THE IMPLEMENTATION OF BCPL ON A MICROCOMPUTER

By

RONALD STEWART HAYTER

B.Sc., The University of British Columbia, 1978

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

October 1983

Department of Computer Science

The University of British Columbia
1956 Main Mall
Vancouver, Canada
V6T 1Y3

Date 1983 Oct 14

Abstract

BCPL/Z80 is a complete system for the development of BCPL programs. It runs on a microcomputer based on the Zilog Z80 processor and consists of a number of independent programs coordinated by an operating system. Among these programs are included an editor, a compiler, and a text formatter, all of which were designed for use on computers both faster and larger than an 8-bit machine. The limited resources of a microcomputer made implementation of a stand-alone program development system for a high-level language challenging. This document describes how this task was accomplished.

Table of Contents

Acknowledgement

I would like to thank John Peck, my supervisor, for his guidance and our many long discussions, and Harvey Abramson for his comments on this document. I am also indebted to the authors of some of the programs incorporated into BCPL/Z80: J E L Peck (the compiler, the CHEF editor, the DORIS formatter), M A Maclean (CHEF), J D Dyment (DORIS), M Richards (the compiler), and a large number of others (the compiler).

1. Introduction

A program development system is the set of programs needed for developing programs in a high-level language. Normally this set of programs runs on a particular computer under some operating system. At a minimum, this set includes: a text editor, for composing and correcting programs and for writing documentation; and either (depending on the high-level language) a compiler, for translating programs into executable form, or an interpreter, for executing programs. Also sometimes included are a few utilities: a text formatter, for arranging documentation on printed pages; a linker, for combining the separately-compiled parts of a program; a debugger, for testing programs and determining the cause of errors; a profiler, for locating bottlenecks in the execution of a program; and perhaps a code optimizer, for improving the size and speed of programs automatically. BCPL/Z80 is such a program development system, running on a Z80-based microcomputer.

Chapter 2 provides background for the work on BCPL/Z80. It describes a number of program development systems for microcomputers. One such system, the UCSD p-System, is used as a yardstick for comparison with BCPL/Z80 throughout this document. Also described are two implementations of BCPL for the Z80, the SLIM intermediate language (a central element in a number of BCPL implementations at the University of British Columbia, including BCPL/Z80), and the Z80 processor. Chapter 3 gives a brief description of the BCPL/Z80 hardware and software. Chapter 4 discusses a number of important implementation issues which were resolved during the development of BCPL/Z80. In particular, the reasons for choosing to interpret compiled code rather than translate it to Z80 code are given. The next three chapters describe the three successive generations of the interpreter used in BCPL/Z80. Chapter 8 compares the performance of BCPL/Z80 with other systems. Chapter 9 contains some concluding remarks. Included as an appendix is the user's manual for the system.

2. Background

In this chapter, a number of program development systems are briefly described. After this, several implementations of BCPL are examined. Finally, the SLIM intermediate machine and the Zilog Z80 processor are described.

8-Bit Program Development Systems

In the world of 8-bit microcomputers, there are a number of program development systems for high-level languages available commercially. Unfortunately, little has been published about them.

A number of these systems are based on the popular CP/M operating system from Digital Research [Kild81]. CP/M is not a pleasant system for naive users but, being one of the first available, much software has been written for it. From various software houses are available powerful full-screen editors as well as compilers for many high-level languages, including Pascal, C, and PL/I. Most compilers produce native machine code, usually for the Intel 8080 processor. Some compilers also emit intermediate code meant for interpretation; by doing so, compile time is shortened during the early debugging stages when the slower speed of program execution is unimportant.

Another program development system is the UCSD p-System [Over80], developed at the University of California at San Diego. It is available for a wide range of 8- and 16-bit machines. The p-System is an easy-to-use system for developing Pascal programs, although compilers are also available for FORTRAN, COBOL, BASIC, and Modula-2. The full-screen editor, the compiler, and a number of utilities are all integrated into an exceptionally pleasant package. The p-System compiler produces intermediate p-code which is then interpreted. The latest version of the p-System (version IV.1) also allows p-code to be translated into native machine code.

BCPL on Microcomputers

The language BCPL has been implemented on many computers of all sizes. One reason for its being so widespread is its portable compiler [Rich71]. This compiler produces code for an intermediate (hypothetical) machine: either OCODE or INTCODE. Programs in the form of, say, OCODE can be easily transported by writing an OCODE interpreter on the new machine. In particular, the compiler, which is written in BCPL, can be transported in this way.

For most machines, however, this method of transporting programs is used only for bootstrapping, since interpreted programs generally run much slower than those translated to the native machine code. Once the compiler, in OCODE form, is running on the new machine, a translator from OCODE to the native code is usually written. By this use of an intermediate machine code, BCPL can be implemented on a new machine by writing only an interpreter and a translator--usually much less work than writing a whole compiler.

Recently, BCPL was implemented on a Z80-based microcomputer under CP/M by Cowderoy and Wallis [Cowd82]. They wrote an OCODE interpreter and transported the compiler to their machine. They did not, however, write a translator, apparently because of the limited memory (36 Kbytes) in their machine. Their compiler is 12 Kbytes of OCODE, but it is 19.5 Kbytes when translated into PDP-11 code; presumably it would be even larger if translated to Z80 code. Rather than reduce the amount of memory left for the symbol table and the parse tree, they accepted a slower compilation speed. They mentioned that their interpreted BCPL compiler on the Z80 is a factor of 8 slower than the directly-executed PDP-11 version. Unfortunately, they neglected to mention which model of PDP-11 they used.

BCPL CINTCODE, by Richards Computer Products [RCP81], also runs under CP/M. Their compiler produces a compact form of INTCODE rather than OCODE, but their system is also interpreted, not translated into native machine code. BCPL CINTCODE is a more ambitious implementation than that of Cowderoy and Wallis, including not only a compiler and an interpreter but also a number of utility programs, debugging aids, and support for overlays and multi-tasking. Although the literature describing BCPL CINTCODE does not include any quantitative performance figures, a few comparisons are given:

> A typical BCPL program in CINTCODE requires about a third of the storage of fully compiled Z80 code.

and:

> BCPL CINTCODE is significantly more compact than UCSD Pascal, and runs faster.

SLIM Intermediate Machine

As mentioned above, most BCPL compilers produce either OCODE or INTCODE as their intermediate code. One exception is a compiler written at the University of British Columbia. This compiler produces an intermediate code called SLIM [Peck83].

The SLIM machine is a hypothetical one, specifically designed for compiling BCPL. It has an accumulator, a stack, and a number of special-purpose registers.

The program counter (C-) register points to the next SLIM instruction to be executed. It is incremented as each instruction is executed but it is also changed by the 'jump' (J), 'jump if true' (T), 'jump if false' (F), 'call' (C), and 'return' (R) instructions.

The environment (E-) register points into the stack to the parameters and local variables of a procedure; the parameters are at negative offsets from the E-register, while the local variables are at positive offsets. It is modified by the 'call' (C) and 'return' (R) instructions.

The high-point (H-) register points to the current top of the stack. It is incremented by the 'push' (P), and 'push then load' (PL) instructions, and changed by the 'modify high-point' (M), 'call' (C), and 'return' (R) instructions.

The global (G-) register points to the base of a vector of cells for BCPL global variables.

(The other two registers, the stack limit (S-) and interrupt (N-) registers, are not important to this discussion.)

Most SLIM instructions implicitly involve the accumulator, and most of these also explicitly involve an operand. For example, in the following sequence of instructions:

```
LIE1 +5 SG101
```

the 'load' (L) instruction loads a value into the accumulator from the first local variable of the current procedure, then the 'add' (+) instruction adds five to the accumulator, and finally the 'store' (S) instruction stores the accumulator in global cell 101.

SLIM has been used to transport BCPL to several machines, ranging in size from an Amdahl 470 to a Data General Nova and, recently, to a 16-bit microcomputer based on the Intel 8088 processor. In all of these implementations, SLIM was translated into the native machine code.

Because SLIM was designed for the representation of compiled BCPL programs rather than as the instruction set of a real machine, the encoding of SLIM instructions as bit patterns is not defined. Such an encoding for a 16-bit machine has been suggested (chapter 10 of [Peck83]) and it was used in a microprogrammed implementation of SLIM. This scheme encodes a SLIM instruction as either one or two 16-bit words. It is designed so that the instructions that are encountered most frequently (those with operands which are small constants or small offsets from base registers) require only one word to encode. Relatively few instructions require two words. Although this encoding for SLIM has been suggested, an implementor is free to choose other encodings.

Zilog Z80 Processor

The Zilog Z80 [Zilo76] is an 8-bit microprocessor. It is a superset of the earlier (and once very popular) Intel 8080. Because the Z80 is a superset, it has surpassed the 8080 in popularity. However, it has inherited an awkward architecture and, despite its new instructions, the instruction set is incomplete.

The Z80 has three 16-bit general-purpose registers (BC, DE, and HL) which alternatively may be used as six 8-bit registers (B, C, D, E, H, and L). It also has an 8-bit accumulator (A), and a number of other special-purpose registers, including a program counter (PC), a stack pointer (SP), a pair of index registers (IX and IY), and a flags register (F) for condition codes.

Ten different addressing modes are defined. Unfortunately, there are many restrictions placed on their use. Not all registers may be used with some modes, and most instructions allow only certain modes to be used. As a result, A and HL are the most useful registers and data must often be transferred between these registers and the others.

Most arithmetic and logical instructions involve the accumulator implicitly. Some of the operations available are:

add, subtract, increment, decrement, compare, and, inclusive-or, exclusive-or, shift, rotate, set bit, reset bit, and test bit. As well as these 8-bit operations, a few 16-bit ones are also available: add, subtract, increment, and decrement.

Data can be transferred between registers, or between registers and memory, either 8 or 16 bits at a time, by the 'load' (LD) instruction. Because of addressing mode restrictions, however, it is often necessary when tranferring data between registers to use memory temporarily.

The program counter is changed by the 'jump' (JP and JR), 'call' (CALL) and 'return' (RET) instructions. Jumps may be relative to the current PC (JR) or to absolute memory locations (JP); calls can only be to absolute locations. The PC is pushed on the stack by a call and popped by a return. There is no addressing mode which easily allows an arbitrary word in the stack to be referenced; only the topmost word may be accessed, by 'push' and 'pop'.

3. Overview of BCPL/Z80

BCPL/Z80 is a self-contained program development system consisting of an operating system, a number of independent utility programs, and a file system. This chapter gives an overview of the facilities available. The use of BCPL/Z80 is described more fully in the user's manual [Hayt83] in Appendix C.

Operating System

A program called the Shell allows the user to run any of the utility programs or any of the user's own. The user can select the particular utility from a menu of choices simply by typing a single letter.

Programs normally have their standard input and output directed to the console. However, like the shell of UNIX [Ritc78], the BCPL/Z80 Shell allows them to be redirected to files.

A subsystem of the Shell is called the Filer. The Filer is used for manipulating files: listing, renaming, and destroying files, and moving files from one disk to another. It, too, is menu-driven.

Program Development

Programs and documents may be composed and modified with the aid of the Editor. It includes the many features found in most large-system line-oriented editors, but also allows full-screen editing. The Compiler translates BCPL programs into SLIM assembly language. These SLIM programs usually can be made smaller and faster by the Improver. The Encoder translates SLIM assembly language programs into executable code. The separately-compiled sections of a program are merged by the Linker. Finally, the Formatter prepares documents for printing.

File System

The BCPL/Z80 file system is organized as a two-level hierarchy. In the first level are two kinds of volumes: character and block. Typical character volumes are the keyboard, the screen, the printer, and the modem. Reading from or writing to a character volume is done one character at a time. The character volumes are known by the names "CONSOLE:" (the keyboard and the screen), "PRINTER:" (the printer), and "REMOTE:" (the modem).

Block volumes are disks and, for these, information is transferred in blocks of 512 bytes at a time. The names of block volumes are chosen by the user at the time that a disk is initialized. Block volumes are structured objects, subdivided into files (the second level in the hierarchy) and a directory. Just as with character volumes, characters may be read from or written to files one character at a time. File names are, of course, chosen by the user.

Volume and file names are used for opening streams to character volumes or files. For example, an input stream from the modem may be established by calling 'FindInput':

```
FindInput ("remote:")
```

Similarly, an output stream may be set up to a file named "OUT.TEXT" on the disk "WORK:" by calling 'FindOutput':

```
FindOutput ("work:out.text")
```

Hardware

BCPL/Z80 was developed on an Exidy Sorcerer microcomputer having a 2.1 MHz Z80 processor and 55 Kbytes of memory. The computer has a built-in keyboard and memory-mapped video screen, as well as both printer and modem interfaces. Attached to the computer is a North Star mini-floppy disk system consisting of two double-density drives, with a combined capacity of 350 Kbytes.

4. Implementation Issues

In implementing something as large as a program development system for a high-level language, many issues must be resolved. This chapter discusses how they were resolved for BCPL/Z80.

Operating System

Much of the early development of BCPL/Z80 was done using the UCSD p-System (version I.5). Superb as it is as a program development system, the p-System was judged to be unsuitable as a host for BCPL/Z80 for several reasons. The p-System occupies about one-third of memory, leaving only 40 Kbytes for user programs. Also, the file system cannot be accessed by a Z80 assembly program; all file manipulation must be done by Pascal code.

The only other operating system available was a primitive one called the North Star Disk Operating System (DOS). DOS is much smaller than the p-System: only 4 Kbytes. It does have an assembly language interface to its file system but the file system itself is very crude compared to that of the p-System. In fact, it consists of only three procedures: one to read a disk directory and locate a file name in it, another to write a directory back on the disk, and the last to read or write a number of disk sectors beginning at a given sector. It is left to the programmer to supply procedures which allocate space on the disk for files, open and close files, and read and write characters. DOS provides little in the way of support for application programs and, for this reason, it too was rejected.

After the available operating systems were rejected, it was decided to write a new one in BCPL. This simple operating system is for a single user, and is very similar to the p-System. It was not only modelled after the p-System, but it also has compatible disk directory and file structures. This compatibility was particularly important before BCPL/Z80 had a full-screen editor. Because of it, for example, programs could be composed using the p-System editor and compiled with the BCPL/Z80 Compiler.

UNIX was another influence on the design of the BCPL/Z80 operating system. The Shells of both systems are simply user programs (although rather sophisticated user programs) and so may be replaced with others if desired. The other useful idea borrowed from UNIX is that of the redirection of the standard input and output of a program. Redirection is not possible in the early versions of the p-System, but this deficiency has been corrected in the latest version.

Another possible host for BCPL/Z80 might have been CP/M. It is almost as small as DOS and yet it has the extensive file system interface that DOS lacks. When a decision had to be made, however, this operating system was not available for the computer used. Although CP/M is now available, the BCPL/Z80 operating system works well and is easier to use. Adapting BCPL/Z80 to run under CP/M would likely be a serious undertaking, but it might be attempted in the future.

Run-Time Library

One of the distinguishing characteristics of BCPL is that it is a small language. Many of the facilities that are built into other languages, such as input/output and storage allocation, are ordinary procedures in BCPL. The proposed draft BCPL standard [Eage82] defines a large number of procedures which might be included in the run-time library.

Many of these procedures are present in BCPL/Z80, including the following: the opening and closing of streams to files, devices, and in-memory character strings; the reading and writing of single characters or arbitrary numbers of characters; the reading and writing of formatted text; the positioning of streams to arbitrary points; several string processing operations; the dynamic allocation of memory from the stack or a heap; and the calling of assembly language procedures. There are, in addition to these standard ones, a number of other procedures: for the renaming and destroying of files; for the positioning of the cursor on the screen of the console; and for the loading and unloading of program overlays. An optional library permits the use of coroutines and multi-tasking. All except a very few are written in BCPL rather than Z80 assembler, making implementation quick and modification easy.

The BCPL/Z80 run-time library was also influenced by the UCSD p-System. As mentioned in the previous section, the disk directories and files used by BCPL/Z80 are compatible with those of the p-System. In addition, the library was designed to be no more machine-dependent than the p-System.

The p-System runs on many different machines. It is easily portable because it relies on only a few machine-dependent procedures (not, of course, including the p-code interpreter which is written in the assembly language of the processor). These procedures read or write disk sectors, and read characters from or write characters to the console, the printer, and the modem. These I/O procedures together with code which initially loads the p-System into memory are all that must be changed to adapt the p-System to another machine (having the same processor). BCPL/Z80 relies on these same machine-dependent procedures and so should be as portable as the p-System. However, this claim has not yet been tested.

Program Development Facilities

An author of a program development system for BCPL begins with a head start: BCPL was designed with portability in mind, and a number of large and portable programs have been written in BCPL.

The BCPL/SLIM compiler is one such program. It has been transferred to a number of different machines at the University of British Columbia. What is more, it is capable of running on machines with relatively small memories. It compiles programs in a few distinct passes and the code for each pass needs to be present in memory only during that pass. To further save memory, the internal form of the program (the parse tree plus the symbol table) may reside in a disk file rather than in memory.

The CHEF text editor [Macl81] is written in BCPL. It has many powerful operators (commands) including 'alter' which allows full-screen editing. CHEF was also designed to be portable [Peck81]. Its memory requirements are modest because CHEF uses a disk file to hold the text being edited. For computers with very small memories, the code for some of the less commonly-used operators can be put into overlays which are only brought into memory when needed.

The DORIS text formatter [Dyme82] is another BCPL program. It is quite powerful yet it is small and easy to use. DORIS is also portable, but largely because it uses standard BCPL and is not big enough to require overlays or temporary disk files.

With these portable programs available and working, it was decided to use them as the heart of BCPL/Z80. Since the compiler produces a SLIM assembly language file, the only missing component needed was a program to put such a file into executable form: either a SLIM assembler or a SLIM-to-Z80 translator.

Interpretation vs Translation

Perhaps the two biggest obstacles to successfully implementing a program development system on an 8-bit microcomputer are the slow speed of the processor and the small size of the memory. BCPL/Z80 was developed on a machine with a 2.1 MHz Z80 processor and 55 Kbytes of memory. Much effort, therefore, was expended making the system small enough to fit yet fast enough to be useful.

An early (and important) decision was to interpret SLIM rather than translate it to Z80 machine code. Two considerations decided the issue in favour of interpretation: the expected size of BCPL/Z80 programs, and the expected speed of BCPL/Z80.

The more important consideration was size. The BCPL compiler and the CHEF editor, the main components of BCPL/Z80, are large programs. Thus it was important to ensure that they would be small enough to fit in memory. Using the SLIM encoding for a 16-bit machine mentioned in Chapter 2, the compiler was estimated to be 25 Kbytes long and CHEF 26 Kbytes. These estimates do not include the memory needed for data structures; each requires several thousand more bytes and both can make use of all available memory to improve performance.

The translation of SLIM to Z80 machine code is, in a sense, easy. SLIM is a higher-level machine than the Z80 and so almost every SLIM instruction must be translated into many Z80 instructions. Consequently, to keep the size of translated programs reasonable, it is necessary to place the Z80 instructions corresponding to most SLIM instructions into procedures. The Z80 code generated for a SLIM instruction is then (usually) a load of the operand value into a register, followed by a call to the procedure implementing the instruction. If this simple scheme were used, a translated program would be roughly three times the size of the encoded program. For small programs, this inflation in size could be justified by the increased speed of execution. Programs as large as the compiler and CHEF, however, would simply be too large to fit into memory. A more sophisticated algorithm might be able to translate programs into a number of bytes comparable to the 16-bit encoding, but this possibility was not explored.

The other consideration was speed. An interpreted program is several times slower than a translated one. However, on a microcomputer, the absolute speed of a program is rarely important. What is important is whether the program is fast enough to do its job without annoying the user. The example provided by the UCSD p-System showed that an interpreted system could be sufficiently fast. Almost the entire p-System, including both the compiler and the editor, are Pascal programs translated to p-code and interpreted. The speed of these programs is impressive and certainly adequate.

Together, these two considerations lead to writing a SLIM interpreter for BCPL/Z80. A goal has been to equal (or surpass) the small size and high speed of the p-System. The next three chapters detail the pursuit of this goal.

5. The First SLIM Interpreter

The first SLIM interpreter on the Z80 evolved considerably during its lifetime. Originally, it was written entirely in UCSD Pascal. As might be expected of an interpreter which was itself being interpreted, it was very slow. Its speed was improved by about 20 times by recoding most of it in Z80 assembly language. It was, however, still quite slow when compared with p-code. It was eventually abandoned when it seemed that its performance could not be improved except by using an entirely different approach.

Instruction Interpretation

This first interpreter used the suggested encoding for a 16-bit machine. Interpretation of each instruction consisted of several steps:

1. determine the type of the operand of the instruction

2. calculate the modified operand, leaving the C-register pointing to the next instruction

3. determine the type of the opcode of the instruction

4. jump to the appropriate service routine

5. return to step 1.

Traps

Almost all of the interpreter was written in assembly language. The part written in Pascal was responsible for initialization and for handling exceptional circumstances. After loading a SLIM program into memory, the Pascal part called the assembly procedure 'Execute' which interpreted SLIM instructions repeatedly.

'Execute' returned to the Pascal part, an operation known as trapping, only when something exceptional occurred. A trap could occur for a number of reasons including dividing by zero, executing an illegal instruction, overflowing the SLIM stack, referencing a bad memory address, calling a missing global procedure, reaching a user-specified breakpoint, and executing the 'quit' instruction.

Recognizing a global procedure which had not been loaded into memory was accomplished by setting all of the cells in the SLIM global vector initially to negative numbers. The cells corresponding to the global procedures which were later loaded would contain the addresses of the procedures, always positive numbers. Other cells would remain negative. The service routine for the 'call' instruction checked whether the address of the called procedure was negative and, if so, it caused a trap.

To make it easier to determine which particular global procedure was missing, the values initially put into the cells were the complements of the global cell numbers. When such a trap occurred, the number of the missing global procedure was then simply the complement of the C-register.

Input/Output

As mentioned in the previous chapter, one of the problems with using the p-System as a host operating system is that it provides only a Pascal interface to its file system; files cannot be accessed from assembly language. This problem was overcome by making use of the trap for missing global procedures.

For most kinds of traps, the Pascal part of the interpreter displayed the cause of the trap on the console and waited for a command to be typed. However, if the trap was the result of a missing global procedure, and if the missing procedure was one of a small set, the trap was treated differently: not as an error but rather as a request.

Initially, the only procedures in this set were 'RdCh' and 'WrCh'. In response to traps for these BCPL procedures, the interpreter performed the corresponding Pascal operations, 'Read' and 'Write', on the console. Later, file input/output was added, with Pascal array indexes serving as BCPL stream numbers. 'FindInput' and 'FindOutput' were mapped into 'Reset' and 'Rewrite', respectively, and both 'EndRead' and 'EndWrite' were implemented by 'Close' (a UCSD Pascal extension). 'SelectInput', 'SelectOutput', 'Input', and 'Output' manipulated the indexes.

This technique of using the trap for missing global procedures proved to be so useful that later versions of the interpreter adopted it. When the p-System was abandoned, these traps were used as a way to give BCPL names to (a very few) procedures written in assembly language.

Performance

Despite the fact that most of the interpreter was written in assembly language for the sake of speed, compared to p-code this interpreter was still slow: a p-code program was more than twice as fast as an equivalent SLIM program. By manually counting Z80 instructions, it was determined that during the interpretation of some of the most common SLIM instructions (loads and stores) about half of the execution time of an instruction was spent in decoding (steps 1 and 3 above). Decoding was a time-consuming procedure because the Z80 is not well suited to extracting arbitrary groups of bits from a 16-bit word. Furthermore, to determine the opcode type of an instruction requires sequentially testing as many as five such groups of bits, and the operand type may require another two.

Some speed improvement likely could have been achieved by streamlining the code of the interpreter further. However, it was decided instead that a different encoding of the SLIM instructions might be better suited to the limitations of the Z80. The technique known as threaded code [Bell73] was tried in the second SLIM interpreter with great success. Threaded code, also used in the p-code interpreter, is described in the next chapter.

6. TC1: SLIM Threaded Code, Version 1

The first SLIM interpreter based on threaded code (explained below) was called TC1. The use of threaded code helped reduce the overhead of instruction decoding. As a result, the initial implementation of the threaded code interpreter was 30 per cent faster than the interpreter using the standard encoding. Fine-tuning further increased the speed of TC1 until it was twice as fast, within 15 per cent of the speed of p-code.

Threaded Code

A program compiled to threaded code consists merely of a sequential list of addresses, each possibly followed by data words. The addresses are those of library routines which perform operations such as addition, multiplication, function calling, and array indexing. The data words are the (constant) arguments of the library routines. Interpretation of threaded code involves the following steps:

1. fetch the address which is in the word pointed to by the program counter (PC) of the threaded code machine

2. increment the PC

3. jump to the library routine at the address just fetched

4. jump back to step 1.

The first three steps form the <u>inner loop</u> of the interpreter and the jump of step 4 is placed at the end of each library routine. On the PDP-11, the inner loop is particularly simple; it requires only a single instruction:

```
NEXT:    JMP     @(R)+
```

where R is the program counter of the threaded code machine.

Following an address of a library routine in the threaded code may be data words. Before it jumps back to 'Next', the routine fetches these values and increments the program counter.

<u>UCSD p-code</u>

The p-code used in the UCSD p-System is a variation of threaded code. The principal difference is that a p-code program consists of a list of offsets rather than addresses. These offsets are used in looking up in a table the addresses of the library routines. The advantage of this scheme is that the library routines may be modified without the need for recompiling all Pascal programs; only the table changes, not the threaded code. Since a p-code opcode is not a machine address but rather an index into a relatively small table, opcodes were defined to be single bytes, allowing 256 library routines.

A further difference between threaded code and the p-code variation is that a number of opcodes have an implicit data value. For example, the first 128 opcodes push a small integer constant (the opcode value itself) onto the p-machine stack, thereby avoiding the need for an explicit data value for this very common operation. Because of this technique, p-code programs are very compact and they are probably faster than they would otherwise be.

Threaded SLIM

Reducing the overhead of instruction decoding required overcoming two bottlenecks: the extraction of groups of bits within a word, and the sequential testing of several values. The first problem was solved by using only whole bytes or words for both opcodes and operands, the second by performing a multi-way jump based on a single value.

SLIM instructions can be divided into two classes, here called A and B. Class A instructions take an operand and have the form:

```
<opcode> <operand>
```

Class B instructions do not take an operand and so have the simpler form:

```
<opcode>
```

The 'LIEn' instruction is an example from class A and 'R' is one from class B. To make it easier to encode SLIM using threaded code, class A instructions were split into two separate instructions, one to load the modified operand into a temporary work register, the W-register, and the other to use the value in the W-register. For example, the 'LIEn' instruction is replaced by 'WIEn' and 'LW'. (By coincidence, this idea was conceived independently by another group [Macl82] at about the same time.) The new W-register instructions ('Wn', 'WIn', 'WCn', 'WICn', 'WEn', 'WIEn', 'WGn', 'WIGn', 'WH', and 'WIH') are class C, and the instructions which take the W-register as the operand form class D.

The instructions in classes B and D have no explicit operands. Of the class C instructions, all but 'WH' and 'WIH' have a raw operand. To simplify decoding, an entire byte was used for each opcode. The size of SLIM programs was reduced by providing class C instructions both for raw operands which can be represented in one byte and those which need a word. In this way, much of the information that the earlier interpreter obtained by extracting and testing instruction fields (namely, whether the raw operand was a byte or a word, the register by which the operand was to be modified, and whether indirection was to be performed) was instead implicitly encoded in the opcode. Decoding was thus reduced to a simple process of jumping to one of a number of library routines, each of which knew the form of the expected operand.

This version of threaded SLIM code was quite similar to p-code. A major difference between them was that not all of the 256 possible values of an opcode byte were defined for SLIM. A SLIM opcode still served as an offset into a table, but instead of that table containing the list of addresses of library routines, the table contained a list of jumps to those addresses. Putting jump instructions in the table allowed the inner loop to be shorter than it would have been otherwise.

Inner Loop

The size of the inner loop is very important to the overall speed of an interpreter. It was not possible to implement the inner loop on the Z80 as a single instruction, but it was nonetheless quite short:

```
NEXT:   LD    A,(BC)      ; Fetch the opcode byte.
        INC   BC          ; Increment SLIM C-register.
```

```
          LD    H,LRPAGE   ; Calculate location in table
          LD    L,A        ;   of jump to library routine.
          JP    (HL)       ; Jump to that jump.
```

After much experimentation, the BC register pair was selected as the threaded code program counter (the SLIM C-register). The table of jumps indexed by the opcode byte was located on a page (256-byte) boundary so that the address of the required jump could be formed in HL by concatenating the page number with the opcode, instead of adding the opcode to the starting address of the table.

This inner loop uses 28 machine cycles plus 10 for the jump in the table plus a further 10 for the jump at the end of each library routine. Forty-eight machine cycles translate into 23 microseconds for a Z80 with a 2.1 MHz clock.

Improving Size and Speed

When programs were encoded using the threaded code described so far, they were about 30 per cent faster than when they were encoded conventionally. However, they were also about twice as large.

A Z80 jump instruction is three bytes long and so, using the opcode byte as an offset into the jump table, there could be at most 86 library routines (86 = Ceiling (256 div 3)). There are 10 instructions in class B (not counting 'O', 'L$', and 'S$'), 18 in class C, and 31 in class D. Initially only the 59 library routines for these instructions were defined. Soon after, SLIM programs were made both smaller and faster by defining the 27 remaining possible library routines.

Library routines were defined for the most common pairs of opcodes and operands; in other words, particular class A instructions were re-introduced and given opcodes. The SLIM code produced by the Compiler was analyzed so that the opcodes could be allocated to the most frequently used class A instructions. A total of over 10,000 instructions were collected from a number of BCPL programs, including the Compiler itself, the CHEF editor, and the BCPL/Z80 run-time system. It was found, for example, that 10 per cent of all instructions were 'LIEb' (with 'b' between -8 and 247), and that 'CIGb' (with 'b' between 0 and 255) made up another 7.5 per cent. The 27 class A instructions which eventually were chosen accounted for 82 per cent of all instructions. Allocating an opcode to each, thereby saving a byte for each use, reduced the size of threaded code programs to within 15 per cent of the conventional encoding. These changes, plus some fine-tuning of the interpreter code, also brought the speed to within 20 per cent of p-code.

Static vs Dynamic Frequencies

The 27 class A instructions that were allocated opcodes were chosen because they were the most common opcode/operand pairs in the programs analyzed. The number of times an instruction appears in a piece of code does not necessarily reflect how often it will be executed, however. As an experiment, the TC1 interpreter was modified to count the number of times each SLIM instruction was executed. Three programs were measured: the Compiler, CHEF, and DORIS. It was discovered that the instructions that were dynamically most frequent were usually those that were also statically most frequent, although there were a few exceptions.

A large number of 'L@n' instructions were counted but they were not executed very frequently. The 'L@n' instruction is most often used in two contexts: initializing the global cells associated with global procedures, and passing format strings to 'WriteF'. The programs analyzed had many global procedures but the global cells are initialized only once, and the programs did not make much use of 'WriteF'.

Also common to the three tests were disproportionately large numbers of executions of the '=H' and '>H' instructions. It was found that these instructions were used in both 'RdCh' and 'WrCh' for file streams. All three programs used these procedures heavily.

Selecting 27 class A instructions to optimize based on static counts results in smaller object programs. In contrast, the use of dynamic counts results in faster object programs. Except for a few anomalies, however, the two counts gave the same results: 23 of the 27 most often executed class A instructions were also among the 27 most often counted.

Input/Output

Shortly after the threaded code interpreter was working, the BCPL/Z80 operating system was written and it became the new host. The only parts of the system not written in BCPL were a few simple assembly language device drivers and the interpreter itself. These device drivers were made to look like BCPL procedures by using the trap for missing global procedures. When one of these drivers was called, the interpreter intercepted the trap and called the appropriate assembly language procedure. After the driver was finished, it returned to 'Next'.

7. TC2: SLIM Threaded Code, Version 2

Despite the large improvement in speed gained by using threaded code, TC1 did not quite achieve the goal of equalling the performance of the p-code interpreter. The next (and final) interpreter, TC2, is not as radically different from TC1 as TC1 was from its predecessor. However, TC2 is faster than the p-code interpreter and SLIM programs for it are smaller than when the standard encoding is used.

Library Routines

The performance of TC1 (both in speed of interpretation and in size of object programs) was greatly improved by using the 27 empty slots in the jump table for library routines for class A instructions. There was not room, unfortunately, for other common opcode/operand combinations such as 'L%IEb', '<=b', or '=IEb'.

One idea that was considered was to replace the jump table with an array of library routine addresses, as is used in the p-code interpreter. This method would allow 128 or 256 library routines to be defined. The idea was rejected because the inner loop would also be considerably slower.

Eventually, it was realized that one of the jump table entries could be used as an escape. The 'Escape' library routine is very similar to 'Next' except that the byte that follows the escape byte is used as an index into a second jump table. The inner loop overhead for an opcode in the second table ('Next' plus 'Escape' plus three jumps) is 83 machine cycles, or 40 microseconds.

By removing the limit of 86 library routines in TC1, it was possible in TC2 to make a number of improvements in performance. One improvement was to move rarely executed SLIM instructions, such as '~', '==W', and 'X', to the second jump table. The freed slots were then used for common class A instructions.

Another improvement was to define library routines for instructions that frequently have particular raw operands. TC2 has opcodes for such instructions as 'L0', 'L1', 'L-1', 'LIE-3', 'LIE1', 'SE1', '+1', '-1', 'L!0', 'L%0', 'M-1', and 'M-2'. Defining these library routines increased the size of the interpreter somewhat, but it also significantly reduced the size of SLIM programs and increased their speed.

Peephole Optimization

It was necessary once again to count SLIM instructions to determine which should be moved into the second jump table, and which opcode/operand combinations should be allocated slots in the first. A total of over 28,000 instructions were collected this time, all from programs running under BCPL/Z80. Instead of counting the instructions produced directly by the Compiler, however, the Improver (mentioned in chapter 3) first processed the SLIM programs.

The Improver uses the technique of peephole optimization, replacing short sequences of instructions with others which are better (shorter or faster or both). The Improver was inspired by the peephole optimizers of Tanenbaum et al [Tane82] and Sweet and Sandman [Swee82], but it is simpler than either. Because the Compiler performs some of its own peephole optimizations, many of the more sophisticated capabilities of those optimizers were not needed.

Instructions are replaced according to rules in a data file. (Appendix A contains the complete list of rules used by the Improver.) Each rule consists of two parts: a pattern and a replacement. If the pattern part of a rule can be matched against a sequence of instructions, the replacement part of the rule is substituted for them. For example, the following rule:

```
[ Q              => CIG98 D0     ]
```

replaces a 'Q' instruction with a call to global routine 98, which is the routine 'Stop' on BCPL/Z80.

Most characters in a rule stand for themselves. The more interesting rules include pattern variables. The following rule replaces, for example, '+-1' with '-1':

```
[ +-r          => -r           ]
```

The 'r' matches a raw operand: either a character constant or a (possibly signed) integer. The variables 's' and 't' also match raw operands.

Another three variables, 'm', 'n', and 'o', match modified operands: 'H', 'IH', or raw operands possibly preceded by '@', 'I@', 'E', 'IE', 'G', or 'IG'. The next example:

```
[ ~=m Tn       => =m Fn        ]
```

converts a test for inequality to one for equality. Once a variable has matched a sequence of characters, it stands for those characters anywhere it appears later in the pattern or the replacement.

The final variable, 'q', matches a quoted string. The following two rules delete the debugging code that some versions of the BCPL compiler produce:

```
[ $q D@r         => $q          ]
[ @r:D0 D0 Dq    =>             ]
```

TC2 Instruction Set

Before instructions were allocated library routines, they were processed by the Improver. This additional step made the selection of instructions somewhat easier.

One way in which improvement helped was that it reduced the number of different commonly-used instructions. For example, SLIM has six comparison operators ('=W', '~=W', '<W', '<=W', '>W', and '>=W') and two conditional jumps ('TW' and 'FW'). Since almost all comparisons are followed by a jump, and since the objects being compared are usually simple variables or constants, most instruction sequences involving comparisons could be rewritten using only two of the operators: '=W' and '<=W'. The following rules accomplish this task:

```
[ Lm >=n         => Ln <=m      ]
[ <m Fn          => >=m Tn      ]
[ <m Tn          => >=m Fn      ]
[ >m Fn          => <=m Tn      ]
[ >m Tn          => <=m Fn      ]
[ ~=m Fn         => =m Tn       ]
[ ~=m Tn         => =m Fn       ]
```

After a program has been processed by the Improver, there are very few occurrences of the other four comparison operators. Thus, the jumps to their library routines were put into the second jump table.

Closely related was the reduction in the number of commonly-used combinations of opcodes and operands. The following rules, for example:

```
[ Lr +Im          => LIm +r         ]
[ Lr L!Im         => LIm L!r        ]
```

try to ensure that the operands of '+W' and 'L!W' are constants whenever possible.

The Improver also allowed two small changes to be made to the SLIM machine. The SLIM document [Peck83] does not specify the behaviour of 'TW' and 'FW' when the accumulator contains a value other than 0 ('False') or -1 ('True'). For TC2, 'False' is defined to be 0 and 'True' is any non-zero value. This redefinition allows the following rules to be used, speeding up SLIM programs and making them smaller:

```
[ =0 Tm           => Fm             ]
[ =0 Fm           => Tm             ]
```

(Recall that most '~=0' instructions are translated to '=0' by the rules given earlier.) Before the Improver was used, the programs analyzed contained over 200 occurrences of '=0'. After improving, only 7 remained.

The other change was the addition of a pair of instructions to increment and decrement a variable. It is quite common in a

BCPL program to add or subtract one from a variable, and the step size in most 'for' commands is also 1 or -1. These new instructions are put into a program by the Improver:

```
[ LIm +1 Sm      => +:m LIm       ]
[ LIm -1 Sm      => -:m LIm       ]
```

(Note that the variable 'm' appears twice in the pattern part of the rule.) '+:W' and '-:W' do not affect the accumulator and it is necessary to explicitly load the result afterwards so that the same effect as before is achieved. In many cases, however, the result is not needed. These rules catch most of these cases:

```
[ Lm Ln          => Ln            ]
[ Lm @r:Ln       => @r:Ln         ]
[ Lm Cn D0       => Cn D0         ]
```

The first two rules are for the cases when the accumulator is immediately reloaded with a new value; only the second value is needed. In the third rule, the value in the accumulator is unimportant if a procedure is called and no arguments are passed to it.

The following sequence of instructions is typical of the code that occurs at the end of a 'for' command:

```
LIE1 +1 SE1 @2:LIE1 <=IE2 T@1
```

The application of two rules results in the following improved sequence:

```
+:E1 @2:LIE1 <=IE2 T@1
```

Appendix B shows the TC2 instruction set, together with its encoding.

Effectiveness of the Improver

The Improver was useful in designing the TC2 instruction encoding, but it was also intended to be used as an optional step in compiling a program. After a program has been debugged, its size and speed can often be improved by the Improver. For some programs, as much as a 10 per cent improvement in both size and speed can be realized. However, it is an optional step because the degree of improvement is usually smaller than that. Typically, a program is reduced in size by about five per cent and it runs about three per cent faster.

Input/Output

It was stated earlier that TC1 did not quite match the speed of the p-code interpreter. During the development of TC2, it was discovered that this statement was untrue. The benchmark programs used for comparison did run faster on the p-System than they did on BCPL/Z80, but it turned out that this result was due not to the difference in speed of the interpreters. TC1 was, in fact, about 20 per cent faster than p-code. However, input/output on BCPL/Z80 was much slower than on the p-System, and the benchmarks did much I/O.

The redesign of SLIM instruction encodings for TC2 was successful in speeding up the benchmark programs, but BCPL/Z80 was only just able to match the p-System. The last major change to TC2 was to re-implement input/output.

The I/O system of BCPL/Z80 is based on two functions: 'ReadBytes' and 'WriteBytes'. They read and write an arbitrary number of bytes on the current input and output streams, respectively. The other procedures, 'ReadN', 'ReadS', 'WriteN', 'WriteS', 'WriteF', and even 'RdCh' and 'WrCh', are implemented using these functions. 'ReadBytes' and 'WriteBytes', written in BCPL, were in turn implemented using a few lower-level procedures to read and write 512-byte blocks (for disks) or single characters (for the console, the printer, and the modem). These two functions were translated into Z80 assembly language and incorporated into the interpreter. Doing so increased the size of the interpreter by about 700 bytes, but decreased the size of the run-time system by about the same amount. It also greatly sped up input/output.

More speed was gained by also translating 'RdCh' and 'WrCh' and later 'ReadS' and 'WriteS' into assembly language. With these changes, the speed of I/O was approximately double that in TC1. BCPL/Z80 at last surpassed the p-System in the benchmark programs.

8. Results

BCPL/Z80 is now considerably faster than the p-System. In this chapter, the performance of the BCPL/Z80 system (TC2) is compared with others, including the previous version of the system (TC1) and the p-System. Both execution times and program sizes are used in the comparisons.

Benchmarks

Several benchmark programs have been used to compare each new version of BCPL/Z80 with its predecessors and with the p-System. Three of these programs are discussed here.

The first program, Hanoi, is a solution to the Towers of Hanoi problem of six discs. It does not require much calculation, but it produces a prodigious amount of output on the screen. The second program, Ack, computes Ackermann's function with the arguments 3 and 5. It does an enormous amount of work (most of it being function calls), but its only output is the final answer. The last program, Compare, is a line-by-line comparison of two text files. It mainly exercises the system file I/O.

The following table shows the sizes of these programs for several systems:

program	size in words				
	Std	TC1	TC2	UCSD	Sirius
Hanoi	85	85	76	102	136
Ack	105	112	92	133	146
Compare	253	258	231	381	378

The column marked Std shows the program sizes when the standard encoding is used, as in the first SLIM interpreter on the Z80, and the UCSD column is for the p-System (version I.5). The final column is for an implementation of SLIM for the Sirius (Victor 9000), a machine based on the Intel 8088 16-bit processor. In this implementation, SLIM is translated into the assembly language of the processor, not interpreted. Translation was the natural choice for the Sirius because that machine has 128 Kbytes or more of memory, at least twice what is possible on a Z80-based machine.

From the above table, it can be seen that TC2 programs are consistently smaller than others by at least 10 per cent. Not too surprising is that programs on the Sirius are substantially larger than TC2. What is surprising, however, is that the p-System programs are only slightly smaller than those on the Sirius; p-code was designed to be compact.

The table below shows the time taken to run the benchmark programs:

program	time in seconds				
	Std	TC1	TC2	UCSD	Sirius
Hanoi	14	8	6	6	2
Ack	-	55	41	69	5
Compare	-	67	24	53	-

(The only time shown for the standard encoding is for Hanoi. Unfortunately, a copy of the system for the Z80 no longer exists and the other two benchmarks were not used while it did; the program sizes given earlier were obtained from a version of the interpreter running on an Amdahl 470/V8.)

The Hanoi program has been used often during the development of BCPL/Z80 as a benchmark, but it gradually became evident that the screen output procedures were the bottleneck, not the interpreter.

The Ack program shows more clearly the differences in speed of calculation. TC2 is 68 per cent faster than the p-code interpreter and 34 per cent faster than TC1. Note that TC1 was also faster than p-code, by 25 per cent. The Sirius implementation is almost an order of magnitude faster than even TC2; a factor of four is perhaps due to the different hardware (it is a 16-bit machine, not 8-bit, and its clock is 5 MHz, not 2.1 MHz), but the rest must be because SLIM is translated to assembly language rather than interpreted.

The Compare program shows the improvement in the speed of file I/O. TC2 is more than twice as fast as the p-System and almost three times as fast as TC1. TC1 is 26 per cent slower than the p-System.

A quotation in Chapter 2 stated that another implementation of BCPL for the Z80, CINTCODE, was significantly more compact than UCSD Pascal and that it runs faster. The same can be said of BCPL/Z80 and TC2.

9. Conclusions

BCPL/Z80 is a practical program development system running on a microcomputer. It was built around three existing portable programs: the BCPL compiler, the CHEF text editor, and the DORIS text formatter. Because these programs were adopted largely unmodified and thus there was no need to write comparable utilities, attention was focussed instead on the interpreter.

With the TC2 interpeter, SLIM programs are small and they are executed quickly. However, as with any large piece of software, the performance of TC2 could be improved with some fine-tuning. For example, it was stated earlier that, in most cases, comparisons can be rewritten using only two of the six comparison instructions: '=W' and '<=W'. Since 'True' was defined to be any non-zero value, the '=W' instruction is also superfluous in most contexts. The following rules would replace tests for equality with subtractions:

```
[ =m Tn          => -m Fn          ]
[ =m Fn          => -m Tn          ]
```

With these rules, '=W' would be rarely needed and the slots in the first jump table used by variations of this instruction could be used for more common instructions. The saving due to this pair of rules likely would be very small, but other similar improvements are probably possible.

Another improvement would be the elimination of some 'void' (V) instructions. Currently, the addresses encoded in instructions and held in variables are SLIM addresses, i.e. word, rather than byte, addresses. As a result, the Encoder is required to insert 'V' instructions into the code to force alignment on word boundaries. Most of these addresses, however, are for the labels used as targets of 'J@n', 'T@n', and 'F@n' instructions. Since only a very few of these labels are accessible to a BCPL programmer, most need not be aligned on word boundaries and instead could have byte addresses. Eliminating many of the 'V' instructions could save as much as 5 per cent in both size and speed.

A final improvement also affects jump instructions. Target labels of most jumps are only a short distance away from the jumps. Although the target address is encoded in an instruction as an offset relative to the location of the instruction, currently this offset is encoded in a word, not a byte. If offsets were encoded in bytes whenever possible, programs would be about 10 per cent smaller, but about as fast.

A goal throughout the project has been to equal or surpass the popular UCSD p-System in the size of object programs and the speed of execution. It required several attempts, but BCPL/Z80 is now faster and smaller than the p-System.

Bibliography

[Bell73] J R Bell, "Threaded Code", Communications of the ACM, vol 16 #6 (1973 Jun).

[Cowd82] R I Cowderoy and P J L Wallis, "The Transfer of a BCPL Compiler to the Z80 Microcomputer", Software--Practice and Experience, vol 12 pp 235-239 (1982).

[Dyme82] J D Dyment, "A Tutorial Guide to DORIS (A Text Formatting Program)", University of British Columbia, 1982 Dec.

[Eage82] R D Eager et al, "Draft BCPL Standard", University of Kent, 1982 Dec.

[Hayt83] R S Hayter, "The BCPL/Z80 Programming System User's Manual", University of British Columbia, 1983 Jul, (included as appendix C).

[Kild81] G Kildall, "CP/M: A Family of 8- and 16-Bit Operating Systems", Byte, vol 6 #6 (1981 Jun).

[Macl81] M A Maclean and J E L Peck, "CHEF: A Versatile Portable Text Editor", Software--Practice and Experience, vol 11 pp 467-477 (1981).

[Macl82] M A Maclean, private communication to J E L Peck, 1982 May.

[Over80] M Overgaard, "UCSD Pascal: A Portable Software Environment for Small Computers", National Computer Conference, 1980.

[Peck81] J E L Peck and M A Maclean, "The Construction of a Portable Editor", Software--Practice and Experience, vol 11 pp 479-489 (1981).

[Peck83] J E L Peck, "The Essence of Portable Programming" (draft), University of British Columbia, 1983.

[RCP81] Richards Computer Products, "More from the Micro with BCPL CINTCODE", advertizing literature, 1981.

[Rich71] M Richards, "The Portability of the BCPL Compiler", Software--Practice and Experience, vol 1 pp 135-146 (1971).

[Ritc78] D M Ritchie and K Thompson, "The UNIX Time-Sharing System", Bell System Technical Journal, vol 57 #6 part 2 pp 1905-1929 (1978 Jul-Aug).

[Swee82] R Sweet and J Sandman, "Static Analysis of the Mesa Instruction Set", Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems, ACM SIGPLAN Notices, vol 17 #4 (1982 Apr).

[Tane82] A S Tanenbaum, H van Staveren, and J W Stevenson, "Using Peephole Optimization on Intermediate Code", ACM Transactions on Programming Languages and Systems, vol 4 # 1 (1982 Jan).

[Zilo76] Zilog, "Z80-CPU Technical Manual", 1976.

Appendix A: TC2 Code Improvement Rules

```
[ LIm +1 Sm    => +:m LIm      ]
[ LIm -1 Sm    => -:m LIm      ]
[ Lm >=n       => Ln <=m       ]
[ Lr +Im       => LIm +r       ]
[ PLr +Im      => PLIm +r      ]
[ Lr L!Im      => LIm L!r      ]
[ Lr S!Im      => LIm S!r Lr   ]
[ Lm Ln        => Ln           ]
[ Lm @r:Ln     => @r:Ln        ]
[ Lm Cn D0     => Cn D0        ]
[ ~=m Fn       => =m Tn        ]
[ ~=m Tn       => =m Fn        ]
[ =0 Tm        => Fm           ]
[ =0 Fm        => Tm           ]
[ =-1 Tm       => +1 Fm        ]
[ =-1 Fm       => +1 Tm        ]
[ =1 Tm        => -1 Fm        ]
[ =1 Fm        => -1 Tm        ]
[ <m Fn        => >=m Tn       ]
[ <m Tn        => >=m Fn       ]
[ >m Fn        => <=m Tn       ]
[ >m Tn        => <=m Fn       ]
[ +-r          => -r           ]
[ Q            => CIG98 D0     ]
[ ~  Fm        => Tm           ]
[ ~  Tm        => Fm           ]
[ $q D@r       => $q           ]
[ @r:D0 D0 Dq =>               ]
```

Appendix B: TC2 Instruction Encodings

Encoding	Mnemonic	
00	V	
FF	B	
06 xx	Wb	xx in [0..255]
09 yy xx	Wc	xxyy in [-32768..32767]
03 02 yy xx	WCc	xxyy in [-32768..32767]
0C xx	WEb	xx-10 in [-10..245]
03 05 yy xx	WEc	xxyy in [-32768..32767]
03 08 xx	WGb	xx in [0..255]
03 0B xx	WGb2	xx+256 in [256..511]
03 0E yy xx	WGc	xxyy in [-32768..32767]
0F	WH	
12 yy xx	WICc	xxyy in [-32768..32767]
15 xx	WIEb	xx-10 in [-10..245]
03 11 yy xx	WIEc	xxyy in [-32768..32767]
18 xx	WIGb	xx in [0..255]
1B xx	WIGb2	xx+256 in [256..511]
03 14 yy xx	WIGc	xxyy in [-32768..32767]
03 17	WIH	
1E yy xx	CICc	
21 xx	CIGb	
03 1A	CW	
24	R	
27 yy xx	JCc	
03 1D	JW	
2A yy xx	FCc	
03 20	FW	
2D yy xx	TCc	
03 23	TW	
03 26	?I	
30	?S	
03 29	N	
03 2C	Q	
33	L-1	
36	L0	
39	L1	
3C xx	Lb	
3F yy xx	Lc	
42 yy xx	LICc	
45	LIE-5	
48	LIE-4	
4B	LIE-3	
4E	LIE1	
51	LIE2	
54	LIE3	
57	LIE4	
5A	LIE5	

5D xx	LIEb
60 xx	LIGb
63 xx	LIGb2
66	LW
69	L!0
6C xx	L!b
6F xx	L!IEb
72	L!W
75	L%0
78 xx	L%IEb
7B	L%W
03 2F	L:W
03 32	LRS
7E	PL0
81	PL1
84 xx	PLb
87 yy xx	PLc
8A yy xx	PLICc
8D	PLIE-4
90	PLIE-3
93	PLIE1
96	PLIE2
99	PLIE3
9C xx	PLIEb
9F xx	PLIGb
A2 xx	PLIGb2
A5	PLW
A8 yy xx	SCc
AB	SE1
AE	SE2
B1 xx	SEb
B4 xx	SGb
B7 xx	SGb2
03 35	SW
BA	S!0
BD xx	S!b
C0 xx	S!IEb
C3	S!W
C6	S%0
C9 xx	S%IEb
CC	S%W
03 38	S:W
03 3B	SRS
CF	P
03 3E	X
D2	M-2
D5	M-1
D8	MW
03 41	|
03 44	-
DB	+1
DE xx	+b
E1 xx	+IEb
E4	+W

E7 xx	+:Eb
03 47	+:W
EA	-1
ED	-W
03 4A	-:W
03 4D	*W
03 50	/W
03 53	/*W
F0 xx	=b
F3	=W
03 56	~=W
F6 xx	<=b
F9 xx	<=IEb
FC	<=W
03 59	<W
03 5C	>W
03 5F	>=W
03 62	<<W
03 65	>>W
03 68	/\W
03 6B	\/W
03 6E	~
03 71	==W
03 74	~~W

Appendix C: BCPL/Z80 User's Manual

1. The BCPL/Z80 Programming System

BCPL/Z80 is a complete system for the development of BCPL programs. It consists of a number of independent programs: the CHEF Text Editor is used to create and modify both programs and documents; the Compiler translates BCPL programs into the SLIM intermediate assembly language; the Improver can be used to make SLIM programs smaller and faster; the Encoder translates SLIM programs into executable binary code; the Linker merges independently-compiled BCPL sections. One of the more useful utility programs is the DORIS Text Formatter which prepares documents for printing.

In many respects, there is a strong resemblance between BCPL/Z80 and the UCSD p-System. The use of the programs described above is coordinated by an operating system program (called the Shell) modelled after that in the p-System. There is a subsystem called the Filer for the manipulation (moving, renaming, destroying and so on) of files. As well, the BCPL/Z80 disk directory structure is identical to the p-System's, and text and data files generated by either system can be read by the other.

Section 2 introduces the important concepts of devices, volumes, and files. Subsequent sections describe the Shell, the Filer and the other programs in more detail.

It is assumed in this document that the reader is familiar with the language BCPL. Some knowledge of the UCSD p-System might also be helpful. The descriptions below apply to the current version of the system (1983 Jun) for the Exidy Sorcerer computer with 55 Kbytes of RAM and two double-density North Star mini-floppy diskettes.

Parts of the BCPL/Z80 system were borrowed from other authors. The CHEF Text Editor was written by M Maclean and J Peck. The BCPL Compiler is a descendent of one written by M Richards. The DORIS Text Formatter was originally written by D Dyment and later rewritten by J Peck.

2. Devices, Volumes, And Files

A typical BCPL/Z80 computer system includes a keyboard, a screen, and several disk drives. There may also be a printer and a modem. These are known as input/output devices.

Devices

Devices are divided into two classes: block and character. For a block device (the disk drives), information is transferred in blocks of 512 bytes. For a character device (the keyboard, the screen, the printer, and the modem), only one byte at a time is transferred.

Associated with each device is a device number:

dev #	device
0	void
1	screen and keyboard with echo
2	screen and keyboard without echo
3	unused
4	disk drive 1
5	disk drive 2
6	printer
7	unused
8	modem
9	disk drive 3
10	disk drive 4

(The differences between devices 1 and 2 will be explained shortly.) Referring to devices by device numbers is awkward at best. Instead, volume names are used.

Volume Names

A volume name may be up to 7 characters long and it is always followed by a ":". These characters may be letters, numbers, ".", "-", "_", "/", or "\". All lower case letters are automatically shifted to upper case.

Character Volumes

The character devices are known as the volumes "CONSOLE:", "SYSTERM:", "PRINTER:", and "REMOTE:". These volume names may be passed as arguments to 'FindInput' or 'FindOutput' to set up streams to the associated devices.

"CONSOLE:" (device #1) is the keyboard for input and the screen for output. As characters are typed at the keyboard, they are echoed to the screen. In addition, the keyboard is buffered; no characters will be given to a program reading from "CONSOLE:" until a <cr> (ASCII carriage return) is typed. Buffering allows the user to correct typing errors. The last character typed can be erased by typing <bs> (backspace) or <del> (delete), and the entire line is erased when <can> (cancel) is typed. An <etx> (end of text) is translated to 'EndStreamCh'. There is room in the buffer for 99 characters. Whenever a <cr> is written to the screen, a <lf> (line feed) is automatically written also, so that subsequent text appears on the next line. Finally, if a <dle> (data link escape) is written, the next character written is taken to be a count (plus 32) of the number of spaces to be displayed on the screen. This two-character sequence is used by BCPL/Z80 to make text files (described below) smaller.

"SYSTERM:" (device #2) is similar to "CONSOLE:". One difference is that the keyboard does not echo characters typed to the screen. Another is that the keyboard is not buffered; a program will receive characters as they are typed, and it will receive a <nul> (null character) if no character is available. The last difference is that a <lf> is not automatically inserted after a <cr> when one is written to the screen, and the two-character <dle> sequence is not expanded into a number of spaces to be displayed. "SYSTERM:" is not used as often as "CONSOLE:" is, but it is occasionally useful.

"PRINTER:" (device #6) is not yet implemented. It will behave similarly to "SYSTERM:".

"REMOTE:" (device #8) is similar to "SYSTERM:" but is associated with the modem. A program reading from this volume will receive characters as they are received by the modem, or a <nul> if none are available. Characters written are sent by the modem, and <cr> and <dle> are not treated specially.

In addition to these character volumes, there is another called "VOID:". "VOID:" (device #0) ignores characters written to it, and returns 'EndStreamCh' whenever an attempt is made to read from it.

Block Volumes

Volume names are also associated with disks. However, a volume name does not refer to the disk drive itself; it refers to the disk that is in the drive. A disk drive is known by the disk that is mounted in it, and so, it may have different names at different times.

For convenience, there are two special shorthand volume names: "*" and ":". The first refers to the system volume, the disk which was in drive 1 at the time BCPL/Z80 was started up. The other name refers to the default volume. Initially, the default volume is the same as the system volume, but it may be changed by the 'P(refix' command described later in section 4.

Occasionally it is useful to refer to volumes by their corresponding device numbers. The special volume names "#1:", "#2:", and so on refer to those devices. Note that "#4:" refers to whatever volume happens to be in drive 1 at the time.

Files

Block volumes, unlike character volumes, are structured objects. On each block volume there can be a number of files (up to 77). A directory on the volume indicates the names of the files, and the location of the files on the disk, among other things.

Files are rather similar to character devices in the sense that characters may be transferred one at a time. The difference is, of course, that the characters are stored permanently on the disk.

The name of a file, preceded by the name of the volume it is on, can be given to 'FindInput' or 'FindOutput' to set up a stream to the file. If 'FindInput' is used, the file must exist already on the disk. 'FindOutput' creates a new file with the name given. To make the new file permanent, the stream must be closed by calling 'EndWrite'. Failing to do so will cause the file to disappear when the program stops.

'FindOutput' may be given the name of file which already exists. In this case also, a new file is created. If 'EndWrite' is used to close it, the old file is removed; otherwise, the new file disappears and the old one remains intact. Requiring an explicit call to 'EndWrite' protects existing files from being lost should the system crash.

File Names

A file name may be up to 15 characters long. As in volume names, these characters may be letters, numbers, ".", "-", "_", "/", or "\". All lower case letters are automatically shifted to upper case.

File Types

There are two types of BCPL/Z80 files: text and data. (The UCSD p-System has several other file types, but BCPL/Z80 regards these all as data files.) The names of text files have a suffix of ".text" and data files do not.

Data files are the simpler of the two types. The bytes in a data file are exactly those which were written to the file.

To make text files smaller, however, spaces at the beginning of each line of text are stored as the two-character <dle> sequence described earlier. This compression is done automatically as characters are written, and they are expanded again when they are read. So that BCPL/Z80 text files are compatible with those of the p-System, the text in such a file is preceded by a two-block header (1024 bytes) of <nul> characters, and enough <nul>s follow the text to make the file an even number of blocks long. BCPL/Z80 skips these <nul>s when the file is later read by a program.

Data bytes should never be put into a text file. The bytes read from a text file will not be exactly the same as those which were written into it because of the extra <nul>s and the special treatment of spaces and <dle>s. In contrast, text may be put into a data file. However, depending on the text, the data file might be longer than the corresponding text file with its compressed spaces.

File Sizes

Files each occupy some number of contiguous blocks on the disk. Normally, when a file is created by 'FindOutput', the largest unused portion of the disk (the largest hole) is allocated. If desired, the length may be explicitly given by appending "[n]" (where 'n' is the requested number of blocks) to the file name given to 'FindOutput'. For example:

```
FindOutput ("bcpl:work.space[16]")
```

The blocks are taken from the first hole large enough, searching from the start of the disk. (A file size may be given when calling 'FindInput' also, but it is ignored.)

If a file size specification of "[*]" is used instead, either the second biggest hole or one-half of the biggest hole is allocated, whichever is larger.

File Specifications

To summarize, 'FindInput' and 'FindOutput' take a file specification as an argument. The specification is made up of three parts: the volume name, the file name, and the file size.

If the volume name is that of a character volume, the file name and size are ignored. An omitted volume name is assumed to refer to the default volume. If the size is omitted, the size of the largest hole is assumed.

Changing Disks

Whenever 'FindInput' or 'FindOutput' is given a file specification in which the volume name is that of a disk (i.e. it is not a character volume name), BCPL/Z80 searches the disk drives to see if there is a block volume of that name. If the volume cannot be found, the user is given an opportunity to put the disk into a drive. In the example given earlier, if the volume "BCPL:" could not be found, the following message would be displayed on the screen:

```
Put BCPL: in and type <cr> (<esc> to abort).
```

At this point the user should put "BCPL:" in one of the drives and then type <cr>. An <esc> (escape) should be typed instead if the user does not want to put in "BCPL:", perhaps because the volume name was misspelt.

Usually, the user should not change disks unless told to do so. Removing a disk with open files on it from a drive will probably cause data for those files to be lost. Even worse, putting a different disk into that drive will probably result in files on the disk being overwritten.

BCPL/Z80 is sometimes, <u>but</u> <u>not</u> <u>always</u>, able to detect when a disk with open files has been removed. If so, the following message is displayed:

```
Put X: back in and type <cr>.
```

The system will not continue until the user puts the volume "X:" back in.

It is always safe to remove or replace a disk if it has no open files. While in the Shell or in the Filer, there are no open files and any disks may be removed or replaced. It is also usually safe to do so whenever a program asks the user for a file specification, but since this is not always true, it is safer to allow BCPL/Z80 to prompt for new disks.

Limits

Here are collected all the size limits relating to devices, volumes, and files:

1. A volume name may be up to 7 characters long, not including the final ":".

2. A file name may be up to 15 characters long.

3. There may be up to 77 files on a block volume.

4. Up to 8 files may be open simultaneously.

5. There are 350 blocks on a disk but, since the directory occupies the first 10 blocks, a file can be no bigger than 340 blocks long.

6. A text file is always an even number of blocks long. It starts with a two-block header of <nul>s and the text is padded at the end with more <nul>s. Thus, text files are at least 4 blocks long.

7. The "CONSOLE:" keyboard buffer is 99 characters long.

3. The Shell

The Shell is the program which runs automatically when BCPL/Z80 is started. The user may run other programs by typing commands to the Shell. After each program finishes, the Shell is run once again.

Startup

After the computer is turned on (or reset), put the system disk in the left drive and then type:

```
go dc00
```

Within a few seconds, the BCPL/Z80 logo will be displayed on the screen. Within a few more seconds, the Shell will begin running.

Commands

While the Shell is running, it displays a menu of commands at the top of the screen:

```
Shell: C(omp, D(ate, E(dit, F(ile, I(mprove, eN(code, eX(ec?
```

This menu lists most of the commands which may be used. Another menu which lists the remaining commands is displayed when a "?" is typed:

```
Shell: A(ssem, L(ink, R(estart?
```

Any of these commands may be performed by simply typing the corresponding letter, the one shown before the "(". Although these letters are shown in upper case, either upper or lower case may be typed. The commands are described below.

It is safe to change disks while the Shell is running.

In this section, and in the others to follow, a number of examples are given. The input that a user would type is underlined.

A(ssem

The 'A(ssem' command is used to run the Z80 assembler. The Assembler, which is not yet finished and so is not described, will be in the file "*system.assmbler".

C(omp

The 'C(omp' command is used to run the BCPL Compiler. The Compiler, described in section 6, is in the file "*system.compiler". The files "*system.c.parse", "*system.c.trans", and "*system.c.error" are used by the Compiler.

D(ate

The 'D(ate' command is used to set the system date. When the BCPL/Z80 system is started, it displays its current date, usually the date this command was last used. (The BCPL/Z80 does not include a real-time clock, so the date must, unfortunately, be set manually.) It is important to keep this date accurate, since it is used by the system whenever a file is created or modified.

When the command is used, the system displays the current date and then asks for today's. If it is supplied, the date is set. The year, the month, and the day must be entered in that order. Any of them, however, may be omitted and those that are will not change. If present, the year must be between 1901 and 1999, the day must be between 1 and 31, and at least the first three letters (upper or lower case) of the month's name must be given.

Example:

```
Shell: C(omp, D(ate, E(dit, F(ile, I(mprove, eN(code, eX(ec? d

Current date is 1983 Jun 30
New date? jul 2
Current date is 1983 Jul  2

Shell: C(omp, D(ate, E(dit, F(ile, I(mprove, eN(code, eX(ec?
```

E(dit

The 'E(dit' command is used to run the CHEF Text Editor. CHEF, described in section 5, is in the file "*system.editor". The file "*system.e.msg" is used by CHEF.

F(ile

The 'F(ile' command is used to enter the Filer subsystem. The Filer, described in section 4, is part of the Shell, not a separate program.

I(mprove

The 'I(mprove' command is used to run the Improver. The Improver, described in section 7, is in the file "*system.improver".

L(ink

The 'L(ink' command is used to run the Linker. The Linker, described in section 9, is in the file "*system.linker".

eN(code

The 'eN(code' command is used to run the Encoder. The Encoder, described in section 8, is in the file "*system.linker".

R(estart

The 'R(estart' command is used to re-run the program which was most recently run. This command is particularly handy when used after a complicated 'eX(ec' command.

eX(ec

The 'eX(ec' command is used to run programs. The Shell asks which file to run. In addition to the program name, the user may specify the initial input and output streams for the program. These streams are named in a manner similar to that used on UNIX. The input file specification is prefixed by a "<" and the output specification is prefixed by a ">". If either or both streams are not redirected in this way, they are initially set to "CONSOLE:".

When a system program (like the Compiler, for example) is run using one of the above single-letter commands ('C(omp'), the input and output streams are set to "CONSOLE:". The 'eX(ec' command can be used to redirect input or output when running one of these programs, if required. (Output redirection was used to produce the examples in this document.) The file names of the system programs were given with the descriptions of the commands above.

Example:

```
Shell: C(omp, D(ate, E(dit, F(ile, I(mprove, eN(code, eX(ec? x

Execute what file? *hanoi <x:h.in.text >x:h.out.text

Shell: C(omp, D(ate, E(dit, F(ile, I(mprove, eN(code, eX(ec?
```

*system.startup

When the BCPL/Z80 system is started, it checks to see if there is a file called "*system.startup". If so, this program is run before the Shell. It is this StartUp program that displays the BCPL/Z80 logo mentioned earlier. If desired, another program can be renamed "*system.startup" and it will then be run whenever the system is started.

4. The Filer

The Filer is a subsystem useful for manipulating files. It is entered by typing the 'F(ile' Shell command.

The Filer is based on a similar program which is part of the UCSD p-System. The current BCPL/Z80 version is much less powerful, however. It is intended that this deficiency will be corrected soon. In the meantime, although it is inconvenient to do so, the UCSD Filer may be used for those commands not yet implemented. Only those commands implemented so far are described here.

Commands

While the Filer is running, it displays a menu of commands at the top of the screen:

```
Filer: C(hange, L(ist, P(refix, R(emove, T(ransfer, Q(uit?
```

These commands are described below. In the examples, the system volume is assumed to be "BCPLZ80:" and the default volume is "WORK:".

It is safe to change disks while in the Filer.

C(hange

The 'C(hange' command is used to change the name of a file. The Filer first asks which file is to be renamed and then for the new name. When the new name is typed, it is not necessary to give a volume name; it is ignored since the file stays on the same volume.

Example:

```
Filer: C(hange, L(ist, P(refix, R(emove, T(ransfer, Q(uit? c

Change what file? *hanoi
To what? system.startup
BCPLZ80:HANOI changed to SYSTEM.STARTUP.

Filer: C(hange, L(ist, P(refix, R(emove, T(ransfer, Q(uit?
```

L(ist

The 'L(ist' command is used to list the files on a volume. The Filer asks for the name of a volume. The list which is then displayed shows, for each file, its name, the date it was created or last modified, the block number at which it starts, its length in blocks, the number of bytes used in its last block, and its type. (As explained earlier, BCPL/Z80 file types are either "text" or "data", but files created under the p-System may have other types.) Any unused parts of the disk (holes) are also shown in the list together with their starting block number and length. After this list are some statistics: the number of files on the volume, the number of disk blocks used and unused, and the size of the largest hole.

Example:

```
Filer: C(hange, L(ist, P(refix, R(emove, T(ransfer, Q(uit? l

List the directory of what volume? bcplz80:

BCPLZ80:           1983 Jul 25
SYSTEM.C.ERROR     1982 Jun  4   10   10  512  text
HEADER.B.TEXT      1983 May 19   20   18  512  text
SYSTEM.SHELL       1983 Jun  3   38   10  312  data
SYSTEM.E.MSG       1982 Oct 18   48   10  512  text
SYSTEM.STARTUP     1983 Jul 25   58    4   80  data
SYSTEM.RUNTIME0    1983 Jul 25   62   32  512  data
SYSTEM.ENCODER     1983 May 26   94   16   20  data
SYSTEM.LINKER      1983 May 26  110    3   12  data
SYSTEM.IMPROVER    1983 Jun 12  113    9  190  data
SYSTEM.EDITOR      1983 Jun  3  122   49  308  data
SYSTEM.COMPILER    1983 May 27  171    6  202  data
SYSTEM.C.PARSE     1983 May 27  177   19   60  data
SYSTEM.C.TRANS     1983 May 27  196   25  386  data
< unused >                      221  129
13 files, using 211 blocks.
129 blocks unused; 129 in largest hole.

Filer: C(hange, L(ist, P(refix, R(emove, T(ransfer, Q(uit?
```

P(refix

The 'P(refix' command is used to set the name of the default volume. If a file specification given to 'FindInput' or 'FindOutput' does not include a volume or has a volume name of ":", it is considered to refer to the default volume. Initially, the default volume is the same as the system volume.

Example:

```
Filer: C(hange, L(ist, P(refix, R(emove, T(ransfer, Q(uit? p

Prefix names with what volume? work:
New prefix is WORK:.

Filer: C(hange, L(ist, P(refix, R(emove, T(ransfer, Q(uit?
```

R(emove

The 'R(emove' command is used to destroy a file. The Filer asks which file is to be removed.

Example:

```
Filer: C(hange, L(ist, P(refix, R(emove, T(ransfer, Q(uit? r

Remove what file? hanoi.s
WORK:HANOI.S removed.

Filer: C(hange, L(ist, P(refix, R(emove, T(ransfer, Q(uit?
```

T(ransfer

The 'T(ransfer' command is used to make a copy of a file. The Filer first asks which file is to be transferred and then the name of the copy. The copy may go to the same volume or to a different one, and it may have the same name or a different one.

A text file may be displayed on the screen by tranferring it to "CONSOLE:", and keyboard input (until an <etx> is typed) may be put into a file by transferring to it from "CONSOLE:".

Example:

```
Filer: C(hange, L(ist, P(refix, R(emove, T(ransfer, Q(uit? t

Transfer what file? hanoi
To where? *system.startup[4]
WORK:HANOI transferred to BCPLZ80:SYSTEM.STARTUP.

Filer: C(hange, L(ist, P(refix, R(emove, T(ransfer, Q(uit?
```

Q(uit

The 'Q(uit' command is used to return to the Shell.

5. The CHEF Text Editor

CHEF is a general-purpose text editor suitable for the creation and modification of both programs and documents. The command language is described in _The CHEF Editor_.

CHEF is run by typing the 'E(dit' Shell command. It then prompts for commands.

Only one command has not been implemented: 'QS'. As a result, it is not possible to execute Shell commands from CHEF, nor is it possible to temporarily leave CHEF and resume it later.

The BCPL/Z80 version of CHEF has the screen mode 'A' command. In screen mode a number of special keys are used:

key	function
<g-UA>	cursor up
<g-DA>	cursor down
<g-LA>	cursor left
<g-RA>	cursor right
<g-i>	begin insert mode
<g-e>	end insert mode
<g-d>	delete character
<gs-I>	insert line
<gs-D>	delete line
<g-1>	exit
<g-2>	renew
<g-3>	merge
<g-4>	mark
<g-5>	save
<g-6>	inject

where <g-UA> means that the 'up arrow' key on the keypad is typed while the 'graphic' key is held down, <g-i> means that "i" is typed with 'graphic' held, <gs-I> means that "I" is typed with both 'graphic' and 'shift' held. For the last 6 functions, the number keys on the top row should be used, not those on the keypad. The prompt at the top of the screen while in alter mode is a reminder of which keys are associated with these 6 functions:

```
>>g1-Exit,g2-Renew,g3-Merge,g4-Mark,g5-Save,g6-Inject<<
```

The system disk should never be removed from the drive while CHEF is running.

Example:

```
Shell: C(omp, D(ate, E(dit, F(ile, I(mprove, eN(code, eX(ec? e

BCPL/Z80 Editor (CHEF)

Enter H for help (Q for quit)
>ef work:hanoi.b.text
488

...editing commands

>wf.
503
>q

Shell: C(omp, D(ate, E(dit, F(ile, I(mprove, eN(code, eX(ec?
```

6. The BCPL Compiler

The Compiler translates BCPL programs into SLIM assembly language. The language accepted by the Compiler corresponds quite closely to that defined in the Draft BCPL Standard (1982 Dec). The standard header file can be included in a program by means of the following get directive:

```
get "**header.b.text"
```

The procedures declared in the standard header are described in more detail in The BCPL/Z80 Run-Time Library.

The Compiler is run by typing the 'C(omp' Shell command. It then prompts for the name of the file containing the BCPL program, and for the name of a file to write the SLIM text. Usually, the BCPL file has a suffix of ".b.text" and the SLIM file has one of ".s".

A BCPL program is compiled in two passes. In the first, a parse tree is constructed corresponding to the program. A "." is printed on the screen for each line of the program as it is processed. Procedure names are also printed as their definitions are reached. Also, as each get is encountered, a message is printed. In the second pass, the parse tree is used to generate an equivalent SLIM program. Again, procedure names are printed, and a "." is printed for each line of SLIM generated. The file to be compiled may contain any number of BCPL sections.

Should the Compiler detect an error during either pass, an error message will be displayed. The user then may type either 'C(ontinue' to allow the Compiler to resume, or 'Q(uit' to abort the compilation and return to the Shell.

The system disk should never be removed from the drive while the Compiler is running.

Example:

```
Shell: C(omp, D(ate, E(dit, F(ile, I(mprove, eN(code, eX(ec? c

BCPL/Z80 Compiler
Compile what file?  work:hanoi.b.text
Into what file?  work:hanoi.s
...
get *header.b.text
................................................................
................................................................
................................................................
.................................
START...........
HANOI.......
Tree size = 2304.
.
START........
HANOI..........
No errors detected.

Shell: C(omp, D(ate, E(dit, F(ile, I(mprove, eN(code, eX(ec?
```

7. The Improver

A SLIM assembly language program can usually be made both smaller and faster by the Improver. It uses the technique of peephole optimization to replace short sequences of instructions by equivalent, but better, sequences. The degree of improvement possible depends on the SLIM program, but typically the improved program is 10% smaller and 5% faster. Because these gains are rather modest, it is probably only worthwhile to improve debugged and often-used programs.

The Improver is run by typing the 'I(mprove' Shell command. It then prompts for the name of the file containing the SLIM program to be improved, and for the name of a file to write the improved program. Usually, the SLIM file has a suffix of ".s" and the improved SLIM file has one of ".is".

Procedure names are printed on the screen as they are processed. Also, a "." is printed for each line of SLIM text written. The file to be improved may contain any number of SLIM sections.

It is safe to remove any disk from its drive when a file name is requested.

Example:

```
Shell: C(omp, D(ate, E(dit, F(ile, I(mprove, eN(code, eX(ec? i

BCPL/Z80 Improver
Improve what file?  work:hanoi.s
Into what file?  work:hanoi.is
.
START.........
HANOI...............
78 instructions read and 78 written.
1 successful pattern matches.

Shell: C(omp, D(ate, E(dit, F(ile, I(mprove, eN(code, eX(ec?
```

8. The Encoder

A SLIM assembly language program must be encoded as binary object code bytes by the Encoder before it can be run.

The Encoder is run by typing the 'eN(code' Shell command. It then prompts for the name of the file containing the SLIM program to be encoded, and for the name of a file to write the binary code. Usually, the assembly language file has a suffix of ".s" or ".is" and the binary file does not have a suffix.

Procedure names are printed on the screen as they are processed. Also, a "." is printed for each line of SLIM text read. The file to be encoded may contain any number of SLIM sections.

It is safe to remove any disk from its drive when a file name is requested.

Example:

```
Shell: C(omp, D(ate, E(dit, F(ile, I(mprove, eN(code, eX(ec? e

BCPL/Z80 Encoder
Encode what file?  work:hanoi.is
Into what file?  work:hanoi
..
START.........
HANOI.............
No errors detected.
91 words of SLIM code and 5 relocations.

.

Shell: C(omp, D(ate, E(dit, F(ile, I(mprove, eN(code, eX(ec?
```

9. The Linker

If a BCPL program has been organized as a number of separately-compiled sections in separate files, the sections must be linked together before the program can be run. It is not, however, necessary to link the sections if they are already in the same code file.

The Linker is run by typing the 'L(ink' Shell command. It then prompts for the names of the files containing the SLIM sections to be linked, and for the name of a file to write the binary code. Usually, none of the files has a suffix.

It is safe to remove any disk from its drive when a file name is requested.

Example:

```
Shell: C(omp, D(ate, E(dit, F(ile, I(mprove, eN(code, eX(ec? l

BCPL/Z80 Linker


Input code file?  doris:d1
Input code file?  doris:d2
Input code file?  doris:d3
Input code file?  doris:d4
Input code file?
4567 words of SLIM code and 224 relocations.
Output code file?  doris:doris

Shell: C(omp, D(ate, E(dit, F(ile, I(mprove, eN(code, eX(ec?
```

10. The DORIS Text Formatter

DORIS is a program which formats documents in preparation for printing them. Although it is simple to use, it is also quite powerful. The use of DORIS is described in *A Tutorial Guide To DORIS (A Text Formatting Program)*.

DORIS is run by typing the 'eX(ec' Shell command and is in the file "doris:doris". It then prompts for the name of the file containing the document to be formatted, and for the name of a file to write the formatted document. It also prompts for the name of a file to be used in constructing an index, but it is only necessary to type a name if the document contains '.fil' commands. Usually, the document file has a suffix of ".di.text", the formatted file has one of ".do.text", and the index file has one of ".dx.text".

It is safe to remove any disk from its drive when a file name is requested.

Example:

```
Shell: C(omp, D(ate, E(dit, F(ile, I(mprove, eN(code, eX(ec? x

Execute what file? doris:doris

BCPL/Z80 Text Formatter (DORIS)

Process what file? work:thesis.di.text
Into what file? work:thesis.do.text
Use what index file?

Shell: C(omp, D(ate, E(dit, F(ile, I(mprove, eN(code, eX(ec?
```