BCPL

For the BBC Micro

Lardo Boffin

Chapter 1 - BCPL Overview

This document shows you how to get started with BCPL on the BBC Micro. It assumes you know what BCPL is (Basic Combined Programming Language – a precursor to B and then C) and how to actually program in a language such as BASIC.

What you need to get going

- A BBC Micro, model B or better (or emulator such as BeebEm)
- A fast modern storage device such as an MMC card (SD card reader), Gotek, Econet or CF Card emulating a HDD in ADFS
- Either Sideways RAM (16K), a modern EEPROM (e.g. a Boobip EEPROM chip installed) or a spare ROM slot in which to put an EPROM for the BCPL ROM
- Ideally a 6502 co-processor of some sort (e.g. a Pi based co-proc) otherwise you will have very little RAM for programs!

In order to learn the language:

- A - read the book “BCPL the language and its compiler” by Martin Richards and Colin Whitby-Stevens
- B - play with it

There are only about 160 pages so there is not a huge amount of reading

Reading this guide will hopefully help you actually use it in practice on the BBC Micro. You can download some “.SSD” files from GitHub (LardoBoffin/BBC-BCPL_Overview/MasterDiscs) with the discs setup ready for use as described in BCPL User Guide – this includes BCPL_Source_Disc.ssd and BCPL_Test_Disc.ssd.

This document assumes your storage device either has the capability to use two ‘disc drives’ under DFS or be able to use ADFS and hold enough files in a single directly to have all necessary files present in one place. If you only have access to a single DFS drive expect to do a lot of disc swapping!

We will be assuming from here on in that you are using two drives in DFS, e.g. with an MMC card reader.

Initially full instruction will be given as to when to switch drives in order to go from compiling (source disc) to running (test disc) etc. but after a while this should be second nature. If you are running ADFS and have put the files from both discs into a single folder (or downloaded the ADF image and saved yourself the hassle) then there is simply no need to type * DRIVE ## from time to time to switch drives as all the files will already be present.

In the folder MasterDiscs there are:

Folder	File	Notes
ADFS	BCPL_Master_Discs.adl	All three DFS disc images but in a single image with sub directories
DFS	BCPL_Master_Disc.ssd	All of the standard BCPL files in a single image. There are 31 files so not much use in DFS!
	BCPL_Source_Disc.ssd	The 16 files recommended for building BCPL programs
	BCPL_Test_Disc.ssd	The 12 files recommended for running and testing BCPL programs
	BCPL_Calculations_Package.ssd	The calculations package files
	BCPL_Standalone_Generator.ssd	The runtime builder and files. This also includes NEEDCIN and EX
ROM	BCPL 7.0.rom	The BCPL ROM image file

Hello, World

The first thing you will want to do is create a hello world program. Apparently the first ever “Hello, World” program was written in BCPL (if the internet is to be believed).

Starting BCPL

To start off with we need to boot up BCPL. To do this from BBC BASIC type *BCPL. If you have the BCPL ROM as the highest priority ROM, then it will boot up in BCPL. The prompt will now be:

```
!
```

and not the usual “>”.

What can I type now?

You can still use all * commands (e.g. *DIR, *DRIVE #, *FX, *BASIC etc.) but not BASIC commands such as HIMEM or MODE 128 (BCPL only does modes 0 to 7) but you can type MODE 0. The built in commands, i.e. those that can be called from the command prompt, are:

Command	Function
CONT	Continue program (after pressing Escape)
COPY	Copy a file
DELETE	Delete a file
END	End a command file
ERRCONT	Continue the command file if there is an error

INIT	Initialise a program for testing
LINK	Link a file into the global vector
LOAD	Format a CINTCODE file for linking
MODE	Change display mode (0 to 7), typing MODE 128 will set the mode to 1
PAUSE	Suspect command file
PROTECT	Hold file in store
READ	Read a file into store
REM or //	Comment
RENAME	Rename a file
SAVE	Save a file
SHUFFLE	Maximise contiguous free store (compact the store)
STORE	Show the store files
TIDY	Free up store
TYPE	Display text file
UNLINK	Unlink file from global vector

Note that in a program the mode is changed by typing MODE (0) not MODE 0.

Say hello to ED

In order to create your first program, you will need a text editor to create a source file – this will be assumed to be ED, the editor that comes with BCPL, so load up the BCPL Source SSD disc image.

To create a new, blank file type (don't type in the "!", this is the command prompt!!!):

```
!ED HelloW (press return)
```

If a file of the name already exists, then it will open the file instead. Otherwise you will get a blank screen other than the words "New File" at the bottom of the screen.

Type the following:

```
GET "LIBHDR"
LET start() BE
        WRITES("Hello, World*N")
```

Don't worry about upper and lower case – this is not C.

Press Ctrl+F8 to save and exit the editor. This will save the file to the STORE, not the filling system.

The STORE and saving files

The program will be saved in the STORE; this is an area in RAM set aside for work in progress files. You will not see the file in DFS if you type *. at this point. If you type:

```
!STORE
```

you will see the files currently held in the STORE in RAM. More on this later. Much later. You can compile from the STORE and not save to disc but if your beeb crashes you have lost everything, so best to save to disc – type:

```
!SAVE HelloW
```

Don't put double quotes around the file name. If you use a different filename at this stage (e.g. just type !SAVE Hello) BCPL will get confused as it will not be able to find the file Hello in the STORE in order to save it to disc. You will get Error 17.

Building the program

Having saved to disc type:

```
!BCPL HelloW HW

You will see:
BCPL - RCP V2.2
Text read
RCP CINTCODE generation
CINTCODE size = 20 words
!
```

This means the file has been successfully compiled and a CINT code (the 'compiled' code that can be run directly under BCPL) file produced (and currently only saved in the STORE). Any other message means either the wrong filename was typed, the file HW already exists or the text was typed incorrectly into ED. Check everything and try again!

If you typed the message a little differently the number of words may be different.

The line BCPL HelloW HW has three elements as shown in the table:

Section	Notes
BCPL	Calls the BCPL compiler
HelloW	The source file to be compiled
HW	The name of the target CINt code file, you choose this

Saving the CINT code file

Type:

```
!SAVE HW
```

This will copy the CINT code file from the STORE to the filling system. Again you do not have to do this but if there is a crash you have lost the compiled file.

Running the program

To run the program simply type:

```
!HW
```

You will see:

```
Hello, World
!
```

Your first BCPL program!

Hello, World with green screen coolness!

The first Hello, World program just uses the built in standard BCPL function WRITES(“”) to produce the output. It is possible to extend this and change the screen colour and MODE etc. to make it look prettier. This can be done by using the VDU command that has kindly been added to this version of BCPL. To do that however we need to tell the compiler that we are going to use it and where to find it. According to the user guide it can be found in the Section “VDU” of the library file LIB. So…

Go back to the previous hello world source SSD file and type:

```
!ED HelloW
```

Amend the program as follows:

```
NEEDS “VDU”
GET “LIBHDR”
LET start() BE
$(
        VDU(“22,0”)
        VDU(“19,1,2”)
        VDU(“17,1”)
        VDU(“12”)
        WRITES(“Hello, World*N”)
$)
```

Note that we are now putting $(and $) around the start function – this is because there is more than one line of code.

Then: -

- Press Ctrl+F8
- Type !SAVE HelloW
- Type !BCPL HelloW HW

Switching to the Test Drive

We were bound to need it sooner or later.

Type !*Drive 1 (to switch to the test drive SSD)
Type `!SAVE HW`

I feel the need, the need for NEEDCIN

Type `!NEEDCIN HW LIB HWEXE`

You should see something like:

```
222 word file HWEXE created
0 NEED(S) unsatisfied
```

Type !SAVE HWEXE
Type `!HWEXE` and watch the fun! Wow, such mode 0 green.

The line NEEDCIN HW LIB HWEXE has four elements as shown in the table:

Section	Notes
NEEDCIN	This command extracts the NEEDed CINt code from the library
HW	The is the BCPL cint code created by the BCPL ## ## command from earlier
LIB	This is the library file where the SECTION "VDU" is to be found
HWEXE	This is the name of the final file with the original BCPL cint code plus the library code. You choose what this is

From here on in I won't be prefixing anything with !, so saying type NEEDCIN #### is the same as saying type !NEEDCIN ####. The BCPL prompt should be familiar by now. ☺

As an exercise you can replace the VDU("22,0") – which I'm sure you know changes the screen mode – to use the command MODE(0) that has also been included in the Beeb BCPL version!

Hello, World with green screen coolness and no BCPL ROM required!

The second program introduces sections of the provided library, LIB, and NEEDCIN to combine your program with the library code. This program goes a little further and removes the need for the BCPL ROM to be present in order to run the program – it will be compiled as a run-time version that can be called from BASIC! Note that the SAG disc has had the files EX and NEEDCIN added to it to make life easier.

Go back to the previous hello world source SSD file and type:

`ED TESTRT`

Type in the following program into the new empty file:

```
//
//Filename TESTRT
//
//29/05/2016
//
//v1.0

NEEDS "VDU"
NEEDS "INTERP"
```

```
NEEDS "RUNPROG"

GET "LIBHDR"

LET START() BE
$(
   MODE(0)
   VDU("19,1,2")
   VDU("17,1")
   WRITES("*N")
   WRITES("Hello world from BCPL Runtime!")
   WRITES("*N")
   RUNPROG("**BASIC")
   STOP(0)
$)
```

Don't forget to Ctrl+F8 to exit and then SAVE TESTRT.

The key thing to note with this program is the additional NEEDS sections (see points 3, 4 and 7 below) at the start and the RUNPROG (see point 6 below) at the end. The NEEDS listing tells the compiler which sections of the runtime library to include when it is being built. The RUNPROG although not strictly necessary gracefully drops the user back into BASIC after running the program.

Standalone Generation Process

1. Create a very simple BCPL program and test it

2. Make any required changes for the SAG. This is basically removing any function calls that are not supported

3. List all of the library routines used in your program and note the section they belong to (see chapter 5 for a list) and add each one as a NEEDS "..."

4. Add NEEDS "INTERP" at the end of the list

5. Look at the IO options required (none for my simple program)

6. Decide on termination options. I went for a very simple return to basic by calling *BASIC at the end. It may well be fine to do nothing and force the user to do a CTRL+Break to exit, i.e. do nothing

7. Add all of the NEEDS either to main program or one of the files in it, or ideally a separate file (I added them to the main program file but will do it properly later on)

8. Compile the BCPL programs as normal but with the switch NONAMES as this reduces file size slightly, in my case "BCPL TESTRT RT NONAMES". Note that you do not do the normal NEEDCIN at this stage, unless you are including a library you have written yourself. All standard library stuff, e.g. VDU will be picked up from the run time library later on

9. Decide on whether to use SYSLIB1 or 2. This mainly comes down to file IO I believe. We are not using any so will stick with SYSLIB1

10. Time for NEEDCIN! Ideally use EX to run a commmand file (I have used this) to do all the heavy lifting of files

11. Use PACKCIN to remove unwanted stuff and reduce the file size a bit

12. Ignore the ROM stuff for now

13. Decide on the location of the global vector (effectively the load location of the program which needs to be at PAGE or above). I have stuck with &1900 assuming a DFS of some sort

14. Use FIXCIN to create the program (ignore ROMs for now)

15. Use FILETRN to copy the file to keep its execution address (I think I used *COPY which seems to work as well)

Say hello to EX (brother of ED?)

All of stages 9 through to 14 are managed by a single command file used in EX (a program that runs command files such as this). I took mine directly from the manual and it works well in this simple example: -

```
.KEY INFILE/A,OUTFILE/A
NEEDCIN <INFILE> SYSLIB1 $TEMP1
PACKCIN FROM $TEMP1 TO $TEMP2
DELETE $TEMP1
FIXCIN FROM $TEMP2 TO <OUTFILE> GV=1900
DELETE $TEMP2
```

To create the file above type:

`ED COMPILE` (this will create a blank file, then type in the above, carefully, Ctrl+F8 to exit, switch to drive 1 and then SAVE COMPILE)

The .KEY line handles the file names. The NEEDCIN line handles points 9 and 10. The PACKCIN line handles point 11. The DELETE lines tidies up an unwanted temp file. The FIXCIN line handles points 13 and 14.

So assuming that our original BCPL compiled file is called RT (and the EX command file is called COMPILE) I would call -

```
EX COMPILE RT HELLO
```

Building and running the Standalone program

The complete sequence to build and deploy this file is therefore: -

- Load BCPL_Source_Disc into drive 0
- Load BCPL_Standalone_Generator into drive 1 (we don't need BCPL_Test for this project)
- On drive 0 type "BCPL TESTRT RT NONAMES"
- Switch to drive 1 and "SAVE RT"
- On drive 1 type "EX COMPILE RT HELLO"

After this has finished running you end up with a file called HELLO that can be run outside of BCPL by typing *HELLO

In terms of file sizes, the initial RT (compiled file) is &90 (144) bytes long and after being converted to runtime is &F82 (3970) bytes long. This is therefore &EF2 (3826) bytes of interpreter and function code. While this is quite a jump in size it is impressive given that it includes the BCPL interpreter! Don't be expecting to do anything useful in MODE 0 on a non-expanded machine though... Adding further function calls will increase the size of the end file as it will need to include more library functions.

When run the HELLO program should change the screen to MODE 0, turn the text green and say hello. It then calls RUNPROG("**BASIC") to terminate and return control to BASIC.

There are a whole raft of options and further settings but this is good enough for now for a very simple demo.

Chapter 2 – Header Files and Libraries

Header Files

So far we have been using simple one file example programs but any serious program will need to be better organised than that. You will need to split the program into multiple chunks if you wish to write a reasonably sized program and the best way to avoid duplicating the required header information, e.g. GET "LIBHDR", in each chunk is to have a header file. Put the GET directives in the header file and then put a GET “your header file” directive in each program file. This also avoids the situation where you add a new GET to several files but manage to miss one.
Also if you wish to share global data between multiple files you will need somewhere to hold GLOBAL variables – this is the header.

The header file therefore contains:

1. All required GET calls
2. GLOBAL variables

Load the BCPL Source SSD disc image into drive 0 and the BCPL Test SSD image into 1.

A Header File

Select drive 0 and (assuming you are in BCPL) type:

```
ED CPYHDR
```

In the blank file type:

```
GET “LIBHDR”
GET “SYSHDR”

MANIFEST $(
UG=FIRSTFREEGLOBAL
$)
  GLOBAL $(
  SFILE:UG
  TFILE:UG+1
  CH:UG+2
$)
```

Ctrl+F8 and then SAVE CPYHDR to drive 0.

A Program using the Header File

Still on drive 0 type:

```
ED COPYFL
```

In the blank file type:

```
GET “CPYHDR”     //we only need the header file reference

LET START() BE
$(
        outputFile()
        findSourceFile()
        getBytes()
$)

AND getBytes BE()
$(
        RCH() REPEATUNTIL CH=ENDSTREAMCH
        ENDREAD()
        ENDWRITE()
        RUNPROG(“**INFO EXMP1B”)
        RUNPROG(“**INFO COPIED”)
        STOP(0)
$)

AND RCH() BE
$(
        CH:=RDBIN()
        SWITCHON CH INTO
        $(
                CASE ENDSTREAMCH:
                ENDCASE
                DEFAULT:
                        WRBIN(CH)
                ENDCASE
        $)
$)

AND findSourceFile() BE
$(
        SFILE:=FINDINPUT(“EXMP1B”)
        SELECTINPUT(SFILE)
$)

AND outputFile() BE
$(
        TFILE:=FINDOUTPUT(“/F.COPIED”)
        SELECTOUTPUT(TFILE)
$)
```

Ctrl+F8 and then SAVE COPYFL to drive 0.

To run the program type BCPL COPYFL CODE to build, SAVE CODE to save it to drive 0 and then type CODE to run it. It should show two *INFO reports once done and the copied file will match the original. It takes over 20 seconds to run on BeebEm with a co-proc present!
The copy method used is far from efficient as it gets and then saves a single byte at a time. Also given that we only have a single program file the global variables are not required and could be defined locally, however it illustrates the points required – any program using GET “CPYHDR” will have access to the global variables.

Libraries

The Beeb, even running with a co-processor, has very limited capacity for running BCPL. There is not a lot of memory and it is fairly slow to compile / build, almost certainly as much from I/O as processor speed, although a superfast Pi co-proc and SSD card should improve the situation. So the best way forward is to split the program into chunks and use libraries for commonly used code.

Libraries have two main advantages for performance (aside the obvious one for consistency and code-reuse):

1. They are already complied and just need to be included
2. They do not need to be held in RAM as part of the main program

Anatomy of a Library file

There are two files required, a header and a code file.

HEADER FILE

The header creates global variables to hold the function names, in this example we have: -

```
//STRHDR

manifest $( ug = firstfreeglobal $)

GLOBAL $(
  LENGTH:ug
  CONCATSIZE:ug+1
  CONCAT:ug+2
  LEFT:ug+3
$)
```

Each function that can be called from the library has a name (that matches exactly) set here. I use the 'firstfreeglobal' constant to avoid having to keep track of them.

CODE FILE

The code file contains one or more sections which have code which will be compiled to the library file. In this example we have: -

```
SECTION "STRINGS"

//Various string functions
//
//27/03/2016 PJ
//
//V1.00
// length(string) - returns a length in bytes
// concatSize(string1,string2) - returns the required vector size
// concat(string1,string2) - joins two strings and returns a vector
```

```
// left(string,length) - returns 'length' number of characters

GET "LIBHDR"
GET "SYSHDR"
GET "STRHDR"

LET LENGTH(S) = VALOF
$(
   RESULTIS S%0 //size of string in bytes is held in byte 0
$)

LET CONCATSIZE(S,T) = VALOF
$(
   //takes two strings and returns the length divided by 2
   //this is the size of the vector to hold the results
   RESULTIS (S%0 + T%0)/2
$)

LET CONCAT(S,T) = VALOF
$(
   let size, vector, ScharCount, TcharCount = 0,0,0,0

   size := concatSize(S,T)*2  //get the size of the total string (this is a VEC
size so *2)
   ScharCount := S%0
   TcharCount := T%0

   vector := GETVEC(size)

   //loop through the first string and add it to the new vctor

   FOR N = 0 TO ScharCount DO
      vector%N := S%N

   FOR N = ScharCount+1 TO Size DO
      vector%(N) :=T%(N-ScharCount)

   vector%0 := Size

   RESULTIS vector

$)

LET LEFT(S,L) = VALOF
$(
   //return the left ## number of characters
   //if ## is greater than string size return the string as is

   let size, vector = 0,0

   size := S%0

   IF size < L THEN L :=size

   vector := GETVEC(L/2)

   FOR N = 0 TO L DO
      vector%N := S%N
```

```
   vector%0 :=L

   RESULTIS vector

$)
.
```

Things to note:

- The first line is a 'SECTION' and there is a full stop at the end of the section. Each section must have a full stop (unless the section ends at the end of the file but best to use one anyway)
- The name in the SECTION is the name to be used in the NEEDS directive in the file that will be calling the library
- Note there is no START() function as this is a library not a program
- You must include the header file in the GET section (GET "STRHDR") for the function names to be recongised in the library file. This call must also be placed in whichever program wants to use the library.

Compiling the Library File

To compile the library file it is a simple case of the usual: -

```
BCPL stings LIBStr
*drive 1
SAVE LIBStr
```

Note the inclusion of the section name in the compiler output.

Once compiled the file can be used as a library in the normal way.

Hello World with string counting from the library

To do this we will create a sample program that uses the library called "STRTEST".

This file will require a NEEDS "strings" at the start and includes GET "STRHDR". As mentioned above in order for this file to be compiled the global variables holding the locations of the library functions need to be present.

Sample code to call the library: -

```
NEEDS "strings"
GET "LIBHDR"
GET "STRHDR"

LET START() BE
$(
   LET l = 0
   LET s,t = "Hello there," , " World!*N"
   LET target = GETVEC(concatSize(s,t))
```

```
   target := concat(s,t)

   WRITES(target)

   l := LENGTH ("string test take 2")
   WRITEN(l)

$)
```

Ideally this program would have its own header file which would include the GET functions.

In order to use this file: -

```
BCPL strtest tmp
*drive 1
SAVE tmp
NEEDCIN tmp LIBSTR exe
*destroy tmp
```

Then type "exe" to run the program. It will show some text and then print out the length of the text "string test take 2".

Chapter 3 - Strings (Vectors) and more for the Library!

Strings and vectors

A string is basically a vector, which is a one dimensional array of BCPL words. Note that a BCPL word on the Beeb is 2 bytes.

The string 'Hello World!' contains 12 characters which would require 6 words in a vector. In a string vector the first byte of the first word holds the length of the string and therefore the length can be accessed by a call such as vector%0 where vector is the name of the string.

The LEFT function using vectors

We will write a program “vecshow” which will list the contents of a vector a byte at a time to illustrate this. But before we get to that, the string function 'left' from the previous library example illustrates a lot of points about using vectors as strings: -

```
LET LEFT(S,L) = VALOF
$(
 //return the left ## number of characters
 //if ## is greater than string size return the string as is

 LET size, vector = 0,0          *1

 size := S%0                     *2

 IF size < L THEN L :=size       *3

 vector := GETVEC(L/2)           *4

 FOR N = 0 TO L DO
  vector%N := S%N                *5

  vector%0 :=L                   *6

 RESULTIS vector                 *7

$)
```

1 - this defines the variables we will use later

2 - this extracts the size of the string (S) passed into the function as the 0th byte of the string holds the length in bytes

3 - this is a check to make sure that we don't try to take more characters from the left of the string than are present

4 - calls the GETVEC command to convert the variable 'vector' into an actual vector. Note that the vector size is the numebr of bytes divided by 2 as there are two bytes per BCPL word and there is no point wasting RAM

5 - set the byte of new vector equal to the byte of the original vector (string S). This looks through L (number of bytes to copy) times

6 - set the length of the new vector to L so that if we print the string out or work with it we know where it ends

7 - return the resulting vector to the caller

A sample vector program

So, create a new blank file called VECSHOW and type the following:

```
// Display contents of string vector example
//
// Filename vecshow
//
// Created 19/05/2016
//
// V1.00
//
// Steps through a vector (string) and shows the contents
//
// Uses VDU so will require NEEDCIN
//

NEEDS "VDU"
GET "LIBHDR"

LET START() BE
$(
   LET stringSample = "Hello world!"
   LET vector = GETVEC(6)
   vector:= stringSample

   MODE(0)

   SHOWVEC(vector)

$)

AND SHOWVEC(vector) BE
$(
   VDU ("31,2,2")

   FOR N = 0 TO (vector%0)+1  DO
   $(
      VDU ("31,%,%",2,(N+4))

      WRITEN(vector %N)

      VDU ("31,%,%",6,(N+4))

      IF N=0 DO
         $(
            WRITES("Length of string")
         $)
```

```
      IF vector %N >31 & vector %0 <127 & N>0 DO
         $(
            VDU ("%",vector %N)
         $)

      IF vector %N = 0 DO
         $(
            WRITES("Terminator")
         $)
   $)

   WRITES("*N")

$)
```

To compile and run type:

```
BCPL vecshow veccode
*drive 1
save veccode
NEEDCIN veccode lib vecexe
*destroy veccode
```

Finally run it by typing VECEXE. You will see various bits of information typed out onto the screen including the length of the string, the ASCII values, the individual letters and finally the 0 terminator.

More sample Vector manipulation

The previous library example can be expanded to include some additional functions: -

Header file

```
//Header for the Strings library
//

manifest $( ug = firstfreeglobal $)

GLOBAL $(
LENGTH:ug
CONCATSIZE:ug+1
CONCAT:ug+2
LEFT:ug+3
RIGHT:ug+4
MID:ug+5
INDEXOF:ug+6
$)
```

Note the extra global variables to hold the function names.

Library file

The extra sections in the library file are shown below (but don't type in the stuff in blue italics!)

```
SECTION "STRINGS"
```

```
// Various string functions
//
// 27/03/2016 PJ
//
// V1.01
// length(string) - returns a length in bytes
// concatSize(string1,string2) - returns the required vector size
// concat(string1,string2) - joins two strings and returns a vector
// left(string,length) - returns 'length' number of characters
// right(string,length)
// mid(string,start point,length) start point is 0 based
// indexOf(string,find string) returns zero based position

//let STARTINIT()=3000

//example usage: -
//let s,t = "Hello there,"," Worldy!"
//let target = GETVEC(concatSize(s,t))
//let leftTest = GETVEC(5)
//target := concat(s,t)
//writes(target)
//let leftTest = GETVEC(10)
//leftTest := left(target,30)
//writes(leftTest)

GET "LIBHDR"
GET "SYSHDR"
GET "STRHDR"
```

Length

```
LET length(S) = VALOF
$(
   RESULTIS S%0 //size of string in bytes is held in byte 0
$)

LET concatSize(S,T) = VALOF
$(
   //takes two strings and returns the length divided by 2
   //this is the size of the vector to hold the results
   RESULTIS (S%0 + T%0)/2
$)
```

Concat

```
AND concat(S,T) = VALOF
$(
   let size, vector, ScharCount, TcharCount = 0,0,0,0

   size := concatSize(S,T)*2  //get the size of the total string (this is a VEC
size so *2)
   ScharCount := S%0
   TcharCount := T%0

   vector := GETVEC(size)

   //loop through the first string and add it to the new vctor
```

```
   FOR N = 0 TO ScharCount DO
      vector%N := S%N

   FOR N = ScharCount+1 TO Size DO
      vector%(N) :=T%(N-ScharCount)

   vector%0 := Size

   RESULTIS vector

$)
```

Left

```
LET left(S,L) = VALOF
$(
   //return the left ## number of characters
   //if ## is greater than string size return the string as is

   let size, vector = 0,0

   size := S%0

   IF size < L THEN L :=size

   vector := GETVEC(L/2)

   FOR N = 0 TO L DO
      vector%N := S%N

   vector%0 :=L

   RESULTIS vector

$)
```

Right

```
LET right(S,R) = VALOF
$(
   //return the right ## number of characters
   let size, vector, startPoint = 0,0,0

   size := S%0

   IF R > size THEN R :=size

   startPoint := size - R

   vector := GETVEC(R/2)

   FOR N = startPoint TO size DO
      vector%(N-startPoint) :=S%N

   vector%0 :=R

   RESULTIS vector
```

```
$)
```

Mid

```
LET mid(S,SP,LN) = VALOF
$(
   //return the LN ## number of characters from S starting at SP (start point)
   //note that the starting point is zero based
   let size, vector = 0,0

   size := S%0

   IF SP >= size THEN
      $( SP :=size
         LN :=0
      $)

   IF (SP + LN) > size THEN
      LN := size - SP

   vector :=GETVEC(LN/2)

   FOR N = SP TO (SP+LN) DO
      vector%(N-SP) := S%N

   vector%0 := LN

   RESULTIS vector

$)
```

IndexOf

```
LET indexOf(S,FS) = VALOF
$(
   //returns the zero based index of the first instance of FS (find string)
   //in S (string) or -1 if not found

   let index, found = -1,0

   //step through the S string and test the first char of FS
   //if it matches try the next one
   //if not move on to the next char in S

   FOR N = 1 TO (S%0 - ((FS%0)-1)) DO
      $(
         //step through S
         found  := 0

         FOR C = 1 TO FS%0
            $(
               //step through FS until no match is found
               TEST FS%C = S%(N+(C-1)) THEN
                  found := found + 1
               ELSE
                  BREAK
            $)

            IF found = FS%0 //if the value of found equals the length of the find
string we have found everything
```

```
            $(
               index := N
               BREAK
            $)
      $)

   RESULTIS index

$)
.
```

Command line utilities and ARGS vector

A command line utility is a program that you run and pass in some parameters or arguments, e.g. a file copy or inspection program. The arguments are typed after the filename to run, such as "BCPL *sourcefile targetfile"* which compiles the source file and produces the target file.

The arguments are passed in to a vector with one element per argument.

So, create a new blank file called MODE and type the following:

```
GET "LIBHDR"

MANIFEST $(avsize = 5 $)

LET start() BE
$(
        LET s, md = 0 , 0
        LET argvec = GETVEC(avsize)

        IF RDARGS("M/A",argvec,avsize) = 0 THEN   *1
                STOP (11)                         *2

        md: = argvec!0                            *3
        md: = md%1                                *4
        md: = md - 48                             *5

        s: = MODE(md)                             *6

        IF s = 0 THEN                             *7
        $(
WRITES("Unable to change to mode ")
        WRITEN(md) $)

$)
```

1 – call the RDARGS function to get the vector of the typed in arguments

2 – check to see if any arguments have been passed in (0 means none have) and call STOP with the error message 11 if none are present

3 – the typed in mode should be in the first element of the arguments vector array

4 – the first element of a vector (vector%0) is its size so our number should be in element 1

5 – remove 48 from the ASCII character number to get an actual number (otherwise we will be passing the character “1” to the mode function which will fail)

6 – call the MODE function returning success or failure to “s”

7 – if s = 0 then the attempt to change mode failed (e.g. a number outside the 0 to 7 range) so report this

Compile this to a file called MD or similar (you can’t use MODE as there is a built in command that does this already) and run with and without arguments and also with obviously wrong arguments to see what happens.

Chapter 4 - Arrays (Vectors)

Under construction.

Appendices

ED (the fabulous Mr 209)

Ed is the provided text editor and it has a lot of functions! A brief overview is shown below.

To start and stop

- Type ED filename to either open (if it exists) or create a file of that name
- Type ED filename NEW to create the file and overwrite any existing file of that name
- If you want to save your changes press Ctrl+F8 and this will save the changes to STORE (bit not the filing system, you will need to type SAVE filename for this)
- Press F9 to just exit without saving – if you have made any changes you will be asked to confirm before exiting. If you say 'Yes' all is lost
- If you do not want to save your changes press Escape and this will take you out of ED. See Escape, below

Escape

Pressing Escape 'interrupts' ED.
Whatever is present in ED will be retained and you can type CONT to continue using ED and then press Ctrl+F0 to refresh the display. And carry on where you left off.
Useful if you need to look at the filing system or confirm anything before saving your changes.

Text Entry and function key controls

It's a pretty straight forward text editor, use the cursor keys to move around, tab to indent and DELETE to delete the character to the left

Function Key	Just the key	Plus Ctrl pressed
F0	Command Mode	Refresh display
F1	Delete character	Delete from current position to EOL
F2	Delete word	Delete current Line
F3	Move to Previous Page	Move to Top of file
F4	Move to Next Page	Move to Bottom of file
F5	Move to End of Page	New Line after current line to jump to it
F6	Move to End of Line	Join next line to this line
F7	Move to Previous Word	Undo current line changes
F8	Move to Next Word	Exit
F9	Quit	

F0 enters the Command Mode which allows you to type a bunch of commands to search and replace, format, move the cursor around and do block moves etc.