction if <expression> has the value <constant>. Execution will continue until an ENDCASE command, or to the end of the SWITCHON compound command. The label 'DEFAULT:' marks the place where execution is to start if <expression> does not evaluate to any of the constants specified by case labels. For an example of the use of SWITCHON see the 'writef' routine in CLIBHDR or ALIBHDR.

Blocks and scope of identifiers

A BCPL program is made up of one or more blocks. A block consists of a sequence of declarations (see below) followed by a sequence of commands. To be precise a BCPL program is a single block, and any block may contain nested sub-blocks.

The scope of an identifier (variable or procedure) defines those parts of a program where that identifier can be used. Each identifier must be declared with a declaration. The scope of that identifier is that same declaration (thus allowing recursive definitions), and all subsequent declarations and commands within that block. Thus if a routine is called which is outside the block (and routines are usually separate blocks) then variables defined within the original block may not be used.

There is an important restriction. Variables defined within a block may not be used within a procedure defined within that block (even though they are in scope). An attempt to do so will produce the error message 'dynamic free variable used'.

Declarations

The LET declaration

All variables and routines must be declared before they can be used. There are several forms of declaration:

LET variable=<expression>

This defines a variable and gives it an initial value. If no initial value is relevant '?' may be used in place of the expression.

LET array=VEC<constant>

This creates a vector with the variable 'array' containing the address of the vector.

Procedures

LET proc(p1,p2,...)=<expression>
LET proc(p1,p2,...) BE <command>

These define a procedure with parameters p1, p2, ... The first form provides a function, as a value is returned. The expression is often a VALOF clause. This takes the following form:

VALOF <command>

The command is usually a compound command which contains one or more RESULTIS commands:

RESULTIS <expression>

When a RESULTIS command is executed the VALOF command has finished and the value of the expression is returned.

The second form of the procedure declaration defines a routine. No value is returned.

Either form of procedure is invoked by a command:

proc(a,b,...)

A function call can appear in an expression.

Other declarations

GLOBAL $(var1:c1; var2:c2; ... $)
STATIC $(var1=val1; var2=val2; ... $)

Dynamic variables (those defined with LET) only have an allocated area of memory when they are within scope. Variables defined with GLOBAL or STATIC have a permanently allocated word of memory. The static declaration allows an initial value to be set. The GLOBAL declaration is redundant in this implementation as it is used when separate compilation is available. It is included for compatibility with other BCPL compilers.

A global vector can be defined as follows. Declare the name of the vector as a static variable, then declare a dummy vector in 'start' and assign this to the static variable. This ensures the vector is in scope throughout the program.

Example:

```
STATIC $( errlev=0 $)
LET start() BE
$(
        LET errvec=VEC 3
        errlev:=errvec
        ...
```

MANIFEST $(name1=val1; name2=val2; ... $)

The MANIFEST declaration attaches a name to a constant. It is not a variable and should be usedby a program to be eeasily changed if necessary.

Simultaneous declarations

A LET declaration may be followed by any number of AND declarations. All these declarations are considered as a single declaration for the purposes of scope of identifiers. This allows mutually recursive procedures.

```
LET proc1(...) BE
$(
...proc2(...)...
$)
AND proc2(...) BE
$(
...proc1(...)...
$)
```

Strings

Strings are stored as follows:

One byte being the length of the string
The string, one character per byte

Since BCPL works with 16 bit words a special means is needed to access the individual characters of a string. This is the infixed byte operator '%'. To access the nth character of a string use : string%n. To access the length of a string use : string%0
These can be read and written to, allowing strings to be manipulated in any desired manner.
Strings can be split over more than one line, by putting one asterisk ('*') at the end of a line, and another immediately before the continuation of the string on the next line.

Transfer of control

FINISH
This causes execution of the program to stop.

RETURN
Control returns to the calling routine.

BREAK
Used within a repetitive command (UNTIL, WHILE, REPEAT, REPEATUNTIL, REPEATWHILE, or FOR) it causes execution to resume at the first command after the repetitive command.

LOOP
Control passes to the point where the repetitive command condition is tested. For a FOR command control passes to the point where the control variable is incremented.

ENDCASE
Control passes to the command after the end of the SWITCHON compound command.

GOTO <expression>
Rarely required in BCPL, the GOTO command evaluates the expression and jumps to the address obtained. The expression is usually a label which is defined by being attached to a command,

e.g. labelname:<command>

Comments and layout of programs

There are two ways of putting comments within BCPL programs.

// causes the remainder of the line to be ignored
/* causes all text to be ignored until the compiler reaches the symbol */

Additionally a > in column 1 causes the whole line to be ignored.
Thus PROTEXT stored commands are ignored.

Multiple command lines are allowed. Commands must be separated by a semicolon (;) if there would otherwise be an ambiguity.

Blank lines and spaces are ignored except where they are required to separate items. PROTEXT soft spaces, soft line feeds, and markers are ignored.

8. THE ARNOR BCPL LIBRARIES

The libraries supplied with the compiler provide procedures for writing to the screen, printer or files and reading from the keyboard or files. Each of these procedures is described below.

The two versions of the compiler need different libraries which, although they look the same from the outside, have different internal workings determined by the different operating systems. There is an additional AMSDOS library which contains routines which are only relevant to the CPC machines.

Manifest constants defined in the libraries

These should be used where appropriate to make programs both readable and portable. They define implementation dependent constants.

name	value	meaning and use
bitsperword	16	BCPL word size, for program portability
bytesperword	2	BCPL word size, for program portability
endstreamch	-1	hard end of file character
maxint	32767	the largest possible integer
minint	-32768	the smallest possible integer
newlinech	269	the new line character, *N, which is expanded to carriage return followed by line feed
printstream	1	stream number to select output to printer
softeofch	26	soft end of file character
tickspersec	300	multiply the value returned by 'time' by this to get the time in seconds. AMSDOS only.
vdustream	0	stream number to select screen/keyboard

Key to procedure descriptions:

Effect: brief explanation of the procedure

Returns: details any value returned by the procedure. All library procedures may be called as routines,whether or not they return a value. Only those procedures that return a value may be called as a function. Note that several procedures return values in a supplied vector, but these will be listed as 'nothing' since there is no function value returned.

Location: this indicates where the procedure is defined. The location for the CP/M version is given first, followed by the location for the AMSDOS version. Most of the library procedures are written in BCPL and are contained in one of the library files. A few routines are pre-defined,i.e. they are written into the compiler.

(a) Output routines

NEWLINE()

Effect: Starts a new line in the currently selected output. The exact effect in Arnor BCPL is to output a carriage return (13) followed by a line feed (10).

Returns: Nothing

Location: CLIBHDR,ALIBHDR

NEWPAGE()

Effect: Starts a new page in the currently selected output. The exact effect in Arnor BCPL is to output the form feed character (12).

Returns: Nothing

Location: CLIBHDR1,ALIBHDR1

WRCH(char)

Effect: The character is sent to the currently selected output stream. Wrch('*N') has the same effect as newline(),i.e. carriage return and line feed are output.

Returns: TRUE if successful, FALSE if an error occurred.

Location: CLIBHDR,ALIBHDR

Examples: wrch(char)
wrch('A')

WRITED(integer,width)

Effect: The integer is output as a signed decimal integer, right justified in a field of the specified width. If this width is insufficient then it will be output in the minimum width. The number output will be in the range (-32768,32767).

Returns: Nothing

Location: CLIBHDR,ALIBHDR

WRITEF(format,a,b,c,d,e,f,g,h,i,j,k)

Effect: WRITEF allows characters to be printed out according to certain formats. The procedure takes one string parameter which gives the template to be used for printing. It also takes another set of parameters (up to 11) which give the items to be printed. Note that it is not necessary to provide a BCPL procedure with all of its formal parameters. The template parameter is a string which may include the % character followed by a conversion character(s). The template string is printed out with the other parameters inserted into it in place of the % character. The conversion characters available are listed below. Some of these must be followed by a number to specify the required field width. These are marked with 'n', which can be a decimal or hex digit (so 'F' gives the maximum width of 15).

- S - print a string
- C - print a character
- Xn - print a hex value. If the field width is too small, only the least significant digits will be output.
- On - print a hex value. If the field width is too small, only the least significant digits will be output.
- In - print an integer in decimal. If the field width is too small, the integer will be output using the least number of spaces.
- N - print an integer in decimal with no leading spaces
- % - print the % character
- $ - skip a parameter

Returns: Nothing

Location: CLIBHDR,ALIBHDR

Example:
```
LET convert=100
writef("%N in hexadecimal is %X4*N",convert,convert)
```

would print

100 in hexadecimal is 0064

WRITEHEX(integer,width)

Effect: The integer is output as an unsigned hexadecimal integer, right justified in a field of the specified width, with leading zeros. If this width is insufficient then the least significant digits are output.

Returns: Nothing

Location: CLIBHDR,ALIBHDR

Example: writehex(number,4)

WRITEN(integer)

Effect: The integer is output in decimal in the minimum width. This is the same as WRITED(integer,0).

Returns: Nothing

Location: CLIBHDR,ALIBHDR

WRITEOCT(integer,width)

Effect: The integer is output as an unsigned octal integer, right justified in a field of the specified width, with leading zeros. If this width is insufficient then the least significant digits are output.

Returns: Nothing

Location: CLIBHDR,ALIBHDR

Example: writeoct(number,6)

WRITES(string)

Effect: The string is output.

Returns: Nothing

Location: CLIBHDR,ALIBHDR

Example: writes("Enter filename:")

WRITET(string,width)

Effect: The string is output in the specified field width. If the width is longer than the string then spaces are output at the right. If the width is shorter than the string, the whole string is printed.

Returns: Nothing

Location: CLIBHDR1,ALIBHDR1

WRITEU(unsigned,width)

Effect: The first parameter is treated as an unsigned integer and output in decimal, right justified in the specified width. If the width is too small the number is printed in the minimum width. The number output will be in the range (0,65535).

Returns: Nothing

Location: CLIBHDR1,ALIBHDR1

(b) Input routines

RDCH()

Effect: Reads a character from the current input stream.

Returns: The character read.

Location: CLIBHDR,ALIBHDR

READN()

Effect: Reads a decimal integer from the current input stream. Leading spaces are ignored.

Returns: The number read.

Location: CLIBHDR,ALIBHDR

READS(string)

Effect: A string is read from the current input stream. The parameter is a vector in which the string will be stored. The string must be terminated by a carriage return. The DEL, or ←DEL key may be used to delete backwards. If ESC or STOP is pressed when the cursor is not at the start of the string, all characters are deleted. If ESC or STOP is pressed when the cursor is at the start of the string, a string of zero length is returned. The string is limited to a length of 255 bytes, so the vector string should be 128 words in length.

Returns: FALSE if ESC or STOP pressed, otherwise TRUE

Location: CLIBHDR,ALIBHDR

Example:
```
LET input=VEC 128
UNLESS reads(input) DO error("Escape")
```

STRTONUM(string,base)

Effect: This function takes a string as it first parameter and the base of the result as its second. It converts the string to a number in that base and returns it as the result.

Returns: The number obtained by the conversion.

Location: CLIBHDR,ALIBHDR

Example: A READX function to input a hexadecimal number is easily written:

```
LET readx()=VALOF

$(
        LET string=VEC 128
        reads(string)
        RESULTIS strtonum(string,16)
$)
```

UNRDCH()

Effect: Puts back the last character read from the currently selected stream. This routine should not be called twice without an intervening call to RDCH, the second call will have no effect. Characters can only be returned to the current stream if RDCH is used, the facility is not provided at the primitive I/O routine level (that is RDVDU and READCH).

Returns: Nothing

Location: CLIBHDR,ALIBHDR

Example:
```
ch:=rdch() REPEATUNTIL ch=13
unrdch()
```

(c) Stream Selection Routines

INPUT()

Effect: This will return a value representing the current input stream. Zero means the keyboard is the current input stream, any other value is the address of the buffer associated with the file from which input is being taken.

Returns: The current input stream

Location: CLIBHDR,ALIBHDR

OUTPUT()

Effect: This will return a value representing the current output stream. Zero means the VDU is the current stream, one means the printer is the current stream, any other value is the address of the buffer associated with the file to which output is being sent.

Returns: The current output stream

Location: CLIBHDR,ALIBHDR

SELECTINPUT(stream)

Effect: This is used to change the currently selected input stream (i.e. that used by rdch). A stream value of zero selects the keyboard. Any other value must be the address of the buffer associated with a file, and causes input to be taken from that file. The file must previously have been opened using FINDINPUT.

Returns: Nothing

Location: CLIBHDR,ALIBHDR

SELECTOUTPUT(stream)

Effect: This is used to change the currently selected output stream (i.e. that used by wrch, writef etc.). A stream value of zero selects the VDU, a value of one selects the printer. Any other value must be the address of the buffer associated with a file, and causes output to be sent to that file. The file must have been previously opened using FINDOUTPUT.

Returns: Nothing

Location: CLIBHDR,ALIBHDR

Examples: selectoutput(outfilebuf)
selectoutput(printstream)

(d) File I/O Routines

ENDREAD()

Effect: The currently selected input stream is closed.

Returns: TRUE if the file was closed successfully, FALSE if not.

Location: CLIBHDR,ALIBHDR

ENDWRITE()

Effect: The currently selected output stream is closed. It is essential that either this routine or CLOSEOUT is used after writing to a file, as it causes the last section of data to be written to the disc.

Returns: TRUE if the file was closed successfully, else FALSE.

Location: CLIBHDR,ALIBHDR

FINDINPUT(filename,buffer)

Effect: To open an input file. The first parameter is a string giving the name of the file. The second parameter is a vector which will be used for the file buffer. This vector should be 82 words long under CP/M and 1024 words long under AMSDOS. The filename must be a valid CP/M or AMSDOS filename, and may include a drive specifier.

Returns: If the file was opened successfully, the value 'TRUE' is returned, otherwise 'FALSE'.

Location: CLIBHDR,ALIBHDR

Example:
```
LET inbuf=VEC 82
TEST findinput("A:names.dta",inbuf)
THEN selectinput(inbuf)
ELSE error("Cannot open file")
```

FINDOUTPUT(filename,buffer)

Effect: To open an output file. The first parameter is a string giving the name of the file. The second parameter is a vector which will be used for the file buffer. This vector should be 82 words long under CP/M and 1024 words long under AMSDOS. The filename must be a valid CP/M or AMSDOS filename, and may include a drive specifier.

Returns: If the file was opened successfully, the value 'TRUE' is returned, otherwise 'FALSE'.

Location: CLIBHDR,ALIBHDR

Example:
```
LET outbuf=VEC 82
TEST findoutput("A:names.dta",outbuf)
THEN selectoutput(outbuf)
ELSE error("Cannot open file")
```

REWIND()

Effect: The current input stream is rewound. If the current stream is the keyboard this has no effect, if a file the file pointer is set to the start of the file, so the next input will be taken from the start. REWIND is not provided in the AMSDOS version.

Returns: Nothing

Location: CLIBHDR

Example of the use of file I/O routines:

```
LET buffer=VEC 1024
LET filename=VEC 20
writef("Please enter the message file name:")
reads(filename)
TEST findinput(filename,buffer) THEN
$(
        LET char=?
        selectinput(buffer)
        char:=rdch()
        UNTIL char=softeofch LOGOR char=endstreamch DO
                $( wrch(char); char:=rdch() $)
        endread(buffer)
        selectinput(vdustream)
$)
ELSE writef("No messages*N")
```

This will ask for a file name and then, if that file exists, will print out the contents of it.

(e) The Primitive I/O Operations

The library contains primitive routines for reading/writing to files, reading/writing to the VDU, and writing to the printer. The higher level RDCH and WRCH routines switch between these routines when called upon to do so by SELECTINPUT or SELECTOUTPUT. It is quite possible to use the primitive routines separately from RDCH/WRCH and so read/write files and the screen/keyboard at the same time without using SELECTINPUT or SELECTOUTPUT. These routines are RDVDU, WRVDU, READCH and WRITECH.

CLOSEIN(buffer)

Effect: The input file associated with the buffer is closed.

Returns: TRUE if the file was closed successfully, FALSE if not.

Location: CLIBHDR,ALIBHDR

CLOSEOUT(buffer)

Effect: The output file associated with the buffer is closed. It is essential that either this routine or ENDWRITE is used after writing to a file, as it causes the last section of data to be written to the disc.

Returns: TRUE if the file was closed successfully, FALSE if not.

Location: CLIBHDR,ALIBHDR

FLUSH()

Effect: Empties the keyboard buffer.

Returns: Nothing

Location: CLIBHDR,ALIBHDR

PRBUSY()

Effect: Finds out if the printer is busy.

Returns: TRUE if busy, FALSE if not

Location: CLIBHDR,ALIBHDR

Example:
```
IF prbusy() THEN writes("Put the printer on line*N")
WHILE prbusy() LOOP
```

PRCH(char)

Effect: Sends a character to the printer. If the printer is busy then PRCH waits until the character can be sent. PRBUSY should be used to test for the printer being busy. This routine is called by WRCH if the printer is the selected output stream.

Returns: TRUE

Location: CLIBHDR,ALIBHDR

RDTEST()

Effect: Finds out if a key has been pressed.

Returns: TRUE if a key has been pressed, FALSE if not.

Location: CLIBHDR,ALIBHDR

Example:
```
WHILE rdtest()  DO RDCH()
```

RDVDU()

Effect: Reads a character from the keyboard, whatever the currently selected input stream. This routine is called by RDCH if the keyboard is the selected input stream. RDVDU is pre-defined in the CP/M compiler so that it can buffer characters to allow for break testing without losing characters.

Returns: The character read.

Location: Pre-defined in CP/M compiler,ALIBHDR

READCH(buffer)

Effect: Reads a character from a file, whatever the currently selected input stream. The buffer is not needed under AMSDOS. This routine is called by RDCH if a file is the selected input stream.

Returns: The character read.

Location: CLIBHDR,ALIBHDR

REWINDIN(buffer)

Effect: The file pointer for the specified file is set to the start of the file. This routine is called by REWIND if a file is the selected input stream. REWINDIN is not provided in the AMSDOS version.

Returns: Nothing

Location: CLIBHDR

WRITECH(char,buffer)

Effect: Writes a character to a file, whatever the currently selected output stream. The buffer is not needed under AMSDOS. This routine is called by WRCH if a file is the selected output stream.

Returns: TRUE if successful, FALSE if a disc error occurred

Location: CLIBHDR,ALIBHDR

Examples:
```
writech('$',outfilebuf)        // CP/M version
writech('$')                   // AMSDOS version
```

WRVDU(char)

Effect: Writes a character to the screen, whatever the currently selected output stream. This routine is called by WRCH if the screen is the selected output stream.

Returns: TRUE

Location: CLIBHDR,ALIBHDR

(f) Miscellaneous Routines

APTOVEC(function,arraysize)

Effect: This will create a dynamic array of size arraysize and then pass the address of this array along with its size to the function. It will return the result of the function and is equivalent to:

```
LET aptovec(function,arraysize)=VALOF

$( LET v=VEC arraysize          // illegal in BCPL
        RESULTIS function(v,arraysize)
$)
```

Returns: the value returned by the function

Location: Pre-defined,pre-defined

BYTEMOVE(start,dest,size)

Effect: This is a block byte move routine. It works with actual byte addresses (that is the addresses used by the Z80 processor) so if array names are used as parameter then they should be multiplied by two as BCPL works with work addresses. The old and new blocks may overlap.

Returns: Nothing

Location: CLIBHDR1,ALIBHDR1

Example: bytemove(buffer*2+36,buffer*2+16,16)

CALL(address,@af,@bc,@de,@hl,@ix)

Effect: Calls a machine code routine at a byte address. All parameters MUST be supplied as values are returned. The values returned by the machine code routine are passed back in the variables.

Returns: Nothing

Location: CLIBHDR1,ALIBHDR1

CAPITALCH(char)

Effect: If the character is a lower case letter the upper case equivalent is returned, otherwise the original character is returned.

Returns: The converted character

Location: CLIBHDR,ALIBHDR

COMPCH(char1,char2)

Effect: Compares two characters, ignoring the distinction between upper and lower case. It returns the difference between the two characters.

Returns: The difference between the characters

Location: CLIBHDR1,ALIBHDR1

COMPSTRING(string1,string2)

Effect: Compares two strings, ignoring the distinction between upper and lower case. The result is zero if strings are the same, positive if string1 follows string2 alphabetically, and negative if string2 follows string1.

Returns: The result of the comparison, as above

Location: CLIBHDR1,ALIBHDR1

DECVAL(digit)

Effect: This returns the decimal value of a hexadecimal digit.

Returns: The decimal value of the digit

Location: CLIBHDR,ALIBHDR

Example: decval('A') // returns 10

DELETE(filename,buffer)

Effect: The first parameter is a string specifying the name of a file. This file is then deleted from the disc. The CP/M version requires an 82 word buffer, the AMSDOS version requires no buffer.

Returns: TRUE if the operation succeeded, FALSE if a disc error occurred.

Location: CLIBHDR1,ALIBHDR1

Examples:
```
delete("tempfile",buffer)      // CP/M version
delete("tempfile")             // AMSDOS version
```

LEVEL(array)

Effect: This returns the state of the BCPL program in the array given. The array must be at least 4 words in size. This routine is used with LONGJUMP to allow jumps between procedures etc.
See section 9 for more details.

Returns: Nothing

Location: Pre-defined,pre-defined

LONGJUMP(address,array)

Effect: This will set the state of the program according to the array given. The array must have been set previously by the use of LEVEL. Once the state has been set it will jump to address.
See section 9 for more details.

Returns: Nothing

Location: Pre-defined,pre-defined

RANDOM()

Effect: This function returns a sequence of pseudo-random numbers. The sequence will always be the same unless the seed is initialised first. The seed is stored in a static variable called 'randomseed', and this should be initialised to different values to produce different sequences. The sequence runs through all possible 2 byte values.

Returns: A randomly generated integer.

Location: CLIBHDR1,ALIBHDR1

RENAME(filename1,filename2,buffer)

Effect: The first parameter is a string specifying the name of a file. The file is renamed to the second name. The CP/M version requires an 82 word buffer, the AMSDOS version requires no buffer.

Returns: TRUE if the operation succeeded, FALSE if a disc error occurred.

Location: CLIBHDR1,ALIBHDR1

Examples:
```
rename("oldname","newname",buffer)   // CP/M version
rename("oldname","newname")          // AMSDOS version
```

STOP(returncode)

Effect: Execution of the program terminates. Under CP/M Plus the supplied parameter is used to set the program return code. By convention a return code between #xFF00 and #xFFFE is an error code. The return code is ignored under CP/M 2.2 and AMSDOS.

Returns: Nothing

Location: CLIBHDR1,ALIBHDR1

UPPERCASE(char)

Effect: This is the same as CAPITALCH.

Returns: The converted character

Location: CLIBHDR,ALIBHDR

VERSION()

Effect: Returns a number identifying the operating system ans computer on which the program is being executed.
The values returned are:

1 = CPC,AMSDOS
2 = CPC,CP/M 2.2
3 = CPC,CP/M Plus
4 = PCW,CP/M Plus

Returns: The version number

Location: CLIBHDR1,ALIBHDR1

(g) Routines provided in the CP/M libraries only

BDOSA(@c,@de)

Effect: The BDOS is called with the parameters passed in the C and DE registers. The addresses of the variables must be passed in order that results can be returned. The returned values of the A and HL registers from the BDOS are returned in the variables c and de respectively.

Returns: Nothing

Location: CLIBHDR only

BDOSB(c,de)

Effect: This is similar to BDOSA except that results are not returned. The values of c and de are passed, not their addresses.

Returns: Nothing

Location: CLIBHDR only

BIOS(n,@a)

Effect: BIOS routine n is called and the value returned by the BIOS in A is returned in the variable a. The address of a must be passed. this routine is used by PRBUSY.

Returns: Nothing

Location: CLIBHDR only

FIRMWARE(address,@af,@bc,@de,@hl,@ix)

Effect: On CP/M Plus, calls the firmware ROM (CPC6128 only) or the extended firmware jumpblock. The values returned by the firmware routine are passed back in the variables. All parameters MUST be supplied.

Returns: Nothing

Location: CLIBHDR1 only

TAIL(string)

Effect: Takes a vector as parameter and stores the CP/M command tail (from #x80) in the vector as string. This allows BCPL programs to access parameters typed on the CP/M command line. The vector should be 64 words long.

Returns: Nothing

Location: CLIBHDR1 only

(h) Routines provided in the AMSDOS libraries only

EXTERNAL(command,nparms,parmblock)

Effect: This allows a BCPL program to access an AMSDOS external command. It is used by DELETE and RENAME. The parameters are passed in the form required by the firmware for calling an external command. See the DELETE and RENAME routines in ALIBHDR1 for details of how to use this routine.

Returns: Nothing

Location: ALIBHDR1 only

TIME()

Effect: Returns the time in units of 1/300 second, as maintained by the AMSDOS firmware.

Returns: The time

Location: ALIBHDR1 only

(i) Routines provided only for compatibility

These are routines that may be used in existing BCPL programs, but the infixed byte operator now makes their use unnecessary.

GETBYTE(string,pos)

Effect: Gets a character from a string. This routine is included for compatibility with older BCPL systems and would usually be replaced by: string%pos

Returns: The character from the string

Location: CLIBHDR1,ALIBHDR1

PACKSTRING(vector,string)

Effect: Packs a vector containing one character per word into a string containing two characters per word. This routine is included for compatibility with older BCPL systems.

Returns: Nothing

Location: CLIBHDR1,ALIBHDR1

PUTBYTE(string,pos,char)

Effect: Puts a character into a string. This routine is included for compatibility with older BCPL systems, and would usually be replaced by: string%pos:=char

Returns: Nothing

Location: CLIBHDR1,ALIBHDR

UNPACKSTRING(string,vector)

Effect: Unpacks a string containing two characters per word into a vector containing one character per word. This routine is included for compatibility with older BCPL systems.

Returns: Nothing

Location: CLIBHDR1,ALIBHDR1

(j) AMSDOS Graphics and other routines

These routines are included in the file "AMSDOS" and are relevant only to CPC machines. They provide the means for a BCPL program to access the computer's graphics and other firmware facilities, and in most cases correspond directly to a Locomotive BASIC command. For details of how to use these routines consult the file AMSDOS, which lists the equivalent BASIC routines.

List of routines:

border, clg, cls, cursdi, cursen, cursoff, curson, draw, drawr, ent, env, frame, gpen, gpaper, gtest, gtestr, gwindow, gwrch, ink, inkey, inkmode, keyvalid, locate, mode, move, mover, opaque, origin, paper, pen, plot, plotr, release,rsx, sound, speedink, speedkey, stream, strswap, tag, tagoff, transparent, window, xpos, ypos

(k) DEBUG

There is one further library routine provided, which is useful when debugging programs. The routine, DEBUG, is supplied in a separate file, also called 'DEBUG', and provides various options for displaying variable names and contents. These options may be changed with later releases of BCPL and so to ensure accuracy, exact details are only given in the file.

To use DEBUG, put the directive 'GET "DEBUG"' near the start of the program - so it is compiled after the library routines and before the first time DEBUG is called.

It is possible to compile the program without DEBUG being called, but without the need to remove the calls themselves. Simply replace 'GET "DEBUG"' by 'LET debug() BE RETURN'. This will substitute a dummy routine.

9. DIFFERENCES FROM STANDARD BCPL

The INLINE command

This is an extension to the normal BCPL command set and allows Z80 code to be incorporated directly into a BCPL program. The inline command is followed by a list of constants separated by commas. The low byte of these constants is taken and inserted directly into the object code. This allows procedures to call firmware routines etc. and has been used in the LIBHDR files. In order to use this facility properly you must be aware of the way the compiler stores variables and registers which must be left intact. See appendix 1 (technical information).

e.g. INLINE 205,#x5A,#xBB

will call the AMSDOS print character routine - TXT OUTPUT.

The infixed operator %

This is an extension to standard BCPL which is supported. The infixed byte operator will extract a particular byte from a vector. In use this operator is very similar to the ! operator except that it cannot be used as a unary operator and that the second operand always gives the byte offset from the first operator. The main use of this operator is with strings to extract a particular character from the string.

e.g.

```
LET v=VEC 19
FOR a=1 to 39 DO v%a:=' '
v%0:=39                          // set the string length byte
```

will create a string containing 39 spaces.

Undefined initial values

The initial value of a simple variable declared in a LET command may be undefined. This is done by using a question mark (?). This is used to aid readability of programs by indicating that there is no relevant initial value of a variable.

FINDINPUT and FINDOUTPUT

The procedures for opening files are slightly different from standard BCPL in that a buffer must be specified as a parameter, instead of them returning a stream identifier.

Global Variables

When a procedure is defined in standard BCPL the compiler checks to see if a GLOBAL variable with the same name is in scope. If it is in scope, then that variable will be initialised with the address of the procedure. If there is no such variable in scope then a new STATIC variable will be created and that variable initialised with the address of the procedure. When a procedure call is made the code looks up the address in the relevant variable and calls the address. The reason for this indirect calling is to enable easy communication between separately compiled units.

As the ARNOR compiler does not allow separate compilation there is no need for this indirect approach and so a call is made directly to the address of the procedure. This also means that the number associated with a GLOBAL variable in its definition is meaningless as there is no actual GLOBAL vector. The index number is, however, retained for compatibility with other compilers. The outcome of all this is that although it is possible to read the address of a procedure, and so pass procedures as parameters, it is not possible to assign to a procedure. You can however, set up a global variable with a different name from a procedure and then assign the procedure's address to that global variable in your program. This means that the procedure can then be assigned to.

LEVEL and LONGJUMP

Level works in a slightly different way from most BCPL implementations. It usually returns a single 1 word value representing the state of the BCPL stack at a point in the program. Due to the limitations of the Z80 in stack addressing it is not possible to pack the required information into one word with this implementation of BCPL. Instead the LEVEL procedure returns its result in a 4 word vector passed to the procedure. See appendix 1 (technical information) for exact details.

e.g.

```
LET errorlevel=VEC 3
level(errorlevel)
```

Extensions not supported

The following features, which are mentioned in the BCPL book, are not supported: separate compilation, floating point extension, the field selector extension.

A1. TECHNICAL INFORMATION

The compiler allocates dynamic storage from the hardware stack for ordinary local variables. These are accessed by indexing the stack using the IX register which points to the high byte of the last formal parameter minus 127. The formal parameters are stored in reverse order, coming down in memory. Immediately below the first formal parameter is the return address from the procedure which called the currently active procedure. Below is the old IX register from the calling procedure and below this a value which represents the state of the vector space. Local variables now come down from this space in the order that they are declared ans DP will point to the low byte of the last local variable currently in scope.

When a new local variable is declared, its initial value is calculated and then pushed onto the stack. When a vector is allocated, space is taken from the vector space, which is directly after the program, and a pointer to this space is pushed onto the stack.

e.g. for the following procedure

```
LET x(a,b) BE
$(
        LET z=0
        ...
$)
```

at the point marked by the dots, the stack looks like this

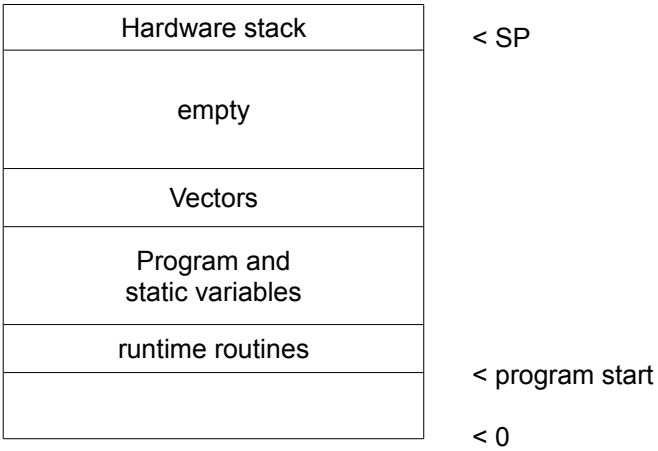
b	IX + 127
a	IX + 125
ret addr	IX + 123
old IX	IX + 121
vec ptr	IX + 119
z	IX + 117, SP points here

When local variables are undeclared (i.e. at the end of their block), the stack pointer is moved up to release the space used.

Static variables are allocated their space at compile time. They are embedded in the code and re-initialised each time the program is run. The stack is placed at the top of the available memory. For CP/M versions this is calculated from the start of the BDOS.

In AMSDOS versions, static variable space is determined at compile time, possibly using the H option. As already mentioned vector space is taken from the area of memory directly after the program. The start of a program is #x100 for CP/M versions and defaults to the first usable byte of memory for AMSDOS. This may be changed by use of the O option.

This leaves a memory map looking like:



When writing any inline code the user must ensure that the IX register and the SP are stored to their former positions when the code finishes. Failure to do this will almost certainly cause the program to crash. The Amstrad also requires that the alternate register set is not used.

The LEVEL library procedure stores information about the state of computation in a 4 word vector. The 4 values which are stored are as follows

word 0 - The old vector base pointer
word 1 - The vector top pointer
word 2 - IX
word 3 - SP

When LONGJUMP is used this information is used to restore the state of computation.

The 14th and 15th bytes of any object code produced always point to the base of the current vector allocation. The 16th and 17th bytes always point to the top of the current vector allocation. All of the vector space required for any particular procedure activation is allocated on entry to that procedure and deallocated on exit. When a vector is declared within a procedure it is given a chunk of memory within this allocation.

The compiler is a two pass compiler. The first pass reads the source text of the program, does syntax analysis of it and produces an applicative expression tree which represents the program. The second pass goes over this tree and produces code for it. The second pass is in fact done twice; the first time no code is produced but the program is checked for any semantic errors such as undefined identifiers or illegal assignments. If this pass runs smoothly with no errors being picked up then the second pass will be repeated, this time generating code.

A2. SUMMARY AND INDEX OF LIBRARY ROUTINES

The letter at the start of each line in the following list of library routines indicates the degree of portability of the routines. Full details are given in Appendix 3.

Note: A means AMSDOS version only, C means CP/M version only

S indicates routine is part of the standard BCPL I/O library
D indicates a standard BCPL routine but with a slight difference in the way it is used.
F indicates frequently found similar routines in BCPL systems
All unmarked routines are Arnor BCPL specific.

	S	packstring	47	compact a string
		prbusy	38	see if the printer is busy
		prch	38	send a character to the printer
	S	putbyte	47	put a byte into a string
	F	random	42	return a pseudo-random number
	S	rdch	31	read a character
		rdtest	38	see if a key has been pressed
		rdvdu	38	read character from keyboard
		readch	39	read a character from the input line
	S	readn	31	read a decimal number
	S	reads	31	read a string
C		readsector	-	read a 128 byte sector from disc
	F	rename	42	rename a file
C	S	rewind	36	rewind the input stream
C		rewindin	39	rewind an input file
	S	selectinput	33	change the input stream
	S	selectoutput	34	change the output stream
C		setfcb	-	set up a file control block
	S	stop	43	stop a program and set return code
		strtonum	32	convert a string to a decimal number
C		tail	46	returns the CP/M command tail as a string
A	F	time	46	return the time
	S	unpackstring	47	expand a string into a vector
	S	unrdch	32	put back the last read character
	F	uppercase	44	same as capitalch
		version	44	return the operating system and computer
	S	wrch	27	write a character
		writech	39	write a character to the output file
	S	writed	27	write a decimal number
	S	writef	28	formatted write
	S	writehex	29	write a hexadecimal number
	S	writen	29	write a decimal number in minimum width
	S	writeoct	29	write an octal number
	S	writes	29	write a string
C		writesector	-	write a 128 byte sector to disc
	F	writet	30	write a string and pad with spaces
	F	writeu	30	write an unsigned decimal number
		wrvdu	39	write character to the screen

A3. PORTABILITY

Most of the above I/O routines will be found in virtually all versions of BCPL. Some are less standard, and some are specific to Arnor BCPL. If a program may be required to run under other versions of BCPL these less standard routines should be used as little as possible.

Routines that are part of the standard BCPL I/O library (those marked 'd' are used in a slightly different way in Arnor BCPL):

aptovec, endread, endwrite, findinput(d), findoutput(d), getbyte, input, level(d), longjump(d), newline, newpage, output, packstring, putbyte, rdch, readn, reads, rewind, selectinput, selectoutput, stop, unpackstring, unrdch, wrch, writed, writef, writehex, writen, writeoct, writes.

Routines that are not entirely standard (though many BCPL systems have something similar):

call, capitalch, compch, compstring, delete, random, rename, time, uppercase, writet, writeu.

Routines specific to Arnor BCPL:

bdosa, bdosb, bios, bytemove, closein, closeout, debug, decval, external, firmware, flush, prbusy, prch, rdtest, rdvdu, readch, readsector, rewindin, setfcb, strtonum, tail, version, writech, writesector, wrvdu, plus all the routines in the file "AMSDOS", which are also CPC machine specific as well.

A4. ERROR MESSAGES

There are three main sets of error messages which can occur. These are fatal errors which cause the compiler to abort immediately, phase 1 errors which will stop compilation after phase 1 and finally phase 2 errors which will stop compilation at the end of the first run of phase 2.

Most error messages give a line number, for example "Expecting command near line 27". The word "near" is used because the compiler gives the line number at which it discovered the problem, and this is not necessarily the same line that needs correcting. The line number refers to the position of the line within the file, and not to the total number of lines compiled so far.

(i) Fatal errors

Out of memory

The compiler has run out of room for the program or its symbol table. Can sometimes be solved by re-using local variable names.

Out of symbol space

Very similar to the above except that the program may not be too big. Try compiling it to disc which allows twice as much symbol space.

If you are compiling a piece of text in memory from PROTEXT or MAXAM then saving the text to disc, clearing the memory and then re-compiling may solve this problem.

I/O error

An error has occurred when writing to the output file. May mean that the disc is full.

Can't open input file

The filename given in a GET directive could not be found on the disc.

Can't open output file

The compiler was trying to open a file to output the compiled code to, but for some reason was not able to open it.

Broken in

ESC or STOP was pressed twice in a row. Pressing it once will halt the compiler and cause it to wait for another key press. Pressing escape again causes this error while any other key causes compilation to continue.

Undefined START

All programs must have a procedure called START. The program submitted did not have this procedure.

Bad GET

Due to the limitations of AMSDOS only one file can be opened for input at any one time. A file was opened which contained a GET directive. Will also occur if a piece of stored PROTEXT/MAXAM text attempts to GET itself.

Non-ASCII file

The compiler was instructed to open a file but that file did not contain ASCII text which is required for the compiler. Users of CPC versions of PROTEXT should use program mode, which will save the file in ASCII format.

File intact

This occurs on CP/M versions when the destination file existed but the user typed N when asked if the file should be deleted.

Bad SOURCE name

CP/M versions check the source name given, and if it is invalid in some way e.g. has an extension of more than 3 letters then this error will be given.

Bad DESTINATION name

As above.

(ii) Syntax errors

Phase 1 errors are all syntax errors within the program. Often the cause will be a missing $) or) which will probably cause what may at first seem to be an odd error message. Syntax errors are reported as occurring near a certain line. This is the line within a particular file. If the compiler listing is switched off when the error occurs then a buffer, holding the last few lines of text read from the source code, will be displayed.

Bad expression

The expression being examined was faulty in some way. Maybe a command name was used as an identifier.

Bad vector definition

A vector definition was bad in some way. In particular only one vector may be declared in any LET command (although simultaneous definitions are allowed).

Bad procedure definition

A procedure definition was badly made. May have been some other form of definition which went wrong.

Bad FOR loop

The FOR loop was badly formed in some way.

need : or = in constant definition

Constant, Global and Static definitions require both : and = between the identifier and the constant expression. One or another is missing.

Missing $)

A $) was expected but not found. Check the number of opening and closing section brackets in the program.

Missing $(

A compound statement was needed but the opening $(was not found. In particular all SWITCHON statements must be compound.

Missing)

A closing bracket in an expression or procedure call/definition was missing.

Section tag mismatch

A section bracket was tagged but the tag did not have a matching opening section bracket. May be caused by omitting the space after a section bracket.

Expecting command

A command was expected but one was not found. This can be caused by, for example, omitting the colon in an assignment command, or by a declaration appearing after a command.

Expecting number

A number was expected but not found. Usually following a #.

Expecting ,

The compiler was expecting a comma (,) but did not find one. Occurs in a conditional expression.

ELSE expected

The ELSE part in a TEST statement is not optional and was not present in this case.

INTO expected

The control expression of a SWITCHON command must be followed by the keyword INTO. In this case it was not.

: is expected

A colon is expected after a case label or DEFAULT but the compiler did not find one.

Expecting identifier

The compiler was expecting an identifier e.g. after LET but did not find one.

String too long

BCPL constant strings can only be 255 characters long.

Bad character

The character read was illegal in the present context. e.g. it could have been a { instead of /*

Bad character constant

A character constant was badly formed, usually caused by a carriage return following an asterisk.

Bad option line

The options in an option command must be separated by commas and must end with a semi-colon or at the end of the line.

? is only allowed in definitions

Self-explanatory. The only undefined values allowed are the initialisation values of simple variables in a LET command. ? anywhere else is illegal.

(iii) Semantic Errors

The rest of the errors are semantic errors which occur during phase two of the compiler. These are usually due to the use of an invalid identifier name or a badly constructed loop.

Expression/id mismatch

The number of left and right hand sides in a multiple assignment or declaration was not equal.

Invalid assignment

You have attempted to assign to something which is not assignable. e.g. a constant or a procedure.

RESULTIS outside a VALOF

A RESULTIS command was found outside a VALOF block.

LOOP found outside loop

A LOOP command is only legal inside one of the looping constructs.

BREAK found outside a loop

A BREAK command is only legal inside one of the looping constructs.

RETURN found outside PROC/FUNC

A RETURN command is only legal within a procedure or function body.

ENDCASE outside a SWITCHON

An ENDCASE statement was found outside the body of a SWITCHON command.

VALOF must contain RESULTIS

A VALOF expression must contain at least one RESULTIS command.

Invalid application of LV

The LV or @ operator was used in an incorrect context i.e. on a constant. LV is 'left value' and means the evaluation of an expression to obtain an address.

Dynamic free variable used

A variable was used in an embedded procedure which had been declared in an outer procedure.

CASE/DEFAULT found outside SWITCHON

CASE or DEFAULT statements are only valid inside a SWITCHON statement.

Undefined identifier

A variable was used which had not been defined or was not in scope when it was used.

Labels need a global

A label was defined when in the scope of a local variable of the same name. As the variable will be initialised to the value of the label, it must be a GLOBAL or STATIC. If no variable of the same name is in scope then one will be created.

Need constant expression

A constant expression was required but not found. Examples of constants being required are the INLINE statement, declarations and CASE labels.

(iv) Internal compile errors

There is a fourth type of error but you should never see these. These are internal compiler errors and indicate possible bugs in the compiler.
Please report any incidence of one of these errors to Arnor.

Error in translation node
Unexpected node in expression
Declaration error

This manual was brought to you by **ROBCFG**

Typesetting by Whitgift Business Services Ltd. The Studio, Ledbury Place. Croydon. 01-680-7997