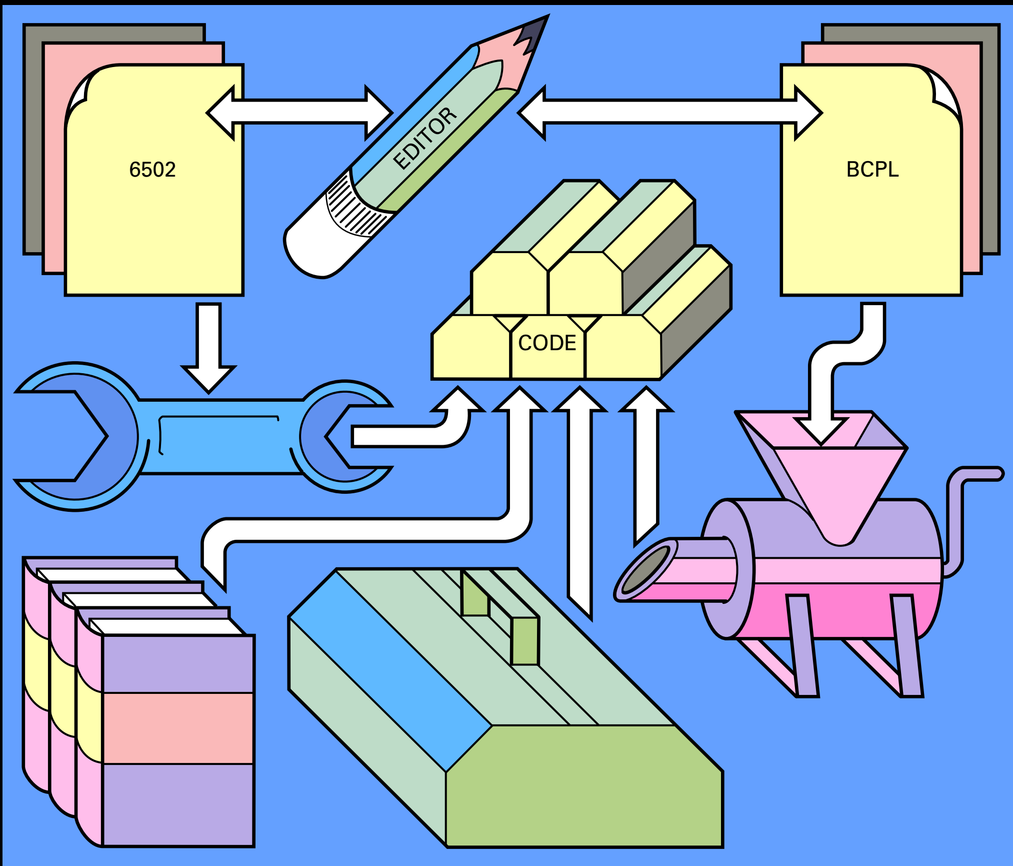
BCPL

for the BBC Microcomputer

User Guide

CHRIS JOBSON
and JOHN RICHARDS



User Guide

BCPL

for the BBC Microcomputer

CHRIS JOBSON
and JOHN RICHARDS

ACORNSOFT

Acknowledgement

We would like to thank Dr Tim King for his work in developing the editor and relocatable assembler.

Disclaimer

Richards Computer Products Ltd. and Acornsoft Ltd. reserve the right to make improvements in the product described in this book at any time and without notice. Every effort has been made to ensure the accuracy of the material presented in this book. Nevertheless, experience shows that some textual and software errors always remain to be discovered. If you find any errors, or have any suggestions on how to improve this book, please contact Richards Computer Products Limited, Brookside, Westbrook Street, Blewbury, Didcot, Oxfordshire OX11 9QA, England.

First published in 1983, by Acornsoft Limited

FIRST EDITION

Re-mastered by dv8 in 2020
First revision, November 2020

For the latest revision of this book go to:
stardot.org.uk/forums/viewforum.php?f=42

Published by:

Acornsoft Limited
4a Market Hill
Cambridge
CB2 3NJ
England

Contents

1 Introduction

This User Guide includes information needed by every user of the Richards Computer Products BCPL CINTCODE system on the BBC Microcomputer.

Overview

The BCPL CINTCODE system for the BBC Microcomputer consists of the BCPL Language ROM together with a comprehensive set of programs and utilities which provide an excellent environment for the development and execution of BCPL programs.

The BCPL CINTCODE system may be used on any BBC Model A or Model B Microcomputer, although a Model B is needed for program development. It is designed for use with any filing system and in particular is very convenient for use with a single disk drive. For program development the use of disks or the Econet is recommended, although large programs can be developed even on tape-based systems.

The BCPL Language ROM plugs into any of the ROM sockets in the BBC Microcomputer. It provides a convenient, flexible and powerful set of commands to manipulate files and execute BCPL programs. In particular it includes a store filing system which is compatible with all other filing systems. The ability to hold files in store greatly enhances the ease of use and the speed of the system.

The programs and utilities include a general-purpose full screen editor (which may be used for letter and document writing as well as for program entry), a BCPL compiler, a relocatable 6502 assembler and a number of program development and debugging tools. Indeed, the system may be used just for its convenient file handling and editing facilities.

BCPL is a high level programming language which is easy to learn and to read but which is extremely flexible and suited to a wide variety of applications. BCPL source is compiled into a very compact code called CINTCODE, which is then interpreted at run time. This gives CINTCODE many of the advantages of both traditional fully compiled languages and of interpreted languages such as BASIC. In particular CINTCODE is compact and executes fast, but is very easy to debug. For example, it is possible to interrupt a running program, use various utilities to examine and change the state of the system and then resume the program at the point where it was interrupted.

HOW TO USE THIS GUIDE

All readers who intend using the BCPL CINTCODE system should read chapter 2, 'Using the BCPL system'.

Readers who want an overview of the features of BCPL CINTCODE should read the rest of this introduction.

Readers who have just received a copy of the BCPL CINTCODE system and wish to make it operable, should turn to chapter 9, 'Getting started'. This chapter includes several examples which will be of use to readers who wish to become familiar with the facilities of BCPL CINTCODE and have access to an installed version.

Experienced users of BCPL who require a quick reference to the features of this implementation may use the summaries in chapter 11.

The main components of the system are:

- the BCPL environment provided by the BCPL Language ROM. This is described in chapter 2 and the built in commands provided are described in chapter 3.

- a full screen text editor for letter writing, program entry and so on. This is described in chapter 4.

- program development in the BCPL language. The language is described in chapter 10 while the utilities for program development are described in chapter 4. Chapter 8 discusses various topics associated with program development.

- the library of procedures for use by BCPL programs. Each procedure is fully specified in chapter 5.

- program development in relocatable 6502 assembler. The assembly language is described in chapter 10, and the assembler itself is described in chapter 4.

- the program development aids are described in chapter 7.

Readers who wish to understand how the features are provided should read chapter 6, 'Runtime features', and the Appendix.

FEATURES OF THE BCPL SYSTEM

The command state

You normally enter the BCPL system in the command state.

In this state you can enter the name of a program or command to be executed, followed by parameters giving details of the operation required. Most commands accept keywords to enable you to specify the parameters in a clear way and in any order. These keywords can be displayed as a reminder of the parameters required.

The command state is re-entered when a command or program finishes, fails or is interrupted. A program may fail through lack of store in which case it is sometimes possible to free more store using built in commands and then resume the program. A program which has been interrupted can also be resumed after the use of built in commands.

Command files may be created using the text editor and executed by the utility EX. EX may be given parameters which are substituted for keywords within the command file. This is a powerful facility allowing general-purpose command files to be created.

A procedure RUNPROG allows any command that can be entered at the console to be executed by a program. Thus programs can make use of the built-in commands and can even call other programs as subroutines.

The command state can be bypassed in dedicated applications. This permits other forms of control, for example by menu selection.

BCPL

BCPL is a well-established systems-programming language that is widely used on at least 25 different computer types. It is easy to learn, easy to read, and very flexible.

BCPL provides for a clear expression of the program logic by the use of structured constructions including FOR, IF, WHILE, UNTIL, REPEAT, REPEATWHILE, REPEATUNTIL, LOOP, BREAK, SWITCHON and CASE.

It also provides a standard method of linking separately compiled segments. This enables complex programs to be developed and tested in small easily managed sections.

By policy the BCPL language is kept simple. It is adapted to many different applications by the provision of a variety of procedures. A number of widely used procedures are specified in the language standard. This implementation provides all these standard procedures and a large selection of others, including a number to take advantage of the special features of the BBC Microcomputer.

CINTCODE

CINTCODE is a compact interpreted code which has been specially designed for implementing BCPL on 8-bit microcomputers and small 16-bit computers. Two major features of CINTCODE are its compactness and portability.

Compactness

A CINTCODE program requires about a third of the storage of fully compiled code. Because of this compactness more comprehensive store resident systems are possible, more code can be held in ROM or on disks, and the time to load code from tape or the Econet can be greatly reduced.

CINTCODE is significantly more compact than UCSD Pascal and Forth.

Comparison with assembler-written programs is difficult because of the wide range of possible applications and programming methods; however it is unlikely that an assembler program of any size would approach the compactness of CINTCODE.

Portability

BCPL is in itself a very portable language and CINTCODE is largely machine independent. It is usually easy to convert a BCPL source program to a new computer. It is often possible to write programs which merely need to be recompiled and can then be run without change in a different computer. In some cases it is possible to move compiled CINTCODE programs of this type from one type of computer to another.

This portability of compiled code naturally requires a CINTCODE system for each computer.

Linking and overlaying

CINTCODE sections link at runtime. This permits easy overlaying and is a great advantage when developing and testing complex systems.

Procedures are provided to load a file of CINTCODE into store and to link the loaded code into the rest of the running system. Complementary procedures can unlink the code, but the file is only deleted from store on request, or when the system requires more store than is available. Thus it is not necessary to reload an overlay or program if it can remain in store when not in use.

Code libraries

Libraries of commonly used application procedures can easily be built up. They can be used in two ways:

- the library can be held in store. As programs are run they automatically link to the procedures they need;

- the utility NEEDCIN can be used to extract the library procedures needed by a particular program and incorporate them into that program.

Input and output

The definition of BCPL includes a standard set of input and output procedures. Additionally this implementation provides a standardised method of identifying devices (such as the console or printer) and files.

This allows device-independent programs (i.e. ones which can take their input from, and write their output to, any device) to be written with ease. The actual devices to be used are specified by user-supplied parameters when the program is run. For example a program might generate a report which could be displayed on the console, printed on a printer or saved to a disk file.

As another example a program designed to process data from an input file and generate an output file might be tested using the console for both input and output (with no change to the program being necessary).

Store files

The provision of store files gives several major advantages:

- Several programs can be held in store at once and can be run when needed. For example, when developing machine code the editor, the assembler, the assembly language source and the object code can all be held in store together.

- Files can be read from disk, tape etc. into store in a single operation. They can then be accessed by programs from store. This can give great speed advantages since it is much quicker to read a file in one operation than it is to read it a byte at a time (typically 10 times quicker for a disk file and better than this for an Econet file if the Econet is heavily loaded). Similarly programs can write data files to store then copy them out to disk etc. when they are complete.

- Store files can be used for communication between programs.

Additional flexibility is provided by procedures which convert between files and data areas. Thus with one procedure call a program can read a file into an internal data area and similarly it can write the contents of an internal data area as a file.

Exceptional programming aids

The CINTCODE environment has been designed to permit symbolic debugging of the final code. The wide range of debugging aids are provided with limited use of precious store. The debugging features include:

- examination of store in decimal, octal, hex or ASCII;

- alteration of store;
- reports on the current stack organisation;
- displays of the use of store;
- breakpoints and the ability to rerun a program to a known point;
- symbolic traces;
- disassembly traces and displays of CINTCODE;
- the ability to collect statistics on program execution.

The development of programs in the CINTCODE environment is eased by automatic protection against stack overflow. This check has very little effect on run-time efficiency.

Multi-tasking

The implementation supports coroutines. Coroutines are independent but cooperating processes in the same machine environment. Each coroutine has its own stack. It receives control of the processor from another coroutine, and hands control back to this coroutine or to another coroutine when its program reaches an appropriate point.

One use of this feature is to support independent processing for different areas on the screen.

Calculations

BCPL does not aim to be an ideal language for scientific calculations, and is designed for integer arithmetic. Calculations in multiple word lengths and fixed point formats are aided by a procedure MULDIV which allows double length integer precision.

The nature of BCPL means that it is straightforward to write sets of procedures to implement calculations using any desired representation of numbers. A BCPL calculations package is available for use with the BCPL CINTCODE system, which implements both floating point and multiple length integer arithmetic. The package includes a range of trigonometrical and exponential functions.

Speed

Because CINTCODE instructions are powerful and because the instructions to execute have been chosen by an optimising compiler, the speed of execution is considerably greater than that of BASIC interpreters. CINTCODE is therefore suitable for most single user applications on micros. Indeed in many cases a CINTCODE implementation will run faster than a machine code system because less code has to be loaded from a slow device.

However, there will be cases in which the greater processing speed possible in assembler code is required. Often in these cases only a small part of a program requires the maximum speed, while the rest benefits from the compactness and easy development of BCPL CINTCODE.

Relocatable machine code

The BCPL CINTCODE system supports the development and use of 6502 code with many of the advantages of the CINTCODE environment. The machine code can be relocated when it is linked into the system, and so can occupy any available space, and separate units of machine code can be linked at runtime. Data in the machine code can be read or written from BCPL and the machine code procedures can be called from BCPL.

The CINTCODE system can also address and use machine code at a fixed address.

Stand-alone programs

Although BCPL CINTCODE is an obvious choice for the development of applications and systems software, it has the disadvantage that the CINTCODE programs produced can only be run on computers equipped with a BCPL Language ROM.

To overcome this problem a stand-alone program generator package can be supplied. This package allows programs developed using the BCPL CINTCODE system to be converted into stand-alone programs that can be run on any BBC Microcomputer. The stand-alone program may be in the form of a file which can be supplied on any suitable medium (e.g tape, disk, ROM) or it may be in the form of a language ROM.

The package works by linking the interpreter and an appropriate subset of the BCPL library procedures with the program to be distributed. Certain features of the BCPL CINTCODE system are not available to the program, in particular store files are not supported (although the ability to convert files to data areas is).

This package is aimed at professional software developers and the price includes a licence for the use of the interpreter and library routines.

COMPUTER REQUIREMENTS

Program development in BCPL or assembler requires a Model B BBC Microcomputer with the BCPL CINTCODE package, and a filing system of some kind. The tape filing system is adequate (preferably using a tape recorder with remote motor control) but development is faster and more convenient using a disk or an Econet.

The system supplied can be used without alteration on the optional second 6502 processor.

It is also possible to develop applications using a Z80 second processor and CP/M, but this requires a separate CP/M based program development package.

Suitable developed systems can be run on a Model A BBC Microcomputer with the BCPL Language ROM.

The BCPL Language ROM can be the main language ROM in the BBC Microcomputer. Thus three other ROM sockets are available on the main board and may be used for programs, data or other languages.

The system needs either version 1.0 of the operating system or a later version. If disks are to be used version 0.90 of the disk operating system or a later version must be used.

2 Using the BCPL System

This chapter is a brief guide to using the BCPL system once it has been installed. Many of the topics covered are explained in more detail elsewhere in this manual. The installation of the system is covered in chapter 9.

ENTERING AND LEAVING BCPL

If the BCPL ROM is installed as the language ROM nearest the edge of the main PCB then BCPL is automatically entered when the computer is powered on. If the BCPL ROM is not so installed (or if another language has been selected) then the command *BCPL enters BCPL.

When BCPL is entered it prompts for a command with the character '!'.

To leave BCPL for another language type '*' followed by the language name (e.g. *BASIC).

Typing *BCPL re-initialises the BCPL system. Pressing **BREAK** has the same effect.

WARNING: BCPL has no equivalent of the BASIC OLD command. Pressing **BREAK** completely re-initialises BCPL and all programs and files in store are lost.

COMMANDS

When the system starts up it enters the command state i.e. it waits for a command to be entered.

A command is entered by typing a line of text terminated by **RETURN**. The system executes the command then returns to the command state, displays a prompt ('!') and waits for a further command.

There are various types of commands:

- Built in commands. These are documented in chapter 3 and are functions provided in the BCPL ROM. Examples of built in commands are:

```
!STORE
!DELETE myfile
!SAVE file1 AS file2
```

- Operating system commands. These are functions provided by the BBC Microcomputer operating system/filing systems. They all start with an asterisk ('*'). Examples are:

```
!*FX 7,4
!*TAPE
!*CAT
```

- Programs. These may be provided with the BCPL system (such programs are called utilities - see chapter 4), purchased from software suppliers or written by the user. Programs are supplied as files (e.g. on disk or tape) and are loaded into store for execution by the system. Examples are:

```
!ED textfile
!BCPL bprog cprog
!JOIN file1 file2 AS file3
```

which respectively edit the file named 'textfile', compile the BCPL source file 'bprog' to the CINTCODE file 'cprog' and join the two files 'file1' and 'file2' to make a new file 'file3'.

When a command is entered the system uses the first item in the command to decide on the type of command (an item is typically a string of characters terminated by a space or the end of the command). All items in the command apart from the first are treated as arguments (or parameters) of the command and their interpretation is dependent on the command.

Commands may be entered in upper- or lower-case, or in a mixture of both. The difference between upper- and lower-case is not significant to any of the built-in commands or supplied programs. The difference may be significant in the parameters of purchased or user-written programs, however.

The convention used in this manual is that capitals are used for:

- BCPL keywords, such as LET, FOR and SWITCHON;
- system procedures and defined manifests such as WRITES, RDARGS and MAXINT;
- keywords in argument keys (see below).

Lower case is used for any arguments or variables which are introduced as examples, and for which any suitable name would be satisfactory.

The normal BBC Microcomputer screen editing facilities are available while entering commands (i.e. the four cursor control keys, **COPY**, **DELETE** and **CTRL-U**). Other control characters may be entered as part of the command line (e.g. **CTRL-B** to copy all screen output to the printer, **CTRL-N** to turn 'page mode' on) and have their normal effects.

WARNING: CTRL-V must not be used to change the display mode since it may crash the system. The correct way to change the display mode is to use the built in command MODE described in chapter 3.

ARGUMENTS TO COMMANDS

The arguments to operating system commands are as specified in the BBC Microcomputer User Guide and the various filing system User Guides. The arguments to built in commands and programs are usually processed by calling the procedure RDARGS (described in chapter 5). This procedure provides a standard method of handling arguments as described in the following paragraphs.

In most cases the arguments are specified in a particular order (e.g. in the command

!BCPL bprog cprog

the first argument is taken as the source file name and the second as the CINTCODE file name), but in some cases an argument is preceded by a keyword which identifies the argument.

Keywords permit the arguments to be entered in any order and make the command line easier to read. Normally when keywords are omitted arguments are assumed to be in order, but some arguments are only accepted if they follow the appropriate keyword.

It is possible to request a display of the arguments accepted by a utility by entering a question mark ('?') in place of the arguments. The system then responds with the definition of the arguments taken by the utility, and waits for the arguments to be entered.

For example the request:

```
!BCPL ?
```

would receive the response:

```
FROM/A,TO/A,REPORT/K,NONAMES/S,MAX/S
```

This indicates that BCPL will accept up to five arguments. The first two arguments are labelled /A which indicates that they are essential. They may optionally be preceded by the keywords FROM and TO respectively. The third argument, labelled /K, will be accepted only if identified by the appropriate keyword REPORT. The keywords NONAMES and MAX require no arguments, since they are labelled with /S.

Examples of valid arguments for BCPL are:

```
!BCPL myprog mycode
```

(It is not necessary to include the keywords FROM and TO. If FROM does not exist in the argument string, the first argument without a preceding keyword is used).

```
!BCPL TO mycode FROM=myprog MAX REPORT=myreport
```

(The keywords need not be entered in the order listed. A keyword requiring an argument such as FROM or REPORT may be separated from its argument either by a space or by an '=' sign).

A fuller description of the use of argument keys is given in the description of the procedure RDARGS in chapter 5.

COMMAND FILES

A command file is a file containing a number of commands which are to be executed as if they had been typed in at the console. There are two methods of executing command files.

Firstly the operating system *EXEC command may be used. This is transparent to the BCPL system (i.e. the BCPL system believes it is reading from the console). A limitation of this method is that each new sequence of commands requires a new command file.

The BCPL system provides a more flexible method of executing command files using the program EX (described in chapter 4). The command files processed by EX may contain keywords. When EX is run it may be supplied with a number of arguments which are substituted for the keywords. Thus one command file may be used for a number of different purposes by varying the arguments of EX.

RUN STATE AND COMMAND STATE

The BCPL system can execute CINTCODE in two different 'states' called the 'run state' and the 'command state'. The code being run in one state is independent of the code being run in the other. At any given moment one state is active and the other is 'suspended'. When the system changes from one state to the other execution of the code that was suspended resumes at the point it had reached when the previous change of state occurred.

The system is initially in the command state. Most built in commands, the debugging aids (described in chapter 7) and the program JOIN (described in chapter 4) run in the command state. Nearly all user-written programs run in the run state.

When a run-state program is active various events can cause the system to suspend it and return to the command state. When such an event has occurred the user may invoke built in commands and run any command-state program before resuming the suspended program by the built in command CONT. The user may also abandon the suspended program by the built in command TIDY.

Events causing program suspension

The following events cause the system to suspend execution of the current run-state program and return (or trap) to the command state:

- The user presses the **ESCAPE** key. The system displays 'Interrupted' before issuing the prompt for a command. The use of **ESCAPE** is discussed further in the next section.

- The program tries to obtain an area from the heap (see the section 'Store management' below) and fails because there is not enough free space. The system displays 'nnn+ store needed' before issuing the prompt for a command, where 'nnn' is the size of the area required. The user will typically free up more heap space by deleting unwanted store files before resuming the program.

- The program calls the library procedure TRAP. The system displays the message 'TRAP' followed by the parameters of the call before issuing the prompt for a command. The reason for the trap to the command state, and hence the required user action, will be specific to the program.

Methods are provided for programs to disable the return to the command state for the first two events mentioned. These methods are discussed in chapter 6.

USE OF ESCAPE

The system uses **ESCAPE** as a method of breaking out from the current activity and returning to the command state. The effect of **ESCAPE** depends on the current activity:

- If the system is executing an operating system command then the effect is dependent on the operating system. Typically **ESCAPE** is ignored by commands which execute quickly but terminates commands which may take a long time to complete (e.g. *CAT of a tape).

- If the system is executing a built in command then **ESCAPE** is ignored by commands which execute quickly but terminates commands which take longer (e.g. COPY). The message 'Error 1017 Escape' is displayed.

- If the system is executing a run-state program then **ESCAPE** normally causes a trap to the command state as described above (unless the program has disabled such trapping, in which case the effect depends on the program). If the program is performing input or output (I/O) when **ESCAPE** is pressed, however, then the normal effect may not occur. Instead the program might either continue running, treating the I/O operation as having failed, or it might ABORT (i.e. terminate prematurely).

DEVICES AND FILES

The BBC Microcomputer distinguishes between devices (e.g. the keyboard, the display, the serial port) and files. Devices are selected by *FX commands and the operating system routines OSRDCH/OSWRCH are used for the actual I/O. Files are accessed by a different set of operating system routines (OSFILE, OSBGET, OSBPUT etc.) which all apply to the current filing system. The current filing system is changed by commands like *DISK, *TAPE etc.

The BCPL system does not make the distinction between devices and files. Instead I/O is organised on the basis of streams, where a stream may correspond to either a device or a file. In addition to the standard BBC Microcomputer devices and the current filing system, the BCPL system provides a store filing system. This is described in more detail later in this chapter. The commands *DISK, *TAPE etc. are still used to change the current filing system.

When opening a stream for input or output the device or file to be used must be specified. A standard convention is used throughout the system for all device and file names.

File names

Any name that does not begin with the character '/' is treated as the name of a file, either in the store filing system or in the current filing system according to the following rules:

(1) If the file is to be written, deleted or renamed then the store filing system is used.

(2) If the file is to be read then if a file of that name exists in the store filing system then it is used, otherwise the current filing system is used.

Thus the following example of the built in command COPY (described in chapter 3):

```
!COPY abcde fghij
```

would look in store for a file named 'abcde'. If it found it, then it would copy it to the store file 'fghij'. If not it would try to copy the file 'abcde' from the current filing system to the store file 'fghij'.

It is possible to specify that a file name applies specifically to a store file or specifically to a current filing system file by prefixing the name with '/S.' for store or '/F.' for the current filing system. Thus

```
!COPY /S.abcde /F.fghij
```

would look for a store file named 'abcde' and copy it to a file named 'fghij' in the current filing system. If 'abcde' did not exist in store then COPY would give up - it would not look in the current filing system as the previous example did.

The prefixes just mentioned must also be used if a file name is required to start with '/'. Thus '/S./fred' refers to a store file named '/fred', and '/F./f./f.' refers to a current filing system file named '/f./f.'.

The only limit imposed on file names by the BCPL system is that they must not be longer than 255 characters. Different filing systems may impose their own restrictions.

Device names

All device names are of the form '/x' where 'x' is a letter.

The input devices supported are:

- /C the keyboard read a line at a time with echo to the display. The normal editing functions (cursor control keys, **COPY**, **DELETE**, **CTRL-U**) are available;
- /K the keyboard read a character at a time with no echo;
- /P the RS423 serial port.

If the screen editing facility is disabled (by *FX 4,1) then the **COPY** key may he used to give an ‘end of file’ character (ENDSTREAMCH) when reading from the keyboard using the procedure RDCH. Screen editing is restored by the BCPL system whenever a program terminates or the TIDY built in command is executed. It may also be restored explicitly by *FX 4,0.

The output devices supported are:

- /C the screen;
- /L the printer;
- /P the RS423 serial port.

One special device provided by the BCPL system is the null device (/N). It may be used for input (when it returns ‘end of file’ to all reads) or output (when all data written is discarded). It may also be used in file-oriented commands such as DELETE and RENAME.

An example of using the null device is:

```
!JOINCIN myfile AS /N
```

This uses the utility JOINCIN (described in chapter 4) to check that the file ‘myfile’ is a valid CINTCODE file. The output of the program is directed to the null device so that no output file is generated.

STORE FILES

A store filing system is provided so that files can be written to store and read from store in the same way as they are written to and read from the current filing system.

The built in commands READ and SAVE allow files to be copied efficiently between store and the current filing system.

There are many ways in which store files can be used to provide a faster and more convenient system. For example:

- one program can create a store file which is used as input by a subsequent program;

- several programs and data files can be held in store at once and used as needed, without having to reload each time. This is a particularly useful feature when working with a slow filing system such as tape;

- some filing systems (e.g. disk and Econet) are much faster at reading/writing an entire file in one go than at reading/writing it a byte at a time. When working with such filing systems input files can be copied into store and read a byte at a time from store. Similarly output files can be written to store and copied to the filing system when complete.

Store files are held physically as a number of blocks of data chained together. A file which has only one data block is called contiguous.

When a program is run the CINTCODE for that program is brought into store as a store file (unless it is already there). Two special terms are used in connection with CINTCODE files - loaded and linked.

A loaded file is one which is in the correct format for execution. Either it is contiguous or its block structure is such that each data block contains exactly one section of CINTCODE (the structure of CINTCODE is explained in chapter 10). CINTCODE files not in the correct format are automatically reformatted when they are run. The built in commands LOAD and LINK also reformat CINTCODE files if necessary.

When a program is run its CINTCODE is automatically linked in to the system so that it can be executed. This means that the entry points to the code are placed in a common area known as the global vector. (The global vector is also used to hold entry points to the BCPL library procedures and common data.) When the program finishes it is unlinked from the system (unless it has terminated abnormally). A CINTCODE file linked in to the global vector is called linked. The built in commands LINK and UNLINK may be used to link and unlink CINTCODE files. One consequence of a file being linked is that the SHUFFLE procedure (see below) cannot move the file in store.

Store files may be protected or unprotected. Unprotected files are deleted if the system needs more heap space (they are actually deleted by the SHUFFLE procedure). Normally all store files are created as protected. One exception is the file created when a program not already resident in store is run. Thus entering the command /F.HEAP would copy the CINTCODE file for the program HEAP from the current filing system to store (as a loaded file named 'HEAP') and execute it. The file would then remain in store but be unprotected. The built in command PROTECT allows the protection status of files to be changed.

STORE MANAGEMENT

All free random access memory (or RAM, i.e. the area between the top of the operating system workspace and the bottom of the display RAM) is organised by the system into a structure called the heap. All requirements for areas of RAM are met by allocating space from the heap. When an area that has been allocated is no longer needed it is returned to the heap and is available for re-allocation. The utility HEAP described in chapter 7 can be used to find out how the space in the heap has been allocated. (When running in the second 6502 processor using the Tube the system automatically extends the heap to use the extra RAM available).

Some areas in the heap are permanently allocated for system use. The rest of the heap is allocated as required for the following purposes:

- store files. Each file needs a file header block, a file name block and one or more data blocks;

- program stacks;

- program data areas (or vectors). This category includes both areas explicitly requested using the library procedure GETVEC and areas allocated by the system for use during I/O (see chapter 6).

Two problems can arise with management of the heap. Firstly an area may be allocated but never returned (e.g. if a program fails or is interrupted and never resumed). Secondly the heap may become fragmented so that although there is a large amount of unallocated space it is split into many small areas. The system provides explicit solutions to both these problems.

The first problem is solved by the TIDY procedure. This procedure frees all allocated areas in the heap except those in use for store files. It is invoked automatically by the system whenever a program terminates normally, but is not invoked if a program fails (since this might destroy valuable diagnostic information). The built in command TIDY invokes the TIDY procedure.

The second problem is solved by the SHUFFLE procedure. This procedure optionally deletes unprotected store files and then shuffles the remaining store files towards the bottom of the heap so as to leave a contiguous area at the top. Files are shuffled even if they are currently being read or written, but not if they are linked. The SHUFFLE procedure is automatically invoked when an attempt to allocate an area from the heap fails, but it can be invoked directly by the built in command SHUFFLE.

BCPL WORD SIZE

The BCPL language is not a typed language i.e. it does not contain a number of different data types such as integer, fixed-point, pointer etc. Instead all data items are of a fixed size and are known as cells (see chapter 10 for more details).

In this implementation the cell size is 16 bits i.e. one word. Thus in particular all addresses within BCPL programs are word addresses. It is important to remember this when using operating system routines which deal with byte addresses.

OVERVIEW OF PROGRAM DEVELOPMENT

This section gives an overview of how various utilities are used during the development of a program.

Considering first the development of a small program which does not use any of the procedures in LIB, the stages might be:

(1) Create the BCPL source file using ED.

(2) Compile the source using the BCPL compiler to give a CINTCODE file. Correct errors in the source using ED and then recompile.

(3) Test the program using DEBUG. TESTPRO might be useful to test particular procedures in isolation. As bugs are found edit the source using ED and recompile.

Development of a larger program involves more stages. Typically a large program is split up into a number of sections, each of which is in a separate source file and is therefore compiled independently. The global vector is used for communication between sections (both for common data and for allowing procedures in one section to call those in another section), and therefore a header file is created containing all the global declarations. Each section uses the GET instruction to include the header file in the compilation. The program may need procedures written in machine code. The source of such procedures is created using the editor, but instead of being compiled it is assembled using RAS, the relocatable assembler.

When all the source files have been compiled/assembled they must be joined into one CINTCODE file (unless the program contains its own overlaying mechanism to load and link various files at run-time). The utility JOINCIN is used for this.

The program may use library procedures (from the library file LIB or from a user-created applications library). These procedures must be extracted from the library file and incorporated into the program using the utility NEEDCIN.

When an error is found only the relevant source file need be recompiled or reassembled, but JOINCIN and NEEDCIN must be run again. The use of an EX command file simplifies this process. Note that an error in a header file means that all sections using that header file must be recompiled.

To summarise, the following stages are involved:

(1) Create the BCPL source files and the Assembler source files using ED. Create a header file. Some of the BCPL source files contain NEEDS instructions for the sections required from LIB or other libraries.

(2) Compile the BCPL source files using BCPL and assemble the assembler source files using RAS. Correct any errors in the source using ED and recompile/reassemble. The result is a number of CINTCODE files.

(3) Join the CINTCODE files into one CINTCODE file using JOINCIN.

(4) Extract the sections needed from LIB and other libraries and join them on to the CINTCODE file using NEEDCIN. The result is a file that can be loaded and executed.

(5) Test the program using DEBUG and, possibly, TESTPRO. When errors are found amend the source using ED and return to step 2 to recompile or reassemble the updated source.

3 Built in Commands

This chapter describes the built in commands contained in the BCPL ROM. All these commands, except PAUSE, may be used when there is an interrupted program without affecting that program. With the exceptions of CONT, INIT and TIDY these commands may be invoked from within a program using the procedure RUNPROG. If any command fails an error number is displayed on the console (or returned to the calling routine if RUNPROG was used). The error numbers are listed in chapter 11. Chapter 2 describes how the arguments to the commands are specified. The following commands are described in this chapter (in alphabetical order):

CONT	Continue program
COPY	Copy file
DELETE	Delete file
END	End command file
ERRCONT	Continue command file if error
INIT	Initialise program for testing
LINK	Link file into global vector
LOAD	Format CINTCODE file for linking
MODE	Change display mode
PAUSE	Suspend command file
PROTECT	Hold file in store
READ	Read file into store
REM or //	Comment
RENAME	Rename file
SAVE	Save store file
SHUFFLE	Maximise contiguous free store
STORE	Catalogue of store files
TIDY	Free up store
TYPE	Display text file
UNLINK	Unlink file from global vector

CONT - continue program

Purpose:

To continue execution of an interrupted program.

Arguments:

None.

Example:

CONT

Remarks:

This command is used to continue a program which has trapped to the command state. The trap may have been caused by **ESCAPE** having been pressed, by the program having run out of heap space or by the program having called the TRAP procedure.

The command is also used to resume a command file which has been suspended by the PAUSE command.

When testing with DEBUG it is possible to run a program up to a certain point under DEBUG, exit from DEBUG and then run the program to completion by CONT.

A program which has completed, aborted or trapped with a fatal error cannot be continued with CONT.

The command cannot be issued from a command file and cannot be invoked by the procedure RUNPROG.

COPY - copy file

Purpose:

To copy a file from one device to another or to duplicate a file.

Arguments:

FROM/A,TO/A

Examples:

COPY file1 TO file2
COPY /P myfile

Remarks:

This command is a general purpose binary copy which copies from any device/file to any other device/file. For copying files from the current filing system to store the READ command should be used. For copying files from store to the current filing system the SAVE command should be used. For copying text files to the screen or printer the TYPE command should be used.

ESCAPE may be used to interrupt the copy. Both files are closed. The output file contains all characters read up to the time **ESCAPE** was pressed.

COPY is particularly useful for reading files from a serial link when there is no positive end of file indication. Simply wait until it is known that the entire file has been received then press **ESCAPE**.

If the output file already exists it is overwritten.

When copying to or from the current filing system the load address and execution address of the file are not set up or read.

DELETE - delete file

Purpose:

To delete a file.

Argument:

FILE/A

Examples:

```
DELETE textfil
DELETE /F.myfile
```

Remarks:

FILE must be either a store file or a current filing system file. If no device is specified store is assumed.

END - end command file

Purpose:

To terminate processing of a command file.

Arguments:

None.

Example:

```
END
```

Remarks:

If this command is encountered in a command file it terminates processing of that file.

If it is entered at the console when there is a suspended command file processing of that file is terminated. It should be followed immediately by the command TIDY to tidy up the currently suspended program.

If it is entered at the console when there is no suspended command file it is ignored.

ERRCONT - continue command file if error

Purpose:

To control the action to be taken if a program being run from a command file fails.

Argument:

OFF/S

Examples:

ERRCONT
ERRCONT OFF

Remarks:

The normal action taken when a program being run from a command file fails (i.e. terminates with a positive error code) is to abandon the command file. Issuing the command ERRCONT overrides this default so that processing of the command file will continue. ERRCONT may be issued within the command file or it may be typed in at the console before running a command file. Once ERRCONT has taken effect it remains in effect until either ERRCONT OFF is issued (from within the command file or at the console) or the command file terminates.

While ERRCONT is in effect, the global variable LASTERROR (declared in SYSHDR) is set to the error code returned by the last program run from the command file, and so a program may test whether the previous program was successful or not.

Note that while ERRCONT is in effect pressing **ESCAPE** while a command-state program or built-in command (e.g. COPY) is running causes the system to terminate that command, set LASTERROR to 1017 and continue with the next command in the file.

If a program run from a command file ABORTs then the command file is terminated regardless of whether or not ERRCONT has been issued.

INIT - initialise program for testing

Purpose:

To initialise or reinitialise a program being tested with DEBUG.

Arguments:

None, but the command is followed by arguments for the program being tested.

Examples:

```
INIT
INIT file1 file2 AS file3
```

Remarks:

INIT initialises an already linked program so that it can be run under DEBUG (see chapter 7). For example the following sequence of commands might be used to run JOINCIN (see chapter 4) under DEBUG:

```
!LINK JOINCIN
!INIT file1 file2 AS file3
!DEBUG
*0C
```

The line 'INIT file1 file2 AS file3' sets up the text 'file1 file2 AS file3' in the input stream for JOINCIN, so that the effect is the same as if JOINCIN had been run by the command:

```
!JOINCIN file1 file2 AS file3
```

This command may not be issued from a command file and may not be invoked using the procedure RUNPROG.

LINK - link file into global vector

Purpose:

To link a CINTCODE file to the global vector, loading it and/or relocating it if necessary.

Arguments:

FILE,SYSTEM/S,LIBRARY/S

Examples:

```
LINK proga
LINK /F.newwrch SYSTEM
LINK LIBRARY
```

Remarks:

There are three variants of this command:

```
LINK file
LINK file SYSTEM
LINK LIBRARY.
```

The first variant searches for the specified file. It then copies it to store (unless it is already in store), loads it, relocates any assembler code (unless the file is already linked) and links it. The file is protected. Note that if the file is a SYSTEM file (see below) it retains the SYSTEM attribute. This command is normally of use in testing new code when it is desirable to link code without executing it (so that DEBUG or TESTPRO can be used to run it as described in chapter 7).

The second variant has the same effect as the first except that the file is made a SYSTEM file, which means that it is not unlinked by TIDY or by normal program completion. Thus it remains linked until explicitly unlinked by the UNLINK command (see below). This command has two main uses. Firstly it allows alternative versions of library procedures to be used. Secondly it allows libraries of application-specific procedures to be linked in for use by various application programs.

The third variant relinks the ROM library. It must be used before UNLINKing any SYSTEM file containing alternative versions of any library procedures. Note that this command replaces all such alternative procedures by the originals in ROM and so it may be desirable to relink all remaining SYSTEM files after using this command.

LOAD - format CINTCODE file for linking

Purpose:

To get a CINTCODE file into store in a suitable format for linking (i.e. either as a contiguous file or with one CINTCODE hunk per file block).

Argument:

FILE/A

Examples:

LOAD myprog
LOAD /F.abcde

Remarks:

If the specified file is already in store and loaded then it is not reformatted. If the file is in store but not loaded then a copy is made in the correct format and the original version is deleted. If the file is not in store it is read into store (using the equivalent of the command 'READ file') in the correct format.

This command leaves the store file protected whereas running a program by just stating its name (e.g. JOIN file1 file2 AS file3) leaves the file unprotected. Thus LOAD can be used to bring a program into store in such a way that it will remain in store to be run when required.

MODE - change display mode

Purpose:

To change the display mode.

Argument:

MODE/A

Example:

MODE 2

Remarks:

The heap is adjusted if possible to free the memory needed for the requested display mode and, if successful, the new display mode is selected. Note that if the system is running in the second 6502 processor (using the Tube) then no adjustment of the heap is necessary.

The parameter must be a digit in the range 0 to 7.

It is inadvisable to change the display mode while a program is suspended, particularly if that program is one that relies on the display characteristics (e.g. the editor).

The SHUFFLE routine is called if the size of the heap has to be decreased and therefore unprotected store files may be deleted.

If the new mode cannot be selected the command generates error number 15.

This command is the only safe way to change the display mode. In particular the use of **CTRL-V** to change the mode can corrupt the heap.

The characteristics of the various modes are:

Mode	Graphics	Colours	Text	Space (words)
0	640x256	2	80x32	10240
1	320x256	4	40x32	10240
2	160x256	16	20x32	10240
3	-	2	80x25	8192
4	320x256	2	40x32	5120
5	160x256	4	20x32	5120
6	-	2	40x25	4096
7	Teletext Display			512

PAUSE - suspend command file

Purpose:
To suspend a command file.

Arguments:
None, but a message giving the reason for the PAUSE may follow the command.

Example:
PAUSE Please load backup disk

Remarks:
When the PAUSE command is encountered in a command file it is displayed on the console in the normal way followed by the text:

Type CONT to resume

A ‘beep’ is also generated to alert the operator.

The system is now in the command state and all the built in commands (except PAUSE) and command-state utilities may be used. To resume the command file type CONT. To terminate the command file type END and then type TIDY (to tidy up PAUSE which is still active).

Note that unlike all the other built in commands PAUSE is a run-state program.

PROTECT - hold file in store

Purpose:

To change the protection status of a store file.

Arguments:

FILE/A,OFF/S

Examples:

PROTECT myprog
PROTECT tempfil OFF

Remarks:

Unprotected files, provided that they are neither linked nor open, are deleted by the built in command SHUFFLE and by one of the two options of the library procedure SHUFFLE. This library procedure may be called whenever a vector is required from the heap (e.g. when opening a stream, writing to a store file or calling GETVEC).

The ability to unprotect a store file is useful if it is convenient, but not essential, to keep the file in store (assuming that a back-up copy exists). Unprotecting it means that it will be left in store if there is room but will be deleted if the space it occupies is needed for some other purpose.

Files created or copied by programs are protected unless the program specifically removes the protection by using RUNPROG to issue the command ‘PROTECT file OFF’, with the exception that files read into store by LOADSEG are unprotected. Program files loaded into store by commands like ‘program-name arguments’ are unprotected.

Note that being protected does not prevent a file from being deleted by the DELETE command or from being deleted if a new file with the same name is written.

READ - read file into store

Purpose:

To copy a file from the current filing system into store or to copy a store file to another store file.

Arguments:

FILE/A,AS=ON=TO

Examples:

(1) READ myfile
(2) READ /F.file1 AS file2
(3) READ /S.abcde ghijk
(4) READ afile TO bfile

Remarks:

The normal use of this command is to copy a file from the current filing system to a store file of the same name. Thus example (1) would copy the file ‘myfile’ from the current filing system into store. Note that the name specified should not contain a device prefix. The command READ /F.myfile would look for a file named ‘/F.myfile’ on the current filing system.

The alternative use of this command is to copy either a store file or a filing system file to a store file with a different name. For this version the source name may contain a device specifier but the destination name should not, since the destination must be store. Example (2) copies the file ‘file1’ from the current filing system to the store file ‘file2’. Example (3) copies the store file ‘abcde’ to the store file ‘ghijk’. Example (4) creates a store file ‘bfile’ as a copy of the store file ‘afile’ if there is one else as a copy of the current filing system file ‘afile’.

In all cases the store file created is a contiguous file if there is room.

When copying from the current filing system this command is faster than COPY.

If an error or interruption (i.e. **ESCAPE**) occurs during a READ then no output file is created (unlike COPY which produces a truncated output file).

If the destination file already exists it is overwritten.

READing a file from the current filing system does not copy the file's load or execution address.

REM (//) - comment

Purpose:
To allow comment lines to be typed at the console or to be included in command files.

Arguments:
None.

Examples:
```
REM This is a comment
// and so is this
```

Remarks:
The rest of the line is ignored. Note that there must be at least one space between REM or // and the text following it.

RENAME - rename file

Purpose:
To rename store files and current filing system files.

Arguments:
FROM/A,TO/A

Examples:
```
RENAME file1 TO file2
RENAME /F.data olddata
```

Remarks:
The device to be used (store or current filing system) is taken from the FROM name. If no device is specified then store is assumed. The TO name must not contain a device specifier.

RENAME for store files fails if the TO file already exists.

SAVE - save store file

Purpose:
To copy a store file to the current filing system or to some other device.

Arguments:
FILE/A,AS=ON=TO

Examples:
```
(1) SAVE myfile
(2) SAVE datafil TO /P
(3) SAVE filea /F.fileb
(4) SAVE afile AS bfile
```

Remarks:

The normal use of this command is to copy a store file to a file of the same name on the current filing system. To achieve this only one file name is specified. This file name must not include a device specifier. Thus example (1) copies the store file 'myfile' to the current filing system file 'myfile'. The command SAVE /S.myfile would copy the store file named '/S.myfile' to a filing system file with the same name.

The second use of this command is to copy a store file to any other device/file. In this case the source file name must not contain a device specifier (since the device is always store), but the destination file name may contain one. Thus example (2) copies a store file to the serial port (/P), example (3) copies a store file to a file named 'fileb' on the current filing system and example (4) copies a store file to another store file named 'bfile' respectively.

SAVE to the current filing system is faster than the equivalent COPY command.

A side-effect of SAVE is that the source file is made contiguous if possible.

If the destination file already exists it is overwritten.

If an error or interruption (i.e. **ESCAPE**) occurs during a SAVE which is copying a store file to another store file then no output file is created (unlike COPY which produces a truncated output file).

A particularly useful feature of SAVE is that it will save a file to the current filing system even if the heap is completely full (in this situation COPY fails through lack of heap space).

A SAVE to the current filing system does not set up the file's load or execution address.

SHUFFLE - maximise contiguous free store

Purpose:
To rearrange the heap so that as far as possible the free areas are made contiguous.

Arguments:
None.

Example:
SHUFFLE

Remarks:
All unprotected store files are deleted and then the remaining unlinked files are moved down in the heap so that the free space is concentrated at the top of the heap.

The command STORE can be used to see how effective the SHUFFLE has been.

STORE - catalogue of store files

Purpose:
To give a catalogue of the files in store and an indication of how much free space is left in the heap.

Arguments:
None.

Example:
STORE

Remarks:
This command produces a two-part display. The first part is a catalogue of the store files. The second part summarises the space available.

The order of the files in the catalogue is not significant. The following information is displayed for each file:

- the file name;

- the total heap space used for the file (including the space used for the file header block and file name block);

- the attributes of the file, shown as letters with the following meanings:

  G file is linked to the **G**lobal vector;
  L file is **L**oaded;
  R file is open for **R**eading;
  S file is linked as a **S**ystem file;
  U file is **U**nprotected;
  W file is open for **W**riting.

The second part of the display is a line in the format 'Data v1 Free v2 + ... = v3' where v1, v2 and v3 represent numbers. v1 is the total heap space currently in use for program data areas (including stacks). v2 is the size of the largest contiguous free area in the heap that could be obtained by a call to GETVEC without GETVEC having to call SHUFFLE (i.e. it is the value that would be returned by a call of MAXVEC). v3 is the total free space in the heap. Since each vector allocated from the heap needs a few words for system use v3 will always be bigger than v2, even when all the free space in the heap is contiguous.

TIDY - free up store

Purpose:

To restore the system to a tidy state after a program has completed.

Arguments:

None.

Example:

TIDY

Remarks:

If a program completes successfully the system automatically performs a TIDY. If a program fails, however, a TIDY is not performed so that the user can obtain as much information as possible about the failure. Thus TIDY should be used when all such information has been obtained. TIDY may also be used to abandon an interrupted program (note that if the interrupted program was run from a command file then that command file will also be abandoned unless ERRCONT is in effect).

The actions performed by TIDY are as follows:

- all linked files except system files are un-linked;

- all coroutines are deleted;

- all files are closed (except the current command file if there is one);

- all data areas allocated from the heap are returned to the heap;

- screen editing (using **COPY** and the arrow keys) is enabled and the function keys are reset to holding strings.

This command may not be invoked by the procedure RUNPROG.

TYPE - display text file

Purpose:

To display a text file on the console or some other device.

Arguments:

FILE/A,AS=ON=TO

Examples:

(1) TYPE text
(2) TYPE /F.myprog ON /L

Remarks:

If the second parameter is omitted (as in example (1)) the FILE is displayed on the console, otherwise it is copied to the specified device. Thus example (2) copies the file 'MYPROG' from the current filing system to the printer.

The command has the same effect as COPY except that character rather than binary I/O is performed. The effect of this is that '*N' (line feed or LF) characters in the source are ignored and that '*C' (carriage return or CR) characters in the source are expanded to LF/CR. Thus when used on text files created by the BCPL system (which have LF/CR as a line terminator) the command is identical to COPY, but when used on text files which use CR alone as a line terminator it generates the necessary LFs to list the file properly on the screen (and, perhaps, printer).

Care should be taken not to TYPE binary files on the console since if certain sequences of control characters are sent to the VDU driver (e.g. to change the display mode) the heap can be overwritten and the system will crash. **ESCAPE** may be used to terminate TYPE.

UNLINK - unlink file from global vector

Purpose:

To unlink a CINTCODE file from the global vector.

Argument:

FILE/A

Example:

UNLINK MYPROG

Remarks:

If the file is not linked no action is taken. If the file is linked then it is unlinked and any assembler hunks are unrelocated. The file is left loaded.

This command is intended principally for unlinking system files - see the remarks for the LINK command above.

4 Utilities

This chapter describes the utility programs provided with the BCPL system. These permit the development of BCPL and assembler programs. Chapter 2 describes how the arguments to a utility are specified.

The programs provided are as follows:

BCPL — is the compiler and generates CINTCODE from BCPL source.

ED — is a full screen editor, supporting block moves, search and replace, command lines and file handling.

TED — or Tiny ED is a small version of the same editor with fewer facilities.

EX — starts the execution of a command file.

JOIN — joins two or more files to make one larger file.

JOINCIN — joins CINTCODE files to make a larger loadable file.

NEEDCIN — takes a CINTCODE file, and adds CINTCODE sections requested by NEEDS directives in the file. The sections are taken from a specified library file.

RAS — is a 6502 assembler which generates output which can be relocated and used in the same way as CINTCODE.

BCPL - the compiler

Purpose:

To compile source text written in BCPL into CINTCODE which can be run by the BCPL system.

Examples:

(1) BCPL myfile mycode
(2) BCPL /F.bfile cfile MAX REPORT=/L NONAMES

Arguments:

FROM/A,TO/A,REPORT/K,NONAMES/S,MAX/S

Function:

The utility compiles from the source file FROM to the CINTCODE file TO. Thus example (1) will compile from 'myfile' to 'mycode'. Chapter 10 describes the features of the language BCPL.

<u>REPORT</u>

Reports and error messages from the compiler are normally sent to the console. The argument:

REPORT reportfile

causes these messages to be stored in **reportfile.** This enables the verification messages to be directed to a store or current filing system file or to the printer (device /L), and is useful if a large number of error messages is expected. Note that if using an Econet system it is inadvisable to specify the Econet printer as **reportfile**, since the intervals between messages are normally long enough to time the printer out.

NONAMES

This keyword causes the compiler to generate CINTCODE without procedure names. This reduces the size of the CINTCODE by five words for each procedure, but prevents symbolic debugging of the code. The option is therefore normally used at the later stages of the development of a system.

MAX

If this keyword is not present then the compiler checks whether the source files it is to read are in store and, if not, it reads them using the procedure READ. The term source files refers both to the file specified by the FROM argument and to any header files specified by GET instructions.

More specifically, if there is no device specifier in the file name then the compiler checks to see if the file exists in store and, if it does not, reads it into store from the current filing system (as an unprotected file). This results in significantly faster compilation times when using the disk and Econet filing systems.

When compiling large files, however, there may not be enough room to keep the source in store and still compile successfully. Specifying MAX in the command line prevents the compiler copying the source files into store.

Remarks:

The compiler can be interrupted by pressing **ESCAPE.**

The GET directive

The header file LIBHDR is included in most programs, since this declares the most frequently used procedures in the BCPL ROM. When compiling from tape, LIBHDR has been placed so that it is the next file to be read when it is normally required.

Input text characters

The compiler ignores the eighth bit of any character read from the source text. It can thus be used on text containing parity bits, or on text created by word processing packages which use the eighth bit for format control.

The compiler ignores most characters which cannot be displayed. This avoids confusing errors if such characters have been accidentally introduced during editing.

The invisible characters which are accepted are as follows:

Character	**Control**	**Decimal**	**BCPL symbol**
Escape		27	*E
Tab	^I	9	*T
New page	^L	12	*P
Carriage return	^M	13	*C
Line feed	^J	10	*N
Rubout		127	

Escape and rubout are treated as illegal characters and generate an error. Tab is treated like space. The other three characters are treated as ‘end of line’.

Compilation Size

The compiler has a limit on the size of section it can compile. On a Model B computer without a second 6502 processor, this will be met with a section of around 200 lines of typical BCPL source. This limit is reduced by any files in store, but small sections can be compiled from source files in store.

The space required for compilation depends both on the code included within the section and on any header files introduced by GET statements in the section. The workspace required is not affected by comments, nor by conditional code which is not compiled.

If a section is too large to compile, it can normally be divided easily into two or more smaller sections, probably with the declaration of a few more globals to link the sections. However larger sections can also be handled if the header files are reduced to the declarations actually needed by the program. It is not advisable to alter the standard header files, but the declarations required from these files could be edited into the header file for the program under development, so that direct references to LIBHDR or SYSHDR could be omitted.

The compiler accepts any number of sections in a source file (except if compiling from tape), but it is usually more convenient to keep the source files as single sections and to join the compiled code as required, using the utility JOINCIN.

See also the description of the MAX keyword.

Compiling from tape

The compiler on tape holds six files in the following order:

```
BCPL
BCPLARG
BCPLSYN
LIBHDR    // the library header file
BCPLTRN
BCPLCCG
```

Before compiling from tape it is necessary to ensure that not more than one source file has to be read from tape. The most convenient method is to place the source file in store with any header files other than LIBHDR.

If the source file is in store then it is only necessary to rewind the tape to before the file BCPL and enter the required compilation command. If the source contains more than one section it is necessary to rewind the tape after BCPLTRN has been loaded to allow BCPLSYN to be reloaded for the next section.

If the source code is to be compiled directly from tape, each file must contain only one section. (This is recommended practice in any case.) All necessary header files must be copied into store, including LIBHDR if required. During the compilation the system will request the source file after BCPLSYN has been loaded. At this stage the tape recorder should be stopped, the system tape should be removed and the source tape should be inserted. When the source tape has been read the message ‘Text read’ will be shown, and the system will ask for the file BCPLTRN. It is then necessary to remove the source tape and replace the system tape, but it is not necessary to rewind the system tape unless it has read past the start of BCPLTRN.

ED (& TED) - the editor

Purpose:

To create and edit text files.

Examples:

ED txtfile
TED newfile NEW

Arguments:

FROM/A,NEW/S

Function:

The editor may be used to create a new file, or to alter an existing one. The text is displayed on the screen, and may be scrolled vertically or horizontally as required.

Two versions are provided. TED provides a useful set of editing functions in a minimal space. This is important if the editor is to be left in store. ED provides the same set of functions plus an additional set including searches, block moves, file access, and repeated commands.

When the editor is running the bottom line of the screen is used as a message area and command line. Any error messages are displayed here, and remain displayed until another editor command is given.

The editor attempts to keep the screen up to date, but if a further command is entered while it is attempting to update the display, the command is executed at once and the display is updated later, when there is time. The current line is always displayed first, and is always up to date.

Editor commands fall into two categories immediate commands and extended commands.

Immediate commands are those which are executed immediately, and are specified by a single key or a control key combination.

Extended commands are typed in on the command line, and are not executed until the command line is finished. A number of extended commands may be typed on a single command line, and commands may be grouped together and groups repeated automatically. Most immediate commands have a matching extended version.

TED does not support any of the extended commands.

A summary of all editor commands is given in chapter 11.

For the rest of this section **f0** denotes function key 0, **f1** function key 1 etc. **CTRL-f0** denotes that **CTRL** and **f0** should be pressed simultaneously. While the **SHIFT** key is obviously significant when entering text, it is not significant for other editor functions. Thus the function keys, for example, have the same effect whether or not **SHIFT** is pressed. Similarly the extended commands may be entered in either upper- or lower-case.

Starting and stopping

The normal method of invoking the editor is by a command of the form:

ED file (or TED file)

The editor searches for the file **file.** If found it is read into an internal buffer and the first few lines are displayed. If not found the editor assumes a new file is to be created and displays a blank screen. In both cases the editor is now ready to accept immediate commands.

An alternative method of invoking the editor is by a command of the form:

ED file NEW (or TED file NEW)

In this case the editor assumes a new file is to be created whether or not a file called **file** already exists.

When reading in a file most control characters are ignored. The exceptions are CR (0D hex) which is treated as end of line and TAB (09 hex) which is converted to one or more spaces, assuming a tab position every eight characters.

The editor always works on a copy of the file being edited. Thus the original file remains unchanged until the editor is exited.

The normal way of stopping is **CTRL-f8.** The editor checks if the file to be created already exists. If so it renames it to 'BACKUP$' and (if it is a store file) leaves it unprotected. The editor then writes out the new file.

It is possible to stop without creating the output file (if a mistake has been made during the editing, for example) by **f9**. If no changes have been made to the file the editor just exits, otherwise it prompts for confirmation (to prevent accidentally losing changes).

Note that although the editor is supplied with only one file name as a parameter, this name is interpreted differently as an input file and an output file. Thus 'ED myfile' looks first for a store file called 'myfile' then for a current filing system file called 'myfile'. In all cases, however, when it exits it creates a store file called 'myfile'. 'ED /F.myfile', on the other hand, looks only for a file in the current filing system, and when it exits it writes to the current filing system.

Interrupting

As with any program, the **ESCAPE** key may be used to trap to the command mode. This may be useful if it is desired to save a copy of the original file before completing the edit, or to display directories or files to find some information needed for the edit. The editor is resumed by CONT, followed by **CTRL-f0** to regenerate the display.

Text entry

Text may be entered at the cursor position simply by typing it in. Any text on the same line at or to the right of the cursor is shifted right. All printable characters may be entered. **RETURN** starts a new line. The screen is scrolled as necessary to keep the cursor visible.

If text is typed past column 253 then a new line is automatically inserted before the current word. In ED the column at which this occurs can be changed by the extended command **RM.**

TAB inserts enough spaces to take the cursor to the next tab position. Initially there is a tab position every three characters. The **TS** extended command (ED only) may be used to change the tab spacing.

CTRL-f5 creates a new blank line after the current line and positions the cursor at the start of it.

<u>Cursor movement</u>

The cursor is moved one position in any direction by the four arrow keys. If the cursor is on the edge of the screen the text is scrolled to make the rest of the text visible. Vertical scroll is done a line at a time, while horizontal scroll is done ten characters at a time. The cursor cannot be moved off the top of the file or off the left hand edge of the text.

Other cursor movement functions are:

f3	scroll down 12 lines vertically;
f4	scroll up 12 lines vertically;
f5	move cursor to the start of the first line on the screen, unless it is already there when it is moved to the start of the last line on the screen;
f6	move cursor to the start of the current line, unless it is already there when it is moved to the end of the current line;
f7	move to the space following the previous word;
f8	move to the first character of the next word;

CTRL-f3 move cursor to first character in file;
CTRL-f4 move cursor to first character in last line of file.

Text deletion

The text deletion functions are:

DELETE delete the character to the left of the cursor;
f1 delete the character at the cursor position;
f2 delete from the cursor position to the end of the current word. A word for this purpose is defined as a contiguous set of spaces, a contiguous set of alphanumeric characters (including '$') or a contiguous set made up of any other characters;
CTRL-f1 delete from the cursor position to the end of the current line;
CTRL-f2 delete the current line.

Note that the functions above cannot be used to join lines by deleting the 'end-of-line' character. To do this use:

CTRL-f6 strip off all leading spaces from the next line then append it to this line. A single space is inserted between the lines.

Miscellaneous immediate commands

CTRL-f0 regenerate the current display;

CTRL-f7 undo changes on current line. The editor takes a copy of the line the cursor is currently on, and modifies this when characters are added or deleted. The changed copy is replaced back into the file when the cursor is moved off the current line (either by cursor control or by deleting or inserting a line). The copy is also replaced when any scrolling, either vertically or horizontally, is performed. The 'undo' command causes the changed copy to be discarded and the old version of the current line to be used instead;

CTRL-C (ED only) centre current line (see 'Formatting text' below);

CTRL-E (ED only) display last extended command (see below);

CTRL-F (ED only) format current line (see 'Formatting text' below);

CTRL-R (ED only) repeat last extended command (see below);

CTRL-S (ED only) display a summary of the current ED parameters.

Overview of extended commands

The rest of this section is concerned with extended commands and therefore applies only to ED.

To enter extended command mode press **f0**. A prompt ('*') is displayed at the bottom of the screen. One or more extended commands may now be typed in. **DELETE** may be used to delete the last character typed. The commands are not executed until either **RETURN** or **f0** is pressed. **RETURN** causes a return to immediate mode when the commands have been executed; **f0** causes a return to extended mode.

Entering a control character or pressing any function key (except **f0**) while in extended mode returns immediately to immediate mode.

If entering more than one command ';' must be used to separate the commands.

Some extended commands include a text string. Such strings must be delimited at both ends by a character which does not appear in the string. Any character except space, ';', '(', ')' and all letters and numbers may be used for the delimiter. In most of the examples '/' is used.

The last extended command entered is remembered and there are two immediate commands which use this:

CTRL-E displays the last extended command as if it had just been typed in. **RETURN** or **f0** can be entered to execute it or it can be modified by using **DELETE** and/or typing in further commands;

CTRL-R repeats the last extended command.

<u>Inserting text with extended commands</u>

A/s/ inserts the string **s** as a line after the current line and leaves the cursor at the end of this new line;

I/s/ inserts the string **s** into the current line at the cursor position (i.e. it is equivalent to typing in the string in immediate mode);

IL/s/ inserts the string **s** as a line before the current line and leaves the cursor at the end of this new line;

S splits the current line at the cursor position (i.e. has the effect that **RETURN** has in immediate mode).

Cursor movement with extended commands

B moves the cursor to the bottom (first character of last line) of the file;

CE moves the cursor to the end of the current line;

CL moves the cursor left one character position;

CR moves the cursor right one character position;

CS moves the cursor to the start of the current line;

N moves the cursor to the beginning of the next line;

P moves the cursor to the beginning of the previous line;

T moves the cursor to the top (first character) of the file.

Deleting text with extended commands

DC deletes the character at the cursor position;

DE deletes from the cursor position to the end of the current line;

DL deletes the current line;

DW deletes from the cursor position to the end of the current word;

J joins the next line to the current line, stripping off leading spaces from the next line but inserting one space where the join occurs.

Finding and replacing text

The find and replace commands all ignore the difference between upper and lower case when matching the given string with the text in the file. When replacing, however, the new text is inserted exactly as typed.

E/s/t/ search the file, starting with the character at the cursor position, for the string **s**. If found replace it with the string **t** and leave the cursor at the next character. If not found leave the cursor at its original position;

EQ/s/t/ same as the command E but ask for confirmation from the user before performing the substitution (if the substitution is not performed leave the cursor at the character beyond the matching string **s** in the file);

F/s/ search the file, starting with the character to the right of the cursor position, for the string **s**. If found leave the cursor at the start of the matching text. If not found leave the cursor at its original position.

Note the difference in the first character searched between the find and the exchange commands.

Blocks and place markers

The editor allows two lines of text to be marked as the block start line and the block end line. This facility can be used in two different ways.

Firstly it can be used to identify one or two lines in the file, so that having moved the cursor to some other place in the file it can be restored to a known position.

Secondly it allows a block of text to be identified. This block may be copied, moved or deleted. When used in this way the block is marked on the screen, either by being displayed in cyan (if in display mode 7) or by being prefixed by a double vertical bar (if in any other display mode).

A block can be hidden. When a block is hidden it is not marked on the screen and the copy, move and delete facilities do not work. The cursor may still be moved to the block start and block end however. This mode is useful to prevent accidental block deletion when using the block facility just as two line markers.

The block commands are:

BE mark the current line as the block end;
BS mark the current line as the block start;
DB delete the current block;
HB if block is not hidden then hide it; if it is hidden then 'unhide' it;
IB insert a copy of the current block into the file just before the current line. Leave the cursor at the start of the copy;
MB move the current block to just before the current line (equivalent to IB followed by DB). Leave the cursor at the start of the block;
SB move the cursor to the start of the block start line;
SE move the cursor to the start of the block end line.

<u>File handling</u>

Commands are provided to read a specified file into the current file and to write out the current file or part of the current file as a specified file. Both commands have a file name as a parameter. This must be delimited in the same way as strings. The commands are:

IF/f/ read the contents of file **f** into the current file just before the current line. Leave the cursor at the first character read in;
SV/f/ write the current file as file **f**. If the file already exists it is over-written;

WB/f/ write the current block as file **f.** If the file already exists it is over-written.

Formatting text

This section covers various facilities which are principally of use when editing documents or letters rather than program source.

A left margin and a right margin can be defined. When typing in text new lines are automatically inserted so **RETURN** need only be used when a blank line is required. When a word is entered which goes beyond the right margin the entire word is moved to the next line (and starts at the left margin position). The left margin may not be positioned beyond the right hand edge of the screen. The right margin may be anywhere from the left margin up to column 255. The initial margin settings are left margin column 1, right margin column 253. The margin commands are:

EX extend right margin (i.e. allow text on the current line to be typed beyond the right margin);
LM n set left margin to column n;
RM n set right margin to column n;
C centre the current line between the current margins and move the cursor to the start of the next line.

The immediate command **CTRL-C** has the same effect as the **C** command.

Tab stops may be set:

TS n sets tab stops every n character positions. The **TAB** key, used in immediate mode, inserts enough spaces to bring the cursor to the next tab stop.

The immediate command **CTRL-F** formats the current line using the current margins as described in the next paragraph.

If the current line does not reach the right margin, the next line is joined to it. Whether or not the join took place, if the current line now exceeds the right margin it is split into two lines at a word boundary and the cursor is placed on the second line. This command will not remove blank lines and so letters to be formatted should have blank lines between paragraphs.

The extended command **FM** has the same effect. It may be used to format many lines of text in one command by use of the **RP** command (RP FM) described in the next section.

Repeating commands

Any command may be repeated n times by preceding it with n (e.g. 4N moves the cursor down four lines, 2F/red/ finds the second occurrence of ‘red’, 3E/blue/green/ replaces the next three occurrences of ‘blue’ by ‘green’).

Any command may be repeated indefinitely (or until an error occurs) by preceding it with RP (e.g. RPE/blue/green/ replaces all occurrences of ‘blue’ by ‘green’).

A repeated command may be interrupted by pressing any key.

Groups of commands may be repeated by enclosing the group in curved brackets. For example, the following extended command replaces every third occurrence of 'abc' by 'def':

```
T;RP(3F/abc/;E/abc/def/)
```

(The initial 'T' ensures the search starts at the beginning of the file).

Normally when executing sets of extended commands the display is updated as each command is performed. The Q (quiet) command prevents the display being updated until the end of the command line is reached. If many commands are being executed this results in a significantly faster response. Note that the command is effective only for the command line containing it - it does not affect subsequent command lines.

EX - execute command file

Purpose:

To start execution from a command file. The command file is created from an exfile which provides a template which may be modified by the arguments to the EX utility.

A command file is a text file containing a sequence of commands which are to be executed as though they had been entered at the console. The use of a command file enables a sequence of operations to be performed when the computer is not being attended (e.g. to perform a series of compilations on a disk-based system). A command file may also be used to repeat sequences of operations which would be tiresome to re-enter each time.

Examples:

```
EX exfile
EX compile FROM mysrce TO myprog
```

Arguments:

The first argument is the name of the exfile. The remaining arguments are interpreted according to the .KEY directive in the exfile (see below).

Function:

EX copies the exfile to a temporary store file (named '$$EX') then exits leaving this temporary file as a command file to be executed. The system deletes the temporary file when it has finished with it (i.e. either when the end of the file is reached or when execution of the file is terminated for some other reason).

The temporary file may be a simple copy of the exfile, or it may be modified by substituting the arguments of EX for keywords in the exfile as described below.

<u>Format of exfiles</u>

Exfiles are text files made up of a number of lines. Each line is either a directive or a normal line.

Directives start with the character '.' (this character may be redefined - see below). Directives are instructions to EX and are not copied to the temporary file.

The following directives exist:

.KEY arguments

This specifies the argument definitions used to interpret the EX command line. The arguments are in the format used by the library procedure RDARGS, but every parameter should be given a keyword. There may be at most one .KEY directive in an exfile.

Keywords may be included in a normal line by enclosing them in angle brackets (‘<’ and ‘>’). When EX is copying the file it replaces the keywords by their associated arguments.

.DEF keyword=value

This specifies a value to be associated with a keyword if the corresponding argument is omitted from the EX command line. If more than one .DEF directive is specified for the same keyword all but the first are ignored.

.BRA character

Redefines the character indicating the start of a keyword within a normal line (initially ‘<’).

.KET character

Redefines the character indicating the end of a keyword within a normal line (initially ‘>’).

.DOT character

Redefines the character introducing a directive (initially ‘.’).

The use of these directives is best explained by example.

Examples

In some examples a file naming convention has been assumed in which BCPL source files begin with 'b.' and CINTCODE files begin with 'c.' (note that this convention is well-suited for use with the disk/Econet filing systems).

(1) No substitution

The simplest form of exfile contains no directives at all. E.g. a file 'mycmds' might be:

```
BCPL b.sega c.sega
BCPL b.segb c.segb
JOINCIN c.sega c.segb as c.myprog
```

This file is executed by the command

```
EX mycmds
```

and performs two compilations and a JOINCIN.

(2) Simple substitution

This example uses the .KEY directive to produce a command file to compile and JOINCIN any two files. If the file 'mycmds' is:

```
.KEY FILE1/A,FILE2/A,PROG/A
BCPL b.<FILE1> c.<FILE1>
BCPL b.<FILE2> c.<FILE2>
JOINCIN c.<FILE1> c.<FILE2> AS c.<PROG>
```

then the command

```
EX mycmds secta sectb mytask
```

generates the following command file:

```
BCPL b.secta c.secta
BCPL b.sectb c.sectb
JOINCIN c.secta c.sectb AS c.mytask
```

Note that in this example all the keywords have the qualifier '/A' meaning that the corresponding argument must be present.

(3) Missing arguments

Arguments for keywords without the '/A' qualifier may be omitted. Consider the file 'dojoin':

```
.KEY PROG/A,FILE1/A,FILE2,FILE3
JOINCIN <FILE1> <FILE2> <FILE3> AS <PROG>
```

The command

```
EX dojoin myprog secta sectb
```

generates the command file:

```
JOINCIN secta sectb AS myprog
```

Here the third keyword, FILE3, has no corresponding argument and so is replaced by a null string wherever it appears.

(4) Default values

Consider a command file to run NEEDCIN (see chapter 4) called 'doneed':

```
.KEY FROM/A,TO/A,LIB
.DEF LIB=mylib
NEEDCIN <FROM> <LIB> <TO>
```

When executed by

```
EX doneed myinput myoutput myprocs
```

the resulting command file is:

```
NEEDCIN myinput myprocs myoutput
```

But when executed by

```
EX doneed myinput myoutput
```

the resulting command file is:

```
NEEDCIN myinput mylib myoutput
```

In the second example the default value for LIB has been used since no value was specified for it in the EX command line.

(5) Redefining the special characters

If an exfile is required in which a normal line begins with ‘.’ or in which the characters ‘<’ or ‘>’ are required then the directives .DOT, .BRA or .KET must be used.

If the file ‘weirdfile’ were:

```
.KEY ARG1,ARG2
.DOT #
.KEY This is a normal line
#BRA (
#KET ]
<ARG1> is not substituted
BCPL (ARG1] (ARG2]
#DOT .
.KET )
(ARG2)<ARG2>
```

then the command

```
EX weirdfile abcd efgh
```

would generate the command file:

```
.KEY This is a normal line
<ARG1> is not substituted
BCPL abcd efgh
efgh<ARG2>
```

The .DOT directive indicates that subsequent directives begin with '#' rather than '.'. Thus the line beginning '.KEY This' is not treated as a directive. The next two lines redefine '<' as '(' and '>' as ']'. The #DOT directive restores '.' as the character introducing a directive and the .KET directive therefore redefines ']' as ')'.

Nested command files

If the EX utility is called from a command file it creates a new command file in the normal way, and then appends the rest of the current command file. The current command file is then deleted and the new one is substituted.

This enables a command file to include one or more calls to the EX utility.

Remarks:

Executing a command file

When the system is executing a command file it reads the next command from the file instead of from the console. The command is echoed to the screen. If the command is to run a program (or utility) that program is entered with the current input stream being the command file. Normally command files are used to run programs which take all their parameters from the command line. If a program is designed to read a number of lines of input from its initial current input stream (normally the console) then these lines are read from the command file, but are not echoed to the screen. The program must cope with unexpectedly reaching the end of the command file (perhaps by selecting the console for input). Note that it should not perform an ENDREAD on the command file input stream.

If a program being run from a command file terminates by calling ABORT then the command file is terminated. If a program terminates by FINISH, calling ENDPROG or returning from START then the next command in the file is executed. If a program terminates by calling STOP then the result depends on the parameter to STOP.

If the parameter is negative (warning) or 0 (success) the next command in the file is executed. If the parameter is positive (error) then the command file is terminated unless the built in command ERRCONT is in effect (see chapter 3), when the next command is executed.

<u>Trapping to the command state</u>

If a program being run from a command file traps to the command state for any reason (including **ESCAPE** being pressed) then the system suspends the file and takes its input from the console instead. The user may then run trap-state programs and built in commands as usual. The command CONT resumes the program and also resumes the command file (so that when the program ends successfully the system reads the next command from the file).

The command TIDY abandons the program that has trapped. If ERRCONT is not in effect it also abandons the command file. If ERRCONT is in effect then it sets LASTERROR to -4 and resumes the command file with the next command.

The command END abandons the command file. It should be followed by TIDY to tidy up the program that has trapped.

If **ESCAPE** is used to interrupt a trap-state program or command (e.g. COPY) being run from a command file then if ERRCONT is not in effect it abandons the command file. If ERRCONT is in effect then it sets LASTERROR to 1017 and resumes the command file with the next command.

Ignoring parts of a command file

The built in command REM (which causes the rest of the line to be treated as a comment) may be used, in conjunction with the ability to omit arguments, to allow parts of a command file to be omitted.

Thus if the file ‘comp’ were:

```
.KEY FILE/A,PROG/A,COMPILE
<COMPILE> BCPL b.<FILE> c.<FILE>
NEEDCIN c.<FILE> mylib c.<PROG>
```

then the command

```
EX comp secta myprog
```

would generate the command file:

```
BCPL b.secta c.secta
NEEDCIN c.secta mylib c.myprog
```

whereas the command

```
EX comp secta myprog compile=REM
```

would generate the command file:

```
REM BCPL b.secta c.secta
NEEDCIN c.secta mylib c.myprog
```

in which no compilation takes place because the command BCPL is preceded by REM.

Terminating a command file

A similar use of the built in command END allows optional termination of a command file before the end of the text in the exfile.

JOIN - join files

Purpose:

To join two or more files to create a larger file.

Example:

JOIN file1 /F.file2 file3 AS bigfile

Arguments:

FROM/A,,,,,,,,,,,,AS=TO/A/K

Function:

JOIN joins up to 12 files together to make one big file. The joined file is given the name specified after the AS (or TO) keyword, which must be present.

Remarks:

The output file must not be the same file as any of the input files.

This utility runs in the trap state and so may be used while there is an interrupted run-state program.

If **ESCAPE** is pressed while JOIN is running it terminates. The output file is truncated.

JOINCIN - join CINTCODE files

Purpose:

To join two or more CINTCODE compatible files to create a larger file, which can be loaded as one segment. It may also be used to add procedures to an existing CINTCODE program.

Example:

JOINCIN segment1 segment2 segment3 AS myprog

Arguments:

FROM/A,,,,,,,,,,,,AS=TO/A/K

Function:

JOINCIN joins up to 12 CINTCODE files. The resulting file is given the name specified by the AS (or TO) keyword, which must be present.

If any of the input files are not valid CINTCODE files no output file is produced.

Remarks:

The output file must not be the same file as any of the input files.

This utility may be used to verify that a file contains valid CINTCODE by a command of the form:

JOINCIN myfile /N

The format of CINTCODE is discussed in chapter 10.

NEEDCIN - extract sections from library

Purpose:

To satisfy NEEDS directives in a CINTCODE file by adding the required sections from a file acting as a library.

Examples:

(1) NEEDCIN mycode mylib myprog
(2) NEEDCIN sega AS /N

Arguments:

FROM/A,LIBRARY,AS=TO/A

Function:

NEEDCIN copies the FROM file to the TO (or AS) file, examining the NEEDS directives in it as it does so. If any NEEDS remain unsatisfied it then opens the LIBRARY file (if one is specified) and searches it for the sections needed. These sections are appended to the TO file, and any unsatisfied NEEDS directives they contain are added to the list of sections required.

When all NEEDS are satisfied, or the end of the LIBRARY file is reached, the TO file is closed and a list of all unsatisfied NEEDS is displayed on the console.

The name for a section is taken from the SECTION directive if there is one otherwise the name of the first procedure in the section is used.

Remarks:

The structure of a LIBRARY file is identical to that of any other CINTCODE segment (see chapter 10). Thus files can be created for use as library files by JOINCIN. The library file provided with the system is LIB.

NEEDCIN can be used as a general method of extracting specified sections from a CINTCODE file. A segment is created containing NEEDS directives for the required sections, and NEEDCIN is applied to this segment specifying the CINTCODE file as the LIBRARY file.

Note that NEEDS directives in sections in the LIBRARY file should not refer to earlier sections in that file (or, if this cannot be avoided, that a second pass of NEEDCIN will be needed to satisfy all NEEDS directives).

NEEDCIN can be used to check if a section has any unsatisfied NEEDS, as in example (2) above.

There is a limit of 39 NEEDS directives in one section.

RAS - the relocatable assembler

Purpose:

To assemble 6502 machine code to a relocatable format that can be used in the BCPL environment.

Examples:

```
RAS myass TO mycode
RAS mysrc1 mysrc2 TO mycode LIST /L
```

Arguments:

```
FROM/A,,,,,,,,,TO=AS/A/K,LIST/K
```

Function:

The assembler source in the FROM files is assembled to CINTCODE in file TO. Up to ten source files may be specified. The assembler produces output compatible with the BCPL CINTCODE system, and can be used to supply assembler routines which can be accessed from BCPL as described in chapter 8. Special facilities are provided to access BCPL global variables, and to supply section and procedure names.

If the key LIST is specified then a listing of the program is written to the associated device/file.

Error messages are written to the terminal unless the listing is going to the terminal, and are reflected in the listing if one has been requested.

If any errors are detected in the source then no output file is created.

If a source file has no device specified in the file name and is not a store file then it is automatically read into store from the current filing system (as an unprotected file).

If more than one source file is specified the source files are read in the order given and processed as if they had been concatenated into one large file.

Chapter 10 defines the assembly language syntax. Chapter 8 discusses the use of machine code routines from BCPL programs.

5 Procedures

This chapter describes the library procedures provided with the BCPL CINTCODE system. The procedures are in alphabetical order.

The procedures are summarised by usage in chapter 11.

The following procedures are described in this chapter:

ABORT	ADVAL	APTOVEC
BACKMOVE	BACKMVBY	
CALL	CALLBYTE	CALLCO
CAPCH	COMPCH	COMPSTRING
COWAIT	CREATECO	
DELETECO	DELFILE	DELXFILE
ENDPROG	ENDREAD	ENDWRITE
ENVELOPE	ERRORMSG	EXTSFILE
FILETOVEC	FINDARG	FINDINPUT
FINDOUTPUT	FINDXINPUT	FINDXOUTPUT
FREEVEC	FSTYPE	
GETBYTE	GETVEC	GLOBIN
GLOBUNIN		
INPUT		
LEVEL	LOADSEG	LONGJUMP
MAXVEC	MODE	MOVE
MOVEBYTE	MULDIV	
NEWLINE	NEWPAGE	

OPSYS	OUTPUT	
PACKSTRING	PUTBYTE	
RANDOM	RDARGS	RDBIN
RDCH	RDITEM	READ
READN	READVEC	READWORDS
RENAME	RESUMECO	RUNPROG
SAVE	SAVEVEC	SELECTINPUT
SELECTOUTPUT	SHUFFLE	SOUND
SPLIT	STACKSIZE	START
STARTINIT	STOP	
TESTFLAGS	TESTSTR	TIME
TRAP		
UNLOADSEG	UNPACKSTRING	UNRDCH
VDU	VDUINFO	VECTOFILE
WRBIN	WRCH	WRITEA
WRITEBA	WRITED	WRITEDB
WRITEF	WRITEHEX	WRITEN
WRITEOCT	WRITES	WRITET
WRITEU	WRITEWORDS	

ABORT

Purpose:

To end a program which has detected an error condition from which it cannot recover.

Examples:

```
ABORT( aborttype)
IF address=0 DO ABORT( 533)
```

Function:

Unless the program is trapping ABORTs (see Remarks below) then the program is terminated. If the program was run from the console a message of the form

Abort nnn

where nnn is the parameter to ABORT is displayed and the program is left linked (so that DEBUG may be used to investigate the cause of the ABORT if required). If the program was run by RUNPROG then the parameter to ABORT is used for the result of RUNPROG.

Remarks:

Abort codes 1-499 and 1000-1255 are reserved for system use. Applications may use all other possible codes.

For most expected error conditions it is better for the program to end by calling STOP with a suitable positive parameter. The use of ABORT should be reserved for exceptional conditions.

Certain programs may wish to trap ABORTs. It is possible to intercept ABORT calls by setting the globals ABORTLEVEL and ABORTLABEL. ABORTLEVEL should be set by a command such as:

```
ABORTLEVEL := LEVEL()
```

and ABORTLABEL should be set to a label in the program.

If the ABORT procedure finds ABORTLEVEL initialised to a location in the current stack, it copies the parameter of ABORT to the global ABORTCODE and LONGJUMPs to the specified ABORTLABEL. ABORTLEVEL is reset to prevent looping on error conditions.

ABORTLEVEL, ABORTLABEL and ABORTCODE are declared in the header file SYSHDR.

The errors that cause the system to call ABORT are those listed in 'Error numbers and trap codes' in chapter 11 with error numbers above 100.

Portability:

The procedure is found in other BCPL implementations, but the provision to trap ABORT calls is specific to this implementation.

ADVAL

Purpose:

To provide a convenient interface to the ADVAL call in the operating system.

Examples:

```
(1) chn1val := ADVAL(1) >> 1
(2) chars.in.buff := ADVAL(-1)
```

Function:

To issue an OSBYTE call with parameter #X80 and return the result of the call.

The effect of the call varies with the parameter.

A parameter of 0 tests the 'Fire' buttons on the games paddles and also finds the channel on which the most recent analogue to digital conversion was performed. It is used as follows :-

```
left.button.pressed  := (ADVAL(0) & 1) NE 0
right.button.pressed := (ADVAL(0) & 2) NE 0
last.channel.cnvrtd  := ADVAL(0) >> 8
// range 1-4 or 0 if no conversions done
```

A parameter in the range 1-4 reads the voltage on the specified analogue input channel. This is a 16-bit unsigned number in the range 0 (0v) to 65520 (approx. 1.8v), with 12-bit precision. Since BCPL arithmetic assumes signed numbers it is convenient to shift this value to occupy only 15 bits (or less) as in example (1) above. This gives a result in the range 0 to 32760.

A parameter in the range -1 to -9 returns the number of characters in an input buffer or the number of free spaces in an output buffer. The buffer numbers are:

```
-1  keyboard
-2  RS423 input
-3  RS423 output
-4  printer
-5 to -8
    sound channels 0 to 3
-9  speech.
```

Remarks:

Further information on the operation of ADVAL is given in the BBC Microcomputer User Guide.

The procedure is contained in section "ADVAL" of the CINTCODE library LIB.

Portability:

Specific to this implementation.

APTOVEC

Purpose:

To call a procedure and to provide it with a vector whose size is chosen at runtime. The vector is obtained from the stack.

Example:

```
res := APTOVEC( f, n)
```

Function:

The procedure **f** is called with two arguments, the vector and the size **n**. The result is the value, if any, returned by the call of **f**. The operation could be described in (illegal) BCPL as:

```
LET APTOVEC( n) = VALOF
$( LET v = VEC n   // illegal (n is variable)
   RESULTIS f( v, n)
$)
```

Remarks:

If there is not enough room for a vector of size **n** in the stack, APTOVEC calls ABORT. The procedure STACKSIZE may be used to check the stack space available before calling APTOVEC.

APTOVEC and GETVEC are alternative ways of reserving variable amounts of space for use by a program.

The main advantage of APTOVEC is that the storage allocated is automatically released on return from the procedure called, whereas with GETVEC the storage must normally be explicitly released using FREEVEC. This point is less important in this system since all vectors obtained by GETVEC are automatically released when the program terminates.

The disadvantage of using APTOVEC is that enough stack space must be allocated to the program to allow for the biggest possible vector that may be required.

APTOVEC is declared in the header file SYSHDR and is contained in section "APTOVEC" of the CINTCODE library LIB.

Portability:
A standard procedure.

BACKMOVE

Purpose:
To provide a rapid transfer of a block of words from one store location to another.

Example:
BACKMOVE(fromaddr, toaddr, words)

Function:
The block of length **words** words starting at word **fromaddr** is copied to a block starting at word **toaddr**, moving the last word in the block first.

Remarks:
If the two blocks do not overlap this procedure is equivalent to MOVE. If they do overlap then BACKMOVE should be used for moving up in store and MOVE for moving down.

If **words** is negative or 0 no data is moved.

BACKMOVE is declared in the header file SYSHDR and is contained in section "BACKMOV" of the CINTCODE library file LIB.

Portability:
Developed for this implementation.

BACKMVBY

Purpose:

To provide a rapid transfer of a block of bytes from one store location to another.

Example:

```
BACKMVBY( frombyaddr, tobyaddr, bytes)
```

Function:

The block of length **bytes** bytes starting at byte **frombyaddr** is copied to a block starting at byte **tobyaddr**, moving the last byte in the block first.

Remarks:

If the two blocks do not overlap this procedure is equivalent to MOVEBYTE. If they do overlap then BACKMVBY should be used for moving up in store and MOVEBYTE for moving down.

If **bytes** is negative or 0 no data is moved.

BACKMVBY is declared in the header file SYSHDR and is contained in section "BACKMOV" of the CINTCODE library file LIB.

Portability:

Developed for this implementation.

CALL

Purpose:

To call assembly language procedures which are located at a word address.

Example:

```
result := CALL( wordaddress, argument1,
                             argument2)
```

Function:

The CALL procedure generates a 6502 machine code call to the specified word address, with **argument1** in register A and **argument2** in registers Y and X, with X holding the least significant byte.

On return, **result** is taken from registers X and Y with X providing the least significant byte. The global MCRESULT contains register A in the least significant byte and the flag register in the most significant byte, unless a fault condition has occurred. In this case the most significant byte is set to #XFF (this is an impossible value for the flag register) and the least significant byte is set to the fault code obtained from the operating system. If a fault has occurred **result** is undefined.

Remarks:

If the called machine code has been generated by the relocatable assembler, and the entry point has been declared as a global, then CALL can use the global to specify the **wordaddress.**

An alternative method of writing machine code procedures so that they can be called as if they were BCPL procedures is described in chapter 8.

Portability:

This feature is similar to the language extension A15, but the declaration of **wordaddress** by EXTERNAL is not supported.

CALLBYTE

Purpose:

To call assembly language procedures at a byte address.

Example:

```
result := CALLBYTE( byteaddress,argument1,
                                argument2)
```

Function:

The CALLBYTE procedure generates a 6502 machine code call to the specified byte address. The parameter passing and return conventions are identical to those for CALL described above.

Remarks:

This procedure is intended for calling machine code procedures which have not been created by the relocatable assembler, and which have fixed entry points which may be on an odd byte address (e.g. the operating system routines OSWORD, OSCLI etc.).

Portability:

This feature is similar to the language extension A15, but the declaration of **byteaddress** by EXTERNAL is not supported. It may be declared as a MANIFEST if required.

CALLCO

Purpose:
To transfer control to another coroutine, passing a parameter, and to receive control back again with a reply.

Example:
```
reply := CALLCO( coroutine, parameter)
```

Remarks:
This procedure is included in section "CORTNS" of the CINTCODE library file LIB.

Portability:
Part of the published method of implementing coroutines in BCPL.

CAPCH

Purpose:
To convert any alphabetic character to upper case, while any other character is returned unaltered.

Examples:
```
capital.char := CAPCH( character)
WHILE 'A' <= CAPCH(ch) <= 'Z' DO
```

Portability:
An equivalent procedure is provided on many systems; in some systems it is called ‘capitalch’.

COMPCH

Purpose:
To compare two ASCII characters.

Example:
comparison := COMPCH(character1, character2)

Function:
To find if two characters are the same, ignoring the differences between upper and lower case. COMPCH is also useful when sorting characters.

Remarks:
The comparison is:

negative	if **character1** comes before **character2**;
zero	if the characters are equivalent;
positive	if **character1** comes after **character2**.

Portability:
This procedure is provided on other BCPL systems.

COMPSTRING

Purpose:
To compare two ASCII strings.

Example:
comparison := COMPSTRING(string1, string2)

Function:

To find if two strings are the same, ignoring the differences between upper and lower case alphabetic characters. COMPSTRING is also useful when sorting strings into lexical order.

Remarks:

The comparison is:

negative	if **string1** comes before **string2;**
zero	if the strings are equivalent;
positive	if **string1** comes after **string2.**

Portability:

This procedure is provided on other BCPL systems.

COWAIT

Purpose:

To suspend a coroutine at a suitable stage, and to wait for reactivation.

Example:

```
parameter := COWAIT( reply)
```

Function:

The **reply** will be returned to the parent co-routine, and the **parameter** will be provided when the waiting coroutine is reactivated.

Portability:

Part of the published method of implementing coroutines in BCPL.

CREATECO

Purpose:

To create a coroutine, with a given entry point and stack size.

Example:

```
coroutine := CREATECO( procedure, stacksize)
```

Function:

The **coroutine** is the reference to be used for the created coroutine. It is zero if the coroutine could not be created for lack of store (note that this can only happen if the program has disabled trapping to the command state on GETVEC failure - see chapter 6).

Portability:

Part of the published method of implementing coroutines in BCPL.

DELETECO

Purpose:
To delete a coroutine.

Examples:

```
is.successful := DELETECO( coroutine)
DELETECO( coroutine)
```

Function:
The **coroutine**'s stack is returned to free store. The result returned is FALSE if the **coroutine** does not exist. DELETECO ABORTs if the **coroutine** is the current coroutine or a parent or ancestor of the current coroutine.

Remarks:
Since DELETECO will not fail if given a non-existent coroutine, it is possible to use a common re-initialising routine, which deletes all possible coroutines.

When a program terminates the system automatically deletes all existing coroutines.

Portability:
Part of the published method of implementing coroutines in BCPL.

DELFILE

Purpose:
To delete a file.

Examples:

```
DELFILE( filename)
is.successful := DELFILE( "/F.myfile")
```

Function:
An attempt is made to delete the file identified by **filename**.

If the deletion succeeds the result is TRUE. If it fails the result is FALSE and RESULT2 contains the appropriate error code.

Remarks:

If the **filename** does not contain a device specifier then the name is treated as the name of a store file. To delete a store file whose name begins with '/' the store file prefix '/S.' must be used.

Files can be deleted from store or from the current filing system. It is an error to attempt to delete a file from any other device except the null device ('/N').

Store files that are open or linked cannot be deleted.

Note that if the current filing system is tape or ROM then no error is generated, even though no file is deleted.

Portability:

Similar functions are provided on other BCPL systems.

DELXFILE

Purpose:

To delete a file on a specified device.

Examples:

```
(1) DELXFILE( filename, device)
(2) is.successful := DELXFILE( "myfile", DV.F)
(3) DELXFILE( "/F.storefile", DV.S)
```

Function:

An attempt is made to delete the file identified by **filename** on the device identified by **device**. **Device** may be 'DV.F' to specify the current filing system or 'DV.S' to specify the store filing system.

If the deletion succeeds the result is TRUE. If it fails the result is FALSE and RESULT2 contains the appropriate error code.

Remarks:

This procedure is useful for deleting a file when the device is known, without having to manipulate filename strings to include a device prefix.

The **filename** is treated as not containing a device specifier. Thus example (3) deletes a store file named '/F.storefile'. If the second parameter had been DV.F then it would have deleted a file named '/F.storefile' from the current filing system (whereas DELFILE("/F.storefile") deletes a file named 'storefile' from the current filing system).

The manifest constants DV.F and DV.S are declared in SYSHDR. Using any other values for the second parameter may give undefined results.

Store files that are open or linked cannot be deleted.

Note that if the current filing system is tape or ROM then no error is generated, even though no file is deleted.

DELXFILE is declared in SYSHDR.

Portability:

Specific to this implementation.

ENDPROG

Purpose:

To provide the user with the option of ending the program.

Example:

```
ENDPROG( endcheck)
```

Function:

If **endcheck** is TRUE a message is displayed as follows:

```
END PROGRAM? (Y/N)
```

If ‘Y’ or ‘y’ is entered the program will end (by calling STOP with a parameter of 0), otherwise the procedure will return to the calling program. If **endcheck** is FALSE, the program ends without requiring confirmation from the operator.

Remarks:

If there is no doubt that the program is to be stopped, the procedure STOP is appropriate. ENDPROG simplifies the handling of cases when a program end may have been requested by accident.

The console must be selected as the current output stream before calling this procedure.

ENDPROG is declared in the header file SYSHDR and is contained in section "ENDPROG" of the CINTCODE library LIB.

Portability:

Developed for this implementation.

ENDREAD

Purpose:
To close the current input stream. If the current input stream is from a file then this procedure closes the file.

Example:
ENDREAD()

Remarks:
Note that in order to close a stream it must be selected.

The dummy stream ERRORSTREAM is selected as the current input stream.

When a program terminates the system automatically closes all open streams.

Portability:
Standard BCPL procedure.

ENDWRITE

Purpose:
To close the current output stream. If the current output stream is to a file then this procedure closes the file.

Example:
ENDWRITE()

Remarks:
Note that in order to close a stream it must be selected.

The dummy stream ERRORSTREAM is selected as the current output stream.

When a program terminates the system automatically closes all open streams, but it is good practice to explicitly close output streams since if the system fails with files open these files may not be recoverable.

Portability:
Standard BCPL procedure.

ENVELOPE

Purpose:
To define an envelope for use with the SOUND facility of the BBC Microcomputer.

Example:
```
LET envtab = TABLE 1,1,4,-4,4,10,20,10,127,
                   0,0,-5,126,126
ENVELOPE( envtab)
```

Function:
The parameter is a fourteen-word table or vector containing the fourteen parameters defining an envelope. The envelope is compacted into a seven-word area and passed to the operating system using the OSWORD call. Details of the envelope parameters are given in the BBC Microcomputer User Guide.

Remarks:
This procedure is provided mainly for ease in converting BASIC programs to BCPL. In many cases it will be easier to make the OSWORD call directly using CALLBYTE.

The procedure is contained in section "ENVELOP" of the CINTCODE library file LIB.

Portability:
Specific to this implementation.

ERRORMSG

Purpose:

To generate an error or warning message on the console, normally when a program terminates with a non-0 parameter to STOP.

Example:

ERRORMSG(stopcode)

Function:

If **stopcode** is 0 no action is taken.

If **stopcode** is non-0 the console is selected as the current output stream and a message of the form:

Warning nnn (or Error nnn)

is generated, where nnn is the value of **stopcode.** The first message is produced if **stopcode** is negative; the second if it is positive.

If **stopcode** is in the range 1000 to 1255 then the error is assumed to be the result of a fault condition generated within the operating system and the fault text is accessed (as described in the BBC Microcomputer User Guide) and displayed. One important exception is that error 1017 may be generated internally by the BCPL system, so for this error number the text 'Escape' is always generated.

Remarks:

This procedure is mainly intended for system use (it is called whenever a program terminates), but it may be of use to application programs e.g. to report the result of RUNPROG.

Note that the current output stream may be changed by this procedure.

Users may wish to write their own versions of the procedure, e.g. to generate text messages for some or all of the system error codes.

ERRORMSG is declared in SYSHDR.

Portability:
Specific to this implementation.

EXTSFILE

Purpose:
To extend a store file with data contained in a vector.

Examples:
```
(1) is.successful := EXTSFILE( name, vector)
(2) v := GETVEC(30)        // any size >= 9 OK
    MOVE(data, v+2, 8)     // 16 bytes of data
    v!1 := 16
    is.successful := EXTSFILE("myfile", v)
```

Function:
Name is a string which is taken to be the name of a store file (thus "/F.myfile" is not interpreted as a file on the current filing system but as a store file called '/F.myfile').

If the file does not exist it is created and **vector** is used as the first block. If the file already exists then **vector** is appended as the last block. In both cases **vector** is truncated if it contains unused words at the end and the space is returned to the heap.

The result is TRUE if the append succeeds, FALSE if it fails. In the latter case RESULT2 contains an error code.

Vector must be set up as follows:

word 0	not used;
word 1	number of bytes of data in the vector;
words 2 on	data.

Remarks:

This procedure is convenient for adding data to the end of an existing store file. (The only other way to extend a file is to open it for read, copy it completely to a new file, write the new data, delete the old file and rename the new file to the name of the old file.)

The procedure uses **vector** as a block in the file created and so once EXTSFILE has been called **vector** must not be used again by the program. In particular it must not be FREEVECed.

Vector must be a heap vector i.e. it must have been obtained by GETVEC. If EXTSFILE is called with a vector that is not a heap vector, or with a vector containing an invalid byte count (word 1 negative or greater than the number of bytes of data that can be held in the vector) it calls ABORT.

A file that is open or linked cannot be extended.

See the procedures VECTOFILE and SAVEVEC for other ways of creating files from vectors.

EXTSFILE is declared in SYSHDR.

Portability:

Specific to this implementation.

FILETOVEC

Purpose:

To convert a file into a vector. If the file is a store file it is deleted by the conversion.

Examples:

```
vector := FILETOVEC( string)
vector := FILETOVEC("/F.fileb")
```

Function:

String is interpreted in the normal way. Thus "/S.myfile" refers to a store file, "/F.myfile" to a current filing system file and "myfile" to a store file if there is one else a current filing system file.

If **string** refers to a store file then that file is converted into a vector if possible. This results in the deletion of the store file. If **string** refers to a file on the current filing system then a vector is allocated and the file is read into it.

The result of the procedure is the address of the vector or 0 for failure. In the latter case RESULT2 contains an error code. The format of **vector** is as follows:

word 0	not used;
word 1	number of bytes of data in the vector;
words 2 on	data.

Remarks:

The vector allocated is just big enough to contain all the data and so the program must not attempt to write past the end of the data in the vector. It is the responsibility of the program to free the vector (by FREEVEC) when it has finished with it (although the system frees all vectors when the program terminates).

String must specify either a store file or a current filing system file. Other devices (e.g. '/P') are not allowed. The file must not be open or linked.

To read a store file into a vector without deleting the file READVEC should be used.

Portability:
Specific to this implementation.

FINDARG

Purpose:
To find if a given string is one of a small number of specified keywords.

Examples:
```
argumentno := FINDARG( keys, string)
dev := FINDARG("N,C,K,L,P,F,S", devname)
```

Function:
The **keys** are specified in a string, separated by commas. If the **string** matches one of the **keys** the result returned is the argument number, starting with 0 for the first argument. If no match is found the result is -1.

Remarks:
All comparisons ignore the differences between upper and lower case. Thus a **key** "abCD" would be matched by the **string** "AbcD".

A **key** may be specified with two or more synonyms, which are separated by '=' signs. e.g.

```
comp := FINDARG("EQUAL=EQU=EQ,NOTEQUAL=NE",
                string)
```

In this example the **string** "EQU" would make **comp** 0, while the **string** "NOTEQUAL" would make **comp** 1.

Each **key** may also be qualified, normally for other purposes, if the qualifiers follow the **key** after a '/'. Thus the following call might arise from RDARGS.

```
comp := FINDARG("FROM/A,,,,,,,,TO,CHECKING/S",
                string)
```

Such qualifiers are ignored in the comparisons made by FINDARG.

Portability:

This procedure is provided on other BCPL systems.

FINDINPUT

Purpose:

To initialise a stream for reading. If the string specifies a file this corresponds to opening the file for input.

Examples:

```
stream := FINDINPUT( string)
consoleinput := FINDINPUT( "/C")
sysinput := FINDINPUT( "myfile")
discinputb := FINDINPUT( "/F.fileb")
```

Function:

The **string** identifies the stream to the BCPL system. The result of the function is a value which represents the stream and may be used by SELECTINPUT.

Remarks:

String is interpreted in the normal way. Thus "/S.myfile" refers to a store file, "/F.myfile" to a current filing system file and "myfile" to a store file if there is one else a current filing system file.

If the stream cannot be initialised for any reason, the value 0 is returned and an error code is given in RESULT2.

Store files which are open for output or are linked cannot be opened for input. A store file can be opened for input even if it is already open for input however.

Portability:

Standard BCPL procedure, but the format of stream names is specific to this system.

FINDOUTPUT

Purpose:

To initialise a stream for writing. If the string specifies a file this corresponds to opening the file for output.

Examples:

```
stream := FINDOUTPUT( string)
consoleoutput := FINDOUTPUT( "/C")
sysoutput := FINDOUTPUT( "myfile")
discoutputb := FINDOUTPUT( "/F.fileb")
```

Function:

The **string** identifies the stream to the BCPL system. The result of the function is a value which represents the stream and may be used by SELECTOUTPUT.

Remarks:

String is interpreted in the normal way. Thus "/S.myfile" and "myfile" both refer to a store file and "/F.myfile" refers to a current filing system file.

If the stream cannot be initialised for any reason, the value 0 is returned and an error code is given in RESULT2.

Store files that are linked or are already open cannot be opened for output. If a store file that already exists is opened for output then the existing file is deleted.

Portability:

Standard BCPL procedure, but the format of stream names is specific to this system.

FINDXINPUT

Purpose:

To open a file on a specific device for input.

Examples:

```
(1) stream := FINDXINPUT( string, device)
(2) stinp := FINDXINPUT( "/F.myfile", DV.S)
(3) fsinputb := FINDXINPUT( "fileb", DV.F)
```

Function:

The **string** is treated as a ‘pure’ file name i.e. it is not checked for starting with a device specifier. **Device** must be either ‘DV.F’ to specify the current filing system or ‘DV.S’ to specify the store filing system. The result of the function is a value which represents the stream and may be used by SELECTINPUT.

Remarks:

This procedure is useful for opening a file when the device is known, without having to manipulate filename strings to include a device prefix. Note that example (2) above opens a store file named '/F.myfile' whereas FINDINPUT("/F.myfile") opens a current filing system file named 'myfile'.

The manifest constants 'DV.F' and 'DV.S' are declared in SYSHDR. Using any other values for the second parameter may give undefined results.

If the stream cannot be initialised for any reason, the value 0 is returned and an error code is given in RESULT2.

Store files which are open for output or are linked cannot be opened for input. A store file can be opened for input even if it is already open for input however.

FINDXINPUT is declared in SYSHDR.

Portability:

Specific to this implementation.

FINDXOUTPUT

Purpose:

To open a file on a specific device for output.

Examples:

```
(1) stream := FINDXOUTPUT( string, device)
(2) stinp := FINDXOUTPUT( "/F.myfile", DV.S)
(3) fsinputb := FINDXOUTPUT( "fileb", DV.F)
```

Function:

The **string** is treated as a 'pure' file name i.e. it is not checked for starting with a device specifier. **Device** must be either 'DV.F' to specify the current filing system or 'DV.S' to specify the store filing system. The result of the function is a value which represents the stream and may be used by SELECTOUTPUT.

Remarks:

This procedure is useful for opening a file when the device is known, without having to manipulate filename strings to include a device prefix. Note that example (2) above opens a store file named '/F.myfile' whereas FINDOUTPUT("/F.myfile") opens a current filing system file named 'myfile'.

The manifest constants 'DV.F' and 'DV.S' are declared in SYSHDR. Using any other values for the second parameter may give undefined results.

If the stream cannot be initialised for any reason, the value 0 is returned and an error code is given in RESULT2.

Store files that are linked or are already open cannot be opened for output. If a store file that already exists is opened for output then the existing file is deleted.

FINDXOUTPUT is declared in SYSHDR.

Portability:
Specific to this implementation.

FREEVEC

Purpose:
To return a vector obtained by GETVEC to free store.

Example:
FREEVEC(vector)

Remarks:
FREEVEC will accept a parameter 0 without complaint. It is thus possible to use a common re-initialising routine which returns all possible vectors, provided the vector pointers are initialised to zero. If vector is neither zero nor a valid allocated vector, FREEVEC calls ABORT.

In this implementation all vectors obtained by a program are automatically returned to free store when the program terminates. Thus programs specific to the BBC Microcomputer will not often need to call FREEVEC.

Portability:
Part of extension A7 to the standard language.

FSTYPE

Purpose:

To provide information on the characteristics of the current filing system.

Examples:

```
characteristic := FSTYPE( mask)
osgbpb.avail := FSTYPE(1)
```

Function:

The current filing system number is determined by calling the operating system routine OSARGS with the A and Y registers both 0. This number is used to access a set of flags indicating the filing system characteristics and these flags are copied into the global MCRESULT.

The result is TRUE if one or more of the flags set in **mask** is true for the current filing system and is FALSE otherwise.

The meanings of the flags are as follows:

1	the filing system supports the OSGBPB routine to read/write a block of bytes;
2	the OSFILE routine can be used to find the length of a file;
4	the console should be the operating system current output device during all calls to file system routines (i.e. the file routines may generate messages to be displayed);
8	the filing system is the cassette filing system;
16	the filing system is the ROM filing system;

32 the BCPL system should display 'opening file xxx' messages when opening files.

Remarks:

This procedure is intended for use by the system to determine the actions it should take when opening/accessing files on the current filing system. It is not anticipated that application programs will need to call this procedure.

The version of this procedure included with the system caters for tape, ROM, disk and Econet filing systems. If other filing systems are added then the user should write a version of this procedure to return the characteristics of such filing systems.

The values returned by the current version are:

Filing System	**Filing System Number**	**Flags**
Tape	1, 2	4 + 8 + 32 (=44)
ROM	3	16
Disk	4	1 + 2 (=3)
Econet	5	1 + 2 (=3)
Others		0

FSTYPE is declared in SYSHDR.

Portability:

Specific to this implementation.

GETBYTE

Purpose:
To obtain a byte from a string.

Example:
byte := GETBYTE(string, byteposition)

Remarks:
Since the introduction of the % operator the function of GETBYTE is better provided by:

byte := string%byteposition

Thus GETBYTE is provided as source code in the file OPT. It has not been included in the standard header files LIBHDR and SYSHDR, and is expected to be useful mainly for conversion of older programs.

Portability:
A standard procedure.

GETVEC

Purpose:
To obtain a vector from free store.

Examples:
vector := GETVEC(vectorsize)
datastore := GETVEC(1000)

Function:
This procedure obtains a contiguous area of store of **vectorsize** + 1 words, (so that **vector**!0 and **vector!vectorsize** are available).

GETVEC returns a pointer to the vector address, i.e. the first word. If enough store is not available GETVEC calls SHUFFLE to re-arrange the store files then tries again. If there is still not enough room it calls SHUFFLE to delete all unprotected store files and re-arrange those remaining then tries once more. If it has still not succeeded the action taken depends on the system state.

If the system is in the run state and trapping on GETVEC failure has not been disabled in SYSINIT, GETVEC calls TRAP which causes the message

nnn+ store needed

to be displayed (where nnn is the GETVEC parameter). The user may then free up store (e.g. by deleting a file) and resume the program by CONT which causes GETVEC to start again as if it had just been called.

If the system is in the trap state or if trapping on GETVEC failure has been disabled then a result of 0 is returned and RESULT2 is set to 51.

Remarks:

The free store is organised in the area known as the heap. If GETVEC finds the organisation of this area is corrupt it calls ABORT.

If the parameter to GETVEC is negative or greater than 32763 then ABORT is called.

Note that run state programs which do not disable trapping on GETVEC failure (i.e. most application programs) may always assume that GETVEC has succeeded. Thus they need never check for a result of 0.

The procedure MAXVEC may be used to check the largest vector available before calling GETVEC.

Portability:
Part of extension A7 of the standard language.

GLOBIN

Purpose:
To link already loaded CINTCODE into the global vector.

Examples:
```
is.successful := GLOBIN( segment)
GLOBIN( LOADSEG( codefile))
```

Function:
The addresses of any global procedures in **segment** are entered in the global vector. Unless the segment is already linked all machine code hunks are relocated.

The result is TRUE for success, FALSE for failure. In the latter case RESULT2 contains the error code.

Remarks:
A newly linked procedure replaces any existing procedure using the same global number. Thus it is possible to redefine library procedures.

A file that has been linked by GLOBIN cannot be moved in store by SHUFFLE.

Portability:
This function is found on other BCPL implementations. However, the relocation of machine code is specific to this implementation.

GLOBUNIN

Purpose:

To unlink a code segment.

Example:

```
next.segment := GLOBUNIN( segment)
```

Function:

Global links to the **segment** which were set by GLOBIN are cleared. All machine code hunks are unrelocated.

The result is the next segment in the chain of linked files or 0 if this is the last segment in the chain. Note that files linked with the built in command 'LINK file SYSTEM' do not appear in the linked files chain.

GLOBUNIN for a file which is not linked has no effect (except to change RESULT2).

Remarks:

GLOBUNIN does not affect any links to the segment which have been made by the program. Thus it cannot, by itself, ensure that the segment can be unloaded safely. Leaving links to an unloaded segment allows a class of error which is intermittent and difficult to find.

When a machine code hunk is relocated by GLOBIN the contents of the hunk are changed (words containing offsets are updated to contain actual addresses). Unrelocating machine code restores the changed words to their original values so that the next GLOBIN works correctly.

A file that has been unlinked by GLOBUNIN may be moved in store by SHUFFLE.

GLOBUNIN does not restore the previous setting of a global. Thus if a segment redefines a procedure in the program or library it may be necessary to replace the earlier definition. If for example the file ‘newwritef’ redefined WRITEF, the following sequence might be used:

```
saved.writef := WRITEF
segment := LOADSEG( "newwritef")
GLOBIN( segment)

// code using new version of writef

WRITEF := saved.writef
GLOBUNIN( segment)
```

Alternatively where appropriate GLOBIN can be used to re-establish the globals of the original library by GLOBIN(LIBBASE). LIBBASE is defined in SYSHDR.

Portability:
Available on some other BCPL implementations. However, the handling of machine code is specific to this implementation.

INPUT

Purpose:
To obtain the identity of the current input stream.

Example:
stream := INPUT()

Remarks:
If there is no current input stream INPUT returns ERRORSTREAM. Any attempt to read from ERRORSTREAM causes an ABORT.

Portability:
Standard BCPL procedure.

LEVEL

Purpose:

To obtain a pointer to the current stack position for possible use by LONGJUMP.

Example:

```
stackpointer := LEVEL()
```

Portability:

Standard BCPL procedure.

LOADSEG

Purpose:

To bring a CINTCODE file into store (or, if it is already in store, to re-arrange it if necessary) so that it can be linked (by GLOBIN) into the global vector and executed.

Examples:

```
(1) segment := LOADSEG( string)
(2) is.successful := GLOBIN( LOADSEG( string))
(3) segment := LOADSEG( "/F.myfile")
```

Function:

String is interpreted as a filename in the normal way. Thus "/S.myfile" refers to a store file, "/F.myfile" to a current filing system file and "myfile" to a store file if there is one else a current filing system file.

The specified file is accessed. If it is found and is not in store then it is copied to store. (The store file created is given the name **string** with any device specifier omitted. Thus

LOADSEG("/F.myfile")

and

LOADSEG("myfile")

both create store files called ‘myfile’.) If it is already in store then it is reformatted if necessary into the loaded format.

Unless applied to a protected store file the store file created is unprotected.

The result is 0 for failure (in which case RESULT2 contains the error code) or a pointer to the loaded file, which can be used as the parameter to GLOBIN.

Remarks:

As shown in example (2) GLOBIN can accept the result of LOADSEG. If LOADSEG returns zero, GLOBIN returns FALSE but in this case it does not alter RESULT2 so that the cause of the load failure can still be determined.

Portability:

LOADSEG is found in other BCPL implementations.

LONGJUMP

Purpose:

To cause a jump to a higher level procedure, typically after an error or exception condition.

Example:

LONGJUMP(stackpointer, label)

Function:

The new **stackpointer** is established and processing continues from the specified **label**.

Remarks:

Since a label can only be addressed from within the same procedure it is common to place both the stackpointer and the label to be used in globals. For example:

```
ljstackpointer := LEVEL()
ljlabel := errorlabel
         . . .

         . . .

LET errorprocedure( number) BE
$( WRITEF("*NERROR %N*N", number)
   LONGJUMP( ljstackpointer, ljlabel)
$)
```

An alternative to assigning the label to a global is simply to declare it as a global (this is particularly useful if there is only one LONGJUMP destination in the program).

Stackpointer must be in the current stack at a lower address than the current stack pointer. LONGJUMP to an invalid stack pointer may crash the system.

Portability:

Standard BCPL procedure.

MAXVEC

Purpose:

To discover the size of the largest vector obtainable by GETVEC without GETVEC having to SHUFFLE.

Examples:

```
(1) max.free.vector := MAXVEC()
(2) UNLESS MAXVEC() = 0 DO
        vector := GETVEC( MAXVEC())
```

Remarks:

To find the largest vector obtainable without deleting unprotected files precede the call of MAXVEC by SHUFFLE(FALSE).

To find the largest vector obtainable by GETVEC precede the call of MAXVEC by SHUFFLE(TRUE).

A result of 0 from MAXVEC implies that there is no free heap space available.

Portability:

Part of extension A7 to the standard language.

MODE

Purpose:
To select the display mode.

Example:
```
is.successful := MODE( 5)
```

Function:
The bottom of screen memory for the new mode is found and the heap adjusted if possible to end just below this point. If the heap cannot be adjusted (because areas that would become part of the screen memory are in use) the result is FALSE and RESULT2 is set to 15. If the heap is adjusted then the new display mode is selected and the result is TRUE.

Remarks:
If the BCPL system is running in the 2nd 6502 processor (using the Tube) then no adjustment of the heap is necessary.

The parameter must be in the range 0 to 7. If it is not RESULT2 is set to 11 and the result is FALSE.

If the heap has to be shrunk then SHUFFLE is called to delete all unprotected files.

The characteristics of the various modes are:

Mode	**Graphics**	**Colours**	**Text**	**Space (words)**
0	640x256	2	80x32	10240
1	320x256	4	40x32	10240
2	160x256	16	20x32	10240
3	-	2	80x25	8192
4	320x256	2	40x32	5120
5	160x256	4	20x32	5120
6	-	2	40x25	4096
7	Teletext	Display		512

Portability:
Specific to this implementation.

MOVE

Purpose:

To provide a rapid transfer of a block of words from one store location to another, or to initialise store.

Example:

```
MOVE( from, to, words)
```

Remarks:

The move starts with the first word in the block, and the two blocks can overlap. Thus MOVE can be used to initialise a buffer as follows:

```
buffer := GETVEC( buffersize)
buffer!0 := 0  // or any other value
MOVE( buffer, buffer+l, buffersize)

// the buffer will contain buffersize+l
// zeroes
```

If it is desired to move a block of memory upwards, with the blocks overlapping but leaving the contents unchanged, then BACKMOVE should be used.

If **words** is 0 the procedure works as expected (i.e. does nothing). If words is negative the result is unpredictable.

Portability:

Specific to this implementation.

MOVEBYTE

Purpose:

To provide a rapid transfer of a block of bytes from one store location to another.

Example:

```
MOVEBYTE( frombyte, tobyte, bytes)
```

Remarks:

The move starts with the first byte in the block. If the two blocks overlap then MOVEBYTE should be used for moving down in store and BACKMVBY for moving up in store. MOVEBYTE can be used to initialise a string in a similar way to MOVE.

If **bytes** is 0 the procedure works as expected (i.e. does nothing). If bytes is negative the result is unpredictable.

MOVEBYTE is declared in SYSHDR.

Portability:

Specific to this implementation.

MULDIV

Purpose:

To allow scaled or multiple length arithmetic.

Example:

```
result := MULDIV( a, b, c)
```

Function:

MULDIV calculates (a*b)/c, holding the intermediate product (a*b) as a double length integer. The remainder from the division is left in global variable RESULT2.

Dividing by zero causes a fatal trap.

Remarks:

If the result does not fit in a normal integer the results returned by MULDIV are not defined.

Portability:

Extension A8 of the standard language.

NEWLINE

Purpose:

To start a new line in the current output stream.

Example:

```
NEWLINE()
```

Remarks:

In this implementation it places a line feed character (0A hex) followed by a carriage return character (0D hex) in the output stream.

Portability:

A standard BCPL procedure.

NEWPAGE

Purpose:

To start a new page in the current output stream.

Example:

```
NEWPAGE()
```

Remarks:

A form feed character (0C hex) is placed in the current output stream.

Portability:

A standard BCPL procedure.

OPSYS

Purpose:

To give access to operating system services.

Examples:

```
value := OPSYS( functionnumber, x, y)
OPSYS( 8, 4) // set TX baud rate to 1200
```

Function:

In this implementation the function calls the operating system procedure OSBYTE. The **functionnumber** is placed in register A, **x** in register X, and **y** in register Y. The return is obtained from registers Y and X with X holding the least significant byte. The global MCRESULT contains the flags in the most significant byte and register A in the least significant byte.

If a fault has occurred then the high byte of MCRESULT is set to #XFF, and the low byte is the fault number. In this case value is undefined.

Remarks:

Calls to other operating system entry points such as OSASCI should use the procedure CALLBYTE.

Portability:

Part of standard language extension Al2. However the direct use of the operating system in this way reduces portability to other operating systems.

See also:

BBC Microcomputer User Guide.

OUTPUT

Purpose:
To obtain the identity of the current output stream.

Example:
```
stream := OUTPUT()
```

Remarks:
If there is no current output stream OUTPUT returns ERRORSTREAM. Any attempt to write to ERRORSTREAM causes an ABORT.

Portability:
Standard BCPL procedure.

PACKSTRING

Purpose:
To convert a vector of characters into a string.

Example:
```
string.words := PACKSTRING( fromunpacked,
                            tostring)
```

Function:
The length (n) of the string is given by **fromunpacked**!0. The characters stored in **fromunpacked**!1 to **fromunpacked**!n are placed in **tostring**%1 to **tostring**%n and the length n is placed in **tostring**%0. The result is the size of **tostring** in words decreased by one. Thus a one word string returns zero.

Remarks:

PACKSTR1NG is primarily intended for use in converting existing BCPL programs. The availability of the % operator makes the use of vectors for characters unnecessary. PACKSTRING is therefore provided in source form in OPT and does not have an address in the standard header files LIBHDR and SYSHDR.

Portability:

Standard BCPL procedure.

PUTBYTE

Purpose:

To place a byte in a string.

Example:

PUTBYTE(string, byteposition, byte)

Remarks:

Since the introduction of the % operator the function of PUTBYTE is better provided by:

string%byteposition := byte

Thus PUTBYTE is provided in source form in OPT and does not have an address in the standard header files LIBHDR and SYSHDR. It is expected to be useful mainly for conversion of older programs.

Portability:

Standard BCPL procedure.

RANDOM

Purpose:

To generate a series of random numbers.

Example:

```
randominteger := RANDOM( randominteger)
```

Function:

The result is obtained from the given parameter by an algorithm which generates an approximately random sequence of numbers, which repeats only after every possible number (both positive and negative) has been generated.

Remarks:

If a series of calls to RANDOM is always started with the same seed number, they will generate the same sequence of 'random' numbers. This may be useful in testing alternative strategies with random data. If only a few bits of random data are required, it is best to use the more significant part of the number, since, for example, the least significant bit is odd and even alternately. A random starting number can usually be obtained by a measure of the time between console key depressions. This requires a point in the program at which it is known that all operator input has been read, and that further operator input is awaited. The seed number is then obtained by:

```
seed := seed+1 REPEATUNTIL
                 TESTFLAGS(CONSOLE.KEY)
```

Portability:

Similar procedures are provided on many implementations.

RDARGS

Purpose:

To perform an initial analysis and verification of the arguments for a utility.

Examples:

```
(1) endofargs := RDARGS( keys, argumentvector,
                             size)
(2) RDARGS( "FROM/A,TO,CHECKING/S", argv, 100)
(3) RDARGS( "FROM,,,TO=AS", argv, size)
```

Function:

One line of input is read and analysed for conformity with the specified **keys.** The individual arguments in the line are separated and placed in **argumentvector**, provided that the space taken in the vector does not exceed **size.** **Size** is the size of **argumentvector** in words (i.e. the GETVEC or VEC parameter).

<u>Keys</u>

The **keys** are specified in a string, with each argument being separated by a comma. An argument may have a keyword, e.g. FROM, TO and CHECKING in example (2). It may also be given alternative keywords, separated by '=', for example TO=AS in (3). Each argument in the **keys** may be qualified by one or more of the following:

/A This argument must be present.

/K This argument is only present if the keyword is present.

/S A state argument, TRUE if the keyword is present and FALSE otherwise.

The input

The following are examples of valid input:

For example (2):

```
FROM a TO b
a b
CHECKING a
```

For example (3):

```
FROM a b AS z
TO z FROM a b c
TO=z FROM=a
TO z FROM "FROM"
a b c z
```

Input arguments qualified by keywords can be accepted in any order, and are normally separated by spaces. Keywords may be linked to their associated arguments by '=' or by a space. If an argument is the same as a keyword, the argument must be placed in double quotes. Double quotes must also be used round arguments containing spaces.

Arguments not following keywords provide successive arguments in the list, omitting those that must have keywords, (qualifiers /K and /S). Thus the input

```
b FROM a
```

for example (2) would fail, since **b** would be taken as the argument for the FROM keyword.

If a user at a console enters **?** the procedure prompts on the console with the required **keys.** The user may then enter the arguments in the usual way.

The argument vector

RDARGS creates an **argumentvector** in the following form. One word is reserved for each argument specified in the keys, starting with **argumentvector**!0 and the remainder of the vector is used to hold any arguments discovered as character strings. The argument word is zero, (FALSE), if the argument is not present. It is TRUE if a state argument is present. Otherwise it points to the string containing the argument.

The result

Normally the result points to the first word not used in the **argumentvector.** The result returned is zero if an unsuitable set of parameters was provided, or if the arguments found would not fit into the **size** of the **argumentvector.**

Remarks:

RDARGS is effective in analysing the arguments given but it does not convert any arguments to an internal form such as a number. Such conversions are not difficult in BCPL but the variety of requirements makes this stage of validation appropriate to individual programs.

Portability:

This procedure is provided on other BCPL implementations.

RDBIN

Purpose:
To read binary data from an input stream.

Example:
```
bincharacter := RDBIN()
```

Function:
The procedure returns the next character from the current input stream, unless the input stream is exhausted when it returns ENDSTREAMCH.

Portability:
Part of extension A10b) of the standard language.

RDCH

Purpose:
To read a character from the input stream.

Example:
```
character := RDCH()
```

Function:
The next character from the current input stream is returned. The carriage return line feed pair are converted to the single character '*N'. (Specifically CR is treated as '*N' and LF is ignored). The end of stream is indicated by ENDSTREAMCH. The character 135 is also treated as ENDSTREAMCH. This character may be obtained from the keyboard by disabling cursor editing (*FX 4,1) and pressing **COPY**. Cursor editing may be re-enabled by *FX 4,0.

Only the bottom seven bits of the character are returned. Thus RDCH can be used on input streams which use bit 7 for parity or other purposes (but note that character 135, which has bit 7 set, is treated as ENDSTREAMCH).

Portability:
Standard BCPL procedure.

RDITEM

Purpose:
To read the next string from the current input stream.

Example:
itemtype := RDITEM(tovector, maximumsize)

Function:
The next argument or tag in the input stream is placed as a string in **tovector**, of size **maximumsize** words. The **itemtypes** returned are:

-3	for the end of the input stream;
-2	for an '=' symbol;
-1	for an invalid item, e.g. if larger than **maximumsize**;
0	for a newline, '*E', or semicolon;
1	for an unquoted item;
2	for an item enclosed in double quotes.

Remarks:
Leading spaces are suppressed.

<u>Unquoted items</u>

These are terminated by space, semicolon, '=', newline, or the end of the input stream. The stream is left so that the next character read is the terminating character.

Quoted items

These start with a double quote character (") and can include any character except a newline character up to the next double quote character. Any *N or *E in the text is converted to its internal single character representation. The string returned does not include the quote characters. The stream is left so that the next character read will be that following the second double quote.

Portability:
Similar procedures are available in other BCPL implementations.

READ

Purpose:
To read a file from the current filing system into store or to copy a store file to another store file or to make a store file contiguous.

Examples:

```
(1) is.successful := READ( fromfile, tofile,
                           contiguous)
(2) READ( "myfile", 0, TRUE)
(3) is.successful := READ( "/F.myfile", "xxx",
                           FALSE)
(4) is.contig := READ( "sfile", "sfile", TRUE)
(5) READ( "file1", "file2", FALSE)
(6) READ( "/S.xxx", 0, FALSE)
(7) READ( "file1", "/F.file", FALSE)
```

Function:

There are two main functions of this routine:

- to read a file efficiently from the current filing system into store without changing the file name;

- to copy a file (from the current filing system or from store) to a store file with a different name.

Both functions have the option of forcing the store file created to be contiguous (see 'Remarks' below for a discussion on the advantages of contiguous store files).

A subsidiary function enables a non-contiguous store file to be made contiguous.

In all cases the result is TRUE if the procedure succeeds and FALSE if it fails. In the latter case RESULT2 contains the error code.

<u>Read file into store</u>

This function is selected by specifying **tofile** as 0.

Fromfile is interpreted as a 'pure' file name i.e. it is not processed to see if it contains a device specifier. The specified file is copied from the current filing system to a store file of the same name (overwriting any existing store file of that name).

Examples (2) and (6) show the use of this function. Example (6) copies a file named '/S.xxx' from the current filing system to store, resulting in a store file named '/S.xxx'.

Contiguous is TRUE if the store file created must be a contiguous file, FALSE if it need not be contiguous.

Copy file to store file

This function is selected by specifying **tofile** as non-zero.

Fromfile is interpreted in the normal way. Thus "/S.myfile" refers to a store file, "/F.myfile" to a current filing system file and "myfile" to a store file if there is one else a current filing system file. The only devices allowed are store and the current filing system - thus specifying **fromfile** as "/P", for example, would be an error.

Tofile is interpreted as a 'pure' file name. Thus specifying "/S.myfile" would copy **fromfile** to a store file called '/S.myfile'.

Examples (3), (5) and (7) demonstrate this. Example (3) copies file 'myfile' in the current filing system to the store file 'XXX'. Example (5) copies the file 'file1' from store (if it exists there) or from the current filing system to the store file 'file2'. Example (7) copies the same file to the store file '/F.file1'.

Contiguous is TRUE if the store file created must be a contiguous file, FALSE if it need not be contiguous.

Make store file contiguous

This option is selected by specifying **fromfile** and **tofile** as the same store file and specifying **contiguous** as TRUE.

This is demonstrated by example (4). Assuming that the store file 'sfile' exists it makes the file contiguous, doing nothing if the file is already contiguous. If **contiguous** were FALSE in this example then the procedure would only make the file contiguous if there was room but would return TRUE in any case. If 'sfile' did not exist in store then the effect would be to copy 'sfile' from the current filing system to store.

Remarks:

When copying a file from the current filing system into store the operating system OSFILE routine is used if possible (i.e. if the size of the file can be obtained without reading the file and if a sufficiently large vector can be obtained to read it into). Use of this routine results in a faster copy than any other method.

READ calls SHUFFLE if necessary to create sufficient heap space, and may therefore delete unprotected files. If it cannot obtain sufficient heap space it returns FALSE and sets RESULT2 to 51. Note that, unlike GETVEC, it does not trap with the 'nnn+ store needed' message.

The advantages of forcing the new store file to be contiguous are:

- if the file is subsequently to be loaded or converted to a vector (by FILETOVEC) then it is already in the right format;

- if the file is subsequently to be copied to the current filing system by SAVE then it is in the right format to enable the fast OSFILE routine to be used;

- if reading from a filing system which enables the file length to be found (e.g. disk, Econet), it guarantees that the fast OSFILE routine will be used.

The disadvantage of forcing the new store file to be contiguous is that the READ may fail because there is not enough contiguous heap space when it would have succeeded in creating the file as non-contiguous.

Portability:
Specific to this implementation.

READN

Purpose:
To read a decimal integer from the current input stream.

Example:
```
integer := READN()
```

Remarks:
Initial spaces, tabs or newlines are ignored. An initial sign is optional. The integer is terminated by the first non-decimal character. If the integer is greater than the maximum or less than the minimum value that can be held in one word the result is undefined.

The next character read will be the first character following the number. There are no error returns.

Portability:
Standard BCPL procedure.

READVEC

Purpose:
To read a file into a given vector.

Examples:

```
(1) is.successful := READVEC( fromfile, tovec,
                               vecsize)
(2) v := GETVEC(20)
    is.successful := READVEC( "myfile", v, 20)
```

Function:

Fromfile is interpreted in the normal way. Thus "/S.myfile" refers to a store file, "/F.myfile" to a current filing system file and "myfile" to a store file if there is one else a current filing system file. The only devices allowed are store and the current filing system - thus specifying **fromfile** as "/P", for example, would be an error.

Vecsize is the size of **tovec** (in the form of a GETVEC parameter i.e. **tovec** has **vecsize**+1 words).

If **fromfile** exists and the specified vector is big enough to contain it then it is copied into the vector. If the copy fails the result is FALSE and RESULT2 contains an error code. If the copy succeeds the result is TRUE and **tovec** is set up as follows:

word 0	not used;
word 1	number of bytes of data in the vector;
words 2 on	data.

Remarks:

If the result is FALSE then the contents of **tovec** are undefined. In particular the byte count in word 1 will not have been set up. Depending on the circumstances the original contents of the vector may or may not have been changed.

If the file is too big to fit in **tovec** RESULT2 is set to 51.

Tovec need not be a heap vector. It may be any data area, provided that **vecsize** is set up appropriately. For example, the following code is quite valid:

```
LET V = VEC 30
READVEC( "file", V+9, 21)
```

Fromfile is not affected by this procedure, even if it is a store file.

Portability:
Specific to this implementation.

READWORDS

Purpose:
To read a number of words of data from the input stream.

Example:
wordsread := READWORDS(destination, words)

Function:
Binary data is read from the current input stream until either the specified words have been read or the stream is exhausted. The result is the actual number of words read.

Remarks:
Using READWORDS is more efficient than repeated use of RDBIN. This is particularly so when reading from a filing system that supports the operating system OSGBPB routine (read/write block of bytes).

When using READWORDS to read until a stream is exhausted a final call of RDBIN should be made in case the stream contained an odd number of bytes.

The use of READWORDS may be combined with the use of RDBIN and RDCH, but a call to UNRDCH immediately following a call to READWORDS may not work.

It is possible to use READWORDS on any input stream.

Portability:

The procedure is provided on some other BCPL implementations.

RENAME

Purpose:

To rename a file on the current filing system or in store.

Examples:

```
RENAME( fromfilename, tofilename)
is.successful := RENAME( "tempfile", "final")
RENAME( "/F.file1", "file2")
```

Function:

If the file **fromfilename** can be found, it is renamed to **tofilename.** The result is TRUE if successful, FALSE if not. In the latter case RESULT2 contains the error code.

Fromfilename may contain a device specifier. The only valid devices are store, the current filing system and the null device. If it does not contain a file specifier then store is assumed.

Tofilename is treated as a 'pure' file name. Thus RENAME("/F.xxx", "/F.yyy") would rename the file 'xxx' in the current filing system to '/F.yyy' rather than to 'yyy'.

Remarks:

For store files, if a file called **tofilename** already exists the rename fails but a file that is open or linked may be renamed.

Not all filing systems support RENAME.

An alternative method of renaming files in the current filing system is to use RUNPROG e.g.

```
RUNPROG( "**RENAME xxx yyy")
```

This may be more convenient than prefixing the name of the first file with '/F.'.

Portability:

Similar procedures are provided on other implementations.

RESUMECO

Purpose:

To transfer control to another coroutine at the same level.

Example:

```
reply := RESUMECO( coroutine, parameter)
```

Function:

If the second coroutine exists and is in a waiting state, it resumes processing and receives the **parameter.** The parent coroutine of the new coroutine becomes the parent of the transferring coroutine.

The transferring coroutine enters the waiting state and may be reactivated by a CALLCO or another RESUMECO. In either case it receives the calling parameter as the **reply** to RESUMECO.

Remarks:

If the second coroutine is not in a suitable state RESUMECO calls ABORT.

RESUMECO is in section "CORTNS" of the CINTCODE library LIB.

Portability:

Part of the published method of implementing coroutines in BCPL.

RUNPROG

Purpose:

To enable a program to run another program or utility or to issue a '*' command to the operating system.

Examples:

```
(1) result :=
    RUNPROG( writefstring, p1, p2, p3, p4)
(2) res := RUNPROG( "TYPE %S", textfile)
(3) RUNPROG( "**TAPE") // select tape filing
                       // system (** must be
                       // used for asterisk
                       // in strings)
```

Function:

A temporary file is created using the parameters as parameters of a call of WRITEF (but note that only 4 parameters apart from the WRITEF string are allowed). The file is terminated with LF/CR.

The first line of the file is then read and processed as if it had been typed in at the console by the user.

If this results in a program being run the program is loaded and is entered (by calling START) with the temporary file as its input stream (the next character read will be that following the program name) and the console as its output stream.

When the program terminates it is unlinked, the temporary file is deleted and control is passed back to the calling routine. **Result** depends on how the program terminated:

- if it terminated by FINISH or by returning from START **result** is 0;

- if it terminated by calling STOP **result** is the parameter of STOP;

- if it terminated by calling ABORT **result** is the parameter of ABORT;

- if the program could not be run (e.g. an invalid program file name was given) **result** is the appropriate RESULT2 error code;

- if the 'program' was actually a '*' command then **result** is 0 if the command succeeded or 1000+fault code if the command caused a fault to be generated.

Remarks:

The procedure ERRORMSG may be useful for printing an error message if the call to RUNPROG fails.

See the section 'RUNPROG' in Chapter 6 for more details on the use of RUNPROG.

Portability:

Developed for this implementation.

SAVE

Purpose:

To copy a store file. A special option is provided for copying a store file to a current filing system file with the same name.

Examples:

```
(1) is.successful := SAVE( fromfile, tofile)
(2) SAVE( "myfile", 0)
(3) is.ok := SAVE ("file1", "/F.file2")
(4) SAVE( "file1", "file1")
(5) SAVE( "/F.file", "/P")
(6) SAVE( "file1", "file2")
```

Function:

The store file **fromfile** is copied either to the file specified by **tofile** or, if **tofile** is 0, to a file named **fromfile** in the current filing system. The result is TRUE if the copy succeeds, FALSE if it fails. In the latter case RESULT2 contains the error code.

The case where **tofile** is 0 is probably the most common case, since it is a simple method of backing up a store file. Thus example (2) copies the store file 'myfile' to a current filing system file called 'myfile'.

Fromfile is interpreted as a 'pure' file name. Thus example (5) copies a store file named '/F.file' to the serial port (device '/P').

If **tofile** is not 0 it is interpreted in the normal way. Thus example (3) copies the store file 'file1' to the filing system file 'file2' whereas example (6) copies the same file to the store file 'file2'.

In all cases **fromfile** is made contiguous if possible. Example (4) is a special case **fromfile** and **tofile** are the same store file. This simply has the effect of making the store file 'file1' contiguous if possible.

Remarks:

Making **fromfile** contiguous (if possible) may involve calling SHUFFLE and hence deleting unprotected store files.

If **fromfile** is successfully made contiguous and is being copied to the current filing system the operating system OSFILE routine is used. Use of this routine is faster than any other method.

To guarantee the use of OSFILE the procedure READ should first be used to force **fromfile** contiguous. If READ fails then appropriate action should be taken (e.g. free more heap space by deleting files). SAVE should only be called after READ has succeeded.

A particularly useful feature of SAVE is that it will work even if the heap is completely full, whereas other methods of copying files (e.g. normal stream I/O) would fail in such a situation through lack of heap space.

Portability:

Specific to this implementation.

SAVEVEC

Purpose:

To write the contents of a vector as a file.

Examples:

```
(1) is.successful := SAVEVEC( fromvec, tofile)
(2) LET v = VEC 10
    MOVE( data, v+2, 9)
    v!1 := 17    // 17 bytes of data
    SAVEVEC( v, "/F.myfile")
(3) SAVEVEC( myvec, "/P")
```

Function:

The data in **fromvec** is written to the file **tofile.** If the file already exists it is overwritten. The result is TRUE for success, FALSE for failure. In the latter case RESULT2 contains the error code.

Fromvec must be set up as follows:

word 0	not used;
word 1	number of bytes of data in the vector;
words 2 on	data.

Tofile is interpreted in the normal way. Thus "/S.myfile" and "myfile" both refer to a store file and "/F.myfile" refers to a current filing system file. Other devices may also be specified (see example (3)).

Remarks:

Fromvec need not be a heap vector. It may be any data area, provided that it is set up in the correct format.

If **tofile** is a store file then SHUFFLE may be called to free up heap space and so unprotected store files may be deleted.

If **tofile** is a store file and there is not enough room in store for it then no file is created and RESULT2 is set to 51 (note that if stream I/O is used to write to a store file then the file is truncated if there is not enough room).

Fromvec is left unchanged by this procedure.

Portability:
Specific to this implementation.

SELECTINPUT

Purpose:
To select the current input stream.

Examples:
```
SELECTINPUT( stream)
SELECTINPUT( FINDINPUT( "/C"))
```

Function:
If **stream** is an identifier, which has been returned by FINDINPUT or FINDXINPUT, the stream is selected for input.

Remarks:
It is possible to reselect the current stream. SELECTINPUT(0) is allowed, and causes selection of the ERRORSTREAM, in which case any read operation will cause an ABORT. If **stream** is not a valid input stream then ABORT is called.

Portability:
Standard BCPL procedure.

SELECTOUTPUT

Purpose:
To select the current output stream.

Examples:
```
SELECTOUTPUT( stream)
SELECTOUTPUT( FINDOUTPUT( "/C"))
```

Function:
If **stream** is an identifier, which has been returned by FINDOUTPUT or FINDXOUTPUT, the stream is selected for output.

Remarks:
It is possible to reselect the current stream. SELECTOUTPUT(0) is allowed, and causes selection of the ERRORSTREAM, in which case any write operation will cause an ABORT. If **stream** is not a valid output stream then ABORT is called.

Portability:
Standard BCPL procedure.

SHUFFLE

Purpose:

To rearrange the store files so as to maximise the contiguous free heap space.

Examples:

```
SHUFFLE(TRUE)
SHUFFLE(FALSE)
```

Function:

If the parameter is TRUE all unprotected store files are deleted.

Whether the parameter is TRUE or FALSE all store files except those that are linked are rearranged (by shuffling down towards the bottom of the heap) as follows:

(1) the first unallocated area in the heap is found;

(2) if the allocated area above this is movable (i.e. if it is part of an unlinked file) then it is moved down into the unallocated area. There is now an unallocated area immediately above the area that has been moved down and the process is repeated for this area;

(3) if the allocated area above the current unallocated area cannot be moved then the heap is searched (working upwards) for the first allocated area that can be moved and that will fit in the unallocated area. If one is found it is moved to the bottom of the unallocated area;

(4) if nothing could be moved to the unallocated area, or the area was completely filled, then the process is repeated for the next unallocated area (working up the heap). If the unallocated area was not completely filled the process is repeated for the remaining unallocated part of this area.

The effect is to make a large contiguous area available at the top of the heap.

Remarks:

The procedure MAXVEC may be used to find the size of the largest contiguous free area.

SHUFFLE is declared in the header file SYSHDR.

Portability:

Specific to this implementation.

SOUND

Purpose:

To generate a sound.

Example:

```
LET stab = TABLE 1, -15, 53, 20
SOUND( stab)
```

Function:

The parameter is a four-word table or vector containing the four parameters defining a sound. The table is passed to the operating system using the OSWORD call. Details of the sound parameters are given in the BBC Microcomputer User Guide.

Remarks:

This procedure is provided mainly for ease in converting BASIC programs to BCPL. In many cases it will be easier to make the OSWORD call directly using CALLBYTE.

The procedure is contained in section "SOUND" of the CINTCODE library file LIB.

Portability:

Specific to this implementation.

SPLIT

Purpose:

To split off a tag or prefix from a string for separate processing.

Examples:

```
endptr := SPLIT( prefix,char,string,
                 startptr)
endptr := SPLIT( device, '.', name, 0)
```

Function:

The parameters are:

prefix	a vector of 16 words for the result;
char	the terminating character;
string	the string being examined;
startptr	the current byte position in the string.

String is searched from **startptr**+1. Initial spaces are skipped and then all characters up to but not including the terminating character **char** are transferred to a string in **prefix.** The result is the byte position in **string** of the specified character. If **string** does not contain **char** after **startptr**, the result is zero. However in this case the remainder of the string is transferred to **prefix.**

Remarks:

SPLIT can be used to extract successive arguments from a string, provided all arguments are separated by the same character. The last argument will return zero. If the string in **prefix** would exceed 30 characters, it is truncated to 30 characters.

Portability:

While procedures similar to SPLIT are provided in several implementations, there are differences which reduce portability.

STACKSIZE

Purpose:

To determine the free space in the current stack.

Example:

```
stack.size.left := STACKSIZE()
```

Remarks:

One reason for STACKSIZE is to permit the vector requested by APTOVEC to be adjusted to the available space.

STACKSIZE is declared in the header file SYSHDR and included in section "STACKSI" of the CINTCODE library file LIB.

Portability:

Part of extension A7 of standard BCPL.

START

Purpose:
To provide a standard start point for any program.

Examples:

```
LET START() BE ...
START( parameter)
```

Function:
Every program must include a START procedure, and START is the entry point to that program. Returning from the START procedure is one way of ending a program.

Remarks:
In this implementation the **parameter** is always zero when START is called by the BCPL system. START is entered with the console or other command line source selected for input and the console selected for output. Reading the input stream will obtain any text in the command line after the command that loaded the program. If the program is entered from the console, reading after the carriage return will obtain input from the console. If the program is entered using a command file (or by the procedure RUNPROG), subsequent input will be obtained from the command file (or temporary RUNPROG file).

It is not necessary to restrict a program to a single START. Programs can be divided into subprograms each with its own START, and a root program can load and call each subprogram in turn.

Portability:
START is a standard procedure, but different implementations have different ways of passing the input parameters to the procedure.

STARTINIT

Purpose:

To enable a program to control its environment.

Example:

```
LET STARTINIT() = 3000  // sets stack size
```

Function:

If STARTINIT is defined, it is called during the initialisation of a program. STARTINIT must return the stack size required by the program.

Remarks:

STARTINIT is called in a privileged state and can alter a number of features of its environment.

The stack size is the only change frequently required by application programs.

Two other possible uses are to change the default options for handling ESCAPE and for handling GETVEC failures. Typical code might be:

```
LET STARTINIT() = VALOF
$( SYSINDEX!I.RSTATE!R.MCST :=
   SYSINDEX!I.RSTATE!R.MCST &
      NOT M.TRAPESC  // disable trapping
                     // on ESCAPE
   RESULTIS 400  // default stack size
$)
```

To disable trapping on GETVEC failure use M.TRAPGV instead of M.TRAPESC.

STARTINIT is declared in the header file SYSHDR. This header file also declares SYSINDEX, I.RSTATE, R.MCST, M.TRAPESC and M.TRAPGV.

Portability:

Specific to this implementation.

STOP

Purpose:

To end a program.

Examples:

```
STOP( 0)
STOP( returncode)
```

Function:

A call to STOP indicates a controlled completion of a program. The result is normally a return to the command state. By convention a zero return code indicates successful completion, a positive code is a fatal error and a negative code is a warning or comment. Codes below -100, between 501 and 999 and above 1256 are available for applications use.

Remarks:

Normally any non-zero **returncode** is displayed by calling procedure ERRORMSG.

If a program run by a command file returns a positive **returncode**, execution of the command file is stopped unless ERRCONT is in force (see chapter 3).

Portability:

A standard BCPL function. A **returncode** of 0 for success is also standard. The use of other **returncode**s is implementation dependent.

TESTFLAGS

Purpose:

To test if there is more input to read.

Examples:

```
flag.was.set := TESTFLAGS( flagmask)
IF TESTFLAGS( CONSOLE.KEY) THEN
WHILE TESTFLAGS( MORE.INPUT) DO
```

Function:

One or more flags, selected by the parameter **flagmask**, are tested. The result is TRUE if any of the tested flags were set, and the state of all tested flags is placed in RESULT2.

In this implementation there are two possible flags:

CONSOLE.KEY
tests if there are more characters in the operating system's console input buffer.

MORE.INPUT
tests if more input can be read from the current input stream without further calls to the operating system. This test is only useful on a console stream, (device /C).

Remarks:

The test for CONSOLE.KEY is normally used to permit a process to be interrupted by any console key depression. The console can be read in two ways.

When the console is read one line at a time as device /C the test indicates that further input following the end of line has been received. It does not show whether there is further input on the current line.

When the console is being read character by character as device /K the test indicates that another character can be read without pausing for operator input.

When applied to a console stream (/C) the test for MORE.INPUT shows whether there is more input in the current line. For example, the code

```
WHILE TESTFLAGS( MORE.INPUT) DO RDCH()
```

reads the rest of the current line, and may be useful for discarding input on detecting an error. When applied to any other stream the test almost always returns FALSE. (Two cases in which it returns TRUE are if UNRDCH has just been called or if READWORDS has just been called and has read to the end of an odd-length file.)

Note that programs which may be run from command files should not use TESTFLAGS to discard the rest of the current line. A suitable technique is described in the section 'Command files' in chapter 6.

See the section 'File and stream handling' in chapter 8 for a method of determining whether the current input stream is the console.

The manifests CONSOLE.KEY and MORE.INPUT are declared in LIBHDR.

Portability:

The procedure is used in other BCPL implementations, but the meaning of the testflag bits is not fixed. TESTFLAGS(1) is often used to decide to terminate a process in response to some operator input. This can carry over unchanged to this implementation, since CONSOLE.KEY is 1.

TESTSTR

Purpose:

To test the characteristics of the current input and output streams.

Examples:

```
is.streamtype := TESTSTR( streamtype)
UNLESS TESTSTR( STYPE.INT) DO
```

Function:

One or more flags, selected by the parameter **streamtype**, are tested. The result is TRUE if any of the tested flags is set, and the state of all tested flags is placed in RESULT2.

Tests are applied to the current input and output streams. The flag positions for the output stream are in the same order as those for the input stream, but shifted left eight places. The following **streamtype** tests are provided:

STYPE.TERM

tests if the device is a terminal on which the result, after outputting text not terminated by '*N', will be a part line of text permitting input on the same line.

STYPE.INT

tests if the device is interactive and so can respond to prompts from the program.

Remarks:

These tests permit a program to be written to provide a helpful interaction with the operator, while adapting correctly if it receives its input parameters from a command file or other non-interactive device (but the system cannot detect whether 'console' input is coming from the console or from a *EXEC file).

See also the section ‘File and stream handling’ in chapter 8.

TESTSTR is in section "TESTSTR" of the CINTCODE library LIB.

The manifests STYPE.TERM and STYPE.INT are declared in LIBHDR.

Portability:
Similar features are provided in some other implementations.

TIME

Purpose:
To read the 15 least significant bits of the elapsed-time clock.

Example:
```
clock.time := TIME()
```

Function:
The bottom 15 bits of the elapsed-time clock are read. Since each tick is 10ms the resulting time wraps round approximately every five and a half minutes (327.68 seconds).

Remarks:
The manifest TICKSPERSEC declared in LIBHDR may be used to convert time intervals into seconds.

TIME may be used to measure the time between two events or to delay a program for a certain time. In both cases allowance must be made for wraparound, and in both cases the programmer must be aware that the ability to interrupt a program with **ESCAPE** could cause a delay of much longer than five minutes to occur.

Suitable code to adjust for wraparound might be:

```
t1 := time()
. . . // code to be timed
interval := time() - t1
if interval < 0 then
   interval := interval + #X8000
```

If longer intervals are to be timed then the operating system routine OSWORD can be called (using CALLBYTE) to read as many bits of the elapsed-time clock as desired.

TIME is in section "TIME" of the CINTCODE library LIB.

Portability:

Part of extension A14 of the BCPL language.

TRAP

Purpose:

To cause a trap to the command state.

Example:

TRAP(traptype, letter)

Function:

The procedure causes a trap with the given **traptype** and **letter**. All values of **traptype** below 10 are reserved for system use. Application programs may use other values as required.

A call of the form TRAP(35, 'Y') would cause a message of the form

Trap 35 Y

to be displayed and the system to be in the command state. Typing CONT would resume the program.

Remarks:

This procedure may be useful when developing programs - for example a call of TRAP may be included so that the program breaks out to the console when some specific condition occurs. In most cases the normal DEBUG facilities will be sufficient, however.

Two of the system TRAP calls may be of use to application programs. TRAP(4) traps with the message 'Interrupted' and TRAP(6) traps with the message 'Type CONT to resume'. In both cases the second parameter is not used and may be omitted.

The **traptypes** used by the system are:

```
-2  Fatal error;
-1  Abort;
 0  End of program;
 2  Break point reached (DEBUG);
 3  Requested trap (DEBUG);
 4  ESCAPE;
 5  GETVEC failure;
 6  PAUSE.
```

Portability:

Specific to this implementation.

UNLOADSEG

Purpose:

In some systems to return to free store the space used by a loaded segment.

Example:

UNLOADSEG(segment)

Function:

In this system UNLOADSEG is a dummy procedure which returns 0. It is provided for compatibility with systems where it must be called after calling GLOBUNIN to release the space occupied by the segment.

Remarks:

UNLOADSEG is declared in SYSHDR.

Portability:

UNLOADSEG is provided by other BCPL implementations.

UNPACKSTRING

Purpose:

To separate the bytes of a string into separate words.

Example:

```
UNPACKSTRING( fromstring, tounpacked)
```

Function:

The length of the string is placed in **tounpacked**!0 and the characters are placed in order starting at **tounpacked**!1.

Remarks:

See PACKSTRING.

Portability:

Standard BCPL procedure.

UNRDCH

Purpose:

To step back the current input stream, so that the next byte read is the same as the last. This permits a separate part of the program to read the last character read. For example a procedure reading a number must often find the terminator before it can return the number. However, another procedure often needs to examine the terminator, so the first procedure normally ends with a call of UNRDCH.

Example:

```
UNRDCH()
```

Remarks:

UNRDCH should not be called more than once between successive read commands on the input stream. The effect of a second call is undefined, as is the effect of a call on a newly opened stream.

UNRDCH should also not be called immediately after a call to READWORDS.

Portability:

Standard BCPL procedure.

VDU

Purpose:

To write a number of bytes to the VDU driver.

Examples:

```
(1) VDU( string, p1, p2, p3, p4, p5, p6, p7,
                 p8, p9, p10)
(2) VDU( "28, 0, 5, 39, 0")
(3) VDU( "31, %, %", x, y)
(4) VDU( "24,%;%;%;%;", left, bottom,
                        right, top)
(5) VDU( "23,&FD,8,#X1C,28,107,127,107,0;")
```

Function:

String is treated as a list of numbers or '%' signs. Items in the list are separated by ',' or ';' and may be preceded or followed by any number of spaces. Numbers are normally decimal (e.g. 123) but are treated as hexadecimal if prefixed by '&' or '#X' (e.g. &1F, &1f, #X1f, #x1F).

Each item in **string** is processed in turn. If it is a number terminated by ',' (or by the end of the string) it is written as a binary byte using the operating system routine OSWRCH. If it is a number terminated by ';' it is written as a binary word, low byte first (e.g. &1234 is written as byte &34 then byte &12). If it is '%' then the next parameter of VDU is written as a byte (if '%' is followed by ',' or the end of the string) or a word (if '%' is followed by ';').

Up to ten parameters after the string may be specified and hence the string may contain up to 10 '%'s.

Remarks:

This procedure is designed to make conversion of BASIC programs to BCPL very simple since the format of **string** corresponds to the format of the BASIC command VDU. It is also very useful in writing BCPL programs that use any VDU driver facilities. For instance, example (3) could form the body of a procedure to set the text cursor to position (x, y) and example (4) could be used for defining a graphics window.

The console must be selected as the current output stream before calling VDU.

If the format of **string** is invalid ABORT is called.

Calling VDU is equivalent to making a series of calls to WRBIN, but is much faster and in many cases is far more convenient.

VDU is in section "VDU" of the CINTCODE library LIB.

The VDU driver facilities are described in the BBC Microcomputer User Guide.

Portability:
Specific to this implementation.

VDUINFO

Purpose:
To give the numbers of rows and columns available with the current display mode.

Examples:

```
No.of.rows := VDUINFO(1)
No.of.cols := VDUINFO(2)
```

Function:
The result is the number of rows or columns of text that can be displayed with the current display mode (e.g. in mode 7 VDUINFO(1) returns 25 and VDUINFO(2) returns 40).

Remarks:
This procedure may be useful in writing programs that adapt their output according to the current display mode.

VDUINFO is declared in SYSHDR and is contained in section "VDUINFO" of the CINTCODE library file LIB.

Portability:
Specific to this application.

VECTOFILE

Purpose:

To convert a vector into a file.

Examples:

```
(1) is.successful := VECTOFILE( fromvec,
                                tofile)
(2) LET v = GETVEC(20)
    MOVE( data, v+2, 19)
    v!1 := 38    // no. of bytes of data
    VECTOFILE( v, "/F.myfile")
```

Function:

Fromvec must be a heap vector (i.e. one obtained by a call of GETVEC) set up as follows :

word 0	not used;
word 1	number of bytes of data in the vector;
words 2 on	data.

Tofile is interpreted in the normal way. Thus "/S.myfile" and "myfile" both refer to a store file and "/F.myfile" refers to a current filing system file. Other devices may also be specified (e.g. "/P").

The data in the vector is written to the specified file (if a file of the specified name already exists it is overwritten). The vector becomes the property of the system and is no longer available to the calling program.

The result is TRUE for success, FALSE for failure. In the latter case RESULT2 contains the error code.

Remarks:

If **fromvec** is not a heap vector or if the byte count in **fromvec**!1 is invalid ABORT is called.

If **tofile** is a store file then the vector becomes the file. If it is not a store file then the vector is returned to the heap after copying the data.

It is very important that the calling program does not attempt to use **fromvec** after a successful call of VECTOFILE. In particular it must not FREEVEC it.

The procedure SAVEVEC provides a way of writing a vector to a file while keeping the vector for use by the calling program.

Portability:
Specific to this implementation.

WRBIN

Purpose:
To write a binary byte.

Example:
WRBIN(binarycharacter)

Remarks:
The **binarycharacter** is added to the output stream, without any checks on its content.

Portability:
Part of extension A10b) of the standard language.

WRCH

Purpose:
To write a character to the output stream.

Example:
WRCH(character)

Function:
If the character is '*N', it is expanded to line feed and carriage return.

Portability:
Standard BCPL procedure.

WRITEA

Purpose:
To output a word address.

Example:
WRITEA(wordaddress)

Function:
If the **wordaddress** is that of a named procedure, the first seven characters of the name are written to the current output stream, following an initial space. Otherwise the **wordaddress** is output. In both cases the field width is 8 characters.

Remarks:
This procedure is used by the debugging facilities, and is expected to be of use to special debugging features.

WRITEA is declared in SYSHDR.

Portability:
Developed for this implementation.

WRITEBA

Purpose:
To output a byte address.

Example:
WRITEBA(byteaddress)

Function:
All addresses in the debug package are given in 16 bit word format, and odd byte addresses are shown as the equivalent 16 bit word address, followed by an 'H' - corresponding to the high byte.

This procedure outputs a byte address in this word address format. Thus **byteaddress** 23 would be output as 11H. The output always occupies 7 character positions with leading spaces as required.

Remarks:
See WRITEA remarks.

WRITEBA is declared in SYSHDR.

Portability:
Developed for this implementation.

WRITED

Purpose:
To write an integer in a given field width.

Example:
WRITED(integer, fieldwidth)

Remarks:
Integer is treated as a number in the range minus 32767 to 32767. It is output at the right of the field. If the **fieldwidth** is not sufficient the integer is output in the minimum fieldwidth necessary.

Portability:

Standard BCPL procedure.

WRITEDB

Purpose:

To output a double length integer.

Example:

WRITEDB(doubleinteger)

Function:

The **doubleinteger** is written to the current output stream, with a fieldsize of 8 characters. **Doubleinteger** must be a 2 word vector and **doubleinteger!**0 contains the least significant 4 decimal digits in binary, modulo 10000, while **doubleinteger!**1 contains the most significant 4 digits in binary.

Remarks:

The double length integer format is restricted to positive integers. This format is used in the internal count of jumps/procedure calls used in DEBUG.

WRITEDB is declared in SYSHDR and included in section "WRITEDB" of the CINTCODE library LIB.

Portability:

Developed for this implementation.

WRITEF

Purpose:

To write one or more arguments in a specified format.

Examples:

```
WRITEF( format,a,b,c,d,e,f,g,h,i,j,k)
WRITEF( "%S is aged %N*N", name, age)
WRITEF( format, argument1, argument2)
```

Function:

The string **format** is written to the current output stream. Each '%' causes the next argument to be converted and output according to the format character following the '%', as follows:

%A	wordaddress (using WRITEA);
%B	byteaddress (using WRITEBA);
%C	single character (using WRCH);
%In	integer in fieldsize n (using WRITED);
%N	integer in minimum size (using WRITEN);
%S	string (using WRITES);
%Xn	hex number in fieldsize n (using WRITEHEX);
%$	skips the next argument. This is useful when applying different formats to the same arguments;
%%	prints a single '%'.

The maximum number of arguments is 11.

Portability:

WRITEF is a standard BCPL procedure. However this implementation provides more conversions than the standard procedure (but does not provide %O to output in octal). Some of these conversions are available on some other implementations.

WRITEHEX

Purpose:
To write a hexadecimal number in a given field width.

Example:
WRITEHEX(integer, fieldwidth)

Function:
A hexadecimal representation of the **integer** is written to the current output stream in the given **fieldwidth.** Leading zeroes are shown. If the **fieldwidth** is too small for the **integer**, only the least significant hexadecimal digits are output.

Portability:
Standard BCPL procedure.

WRITEN

Purpose:
To write an integer in the minimum field width

Example:
WRITEN(integer)

Portability:
Standard BCPL procedure.

WRITEOCT

Purpose:
To write an octal number in a given field width.

Example:
```
WRITEOCT( integer, fieldwidth)
```

Function:
An octal representation of the **integer** is written to the current output stream in the given **fieldwidth.** Leading zeroes are shown. If the **fieldwidth** is too small for the **integer,** only the least significant octal digits are output.

Remarks:
WRITEOCT is declared in SYSHDR and included in section "WRITEOC" of the CINTCODE library file LIB.

Portability:
Standard BCPL procedure.

WRITES

Purpose:
To write a string.

Examples:
```
WRITES( string)
WRITES( "*NThis text is output")
```

Function:
The fieldwidth is the length of the string.

Portability:
Standard BCPL procedure.

WRITET

Purpose:
To write a string in a specified field width.

Examples:
```
WRITET( string, fieldwidth)
WRITET( "This is output in 30 chars", 30)
```

Function:
Trailing spaces are added if necessary to make the string fit the field width. If the text is too long to fit, the minimum field width to hold the text is used.

Remarks:
WRITET is declared in SYSHDR and included in section "WRITET" of the CINTCODE library file LIB.

Portability:
This procedure is found in other BCPL implementations.

WRITEU

Purpose:
To write an unsigned integer in a given field width.

Example:
```
WRITEU( unsignedinteger, fieldwidth)
```

Function:
The **unsignedinteger** is treated as a number between 0 and 65,535. If the **fieldwidth** is not sufficient for all significant digits, the **unsignedinteger** is output in the minimum necessary field width. Leading zeroes are printed as spaces.

Remarks:

WRITEU is declared in SYSHDR and included in section "WRITEU" of the CINTCODE library LIB.

Portability:

This procedure is found in other BCPL implementations.

WRITEWORDS

Purpose:

To write a number of words of data to the current output stream.

Example:

```
WRITEWORDS( source, words)
```

Function:

The specified number of words of binary data are written to the current output stream.

Remarks:

This procedure gives a more efficient transfer of data than repeated calls to WRBIN. This is particularly so when writing to a filing system which supports the operating system OSGBPB (read/write block of bytes) routine.

The use of WRITEWORDS may be combined with the use of WRBIN and WRCH and other output procedures. It is possible to use WRITEWORDS on any output stream.

Portability:

The procedure is provided on some other BCPL implementations.

6 Run Time

This chapter provides background information on the BCPL CINTCODE system. An understanding of the features described is not necessary for many users of the system. However, experienced programmers should find the contents of this chapter help them in making full use of the system. Further details of some of the topics discussed are given in the Appendix.

The system is described under the following headings:

Command files

Fault, event and ESCAPE handling

GETVEC

Layout of store

Loading and linking code

RUNPROG

Stacks

Starting and stopping

Store files

Streams

Use of filing systems

COMMAND FILES

Executing command files

When the EX utility is run by the command

EX exfile parameters

it copies exfile to a store file named $$EX, substituting the parameters for keywords in the exfile as it does so. When it has finished it exits leaving the file $$EX as the current input stream for the command state.

The command state thus reads the next command from the file and continues reading commands from the file until the end of the file is reached or execution of the file is terminated (e.g. because the command END is executed). When the file is terminated it is deleted by the system. As each command is read from the file the entire line is displayed on the screen.

When a program being run from a command file traps to the command state (e.g. because **ESCAPE** has been pressed) the system reverts to reading commands from the console until the program is resumed (by the command CONT) or abandoned (by TIDY). Note that unless ERRCONT is in effect TIDY terminates the command file.

If EX is invoked from within a command file it creates a temporary file ($$EXTMP) from the exfile then reads the rest of the current command file and appends it to the temporary file. Finally it deletes the current command file and renames $$EXTMP to $$EX.

Programs run from command files

A program run from a command file is entered with the command file as the current input stream. The next character read will be that following the program name. In most cases the program will not read beyond the end of the current line (e.g. if it uses RDARGS), but it may read subsequent lines if necessary. These lines are not automatically echoed to the screen. When the program returns to the command state the remainder of the current line in the command file is discarded.

Any program which reads (apart from calling RDARGS) from the input stream selected when the program is entered must allow for that stream being a command file rather than the console. Two particular points to bear in mind are:

(1) The program must cater for encountering the end of the file (reading ENDSTREAMCH) and take appropriate action. This action might be to generate an error condition or to select a console input stream (this is one of the occasions when it might be reasonable to use the stream CNSLINSTR which is normally reserved for system use) and prompt the user for input.

(2) Code to discard the rest of the current line must be written something like:

```
UNRDCH()
CH := RDCH() REPEATUNTIL CH='*N' |
                         CH=ENDSTREAMCH
```

rather than the following, which is suitable only for reading from device /C (the console read a line at a time):

```
WHILE TESTFLAGS( MORE.INPUT) DO RDCH()
```

The section 'File and stream handling' in chapter 8 explains how a program can tell whether it is reading from the console or from a file.

Error handling in command files

If a program run from a command file terminates with a fatal trap or by calling ABORT then the command file is also terminated. If a program terminates by calling STOP with a positive (non-0) parameter or is terminated by the built in command TIDY then the command file is also terminated unless the built in command ERRCONT has been issued (at any time after the previous command file, if any, terminated).

The effect of ERRCONT can be cancelled by the command ERRCONT OFF. Whenever a command file terminates the system automatically performs an ERRCONT OFF.

When a program run from a command file terminates (or if it cannot be run) the global LASTERROR is set up as follows:

- if the program cannot be run LASTERROR is set to the appropriate error code;

- if the program terminates by FINISH, ENDPROG or returning from START then LASTERROR is set to 0;

- if the program terminates by calling STOP then LASTERROR is set to the parameter of STOP;

- if the program is terminated by TIDY then LASTERROR is set to -4.

Any program run from a command file may test LASTERROR and thus find out whether the previous program completed successfully. This is particularly useful when ERRCONT is in effect.

FAULT, EVENT AND ESCAPE HANDLING

This section explains how the BCPL system handles faults and events generated by the operating system and how it handles the **ESCAPE** key being pressed in various circumstances. The BBC Microcomputer User Guide details how faults and events are generated.

Faults

Faults can only occur when the system is executing machine code called from BCPL. The machine code may be user-written code or an operating system routine.

The BCPL fault-handling routine clears any escape condition, sets the global variable MCRESULT to #XFFnn where nn is the fault number and resumes execution of CINTCODE as if the machine code that was being executed had returned normally.

Most system routines that call machine code check for a fault having occurred and take appropriate action. Often this is to set RESULT2 to 1000+nn (decimal) and return with an error indication.

The section 'Machine code' in chapter 8 describes how users may replace the BCPL fault-handling routine.

Events

The BCPL system does not enable any events and does not contain an event-handling routine. Any application wishing to use events must include suitable machine code to set up an event-handling routine and enable the events.

ESCAPE

Operating system action

When **ESCAPE** is pressed the operating system detects it. Assuming that the escape event is not enabled (which is the normal situation) then the operating system takes one of three actions:

(1) If the computer is currently executing certain operating system routines then these routines return control to the caller, passing back an indication that **ESCAPE** was pressed. The main routines involved are OSRDCH and OSWORD when reading from the console or serial port.

(2) If the computer is currently executing certain I/O routines then a fault condition is declared (fault number 17) and control is passed to the fault-handling routine.

(3) In any other case an escape condition is declared. The top bit of byte 255 is set and any subsequent calls to the routines mentioned in (1) and (2) above have the effects described.

BCPL system action

The action taken by the BCPL system depends firstly on which of the three types of operating system action (see above) was taken and secondly on whether the system is in the command state or the run state. A run state program can disable the normal system actions on **ESCAPE**. If a program does this then the actions taken are those described for the command state.

In all the cases described the BCPL system clears the escape condition when it takes notice of **ESCAPE** having been pressed.

Note that if **ESCAPE** is pressed while the system is executing machine code called from BCPL then the BCPL system can take no action until control is passed back to it.

The BCPL system actions corresponding to the three operating system actions above are:

(1) In the command state ABORT(1017) is called. This ABORT may be trapped in the normal way by setting ABORTLEVEL and ABORTLABEL. In the run state the system traps to the command state and displays 'Interrupted'. The command CONT resumes the run-state program. In both cases any characters input on the current line before **ESCAPE** was pressed are lost.

(2) The fault routine clears the ESCAPE condition and sets MCRESULT to #XFF11. The subsequent action is up to the routine that called the operating system routine. In the case of BCPL library routines the action is the same for the command and trap states. If the routine is one which is able to return a fault code back to the caller then it does so (e.g. FINDINPUT returns 0 with RESULT2 set to 1017). If the routine cannot return a fault code (e.g. RDCH) it calls ABORT(1017).

(3) In the command state the BCPL system generally ignores an escape condition. Certain parts of the system explicitly check for it (by testing byte 255) and react by abandoning the current activity (e.g. COPY checks and truncates the copy, HEAP checks and curtails the heap display). In the run state the BCPL system traps to the command state with the 'Interrupted' message unless it is currently executing library routines contained in the BCPL Language ROM. In this case the trap to the command state is delayed until execution of the library routines is complete.

A program may choose for the BCPL system to react to **ESCAPE** as if it were in the command state rather than the run state. In particular this means that the program will never trap to the command state with the 'Interrupted' message. Reasons for doing so might be:

- the program may not want the user to be able to interrupt it at all (an alternative method of doing this would be by the operating system command '*FX 229,1', but this would disable **ESCAPE** for all subsequent programs as well);

- the program may want to inhibit the user interrupting a critical section of code (perhaps because the system would be in an inconsistent state if he were to do so).

Programs which have disabled the normal **ESCAPE** action may check explicitly for an escape condition using code similar to the following:

```
IF 0%255 > 127 THEN
$( OPSYS(#X7E)  // acknowledge ESCAPE
   ...          // perform required
   ...          // action
$)
```

A method of disabling normal **ESCAPE** handling for the entire program is to include the procedure STARTINIT and include the following line in it:

```
SYSINDEX!I.RSTATE!R.MCST :=
   SYSINDEX!I.RSTATE!R.MCST & NOT M.TRAPESC
```

Note that if the program is run by RUNPROG then STARTINIT is not invoked.

To disable normal **ESCAPE** handling from within the program include the line:

```
SYSINDEX!I.CSTATE!R.MCST :=
   SYSINDEX!I.CSTATE!R.MCST & NOT M.TRAPESC
```

To re-enable normal **ESCAPE** handling include the line:

```
SYSINDEX!I.CSTATE!R.MCST :=
   SYSINDEX!I.CSTATE!R.MCST | M.TRAPESC
```

All the symbols used above are declared in SYSHDR.

GETVEC

GETVEC is called to obtain a vector of a certain size from the heap. It searches the heap from the bottom until it finds a free area equal to or bigger than the size required. The vector is then allocated from the bottom of this area.

If GETVEC fails to find an area big enough it calls SHUFFLE to move all unlinked store files down the heap (if possible) and then tries again. If it still fails it calls SHUFFLE again which this time deletes all unprotected store files before moving the remaining unlinked files down the heap.

(Note that SHUFFLE can move store files even if they are currently being read or written. SHUFFLE automatically updates the relevant pointers in the stream control blocks (see 'Streams' below). Users must not write programs which access store files directly since such programs could fail if SHUFFLE were invoked while they were running. Access to store files must be made either through the normal I/O procedures or by READVEC/ FILETOVEC.)

If GETVEC fails for the third time then the action it takes depends on whether the system is in the command or the run state. In the command state it returns a value of 0 and sets RESULT2 to 51. In the run state it traps to the command state with the message 'nnn + store needed' where nnn is the size of the vector required.

Certain programs may prefer to have GETVEC return a failure indication than to trap to the command state. For example the program may be designed to be operated by someone who would not know how to react to the 'store needed' message.

One method of achieving this is to include the procedure STARTINIT containing the line:

```
SYSINDEX!I.RSTATE!R.MCST :=
   SYSINDEX!I.RSTATE!R.MCST & NOT M.TRAPGV
```

Note that if the program is run by RUNPROG then STARTINIT is not invoked.

Another method is to include the following line in the program before any call of GETVEC (or call of any library procedure that might call GETVEC):

```
SYSINDEX!I.CSTATE!R.MCST :=
   SYSINDEX!I.CSTATE!R.MCST & NOT M.TRAPGV
```

To re-enable the normal action of trapping on GETVEC failure include the line:

```
SYSINDEX!I.CSTATE!R.MCST :=
   SYSINDEX!I.CSTATE!R.MCST | M.TRAPGV
```

All the symbols used above are declared in SYSHDR.

LAYOUT OF STORE

The use of store is divided into the following areas:

Zero page:
A 256 byte area used by the operating system and the BCPL Language ROM. Bytes 112 to 143 are available for use by applications.

Operating system workspace.

BCPL language workspace:
This area starts at word 512 and holds some system data and the root stack (the stack used by the system when in the command state). For convenience the heap actually starts in this area and the root stack and operating system/filing system areas mentioned below are treated as permanently allocated heap vectors.

Operating system input and output buffers:
A fixed area from word 1024 to word 1663.

Filing systems fixed workspace:
Starts at word 1664, and extends to the largest size required by an installed filing system.

Filing system contexts:
A dedicated area is reserved for each installed filing system.

Main part of BCPL heap:
A free store area used for global vector, stacks, loaded code, data, stream control and store files.

VDU buffer:
In display mode 7 this is a 1024 byte buffer. In other display modes it is larger (see the description of procedure MODE in chapter 5).

Top of RAM:
    8192 words for Model A. 16384 words for Model B.

BCPL ROM, library and interpreter:
    The BCPL Language ROM occupies word addresses 16384 to 24575 and contains CINTCODE library procedures, both for use by application programs and to provide the system functions. The interpreter contains the machine code to execute CINTCODE. The same address space can hold code for filing systems, other languages and also files in ROM.

Operating system:
    The operating system ROM uses the addresses from word 24576 to 32767.

LOADING AND LINKING CODE

Each section of BCPL source text is compiled into a block of CINTCODE known as a hunk. Since the source text for compilation may contain a number of sections, the CINTCODE file created by the compiler can contain a number of hunks.

For compatibility the assembler also creates code in similar hunks. The group of hunks held in one file is known as a segment. Compiled segments can be joined into larger segments by the utility JOINCIN.

Loading a file (or segment)

Before a hunk can be used as code, it must be held in a contiguous area of store, so that references in the hunk are resolved correctly. The process of converting the file into this form is called loading. However a loaded file can still be read as a file.

CINTCODE hunks are loaded in the order they are held in the file. If possible all the hunks in a file are loaded together into one contiguous area, but if this is not possible then a separate vector is used for each hunk. This means that the hunks are not necessarily held in store in the order they were loaded. However they still form a file and can be acted on as a whole by procedures such as GLOBIN.

The procedure LOADSEG loads a file and returns a segment identifier which is used as a parameter to GLOBIN (see below) which links the segment.

In this system there is no concept of unloading a segment. Provided the segment is not linked to the global vector the file containing it can be deleted if required. However, in some other BCPL implementations a procedure UNLOADSEG is necessary to delete loaded segments. It is provided as a dummy call returning zero in this implementation so that common code can be used on a variety of installations. Code intended to be portable should call UNLOADSEG (after calling GLOBUNIN - see below) when the loaded segment is no longer needed.

Linking to the global vector

Each hunk includes information on the position of each global procedure within it. When the BCPL Language ROM is initialised, the global vector is created and then the relevant globals are initialised to point to the global procedures defined in the ROM library.

When a program is run, the specified CINTCODE file is loaded and then linked by GLOBIN, which places references in the global vector to any global procedures in the segment. GLOBIN also relocates any machine code hunks in the segment. Once a segment has been linked it cannot subsequently be moved (by SHUFFLE) or deleted until it has been unlinked by GLOBUNIN. In this process any global which is still pointing to the correct entry address in the segment is reset, and any machine code hunk is relocated back to its original contents. A file which has been unlinked remains loaded but can be deleted, SHUFFLEd etc.

Note that a file which is linked cannot be read by a program or copied (because its contents may have been changed by relocation of machine code).

Since the last setting of a global overwrites any previous setting a newly linked segment may redefine any existing global procedure. The new definition will then apply to all calls of the procedure.

Relevant built in commands

The commands LOAD, LINK and UNLINK correspond closely to the procedures LOADSEG, GLOBIN and GLOBUNIN respectively. The principal differences are:

- LOAD leaves the loaded file protected, whereas LOADSEG leaves it unprotected (unless the file already existed as a protected file);

- LINK and UNLINK apply to files rather than segments. Thus they have file names as parameters whereas GLOBIN and GLOBUNIN use segment identifiers (as returned by LOADSEG). LINK automatically loads the file if it is not loaded;

- LINK allows a file to be linked as a SYSTEM file. Such a file remains linked until explicitly unlinked by the UNLINK command. SYSTEM files are useful for providing alternative versions of standard library routines and for libraries of application-specific procedures. Note that after relinking the library in ROM (by GLOBIN(LIBBASE) or by the built in command LINK LIBRARY) any SYSTEM files containing replacements for routines in the ROM library must be relinked.

Use of the loading and linking routines

LOADSEG and GLOBIN may be linked in one statement since the result of LOADSEG can be the parameter for GLOBIN. For example:

```
is.successful := GLOBIN( LOADSEG( filename))
```

If the sequence is not successful it is still possible to discover the underlying reason by reading RESULT2.

Since LOADSEG places the segment in free store, it will fail to load if there is not enough room in store, or even if the free store is fragmented so that one section of the segment cannot fit. Thus it may be necessary to provide some way of ensuring the load will fit, either by reserving sufficient space which is released before the load, or by having a recovery strategy if the load will not fit.

On this implementation the system traps to the command state if it runs out of store allowing the user to take an appropriate action before resuming the program (see 'GETVEC' above). However there are many cases in which the user cannot be expected to know the appropriate action to take.

A linked procedure can replace an existing global procedure, whether from the library in ROM, or an earlier link. If the new procedure is unlinked by GLOBUNIN the global is set to the special value GLOBWORD rather than to its previous value. If the global is required in its earlier form it must be reinitialised, either by code which has remembered its earlier setting, or by reapplying GLOBIN to the relevant area. For example:

```
GLOBIN( LIBBASE)              // relink library
```

or

```
GLOBIN( firstsegmentloaded)
```

LIBBASE is declared in the header file SYSHDR.

Although GLOBUNIN unlinks the global references to the segment, it is the programmer's responsibility to ensure that no other references to the code are used. Such references could be saved by copying a global, saving the addresses of statics, or by using a reference in a procedure call.

RUNPROG

The procedure RUNPROG allows a program to run another program or utility or to execute an operating system command. RUNPROG may have from one to five parameters. The first is a string (in the format accepted by the procedure WRITEF). The other four are parameters for this string.

It is assumed that readers are familiar with the description of RUNPROG in chapter 5.

When RUNPROG is called it creates a temporary store file and writes out its arguments to this file (using WRITEF). It also writes out a new line character at the end of the file. The temporary file is named '$$RPx', where x is 'A' for the first invocation of RUNPROG, 'B' if RUNPROG is called by the program being run by RUNPROG, 'C' if the program run by the second call of RUNPROG itself calls RUNPROG etc. (Calls of RUNPROG may be nested up to a maximum depth of 26, but a program may not RUNPROG itself.)

RUNPROG then reads back the file interpreting it as if it were a command entered at the console. If the command starts with an asterisk then it is passed to the operating system command line interpreter. If the command is a built in command then it is executed.

If the command is neither of these (i.e. it is to run a program or utility) then the specified file is loaded and linked. If the file contains a procedure TRAPSTART then that procedure is called. If not then if the file contains a procedure START then it is called otherwise an error is generated. Both START and TRAPSTART are called with a parameter of 0.

Note that even if the specified file contains a procedure STARTINIT it is not called and so the program being run is entered with the same stack and same options for handling **ESCAPE** and running out of store as the program calling RUNPROG.

On entry to the program being run the output stream is the console and the input stream is the temporary file created by RUNPROG (the next character read will be that following the program name).

When the program or command terminates control is returned to RUNPROG which deletes the temporary file. If a program was linked by RUNPROG then it is unlinked (but any files linked by the program itself are left linked). RUNPROG restores the current streams and the values of ABORTCODE, ABORTLEVEL, ABORTLABEL, START, STARTINIT, STOP and TRAPSTART to the state they were in when RUNPROG was called. It does not close any streams left open by the program that was run.

The result returned by RUNPROG indicates the success or otherwise of the program or command run. For details see the section on RUNPROG in chapter 5.

If the program being run traps to the command state for any reason the effect is just the same as if the calling program had trapped.

When using RUNPROG the following points should be considered:

- The program being run should cope with its input stream being a file rather than the console (this is discussed in more detail in 'Command files' above).

- What to do if either of the programs involved uses non-standard options for ESCAPE or GETVEC failure handling.

- The calling program must have a big enough stack to cater for all the stack requirements of the called program (plus an overhead of about 30 words for the call of RUNPROG itself).

- A program may not RUNPROG itself.

- The program being run should close any new streams it opens and unlink any files it links.

- RUNPROG cannot be used to issue the commands CONT, INIT and TIDY.

- If the called program uses any globals used by the calling program then on return from RUNPROG the calling program must reset those globals (by relinking itself if necessary). The built in commands do not use any globals above 249 and so it is safe to RUNPROG them from any application program.

STACKS

BCPL systems always hold the details of the current procedure, its parameters, local variables and any intermediate results in a stack. Thus much of the current state of processing is related to the position of the stack pointer, often called the P pointer.

When a procedure is called the stack pointer is moved past the parameters and local data of the calling procedure to a new position at a higher address than the previous position. On return from the called procedure the stack pointer is reset to its previous position. The utility STACK (described in chapter 7) shows the value of the stack pointer for each procedure in the current chain of procedure calls.

In this implementation the first few words in the stack following the stack pointer position are:

- the stack pointer for the calling procedure (used to reset the stack pointer on return from the current procedure);

- the address for returning to the calling procedure (as a byte address);

- the word address of the called procedure (used only for debugging purposes);

- the parameters of the procedure (in the order specified in the call), immediately followed by the local variables of the procedure. (But if fewer parameters are specified in the call than are declared in the procedure declaration, spare stack space is left for the unused parameters.)

When a program is started it is provided with a stack, known as the main stack, which by default is 400 words long. The program can request a different size of main stack by defining a procedure STARTINIT which returns the stack size required.

In many cases the main stack is all that the program needs. However in some cases it is convenient to divide a system into a number of coroutines, each of which has a separate stack. The stack for a coroutine is created by CREATECO and deleted by DELETECO. For more details see 'Coroutines' in chapter 8.

The code which implements the command state is written in BCPL and uses a stack known as the root stack which is permanently allocated from the heap. This stack is used when a program traps to the command state for any reason, so that the state of the current stack (i.e. program stack) is preserved.

The ability to run in the command state while maintaining the current run state is used to provide the extensive debugging features described in chapter 7. (Utilities such as DEBUG, HEAP etc. run using the root stack.)

Thus when a program is being debugged, processing switches from the program procedures running in the current stack to the DEBUG procedures running in the root stack and back again.

The built in commands, described in chapter 3, are normally executed using the root stack to avoid changing the current stack. However if these commands are executed from a program using the procedure RUNPROG, they run in the current stack.

When a stack is created it is initialised to contain zeroes. It is thus possible to show approximately how much of a stack has been used by finding the last non-zero word. The DEBUG facility to report on stacks and the STACK utility display this value.

Stack overflow is always trapped as an error. The overhead of checking for this condition is minimised by ensuring that there are always at least 16 words of stack left when a procedure is called. This means that instructions referencing the first few words after the stack pointer do not need to be checked.

The procedure STACKSIZE may be used to determine the amount of the current stack that is left. Users should remember that stack overflow occurs if there are less than 16 words left when a procedure is called.

The procedure LONGJUMP provides a means of unwinding a chain of procedure calls by returning directly to a specified label and stack pointer position. A label can only be addressed directly if it is in the same procedure. Thus a label for use by LONGJUMP is normally transferred to a global or static variable. Similarly the procedure to which LONGJUMP will jump normally initialises a global or static variable with the stack pointer (obtained by the procedure LEVEL).

LONGJUMP only works within the current stack. An attempt to LONGJUMP to a different stack normally causes an ABORT but in some circumstances may crash the system.

STARTING AND STOPPING

Initialisation

If the BCPL Language ROM is the right-most language ROM in the ROM sockets, it is initialised automatically on power up. Otherwise it can be initialised by the operating system command '*BCPL'.

System initialisation establishes the data areas in RAM used by the interpreter and creates the heap, the root stack and a global vector of 768 words. It also opens a console input stream, a console output stream and an error stream for use when no other stream is selected.

The system then enters the command state to accept commands from the user.

Pressing **BREAK** or entering the command '*BCPL' re-initialises the system (all store files are lost).

Starting

When a command line is entered at the console the first item in the command is read and (unless it is an operating system command or a built in command) a CINTCODE file of this name is loaded and linked.

If the program contains the global procedure TRAPSTART then that procedure is called. This is used by debugging aids and other programs that run in the command state. The Appendix contains a section on writing command-state programs.

Assuming the program does not contain TRAPSTART then if it does not contain START either an error is generated. Otherwise a special version of the TIDY routine is called (which deletes all co-routines, closes all streams (except for CNSLINSTR, CNSLOUTSTR, ERRORSTREAM and the current command file) and frees all data areas allocated from the heap). Streams for use by the program are next created. The output stream is CNSLOUTSTR. The input stream is either the current command file or, if there is no command file, a copy of the current console input stream (containing the command line used to run the program). The next character read will be that following the program name.

The system then creates a stack for the program. If the program contains STARTINIT that procedure is called (in the command state, using the root stack and with the command state streams selected) and the result used as the stack size. If there is no STARTINIT the default stack size is used.

Finally the program is entered by calling START with a parameter of 0.

Stopping

The normal methods of terminating a program are to return from START, call STOP with a parameter of 0 or execute the BCPL statement FINISH.

STOP may be called with a non-0 parameter in which case the command state calls the procedure ERRORMSG to display a message. Negative parameters are treated as warnings. Positive parameters are treated as errors and will terminate execution of the current command file unless ERRCONT is in effect. Users may provide their own versions of ERRORMSG to generate alternative messages. See the section on ERRORMSG in chapter 5 for details.

The procedure ENDPROG is appropriate if the program needs to confirm that the user wishes to stop. ENDPROG asks the user for confirmation and stops the program (by calling STOP with a parameter of 0) if the user types 'Y' or 'y' in reply.

In all the cases described above the system calls the TIDY routine which tidies the heap (as described in the previous section) and also unlinks all linked files (except SYSTEM files which can only be unlinked by the UNLINK built in command).

An ABORT arises from an error discovered by a BCPL procedure which calls ABORT. This outputs an error message and then traps to the command state in a way which indicates that an error has occurred. The current state is not changed and can be investigated by the DEBUG facilities (in particular the TIDY routine is not called). However certain errors, in particular the corruption of the heap, are likely also to affect the DEBUG facilities, and then the facilities to analyse the problem are limited. Programs can trap calls to ABORT as described in the section on ABORT in chapter 5.

A fatal trap is an error discovered by the interpreter. It causes an entry to the command state in the same way as ABORT, but it cannot be trapped by the program. A fatal trap typically arises from stack overflow or calling a non-existent procedure.

The different fatal traps have single letter error codes which are listed in chapter 11.

Occasionally an error in a program will lead to a fatal error in the command state processing. In this case, the system displays a message of the form:

*** ERROR: x

where x is one of the single letter error codes mentioned above. When this occurs the system must be restarted by pressing **BREAK.**

The procedure TRAP allows the program to simulate the effect of traps. In particular a call of TRAP with parameter 0 terminates the program.

The user may terminate a program which has trapped to the command state (perhaps because **ESCAPE** was pressed) by the built in command TIDY. If the program was run from a command file this also terminates the command file unless ERRCONT is in effect.

STORE FILES

Provision for store files is an important feature of the system (particularly when tape is the only other storage device), since it enables a number of files to be available for immediate use. For the BCPL programmer store files can be read and written in a very similar way to files on any other filing system (e.g. disk).

All space used for store files is allocated from the heap. Every store file has a 'file control block' (FCB) and a 'file name block' (FNB). The FCB contains pointers to the other blocks associated with the file and links to other files. The FNB just holds the file name. Unless the file is a zero-length file it also has one or more data blocks. These hold the contents of the file. The formats of the various blocks are detailed in the Appendix.

When store files are created or extended the various blocks are allocated from the heap using GETVEC. If there is not enough space available then SHUFFLE is used as described in 'GETVEC' above.

A store file may be marked as protected. This means that it is not deleted automatically by SHUFFLE. However a protected file can be deleted by the DELETE command, the DELFILE/DELXFILE procedures or by opening for output a file of the same name. Files are protected when opened by FINDOUTPUT/FINDXOUTPUT or read into store by the commands COPY, READ, LOAD and LINK, but are not protected if brought into store by LOADSEG. The protection may be changed by the PROTECT command.

Store files may be loaded (by LOADSEG) and linked (by GLOBIN) as described in 'Loading and linking code' above.

Note that, unlike files in the BBC Microcomputer filing systems, store files do not have load and execution addresses associated with them.

STREAMS

A stream is created by FINDINPUT, FINDOUTPUT, FINDXINPUT or FINDXOUTPUT. These procedures create a 'stream control block'. This contains control information on the device, and a suitably sized buffer to hold the data read from the input device or to be written to the output device. The Appendix details the format of stream control blocks.

The stream identifier, used in this implementation, is a pointer to the relevant stream control block.

When a stream is selected by SELECTINPUT or SELECTOUTPUT, key information from the stream control block is transferred to globals so that it is immediately available to the input and output procedures such as RDCH and WRCH. When another stream is selected for the same direction of transfer, the information for the previous stream is replaced in its stream control block.

A special ERRORSTREAM is created initially. This causes an ABORT if any character is written to it or read from it. On ENDREAD or ENDWRITE the relevant current stream control block is freed and the ERRORSTREAM is selected. This ensures that any attempt to access a non-existent stream causes an abort.

Two other special streams that are created initially are CNSLINSTR and CNSLOUTSTR (keyboard input and screen output respectively). Programs may select CNSLOUTSTR as required, but CNSLINSTR should only be selected in error conditions, since it is used by the command state and DEBUG for input (and strange interactions may occur if debugging a program that is reading from CNSLINSTR).

USE OF FILING SYSTEMS

One of the ‘devices’ recognised by BCPL is ‘/F.’ - the current filing system. BCPL itself never changes the current filing system, although the user can do so either directly e.g.

!*DISK

or from within a program e.g.

RUNPROG("**ROM")

There is an operating system restriction that files should not be open on more than one filing system at the same time. BCPL does not check this and it is up to the user to ensure that this restriction is obeyed. BCPL also cannot trap errors such as changing disks when there are open disk files. (Note that use of the procedures FILETOVEC, READ, READVEC, SAVE, SAVEVEC and VECTOFILE to transfer complete files between store and disk minimises the number of open disk files and so provides greater flexibility for swapping disks as required.)

The BCPL system is written so that it will work without any modifications with any filing system that may be introduced. Because some of the standard operating system routines are not supported by all filing systems, however, BCPL can perform I/O more efficiently if it is aware of some of the characteristics of each filing system.

The procedure FSTYPE (see chapter 5) provides the information required. The standard version of this procedure caters for the tape, ROM, disk and Econet filing systems. If new filing systems are added then users can replace the standard version with a version of their own containing the characteristics of the new filing system. It should be stressed, however, that if they do not replace the standard FSTYPE then BCPL will still work with the new filing system, but certain operations may be slower than they need be.

The information required about each filing system is:

- Whether OSGBPB (read/write block of bytes) is supported. If it is then READWORDS and WRITEWORDS will use this routine. If not they will just call RDBIN and WRBIN (default: not supported).

- Whether the OSFILE call to read the file's attributes can be used to find its length. If so then when reading the entire file (procedures READ, READVEC and FILETOVEC) OSFILE will be used (if appropriate) to read the file in one operation. If not then these procedures will read the file in blocks using READWORDS (default: not supported).

- Whether the filing system may generate messages (which must be displayed) while performing I/O (e.g. the tape filing system does this). If so then BCPL ensures that the console is the current OSWRCH device when calling any filing system routine (default: no messages).

- Whether the filing system is the tape or the ROM filing system. This information is used to set up message and error recovery options using the *FX 139 command before opening a file. Specifically for the tape filing system the commands:

  ```
  *FX 139,1,1  // short messages
  *FX 139,2,1  // re-try on errors
  ```

  are issued and for all other filing systems the command:

  ```
  *FX 139,1,0  // no messages
  ```

  is issued (default: neither tape nor ROM).

- Whether BCPL should display messages giving the name of the file being opened before opening any file (default: no messages).

The Appendix details the operating system calls used by BCPL for all types of I/O.

One feature of the current filing system not supported by BCPL is the concept of load and execution addresses. These file attributes are unnecessary for CINTCODE files and for files read by BCPL programs. Thus when a file is created by a BCPL program using the standard I/O facilities the load and execution addresses are set to random values. One important consequence is that COPY, READ, SAVE etc. should not be used to copy files (e.g. from one disk to another) where these addresses are important (e.g. machine code programs run by the '*name' command).

It is of course a simple matter to write a copy program in BCPL to preserve these addresses, by calling the OSFILE routine directly to read and save the file's attributes.

7 Debugging Aids

This chapter describes the range of debugging aids provided with the BCPL CINTCODE system. Experienced users should find the summary of the debugging commands given in chapter 11 is a sufficient reminder for most purposes.

The principal debugging aid is the utility DEBUG which provides powerful facilities for monitoring program execution. In addition there are a number of display utilities and a utility to allow testing of an individual procedure.

The debugging aids are described in this chapter under the following headings:

DEBUG - program test utility

GLOBALS - display global vector

HEAP - display heap

I/O - display I/O streams

STACK - display stacks

TESTPRO - test a procedure.

DEBUG - program test utility

The facilities of the DEBUG utility are described in this section under the following headings:

Introduction
    Covers topics applying to all facilities.

Altering memory

Bases
    How to work in decimal, octal, hexadecimal or characters.

Breakpoints

Calculating

Counting
    The facility to run to a known point.

Displays

Limits
    Setting the range for traces or statistics collection.

Memory search
    Finding words or procedures in memory.

Overlays
    How the debugging functions are brought into store.

Statistics
    How to collect information on program operation.

Traces
    How to monitor the execution of a program at the instruction, jump or procedure level.

Introduction

The CINTCODE debugging facilities are used on normal compiled code. It is not necessary to set any special compiler options to include the debugging facilities, and the code can be run without the debugging facilities at any stage.

The debug facilities can refer to procedures by name, because these names are included in the normal compiled CINTCODE. When the size of code must be minimised it is possible to compile CINTCODE without these names, saving five words per procedure. Normally naming can be retained while the code is being developed.

Entering, leaving and re-entering DEBUG

DEBUG is entered by

!DEBUG

On being entered DEBUG takes over control of the console using the prompt '*'.

The only way to exit from DEBUG back to the command state is to type 'X' (followed by **RETURN**) in response to the prompt.

While in DEBUG **ESCAPE** may be used to interrupt displays etc. but it always returns to the DEBUG prompt rather than the command state.

It is possible to enter DEBUG, set up breakpoints etc., exit to the command level then re-enter DEBUG and find that the breakpoints etc. are still set up (provided that the TIDY routine has not been invoked since leaving DEBUG). This gives great flexibility when using DEBUG to test a program, since it is possible to exit from DEBUG and use other debugging aids (e.g. HEAP) then resume testing as if DEBUG had never been exited.

Testing a program with DEBUG

The normal sequence of events when testing a program is as follows:

(1) Link the program to the global vector by the command:

    !LINK filename

(2) Initialise the program by the command:

    !INIT parameters

(3) Enter DEBUG as described above and commence testing by setting breakpoints, running to a count etc.

(4) Control returns to DEBUG either because a breakpoint is reached or a count is up, or because some event has occurred that would normally return control to the command level (e.g. **ESCAPE** pressed, program ended, program error such as calling an uninitialised global).

(5) At this stage the various DEBUG facilities to examine and alter store can be used.

(6) If the program has ended it can be restarted by leaving DEBUG and resuming at step (2).

(7) If the program has not ended then there are several options. The program can be run to another breakpoint or count or DEBUG can be exited. In this case testing can be abandoned by:

!TIDY

or the program can be re-run by returning to step (2), or the program can be run to completion by:

!CONT

or various trap state programs (e.g. COPY or the display utilities described later in this chapter) or built in commands can be executed and DEBUG re-entered. If DEBUG is re-entered then testing can continue just as if DEBUG had never been exited.

Provided that there is room in store for DEBUG, it is also possible to use DEBUG to test a program that has been started normally then interrupted (by **ESCAPE**). Simply enter DEBUG and carry on from step (6) above.

Commands

Commands to DEBUG are acted on one line at a time after the **RETURN** key is pressed. Until this moment the line can be edited using the normal editing functions (arrow keys, **COPY**, **DELETE** and **CTRL-U**).

The line may contain several commands which are executed in order from left to right. Commands may be separated by spaces. Certain commands require the name of a segment or procedure. This name must end with a space or be the last item in the line. Any alphabetic characters in a command or name may be entered in upper or lower case. However upper and lower case character constants are distinguished, thus 'a = 97 and 'A = 65.

Numbers and addresses

By default the DEBUG system accepts numbers in decimal, and outputs numbers in decimal. Other bases may be used as described below.

Addresses are word addresses. If a byte address is required the word address refers to the first byte. The address is followed by 'H' to reference the second or **H**igh byte in the word.

In some contexts an address of a procedure is replaced by the first seven characters of the procedure name.

The current variable

Many DEBUG commands require a number, for example the address for a breakpoint or the number of lines to trace. In simple cases this is entered as a number just before the command, but in other cases the system remembers and uses the last number specified. This 'current variable' can be set in a variety of ways, for example to the address of a named procedure, or the contents of a global. It can also be calculated on. The setting of the current variable can be displayed by the '=' command. Thus the following example sets the current variable to 1000, displays the setting and then sets a breakpoint at that point.

```
*2000/2 = B
```

The repeat count

A number of commands require a repeat count, for example to determine the number of lines to trace. The repeat count is set to 1 at the start of a line, but it becomes equal to the current variable as soon as that is set or changed in the line.

Thus

*T

traces one line, while

*12T

traces 12 lines.

Globals

DEBUG can reference any global by 'Gn' where n is the global number in the range 0 to 767. The effect is normally to set the current variable to the contents of the global but it is also possible to alter global data. Thus the following command would set a breakpoint on a procedure defined as global 300:

*G300 B2

Globals referenced in this way may be used at any point at which an integer number may be required and have the same effect as entering the contents of the global. Thus it is possible to calculate using globals. The following example would add globals 400 and 401 and display the result:

*G400 + G401 =

NOTE: When testing a program with DEBUG, misleading results are obtained if any of globals 4 to 15 (e.g. ABORTCODE, MCRESULT, RESULT2) are examined or changed. This is because while DEBUG is running these globals are used by DEBUG itself. The values of the globals for the program being tested are saved in an area accessed through global 14. See section 'System data areas' in the Appendix for full details.

Variables

DEBUG maintains ten variables, V0 to V9. These may be used for storage and retrieval of any integer data. Thus:

```
*SV3 234
```

sets variable 3 (V3) to 234, and:

```
*V3 =
```

would then print 234.

Variables referenced in this way may be used at any point at which an integer number may be required and have the same effect as entering the contents of the variable. Thus it is possible to calculate using variables. The following example displays 13 times the contents of V4 added to global 450:

```
*V4 * 13 + G450 =
```

Altering memory

Memory is altered by the 'S' command. There are two forms of this command. The first form alters one or more consecutive words. The second form displays the contents of successive words and optionally allows them to be altered.

The format of the first form is:

S address value

or

S address value1 value2 etc.

If only one value is specified then the word specified by address is updated to that value. If a list of values (separated by spaces) is specified then successive words, starting at address, are updated to value1, value2 etc.

This command must be the last or only command on the input line. The command terminates if an invalid value is encountered.

Examples of this command are:

```
*S1234 367
```

which sets location 1234 to 367.

```
*S#X3A48 123 #XFF -654 'A
```

which sets location 3A48 (hex) to 123, 3A49 (hex) to FF (hex), 3A4A (hex) to -654 and 3A4B (hex) to the ASCII code for ‘A’.

```
*SG300 0 0
```

which sets globals 300 and 301 to 0.

```
*SV4 1 2 3
```

which sets variables 4, 5 and 6 to 1, 2 and 3 respectively.

The format of the second form of the ‘S’ command is simply:

```
S address
```

(This must be the last or only command in the line). The address (in decimal and hex) and its contents (in decimal, hex and character formats) are displayed followed by the prompt ‘:’.

The available options at this stage are:

ESCAPE
Terminate the command and return to the '*' prompt.

RETURN
Leave the contents of this address unchanged. Display the next address and contents in a similar way.

value **RETURN**
Update the contents of this address to value and display the next address and contents in a similar way.

value1 value2 etc. **RETURN**
Update the contents of this address to value1, of the next address to value2 and so on. If n values are specified the next address to display is the current address + n.

The command is terminated if any value is invalid.

In the following examples computer output is underlined, **RETURN** is denoted by <CR> and **ESCAPE** by <ESC>.

```
*S5260<CR>
  5260 (148C):   16965 4245 EB :123<CR>
  5261 (148D):  -12345 CFC7 GO :<CR>
  5262 (148E):       0 0000 .. :#X56A2 26 26<CR>
  5265 (1491):    2560 0A00 .* :<ESC>
*
```

The above example sets locations 5260 (or 148C hex) to 123, 5262 to 56A2 hex, 5263 to 26 and 5264 to 26. The previous contents of 5260 were 16965 (or 4245 hex or the characters 'EB'). Note that non-printing characters are shown as '.' except for CR and LF which are shown as '*'.

The next two examples show how the command can be used to alter globals and variables.

```
*SG353<CR>
 G353  (0DE3):    4321 10E1 a. :0 -888<CR>
 G355  (0DE5):    -888 FC88 .| :<CR>
 G356  (0DE6):       0 0000 .. :-888<CR>
 G357  (0DE7):      -1 FFFF .. :<ESC>
*

*SV3<CR>
 V 3   (1FA9)       66 0042 B. :'A<CR>
 V 4   (1FAA)       67 0043 C. :'B<CR>
 V 5   (1FAB)        0 0000 .. :<ESC>
*
```

Bases

Numbers are normally input and output in decimal. It is possible to input a number in other forms:

'a   enters the ASCII equivalent of the character ‘a’;

#ooo   enters an octal number ooo;

#Xhhhh enters a hexadecimal number hhhh.

The default output format can be altered as follows:

$C requests character format, with two characters per word. In this format most non-displayable bytes are shown as ‘.’ but carriage control characters are shown as ‘*’;

$D re-establishes decimal as the default base;

$O sets the default base to octal;

$X sets the default base to hexadecimal.

Once the format has been changed the new format is used as the default until the format is changed again.

Breakpoints

The system supports nine breakpoints, which can be set at any instruction. It also supports a breakpoint which can be set on the return from the current procedure.

The command 'Bn' sets breakpoint n to the current variable (if n is omitted it is taken as 1) while '0Bn' clears the breakpoint. '0B' clears all breakpoints.

The command 'R' sets the breakpoint on return from the current procedure, and '0R' clears it. Breakpoints must be cleared before being reset, thus the command sequence to reset the breakpoint on return is '0R R'.

Breakpoints must be set on the first byte of a CINTCODE instruction. It is thus convenient to set a breakpoint on the first instruction of a procedure. 'G300B7' would set a breakpoint at the first instruction of a procedure referenced by global 300. Alternatively the memory search function can be used to find a procedure start address, e.g. 'MP myproc B8'.

Other instruction locations can be found, either from the trace displays or by displaying instructions as described below in 'Displays'.

Note that it is not possible to set breakpoints on library procedures in the BCPL ROM.

Running to a breakpoint

The command 'F' is used with breakpoints. By itself it causes the program to run to the next breakpoint, or to the end if no breakpoint is encountered. The command 'nF' causes the program to run until the nth breakpoint is met.

The command '?' obtains a report on the current settings of the breakpoints.

Breakpoints also stop processing if they are encountered while tracing.

Calculating

The DEBUG facility provides a method of performing the simple calculations and number processing normally required when debugging programs. These can use decimal, octal or hex numbers, as explained in 'Bases' above.

The commands all operate on the current variable and the result (in the current variable) can be displayed in the current default base by the command '='.

The calculating commands are:

```
+n  Add n
-n  Subtract n
*n  Multiply by n
/n  Divide by n
%n  Obtain the remainder after dividing by n
<   Shift left one bit
>   Shift right one bit
&n  Logical AND with n
|n  Logical OR with n
!   Obtain the contents of the address pointed to
    by the current variable
.   Equivalent to !
```

One line may contain several commands. If so they are executed from left to right. The following examples illustrate the use of several commands in a line:

- to display the octal equivalent of 1000 decimal:

  ```
  *1000 $O =
  ```

  This enters 1000, selects octal as the output format and then displays the current variable (1000) in octal.

- to display 9999 shifted left one bit in hex:

  ```
  *9999 < $X =
  ```

- to display the contents of a word in the area pointed to by global 444:

  ```
  *G444 + 5 ! =
  ```

  This sets the current variable to the contents of global 444, adds 5 and then replaces the current variable with the content of the word it points to, i.e. the contents of the fifth word in the area. The current variable is then displayed.

Note that to enter a negative number '0-n' rather than just '-n' must be entered. The latter subtracts n from the current variable.

Counting

The counting feature enables the user to execute a known amount of processing before returning to the DEBUG state. This may be useful in running a program to the stage at which it is desired to use other debugging features, and when working with a program which loads and unloads code.

The feature works by counting the number of branches (including procedure calls and returns) made by the program. Provided a program is run in an identical environment with identical input then the count is repeatable (i.e. it is always the same at a given point in the program). Note that slight differences in the heap can cause variations in the count (because GETVEC may have more, or less, work to do) and therefore when running to a known point the count may only be an approximate guide (unless the system is rebooted before each run and an identical sequence of operations carried out).

The count is only maintained when a program is being run under DEBUG. It is reset when a program is started or initialised (by the INIT command).

The command '?' displays the count.

'nC' processes for n increments of the count, i.e. allows the execution of n branches.

'0C' processes until **ESCAPE** is pressed or the program reaches a breakpoint or terminates. Even if the program is one which disables trapping on **ESCAPE**, when it is run by '0C' **ESCAPE** returns to DEBUG.

Note that when interrupting '0C' by **ESCAPE** there may be an interaction between the program detecting **ESCAPE** and DEBUG detecting **ESCAPE**. This manifests itself by the following sequence of events occurring:

(1) **ESCAPE** traps back to DEBUG which produces the 'RUN STOPPED' message (indicating it has detected **ESCAPE** being pressed);

(2) the program is continued by '0C' or 'F' etc.;

(3) the program traps immediately back to DEBUG which produces the 'Interrupted' message;

(4) the program is continued again and runs normally.

Displays

The display of store

The command ':' displays the next eight words from the address pointed to by the current variable, while the command ':n' displays the next n words. The output format follows the current default, so store can be examined in character, octal, decimal or hexadecimal format. The command does not change the current variable and so the same area of store can easily be displayed in two or more formats. e.g.

```
*$X:20$C:20
```

The display can be terminated by **ESCAPE**.

Other displays

The other displays provided are as follows:

D Display the current stack (in the same format as produced by the utility STACK described below).

DI Display 12 CINTCODE instructions following the current variable address. The current variable is set just after the last instruction shown, so that further instructions can be displayed by repeated use of 'DI'. 'CINTCODE' in chapter 11 contains a list of all the CINTCODE instructions.

Limits

It is often convenient to set limits for tracing or for statistics collection. This permits investigations to be concentrated on the code under development and can avoid the display of details of the execution of library procedures.

Limits can be set by the command ‘Ln’. This sets the lower limit at the current variable and the upper limit at n. If n is zero no upper limit is set.

For example:

*G340L10000

would set a low limit at the start of the procedure corresponding to global 340, and an upper limit at word address 10000.

*0L0

would clear both limits.

Once set the limits will apply to any trace or statistics collection function until they are altered. The limits are initialised to the top and bottom of the heap.

The command ‘?’ displays the current limits.

Memory search

There are two memory search commands, ‘M’ searches for a number or a bit pattern, while ‘MP’ looks for a procedure by name.

M word, mask, limit

searches from the current variable to the limit for an address whose content matches the given word for all bits specified in the mask. The specification of the mask and limit is optional. If the limit is not specified the search continues to the top of store. If the mask is not specified all bits must match.

```
MP myproc
```

searches for a named procedure myproc. The linked store files are searched first followed by the library in ROM. The order in which the store files are searched is not guaranteed. If the procedure is found its address is output. Comparisons are only made on the first seven characters of the procedure name, since these are all that are held in store.

Note that not all the procedures in ROM are compiled with names. Those that are not can be accessed by global number. See the section 'Global vector' in chapter 11.

Overlays

The DEBUG facilities are introduced when the file DEBUG has been loaded into store. This must stay in store throughout the time DEBUG is being used. However many of the debugging functions are provided by separate overlay segments, since this minimises the use of store.

The overlays available to DEBUG are:

- INSTR which implements displays and traces of CINTCODE instructions;
- STATS which supports the collection of statistics on program operation. This also creates a vector to hold the results;
- TRACE which implements jump and procedure traces.

DEBUG loads and links the overlay required automatically when a function it provides is requested. Only one of these overlays is linked at any time, and the current overlay can be unlinked by the command 'W'. If the STATS overlay has collected some statistics, these would be lost if the overlay were deleted. To prevent accidental deletion, requests for functions in other overlays are superseded until STATS is unlinked, typically by the command 'W'.

When DEBUG is exited it always unlinks the current overlay but leaves it as a protected file. When unlinking overlays at other times (i.e. in response to 'W' or because a different overlay is needed) the overlay file is given the protection status it had when it was linked. Thus when leaving DEBUG and not intending to re-enter it it is advisable to type 'W' to ensure that all the DEBUG files are deleted when the heap space is needed.

In some cases there may not be room for the running program and the overlay required. DEBUG reports if it cannot find or cannot load one of these overlays when required.

If the program is designed to expand to use the available space, it may be desirable to load DEBUG and a required debugging overlay using the built in command LOAD before running the program.

Statistics

The statistics collection facilities may be used to investigate the operation of a program, by recording the number of times that different parts of a program are executed. This helps to indicate where any effort to optimise the program should be concentrated. Statistics collection may show errors in the program, for example if some code is never executed. It may also be used to confirm that a test procedure tests all program paths.

Two levels of statistics collection are supported:

- The branch level collects statistics on the number of times each branch in the program is executed. A branch is considered to be any sequence of instructions ending in a procedure call, jump, conditional jump, or return.

- The procedure level collects statistics only on the number of times each procedure is entered.

Statistics are collected on a maximum of 100 branches or procedures. The area of code examined can be specified by setting the limits. If statistics could be collected on more than 100 branches or procedures within these limits then data on only the first 100 is retained. In this case the high limit is automatically changed to cover the range of addresses actually being used.

The commands controlling the statistics facilities are as follows:

N introduces the statistics facility, and selects branch level statistics collection.

K in the statistics facility selects collection only at the procedure level.

J in the statistics facility restores branch or jump level statistics collection.

T collects statistics on the number of branches or procedure calls specified in the preceding count. Thus ‘1000T’ collects information during 1000 branches or procedure calls. Only information within the current limits will be saved, but several passes of the program may be used with different limits to build up a complete picture.

0T collects statistics until **ESCAPE** is pressed, a breakpoint is reached, or the program ends.

D displays the collected statistics. The display indicates procedure entry points, and within procedures it gives some information on the program logic by decompiling the CINTCODE.

Z clears the collected statistics.

W unloads the statistics facility.

It is often convenient to use breakpoints to determine the period over which statistics are collected.

Traces

Three levels of tracing of program execution are provided:

I introduces tracing at the CINTCODE instruction level. (See the list of CINTCODE instructions in chapter 11.)

J introduces tracing at the branch level, where a branch is a continuous sequence of instructions ending with a procedure call, a jump, a conditional jump or a return.

K introduces a trace on procedure calls and returns. The trace includes an indication of the nesting level.

When one of these levels has been selected it is possible to trace for a number of displayed lines specified by 'nT' where n is the count of the number of lines to trace. '0T' traces until **ESCAPE** is pressed or the next breakpoint is reached.

Tracing only takes place between the limits currently selected. Thus it is possible to trace the operation of a single procedure in a complex system.

‘W’ unloads the current tracing facility, if any.

It is often convenient to use the breakpoint facilities to reach the point at which it is desired to begin tracing. The count facility using ‘C’ can also be used to run code for which tracing is not required, without losing control.

GLOBALS - display global vector

GLOBALS is run by

!GLOBALS

It displays each initialised entry in the global vector. If an entry contains the address of a procedure the procedure name is displayed otherwise the value of the global is displayed.

The display can be aborted by **ESCAPE**.

HEAP - display heap

HEAP is run by

!HEAP

It displays the contents of the heap (in ascending address order), showing each vector and, if possible, what it is being used for.

Note that the addresses shown for vectors are two less than the addresses used by the program. For example if a program called GETVEC and the value returned was 5874, then the HEAP display would show a data vector at 5872.

The display can be aborted by **ESCAPE**.

IO - display I/O streams

IO is run by

!IO

It displays each stream that exists, identifying the current run state and trap state streams. If possible the most recent characters read from or written to the stream are displayed.

The display can be aborted by **ESCAPE**.

STACK - display stacks

STACK is run by

!STACK

to display the current run state stack, or

!STACK ALL

to display all the run state stacks.

The display gives the address and size of the stack and the number of words used. For each procedure call in the current chain the following information is shown:

- the stack pointer;

- the return address;

- the name (or address) of the procedure;

- the first few items (up to four) on the stack. These items are the parameters of the procedure followed by the local variables. See the section 'Stacks' in chapter 6 for more details.

The display can be aborted by **ESCAPE**.

TESTPRO - test a procedure

This utility allows a single procedure to be tested in isolation. The parameters to the procedure may be specified and the result of calling the procedure is displayed.

The procedure may be called with up to six parameters. The procedure may be specified by name, by global number or by address. The parameters may be specified as global variables, numeric values or strings.

Unlike the other debugging aids, TESTPRO runs as a run-state program. This allows the use of all the DEBUG facilities to aid the testing of the procedure.

TESTPRO may also be useful for experimenting with the library procedures or for executing commands from the console that would otherwise have to be executed by a special program (e.g. to change the display colours the procedure VDU (in LIB) can be run from TESTPRO).

To enter TESTPRO type

```
!TESTPRO
```

TESTPRO prompts for input with the character '#'.

Before entering TESTPRO the procedure to be tested must be linked by the built in command 'LINK file-name'.

If intending to use DEBUG in conjunction with TESTPRO then the following sequence of commands should be used:

```
!LINK file containing procedure
!LINK TESTPRO
!INIT
!DEBUG
*
```

Having entered DEBUG a breakpoint could, for example, be set on the procedure and then 'F' used to run to this breakpoint. This would then start TESTPRO and the '#' prompt would be issued. See the section on 'DEBUG' earlier in this chapter for more details.

If using TESTPRO with DEBUG to experiment with a procedure in the BCPL ROM then the breakpoint facility cannot be used (a breakpoint cannot be set in ROM).

The following sequence may be used, however:

```
!TESTPRO
#T procedure to be tested
#P parameters for procedure
#ESCAPE
Interrupted
!DEBUG
*K 10T
TRACE
R
```

This returns to DEBUG having just called the procedure to be tested. (Note that if the procedure is to be traced then the limits for tracing must be altered to include the BCPL ROM.)

Note that TESTPRO always reselects the console streams after calling the specified procedure, and therefore if it is used to call SELECTINPUT or SELECTOUTPUT the effects of the call are immediately nullified.

Commands

TESTPRO has six commands:

D   displays the current TESTPRO parameters;

H   displays a list of the available commands;

P   specifies the parameter(s) to be passed to the procedure under test;

R   runs the procedure under test;

T   specifies the procedure to be tested;

X   exits.

The D command

This command simply displays the procedure being tested and the current values of the six parameters (strings longer than 10 characters are truncated).

The P command

This command specifies the parameters to be passed to the procedure being tested. Up to six parameters may be specified. Once a parameter has been set up that value is retained until it is altered by a subsequent P command.

The format of the command is:

nP value1 value2 ...

n specifies the first parameter to be updated. It must be in the range 1 to 6 and may he omitted (when 1 is assumed). No spaces are allowed between n and P. The values are assigned to the parameters in order.

Each value may be either a decimal number, a hex number (preceded by '#X'), a global variable (a decimal or hex number preceded by 'G') or a string (in double quotes).

Examples are:

#P 10 #X2A01 G500 "ABCDE"

which sets the first four parameters to 10, 2A01 hex, the value of global variable 500 and the string "ABCDE".

#4PG#X200

which sets parameter 4 to the value of global variable 512 (200 hex).

#5P "string" "STRING"

which sets parameters 5 and 6 to strings.

The R command

This command calls the procedure with the specified parameters. On return from the procedure the result and the values of the globals RESULT2 and MCRESULT are displayed in both decimal and hex.

Having run the procedure once a second R command will run it again with the same parameters. Alternatively the procedure to be tested and/or one or more parameters can be altered first.

The T command

This specifies the procedure to be tested. There are three forms of this command:

T name tests the procedure with the specified name. The linked store files are searched first then the library in ROM. Only the first seven characters of the name are significant.

T Gnnn tests the specified global procedure. nnn is decimal unless preceded by '#X' when it is hex.

T nnn tests the procedure whose word address is nnn. nnn is decimal unless preceded by '#X' when it is hex.

Examples are:

```
#T myproc
#T G115
#T G#X73
#T 5868
#T #X16EC
```

which test the procedure "myproc", global 115 (which is actually MODE), global 73 hex (=115), the procedure at word 5868 and the procedure at word 16EC hex respectively.

The X command

This command exits from TESTPRO. If TESTPRO is re-entered then the procedure to be tested and all non-string parameters are remembered (providing TESTPRO remains in store). Thus it is possible to exit from TESTPRO, run another program and resume testing without having to re-enter the parameters.

8 Discussion

This section discusses a number of different features of the system. They are discussed in alphabetical order of topic.

The following topics are included:

Converting BCPL programs

Coroutines

File and stream handling

Input and text processing

Machine code

Optimising code

Output formatting

Portability

Standardisation

Start-up options

Use of static variables

User-defined characters

CONVERTING BCPL PROGRAMS

Because of the portability of BCPL code it is often possible to convert large BCPL programs to the CINTCODE environment rapidly and easily. The two most time-consuming parts of the process are likely to be the obtaining of the source code on the BBC Microcomputer environment and the testing to confirm that the program still runs correctly in the new environment.

Both these stages are very dependent on individual circumstances. This discussion therefore concentrates on the intermediate stages, where similar problems may be found in many conversions.

While the source of the program is being transferred to the BBC Microcomputer, and converted to ASCII code if necessary, it will usually be found convenient to divide the source code into separate files, each containing one section of not more than 250 lines. This limits the time taken to edit and compile each file.

Typical conversion requirements then include:

(1) Adjusting the header file or any other global definitions to ensure that the first global used is not less than 250. It may be decided to include declarations of the required standard library procedures in the header for the program.

(2) Removing procedures which are now included in the runtime library.

(3) Ensuring that each section accesses the correct header file or files. Typically a section of a program "ABCD" would start with:

```
SECTION "ABCD4"

GET "LIBHDR"
GET "ABCDHDR"
```

It is best to declare the program specific header file after LIBHDR to permit the program to redefine names used in LIBHDR. (Alternatively ABCDHDR might include the GET "LIBHDR" so each program section would just GET "ABCDHDR".)

(4) Examining any FINDINPUT, FINDOUTPUT, DELFILE and RENAME calls and choosing appropriate names for the new system.

(5) Examining any overlaying features, and modifying these, if necessary, to use LOADSEG, GLOBIN etc.

(6) Examining the way the command string is provided for the program. The convention, in this implementation, is that the command string is provided as the currently selected input stream. It can then be analysed by RDARGS. Some other systems provide the command text as a string parameter of START. If this or any other different convention is used it requires conversion.

(7) Discovering if the code includes GETBYTE, PUTBYTE, PACKSTRING and UNPACKSTRING. If so the source code in OPT may be included, and the procedures declared as globals in the program's header file. Alternatively the relevant code from OPT may be included in each section referencing one of these procedures.

(8) Compiling the sections of the program. Common problems include:

   (a) A section is too large to compile. Compilation space may be saved by creating a private header file containing only the definitions used. However usually the section should be split into smaller sections, and additional globals declared if necessary to link the new sections. See also the discussion of this problem in the section on the compiler in chapter 4.

   (b) Some procedure names are not declared. These were usually in the common library of the previous system. It is necessary to discover the original function of these procedures and to provide alternatives. The procedures are often equivalent to library procedures such as CAPCH, DELFILE or RENAME, but have different names.

   (c) Single quotes have been used round strings. In some implementations strings could be declared in single quotation marks e.g.

   ```
   'This used to be an acceptable string'
   ```

   In standard BCPL the string must be given full quotation marks as follows:

   ```
   "This is an acceptable string"
   ```

   (d) The end of section marker, a period '.', is omitted at the end of a section. This compiler requires an end of section marker before it will accept a further section.

(9) When all sections have compiled, the program can be linked using JOINCIN (and using NEEDCIN if appropriate to include procedures from LIB) and tested. One problem at this stage may be excessive use of stack or storage. The correcting options include:

   (a) increasing the stack size available by the declaration of STARTINIT.

   (b) overlaying the program using LOADSEG, GLOBIN and GLOBUNIN. This is discussed under 'Loading and linking code' in chapter 6.

   (c) modifying the program logic or vector sizes to reduce the store requirements.

COROUTINES

Purpose

There are many computer applications in which it is desirable to arrange that two or more separate tasks share the use of the computer, while each task proceeds at its own pace with little knowledge of the state of the other tasks.

The tasks may be completely independent, they may be parts of the same application where it is desired to give part of the application priority when it needs it, or they may be parallel processes possibly working with different areas on the display.

On larger computers these requirements are met by complex operating systems which can switch control between different tasks without the task code knowing that the switch has taken place. Such systems use a significant proportion of the computer's power, and although they are very convenient for independent tasks they introduce considerable complexity into tasks which need to work with the same data, or to interact with each other.

Coroutines provide a much simpler method of scheduling tasks, by requiring the individual tasks to specify when the change of control is to take place. Though the involvement of tasks in their own scheduling has disadvantages, it can be very convenient when the different tasks use the same data, since each task can choose the stages in its processing at which the data may be altered by another task. Between these stages it can assume that the data remains unaltered by any other task.

Coroutines may also be used to provide a clearer structure for some programs which do not require division into separate tasks. For example one coroutine may extract data from a file and pass it on, item by item, to a second coroutine which generates a display or printout.

Method

BCPL programs are always executed using an area of store known as the stack to hold both local variables and the current sequence of procedure calls. Coroutines are each given a separate stack, but otherwise run in the same runtime environment.

A coroutine suspends itself by calling a procedure which switches processing to another stack. Thus while waiting, the current state of the coroutine is held in its stack, terminating with the procedure which switched stacks. When the coroutine receives control again its stack becomes the current stack and the procedure which suspended the coroutine returns, causing the processing of the coroutine to resume from the point at which it paused.

Coroutines are usually executed in a hierarchy. One coroutine can call another coroutine using CALLCO, and it then becomes the parent of the called coroutine. When the called coroutine suspends itself using COWAIT, control returns to its parent. The called coroutine may, itself, call another coroutine which will have the first coroutine as a grandparent.

Coroutines may also call each other at the same level using RESUMECO. In this case the called coroutine takes the same parent as the calling coroutine, and may return control to the parent using COWAIT. There are thus two ways of scheduling a number of coroutines of similar priority. In one case the scheduling is organised by a master coroutine, which calls each lower level coroutine, and receives back control when the lower level coroutine waits. In the other case the lower level coroutines can pass control between themselves using the RESUMECO procedure.

In all transfers of control between coroutines a single parameter can be passed. This is specified in the procedure call which suspends the current coroutine, and appears as the return from the coroutine procedure in the coroutine which is resumed. Since both coroutines are in the same environment this parameter may be a pointer to a larger data area.

The system establishes the main coroutine before entering START, but any other coroutine must be created by the running program. A coroutine is created by the procedure CREATECO which requires two parameters:

- a procedure to be entered the first time the coroutine is called;

- the stack size for the stack associated with the coroutine.

The procedure is entered the first time the coroutine is called. A return from the procedure is equivalent to a COWAIT and leaves the coroutine ready for the procedure to be entered again the next time the coroutine is called.

In this mode of operation the parameter provided by CALLCO becomes the parameter of the coroutine procedure, and a reply can be provided by a RESULTIS command.

The stack size required depends very much on the processing performed by the coroutine. A very simple process may require less than 100 words, but it is better if possible to allow at least 250 words to start with.

The system always traps stack overflow, and if it does occur it is possible to use the DEBUG facilities to display the stack to find the processing that caused the problem. However it is normally preferable to allocate a generous size of stack since the DEBUG facilities can also be used to indicate approximately how much of the stack has actually been used. This permits the size of the stack to be adjusted during the later stages of development.

The coroutine procedures

CREATECO	creates a new coroutine and returns a reference to the coroutine for use in any subsequent calls.
DELETECO	deletes a specified coroutine.
CALLCO	transfers control to a specified coroutine which can return control to the calling coroutine by COWAIT. CALLCO is in section "CORTNS" of the library file LIB.
COWAIT	suspends a coroutine and returns control to its parent.
RESUMECO	transfers control to a specified coroutine which takes on the parent of the calling coroutine. RESUMECO is in section "CORTNS" of the library file LIB.

FILE AND STREAM HANDLING

BCPL defines standard stream organisation procedures which are available on all BCPL implementations, and are a major factor in enabling BCPL programs to be transferred from one environment to another.

These procedures are:

FINDINPUT	which initialises a stream for reading;
SELECTINPUT	which causes all input to be taken from the selected stream until the next call of SELECTINPUT;

INPUT	which returns the identity of the current input stream for use in later calls to SELECTINPUT;
ENDREAD	which closes the current input stream;

and a similar set of procedures for output:

FINDOUTPUT SELECTOUTPUT OUTPUT and ENDWRITE.

This implementation also provides a number of other compatible procedures:

DELFILE	to delete a file from store or the current filing system.
DELXFILE	like DELFILE, but specifying the device (store or filing system) explicitly rather than in the file name.
EXTSFILE	to extend a store file using data in a vector.
FILETOVEC	to convert a file into a vector.
FINDXINPUT	like FINDINPUT for a file, but specifying the device (store or filing system) explicitly rather than in the file name.
FINDXOUTPUT	like FINDOUTPUT for a file, but specifying the device (store or filing system) explicitly rather than in the file name.
READ	to copy a file into store.
READVEC	to read a file into a vector.
RENAME	to rename a store or filing system file.
SAVE	to copy a file from store.
SAVEVEC	to copy a vector to a file.
VECTOFILE	to convert a vector into a file.

The operating system device and file handling routines can also be called directly using OPSYS and CALLBYTE.

Many BCPL systems support a procedure REWIND, which re-initialises the current input stream so that reading restarts at the beginning of the file. This is not provided in this implementation, but the same effect can be achieved by ENDREAD followed by a new FINDINPUT for the same file.

This implementation uses a common naming convention for the identification of devices and of files on the file structured devices i.e. store and the current filing system.

The section 'Devices and files' in chapter 2 explains this convention.

Adapting to the current stream

For most purposes all input streams are equivalent, and once the stream has been created by FINDINPUT or FINDXINPUT, the program need not distinguish between reading from the console and reading from a file.

Similarly different kinds of output streams can also be treated in the same way.

However there are cases in which it is useful to distinguish different kinds of device, and so the procedure TESTSTR is provided to enable the program to determine some of the characteristics of the current input and output streams.

TESTSTR enables the program to determine:

- (STYPE.INT) if the stream is from an interactive device to which prompts might usefully be sent. This can be important to allow a utility to adapt its action depending on whether it is being run directly from a console, or from a command file (but the system cannot distinguish between the console and a *EXEC command file);

- (STYPE.TERM) if the stream can output part lines (i.e. if characters written to the stream are displayed immediately rather than being buffered up until '*N' is written).

In this implementation both characteristics are true for the console (device /C) but for no other device.

The procedure TESTFLAGS provides a means of discovering some information on the current state of physical input. Two tests are supported.

The test for CONSOLE.KEY discovers if another character is ready to be read from the console. It is normally used when the console is being read as device /K. The test has two main purposes:

- to allow programs which interact with the console to continue processing between input characters. The program will test fairly frequently for console input to avoid loss of input characters;

- to allow programs to be stopped by pressing any key. Such programs can test for an input character less frequently.

The test for MORE.INPUT allows the program to determine whether further characters can be read before any physical input is required. This can be useful when discarding input from the console after an error.

In this implementation MORE.INPUT is only appropriate on the console input stream, it does not examine the operating system buffers for other devices.

In some circumstances it may be useful to determine the device associated with a stream. If s is a stream identifier (as returned by FINDINPUT, INPUT etc.) then s!5 is a number in the range 1 to 7 which identifies the device as follows:

1 /K (keyboard);
2 /C (console);
3 /P (serial port);
4 ERRORSTREAM;
5 /L (printer);
6 /N (null device);
7 /S. (store file);
8 /F. (current filing system file).

Use of store files

The ability to use store files allows many operations to be performed much faster and more conveniently than would otherwise be the case. This is particularly so when considering the procedures that allow files to be converted into vectors and vice versa.

The use of store files allows the most efficient use to be made of the various filing systems available.

Some filing systems (e.g. tape) are inherently slow and are restricted to accessing one file at a time. Store files help in two ways:

- files can be read into store and then re-used as required without having to be read from the tape (or other device) each time;

- programs can use store files as buffers for files that are eventually to be written to tape. Thus a program can write two files simultaneously and copy them one at a time to tape when they are both complete.

Other filing systems are much faster (e.g. disk, Econet), but some operations are less efficient than others. With both the examples mentioned reading/writing a complete file in one operation (using the OSFILE routine) can be up to 10 times faster than reading/writing it a byte at a time (using OSBGET/OSBPUT) or a block of bytes at a time (using OSGBPB). By using store files a file can be copied into store then read from there (a byte at a time) by the program. Similarly a program can write to a store file then copy the file to the filing system in one operation.

The procedure READ copies a file from the filing system into store using OSFILE if possible. The procedure SAVE copies a store file to the filing system, again using OSFILE if it can. The built in commands READ and SAVE allow files to be copied to and from store directly by the user.

Another use of store files is to copy files from one filing system to another (the operating system does not, in general, permit files to be open on more than one filing system simultaneously). For example to copy a file from disk to tape the following sequence of commands could be used:

```
!*DISK
!READ MYFILE
!*TAPE
!SAVE MYFILE
!
```

Two points to beware of are:

- the current filing system should not be changed if any files are open;

- the READ and SAVE commands do not copy or set up the file's load and execution addresses. Thus they should be used only for files where this does not matter (such as all files which are part of the BCPL system).

Converting between files and vectors

Often it is convenient to treat a set of data items either as a file (e.g. so that formatting procedures such as WRITEF or analysis procedures such as RDITEM can be used) or as a simple data area (e.g. to update an item in the middle of the set).

The BCPL system provides a set of procedures which allow easy conversion between files and vectors. The only restrictions are that the first two words of the vector must be reserved for system use and that in many cases the vector involved must be a heap vector (as obtained by GETVEC) i.e. it cannot be a data area taken from the stack by the VEC statement.

Two procedures are provided for converting a file into a vector:

- READVEC copies the contents of a file into a vector supplied by the program. The file is not affected in any way by the operation.

- FILETOVEC converts a file into a vector. In this case the vector is supplied by the system rather than the program. If the file is a filing system file then it is not affected by the operation, but if it is a store file then it is destroyed (because FILETOVEC uses the area that held the data within the file for the vector).

Three procedures are provided for converting a vector into a file:

- EXTSFILE adds the data in the vector to the end of an existing store file. The vector becomes the property of the system i.e. the calling program may not re-use it.

- SAVEVEC copies the contents of a vector to a file. The vector is not affected and remains the property of the calling program.

- VECTOFILE converts a vector into a file. The vector becomes the property of the system. If the file is a store file then the vector is used as the data part of the file.

Note that the advantage of FILETOVEC and VECTOFILE when dealing with store files is that the same heap space is re-used (although FILETOVEC for a non-contiguous file effectively copies the file first and so enough heap space must be available to do this).

An example of the way some of these procedures are used is the screen editor. This allocates itself a large vector and reads the file to be edited by READVEC. READVEC is also used when reading a file to be copied into the file being edited. Saving part of the file being edited as another file is accomplished by SAVEVEC. Finally when all edits are complete VECTOFILE is used to create the output file (if this is a store file the use of VECTOFILE means that very little extra heap space is needed, whereas writing out the vector by SAVEVEC or by normal stream I/O would need as much extra heap space as the size of the file).

INPUT AND TEXT PROCESSING

The procedures provided in this implementation can give considerable support to the programming and analysis of character input.

This section introduces the procedures and their expected use.

The input procedures

General input

RDCH — reads the next character from the current input stream, it converts the two end of line characters carriage return and line feed into the newline representation used by BCPL, which is represented by the single character '*N'. If cursor editing is disabled then the **COPY** key can be used to give ENDSTREAMCH when reading from the keyboard.

RDBIN — reads the next character from the current input stream but makes no assumptions about the form of the character.

READWORDS — provides a method of reading a large number of characters. It is similar in effect to repeated use of RDBIN, but is significantly faster when reading word-aligned data from files.

UNRDCH — steps back the input stream so that the next byte read will be the same as the last. This is very effective in structuring input processing when the terminator of one field, such as a number, may be required by the next stage of processing.

If RDCH or RDBIN read past the end of the input, they return the constant ENDSTREAMCH (-1) which cannot represent a valid character.

Field conversion

RDITEM	reads the next item from the input stream as a string. Items may be separated by space, ';', '=', or a newline.
READN	reads a decimal integer from the input stream.

Input processing

RDARGS	provides a set of powerful facilities to extract arguments from an input stream in accordance with an argument specification.

Character processing

CAPCH	takes a character and transforms any lower case code to the equivalent upper case character.
COMPCH	compares two characters, ignoring the difference between upper and lower case.

String processing

COMPSTRING	compares two strings. It can match equivalent strings and place strings in alphabetical order, ignoring the difference between upper and lower case.
FINDARG	discovers if a string corresponds to one of a set of keys.

SPLIT — extracts a string from a larger string, starting at a given point in the string and ending just before a specified delimiting character.

Comments

When compared with output processing, these procedures provide few options for converting input text or strings into numbers. This arises because of the wide variety of possible requirements, and because it is not difficult to create procedures to conform to individual requirements.

Suitable code of READN, for example, is:

```
LET READN() = VALOF
$( LET sum, ch, negative = 0, ?, FALSE

rdnxt:
   ch := RDCH()
   SWITCHON ch INTO
   $( CASE '*S': CASE '*C': CASE '*N': CASE '*T':
         GOTO rdnxt  // ignore leading spaces and
                     // format characters

      CASE '-':     negative := TRUE
      CASE '+':     ch := RDCH()
   $)

   WHILE '0'<=ch<='9' DO
   $( sum := 10*sum + ch - '0'
      ch  := RDCH()
   $)
   UNRDCH()  // next read to be first char.
             // after number

   IF negative DO sum := -sum
   RESULTIS sum
$)
```

Many realtime input processes are designed to interact with the user at his keystation. However such interaction is not appropriate if the process is receiving its input from a command file rather than directly from a keyboard. The program can discover whether the current input is interactive by using TESTSTR to test STYPE.INT. For example:

```
UNLESS TESTSTR( STYPE.INT) DO commanderror()
```

When interacting with a keyboard it is often convenient to output a prompt, and then accept input on the same line. It is possible to test if the current input stream is from a terminal which permits this by using TESTSTR to test STYPE.TERM.

MACHINE CODE

There can be a variety of reasons for wishing to access native machine code from the CINTCODE environment.

- a machine code implementation of a few critical subroutines may be essential to provide the performance needed;

- it may be desired to use an existing procedure in assembler;

- it may be desired to use a procedure provided by the operating system;

- events or interrupts may need to be handled.

However, because of the flexibility of BCPL it is rare for a procedure to be implemented in assembler because it cannot be expressed in BCPL. For example bits can be packed in a word using the '&', '|', NEQV and shift operators, and it is possible to write to or read from absolute byte locations using the '%' operator.

The BCPL system provides four different methods of entering machine code:

- the procedure OPSYS can be used to call the operating system OSBYTE procedure;

- the procedure CALLBYTE can be used to call machine code at a fixed byte address, passing parameters in the 6502 registers. This is intended mainly for use in calling other operating system procedures (e.g. OSWORD). The section on CALLBYTE in chapter 5 gives details of the parameter interface. Note that the address can be declared using a manifest constant e.g.

```
MANIFEST $( OSWORD = #XFFF1 $)

RESULT := CALLBYTE( OSWORD, 0, BUFFER << 1)
```

- the procedure CALL can be used to call machine code at a word address, passing parameters in the 6502 registers. If the machine code has been assembled using the 6502 assembler RAS supplied with the BCPL system then the code can be linked automatically to the global vector and the name of the corresponding global variable can be used in the CALL. The section on CALL in chapter 5 gives details of the parameter interface;

- a machine code procedure can be called as if it is a BCPL global procedure i.e. the calling code need not be aware of whether or not the procedure being called is in CINTCODE or machine code. To achieve this the machine code procedure must be assembled using RAS and must begin with one of two special bytes. This is the preferred method.

It is expected that machine code will normally be produced by the relocatable assembler, RAS. This assembler generates relocatable code that is loaded and linked in the same way as CINTCODE. The assembler can be used in three ways:

- to generate relocatable code which is called from BCPL as if it were CINTCODE;

- to generate relocatable code which is called from BCPL using CALL;

- to generate code to run at a fixed address.

Code called like CINTCODE

A machine code procedure can be written so that it can be called from CINTCODE with up to three parameters. It can return a result if appropriate. A specified machine code 'fault routine' (to handle faults generated by the operating system) can optionally be swapped for the standard BCPL fault routine when the machine code is entered and the standard fault routine replaced when the machine code returns to BCPL. The swapping of fault routines is discussed in more detail below.

The machine code procedure has its entry point declared as a global using the GLOBAL directive. The first byte of the procedure must be:

0D hex procedure is called without swapping fault routines;

or

CF hex procedure is called with fault routines swapped.

Note that neither of these bytes is a valid 6502 instruction code and so any machine code calls to such a procedure must skip past the first byte.

The machine code for the procedure follows this special byte and the procedure terminates with an RTS instruction.

The entry conditions are:

parameter 1:	min X/Y registers (low byte in X) and also in page 0 bytes 52/53 hex (low byte in byte 52 hex);
parameter 2:	in page 0 bytes 54/55 hex;
parameter 3:	in page 0 bytes 56/57 hex.

The exit conditions are:

result:	in X/Y registers (low byte in X).

<u>Example</u>

The following code defines a procedure to search a BCPL string for a given character starting from a specified position. It returns the character position or -1 for failure. The start position is not checked to be within the string.

```
        SECTION "FINDCH"

FINDCH  GLOBAL  300

TEXT    EQU     $52      ; string
STPOS   EQU     $54      ; start position
CHAR    EQU     $56      ; character

MAXIND: DS      1        ; no. of chars in string

        PROC    "FINDCH"
FINDCH: DB      $0D      ; special byte
        ASL     TEXT     ; convert to byte addr
        ROL     TEXT+1
        LDY     #0
        LDA     (TEXT),Y ; no. of chars in string
        STA     MAXIND
        LDY     STPOS    ; index to first char

LOOP:   LDA     (TEXT),Y
        CMP     CHAR     ; character found ?
        BEQ     FOUND
        CPY     MAXIND   ; end of string ?
        BEQ     NOTFND
        INY
        BNE     LOOP     ; unconditional branch

FOUND:  TYA              ; return position in
        TAX              ; X (lo)/ Y(hi)
        LDY     #0
        RTS

NOTFND: LDX     #$FF     ; return -1
        LDY     #$FF
        RTS

        END
```

This procedure could be called by the following BCPL code:

```
SECTION "TESTFC"
GET "LIBHDR"

GLOBAL $( FINDCH:300 $)

LET START() BE
   TEST FINDCH("ABCDEFAB", 3, 'A') = 7 THEN
      WRITES("OK*N")
   ELSE
      WRITES("Not OK*N")
```

Swapping fault routines

The processing performed by the standard BCPL fault routine is covered in chapter 6. Briefly, if a fault is generated while the system is executing machine code then control returns to the CINTCODE that called the machine code routine (as if the machine code had returned in the normal way). The calling routine can tell if a fault has occurred by checking if the top byte of MCRESULT is FF hex (but for this to be effective the calling routine must ensure that the top byte of MCRESULT is not FF hex before calling the machine code routine).

In some cases the machine code that is called may need to substitute its own fault routine for the duration of the call. It could, of course, do this directly by saving the current fault routine address, storing the address of its own fault routine in the fault routine vector (202/203 hex - see the BBC Microcomputer User Guide) and restoring the original address just before returning. The BCPL system provides a much easier method, however.

When entering machine code starting with the special byte CF hex, the system checks whether the global FAULTROUTINE (global 205 declared in SYSHDR) is defined. If it is then the contents are assumed to be the word address of the fault routine to be used and the equivalent byte address is set up in the fault routine vector.

On return from machine code which started with either 0D hex or CF hex the system restores the fault routine address from locations 40E/40F hex. These locations are initialised to the byte address of the standard BCPL fault routine.

If a user-written fault routine is permanently to replace the BCPL fault routine then the byte address of this routine should be stored in locations 40E/40F as well as in the fault routine vector. If this is done the user must ensure that his fault routine handles faults caused by operating system calls made by BCPL library routines in the same way as the standard BCPL fault routine does (in particular that MCRESULT is set up as required). It is also the user's responsibility to ensure that the file containing his fault routine remains linked by linking it as a system file.

Note that the standard BCPL fault routine also resets the fault routine vector from locations 40E/40F hex.

The BCPL system provides several special ways for user-written fault routines to exit. The following possibilities exist:

- The fault routine may be able to unwind the stack and jump back into the code that was executing when the fault occurred.

- The fault routine might be able to unwind the stack to the state it was in on entry to the machine code and so return to BCPL in the usual way (by RTS).

- The fault routine can jump to the standard BCPL fault routine (or its permanent replacement) by:

  JMP ($40E)

  The BCPL fault routine resets the fault routine vector and resumes the CINTCODE as described above.

- The fault routine may return to the CINTCODE instruction following the call to machine code by:

  JMP ($410)

  This also resets the fault routine vector from $40E/40F.

- The fault routine may return to the CINTCODE instruction following the call to machine code (but leaving the fault routine vector unchanged) by:

  JMP ($402)

The following example shows a procedure (FAULTY) which returns if called with a parameter of 0 but causes a fault if it is called with any other parameter, and a fault routine (FTPROC) which displays the message 'FAULT'. This fault routine is only in effect while FAULTY is being executed.

```
        SECTION "FAULTY"

FAULTY  GLOBAL  300
FTPROC  GLOBAL  205     ; FAULTROUTINE

OSASCI  EQU     $FFE3

MESS:   ASC     "FAULT*C"
        DB      0

FTPROC: LDY     #0      ; fault routine - output
LOOP:   LDA     MESS,Y  ; message
        BEQ     MSDONE  ; terminated by 0 byte
        JSR     OSASCI
        INY
        BNE     LOOP    ; unconditional branch
MSDONE: JMP     ($410)  ; simplest way back to
                        ; original caller


        PROC    "FAULTY"
FAULTY: DB      $CF     ; special byte
        CPX     #0      ; is parameter 0 ?
        BNE     GENFLT
        CPY     #0
        BNE     GENFLT
        RTS             ; normal return

GENFLT: BRK             ; generate fault

        END
```

Code to be accessed by CALL

The machine code procedure has its entry point declared as a global using the GLOBAL directive (as shown in the examples above). However there is no special byte preceding the code. The procedure terminates with an RTS instruction. The section on CALL in chapter 5 describes the method of passing parameters and returning a result.

There is no provision for automatically swapping fault routines when entering machine code by CALL.

Code to run at a fixed address

This is created using the PHASE and DEPHASE assembler directives (see chapter 10). Note that although a file is generated containing code which will execute only at the specified address it is the responsibility of the user to ensure that the code is loaded at the correct address.

There may be cases in which it is desired to convert the relocatable format provided by the assembler to a fixed format. This conversion is not supported in this implementation, because the requirements are rare and diverse. However users can create their own utilities using the information on the relocatable format given in chapter 10.

Common features of machine code

This section contains miscellaneous information that may be of use to all writers of machine code routines that are to be called from BCPL.

Calling operating system routines

In general machine code can call the operating system routines as required, providing the following restrictions are observed:

- The OSRDCH source device and the OSWRCH destination device should not be changed unless they are later restored to their original values. This can be done by immediately following the call to machine code by:

  ```
  SELECTINPUT( INPUT())
  SELECTOUTPUT( OUTPUT())
  ```

- If any events are enabled, they should be disabled before returning to BCPL unless the machine code includes a permanent event handling routine.

- The display mode must not be changed.

Calling other machine code procedures

If one machine code procedure needs to call another in the same assembly unit then JSR can be used in a straightforward manner after setting up the parameters in the appropriate way. (When calling a procedure beginning with one of the special bytes 0D or CF then the JSR must be to the procedure entry point + 1).

If a machine code procedure needs to call another machine code procedure in a different assembly unit then the global vector linkage must be used. The recommended method of doing this is illustrated by the following example. The machine code routine PROCA is calling the machine code routine PROCB which is global 300. It is assumed that PROCB does not begin with one of the special bytes 0D or CF. The extra line needed if it does is commented out. If necessary PROCA could examine the first byte of PROCB to see whether or not it began with a special byte.

```
PROCB   GLOBAL  300

; Modify address in JSR instruction.  (Need only
; be done once while the called address remains
; constant).

PROCA:  LDA     PROCB   ; lo byte of word addr
        ASL     A       ; convert to byte addr
        STA     LCALL+1 ; storein JSR instr
;       INC     LCALL+1 ; skip special byte if
;                       ; necessary
        LDA     PROCB+1 ; repeat for hi byte
        ROL     A
        STA     LCALL+2

; Include code here to set up any parameters.

LCALL:  JSR     0       ; ends up as JSR to
                        ; PROCB
```

<u>Page 0 usage</u>

Bytes 52-59 hex and 70-8F hex are available for use as required.

<u>Restrictions on use of relocatable addresses</u>

The relocation mechanism only allows for relocating words holding byte addresses (see the section on CINTCODE in chapter 10). Thus statements such as:

```
WDFRED: DW      FRED>1  ; word address of FRED
```

or

```
        LDA     #<FRED  ; low byte of FRED
```

where FRED is relocatable (e.g. a label) cannot be used.

The solution is to define a word containing the byte address of FRED and then set up the required values at runtime. For example:

```
WDFRED: DS      2
BYFRED: DW      FRED    ; byte address of FRED
        CLC
        LDA     BYFRED+1
        ROR     A       ; shift hi byte right
        STA     WDFRED+1
        LDA     BYFRED
        ROR     A       ; repeat for low byte
        STA     WDFRED
```

or

```
BYFRED: DW      FRED    ; byte address of FRED
        LDA     BYFRED  ; low byte of FRED
```

Serial re-usability

All machine code should be serially re-usable. That is, it should be capable of being executed several times in succession without being reloaded from a storage device (e.g. disk, tape). The reason is that a file containing relocated machine code may be loaded into store, linked, used and unlinked and then left in store for some time. The next time the file is required it may still be in store.

The main feature to avoid is relying on pre-set data which is then changed. Thus code such as:

```
COUNT:  DB      0
        INC     COUNT
```

will work the first time the code is run, but if it is subsequently run without being reloaded from a storage device then COUNT will start not at 0 but at whatever value it had at the end of the first run.

Another feature to avoid is changing the contents of relocated data since this will cause incorrect values to be set up the next time the code is relocated. Thus a statement such as:

```
BYFRED: DW      FRED
```

where FRED is relocatable, actually stores the byte offset of FRED from the start of the machine code hunk in the location BYFRED. This offset is used by GLOBIN to set up the correct value in BYFRED when the code is linked. Similarly GLOBUNIN resets BYFRED to contain the offset. If the machine code changes the contents of BYFRED then when the code is unlinked GLOBUNIN will store the wrong offset in BYFRED and if the code is relinked without being reloaded from a storage device then GLOBIN will set up the wrong value.

<u>Use of BCPL data areas</u>

The addresses of BCPL data areas can be passed as parameters to machine code for the machine code to examine or modify. Remember that all addresses in BCPL are word addresses and that either the calling BCPL procedure or the machine code procedure should convert the addresses to byte addresses by shifting left one place.

Machine code can access data in the Global Vector directly by using the GLOBAL directive. For example the following code sets global 300 to -2:

```
G300    GLOBAL  300

        LDA     #<(-2)
        STA     G300    ; low byte
        LDA     #>(-2)
        STA     G300+1  ; high byte
```

OPTIMISING CODE

In many cases no optimising of CINTCODE is required and programmers will find that the CINTCODE produced by the compiler is fast enough and sufficiently compact. However in some cases there are good reasons to seek to optimise the code. The aim of the optimisation may be:

- to minimise code size;

- to maximise performance;

- to minimise the use of stack.

Though the BCPL CINTCODE compiler includes optimisation to minimise the number of instructions used to execute the program as written, it does not attempt to optimise the sequence of processing and so the programmer is able to relate the code produced to the BCPL instructions.

Because CINTCODE has been designed to implement well-structured BCPL, programmers will normally find that improving the structure of their code reduces code size and increases performance.

In particular:

- CINTCODE procedure calls are compact and fast, particularly if the procedures have only one or two parameters, and references to the first few local variables for a procedure are efficient.

- CINTCODE can retain the last variable in a register between operations. Thus if several operations are to be performed using one variable it is usually more efficient if these are coded together rather than being separated by other operations.

Factors likely to reduce CINTCODE size

(1) Declare the most used local variables first, and declare any local vectors after any integers, since operations on the first few variables declared are compact and efficient.

(2) Use local variables rather than global variables, and global variables rather than static variables.

(3) When the code has been debugged, compile it with the no names option. This saves five words per procedure.

(4) Use library procedures in preference to developing special and possibly more appropriate procedures.

(5) Reduce the size of the global vector. The default size allows for global numbers up to 767. If a program only uses global numbers up to 500, say, then the space reserved for globals 501-767 could be returned to the heap. Note that once the global vector size has been reduced the only way to increase it again is to restart the system. The following program reduces the size of the global vector (readers interested in the mechanism used should refer to the description of the heap in the Appendix):

```
SECTION "CHOPGV"
GET "LIBHDR"
GET "SYSHDR"

LET START() BE
$( LET GVADDR = (@MAXGLOB)-2
   LET NEWMG = READN()  // read highest global
                        // required
   LET ENDGV = (GVADDR+NEWMG+4) & #XFFFE

   UNLESS FIRSTFREEGLOBAL <= NEWMG <= MAXGLOB
      DO STOP(11)       // invalid parameter
```

```
    MAXGLOB := NEWMG
    UNLESS ENDGV = GVADDR!0 DO
    $( ENDGV!0 := GVADDR!0
       ENDGV!1 := 0
       GVADDR!0 := ENDGV
    $)
 $)
```

Factors likely to increase execution speed

(1) Minimise transfers to slow speed peripherals, e.g. by the use of store files, or by holding data in store. In particular use READ to copy an entire file into store before processing it and similarly write output files to store then use SAVE to copy them to disk/tape etc.

(2) Use READWORDS and WRITEWORDS where appropriate.

(3) Collect statistics on the system using the DEBUG statistics facilities, and concentrate optimisation on the most used procedures and loops.

(4) Use assembler for very frequently executed procedures.

(5) Take as much processing as possible out of frequently executed loops.

(6) Minimise the use of loops. Initialisation of a data area can often be done with MOVE, MOVEBYTE, BACKMOVE or BACKMVBY e.g.

```
FOR i=0 TO 50 DO databuffer!i := 0
```

is much slower than:

```
databuffer!0 := 0
MOVE( databuffer, databuffer+l, 50)
```

Repeated calculation such as:

```
FOR i=1 TO 4 DO ptr!i := i*3
```

is slower than:

```
ptr!1, ptr!2, ptr!3, ptr!4 := 3, 6, 9, 12
```

(7) Avoid multiplication and division if addition or shifts can be used instead.

(8) Minimise calls to GETVEC and FREEVEC. These are best for obtaining 50+ words of store for long periods. Local vectors and APTOVEC are faster, but require stack space.

(9) Use the procedure VDU in preference to repeated calls of WRBIN/WRCH to output to the screen.

Factors likely to reduce the stack used

(1) Break processing into separate procedures called in turn from a root procedure.

(2) Declare variables with the minimum necessary scope, using blocks within procedures.

(3) Find the condition using most stack, and reduce its usage.

(4) Use data areas outside the stack.

(5) Use a loop within a procedure instead of recursion.

(6) Do not include procedure calls in the parameters of other procedure calls.

```
proc1( a, b, proc2())
```

uses more stack before calling proc2 than:

```
c := proc2()
proc1( a, b, c)
```

OUTPUT FORMATTING

A variety of procedures are provided to output numbers and text. Programmers can provide additional procedures if their requirements are not met by the standard range. A flexible procedure, WRITEF, enables a selection of the standard output procedures to be applied according to a format string.

The individual formatting procedures

NEWLINE writes a newline, generating a line feed and carriage return in the output stream.

NEWPAGE writes a form feed character.

WRBIN writes a given binary character.

WRCH writes a character, converting a newline character to line feed/carriage return.

WRITEA writes a word address, converting it to a procedure name if appropriate.

WRITEBA writes a byte address as a word address followed by 'H' if the High byte is addressed e.g. 21466H.

WRITED writes a decimal number in a given field width.

WRITEDB	writes a decimal double word positive integer.
WRITEF	writes a number of values in different formats according to a format string. It is discussed in more detail below.
WRITEHEX	writes a number in hex.
WRITEN	writes a number in decimal in the minimum appropriate field width.
WRITEOCT	writes a number in octal.
WRITES	writes a string.
WRITET	writes a string in a specified field width.
WRITEU	writes a number as an unsigned integer (i.e. in the range 0 to 65535).
WRITEWORDS	provides a method of writing a large number of characters. It is similar in effect to the repeated use of WRBIN, but is significantly faster when writing word-aligned data to files.

Only the most frequently used of these procedures are held in the BCPL Language ROM. These are WRBIN, WRCH, WRITEA, WRITEBA, WRITED, WRITEF, WRITEHEX, WRITEN, WRITES and WRITEWORDS. The remainder are available as individual sections in the library file LIB, and should be included in any programs that need them.

The use of WRITEF

WRITEF is specified in the procedure definitions in chapter 5. It uses the output procedures described above.

This discussion illustrates some of the ways WRITEF can be used.

(1) It may be used with an explicit format string, thus

```
WRITEF("The HEX equivalent of %N is %X4",
       1000, 1000)
```

would cause the output:

```
The HEX equivalent of 1000 is 03E8
```

(2) The format string may include newline characters (*N), and the parameters may be calculated, thus

```
WRITEF("DECIMAL  HEX*N%I7  %X4*N%I7  %X4",
       n, n, 2*n, 2*n)
```

would cause the output, with n=1000, of:

```
DECIMAL  HEX
   1000  03E8
   2000  07D0
```

(3) The format string may be declared once and used as required:

```
LET argerror =
   "*NERROR IN ARGUMENT NUMBER %N: %S"

WRITEF( argerror, argnumber, argstring)
```

could cause the output:

```
ERROR IN ARGUMENT NUMBER 3: Too Long
```

(4) The format string can be varied, and applied to the same set of parameters. Thus an error routine might be:

```
AND errorroutine( errorformat) BE
$(
   WRITEF( errorformat, line, character,
           step, datasize)
   .
   .
$)
```

For this kind of use the format may include '%$' which skips the next parameter.

Graphics

The BBC Microcomputer provides various graphics facilities accessed through sequences of control codes sent to the VDU driver. The procedure VDU gives a convenient method of sending such sequences from a BCPL program. The procedure MODE allows the appropriate display mode to be selected.

PORTABILITY

In the rapidly changing technology of small computer systems, it is natural that programmers should aim to make the results of their efforts portable to computer systems other than that for which the program was first developed. With the BCPL CINTCODE system portability can be achieved at any of three levels.

(1) Easy conversion. The source code can easily be converted to a new environment.

(2) Source code portability. The source code only needs recompiling to run in the new environment.

(3) CINTCODE portability. The compiled CINTCODE can be run unchanged in the new environment.

This discussion suggests some rules to assist in achieving each level of portability and then describes the reasons behind these suggestions. The suggestions are designed to obtain the greatest possible portability. If the required portability is known to be restricted to a limited class of systems, such as one processor range, some of the rules can be relaxed.

Suggestions for easy conversion

- Keep all use of device and file names, and of the procedures DELFILE, DELXFILE, FINDINPUT, FINDXINPUT, FINDOUTPUT, FINDXOUTPUT, LOADSEG and RENAME in one section.

- Keep all handling of any initial command text in one section.

- Keep any direct use of computer or operating system features localised in a few procedures. Similarly keep any use of non-standard procedures (e.g. READ, READVEC, VDU) localised.

- Do not use device identifiers in GET statements, DELFILE, LOADSEG, FINDINPUT, FINDOUTPUT etc.

- Do not rely on the order of bytes in a word.

- Do not rely on the use of a 16-bit word, for example by testing the most significant bit as a sign bit.

- Never use the binary form for a character, but always declare characters symbolically as, for example 'A' or '9'.

- Do not rely on characters that may not be available in all cases.

- If possible, avoid use of features that cannot be supported on all target operating systems e.g. assuming that console output is displayed as it is written rather than being buffered until a carriage return.

- Avoid use of specific RESULT2 error codes.

Suggestions for source code portability

Follow the suggestions for easy conversion, and in addition:

- Keep names unique in the first six characters.

- Ensure that the program can run in the store available on each target computer, and if necessary design it to adapt to the storage available.

- Use only the library procedures that are marked as CP/M-compatible in the summary of the global vector in chapter 11.

Suggestions for CINTCODE portability

Compatible CINTCODE environments can be provided on a wide range of computers, including 8-bit, 16-bit and 32-bit machines. Portability at the compiled CINTCODE level is inefficient on computers that do not use the ASCII character codes, or on computers that do not provide for the addressing of 8 bit bytes. These inefficiencies affect mainly the older computer designs, e.g. IBM computers based on the 360 series use EBCDIC as their character code, while the DEC series 20, derived from the PDP10, has a 36-bit word and expects to use 6 bits for each character.

For CINTCODE portability follow the suggestions for source code portability, and in addition:

- Minimise the use of static integers, and never address them indirectly. (However static strings can be used freely.)

- Do not use the TABLE construct, unless the TABLE is initialised to zeroes.

Explanation

BCPL has been implemented on a wide variety of computers, and programs in BCPL can be moved easily from one computer to another. This is achieved:

- through the design of the language, which conceals differences such as character sets and word lengths;

- through the use of a common compiler front end, which ensures that each implementation interprets the program in the same way;

- through the use of standard procedures for input and output, which minimise the variations between computers and operating systems.

The Richards Computer Products implementation of BCPL can provide an even greater degree of portability. It is fully compatible with the standard for the language, as defined in October 1979. All implementations will, as far as possible, support the same range of procedures, and in many cases the same compiled code may be run on different computer types.

There are, of course, limits on the extent to which different computers can be made to work in the same way. Many of these can be minimised by care in programming.

Machine characteristics

Unless part of the program is written in assembler, the programmer is often protected by the operating system and the CINTCODE library from needing compatibility with the instruction set of the computer.

However, every program is influenced by the storage provided for its use. To some extent programs can adapt themselves to the storage available (e.g. by selecting suitable buffer sizes and overlay strategies) but each program has its own minimum store requirements.

Word length

This implementation uses a 16-bit word size, and the '%' operator addresses an 8-bit byte.

Programs written for this word size can be run successfully on computers with larger word sizes. Such programs must not rely on the 16-bit word length. For example they must not:

- test the most significant bit of the 16-bit word by comparing the word with zero;

- assume a shift right clears bit 15.

Programs for larger word sizes can be run on this implementation if they do not rely on the precision of the larger word length.

The order of bytes

The 6502 microprocessor assumes that the first byte of an integer is the least significant. Other processors using this convention include the 8080 range, the Z80 and PDP11s.

However processors such as the Motorola 6800 series and IBM computers make the first byte the most significant.

Programmers of portable software should therefore avoid using byte addressing on numeric data, which in any case is normally considered to be bad practice for other reasons.

The most serious effect of byte order is on the portability of compiled CINTCODE. This can contain integers that can be addressed by the program both in TABLES and in static data. Directly addressed data will be assumed to be held least significant byte first, but the system cannot detect all possible accesses to static data and so references to data structures, such as TABLES, will assume that integers are in the format used by the computer being used.

Character sets

BCPL code can be written to be independent of the differences between the most frequently used character sets - ASCII and EBCDIC. The main requirement for portability is that the programmer uses the character symbol, e.g. 'A', rather than its numerical equivalent in one coding system.

If it is desired to run the same compiled CINTCODE on computers with different character sets, the runtime environment must convert each character on input and output. This reduces the performance of any computer whose character set requires adaptation. The block transfer procedures READWORDS and WRITEWORDS do not perform any conversions, and so a program which uses these procedures to handle text must arrange for any necessary conversion of character codes.

Operating systems

Any direct use of a native operating system, for example through OPSYS, requires conversion on moving to a new operating system environment. It aids conversion if all such uses are placed in one section, and are hidden from the rest of the program in user defined procedures. These procedures can then be rewritten to provide the same function in a new operating system environment.

The most common adaptation required for different operating systems is in the names of files and input/output devices in the GET instruction and the procedures FINDINPUT, FINDOUTPUT, RENAME and DELFILE. While all uses of these procedures can often be confined to the root segment of a program, and it is not difficult to change the names on conversion, programs which are intended to be transferred without modification should avoid the use of device names and of non-alphanumeric characters in file names if possible. It is also desirable to ensure that the first six characters of the file name are unique, since some operating systems are limited to this size.

STANDARDISATION

This section summarises the facilities offered by this implementation against the Proposed Definition of the Language BCPL published in October 1979.

BCPL level 0

The implementation is designed to be fully compatible with BCPL level 0. Certain procedures, which are considered to be of mainly historic significance, are only provided in source form. These are GETBYTE, PUTBYTE, PACKSTRING, UNPACKSTRING. They are supplied in the file OPT.

BCPL standard extensions supported

Extension A1
    Character constants *C, *B and *E (but the latter is now used for ‘escape’ rather than its original meaning of ‘output buffered characters’).

Extension A2
    Character operator %.

Extension A4
    Optional compilation using $$tag, $<tag and $>tag.

Extension A6
    SECTION and NEEDS. NEEDS directives can be satisfied by the utility NEEDCIN.

Extension A7
    Store allocation using GETVEC, FREEVEC, MAXVEC and STACKSIZE.

Extension A8
    Scaled arithmetic using MULDIV.

Extension A12
    System services with OPSYS.

BCPL extensions not supported in full

Extension A3
    Field selectors are not supported.

Extension A5
    Compound assignment is not supported.

Extension A9 - Block I/O
    Fast I/O is provided in this implementation by READWORDS and WRITEWORDS which are compatible with some other BCPL implementations, but not with this standard extension.

Extension A10b) - Binary I/O

Binary I/O by RDBIN and WRBIN is not fully supported. RDBIN and WRBIN are provided but without the defined method of separating records in binary streams.

Extension A11

Direct access I/O is not supported.

Extension A13

Floating point is not supported, but is available as an additional package (which conforms to Extension A13b) - Floating Point Procedure Packet).

Extension A14 - Time and Date

TIME is supported. TIMEOFDAY and DATE are not supported.

Extension A15 - External Procedure

A single procedure CALLBYTE is provided to link to native code. However declaration of the procedure by EXTERNAL is not supported, and the use of a commented MANIFEST constant is recommended. Special provisions are made for entering code generated by the relocatable assembler.

START-UP OPTIONS

This section explains how to arrange for a CINTCODE program to be entered automatically when the computer is switched on.

If the computer is arranged to start up with the ROM filing system as the current filing system (this involves changing some links on the keyboard) then if there is a file named 'MAIN' in the ROM filing system the BCPL system will assume this is a CINTCODE file and load and execute it. Note that the BCPL Language ROM must be the right-most language ROM in this case.

If the computer is arranged to start up with any other filing system as the current filing system then automatic running of a CINTCODE program depends on the start-up options provided by that filing system. If the filing system has the option to automatically execute a command file (by *EXEC) then that command file can be used to run the required program (and to enter BCPL first by '*BCPL' if necessary). If the filing system does not have such an option then there is no easy way to automatically run a CINTCODE program.

USE OF STATIC VARIABLES

Some care is needed in writing and developing programs which use static variables or which write to tables.

Static variables and tables are allocated storage space within the CINTCODE file and their initial values are preset by the compiler. If the program changes these values then when the program finishes the new values remain in the CINTCODE file so long as it remains in store. If the program is run again, using the same store file, then the initial values for the new run will be the final values from the old run.

The way to avoid the problem is always to initialise static variables as part of the program initialisation. Thus code such as:

```
STATIC $( COUNT=0 $)

LET START() BE
$(  ...
    ...
   COUNT := COUNT + 1
    ...
    ...
$)
```

should be replaced by:

```
STATIC $( COUNT=? $)

LET START() BE
$( COUNT := 0
    ...
    ...
   COUNT := COUNT + 1
    ...
    ...
$)
```

The same problem occurs with tables and strings if they are changed. The same solution applies.

Very occasionally it may be useful to take advantage of the fact that static variables are preserved between runs of a program so that a program can remember certain features of its environment, provided that it remains in store (e.g. TESTPRO uses this method to remember the procedure being tested and the current parameters).

Note that if such a program is run, thus changing its static variables from their initial value, and is then saved to disk, tape etc. the saved version contains the changed values. Thus if the program is reloaded from the saved version the changed values become the new initial values.

One further point to note is that although local procedure names and labels are technically static variables they may not be assigned to. If it is required to assign to them they must be made global. For example, the following code would not compile:

```
GET "LIBHDR"

LET PROCA() BE
$(   ...   $)

AND PROCB() BE
$( PROCA := WRCH $)
```

whereas if the statement

```
GLOBAL $( PROCA:300 $)
```

were included then it would be valid.

USER-DEFINED CHARACTERS

By default the BBC Microcomputer operating system reserves space for 32 user-defined characters (ASCII codes 224 to 255). The '*FX20' command allows up to 224 of the 256 possible characters to be redefined (see the BBC Microcomputer User Guide for details). The space used for these extra character definitions normally forms part of the BCPL heap and therefore the heap must be moved before defining such extra characters.

It is not possible to move the heap and preserve all store files etc., but a facility is provided whereby a program can restart the BCPL system reserving a certain amount of space. Note that restarting the system deletes all store files, and hence the program itself (but the program can arrange to be automatically re-run as the example will show).

Although this facility is intended for extra character definitions it may be used by a program which wants to reserve a fixed area of memory for its own use.

The use of the facility is best shown by example. The following program, CHARDEF, reserves 768 words (enough for all possible extra character definitions) then re-runs itself:

```
SECTION "CHARDEF"
GET "LIBHDR"
GET "SYSHDR"     // for SYSINDEX and I.xxx

LET START() BE
$( UNLESS SYSINDEX!I.DEFSPACE = 768 DO
   $(
      // SYSINDEX!I.DEFSPACE is number of words
      // reserved or to be reserved.

      SYSINDEX!I.DEFSPACE := 768 // no. to reserve

      // store "CHARDEF" in console input buffer
      // so program is re-run after restart

      FOR I = 1 TO 8 DO
         OPSYS( 138, 0, "CHARDEF*C"%I)

      // reboot system (not by *BCPL)

      CALLBYTE( SYSINDEX!I.RESTART)
   $)
   // space now reserved - make it usable for
   // character definitions

   OPSYS( 20, 1, 0)

   etc.
$)
```

The specific points to note are:

- The number of words to be reserved is stored in SYSINDEX!I.DEFSPACE. This word is initially 0 and so can be tested on entry to CHARDEF to see if this is the first or second entry.

- CALLBYTE to SYSINDEX!I.RESTART restarts the system reserving the space. (*BCPL re-initialises completely i.e. with no space reserved).

- The program can cause itself to be re-run using the *FX138 (OPSYS is equivalent to *FX) command which inserts characters into the keyboard buffer.

- The header file SYSHDR must be included.

Programs using this technique to reserve a data area can find the byte address of the start of the area reserved by:

```
byteaddr := OPSYS(131)
```

9 Getting Started

This chapter describes how to install the BCPL Language ROM and how to get the BCPL system working. It then describes some exercises to introduce a new user to the features of the BCPL CINTCODE system. New users are urged to work through these exercises in the order given (some exercises depend on the results of earlier exercises).

In this chapter it is assumed that the system is distributed on a 40-track disk for use on a Model B with a single disk drive.

Users with 80-track dual disk drives can follow the exercises with no alterations (although they may prefer to use disks in both drives rather than swapping disks in and out of one drive).

Users with tape (but no disks) can follow most of the exercises by using the BCPL tape to load the system programs and their own tape to save the program sources and CINTCODE created. Such users should read the remarks on compiling from tape in the section 'BCPL - the compiler' in chapter 4. Chapter 11 lists the order of the files on the system tape. Note that if disks are subsequently added it is a simple matter to copy the files from the BCPL tape to a disk (using the READ and SAVE built in commands). The files supplied on tape are identical to those supplied on disk.

Users with a Model A should follow the installation procedure but will not be able to perform the exercises involving compilation.

All examples in this chapter use the convention that computer output is underlined. All user input is terminated by the **RETURN** key unless otherwise stated. User input shown in bold type means that the key indicated should be pressed. Thus **COPY** would mean that the COPY key should be pressed, whereas if it were not in bold type it would mean that the characters 'C', 'O', 'P' and 'Y' should be entered.

INSTALLATION AND INITIAL CHECKING

The BCPL CINTCODE system is distributed as a language ROM containing the interpreter and the majority of the library routines plus either a disk or a tape containing various files (see the list below). Before starting the installation procedure check that you have a tape or disk as appropriate and, if you have a disk, that it is in the correct format (40 or 80 tracks) for your disk drive. Also check that your version of the operating system is 1.0 or higher ('*FX 0' displays the operating system version number). If the version is below 1.0 the BCPL system should not be installed.

Installing the language ROM

The BCPL Language ROM may have been installed by your dealer. If not then the installation procedure is as follows:-

(1) Unplug the computer.

(2) Remove the cover by undoing the two screws labelled 'FIX' on the back and the two screws labelled 'FIX' on the bottom (near the front).

(3) Unfasten the keyboard by removing the nuts from the two bolts on the left of the keyboard and the single bolt on the right of the keyboard.

(4) Carefully move the keyboard towards the front of the computer to reveal the four sockets on the front right of the main printed circuit board. (Be careful not to strain the multi-way cable linking the keyboard to the main circuit board while doing this.)

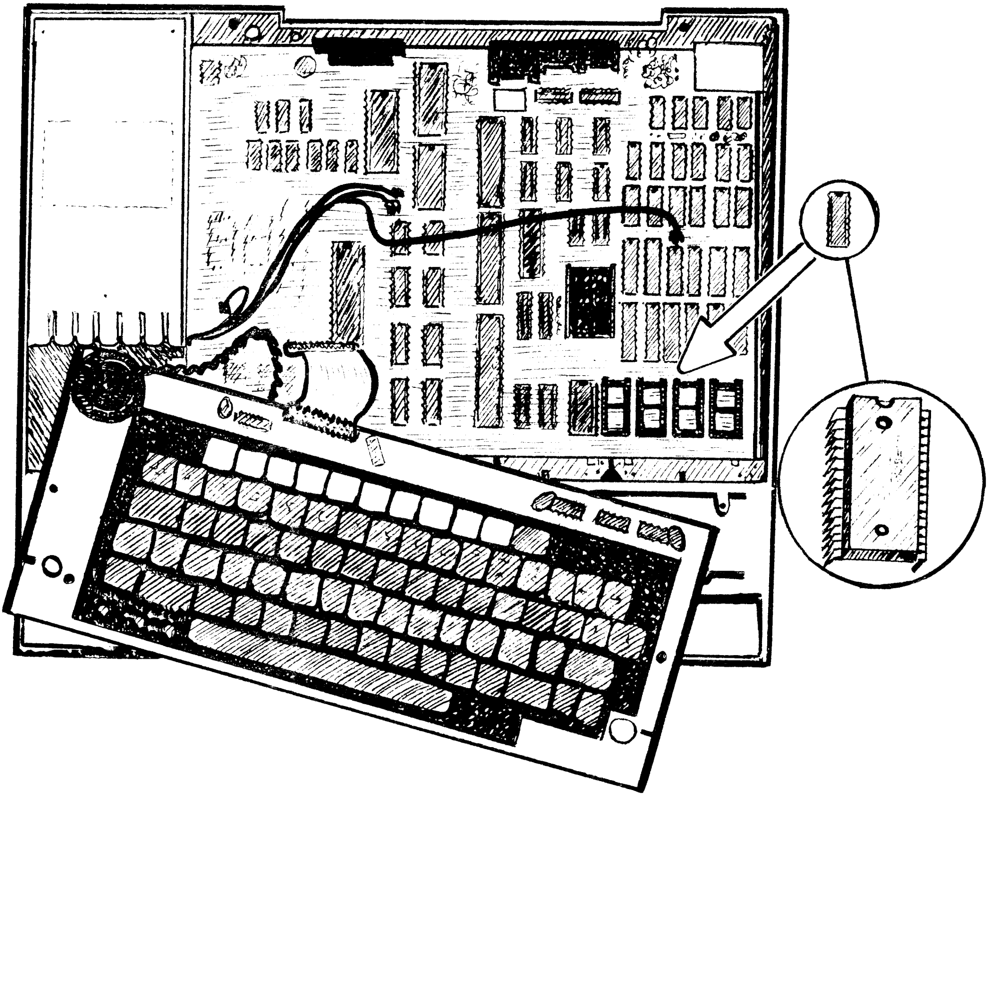
(5) Select a free socket for the BCPL ROM. In most cases one socket will already contain the BASIC Language ROM and another the disk filing system (or perhaps Econet filing system). Any free socket may be used. If the BCPL ROM is inserted to the right of the BASIC ROM then when the computer is powered up it will enter BCPL rather than BASIC. Similarly if the BCPL ROM is to the left of the BASIC ROM then the computer powers up into BASIC.

(6) Insert the BCPL ROM into the chosen socket (see the diagram on page 302). When handling the ROM avoid touching the pins. The ROM must be inserted with pin 0 towards the back of the computer. The pin 0 end of the ROM is indicated by a small cut-out in the body of the ROM. Ensure that the ROM is pushed firmly down into the socket.

(7) Return the keyboard to its original position and secure it with the three nuts.

(8) Replace the cover and fasten it with the four screws.

Once the ROM is installed switch on the computer. If the ROM was installed as the right-most language ROM then the normal 'BBC Computer' message (and, perhaps, filing system message) should appear followed by 'BCPL' and the prompt '!' with the cursor to the right of it.



Inserting the BCPL ROM

If the ROM is not the right-most language ROM then the computer should start up as normal. Typing '*BCPL' (followed by **RETURN**) should enter the BCPL system and 'BCPL' followed by the prompt '!' should be displayed.

If the computer will not enter BCPL check the following points:

(1) The BCPL Language ROM is installed the right way round and is firmly in the socket with no pins bent or misplaced.

(2) The operating system is version 1.0 or higher.

(3) The operating system ROM was not loosened in its socket while installing the BCPL ROM.

Initial checking

Type various characters and see that they are echoed. Check that **DELETE** deletes the last character entered and that **CTRL-U** deletes the whole line.

Enter the command STORE (i.e. type 'STORE' then press **RETURN**). A line such as

```
Data     0    Free 11489 + ... = 11492
```

should be produced (the actual figures may vary). This shows that no data vectors have been allocated from the heap, that the largest vector that may be obtained from the heap is 11489 16-bit words and that the total free heap space is 11492 words. The example figures are typical for a Model B with a disk filing system. Typical figures for a Model B with no filing system (other than tape) are 12897/12900 and for a Model A with no filing system 4705/4708.

Press **ESCAPE.** The message

Error 1017 Escape

should be displayed (accompanied by a 'beep'). **ESCAPE** may be used to interrupt the current program or command. If it is used to interrupt a program then the message

Interrupted

is produced. If **ESCAPE** is pressed after characters have been typed (but before **RETURN**) the characters typed are ignored.

SETTING UP DISKS

This section covers setting up disks ready to use the BCPL CINTCODE system. It should be ignored by users without disk systems.

As a first step the BCPL disk should be copied and the original kept in a safe place. The disk may be copied in the normal way (*BACKUP). All references to the 'BCPL disk' in the rest of this chapter refer to the copy. Note that after *BACKUP has completed an error message is generated of the form:

*** ERROR: X

This message is produced because the disk filing system overwrites the memory used for the BCPL heap. The BCPL system must be rebooted by pressing **BREAK.** Other disk filing system commands that overwrite the heap are *COMPACT and *COPY.

The BCPL disk should contain the following files (in alphabetical order):

BCPL	the root of the BCPL compiler
BCPLARG	a compiler overlay
BCPLCCG	a compiler overlay
BCPLSYN	a compiler overlay
BCPLTRN	a compiler overlay
DEBUG	the main debugging utility
ED	the screen editor
ENCODEB	the source of a simple example program
EX	the command file execution utility
EXAMPLE	the CINTCODE of a complex example program
EXMP1B	the source of one segment of the complex example
EXMP2B	the source of one segment of the complex example
EXMP3B	the source of one segment of the complex example
EXMPHDR	a header file for the complex example
GLOBALS	a utility to display the global vector
HEAP	a utility to display the heap
INSTR	a debug overlay
IO	a utility to display the I/O streams
JOIN	a utility to join files
JOINCIN	a utility to join CINTCODE files
LIB	the CINTCODE library
LIBHDR	the main header file for BCPL programs containing all commonly needed declarations
NEEDCIN	utility to extract sections from a library
OPT	the source of rarely required library procedures
RAS	the relocatable assembler
STACK	a utility to display the current stack(s)
STATS	a debug overlay
SYSHDR	a header file for BCPL programs containing rarely needed declarations
TED	a small version of the screen editor
TESTPRO	a utility to test a procedure
TRACE	a debug overlay.

Since a maximum of 31 files can be held on a disk there is no room for user files on the BCPL disk. One possible method of working is to hold all user files on one or more disks separate from the BCPL disk. For program development it is likely to be more convenient to split the BCPL files over one or more disks, with each disk containing a mixture of BCPL and user files. A split into two disks (a ‘source’ disk and a ‘test’ disk) is suggested, and this split is assumed in the exercises given later in this chapter.

Source disk

This disk contains the BCPL files needed to edit and compile BCPL source. It is used to hold the source of the BCPL programs under development. It should contain the following files copied from the BCPL disk:

```
BCPL      BCPLARG  BCPLCCG  BCPLSYN  BCPLTRN
ED        ENCODEB  EX       EXMP1B   EXMP2B
EXMP3B    EXMPHDR  LIBHDR   RAS      SYSHDR
TED
```

(ENCODEB and the four files concerned with the complex example are needed only for the exercises later in this chapter.)

Test disk

This disk contains the BCPL files needed to link and test CINTCODE. It is used to hold the CINTCODE of BCPL programs under development. It should contain the following files copied from the BCPL disk:

```
DEBUG     GLOBALS  HEAP     INSTR    IO
JOINCIN   LIB      NEEDCIN  STACK    STATS
TESTPRO   TRACE
```

Note that the files JOIN and OPT are not copied to either the source disk or the test disk. These files are only rarely used and are accessed from the BCPL disk when needed.

Creating the source and test disks

(1) Format two disks in the usual way.

(2) Insert the BCPL disk in the drive.

(3) Copy the files for the source disk into store until there is no space left e.g.

```
!READ BCPL
!READ BCPLARG
!READ BCPLCCG
!READ BCPLSYN
!READ BCPLTRN
!READ ED
Error 51
!
```

The READ command copies the specified file from disk into store (see chapter 3). The 'Error 51' message means that there is no space left in store (see the list of error numbers in chapter 11). Typing the command STORE lists the files in store.

(4) Remove the BCPL disk and insert the source disk.

(5) Copy the files in store onto the disk using the SAVE command e.g.

```
!SAVE BCPL
!SAVE BCPLARG
!SAVE BCPLCCG
!SAVE BCPLSYN
!SAVE BCPLTRN
```

(6) Remove the source disk and insert the BCPL disk. Reboot the BCPL system by pressing **BREAK** to delete all the files in store (an alternative is to use the DELETE command e.g.

```
!DELETE BCPL
!DELETE BCPLARG
etc.).
```

(7) Repeat the process for the next set of files required, starting with the file on which 'Error 51' occurred e.g.

```
!READ ED
!READ ENCODEB
etc.
```

When all the files have been transferred check that the source disk is complete by

```
!*CAT
```

If any files have been omitted copy them using READ and SAVE.

(8) Repeat the whole process for the test disk.

CREATING A SIMPLE PROGRAM

This section covers in detail the creation of a very simple program. It is intended as an introductory exercise to the features of the system.

Three stages are involved - creation of the BCPL source, compilation of the source to CINTCODE and testing of the program.

The program merely displays the message 'Hello world'.

(1) Reboot the system by pressing **BREAK**.

(2) Insert the source disk.

(3) Use the screen editor to create a BCPL source file called PROGA as follows:

    !ED PROGA

The screen should be cleared and the message ‘New file’ displayed. Type in the following program, pressing **RETURN** at the end of each line:

```
SECTION "PROGA"
GET "LIBHDR"
LET start() BE
   WRITES("Hello world*N")
```

It is not necessary to copy exactly the use of upper and lower case in the program. Upper and lower case are equivalent (except in the string "Hello world").

The first line gives a name to the section of CINTCODE. The second line specifies that the header file LIBHDR is to be read (this file declares START and WRITES as globals). The third line introduces the procedure START which must be included in every program and is the program entry point. The last line is the body of START and is simply a call to the ‘write string’ procedure. The string is terminated with a new line character (*N), so that when the program is run any output following it (e.g. the next prompt) is on a new line.

(4) Exit from the screen editor by pressing **CTRL-f8** (i.e. pressing the **CTRL** key and function key 8 together). Then:

```
!STORE
 4612 UL  ED
   56     PROGA
Data     0    Free  5329 + ... = 6824
```

(The numbers may differ from this example). This display shows that there are two files in store. One is the editor which is a loaded CINTCODE file (denoted by 'L') and is unprotected (denoted by 'U'). The other is the file just created (PROGA). The figures to the left of the file names show the heap space used by the file.

(5) Save the file to disk by:

```
!SAVE PROGA
```

(This is a precautionary measure. The file need not be saved until the system is powered down or rebooted, but it is good practice to take frequent backups.)

(6) Compile the file by:

```
!BCPL PROGA CODEA
BCPL - RCP V2.1
Section PROGA
Text read
RCP CINTCODE generation
CINTCODE size = 24 words
```

(The messages produced may differ in detail.) This produces a CINTCODE file named CODEA. If the 'RCP CINTCODE generation' message does not appear but instead one or more error messages are generated the reason will be that the file PROGA was incorrectly typed in.

To correct the file return to step (3), but this time the command 'ED PROGA' displays the file that was created. Chapter 4 describes the editor commands available to alter text.

(7) !STORE

This time the files listed include PROGA, CODEA, BCPL and one or more of the compiler overlays. LIBHDR will have been read in by the compiler as an unprotected file then subsequently deleted to make room for one of the later compiler overlays. Note that ED, which was unprotected, has also been deleted to make room for the compiler overlays.

(8) Insert the test disk and save the CINTCODE file by:

!SAVE CODEA

It is always a good idea to save any store files which have been created or changed before testing a new program, since the program might contain bugs which would crash the system.

9) Try the program by:

!CODEA

The message 'Hello world' should be displayed.

FURTHER SYSTEM FEATURES

This section introduces a number of features of the system, using a simple file handling program as an example.

The program used is the example program ENCODEB which has been copied from the BCPL disk to the source disk. This program copies text from one file to another, encoding the text as it does so. The encoding algorithm is simply to replace ‘a’ with ‘z’, ‘b’ with ‘y’ ... ‘z’ with ‘a’. The encoded text can be converted back to its original form by running the program on it a second time.

The program can be displayed by inserting the source disk and:

    !TYPE ENCODEB

The normal methods of controlling the display are available i.e. **CTRL-SHIFT** temporarily halts the display and **CTRL-N** and **CTRL-O** turn ‘paged mode’ on and off.

For convenience the code of the program is reproduced here:

```
SECTION "ENCODE"
GET "LIBHDR"

MANIFEST $( avsize = 20 $)

LET start() BE
$( LET ch = ?
   LET infile, outfile = ?, ?
   LET argvec = VEC avsize

   IF RDARGS("FROM/A,TO/A", argvec, avsize) = 0
      THEN STOP(11)  // invalid arguments

   infile := FINDINPUT(argvec!0)
   IF infile = 0 THEN
      STOP(RESULT2)  // invalid in file
   outfile := FINDOUTPUT(argvec!1)
   IF outfile = 0 THEN
      STOP(RESULT2)  // invalid out file

   SELECTINPUT(infile)
   SELECTOUTPUT(outfile)
```

```
   ch := RDCH()
   WHILE ch NE endstreamch DO
   $( WRCH( codechar(ch) )
      ch := RDCH()
   $)

   ENDREAD()  // not strictly necessary
   ENDWRITE() // but good practice
$)

AND codechar(char) = VALOF
$( TEST 'A' <= char <= 'Z' THEN
      char := 'A' + 'Z' - char
   ELSE IF 'a' <= char <= 'z' THEN
      char := 'a' + 'z' - char
   RESULTIS char
$)
```

When the program is entered it uses RDARGS to read in two file names and then opens the first one for input and the second one for output. If both files are opened successfully they are selected as the current input and output streams respectively. Characters are read one at a time from the input file until the end of the file is reached. For each character the procedure ‘codechar’ is called to convert it to the coded form, which is then written to the output file. Finally both files are closed.

The procedure ‘codechar’ checks if the character is a letter and if so converts it using the rule given above. Upper and lower case are handled separately. If the character is not a letter it is not converted.

Compiling the ENCODE program

(1) Reboot the system by **BREAK.**

(2) Insert the source disk and compile the program by:

!BCPL ENCODEB ENCODE

(3) Insert the test disk and save the CINTCODE file by:

!SAVE ENCODE

Using the ENCODE program

The program is run by a command of the form:

!ENCODE fromfile tofile

where fromfile and tofile are file or device names. The rules for file and device names are given in chapter 2. Briefly any name beginning with '/' is treated as a device and any other name is treated as a file name. The device '/C' is the screen for output and the keyboard read a line at a time for input. Files may be store files or current filing system (e.g disk) files. A file name may be prefixed by '/S.' for store or '/F.' for the current filing system. A file name without a prefix is interpreted as a store file name for output and a store file name (if such a store file exists) or a current filing system file name for input.

Insert the source disk and read LIBHDR into store by:

!READ LIBHDR

then try the following examples of ENCODE, in which all the file/device names used are valid:

(1) !ENCODE LIBHDR /C

which reads the file LIBHDR (which is in store) and displays the encoded version on the screen.

(2) !ENCODE LIBHDR ENCODEDLIBHDR

which reads the file LIBHDR and writes the encoded version as the store file ENCODEDLIBHDR. Use the STORE and TYPE commands to verify that the file has been created and that it is a coded form of LIBHDR.

(3) !ENCODE ENCODEDLIBHDR /C

which reads the encoded version of LIBHDR and converts it back to the original form displayed on the screen.

(4) !*FX 4,1
!ENCODE /C CONFILE
ABCDE
fghij
COPY
!

This example shows how a file may be read in from the console. The first command (*FX 4,1) disables the cursor editing facility. The next line runs ENCODE, specifying that the input is to be read from the console and the output written to the file CONFILE.

The program is now reading from the console so all lines typed in are now read as data rather than as commands. Thus the lines 'ABCDE' and 'fghij' are read as data. A special facility is provided by the BCPL system to allow 'end-of-file' to be read from the console - when cursor editing is disabled the **COPY** key acts as end-of-file. Thus the final line of the example (pressing **COPY** followed as usual by **RETURN**) causes ENCODE to read ENDSTREAMCH from its input stream and terminate.

The file CONFILE may now be displayed using TYPE.

Note that the BCPL system automatically re-enables cursor editing when a program finishes and so there is no need for the user to cancel the effect of the *FX 4,1.

(To issue operating system commands from within a program the procedure OPSYS should be used for *FX commands and RUNPROG for all other commands.)

(5) !ENCODE CONFILE /F.CONFILE

This reads the file just created and writes the encoded version to a disk file called CONFILE. Note that

!TYPE CONFILE

and

!TYPE /S.CONFILE

both display the store file named CONFILE whereas

!TYPE /F.CONFILE

displays the disk file named CONFILE.

If the store file is deleted by

!DELETE CONFILE

then

!TYPE CONFILE

now displays the disk file because it fails to find a file named CONFILE in store.

(6) !ENCODE CONFILE /C

reads the disk file CONFILE and displays the encoded version on the console.

The next example introduces a new device '/K'. This device is input only and reads the keyboard a character at a time without echo. When reading using '/C' each character is echoed (i.e. displayed) as it is typed and no characters are available to the program until **RETURN** is pressed (before **RETURN** is pressed **DELETE** etc. may be used to edit the line). When using '/K' the character typed is not automatically displayed and is immediately available to the program.

(7) !*FX 4,1
!ENCODE /K /C

The first line, as in example (4), disables cursor editing so that **COPY** can be used to indicate 'end-of-file'.

The parameters to ENCODE specify that it is to read from the keyboard a character at a time and is to write the encoded version to the screen. Thus as characters are typed in the encoded form is displayed (e.g. enter 'ABc' and 'ZYx' is displayed).

To exit from the program press **COPY** (no **RETURN** is needed this time).

One further device that is useful if you have a printer is '/L'. This is an output device that directs output to the printer (but not to the screen). It takes note of the printer ignore character that is specified by the *FX 6 command. Output to the screen (device '/C') can be copied to the printer in the usual way by entering **CTRL-B**/**CTRL-C**.

The above examples demonstrate the flexibility of the BCPL input/output system.

The next set of examples demonstrates how errors are handled. In each case running ENCODE with invalid parameters generates an error message including an error number. Chapter 11 lists the meaning of each error number.

(1) !ENCODE

gives error 11 - invalid parameters (none specified).

(2) !ENCODE /C /K

gives error 60 - an input only device (/K) was specified for output.

(3) !ENCODE ZZZZ /C

gives error 27 - no such file (there is no file ZZZZ).

(4) !ENCODE /Z /C

gives error 52 - invalid device (/Z).

(5) !ENCODE LIBHDR LIBHDR

gives error 20 - file already open. The store file LIBHDR is first opened for input then an attempt is made to open it for output which fails.

Interrupting a program and use of TIDY

This section introduces the use of the **ESCAPE** key to interrupt a running program. As a first example enter:

```
!ENCODE LIBHDR /C
```

then press **ESCAPE** while the program is writing to the screen. The message 'Interrupted' is displayed. Examine the store files with the STORE command and note that ENCODE is marked with a 'G', indicating that it is linked to the global vector (thus showing that it was the program that was interrupted) and that LIBHDR is marked with an 'R', showing that it is open for read (a file that is open for write is marked with a 'W').

It is now possible to use built in commands and operating system commands without affecting the interrupted program.

To resume the interrupted program enter:

```
!CONT
```

and output resumes where it left off.

A program can be interrupted by **ESCAPE** even if it is reading from the console. As an example consider what happens when running ENCODE as described in example (4) in the previous section, but forgetting to disable cursor editing first:

```
!ENCODE /C CONFILE
ABCDE
fghij
```

At this point **COPY** still has its cursor editing function and so there is no way of entering ‘end-of-file’. The solution is to press **ESCAPE** and then:

```
Interrupted
!*FX 4,1
!CONT
```

Now **COPY** (followed by **RETURN**) has the desired effect of terminating ENCODE.

Having interrupted a program by **ESCAPE** you may wish to abandon it rather than run it to completion by CONT. This is achieved by the TIDY command. As an example enter:

```
!ENCODE LIBHDR /C
```

and interrupt it by **ESCAPE** while it is outputting.

The STORE command shows that ENCODE is linked, LIBHDR is open for read and 404 words of data are in use (for the stack used by ENCODE). Now enter:

```
!TIDY
!STORE
```

The display now shows that ENCODE is no longer linked (although it is loaded i.e. is in the right format for linking and running), that LIBHDR is no longer open and that no words of data are in use. An attempt to continue the program by CONT gives error 12 - no program to continue.

DEBUGGING

This section uses the example program ENCODE (described in the previous section) to illustrate the use of some of the debugging aids. The debugging aids are fully described in chapter 7. Only a small subset of the facilities available is used in the examples given here. Note that it will probably be helpful to refer to chapter 7 as new commands are introduced for full descriptions of these commands. It will also be helpful to refer to the listing of ENCODE given above.

The principal debugging aid used is the utility DEBUG. This utility is used to monitor the execution of the ENCODE program. Other debugging aids (e.g. HEAP, IO) are introduced as appropriate.

To get the system into the initial state assumed by the examples:

(1) Reboot by **BREAK.**

(2) Insert the source disk and

    !READ LIBHDR

(3) Insert the test disk and

    !READ ENCODE

    (It is assumed that you have created the program ENCODE on the test disk as described in the previous section.)

Entering DEBUG

DEBUG will be used to monitor the operation of ENCODE when executing the command 'ENCODE LIBHDR RDHBIL' (which reads the file LIBHDR and writes the encoded version to the store file RDHBIL).

To initiate the debugging session enter the following commands:

```
!LINK ENCODE
!INIT LIBHDR RDHBIL
!DEBUG
*
```

The first two commands are used to initialise the program ENCODE so that it is ready to run. The LINK command links it into the global vector. The INIT command specifies the parameters. The over-all effect of these two commands is that the program is in the same state as if 'ENCODE LIBHDR RDHBIL' had been entered and then **ESCAPE** had been pressed just as the program was about to start.

The third command runs DEBUG which takes over control of the console. DEBUG prompts with '*'. Commands are given to DEBUG a line at a time, terminated by **RETURN** in the usual way. While in DEBUG **ESCAPE** returns to the DEBUG prompt. The only way to leave DEBUG is by the command 'X'. Thus:

```
*X
!DEBUG
*
```

Tracing and breakpoints

Two of the most useful features of DEBUG are the ability to trace the execution of a program and the ability to set breakpoints. As an example we will trace the execution of ENCODE up to the first call of the procedure 'codechar'.

```
*MP CODECHAR
= 5003
*B
*K
TRACE
*CTRL-N
*0T
```

The 'MP' command searches for the specified procedure, which is found at word address 5003 (this value may be different in different systems). The 'B' command sets a breakpoint at the start of this procedure. The 'K' command selects the 'trace calls and returns' option, which means that each procedure call and return will be reported. The next line sets page mode (so that output is not scrolled off the screen before it can be studied). Finally the command '0T' means start tracing and continue until **ESCAPE** is pressed or a breakpoint is reached.

A two-line display is produced for each procedure call or return. The first line shows the addresses involved and an indication of the 'nesting level'. The second line shows the name of the procedure called (if known) and the first 3 parameters (which may not all be relevant) or the value returned by the procedure (which may not be relevant) and the name of the procedure returned to.

The display generated by the commands above should show:

START called;

> RDARGS called (note that the third parameter is 20 which is avsize) and returns a non-0 value;
>
> FINDINPUT called (shown as FINDINP since procedure names are truncated to seven characters) and returns a non-0 value;
>
> FINDOUTPUT called and returns a non-0 value (note the value of the first parameter for use below);
>
> SELECTINPUT called;
>
> SELECTOUTPUT called;

RDCH called and returns 10 (this is the value of '*N' (new line), the first character in LIBHDR);

CODECHAR called with parameter 10.

At this point the breakpoint is reached. The use of DEBUG displays can now be demonstrated.

Displays

The store displays can be demonstrated by examining the parameter of the call to FINDOUTPUT, which should be the address of a string containing "RDHBIL". If this value was 5123 then the command:

*5123:

displays the eight words from this address in decimal. A more useful display is obtained by:

*$X:

which displays them in hex. The first word is 5206 hex which is in the correct format for the first word of a six-character string (byte 0 of a string contains the character count). Displaying the words in character format by:

*$C:

gives the required result.

The command:

*D

displays the current stack giving a history of the current chain of procedures. The first few parameters and local variables are shown for each procedure. The example display should include a pair of lines such as:

```
5110:  (return to  18610H)  START  ()
   10    8312    8332    5117
```

(The actual numbers may vary). This shows that START was called with the stack pointer at 5110. Parameters start three words after this and local variables immediately follow parameters. Thus since START has no parameters the first local variable (ch) is at address 5113 and contains the value 10. This can be verified by examining memory (in decimal) by:

*$D5113:

Breaking out of DEBUG and use of IO

Currently the program has reached the first call of ‘codechar’. This can be verified by the command:

*?

which displays the current state of DEBUG.

Now run the program (without tracing) until 10 characters have been processed i.e. until the breakpoint on ‘codechar’ is reached for the tenth time after resuming. This is achieved by:

*10F

It is possible to break out of DEBUG to use other utilities then re-enter it and carry on. Exit from DEBUG using the 'X' command and then run the utility IO:

```
*X
!IO
```

This gives a display of all the input/output streams that currently exist. Streams labelled 'Run' are the current streams in use by the program. Thus it can be seen that the current input and output streams are the store files LIBHDR and RDHBIL respectively. The last few characters read or written are displayed as are the characters next to be read (for the input file only). Streams labelled 'Cmd' are the streams in use by the command state (see chapter 2). It can be seen that the last command input was 'IO'.

To look at the characters that have been written so far by the program note the buffer address given by the IO display for the program's output stream. Suppose it is 8348. Re-enter DEBUG and use the examine memory command to look at the buffer in character format:

```
!DEBUG
*$C8348:
```

The first word contains CR/LF (displayed as asterisks). Thereafter the encoded text begins '//Xlkb'. To verify this run the program to completion by:

```
*0B
*0C
```

The first command cancels the breakpoint. The second runs the program until it completes or **ESCAPE** is pressed. The program takes a few seconds to complete. When it does so exit from DEBUG and display the output file produced, noting that the first few characters are as noted above:

```
*X
!TYPE RDHBIL
```

One final point to note is that the STORE display shows that ENCODE is still linked. This is because it finished under the control of DEBUG and was left linked in case further debugging was required. To unlink it and free the data areas used:

```
!TIDY
```

HEAP MANAGEMENT

The BCPL system organises the free memory of the computer into a structure called the heap. When a contiguous area of memory is needed it can be allocated from the heap. It is returned to the heap when it is no longer required. Some of the purposes for which heap space is allocated are:

- store files;
- input/output streams;
- program stacks;
- vectors used by programs (obtained by calling GETVEC).

Various areas are permanently allocated for system use.

The utility HEAP displays the use of the heap. For example:

(1) Reboot the system by **BREAK.**

(2) Insert the test disk and:

    !HEAP

In the example there are eleven allocated areas - three for system use, one for the root stack (the stack used by the BCPL system), one for the global vector, three for streams and three for the file HEAP itself. There are two free areas - a small one in the middle of the allocated areas and a large one at the end.

Heap fragmentation and SHUFFLE

One of the problems with the heap is that it may become fragmented i.e. there may be a large amount of free space split up into many small areas, so that a large contiguous area cannot be allocated. This problem, and the facilities provided to cope with it, will be illustrated using the procedure test program TESTPRO. TESTPRO is fully described in chapter 7. It is used here merely as a convenient method of calling the GETVEC library routine (which allocates vectors from the heap).

(1) Reboot the system by **BREAK.**

(2) Insert the test disk and enter the following commands which create a number of store files:

    !READ HEAP
    !READ IO
    !READ TESTPRO
    !HEAP

    The HEAP display shows the three files are together near the bottom of the heap, which contains one contiguous free area at the top.

(3) We will now use TESTPRO to try and allocate a bigger vector than is available as follows:

```
!TESTPRO
#T MAXVEC
#R
```

TESTPRO is a program which allows a specified procedure to be called with specified parameters. It prompts with the character '#'.

The 'T' command specifies that the procedure to be called is MAXVEC (which returns the size of the biggest heap vector available) and the 'R' command calls it. The result is displayed. Assume it is approximately 9300. Now enter:

```
#T GETVEC
#P 9350
#R
9350 + store needed
!
```

The 'P' command specifies the parameter to GETVEC. It should be about 50 words bigger than the result of MAXVEC.

(4) When GETVEC finds that it cannot allocate the required vector it displays a message showing the size of vector required and enters the command state (the system is now in the same state as if **ESCAPE** had been pressed while the program was running). The user now has the options of abandoning the program (by TIDY) or of making enough space available.

(5) Looking at a STORE display it would appear that deleting IO would make enough space available so try this:

```
!DELETE IO
!CONT
9350 + store needed
```

(6) The allocation still fails. A STORE display shows that although there is enough space in total there is not a large enough contiguous area. A HEAP display shows that TESTPRO and its stack are splitting the free area in two.

A built in command SHUFFLE is provided which attempts to maximise the contiguous heap space by moving all store files towards the bottom of the heap (it also deletes all unprotected files). Since a file cannot be moved if it is linked SHUFFLE is no immediate help and there is no way of obtaining the required vector without abandoning the program and restarting it. Often, however, deleting a file will give enough room and the program requiring the space can be successfully continued by CONT.

(7) Abandon the program TESTPRO then use SHUFFLE to move the file TESTPRO into the space that was occupied by IO. Use STORE and HEAP to observe the effect:

```
!TIDY
!STORE
```

(More space is now free because the stack used by TESTPRO has been deleted.)

```
!SHUFFLE
!STORE
!HEAP
```

Note that all the free space is now once again contiguous.

(8) Allocate the vector required by TESTPRO to verify that GETVEC can now obtain an area of the required size:

```
!TESTPRO
#T GETVEC
#P 9500
#R
```

The result is the address of the vector allocated. Exit from TESTPRO by:

#X

This automatically releases the vector back to the heap.

USE OF EX, JOINCIN & NEEDCIN

This section introduces command files and the utilities which are of use mainly in the development of larger programs. It uses the program EXAMPLE as an example. This program is supplied in both CINTCODE and source form on the BCPL disk. EX, JOINCIN and NEEDCIN are described in chapter 4.

The source of EXAMPLE is split into three files, each containing one BCPL section. A fourth source file is a header file containing declarations used by all the sections. These source files are worthy of study as they contain examples of various useful coding techniques - in particular concerning use of the sound and graphics facilities from BCPL.

The source files are:

EXMPHDR the header file;
EXMP1B contains START and various utility routines. Also contains NEEDS directives for library routines from LIB;
EXMP2B contains procedures to play a tune;
EXMP3B contains procedures to display the voltages of the four analogue input channels.

A command file will be developed to automate the production of the final CINTCODE file from the source files.

Simple use of EX

As an introduction to the execute command file utility EX, create a very simple command file to carry out two commands:

(1) Reboot the system by **BREAK.**

(2) Insert the source disk.

(3) Create a file MYFILE using the built in command COPY as follows:

```
!COPY /C MYFILE
STORE
*CAT
ESCAPE
Error 1017 Escape
!
```

Note that this illustrates a simple method of creating small files directly from the console. **ESCAPE** terminates the COPY and closes the file MYFILE.

(4) Execute the command file by:

```
!EX MYFILE
!STORE
      ... usual, store display
!*CAT
      ... usual disk directory display
!
EX File Terminated
!
```

The commands in the file MYFILE are executed as if they had been typed in at the console. MYFILE is not changed in any way. Note that the STORE display showed a file named $$EX. This is a temporary file used by the system and is deleted when the command file ends. Various other temporary files are used by the system for other purposes. Their names all begin with '$$'.

Command file to produce EXAMPLE

The operations necessary to produce the CINTCODE for EXAMPLE are:

(1) Compile EXMP1B, EXMP2B and EXMP3B to CINTCODE files EXMP1, EXMP2 and EXMP3 and save them on the test disk.

(2) Use the Join CINTCODE utility JOINCIN to merge these three CINTCODE files into one CINTCODE file named TEMP.

(3) Use the Extract Library Sections utility NEEDCIN to copy TEMP to EXAMPLE, extracting various sections from the CINTCODE Library File LIB and including them in EXAMPLE. (The sections required are specified by NEEDS directives in the file EXMP1B.) Save the CINTCODE file EXAMPLE.

Step (1) is performed in three parts - compiling one file and saving the CINTCODE before compiling the next one. This method means that the CINTCODE from one compilation can be deleted before starting the next compilation. If this is not done the third compilation might run out of heap space (depending on which filing systems are installed).

First define a command file to perform one compilation. The command file has two parameters - the number of the source file to be compiled (1, 2 or 3) and an indication of whether the source disk should be re-inserted after saving the CINTCODE on the test disk.

Use ED to create this file, named EXCOMP, as follows:

(1) Reboot the system by **BREAK**.

(2) !ED EXCOMP

The screen is cleared and 'New file' displayed. Type in the following file:

```
.KEY NUMBER/A,MORE
BCPL EXMP<NUMBER>B EXMP<NUMBER>
PAUSE please insert test disk
SAVE EXMP<NUMBER>
DELETE EXMP<NUMBER>
<MORE> PAUSE please insert source disk
```

Then press **CTRL** and function key 8 together to exit from ED and save the file to disk by:

!SAVE EXCOMP

The first line indicates that the command file has two parameters. The first one must always be specified (/A). The second one is optional. The second line performs the compilation. If a parameter name appears in angle brackets then the value of that parameter is substituted. Thus executing the command file with NUMBER as 1 causes the second line to become:

```
BCPL EXMP1B EXMP1
```

The third line uses the PAUSE built in command to suspend the command file while disks are exchanged. The command file is resumed by the command CONT. The next two lines save and then delete the CINTCODE file produced by the compilation. The final line is another PAUSE command to swap the disks back. If the command file is run with no value specified for the second parameter MORE then '<MORE>' is deleted from the file and so the PAUSE is executed. If the value '//' (or REM) is specified for the second parameter, however, then the line becomes:

```
// PAUSE please insert source disk
```

which is treated as a comment and ignored.

We can now create the command file to produce EXAMPLE. Use ED in the same way as above to create the file EXEXMP as follows:

```
EX EXCOMP 1
EX EXCOMP 2
EX EXCOMP 3 //
JOINCIN EXMP1 EXMP2 EXMP3 AS TEMP
NEEDCIN TEMP LIB EXAMPLE
SAVE EXAMPLE
DELETE TEMP
```

Save this file to disk by

```
!SAVE EXEXMP
```

First the file EXCOMP is used to compile the three sections. The third time it is used a second parameter of '//' is specified since we want the test disk left in the drive. JOINCIN is used to join the three CINTCODE files into a file called TEMP and then NEEDCIN is used to create EXAMPLE from TEMP and sections extracted from LIB. Finally EXAMPLE is saved to disk and TEMP is deleted.

To execute the command file:

!EX EXEXMP

then just follow the instructions as they are displayed.

When the command file has finished try the program by:

!EXAMPLE

The program is self-explanatory. (The program uses various different colours. Some adjustment of the controls of monochrome monitors/televisions may be necessary to achieve the best results.)

10 BCPL, CINTCODE and Assembler

This chapter describes the features of the BCPL language supported in this implementation, and then discusses the compact interpretive code known as CINTCODE which is used in this implementation. Finally the 6502 assembler language is described.

THE BCPL LANGUAGE

The book Beginning BCPL on the BBC Microcomputer by Paul Martin published by Acornsoft provides a suitable introduction to the language. A more advanced description is given in BCPL the Language and its Compiler by Martin Richards and Colin Whitby-Strevens, published by the Cambridge University Press. A formal definition of the language is provided by the BCPL Standards Committee.

This section assumes the reader has some knowledge of block-structured languages (e.g. Pascal). It describes the version of the BCPL language supported by this implementation, which is a full implementation of the BCPL standard including many of the optional extensions. The relationship to the BCPL standard is covered in 'Standardisation' in chapter 8.

BCPL provides a general structured way of expressing the logic of a program, and of declaring the data required. However the language is deliberately designed to provide only a basic set of features, since these can be enhanced to any desired extent by the definition of procedures. These features provide the flexibility required for a system programming language. They include bit manipulation, flexible addressing and the ability to handle data in many different ways.

The language standard specifies or recommends a number of general purpose procedures, many of which are included in this implementation. These procedures, and the additional procedures provided, are defined in chapter 5, discussed under appropriate headings in chapter 8 and summarised in chapter 11.

Data

All data in BCPL is held in one or more words. In this implementation a 16 bit word size is used. The contents of a word may be a number, a character, an address, a state TRUE or FALSE, or anything else the programmer chooses to store in it.

BCPL is not a typed language. This means that it is not necessary to tell the compiler what a particular word is being used for. Though the BCPL compiler cannot perform type checks on the programmer's code, the absence of typing does support a number of useful features including:

- the easy linking of separately compiled code;

- the ability to read data without knowing in advance what its structure is;

- the ability to mix types when appropriate (for example if n is a number between 0 and 9 the corresponding character code is '0' + n).

Types of data

There are four types of data - constants, vectors, tables and strings.

A constant is simply a 16-bit value. It can be represented as a number in decimal (e.g. -1234 or 761), octal (#7654), binary (#B1011) and hex (#X1A53), as an ASCII character (e.g. 'A' or 'x') or as a logical value TRUE or FALSE.

The characters single quote, double quote and asterisk have special representations '*'', '*"' and '**' respectively. '*Xnn' gives the character whose hex value is nn (e.g '*X20' gives a space) and '*Onnn' gives the character whose octal value is nnn. There are also special representations for certain control characters - '*N' for newline, '*C' for carriage return, '*T' for tab, '*S' for space (' ' is also valid), '*B' for backspace, '*P' for form feed and '*E' for escape.

A vector is a set of adjacent words. It corresponds to an array in languages such as BASIC. A declaration such as

```
LET myvec = VEC 10
```

reserves an area for the vector and stores the address of the first word in the variable 'myvec'.

The '!' operator is used to access words within a vector - myvec!0 references the first word, myvec!1 the next and so on up to myvec!10 (note that VEC n reserves n+1 words so that vector!0 and vector!n can both be used).

The 'VEC n' statement takes the space for the vector from the stack and n must be a constant. The procedure GETVEC described in chapter 5 allocates vectors from the heap and allows the length of the vector to be determined at run-time.

A table is a set of adjacent words which is initialised by the compiler. For example

```
LET mytab = TABLE 123, -76, 0, 144
```

defines a four-word table. The space for a table is permanently allocated within the CINTCODE.

A string is similar to a table but is preset to hold a string of characters. For example

```
LET mystring = "This is a string*N"
```

The first byte of the string is preset to the number of characters in the string. The characters themselves are stored in subsequent bytes.

The '%' operator can be used to access individual bytes (thus mystring%0 would give 17 which is the number of characters in the string, mystring%4 would give 's' and mystring%17 would give '*N').

Strings too long to fit onto one source line are coded as follows:

```
"A very long string can be coded*
* like this.  The asterisks are NOT*
* part of the string"
```

Data declarations

Data names must be declared before they are used. Names may include letters, numbers and the character '.' but must begin with a letter. They may be as long as required. There is no distinction between upper and lower case letters.

Data may be declared to be manifest, local, static or global.

Manifest data is used by the compiler but does not take any space in the resulting program. It allows meaningful names to be used for constants. For example:

```
MANIFEST $( Monday=1; Tuesday=2 $)
```

Local data is data which is local to a procedure or block (see below). It includes the formal parameters of a procedure. The space for local data is allocated from the stack. The following defines a procedure with three local variables:

```
LET proc( local1, local2) BE
$( LET local3 = 0
   .  .  .
$)
```

Static data is stored with the code. It can be preset by the compiler, and if it is changed by the program it retains its new value when next read. The following declaration defines two static variables:

```
STATIC $( static1 = 0; static2 = -1600 $)
```

Static data is normally declared before any procedures in a section and may then be used by all the procedures in that section.

Global data is stored in a vector, known as the global vector, where it can be addressed by any procedure in the program. Global variables must be declared with their position in the global vector. The following declaration defines three globals:

```
GLOBAL $( glob1:300; glob2:301; glob3:488 $)
```

Although globals may be used for data items, most global variables are used for procedures. If a procedure name has been declared as a global, the system initialises the global word to point to the procedure's entry point. This is used to enable procedures in different sections to call each other.

This implementation permits up to 767 globals, but users should use global addresses above 250 for new globals, to avoid redefining globals allocated by the system.

Segments and sections

The unit of code for compilation or loading is known as a segment. This is held in a file, and is called by the file name.

A segment consists of one or more sections, which are the units of code for compilation. There is a limit on the size of a section which depends on the store available for compilation. However any number of sections may be included in one segment. Static data can only be addressed within the section in which it is declared.

A section may start with a SECTION declaration, such as:

```
SECTION "sectname"
```

and should finish with a period '.'. This period may be omitted at the end of a segment. The section name is used in reporting on compilation.

Code libraries and the NEEDS command

The NEEDS directive allows a section to request other sections from a library of compiled code. One or more NEEDS directives can be included at the start of a section, after the SECTION directive if present, but before any other code. e.g.

```
SECTION "mysect"
NEEDS "libsec1"
NEEDS "libsec8"
```

Only the first seven characters of the section name are checked when satisfying NEEDS directives, so any library section names should be unique within these seven characters.

NEEDS directives are expected to be satisfied using the NEEDCIN utility described in chapter 4.

Data declarations and the GET command

A SECTION contains data declarations and procedures. All variables must be declared before they are used. It is very desirable to be able to declare the same MANIFESTS and GLOBALs to all sections in a program. The GET directive permits another file, often called a header file, to be included in the compilation of a section. Thus typically one or more files of declarations are prepared, and are included in each section of a program. For example:

```
SECTION "prog1"

GET "libhdr"
GET "proghdr"
```

The standard header file LIBHDR defines globals and manifests provided by the system. More rarely used declarations are provided in the header file SYSHDR. For maximum section size the declarations required from LIBHDR and SYSHDR may be combined with the header for the program under development, so that the number of declarations is minimised.

Procedures

The code in a section is organised in procedures and normally a section contains a number of procedures after the initial declarations. Typically the first procedure in a section is declared by a LET instruction, e.g.

```
LET firstprocedure( param) BE
$( .  .  .
$)
```

while the remainder are declared by an AND, e.g.

```
AND nextprocedure( param1, param2) =
   param1 + param2
```

With this use of LET and AND, any procedure in the section can call any other procedure in the section.

A procedure may either be a function, which returns a result, or a routine which does not.

A routine is declared with a BE as for 'firstprocedure' above. The routine may contain one or more RETURN instructions. A routine also returns when it reaches the end of the defined code.

A function is declared with an '=' sign as for 'nextprocedure' above. The reply is either a single expression, or is specified as the VALOF of a block containing one or more RESULTIS statements, e.g.

```
LET funct(x) = VALOF
$( IF x < 1000 RESULTIS x
   RESULTIS x/2 + 500
$)
```

Procedures can be used at any appropriate point by specifying the procedure identifier followed by the parameters in brackets, e.g.

```
firstprocedure( x)
y := nextprocedure( a, 25)
z := funct( b) /20
var := basicfunction()
var2 := funct( b+3) / basicfunction()
```

Parameters need not have the same name as in the procedure definition. The parameters passed are not altered by the called procedure which works with its own copy of the parameters. Procedure calls may include expressions to calculate each parameter. Even procedures with no parameters such as 'basicfunction' must be followed by a pair of brackets.

Blocks

Commands may be linked together as a block, enclosed by:

```
$( . . . $)
```

Such a block is equivalent to a single command to the code containing it, and so blocks may be used in defining a procedure, or to specify the operations required when a condition is satisfied. For example:

```
IF running THEN $( p := p+1; q := q-2 $)
```

Blocks may also be used to limit the scope, or area of relevance, of local variables.

If any local variables are declared in a block they must be placed at the beginning before any other commands, e.g.

```
$( LET a = 0
   LET b = param1*param2
   LET c = ?
   IF param3 = 0 THEN . . .
   . . .
$)
```

Section brackets $(and $) may be tagged by following the bracket with an identifier, e.g.

```
$(sort
   . . .
$)sort
```

A tagged opening section bracket must be closed with a closing section bracket with the same tag. A tagged closing bracket closes several nested blocks if this is required to provide a match with the equivalent tagged opening section bracket.

Commands within a block

Commands in a block may be placed on successive lines, or may be separated by semicolons (';'s).

The commands supported include:

Declarations

These define a new variable, and often define its initial value.

```
LET variable = value
```

The value may be a constant, an expression, a VEC, a string, a TABLE or the logical values TRUE or FALSE.

The value may be left undefined by:

```
LET variable = ?
```

Several variables can be defined in one declaration.

```
LET a, b, c = 0, 1, 2
```

Assignments

These set the value of a variable that has already been declared as a global, static or local variable.

```
y := x*x + 5*x - 3
```

When this statement is executed, the value of the right hand side is calculated and then the result replaces the existing contents of the variable y.

Several assignments may be combined in one statement.

```
x, y, z := 0, 2*a+1, -3
```

Conditionals

```
IF condition THEN command
UNLESS condition DO command
TEST condition THEN command1 ELSE command2
variable := condition -> valuetrue, valuefalse
```

The condition must give a TRUE or FALSE result. THEN and DO may be used interchangeably. ELSE and OR may be used interchangeably. The TEST conditional must be used if the command includes an ELSE clause.

Goto and labels

```
GOTO label
```

Processing continues from the command following the label, which must be in the same procedure as the GOTO statement.

Labels are declared by placing the label name, followed by a colon, at the appropriate point in the procedure. The following declaration defines labela:

```
labela:
```

Unless the label name has been declared as global, a label has the scope of the block in which it is declared.

Repetitive commands

A full range of repetitive commands is supported. Similar commands are available in many languages, and they enable the same code to be executed a number of times until a condition is satisfied.

The BCPL commands are:

```
FOR ident = exp1 TO exp2 DO command
FOR ident = exp1 TO exp2 BY constant DO command
WHILE condition DO command
UNTIL condition DO command
command REPEAT
command REPEATWHILE condition
command REPEATUNTIL condition
LOOP
BREAK
```

The FOR command declares a new ident for the duration of the loop. If the ‘BY constant’ is omitted from the FOR loop, BY 1 is assumed.

The command LOOP causes an immediate recycle of the repeated code starting with any conditional test or FOR loop calculation.

The command BREAK can be included in any repetitive command block, and causes an exit to just after the repeated commands.

Switchon and case

A SWITCHON command tests the current value of a specified variable, and then jumps to the command following the matching CASE label.

```
SWITCHON variable INTO
$( DEFAULT: commands
            ENDCASE

   CASE 'A': commands1()

   CASE 'B':
   CASE 'C': commands2()
             ENDCASE

   CASE -1: commands
            ENDCASE
$)
```

Each case value must be a constant.

The ENDCASE command causes a jump to the first command after the SWITCHON block. If ENDCASE or GOTO is not included, processing continues with the commands following the next CASE label. Thus one CASE label can provide additional processing to the next.

The DEFAULT case is optional. If it is omitted and no case is valid, processing continues following the SWITCHON block.

A SWITCHON construction may return a value.

```
x := VALOF SWITCHON char INTO
$( CASE 'A': RESULTIS 0
   CASE 'B': RESULTIS 10
   CASE 'C': RESULTIS 15
   DEFAULT: RESULTIS -1
$)
```

Operators

<u>Addressing</u>

@ gets the address of a variable e.g. @var gets the address of var. LV may be used instead of @.

! gets the contents of an address e.g. !x gets the contents of address x and x!y gets the contents of address x+y. RV may be used instead of !.

% addresses a byte e.g. address%offset addresses the byte offset from the address.

Arithmetic

*	multiply
/	divide (integer division i.e. 19/5 gives 3)
+	add
-	subtract
ABS	absolute value of next variable
REM	remainder after integer division by next variable (e.g. 19 REM 5 gives 4).

Logical

<table>
<tr><td>& or /\ or LOGAND</td><td>logical AND</td></tr>
<tr><td>| or \/ or LOGOR</td><td>logical OR</td></tr>
<tr><td>EQV</td><td>equivalent (bit by bit)</td></tr>
<tr><td>NEQV</td><td>not equivalent</td></tr>
<tr><td>NOT or ~ or \</td><td>logical NOT (one’s complement)</td></tr>
<tr><td><< or LSHIFT</td><td>shift left (e.g x<<3 shifts x 3 bits left) padding with zeroes</td></tr>
<tr><td>>> or RSHIFT</td><td>shift right padding with zeroes.</td></tr>
</table>

Conditional

= or EQ
~= or NE or \=
< or LT or LS
<= or LE
> or GT or GR
>= or GE

Precedence

The precedence of operators is as follows (highest precedence = most binding first):

```
()
Procedure call
! (dyadic) e.g. x!y
@ ! (monadic) e.g. !x
* / REM ABS
+ -
= NE < <= > >=
<< >>
NOT
&
|
EQV NEQV
-> , (conditional comma)
TABLE , (comma in a table)
VALOF
```

Operators of equal precedence are evaluated from left to right.

Brackets

These may be used in the conventional way to enclose an expression, e.g.

```
x := y*(z+3)
```

They are also used to enclose the parameters of a procedure, e.g.

```
proc( param1, param2)
```

This is essential even if the procedure has no parameters, e.g.

```
t := TIME()
```

Brackets are not used for indexing into arrays. Only single dimensioned arrays are directly supported by the language, and these are referenced using the '!' operator.

Comments

Comments on one line begin with // or || and are terminated by the end of the line.

Comments lasting several lines start with /* and end with */

Optional compilation

Optional compilation is useful when it is desired to maintain only one version of source code and to generate two or more variants of the object code.

Any part of a section may be enclosed between:

```
$<tag  .  .  .  $>tag
```

where tag is any identifier. If so the enclosed code is only compiled if tag is TRUE. All tags are FALSE at the start of a section, and may be complemented by a directive:

```
$$tag
```

so that the first occurrence of the directive makes tag TRUE and a second occurrence makes it FALSE. The $$tag directive is only executed if it is in code that is not being skipped.

Starting and stopping

A BCPL program is started by entering a procedure in the program named START.

It ends if this procedure returns, or if the statement FINISH is executed, or if one of the procedures that end a program is called. These include STOP and ABORT.

It is recommended that programs normally end by calling STOP with one parameter which is zero on successful completion. This subject is discussed further in chapter 6.

CINTCODE

The BCPL CINTCODE system compiles BCPL source into a Compact INTerpretive CODE known as CINTCODE. This has been designed to hold the program in a very concise form, so that it minimises the use of store and disk.

Running a program in this form requires an interpreter for the processor being used. This is a program which translates the CINTCODE instructions into the instructions supported by the processor. A suitable interpreter is included in the BCPL Language ROM.

Such interpreted execution of programs is very common on microcomputers. Although it leads to slower execution than fully compiled systems, the interpreted code can provide a number of advantages including compactness, and for many applications microcomputers are more limited in storage than in processing power.

Interpretation of an intermediate code such as CINTCODE has a number of differences from the interpretation of source text, which is usual in BASIC implementations. CINTCODE is more compact than BASIC, and is executed faster. On the other hand CINTCODE requires compilation and thus loses the direct execution of a new program offered by BASIC interpreters.

The advantages

The advantages of CINTCODE are not limited to compactness. Other advantages include:

<u>Portability</u>

In appropriate cases the same CINTCODE can be run on a variety of processors. See 'Portability' in chapter 8 for more details.

<u>Relocatability and linking</u>

CINTCODE is fully relocatable i.e. it can be loaded at any address in store and run without modification. CINTCODE is linked to the global vector when it is loaded. These two attributes make CINTCODE extremely convenient for developing large systems by combining a number of smaller modules.

<u>Debugging</u>

The CINTCODE interpreter includes features to support program development in a high level language environment. These include tracing the code at various levels and the setting of breakpoints. They are described in chapter 7.

<u>Decompilation</u>

Since CINTCODE is a compact representation of the original program, it is possible to decompile the code into a recognisable version of the original program. This is used in the powerful debugging facilities.

<u>Protection</u>

The interpreter can detect a number of errors during execution, such as calling a non-existent procedure, which cause very confusing errors in less protected systems.

The instructions of CINTCODE are described in outline in chapter 11. Their compactness can be illustrated by the CINTCODE form of the following statements:

```
list!i := number
number := RANDOM( number)
```

The CINTCODE form takes just seven bytes:

Byte	CINTCODE	Explanation
1	LP4	get number from fourth word in stack;
2	LP5	get i from fifth position in stack;
3	STP3	store number in list!i (list is in third position in the stack);
4	LP4	get number again;
5,6	K6G 111	call global 111 (RANDOM) moving six words down the stack;
7	SP4	save result in fourth word in stack i.e. in number.

The format of CINTCODE

When a source segment (i.e. file) is compiled it produces a corresponding CINTCODE file (also referred to as a segment). Each section in the source segment corresponds to a CINTCODE hunk in the CINTCODE file.

When the CINTCODE file is loaded into store each hunk is placed in a contiguous area of store. There are two possible arrangements - either each hunk is in a separate contiguous area or all the hunks are together in one contiguous area. The hunks may be linked to each other and the runtime library by the procedure GLOBIN which uses global linkage information included in each hunk to update the global vector.

Two or more CINTCODE files can be joined into a larger CINTCODE file. This join is performed by the utility JOINCIN. The larger CINTCODE file may then be loaded in one operation. JOINCIN is described in chapter 4.

The runtime system includes a large number of useful procedures in the BCPL Language ROM. Most of these procedures are provided in CINTCODE, and all are linked to loaded CINTCODE files through the global vector.

Other useful procedures are included in the CINTCODE library file, LIB. Programs using these procedures should contain NEEDS directives in the source specifying the names of the hunks required from LIB. The utility NEEDCIN (described in chapter 4) reads the program CINTCODE files, extracts the required hunks from LIB and creates a new CINTCODE file containing the hunks from both the program and LIB.

Compatible machine code

The BCPL CINTCODE system also supports machine code in a compatible way. This gives programs written in assembler some of the advantages of CINTCODE, in particular:

<u>Relocation</u>

The machine code is relocated at runtime. This makes it easy to hold a selection of machine code programs in store at the same time, and very much simplifies the use of the same machine code on different versions of the BBC Microcomputer.

<u>Overlaying</u>

Since the machine code can be linked through the global vector in the same way as CINTCODE, it is not difficult to overlay sections of machine code as required, and the same machine code section can be used in several different programs.

Two additional kinds of hunk are supported for machine code. These are handled in a similar way to CINTCODE hunks by the utilities such as JOINCIN and NEEDCIN, and by the procedures handling CINTCODE such as LOADSEG and GLOBIN.

The code hunk contains the machine code, but otherwise has a similar structure to a CINTCODE hunk.

The relocation hunk applies to the immediately preceding code hunk and specifies the relocation required. This relocation is applied by GLOBIN.

The assembler supports references to absolute locations, locations in the code, and locations in the global vector. However the linkage provided by the relocation hunk also allows for references to two common areas. This is intended for use by machine code generated from other languages.

Hunk formats

All hunks start with a word containing the hunk type, followed by a word containing the length of the hunk in words. The length excludes these first two words. The maximum length of a hunk is 4092 words.

The hunk types allocated are:

```
T.HUNK  (1000)  CINTCODE hunk
T.MC    (1001)  machine code hunk
T.RELOC (1002)  relocation hunk
(1003 to 1007)  reserved.
```

The last hunk in a CINTCODE File is followed by a special word:

```
T.END   ( 992)
```

which serves to check that the file has not been truncated.

CINTCODE and machine code hunks

These have a common structure to permit the hunk to be linked to the global vector:

- hunk type word;

- hunk length word (length in words excluding the type and length words);

- body of hunk;

- word containing 0 (marks the end of relocation data when working backwards from the end of the hunk);

- a pair of words for each global defined in the hunk;

- a word containing the highest global number referenced in the hunk.

The base address of the hunk is the point after the type word and the length word. The first word at this point is traditionally a repeat of the length of the hunk. This is not used by the CINTCODE system, which will accept a section, needs or procedure directive in this position. However the first word does not usually contain code because the base address cannot be linked to the global vector.

The pair of words for each global defined in the hunk are as follows:

- the global number of a global label in the hunk;

- the offset in bytes of this global label from the base address.

Global labels must start on an even byte since they are referenced as words in the global vector.

Relocation hunks

A relocation hunk contains information for relocating the immediately preceding hunk. The format starts with the hunk type and hunk length and the rest of the hunk is filled with words of relocation information.

The least significant thirteen bits of each word is the address to be relocated expressed as the byte offset from the base address of the preceding hunk. The most significant three bits specify the type of relocation.

0	Relative to the base address of the preceding hunk.
1	Relative to the base of the global vector.
2, 3	Relative to the words pointed to by the globals COMMON2 and COMMON3 (declared in SYSHDR). These globals must have been initialised before the segment is linked. This permits linking to common areas as supported by languages such as FORTRAN.

Before relocation the address in the code is the correct byte offset from zero.

The relocation specified above is byte-relative. Thus suppose a word of relocation information contains 1000 in the bottom thirteen bits and one in the top three bits and that the base of the global vector is at word 650 (byte 1300). The effect of relocation is to add 1300 to the contents of the word at byte offset 1000 from the base of the preceding hunk.

Relocation types 4-7 have similar effects to types 0-3 but with word-relative relocation. Thus if the relocation type in the example above were 5 the effect of relocation would be to add 650 to the contents of the word at byte offset 1000 from the base of the preceding hunk. These relocation types are not currently used by the assembler.

Section and procedure names (and NEEDS)

Both CINTCODE and machine code hunks may include section and procedure names and may include NEEDS directives to refer to other sections.

The section name identifies the hunk.

The NEEDS directive specifies that a given section is required by the section containing the NEEDS directive.

Procedure names can be included just before the start of the code of a procedure, for use in symbolic debugging.

In each case the name occupies five words. The first word specifies the type of reference:

#XFDDF for section
#XFEED for NEEDS
#XDFDF for procedure.

The next four words contain the name in a seven-byte string. The name is padded with spaces if necessary, while longer names are truncated.

The section name, if present, starts at the third word in the hunk (i.e. immediately after the length word). Any NEEDS directives immediately follow the section name. If there are NEEDS directives and no section name the first NEEDS directive starts at the third word of the hunk.

If a hunk contains no section directive and no NEEDS directives then, if procedure names are included, the name of the first procedure is used to identify the hunk.

ASSEMBLER

This section gives a brief outline of the 6502 assembler language supported by the relocatable assembler RAS and describes the assembler directives supported.

The language is very similar to other implementations of 6502 assembler, and some of the assembler directives are common to other implementations - though there is little standardisation on these features. However a number of the assembler directives are specific to this implementation, and are designed to ease communication with the rest of the BCPL system.

Thus this section provides full details of the assembler directives, but is not intended as an introduction to 6502 assembler programming. For such an introduction the reader is referred to books available on the subject.

The operation of the assembler is described in chapter 4. The use of assembler in the BBC Microcomputer is discussed in 'Machine code' in chapter 8.

Syntax

The syntax of an assembly language program is a number of lines, each of which is either blank, a comment line or an instruction line. A blank line has no data on it, while a comment line is any text preceded by an asterisk (*) or semicolon (;). The asterisk or semicolon must be the first character on the line.

An instruction line has the following form

Label: Opcode Operand ;Comment

where each item is separated by one or more spaces or tabs.

Each instruction line must contain either an opcode or a label. An opcode is either a 6502 instruction or a directive to the assembler. A list of the 6502 instructions and descriptions of the assembler directives are given below. Throughout the assembler all characters in labels and opcodes are compared irrespective of case, so that upper or lower case characters can be used interchangeably.

The label is normally optional, and if given must be a name starting with a letter, followed by up to 25 further letters or numbers. It may optionally be followed by a colon (:). Certain assembler directives require a label to be supplied. If the label is omitted then at least one space or tab is required before the opcode. A line may contain just a label.

The comment can be any text after the initial semicolon, and is always optional.

The operand varies according to the opcode used. Possible types of opcode are described more fully later.

Expressions

The 6502 instruction set provides a number of different addressing modes which may be used according to the opcode provided. A full list of valid opcode and address mode combinations is given later. The addressing modes are described below, but most of them require an expression as part of their structure.

The simplest form of an expression is a decimal number, and in this case the number is simply written as the expression. A hexadecimal number is composed of digits and the characters 'A' to 'F', preceded by a dollar sign ($).

A number may be specified in binary by preceding the number by a percent sign (%). A character value may be specified by supplying a character preceded, and optionally followed by, a single quote ('). In this case the ASCII value of the character is used. A number of special characters may be used:

```
'*'  represents '
'*"  represents "
'**  represents *
'*B  represents backspace (ASCII 8)
'*C  represents carriage return (ASCII 13)
'*E  represents escape (ASCII 27)
'*N  represents new line (ASCII 10)
'*P  represents form feed (ASCII 12)
'*S  represents a space (' ' is also valid)
'*T  represents a tab (ASCII 9).
```

An expression may also be the name of a label. The value used is the value of the program counter at the time the label was defined, or the value assigned to the label by the EQU or GLOBAL directive. Unlike the previous values for expressions, which are all absolute values, a label is relocatable if the program counter was relocatable when the label was defined.

The asterisk symbol (*) can be used to represent the value of the current program counter, which again is relocatable if in a relocatable section.

An expression can be preceded by an operator, which modifies the value of the expression as follows:

- return negative value of expression
+ return positive value of expression
~ return logical NOT of expression
< return low order byte of expression
> return high order byte of expression.

Any expression can be combined with any other expression using the binary operators described below. Note that the precedence of all operators is the same, so that a+b*c returns the value of (a+b)*c. Brackets can be used in order to force precedence, but care should be taken not to accidentally specify an indirect addressing mode (see below).

* return first expression multiplied by second
/ return first expression divided by second (integer division)
+ return first expression added to second
- return second expression subtracted from first
& return logical AND of expressions
| return logical OR of expressions
! return logical exclusive OR (XOR) of expressions
> return first expression shifted right the number of places specified by the second expression
< return first expression shifted left the number of places specified by the second expression.

In normal use, assembler routines called from BCPL will be assembled in relocatable mode. This is the default action, and can only be turned off by using the PHASE directive (see below). Relocatable values are labels defined in relocatable sections or defined via the GLOBAL directive (see below), while absolute values are numbers or labels defined in absolute sections.

There are a number of restrictions which must be observed when using relocatable symbols in mathematical expressions. These ensure that the final value of the expression can be relocated correctly. A relocatable value may have an absolute value added to or subtracted from it, in which case the result is relocatable. Two relocatable values which are not GLOBAL may be subtracted, resulting in an absolute value. No other binary operations are valid on relocatable values.

Given the restriction on relocatable values, expressions may be combined at will. The operations are performed on 16-bit values, not 8-bit values, so the expression ~$FF will be hexadecimal FF00, not 00, and so will not fit into a byte. Care should be taken when using expressions in places where a byte value is expected, for example DB. Values and operators within expressions may be separated by spaces.

<u>Examples</u>

```
$10 & ($AA | %0100)
1 + >XYZ
-XYZ*10
('A' < 8) | ('B')
```

Addressing modes

Not all addressing modes are valid for all instructions. A full list of allowable combinations is given later. In the description below, <expr> is a valid expression as described above.

Immediate - Assembler syntax: #<expr>
The value of <expr> must fit into a byte.

Absolute - Assembler syntax: <expr>
Zero Page - Assembler syntax: <expr>
The assembler automatically chooses between absolute and zero page instructions (if a choice exists) depending on the value of <expr>. If <expr> contains a reference to a label defined later in the program, or to a relocatable value, zero page mode will not be used. Note however that <expr> must not start with an opening bracket, otherwise the assembler will attempt to use indirect mode. If an expression which starts with a bracket is needed, it should be preceded by the unary operator '+'.

Indexed Indirect - Assembler syntax: (<expr>,X)
The value of <expr> must fit into a byte. The assembler looks for brackets specifying indexed mode before it attempts to parse the <expr>. Thus any brackets within <expr> must be in addition to the outer pair.

Indirect Indexed - Assembler syntax: (<expr>),Y
The value of <expr> must fit into a byte, and similar comments apply as above to the use of brackets within <expr>.

Absolute Indexed - Assembler syntax: <expr>,X
or <expr>,Y

Zero Page Indexed - Assembler syntax: <expr>,X
or <expr>,Y

If <expr> will fit into a byte, and zero page indexed mode is allowed for the specified opcode, then this is used. If <expr> is relocatable, contains a forward reference or will not fit into a byte, an attempt is made to use the absolute indexed mode if this is valid. Note however that <expr> must not start with an opening bracket, otherwise the assembler will attempt to parse the operand as an indirect expression. If brackets are required in <expr> then <expr> should be preceded by the unary operator '+'.

Extended Indirect - Assembler syntax: (<expr>)

If <expr> contains brackets they should be in addition to the ones representing the addressing mode.

Relative - Assembler syntax: <expr>

The value of <expr> must be such that the relative offset from the current value of the program counter can be represented by a signed byte. If not the error 'Relative branch too long' is given. The <expr> may not contain a label defined as GLOBAL.

Accumulator - Assembler syntax: A

The operand must be the letter A, and the opcode refers to the accumulator.

Implied

In this case no operand is required.

String - Assembler syntax: "<text>"
This mode is only used by directives. The <text> may be any set of characters except asterisk, newline and double quotes. These are represented by **, *N and *" respectively. The other special characters listed above may also be used in strings. There may not be more than 255 characters in a string.

Directives

Directives are commands to the assembler rather than valid 6502 instructions. They are all described in this section.

ASC <string>
This directive is used to initialise bytes to characters. The operand must be a string, as defined above. A byte is allocated for each character and initialised to the ASCII value of that character.
Example:
```
ASC "Table overflow"
```

DB <expr>
This is used to allocate bytes of memory, and to fill them with the values given by the operand. The values specified must each fit into a byte. A number of bytes may be defined by a single DB instruction by supplying a list of byte values separated by commas.
Examples:
```
COUNT DB 0
FLAGS DB $A0|$0F|INTFLG
TABLE DB 'A','B','C','D',0
```

DEPHASE

This statement resets relocatable mode. It should only be used in conjunction with the PHASE directive described below. The program counter is reset to the value it would have had if the preceding PHASE directive had not been encountered. This means that space is allocated for the code produced by PHASE inside the current relocatable region, and any subsequent code is placed after it.

DS <expr>

This directive reserves a number of memory bytes, which are zeroed. The number of bytes to be reserved is given by the value of the operand.

Example:

```
SAVEAREA DS 6
```

DW <expr>

This directive is used to allocate two consecutive bytes of memory. The first byte is filled with the low order byte of the value given by the operand. The second is filled with the high order byte.

Example:

```
JMPADR DW $EF20
```

END

This directive is supplied merely for compatibility with other assemblers. The directive is ignored.

EQU <expr>

This is used to set a label to a value given by the operand. No code is generated by this instruction line, but further references to the label so defined will result in the operand value being used instead. If the <expr> is relocatable, then so is the defined label. If the <expr> refers to a GLOBAL value, then the defined label is also GLOBAL.

Examples:
STATUS EQU $EC20
NCHARS EQU (PSIZE-1)*80

EVEN

This forces the program counter to be even, by inserting a NOP instruction if it is not. This is required in programs where a pointer is to be returned as a BCPL address. Since BCPL addresses are word pointers, the item being pointed at must be aligned at an even boundary. Procedures to be called from BCPL must also start on even boundaries.

GLOBAL <expr>

This directive specifies a label as representing a BCPL global cell. It must be followed by an operand which represents the global number involved. Subsequently the contents of the BCPL global cell may be referenced by using the label as an operand, e.g.

```
MYPROC  GLOBAL  301

MYPROC: code of the procedure
```

It should be noted that labels defined by GLOBAL are relocatable. Other labels defined by the EQU directive to an expression involving global labels are themselves defined as global. If the label defined by GLOBAL is subsequently redefined by using the same name as a label on an instruction line, then the corresponding global cell in the global vector will be initialised to the relocated word value of the label when the assembler program is loaded.

This enables routines written in assembler to be called from BCPL. Using a label defined as GLOBAL in this way also causes the program counter to be word aligned (an automatic EVEN directive is executed). Subsequent references to a redefined global label refer to the relevant cell of the BCPL global vector, and in particular such labels may not be used in relative addressing modes. A label should be defined as GLOBAL before it is redefined.

NEEDS <string>

The string given specifies another section required in conjunction with this one, and is identical to the NEEDS statement in BCPL. The NEEDS statement must come after the SECTION name, if given, but before any code. If the string is longer than seven characters it is truncated. It is padded with spaces if it is shorter than this. The program counter is word aligned.

Example:

```
NEEDS "EXIO"
```

PHASE <expr>

This directive causes the assembler to enter absolute mode, and is only valid if the assembler was in relocatable mode (the default). The program counter is reset to the value of <expr>. Any labels defined while in absolute mode will have absolute values, and the code produced will only run at the location indicated by the value of <expr>. The code is not placed at location <expr> when it is loaded, however, and this must be done by the user's program.

Relocatable mode is re-entered by the DEPHASE directive.

PROC <string>

This directive may be used to introduce a procedure written in assembler in such a way that it can be identified by the BCPL debugging routines. The operand following it must be a string, which is truncated or expanded to seven characters as in NEEDS. It also word aligns the program counter, so that the entry point to the procedure is on an even boundary.

SECTION <string>

This directive is similar to PROC, and the operand must again be a string, which is forced to seven characters. It identifies a section of assembler in the same way that the SECTION statement in BCPL identifies a section of BCPL. If used, the BCPL debugging routines will be able to identify the assembler section. The program counter is word aligned. If given, a SECTION directive must be the first instruction in a program.

STRING <string>

This is used to allocate a BCPL string. The operand must be a string, as defined above. The program counter is first word aligned, and then the length of the string is written out (as a byte value), followed by the characters specified in the string.

Examples:

```
MESS1 STRING "Error in expression"
MESS2 STRING "Type *"HELP*" for help*N"
```

Instructions

The following instruction mnemonics are provided. It is assumed that the reader is familiar with the 6502 instruction set. Note that BLT, BGE, BTR, BFL and XOR are synonyms for BCC, BCS, BNE, BEQ and EOR respectively. The table shows valid addressing modes for each instruction, where the addressing modes are indicated as follows:

Imm	Immediate mode - #<expr>
Abs	Absolute mode - <expr>
IndI	Indexed Indirect mode - (<expr>,X)
IInd	Indirect Indexed mode - (<expr>),Y
IndX	Absolute Indexed on X - <expr>,X
IndY	Absolute Indexed on Y - <expr>,Y
Ind	Extended Indirect - (<expr>)
Rel	Relative mode - <expr>
Acc	Accumulator mode - A
Imp	Implied mode

<u>6502 opcodes</u>

ADC	Imm	Abs	IndI	IInd	IndX	IndY
AND	Imm	Abs	IndI	IInd	IndX	IndY
ASL	Acc	Abs	IndX			
BCC	Rel					
BCS	Rel					
BEQ	Rel					
BFL	Rel					
BGE	Rel					
BIT	Abs					
BLT	Rel					
BMI	Rel					
BNE	Rel					
BPL	Rel					
BRK	Imp					
BTR	Rel					
BVC	Rel					
BVS	Rel					
CLC	Imp					
CLD	Imp					
CLI	Imp					
CLV	Imp					
CMP	Imm	Abs	IndI	IInd	IndX	IndY
CPX	Imm	Abs				
CPY	Imm	Abs				
DEC	Abs	IndX				
DEX	Imp					
DEY	Imp					
EOR	Imm	Abs	IndI	IInd	IndX	IndY
INC	Abs	IndX				
INX	Imp					
INY	Imp					

```
JMP      Abs Ind
JSR      Abs
LDA      Imm Abs IndI IInd IndX IndY
LDX      Imm Abs IndY
LDY      Imm Abs IndX
LSR      Acc Abs IndX
NOP      Imp
ORA      Imm Abs IndI IInd IndX IndY
PHA      Imp
PHP      Imp
PLA      Imp
PLP      Imp
ROL      Acc Abs IndX
ROR      Acc Abs IndX
RTI      Imp
RTS      Imp
SBC      Imm Abs IndI IInd IndX IndY
SEC      Imp
SED      Imp
SEI      Imp
STA      Abs IndI IInd IndX IndY
STX      Abs IndY
STY      Abs IndX
TAX      Imp
TAY      Imp
TSX      Imp
TXA      Imp
TXS      Imp
TYA      Imp
XOR      Imm Abs IndI IInd IndX IndY
```

11 Summaries

This chapter summarises the features of the BCPL CINTCODE system under the following headings:

Assembler

Built in commands and utilities

CINTCODE

DEBUG commands

ED and TED commands

Error numbers and trap codes

Global variables

Global vector

Manifest constants

Procedures

Tape files

TESTPRO commands

ASSEMBLER

The relocatable assembler accepts the following opcodes (synonyms are shown in brackets).

ADC	BRK	DEY	ORA	SEI
AND	BVC	EOR (XOR)	PHA	STA
ASL	BVS	INC	PHP	STX
BCC (BLT)	CLC	INX	PLA	STY
BCS (BGE)	CLD	INY	PLP	TAX
BEQ (BFL)	CLI	JMP	ROL	TAY
BGE	CLV	JSR	ROR	TSX
BIT	CMP	LDA	RTI	TXA
BLT	CPX	LDX	RTS	TXS
BMI	CPY	LDY	SBC	TYA
BNE (BTR)	DEC	LSR	SEC	
BPL	DEX	NOP	SED	

The addressing modes in the operand are:

Absolute	exp
Immediate	#exp
Absolute indexed by X	exp,X
Absolute indexed by Y	exp,Y
Indirect	(exp)
Indexed indirect	(exp,X)
Indirect indexed	(exp),Y
Accumulator	A

The assembler directives are:

ASC	Define ASCII string
DB	Define bytes
DEPHASE	End of absolute code
DS	Define space (reserve a number of bytes)
DW	Define word
END	End program
EQU	Equals
EVEN	Even byte
GLOBAL	Declare as global
NEEDS	Needed section
PHASE	Absolute code
PROC	Procedure name

<table>
<tr><td>SECTION</td><td>Section name</td></tr>
<tr><td>STRING</td><td>BCPL string</td></tr>
<tr><td>$</td><td>Hexadecimal</td></tr>
<tr><td>&</td><td>And</td></tr>
<tr><td>|</td><td>Or</td></tr>
<tr><td>!</td><td>Exclusive or</td></tr>
<tr><td>~</td><td>Not</td></tr>
<tr><td><</td><td>Shift left (binary operator)</td></tr>
<tr><td><</td><td>Low order byte (unary operator)</td></tr>
<tr><td>></td><td>Shift right (binary operator)</td></tr>
<tr><td>></td><td>High order byte (unary operator)</td></tr>
</table>

BUILT IN COMMANDS AND UTILITIES

```
BCPL    - the compiler
    FROM/A,TO/A,REPORT/K,NONAMES/S,MAX/S

CONT    - continue program
    No parameters

COPY    - copy file
    FROM/A,TO/A

DEBUG   - program test utility
    See page 387 for commands.

DELETE  - delete file
    FILE/A

ED      - the editor
    FROM/A,NEW/S
    See page 389 for commands.

END     - end command file
    No parameters

EX      - execute command file
    FILE/A
    Directives within the file are:
        .BRA character
            redefine '<' (start of keyword)
        .DEF keyword value
            set up a default for a keyword
        .DOT character
            redefine '.' (start of directive)
        .KET character
            redefine '>' (end of keyword)
        .KEY keys
            define argument keys

ERRCONT - continue command file if error
    OFF/S

GLOBALS - display global vector
    No parameters
```

```
HEAP    - display heap
    No parameters

INIT    - initialise program for testing
    Parameters are the parameters for the
    program being initialised.

IO      - display I/O streams
    No parameters

JOIN    - join files
    FROM/A,,,,,,,,,,,,AS=TO/A/K

JOINCIN - join CINTCODE files
    FROM/A,,,,,,,,,,,,AS=TO/A/K

LINK    - link file into global vector
    FILE,SYSTEM/S,LIBRARY/S

LOAD    - format CINTCODE file for linking
    FILE/A

MODE    - change display mode
    MODE/A

NEEDCIN - extract sections from library
    FROM/A,LIBRARY,AS=TO/A

PAUSE   - suspend command file
    No parameters but may be followed by message.

PROTECT - hold file in store
    FILE/A,OFF/S

RAS     - the relocatable assembler
    FROM/A,,,,,,,,,,TO=AS/A/K,LIST/K

READ    - read file into store
    FILE/A,AS=ON=TO

REM or
//      - comment
    Rest of line is ignored.
```

```
RENAME  - rename file
    FROM/A,TO/A

SAVE    - save store file
    FILE/A,AS=ON=TO

SHUFFLE - maximise contiguous free store
    No parameters

STACK   - display stacks
    ALL/S

STORE   - catalogue of store files
    No parameters

TED     - tiny editor
    FROM/A,NEW/S
    See page 389 for commands.

TESTPRO - test a procedure
    See page 411 for commands.

TIDY    - free up store
    No parameters

TYPE    - display text file
    FILE/A,AS=ON=TO

UNLINK  - unlink file from global vector
    FILE/A
```

CINTCODE

The following summary of CINTCODE instructions is intended to assist in the understanding of CINTCODE instruction traces and displays. These are described in chapter 7.

The list of instruction codes may be useful, for example if CINTCODE must be interpreted from a hexadecimal dump.

The symbols used are:

A, B, C	The three registers of the virtual machine. Normally when A is loaded its previous value is saved in B.
G	The base of the global vector.
P	The stack pointer of the virtual machine.
b	A byte.
g	A global variable number.
(i)	A qualifier on global instructions. Omitted if the global is below 256, 1 if the global is in the range 256 to 511 and 2 if it is between 512 and 767.
l	An address in CINTCODE, e.g. of a label, procedure or static variable.
m	A single digit number.
n	A small number.
w	A 16 bit word.

($) Addresses in CINTCODE can be referenced either directly or indirectly. The use of indirect addressing is shown by a '$' after the instruction code. Thus there are two Jump instructions, J and J$.

The CINTCODE instructions

A b	A := A + b
An	A := A + n
ADD	A := B + A
AG(i) g	A := A + G!g
AND	A := B & A
AP b	A := A + P!b
APn	A := A + P!n
APW w	A := A + P!w
ATB	B := A
ATC	C := A
AW w	A := A + w
BRK	Break to command state
CODE1	Enter machine code without swapping fault routine
CODE2	Enter machine code, swapping fault routine if appropriate
DIV	A := B / A
FHOP	A := FALSE; skip one byte
GBYT	A := B%A
GOTO	Jump to A
J($) l	Jump to l
JEQ($) l	Jump to l if B = A
JEQ0($) l	Jump to l if A = 0
JGE($) l	Jump to l if B >= A
JGE0($) l	Jump to l if A >= 0
JGR($) l	Jump to l if B > A
JGR0($) l	Jump to l if A > 0
JLE($) l	Jump to l if B <= A
JLE0($) l	Jump to l if A <= 0

```
JLS($) l  Jump to l if B < A
JLS0($) l Jump to l if A < 0
JNE($) l  Jump to l if B NE A
JNE0($) l Jump to l if A NE 0

Kn        Call A;  P := P + n
KnG(i) g  Call G!g;  P := P + n
K b       Call A;  P := P + b
KW w      Call A;  P := P + w

L b       A := b
Ln        A := n
LG(i) g   A := G!g
LmG(i) g  A := G!g!m
LL($) l   A := !l
LLG(i) g  A := G + g
LLL($) l  A := l
LLP b     A := P + b
LLPW w    A := P + w
LM b      A := -b
LMn       A := -n
LPn       A := P!n
LP b      A := P!b
LPW w     A := P!w
LmPn      A := P!n!m
LSH       A := B << A
LW w      A := w

MUL       A := B * A

NEG       A := -A
NOP       No operation
NOT       A := NOT A

OR        A := B | A

PBYT      B%A := C
```

```
REM        A := B REM A
RSH        A := B >> A
RTN        Return
RV         A := !A
RVn        A := A!n
RVPn       A := P!n!A

Sn         A := A - n
SG(i) g    G!g := A
S0G(i) g   G!g!0 := A
SL($) l    l := A
SP b       P!b := A
SPn        P!n := A
SPW w      P!w := A
ST         !A := B
STn        A!n := B
STPn       P!n!A := B
STmPn      P!n!m := A
SUB        A := B - A
SWB        SWITCHON A (binary chop switch)
SWL        SWITCHON A (range switch)

XCH        Exchange A and B
XOR        A := B NEQV A
```

The CINTCODE instruction set

	0 (00)	32 (20)	64 (40)	96 (60)
0(00)	--	K	LLP	L
1(01)	--	KW	LLPW	LW
2(02)	BRK	S0G	S0G1	S0G2
3(03)	K3	K3G	K3G1	K3G2
4(04)	K4	K4G	K4G1	K4G2
5(05)	K5	K5G	K5G1	K5G2
6(06)	K6	K6G	K6G1	K6G2
7(07)	K7	K7G	K7G1	K7G2
8(08)	K8	K8G	K8G1	K8G2
9(09)	K9	K9G	K9G1	K9G2
10(0A)	K10	K10G	K10G1	K10G2
11(0B)	K11	K11G	K11G1	K11G2
12(0C)	K12	K12G	K12G1	K12G2
13(0D)	CODE1	L0G	L0G1	L0G2
14(0E)	LM	L1G	L1G1	L1G2
15(0F)	LM1	L2G	L2G1	L2G2
16(10)	L0	LG	LG1	LG2
17(11)	L1	SG	SG1	SG2
18(12)	L2	LLG	LLG1	LLG2
19(13)	L3	AG	AG1	AG2
20(14)	L4	MUL	ADD	RV
21(15)	L5	DIV	SUB	RV1
22(16)	L6	REM	LSH	RV2
23(17)	L7	XOR	RSH	RV3
24(18)	L8	SL	AND	RV4
25(19)	L9	SL$	OR	RV5
26(1A)	L10	LL	LLL	RV6
27(1B)	FHOP	LL$	LLL$	RTN
28(1C)	JEQ	JNE	JLS	JGR
29(1D)	JEQ$	JNE$	JLS$	JGR$
30(1E)	JEQ0	JNE0	JLS0	JGR0
31(1F)	JEQ0$	JNE0$	JLS0$	JGR0$

	128 (80)	160 (A0)	192 (C0)	224 (E0)
0(00)	LP	SP	AP	A
1(01)	LPW	SPW	APW	AW
2(02)	--	--	--	--
3(03)	LP3	SP3	AP3	L0P3
4(04)	LP4	SP4	AP4	L0P4
5(05)	LP5	SP5	AP5	L0P5
6(06)	LP6	SP6	AP6	L0P6
7(07)	LP7	SP7	AP7	L0P7
8(08)	LP8	SP8	AP8	L0P8
9(09)	LP9	SP9	AP9	L0P9
10(0A)	LP10	SP10	AP10	L0P10
11(0B)	LP11	SP11	AP11	L0P11
12(0C)	LP12	SP12	AP12	L0P12
13(0D)	LP13	SP13	--	--
14(0E)	LP14	SP14	--	--
15(0F)	LP15	SP15	CODE2	--
16(10)	LP16	SP16	NOP	--
17(11)	--	S1	A1	NEG
18(12)	SWB	S2	A2	NOT
19(13)	SWL	S3	A3	L1P3
20(14)	ST	S4	A4	L1P4
21(15)	ST1	XCH	A5	L1P5
22(16)	ST2	GBYT	RVP3	L1P6
23(17)	ST3	PBYT	RVP4	L2P3
24(18)	STP3	ATC	RVP5	L2P4
25(19)	STP4	ATB	RVP6	L2P5
26(1A)	STP5	J	RVP7	L3P3
27(1B)	GOTO	J$	ST0P3	L3P4
28(1C)	JLE	JGE	ST0P4	L4P3
29(1D)	JLE$	JGE$	ST1P3	L4P4
30(1E)	JLE0	JGE0	ST1P4	--
31(1F)	JLE0$	JGE0$	--	--

DEBUG COMMANDS

Enter by: !DEBUG

The DEBUG Commands are:

Bn	Set breakpoint n
0Bn	Clear breakpoint n
0B	Clear all breakpoints
C	Execute one jump
nC	Execute n jumps
0C	Execute until **ESCAPE** pressed
D	Display current stack * Display collected statistics
DI	Display instructions
F	Run to breakpoint
nF	Run to nth breakpoint
Gn	Global n
I	Select instruction trace
J	Select jump trace * statistics on branches
K	Select procedure trace * statistics on procedures
mLn	Set limits for tracing etc.
M word,mask,limit	Memory search for word
MP ppp	Memory search for procedure ppp
N	Enter statistics collection
R	Set break on return from procedure
0R	Clear return break
S addr	Set contents of address. Address and contents are displayed followed by prompt ':'. Options are: **ESCAPE** terminate command **RETURN** leave contents unchanged and go on to next address value [value ...] **RETURN** update contents [and contents of next address ...] and go on to next non-updated address
S addr value [value ...]	Set contents of address to value [and subsequent addresses to subsequent values]
T	Trace
nT	Trace n lines

0T	Trace until **ESCAPE** pressed
Vn	Variable n
W	Unload debug overlay e.g. remove statistics collection
X	Exit (return to command state)
Z	* Clear collected statistics
?	Display current state
=	Display current variable
:	Display 8 words
:n	Display n words
$C	Set output to characters
$D	Set output to decimal
$O	Set output to octal
$X	Set output to hex
'a	Character a
#octal	Octal number
#Xhex	Hex number
+ - * / %(REM)	Arithmetic operators
& |	Logical operators
!	Indirection operator
< >	Shift (one bit) operators

* These comments apply when statistics collection is loaded.

ED AND TED COMMANDS

Enter by: !ED file or !ED file NEW
or !TED file or !TED file NEW

Commands available in both ED and TED

Cursor control keys	Move cursor
TAB	Insert spaces to next tab stop
DELETE	Delete character to left of cursor
RETURN	Split current line
f1	Delete character at cursor position
f2	Delete word
f3	Previous page
f4	Next page
f5	Start/end of page
f6	Start/end of line
f7	Previous word
f8	Next word
f9	Quit (abandon all changes)
CTRL f0	Verify (regenerate display)
CTRL f1	Delete to end of line
CTRL f2	Delete line
CTRL f3	Top of file
CTRL f4	Bottom of file
CTRL f5	Open new line
CTRL f6	Join line
CTRL f7	Undo changes to current line
CTRL f8	Exit

Commands available in ED only

f0	Enter command mode
CTRL-C	Centre line between margins
CTRL-E	Get command line
CTRL-F	Format two lines
CTRL-R	Repeat current command
CTRL-S	Show current state

Extended commands (ED only)

Commands are separated by ';' and terminated either by **RETURN** (to return to immediate mode) or **f0** (to stay in extended mode).

In the list of commands '/s/' denotes a string enclosed by delimiters and 'n' denotes a decimal number. Delimiters may be any non-alphanumeric character except space, ';', '(' and ')'.

The extended commands are:

A /s/	Insert line after current line
B	Bottom of file
BE	Mark block end
BS	Mark block start
C	Centre line between margins
CE	Cursor to end of line
CL	Cursor left
CR	Cursor right
CS	Cursor to start of line
DB	Delete block
DC	Delete character
DE	Delete to end of line
DL	Delete line
DW	Delete word
E /s/t/	Exchange t for s
EQ /s/t/	Exchange t for s with query
EX	Extend right margin
F /s/	Find s
FM	Format line
HB	Hide block
I /s/	Insert at cursor position
IB	Insert copy of block
IF /s/	Insert file
IL /s/	Insert line before current line
J	Join lines
LM n	Set left margin
MB	Move block
N	Next line
P	Previous line
Q	Quiet (update display only at end of command)
RM n	Set right margin

```
RP command  Repeat command until error or any key
            pressed
S           Split line at cursor position
SB          Show block
SE          Show end of block
SV /s/      Save all text to file
T           Top of file
TS n        Set tab spacing
WB /s/      Write block to file
n command   Repeat command n times
RP (command; command ...) or
n  (command; command ...)
            Repeat group of commands
```

ERROR NUMBERS AND TRAP CODES

Error numbers

A single system of error numbers is used for error messages displayed on the console, for error returns from RUNPROG, for errors reported in RESULT2 and for ABORT codes. In general error numbers above 100 represent ABORT conditions.

Negative values represent warnings (or qualified success), 0 represents success and +ve values represent errors.

The error numbers used by the system and their meanings are:

- -4 TIDY used to abandon a program run from a command file. Unless ERRCONT is in effect the command file is also abandoned.
- -3 attempt to delete non-existent store file.
- -2 NEEDCIN - one or more unsatisfied NEEDS directives.
- -1 no START or TRAPSTART linked. One common cause is trying to INIT a program which has run to completion and is therefore not linked. Another is trying to run a CINTCODE file which is a program overlay and therefore has no START.
- 11 invalid arguments to built in command or utility or program. Also used by the MODE library procedure to indicate an invalid parameter.
- 12 CONT - no interrupted program. A common cause is trying to resume a program that has terminated or aborted.
- 13 RUNPROG - no START or TRAPSTART in specified program or program is attempting to RUNPROG itself.
- 14 RUNPROG - invalid parameter e.g. RUNPROG("") would give this error.
- 15 cannot free necessary heap space to change mode. Used by the MODE built in command and by the MODE library procedure.

17 store file not found. This error code is normally returned only when trying to access a store file specifically e.g. TYPE /S.FRED would give this error if FRED did not exist, whereas TYPE FRED would give error 27 since it would look in the current filing system after looking unsuccessfully in store.

18 illegal device specified to RENAME command or procedure (e.g. /L).

19 RENAME of store file - 'FROM' file does not exist or 'TO' file already exists.

20 cannot open, link or delete store file - file is already open or linked. One common cause of this error is specifying the same file for input and output (e.g. COPY FRED FRED). Once the file has been opened for input the attempt to open it for output fails.

23 delete function not allowed on specified device (e.g. DELETE /L would give this error).

27 file does not exist in current filing system (see the comments for error 17).

30 the command file being executed contains the command INIT or CONT.

40 general error return from utilities (e.g. compilation failed because of errors in the source).

45 RAS - fatal error. This error indicates a program failure and should not normally occur.

50 call of READ, READVEC or FILETOVEC with 'from' device neither store nor the current filing system (e.g. the command READ /P XXX would cause this error).

51 not enough store to get vector or vector too small. This error is generated by GETVEC if it fails to get a vector and also by the procedures FILETOVEC, READ, READVEC, SAVE, SAVEVEC and VECTOFILE if they fail because there is not enough store or the supplied vector is not big enough.

52 device not recognised. A device/file name begins with '/' but the next character is not one of the devices recognised by the system. This error is also generated if the '.' separating '/S' or '/F' from the filename is omitted.

54 unable to load CINTCODE - end of file reached prematurely. The most likely cause is loading a CINTCODE file which has been accidentally truncated.

55 unable to load CINTCODE - invalid format. The most likely cause is trying to LOAD, LINK or execute a file which is not a CINTCODE file.

56 unable to link CINTCODE. The file being linked uses or defines a global variable beyond the end of the global vector. This error might arise if the global vector has been cut down for some reason or if global 0 has been corrupted (global 0 holds the maximum global number - normally 767).

57 unable to link CINTCODE. The link data in the file is corrupt. It is likely that the file has been corrupted at some time.

59 cannot link file as file is not loaded. This error could be caused by a program calling GLOBIN without first calling LOADSEG.

60 attempt to use input device for output or vice versa (e.g. TYPE /L).

61 GLOBUNIN was called to unlink a file which was not linked.

62 cannot unlink file - bad link data. This error probably means that the file has been overwritten in store. To recover from this error the system may have to be restarted.

99 ED - fatal error. This error indicates a program failure and should not normally occur.

101 cannot write to store file - no room.

103 EXTSFILE or VECTOFILE called with a non-heap vector. Only vectors obtained from the heap by GETVEC may be used in these calls.

104 EXTSFILE, SAVE, SAVEVEC or VECTOFILE called with vector containing invalid byte count in word 1. The count is either negative or, in the case of EXTSFILE and VECTOFILE, too big for the size of the heap vector.

105 attempt to select a non-existent stream or to read from an output stream or write to an input stream. The most likely cause is closing a stream (by ENDREAD or ENDWRITE) then attempting to select it or read from it or write to it.

106 attempt to close a non-existent stream. This should not normally occur.

110 CALLCO to an active coroutine.

111 RESUMECO to an active coroutine.

112 DELETECO of an active coroutine.

113 the system cannot re-allocate the reserved heap area used by SAVE. This indicates a system failure and should not occur.

114 the first parameter to the procedure VDU is invalid. This parameter should be a string composed of numbers or percent signs separated by commas or semicolons.

121 FREEVEC - attempt to free non-existent vector. This error is generated if the parameter to FREEVEC is an address outside the heap.

122 heap corrupt. If this error occurs the system must be restarted. A possible cause is a program writing beyond the end of a heap vector.

123 APTOVEC - no room in stack.

124 invalid parameter to GETVEC. The parameter is either negative or greater than 32763.

125 SHUFFLE found heap vector marked as movable but not part of a file. A possible cause is a program writing beyond the end of a heap vector or writing to vector!(-1). The system will probably have to be restarted.

130 store file linkage corrupt. A possible cause is a program writing beyond the end of a heap vector. The system must be restarted.

131 linked files linkage corrupt. A possible cause is a program writing beyond the end of a heap vector. The system must be re-started.

Error numbers in the range 1000-1255 correspond to the operating system error numbers 0-255. See the BBC Microcomputer User Guide for full details of these errors. One of the most common, however, is:

1017 ESCAPE pressed.

Note that if an operating system error is generated while a file is being accessed then the system will in due course attempt to close the file. In some cases the filing system may have already closed the file and so the close causes another error (often a 'channel' error) which may mask the original error.

Trap codes

If the interpreter detects an error while running a run-state program it traps to the command state and a message of the form 'Trap -2 x' is generated where x is a letter. If the interpreter detects an error while running a trap-state program it displays a message of the form '*** ERROR: x' where x is a letter and the system must then be restarted.

The meanings of x are:

D The fault handling routine was entered while the interpreter was executing CINTCODE. (The routine is expected to be entered only if executing machine code called from CINTODE.)

G Call to uninitialised global. A common cause is running a CINTCODE file that includes unsatisfied NEEDS directives, or omitting a NEEDS directive (e.g. for a procedure in the library LIB).

S Stack overflow. This may be caused by accidental recursion or it may simply be that the default stack size of 400 words is too small for the needs of the program, in which case STARTINIT should be used to obtain a bigger stack.

X Invalid CINTCODE instruction. Possible reasons are that the CINTCODE being executed has been corrupted, that a return address on the stack has been corrupted or that a procedure call was to an invalid address.

Z Attempt to divide by zero.

If the '*** ERROR: x' message is generated with any other letter there are two possible causes. One is that the page 0 locations used by the system (particularly those below byte 50 hex) have been corrupted. The other is that a trap-state program has called the procedure TRAP.

GLOBAL VARIABLES

This section lists (alphabetically) the global variables which may be useful in application programs. The global number and 'L' for LIBHDR or 'S' for SYSHDR are given in brackets after the global name.

ABORTCODE (4 S)
ABORTLABEL (5 S)
ABORTLEVEL (6 S)

See the description of the procedure ABORT in chapter 5.

CNSLINSTR (52 L)

Initialised by the system to a console input stream. It may be selected by a program in error conditions but should not be used in normal circumstances. This stream is used by DEBUG for its input and so if used by a program being tested with DEBUG strange interactions may occur.

CNSLOUTSTR (53 L)

Initialised by the system to a console output stream. It may be selected by a program whenever output to the screen is required.

CURRCO (22 S)

The current coroutine. It may be useful to return this identifier when COWAIT returns control to a parent coroutine.

ERRORSTREAM (58 S)

The stream that is selected for input following ENDREAD and for output following ENDWRITE. Also selected by SELECTINPUT(0) and SELECTOUTPUT(0). Any attempt to read from or write to this stream causes an ABORT.

LASTERROR (21 S)

Set to the error code returned by the last program run from a command file. May be tested by a program run from a command file to see whether the previous program completed successfully or not.

LIBBASE (117 S)

The address of the start of the runtime library procedures.

MAXGLOB (0 S)

Contains the maximum global number that may be used. Since MAXGLOB is global 0 the expression '@MAXGLOB' may be used as the address of the global vector.

MCRESULT (11 L)

Holds the A register and flags following a call to machine code using CALL or CALLBYTE. If a fault has occurred while in machine code then it holds #XFFnn where nn is the fault code. MCRESULT is used by the system for various purposes and may be changed by any library procedure. Also used to return information from the procedure FSTYPE.

RESULT2 (15 L)

Gives a second result in addition to the normal return parameter from a function. For example RESULT2 holds the remainder returned by MULDIV, additional information from TESTFLAGS and TESTSTR and error return codes from a number of other procedures.

SYSINDEX (18 S)

Pointer to an area of system information. See the description of STARTINIT in chapter 5 for examples of use.

GLOBAL VECTOR

The globals declared in the header files LIBHDR and SYSHDR are listed here alphabetically within each file. The following information is given with each global:

- the global number;

- the type - either a named procedure (N), an unnamed procedure (U), a user-supplied procedure (S) or data (D). Procedures in LIB are marked by '/L';

- if the global has the same meaning in the RCP BCPL CINTCODE system for the CP/M operating system.

LIBHDR

NAME	NUMBER	TYPE	CP/M-COMPATIBLE
ABORT	23	N	Y
ADVAL	201	N/L	-
CALL	31	N	-
CALLBYTE	121	N	-
CALLCO	43	N/L	Y
CAPCH	59	N	Y
CNSLINSTR	52	D	Y
CNSLOUTSTR	53	D	Y
COMPCH	42	N	Y
COMPSTRING	95	N	Y
COWAIT	44	U	Y
CREATECO	45	U	Y
DELETECO	46	U	Y
DELFILE	64	N	Y
ENDREAD	71	N	Y
ENDWRITE	72	N	Y
ENVELOPE	129	N/L	-
FILETOVEC	125	N	-
FINDARG	96	N	Y
FINDINPUT	66	N	Y
FINDOUTPUT	67	N	Y
FREEVEC	25	N	Y

NAME	NUMBER	TYPE	CP/M-COMPATIBLE
GETVEC	26	N	Y
GLOBIN	100	U	Y
GLOBUNIN	101	U	Y
INPUT	73	N	Y
LEVEL	27	N	Y
LOADSEG	104	N	Y
LONGJUMP	28	N	Y
MAXVEC	29	N	Y
MCRESULT	11	D	-
MODE	115	U	-
MOVE	36	N	Y
MULDIV	37	N	Y
NEWLINE	84	N	Y
NEWPAGE	85	N	Y
OPSYS	35	N	-
OUTPUT	74	N	Y
RANDOM	111	N	Y
RDARGS	97	N	Y
RDBIN	75	N	Y
RDCH	76	N	Y
RDITEM	98	N	Y
READ	169	N	-
READN	78	N	Y
READVEC	168	N	-
READWORDS	106	N	Y
RENAME	70	N	Y
RESULT2	15	D	Y
RESUMECO	47	N/L	Y
RUNPROG	118	U	-
SAVE	123	N	-
SAVEVEC	204	N	-
SELECTINPUT	79	N	Y
SELECTOUTPUT	80	N	Y
SOUND	128	N/L	-
SPLIT	99	N	Y
START	1	S	Y
STARTINIT	3	S	Y
STOP	2	U	Y
TESTFLAGS	108	N	Y
TESTSTR	77	N/L	Y
TIME	112	N/L	-
TRAP	38	N	Y

NAME	NUMBER	TYPE	CP/M-COMPATIBLE
UNRDCH	83	N	Y
VDU	200	N/L	-
VECTOFILE	126	N	-
WRBIN	81	U	Y
WRCH	82	U	Y
WRITED	88	N	Y
WRITEF	90	N	Y
WRITEHEX	91	N	Y
WRITEN	92	N	Y
WRITES	94	U	Y
WRITEWORDS	109	N	Y

SYSHDR

NAME	NUMBER	TYPE	CP/M-COMPATIBLE
ABORTCODE	4	D	Y
ABORTLABEL	5	D	Y
ABORTLEVEL	6	D	Y
APTOVEC	127	N/L	-
BACKMOVE	219	N/L	-
BACKMVBY	218	N/L	-
CLIINSTR	33	D	-
COLIST	166	D	-
CONTPRG	149	U	-
COMMON2	62	D/S	-
COMMON3	63	D/S	-
CURRCO	22	D	-
DELXFILE	65	N	-
ENDPROG	24	N/L	Y
ENDTRAP	34	U	Y
ERRORMSG	202	U/S	-
ERRORSTREAM	58	D	Y
EXTSFILE	102	N	-
FAULTROUTINE	205	S	-
FINDXINPUT	68	N	-
FINDXOUTPUT	69	N	-
FSTYPE	39	U/S	-
HEAP	48	D	-
HEAPEND	54	D	-
LASTERROR	21	D	-

NAME	NUMBER	TYPE	CP/M-COMPATIBLE
LIBBASE	117	D	-
LINKEDFILES	114	D	-
MAINSTACK	20	D	-
MAXGLOB	0	D	Y
MOVEBYTE	213	N	-
SHUFFLE	203	U	-
STACKSIZE	30	N/L	Y
STOREFILES	16	D	-
STREAMCHAIN	17	D	-
SYSINDEX	18	D	Y
TIDYSTATE	208	D	-
TRAPSTACK	19	D	Y
TRAPSTART	122	S	Y
UNLOADSEG	105	U	Y
VDUINFO	116	N/L	-
WRITEA	86	U	Y
WRITEBA	87	U	Y
WRITEDB	89	N/L	Y
WRITEOCT	93	N/L	Y
WRITET	40	N/L	Y
WRITEU	41	N/L	Y

The first free global for general use is number 250.

MANIFEST CONSTANTS

This section lists (alphabetically) the manifest constants which may be useful in application programs. The value of the constant and 'L' for LIBHDR or 'S' for SYSHDR are given in brackets after the name.

BITSPERWORD (16 S)
    The number of bits in a BCPL word.

BYTESPERWORD (2 S)
    The number of bytes per word in strings.

CONSOLE.KEY (1 L)
    A mask bit used with TESTFLAGS to find if there has been a new key depression on the console.

DV.F (8 S)
DV.S (7 S)
    Device codes for use in calling DELXFILE, FINDXINPUT and FINDXOUTPUT.

ENDSTREAMCH (-1 L)
    The result returned by RDCH/RDBIN when a stream is exhausted.

FIRSTFREEGLOBAL (250 L)
    The number of the first global available to the user.

GLOBWORD (-888 L)
    When the system starts up all globals are initialised to this value. When a global procedure is unlinked the global is set to this value. The value does not correspond to a valid address and is negative, so for most practical purposes a global containing this value can be assumed to be in its initial state.

Because globals used to hold data are not re-initialised when a program is unlinked, it is dangerous to assume that because a global does not contain this value it is initialised to a global procedure address.

I.DEFSPACE (12 S)
I.RESTART (13 S)
See 'User-defined characters' in chapter 8.

I.RSTATE (7 S)
See description of STARTINIT in chapter 5.

MAXINT (32767 S)
The most positive number which may be held in a BCPL word.

MININT (-32767 S)
The most negative number which may be held in a BCPL word.

MORE.INPUT (2 L)
A mask bit used with TESTFLAGS to find if more input can be read without physical input.

M.TRAPESC (2 S)
M.TRAPGV (4 S)
See description of STARTINIT in chapter 5.

R.MCST (0 S)
See description of STARTINIT in chapter 5.

STYPE.TERM (4 L)
STYPE.INT (8 L)
These are for use with TESTSTR to determine the characteristics of the current streams. See the description of TESTSTR in chapter 5.

TICKSPERSEC (100 L)
The number of clock ticks in a second. Used to convert values obtained by TIME into seconds.

PROCEDURES

In this summary, the procedures are shown in **bold**, and the parameters and the returned value are given appropriate names. The full definition of each procedure is given in chapter 5.

Coroutines

reply := **CALLCO**(coroutine, parameter)
parameter := **COWAIT**(reply)
coroutine := **CREATECO**(procedure, stacksize)
is.successful := **DELETECO**(coroutine)
reply := **RESUMECO**(coroutine, parameter)

File and stream handling

is.successful := **DELFILE**(filename)
is.successful := **DELXFILE**(filename, device)
ENDREAD()
ENDWRITE()
is.successful := **EXTSFILE**(filename, vector)
vector := **FILETOVEC**(filename)
stream := **FINDINPUT**(string)
stream := **FINDOUTPUT**(string)
stream := **FINDXINPUT**(string, device)
stream := **FINDXOUTPUT**(string, device)
stream := **INPUT**()
stream := **OUTPUT**()
is.successful := **READ**(fromfile, tofile, contig)
is.successful := **READVEC**(fromfile, tovec, size)
is.successful := **RENAME**(fromfilename, tofilename)
is.successful := **SAVE**(fromfile, tofile)
is.successful := **SAVEVEC**(fromvec, tofile)
SELECTINPUT(stream)
SELECTOUTPUT(stream)
flag.was.set := **TESTFLAGS**(flagmask)
is.streamtype := **TESTSTR**(streamtype)
is.successful := **VECTOFILE**(fromvec, tofile)

Input and text processing

capital.character := **CAPCH**(character)
comparison := **COMPCH**(character1, character2)
    comparison is: negative if character1 is before character2, zero if both are the same, positive if character2 is before character1
comparison := **COMPSTRING**(string1, string2)
    comparison is: negative if string1 is before string2, zero if both are the same, positive if string2 is before string1
argumentno. := **FINDARG**(keys, string)
end.of.args := **RDARGS**(keys, toargvector, size)
bincharacter := **RDBIN**()
character := **RDCH**()
itemtype := **RDITEM**(tovector, maximumsize)
integer := **READN**()
wordsread := **READWORDS**(destination, words)
endpointer := **SPLIT**(prefix, character, string, startpointer)
UNRDCH()

Loading and unloading code

is.successful := **GLOBIN**(segment)
next.segment := **GLOBUNIN**(segment)
segment := **LOADSEG**(string)
UNLOADSEG(segment)

Output formatting

NEWLINE()
NEWPAGE()
WRBIN(binarycharacter)
WRCH(character)
WRITEA(wordaddress)
WRITEBA(byteaddress)
WRITED(integer, fieldwidth)
WRITEDB(doubleinteger)
WRITEF(format, a, b, c, d, e, f, g, h, i, j, k)
WRITEHEX(integer, fieldwidth)
WRITEN(integer)
WRITEOCT(integer, fieldwidth)
WRITES(string)

```
WRITET( string, fieldwidth)
WRITEU( unsignedinteger, fieldwidth)
WRITEWORDS( source, words)
```

Runtime control

```
ABORT( aborttype)
res := APTOVEC( procedure, size)
ENDPROG( endcheck)
FREEVEC( vector)
vector           := GETVEC( vectorsize)
stackpointer     := LEVEL()
LONGJUMP( stackpointer, label)
max.free.vector  := MAXVEC()
stack.size.left  := STACKSIZE()
START( parameter)
stacksize.requ.  := STARTINIT()
STOP( returncode)
TRAP( traptype, letter)
```

Utility procedures

```
res := ADVAL( parameter)
BACKMOVE( from, to, words)
BACKMVBY( frombyte, tobyte, bytes)
XY :=     CALL( wordaddress, A, XY)
    mcresult := flags and A
XY := CALLBYTE( byteaddress, A, XY)
    mcresult := flags and A
ENVELOPE( envelope.table)
ERRORMSG( errorcode)
file.system.characteristic := FSTYPE( mask)
byte := GETBYTE( string, byteposition)
  (GETBYTE is in OPT)
is.successful := MODE( display.mode)
MOVE( from, to, words)
MOVEBYTE( frombyte, tobyte, bytes)
```

```
result   := MULDIV( a, b, c)
    result := a*b/c
    result2 := remainder
XY := OPSYS(A, XY)
    mcresult := flags and A
string.words.used := PACKSTRING( fromunpacked,
                                           tostring)
  (PACKSTRING is in OPT)
PUTBYTE( string, byteposition, byte)
  (PUTBYTE is in OPT)
randominteger := RANDOM( randominteger)
res := RUNPROG( writefstring, param1, param2,
                                  param3, param4)
SHUFFLE( delete.unprotected.files)
SOUND( sound.table)
time := TIME()
UNPACKSTRING( fromstring, tounpacked)
  (UNPACKSTRING is in OPT)
VDU( string, a, b, c, d, e, f, g, h, i, j)
size := VDUINFO( rows.or.cols)
```

TAPE FILES

The order of the files on a BCPL system tape should be as follows:

```
EX        // small utilities
JOIN

RAS       // assembler next to editor
TED
ED

BCPL      // compiler next to editor
BCPLARG
BCPLSYN
LIBHDR    // main header file
BCPLTRN
BCPLCCG

SYSHDR    // rarely used header file

JOINCIN   // program development aids
NEEDCIN
LIB       // CINTCODE library

DEBUG     // debug files
INSTR
TRACE
STATS

GLOBALS   // debug utilities
HEAP
IO
STACK
TESTPRO

OPT
Examples
```

TESTPRO COMMANDS

Enter by: !TESTPRO

Commands are:

D	Display procedure being tested and current values of the parameters.
H	Display list of commands.
P par par ...	Set parameters (from 1 to 6 values may be specified).
nP par par ...	Set parameters from parameter n (1-6) onwards. From 1 to 7-n values may be specified.
R	Run test.
T proc	Select procedure proc to be tested.
X	Exit.

Parameters are decimal numbers (e.g. 1234), hex numbers (e.g. #X89AB), values of global variables (e.g. G300) or strings ("ABCDE").

Procedure is name (e.g. GETVEC), decimal number, hex number or global variable.

Appendix

This appendix contains details of many of the internal data areas and procedures used by the BCPL system. It also contains details of the operating system interfaces used, hints on writing command-state programs and and the effects of running in the second 6502 processor using the Tube.

The information in this appendix is expected to be of use either simply for interest or to help with debugging awkward problems. Note that whereas it is intended to retain features described elsewhere in this guide in future releases of the system, the same does not apply to the structure of internal data areas etc.

This appendix is divided into the following sections:

Running in the second 6502 processor

System data areas

System procedures

Use of operating system routines

Writing command-state programs

RUNNING IN THE SECOND 6502 PROCESSOR

The BCPL system runs with no alterations in the second 6502 Processor (using the Tube). The system is automatically copied over to the second processor by the operating system. It adapts itself by taking advantage of the extra RAM to increase the size of the heap. The heap starts at a lower address than normal and extends up to the operating system ROM. The BCPL Language ROM thus becomes a permanently allocated area within the heap.

The only other feature of the system that appears different is that the heap is not adjusted by the MODE command/procedure and so error 15 (not enough heap space to change display mode) can never occur.

SYSTEM DATA AREAS

This section details the format of various data areas used by the BCPL system. The names of global data areas are given even when no header file declaring the name is provided. The names of globals are followed by the global number (and 'S' for SYSHDR if appropriate) in brackets.

The data areas are described under the following headings:

Heap

Language RAM

Miscellaneous

Page 0

Stacks

Store files

Stream control blocks

System index

System saved data

Heap

The heap is made up of a number of contiguous areas, each an even number of words long. The first two words of each area are for system use. The rest of the area is data. When a vector is allocated by GETVEC the value returned is the address of word two of the area allocated.

The format of the first two words of each area is:

```
word 0: address of the next area
word 1: bits 0-7: area type as:
                    0 - free (in this case bits 8-
                        15 must all be 0 too)
                    1 - root stack
                    2 - program stack
                    3 - global vector
                    4 - stream control block
                    5 to 63
                        area allocated for system
                        use
                  127 - program data area
                  128 - file control block
                  129 - file data block
                  130 - file name block
        bit 8:    set if area can be moved by
                  SHUFFLE (must only be set on
                  areas of type 128-130 for un-
                  linked files)
        bits 9-14:reserved
        bit 15:   set if area is protected (for
                  file areas it need only be set
                  for the FCB and prevents the
                  file being deleted by SHUFFLE,
                  for other areas it prevents the
                  area being freed by TIDY).
```

The global HEAP (48 S) points to the first area in the heap. The global HEAPEND (54 S) points to the last area which has a special format:

```
word 0: 0
word 1: -1
```

Language RAM

This section describes the layout of the language RAM which is the area from byte 400 hex to byte 7FF hex.

Index and miscellaneous data

The first section is an eleven-word data area. This starts at byte 400 hex. All other areas in the language RAM can be accessed through this data area:

- word 0: word address of the system index (see page 425)
- word 1: byte address of the despatch routine (see below)
- word 2: word address of the start of the heap
- word 3: word address of heap vector reserved for use by the SAVE procedure
- word 4: length of heap vector reserved for use by the SAVE procedure
- word 5: word address of the root stack
- word 6: word in which GETVEC saves the size of vector required when trapping to the command state
- word 7: byte address of current fault routine
- word 8: byte address of machine code to set up fault routine address (from word 7) and jump to despatch routine
- word 9: word address of code to implement MOVE procedure
- word 10: word address of code to implement OPSYS procedure.

Despatch routine

The data area is followed by the despatch routine, which is the heart of the interpreter. It is called to access and process the next CINTCODE instruction. It is held in RAM so that it may be modified if required (e.g. to check for interrupts having occurred while the last CINTCODE instruction was being processed). The code is:

```
LDNEXT:          ; process next instruction
        LDY     #0
        LDA     ($06),Y ; next instruction
ZZTRAP: TAY
        INC     $06     ; update PC
        BEQ     LDNEX4
LDNEX2: LDX     $BE00,Y ; index to jump table
        STX     $52     ; save in work area
        LDX     $BF00,Y
        STX     $53
        JMP     ($52)   ; process instruction
LDNEX4: INC     $07
        BNE     LDNEX2
```

When a program is running under DEBUG and trapping on every instruction then the byte at ZZTRAP is changed to NOP. Thus on entry to the code to process a CINTCODE instruction A=Y=the instruction unless ZZTRAP has been changed when A=the instruction and Y=0.

Other data areas

The remainder of the language RAM contains the system index, the heap and the root stack. These areas are all detailed in other subsections.

Miscellaneous

This section describes miscellaneous global data items which are not covered elsewhere. They are listed in alphabetical order.

CLIINSTR (33 S) the stream identifier for the current command file (0 if there is no current command file).

EXERROR (207) TRUE if ERRCONT is not in effect.

PRGENDLABEL (8) } PRGENDLEVEL (7) }	the current recovery point. LONGJUMP is used to transfer control here in a variety of circumstances.
RUNSUSP (206)	TRUE if there is a suspended run-state program.
TIDYSTATE (208 S)	Set to 2 when a TIDY is performed. Set to 1 when a modified TIDY (same as TIDY except that no files are unlinked) is performed unless it is already non-0. Intended for use by debugging utilities, which may reset it to other values as required.

Page 0

This section describes the use the BCPL system makes of page 0. Many of the areas described should be accessed via the system index (see page 425) rather than directly. Areas marked with '*' are described under 'System index' on page 425.

```
Words  0- 6: current state *
Word   7   : word address of the global vector
Word   8   : byte address of RESULT2
Word   9   : byte address of MCRESULT
Word  10   : byte address of FAULTROUTINE
Words 11-16: flags *
Words 17-18: limits *
Words 19-25: last saved trap state *
Words 26-28: trap count *
Words 29-30: count*
Word  31   : flag indicating length of last
             jump/call/return instruction (used by
             DEBUG):
             -ve 1 byte
             0   2 bytes
             +ve 3 bytes
Words 32-38: last saved run state *
```

Words 39-40: last jump addresses *
Words 41-44: interpreter work area (may be used as temporary storage by machine code routines if required but must not be accessed by CINTCODE)
Words 45-55: reserved for use by future packages (e.g. floating point)
Words 56-71: available for users
Words 72 on: used by operating system.

Stacks

The following global variables are associated with stacks:

COLIST (166 S)	Head of a chain through all the run-state stacks. Either the address of the first stack on the chain or 0 if there are no run-state stacks.
CURRCO (22 S)	Address of the current run-state stack. If there are no run-state stacks (COLIST=0) its value is undefined.
MAINSTACK (20 S)	Address of the main run-state stack (i.e. the one created for use by START). If there are no run-state stacks its value is undefined.
TRAPSTACK (19 S)	Address of the root stack.

Chapter 6 describes the layout of data on the stack on entry to a procedure. The first few words of each run-state stack are used for various special purposes however (the first few words of TRAPSTACK are slightly different):

Word 0: used when the stack is not the current stack to save the value of the stack pointer (initialised to point to word 5 of the stack)
Word 1: the parent stack (when a stack is created by CREATECO this word is set to CURRCO; when a stack is created for START it is set to -1)
Word 2: the COLIST chain word. Address of next stack in the chain or 0
Word 3: address of the main procedure for the stack (i.e. START or the first parameter to CREATECO)
Word 4: size of the stack
Word 5: address of the stack (i.e. pointer to word 0).

Words 6 onwards are initialised to 0.

Store files

The global STOREFILES (16 S) heads a chain linking the file control blocks of all store files. It is 0 if there are no store files.

The global LINKEDFILES (114 S) heads a chain linking all linked store files (excluding files which are linked as system files). It is 0 if there are no linked files.

The format of a file control block is:

Word 0: the STOREFILES chain
Word 1: -3 file is linked as a system file
-2 file is neither linked nor loaded
-1 file is loaded
>=0 file is linked and this word is the LINKEDFILES chain
Word 2: address of the first file data block or 0 if this is a 0-length file
Word 3: address of the file name block.

A file name block is simply a BCPL string containing the file name.

The format of a file data block is:

Word 0 : address of the next file data block or 0 if this is the last file data block
Word 1 : number of bytes of data in this block
Words 2 on : data.

Note that when a file is being written, word 1 in the current file data block is not updated until the block is full or the file is closed. When a file is being written blocks of a certain size are allocated (see the next section). When one block is full another is allocated. When the file is closed the last block is truncated to the minimum size big enough to contain the data written.

Stream control blocks

A stream control block is created for each stream opened by FIND(X)INPUT and FIND(X)OUTPUT. The format is as follows:

Word 0: the STREAMCHAIN chain (i.e. address of next SCB in chain or 0).
Word 1: address of the 'get function'. The 'get function' is a procedure which is called when further physical input is required. It reads characters into the input buffer and sets the globals CISPOS and CISLIM (see below) as appropriate. The result of the function is 0 for success, non-0 for failure (treated as ENDSTREAMCH).
Word 2: address of the 'put routine'. The 'put routine' is a procedure which is called when physical output is required. It writes the contents of the buffer to the physical device and resets the global COSPOS (see below). It returns 0 for success, non-0 for failure (treated by calling ABORT(101)).

Word 3: address of the 'close routine'. This routine is called when the stream is closed (by ENDREAD, ENDWRITE or TIDY) with two parameters - 0 and the SCB address (when the routine is called the stream is no longer one of the currently selected streams). It does not return a result.

Word 4: type word:
bit 0: set for an input stream
bit 1: set for an output stream
bit 2: set for a terminal stream
bit 3: set for an interactive stream

Word 5: device:
1 /K (keyboard read a key at a time)
2 /C (console)
3 /P (serial port)
4 /E (errorstream)
5 /L (printer)
6 /N (null device)
7 /S. (store file)
8 /F. (current filing system file)

Word 6: current buffer position. For input streams this is the byte offset within the buffer of the next character to read. For output streams it is the byte offset within the buffer of where the next character is to be written.

Word 7: buffer limit. The byte offset of the last position in the buffer.

Word 8: address of the buffer for the stream.

For streams accessing store files the following extra words are used:

Word 9: address of the store file file control block.

Word 10: size of the file data blocks to be allocated (default 128 words). A value of -1 means that blocks as big as possible are to be used. This word is only applicable to output streams.

For streams accessing current filing system files the following extra words are used:

Word 9: a one-character buffer.
Word 10: the 'handle' returned by the call of OSFIND (shifted left 8 bits).
Word 11: a BCPL string containing the file name. The size of SCB allocated allows for up to 15 characters. If the name is longer than this then it is truncated.

For console input streams the stream control block from word 9 onwards is an area in the format required by OSWORD(0) - read a line from the current operating system input device. Specifically:

Word 9: the byte address of the data buffer (which begins at word 12).
Word 10: byte 0 is the maximum line length (127 characters). Byte 1 is the low limit on the range of characters accepted (32).
Word 11: the high limit on the range of characters accepted (255).

For all other streams the following extra word is used:

Word 9: a one-character buffer.

Note that words 6, 7 and 8 of the SCBs for the current streams may not be up to date, since these words are copied into global variables - see below.

Also note that output to the screen and the serial port bypasses the stream control block altogether. WRCH, WRBIN and WRITES are replaced by alternative versions which call OSWRCH directly.

Associated globals

CIS (9)	the address of the SCB for the current input stream.
CISBUF (49) } CISLIM (50) } CISPOS (51) }	words 8, 7 and 6 respectively of the SCB for the current input stream. These globals, rather than the SCB, are updated while the stream remains the current stream and are copied back to the SCB by SELECTINPUT.
COS (10)	the address of the SCB for the current output stream.
COSBUF (55) } COSLIM (56) } COSPOS (57) }	as CISBUF etc. but for the current output stream.
STREAMCHAIN (17 S)	head of a chain linking all the stream control blocks.

System index

The global SYSINDEX (18 S) points to a data area which contains or points to various data items fundamental to the operation of the BCPL system. All of the offsets used are allocated MANIFEST names which are declared in SYSHDR. These names are used here followed by the value of the manifest in brackets. Some items within the system index are not used but space is reserved for them for compatibility with other implementations. The offsets within the system index are:

I.LIBBASE (0)	address of the start of the BCPL library in the ROM
I.TRST (2)	address of the trap count. When a run-state program traps the reason is stored in the trap count area. Several traps may occur on the same CINTCODE instruction so the first byte of the area is a count of the number of traps and successive bytes contain letters identifying the traps as follows:

B abort
E attempt to restart program after fatal trap/program end without re-initialising
G call of uninitialised global procedure
I requested trap on next instruction
J requested trap on next jump
K requested trap on next procedure call/return
L breakpoint reached
N count reached
S stack overflow
X non-existent CINTCODE instruction
Z divide by zero.

I.FLAGS (3) address of the flags area used mainly by DEBUG. Note that these flags are ignored unless the appropriate bits in the current state are set (see below). The structure of this six-word area is:

0: low byte non-0 if trap on next instruction required
1: low byte non-0 if trap on next jump required
2: low byte non-0 if trap on next call/return required
3: low byte non-0 if trap when count (in word 5) reaches 0 is required
4: type of last trap to command state as follows:

-3 no program to start
-2 fatal trap
-1 abort
0 program ended
1 program initialised
2 breakpoint
3 requested trap on next instruction/jump/call/return or count up
4 **ESCAPE** pressed
5 run out of heap space
6 PAUSE command executed

5: if word 3 is non-0 this word is decremented on each jump/call/return.

I.LIMIT (4) address of limits area. This is a two-word area. Word 0 contains the negative of the address of the top of the current stack. Word 1 contains the negative of the maximum allowable value of the stack pointer.

I.CSTATE (5)	address of the current state area. This area is used by the interpreter for the registers etc. used by CINTCODE. Its structure is described below.
I.TSTATE (6)	address of the last saved trap state. This area is used to save the current state for the command state while the system is in the run state.
I.RSTATE (7)	address of the last saved run state. This area is used to save the current state for the run state while the system is in the command state. When testing a program with DEBUG this area may be examined to determine the state of the program.
I.TIME (8)	address of the count area. This area is a double-word count of the number of jumps/calls/returns made (in the format described for WRITEDB in chapter 5).
I.JADD (9)	address of the last jump addresses area. This area holds the ‘from’ and ‘to’ byte addresses involved in the last jump/call/return. It is used by DEBUG.
I.DEFSPACE (12)	the number of words to be reserved for extra character definitions (see chapter 8).
I.RESTART (13)	the byte address to call (using CALLBYTE) to restart BCPL reserving space for extra character definitions (see chapter 8).

Current state

The structure of the current state is as follows:

R.MCST (0)	the machine state. The following bits are defined: bit 0 (M.TIMED) set if the count in I.TIME is to be updated bit 1 (M.TRAPESC) set if ESCAPE should trap to the command state bit 2 (M.TRAPGV) set if GETVEC should trap to the command state if it cannot allocate a vector bit 7 (M.TRAPFLAGS) set if the 'trap' flags and count in I.FLAGS are to be processed (also causes the count in I.TIME to be updated) bit 8 (M.TRAPPED) set if the system is in the command state Bits 0-7 may only be set in the run state. The defaults are bits 0 and 7 clear and bits 1 and 2 set. DEBUG sets bits 0 and 7. Note that the system runs faster if these bits are clear.
R.CURRCO (1)	address of the current stack.
R.SP (2)	the stack pointer.
R.PC (3)	the CINTCODE program counter (holds byte address of next CINTCODE instruction).
R.A (4)	the CINTCODE A register.
R.B (5)	the CINTCODE B register.
R.C (6)	the CINTCODE C register.

System saved data

Certain global variables (e.g. RESULT2) are used by both run-state and command-state code, but must have their run-state value preserved when command-state code is executing. The system saved data is an area used to preserve run-state globals while the command state is executing and vice versa.

The global SYSDATA (14) points to an area structured as follows:

```
Word   0: not used
Word   1: command-state ABORTLABEL
Word   2: command-state ABORTLEVEL
Word   3: command-state PRGENDLEVEL
Word   4: command-state PRGENDLABEL
Word   5: command-state CIS
Word   6: command-state COS
Word   7: command-state MCRESULT
Word   8: command-state FNAMBUF
Word   9: command-state STOP
Word  10: not used
Word  11: command-state RESULT2
Words 12 to 140:
          command-state buffer pointed to by
          FNAMBUF (see FILENAME in the next
          section)
Word 141: run-state ABORTCODE
Word 142: run-state ABORTLABEL
Word 143: run-state ABORTLEVEL
Word 144: run-state PRGENDLEVEL
Word 145: run-state PRGENDLABEL
Word 146: run-state CIS
Word 147: run-state COS
Word 148: run-state MCRESULT
Word 149: run-state FNAMBUF
Word 150: run-state STOP
Word 151: not used
Word 152: run-state RESULT2
Words 153 to 281:
          run-state buffer pointed to by FNAMBUF.
```

The first 141 words are only valid when the system is in the run state. The second 141 words are only valid when the system is in the trap state.

Note that when using DEBUG to test a program it is necessary to use SYSDATA to examine/alter RESULT2, MCRESULT etc. for the program under test.

SYSTEM PROCEDURES

This section describes various global procedures used by the BCPL system. The names of the procedures are given even when no header file declaring the name is provided. The names are followed by the global number (and S for SYSHDR if appropriate) in brackets.

CHANGECO (32)

Swaps coroutines. Called by:

```
p1 := CHANGECO( p2, cortn)
```

Control returns to the statement after the call but the current stack is now **cortn**. **p2** is copied to **p1**. If **cortn** is -1 it performs an end of program trap.

CLOSESTREAM (120)

Closes a stream. Called by:

```
CLOSESTREAM( stream)
```

CONTPRG (149 S)

Called by debugging utilities to continue the current run-state program. It saves the command-state globals in SYSDATA, restores the run-state globals and calls ENDTRAP (see below). On return from ENDTRAP it saves the run-state globals in SYSDATA and restores the command-state globals (except that it always reselects CNSLINSTR and CNSLOUTSTR as the current streams). Called by:

```
res := CONTPRG( par)
```

where **res** and **par** are as for ENDTRAP.

ENDTRAP (34 S)

Enters the run state from the command state. Called by:

    res := ENDTRAP(par)

where **par** is either 0 (in which case execution resumes with the instruction at the run-state program counter) or a CINTCODE instruction code (in which case this instruction is executed as if it were the instruction at the run-state program counter). Control returns to the caller when the run-state program traps or finishes. **res** is the trap type as listed under I.FLAGS on page 427.

FILENAME (60)

Parses a device/filename string and strips off the device prefix (if any). The ‘pure’ file name is set up as a BCPL string in the buffer pointed to by global FNAMBUF (12). The string is followed by a byte containing OD hex.

Called by:

    dev := FILENAME(string, type)

If **type** is 0 then **dev** is the device code (values listed on page 423) or 0 if no device is specified in **string** or -1 if the device is invalid.

If **type** is 1 then **dev** is as above except that if no device is specified in **string** then **dev** is set to 7 if a store file of the specified name exists else it is set to 8.

If **type** is 2 then **dev** is as above except that if no device is specified in **string** then **dev** is set to 7.

If **type** is -1 then **string** is simply copied to FNAMBUF and **dev** is set to 0.

FINDSTFILE (148)
Checks if a store file of a given name exists. Called by:

```
fcb := FINDSTFILE( string)
```

If a store file called **string** exists then **fcb** is set to the address of the file control block, otherwise it is set to 0.

RDTOBLOCK (209)
Reads from the current input stream into a vector. Called by:

```
eof := RDTOBLOCK( vector, count)
```

count bytes are read into the **vector** starting at address **vector**+2. **vector**!1 contains the number of bytes actually read. **eof** is TRUE if the end of the input stream has been reached (i.e. if the next read would give ENDSTREAMCH).

SFCNTRL (113)
Performs miscellaneous operations on store files. Five types of call:

```
is.successful := SFCNTRL( name, 1)
```

deletes the specified file.

```
is.successful := SFCNTRL( name, 2, block)
```

extends the file with the block (equivalent to EXTSFILE).

```
is.successful := SFCNTRL( name, 3, p3)
```

makes the file contiguous. Fails if the file is open unless the file is open for read and is already contiguous and **p3** is true.

```
is.successful := SFCNTRL( name, 4)
```

creates a 0-length file.

```
length := SFCNTRL( name, 5)
```

returns the GETVEC parameter needed to allocate a file data block big enough to hold the contents of the file. A result of 0 indicates an error.

SFSTATE (124)

Returns information about a store file. Called by:

```
bit.set := SFSTATE( fcb, mask)
```

fcb is a store file file control block. **mask** is a bit mask with bits as follows:

```
bit 0: file is open for read
bit 1: file is open for write
bit 2: file is linked
bit 3: file is loaded.
```

bit.set is TRUE if at least one of the bits in **mask** is true for the file. The values of all the bits for the file are stored in MCRESULT.

STRCNTL (61)

Opens a stream. Called by:

```
scb := STRCNTL( name, type, dev, bsize)
```

name is the stream name.

type is 1 for input, 2 for output.

dev is -1 if **name** has not been stripped of any device prefix, else it is the device code returned by FILENAME (see above). Note that it must not be 0.

bsize only applies to output store files and is the required file data block size or -1 if the file data blocks are to be as big as possible.

scb is the address of the stream control block or 0 for failure.

TRAPSTART (122)

The equivalent of START for command-state programs. See ‘Writing command-state programs’ in this Appendix for more details.

TRUNCVEC (103)

Given a vector in the format of a file data block, it truncates the vector if it is bigger than it need be to hold the data content. Called by:

```
TRUNCVEC( vector)
```

USE OF OPERATING SYSTEM ROUTINES

This section lists the main operating system calls made by the system when performing input and output to devices other than store files and the null device. It should be noted that the system calls SELECTINPUT and SELECTOUTPUT in many different places and therefore the operating system calls made by these routines may be invoked by a wide range of operations.

In various places the calls used depend on the result of calling the procedure FSTYPE described in chapter 6.

The I/O operations can be divided into the following categories:

(1) opening a stream (FINDINPUT, FINDXINPUT, FINDOUTPUT, FINDXOUTPUT);

(2) closing a stream (ENDREAD, ENDWRITE, TIDY);

(3) reading from a stream (RDCH, RDBIN, READWORDS etc.);

(4) writing to a stream (WRCH, WRBIN, WRITEWORDS etc.)

(5) selecting a stream (SELECTINPUT, SELECTOUTPUT);

(6) reading an entire file (FILETOVEC, LOAD, READ, READFILE);

(7) writing an entire file (VECTOFILE, SAVE, SAVEVEC);

(8) deleting a file (DELFILE, DELXFILE);

(9) renaming a file (RENAME).

Opening a stream

If the device is the current filing system then:

(1) (depending on FSTYPE) *FX139 calls to select error/message options as described in chapter 6;
(2) SELECTOUTPUT to select the console;
(3) (depending on FSTYPE) a message giving the file name is displayed;
(4) OSFIND to open the file;
(5) SELECTOUTPUT to restore the previously selected output stream.

For all other devices no calls are made.

Closing a stream

If the device is the current filing system then:

(1) *FX3,4 to select the screen for OSWRCH;
(2) OSFIND to close the file;
(3) *FX3,0 to restore the previous output device (*FX3,7 if the previous output device was the serial port).

For all other devices no calls are made.

Reading from a stream

For the console:

(1) SELECTOUTPUT for output to the console;
(2) OSWORD to read a line of input;
(3) SELECTOUTPUT to restore the previous stream.

For the keyboard and the serial port:

OSRDCH to read a character.

For single-character reads from the current filing system:

(1) (if console must be output device) *FX3,4;
(2) OSBGET to read one byte;
(3) (if console must be output device) *FX3,0 or *FX3,7 depending on the current output stream.

READWORDS is equivalent to multiple single-byte reads unless the device is a filing system that supports OSGBPB in which case:

OSGBPB to read the required number of words.

Writing to a stream

For all devices but the printer and the current filing system:

OSWRCH to write a character.

For the printer:

(1) OSBYTE(#XF5) to read the printer ignore character. Carry on only if this is not the character to be printed;
(2) OSWRCH to write CTRL-A (send next character to printer only);
(3) OSWRCH to write the specified character.

For single-character writes to the current filing system:

OSBPUT to write one byte.

WRITEWORDS is equivalent to multiple single-byte writes unless the device is a filing system supporting OSGBPB in which case:

OSGBPB to write the required number of words.

Selecting a stream

For SELECTINPUT:

if the stream is the keyboard or console then *FX2,0 else if it is the serial port then *FX2,1 else no operating system calls are made.

For SELECTOUTPUT:

(1) *FX3,0 to restore the normal OSWRCH destination;
(2) if the old output device was the printer then OSWRCH to write CTRL-C (disable printer);
(3) if the new device is the serial port then *FX3,7 else if it is the printer then OSWRCH to write CTRL-B (enable printer) else no further operating system calls are made.

Note that SELECTOUTPUT swaps between two versions of each of WRBIN, WRCH and WRITES. One set is used for output to the screen and serial port and the other set for all other devices.

Reading an entire file

If the current filing system does not support the use of OSFILE to find the length of a file, or if there is no suitable contiguous area to read the file into then reading an entire file is equivalent to opening it, reading blocks by READWORDS then closing it. All these operations have been covered above.

Otherwise:

(1) (depending on FSTYPE) *FX139 calls to select error/message options as described in chapter 6;
(2) SELECTOUTPUT to select the console;
(3) (depending on FSTYPE) a message giving the file name is displayed;

(4) OSFILE to read the file's catalogue information;
(5) SELECTOUTPUT to restore the previously selected output stream;
(6) at this stage it may be found that there is no area big enough to read the file contiguously and so it may be read conventionally as described above;
(7) (depending on FSTYPE) *FX139 calls to select error/message options as described in chapter 6;
(8) SELECTOUTPUT to select the console;
(9) (depending on FSTYPE) a message giving the file name is displayed;
(10) OSFILE to read the file;
(11) SELECTOUTPUT to restore the previously selected output stream.

Writing an entire file

If the data to be written is not contiguous then the file is written by opening it, writing blocks with WRITEWORDS then closing it. All these operations have been covered above.

Otherwise:

(1) (depending on FSTYPE) *FX139 calls to select error/message options as described in chapter 6;
(2) SELECTOUTPUT to select the console;
(3) (depending on FSTYPE) a message giving the file name is displayed;
(4) OSFILE to write the file;
(5) SELECTOUTPUT to restore the previously selected output stream.

Deleting a file

OSFILE to delete the file.

Renaming a file

RUNPROG is used to issue a '*RENAME' command.

WRITING COMMAND-STATE PROGRAMS

Two reasons for writing a program to run in the command state rather than in the run state are:

- to produce a utility which can be run while a run-state program is suspended without affecting that program;

- to produce a debugging aid.

Writing utilities

The general approach recommended is to write and test the program as a run-state program (so that all the debugging aids can be used) then convert it to a command-state program by replacing START with TRAPSTART (global 122 declared in SYSHDR).

The principal differences between trap-state and command-state programs that the author of command-state programs must be aware of are:

- there is no equivalent of STARTINIT. The program must run using the root stack (on entry to the program there are approximately 400 words available on the stack);

- the program must handle **ESCAPE** and GETVEC failure itself (see chapter 6 for details);

- the program must not call TRAP, or RUNPROG any program or command that might call TRAP (e.g. PAUSE). Any event that causes a run-state program to trap to the command state (e.g. division by zero) stops the system if it occurs in the command state;

- the procedure STACKSIZE cannot be used;

- coroutines cannot be used;

- on program completion TIDY is not called so the program must ensure it unlinks any files linked, FREEVECs all vectors obtained by GETVEC and closes all streams it opens;

- the system debugging aids cannot easily be used to debug a trap-state program.

Writing debugging aids

All the remarks in the preceding section naturally apply to writing debugging aids.

Normally a debugging aid is used by using LINK and INIT to initialise the program to be debugged then running the debugging aid. The procedure CONTPRG (see 'System procedures' above) can be used to run the program to be debugged. Note that any event that normally causes a trap to the command state now returns to the debugging aid, which must generate appropriate messages etc. itself.

Various facilities are built into the interpreter to trap to a debugging aid on every instruction, every jump or every call/return or to trap after a given number of jumps/calls/returns have been executed. The data areas associated with these facilities are detailed on page 425.

Index

BCPL

for the BBC Microcomputer

About this book

This is the essential reference manual for the BCPL system on the BBC Microcomputer. It describes the functions supported by the BCPL language ROM, and the use of the BCPL Compiler, the Screen Editor, the Assembler, and the other utilities which are part of the BCPL language package. It also contains the background information to enable the most demanding users to take full advantage of the wide range of features provided.

Acornsoft Limited, 4a Market Hill, Cambridge, CB2 3NJ, England

This BCPL implementation was developed by Richards Computer Products Ltd., a company that specialises in providing BCPL systems. It was developed in collaboration with Dr Martin Richards of the Computing Laboratory, Cambridge University, who invented BCPL in 1967.

ISBN 0 907876 10 2

SBD10