The BCODE System

S Ma & R. Agarwal
Dept. of Computer Science,
University of British Columbia
Vancouver, B.C. V6T 1W5
CANADA.

TM-24

Table of Contents

Project Documentation
Computer Science
submitted by R Agarwal and S Ma

TM-24
DEPT. OF COMPUTER SCIENCE
UBC APRIL, 1978.

0. Introduction

In an attempt to achieve program portability, the processor for the language BCPL [R] was designed to generate code for a virtual machine. The code for this machine was termed OCODE[1]. The structure of this language, though well-suited as a target language for the translation of BCPL, was nevertheless thought to be more complex than necessary. Hence, the Intcode machine appeared, which is both easy to understand and for which an interpreter is readily written for most real machines available today. Since an OCODE to Intcode translator exists, it is clear that in theory any BCPL program (after suitable translations) may be made to execute on any machine where an Intcode interpreter exists.

Several problems arise when one actually tries to transport programs via the above method. Within the environment of minicomputers, one major problem is that of insufficient memory space. This lack arises not only because most minicomputers can address only 32K 16-bit words directly but also because many programs being transported to a minicomputer are originally written on a large machine (such as the IBM 370) where memory space is virtually infinite.

Creating segment overlays to alleviate the space problem is possible to a certain extent but is not very easily done since BCPL lacks language primitives to define (and restrict) interactions

---

[1] The form of OCODE has been modified at UBC to remove some of its machine dependant aspects. The new (machine independent) form is called MCODE.

[2] Like MCODE, Intcode has been modified at UBC to remove some of its machine dependent characteristics. The resulting language is called Minicode.

between different overlays. However, one promising approach is to increase the density of code generated by the overall translation process of BCPL. The most readily apparent way to achieve this is to write an optimizing compiler for every machine; but this detracts from the goals of compiler and language portability.

A "middle-ground" solution which maintains portability is to delete the OCODE to Intcode translation step and to interpret the OCODE directly. Since OCODE describes a more powerful machine than Intcode, the code generated for it is often much smaller than the equivalent Intcode. Described herein is the implementation of a complete OCODE interpreter on the HP21MX as well as a code generator used to transfer the OCODE from the BCPL compiler to the Hewlett-Packard minicomputer. Also described is the design for a machine independent linker/loader which has not been implemented. Currently, HP assembler source is used for the transfer of programs.

0.1. The HP21MX Minicomputer Implementation

Since the HP21MX minicomputer is a microprogrammed machine, it is natural that the OCODE machine's instruction repertoire be microprogrammed. However, examination of the machine's assembler language description[3] reveals taht it was probably never designed to be used as an assembler per se. The language is both very unreadable in form and also has several opcodes which possess semantic information concerning theBCPL source program which has little to do with a general purpose assembler[4]. Similarly, some

---

[3] No complete description of the OCODE machine is known to exist. An almost complete description is to be found in [W].

commands exist which are best viewed as simple macros; that is, they are definable as a fixed sequence of other OCODE commands. There also exist a class of instructions for which no convenient execution strategy exists. These tend to point out shortcomings in OCODE if viewed as an actual machine; they also serve to strengthen out earlier belief about the original design intentions of OCODE.

Although none of the above criticisms prevent the current description of OCODE from defining a machine unambiguously, it is nonetheless highly desirable to have a symbolic language which defines the machine in a convenient manner. To this end we have devised the BCODE machine (which is described in section 1.) The assembler language for this machine follows the normal assembler structure more closely. The mapping between the assembly language and its machine code is also defined (no such translation scheme exists for OCODE.) We also note that the machine code encoding for BCODE closely resemble those of Intcode [P].

We should note that the microprogrammability of the 21MX implies that the BCODE interpreter for it transforms the machine into a true BCODE machine (with the execption of a few system calls which utilize the standard instruction set to access the resources management facilities provide by RTE/II.)

---

[4] An example is the SAVE, STACK, and STORE operators of OCODE which have the same meaning within the OCODE environment but which carry different meanings as to the point of origin within the BCPL source.

1. The BCODE Machine.

BCODE is an abstract machine intended for efficient execution -- in both time and space -- of BCPL programs. As such, it has some features that are unusual when compared with "normal" machines like the IBM 370.

(1) The word size is unspecified; in theory, BCODE programs work with any word size. This concept is the key one in making BCODE so portable.

(2) The machine is stack oriented, allowing arithmetic expressions and the like to translate almost one-for-one into machine operations. In addition, the zero-address instructions made possible by stack architecture greatly reduce program size.

(3) On a typical machine, a compiler for BCPL usually has a hard time generating code for procedure calls, parameter transmission, and procedure exits. This is not so in BCODE, where special instructions exist to perform these operations.

These features, in combination with others, should make it clear why we think BCODE is an "ideal" machine for BCPL.

1.1 Basic Machine Architecture.

1.1.1 Storage.

Internal storage for the BCODE machine consists of a sequence of words with addresses increasing by one. The number W of bits in a word -- the word size -- is deliberately unspecified; however, a word should be large enough to hold any address.

As a result of its BCPL orientation, the basic BCODE machine has at least three storage areas:

(1) The global vector. This area (of unspecified size) is a contiguous sequence of cells common to all procedures in a BCPL program. The first 100 of these cells are reserved for the operating system; the remainder can be used for inter-procedure communication in a BCPL program.

(2) The program vector. This area holds all the code and static data for an executing BCPL program.

(3) The stack. This area holds all temporaries, dynamically-allocated variables, and inter-procedure linkages. The stack grows in the direction of increasing addresses, so a push operation always increments the stack top's address by one.

In addition, more sophisticated systems may have a fourth area of

storage:

(4) The heap. This area contains all storage that can be allocated and released by calls to GETSPACE() and FREESPACE(). Allocation and de-allocation is completely under programmer control and is independent of stack movement. Standard BCPL does not support the heap concept.

The locations of the above four storage areas are left unspecified.

1.1.2 Internal Registers.

The BCODE machine has five internal registers, each a word in size. Unlike most machines, BCODE's stack architecture allows these registers to operate implicitly; they are not, in general, under direct programmer control. The five registers are:

(1) G -- This contains the address of the fist cell in the global vector. The G register usually remains unchanged throughout the execution of a BCPL program.

(2) C -- This register (the program counter) always contains the address of the next instruction to be executed.

(3) S -- This register always points to the top cell on the stack. All load instructions increase S by 1 before placing a value in the resulting address.

(4) P -- This is the stack frame pointer. Every procedure allocates (on entry) its own frame for storage of dynamic variables and releases the frame on exit; nested procedures stack frames on top of each other. At any instant, a program can access only those dynamic variables in the current (topmost) stack frame.

(5) T -- This is a temporary register used mainly in returning a function's result to the calling procedure. (See the BCODE instructions FNRN, PUSHT, MOD, and MODST.)

1.1.3 Stack Operation.

The BCODE machine is stack oriented, so all operations are done in postfix order: each machine function takes its arguments from the top of the stack and replaces them with the result.

One of the most important and most neglected areas of machine design is the architecture of procedure calling sequences. All too often, this area is given only token attention,

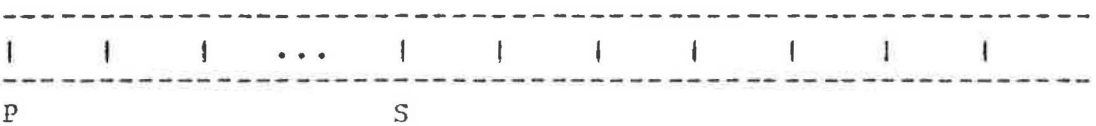
with the result that procedure calls seemed to be added only as an afterthought. This consequently places a tiresome burden on the programmer, who must hunt around for clever dodges around a machine's restrictions.

BCODE's design tries to remedy this. Because of its BCPL orientation, facilities for procedure handling are integrated with the rest of the instruction set. As a result, the hardware performs the bookeeping -- not the programmer.

The facilities BCODE provides is best illustrated by example. Suppose the current stack fram is 40 words long, and we want to execute the following BCPL call:

```
SUB(P1, P2, P3)
```

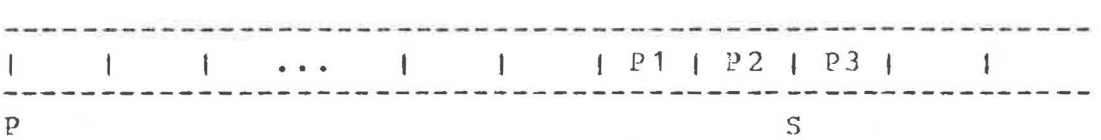
Suppose the stack looks like the following:



where P is the stack frame pointer, and S is the stack top pointer. (We shall call this the "initial configuration".) To process the call, we must first load the parameters:

```
SETS 42          ;Reserve current stack frame
LIP P1           ;Load first parameter
LIP P2           ;Load second parameter
LIP P3           ;Load third parameter
```

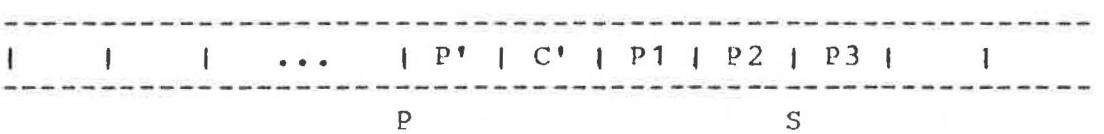
At this point, the stack looks like:



We now call the subroutine:

```
LI SUB           ;Get address of subroutine
CALL 40          ;Enter subroutine
```

We are now at the subroutine's entry point. The stack looks like:



where P' is the old frame pointer and C' is the return address. SUB can now run. When it wishes to return, it issues the instruction

```
RTPN             ;Return from subroutine
```

and the stack will be reset to its initial configuration. If SUB is a function, it should leave the result on the top of the stack and issue

```
        FNRN            ;Return from function
```

This places the function result into the T register and returns like RTRN. The calling program must then issue

```
        PUSHT           ;Get function result
```

to retrieve the function's value.

1.2. Notation.

We will be using the following notation in the rest of this chapter. In general, the notation follows BCPL conventions.

The letters G, C, P, S, and T represent internal registers as described in section 1.1.2. Other symbols are as follows:

Symbol	Description
adr	An expression having an integer result. This is interpreted as an address.
!adr	Contents of the location whose address is given by adr.
@adr	Address of the location given by adr.
adr1 ! adr2	Equivalent to !(adr1 + adr2).

1.3. Instruction Format.

BCODE instructions come in two sizes, single word and double word. The two sizes have the same general format, and are generally interchangable in that any given instruction may use either size; however, the single word size has a restricted addressing range, and can be used only when the operand is small. The two instruction sizes look like the following:

```
Single word:    Opppmmmddddddddd

Double word:    1pppmmmddddddddd dddddddddddddddd
```

The two formats differ only in the state of the most significant bit: a single word instruction has this bit set to zero, while a double word instruction has it set to one. Other fields in an instruction are defined as follows:

ppp This 3-bit field defines the operation code (called OP).

mmm This 3-bit field (called MOD) defines the modifi-

cations to be performed on the operand. (See Section 1.4 for the format of this field.)

d..d This field (called D) is the operand. If the word size is W bits, the operand field for a single word instruction is W-7 bits, while that for a double word instruction is 2W-7 bits. Note that D is a signed, two's complement number.

1.4. Instruction Execution.

The BCODE machine goes through three distinct phases during the execution of a single instruction:

(1) First, it fetches the current instruction being pointed to by the program counter (C) and decodes it. The program counter is incremented past the instruction.

(2) The machine then computes the effective operand (EFF) from the instruction's MOD and D fields, as described below.

(3) Finally, the machine performs the indicated operation, using the the effective operand (EFF) and the stack.

BCODE repeats this entire process for each instruction; it halts only upon encountering a FINISH.

As indicated above, the calculation of the effective operand (EFF) uses both the modifiers (MOD) and the raw operand (D). This calculation proceeds in two steps. First, if

```
MOD = x00   then EFF := D   (no modification)

MOD = x01   then EFF := G+D   (global relative)

MOD = x10   then EFF := P+D   (stack relative)

MOD = x11   then EFF := C+D   (program relative)
```

(Note that, in the above, the program counter (C) points past the instruction.) Then, if

```
MOD = 0xx   then EFF := EFF   (direct)

MOD = 1xx   then EFF := !EFF   (indirect)
```

EFF is now ready for use by the instruction.

1.5. Instruction Set.

This section defines the BCODE machine's instruction set. While each instruction has its own assembler-like mnemonic, this

symbol is used for desciptive purposes only; there is no "BCODE assembly language".

For each BCODE instruction, the operation code (OP) identifies the action to take upon the effective operand (EFF). EFF is always computed (see Section 1.3) before any action occurrs.

1.5.0 Load (OP = 0)

Push the effective operand onto the stack.

```
Operation:      S := S + 1      (S is the stack top pointer)
                !S := EFF

Examples:       L 29     (Load constant 29)
                LIP 7    (Load contents of P+7)
                LG 55    (Load address of global 55)
```

1.5.1 Store (OP = 1)

Pop the top element off the stack and store it into the effective operand.

```
Operation:      !EFF := !S
                S := S - 1

Examples:       S 2476   (Store into location 2476)
                SP 9     (Store into location P+9)
                SIG 123  (Store indirect thru global 123)
```

1.5.2 Set stack pointer (OP = 2)

Set the stack top pointer to a given number of cells above P. This effectively sets the current stack frame size.

```
Operation:      S := P + EFF

Examples:       SETS 27          (Set frame size to 27)
```

Note that SETS 0 sets S equal to P.

1.5.3 Unconditional jump (OP = 3)

Jump unconditionally to the location given by the effective operand.

```
Operation:      C := EFF

Example:        JUMP 5276        (Jump to location 5276)
```

1.5.4 Jump if false (OP = 4)

Jump if the stack top contains false. Pop this element off after testing. (Note that the constant false in BCODE is zero.)

```
Operation:      if  !S = 0  then  C := EFF
                S := S - 1

Example:        JF 9700           (Jump if false to location 9700)
```

1.5.5. Jump if true (OP = 5)

Jump if the stack top does not contain false. Pop the element off after testing.

```
Operation:      if  !S ≠ 0  then  C := EFF

Example:        JT 9700           (Jump if true to location 9700)
```

1.5.6. Procedure call (OP = 6)

Call the procedure whose address is on the top of the stack. EFF gives the curent stack frame size.

```
Operation:      temp := P + EFF  (address of new frame)
                temp!0 := P      (save old frame pointer)
                temp!1 := C      (save return address)
                P := temp        (set new stack frame)
                C := !S          (transfer to procedure)
                S := S - 1       (pop procedure address)

Example:        CALL 40    (call procedure with frame size 40)
```

1.5.7. Execute (OP = 7)

Execute the operation determined by the effective operand (EFF). This instruction is actually a whole class of zero-address instructions that operate unpon the stack.

The following tables list all execute operations currently defined. Note that no floating point operations are included.

Table 1-1 Unary Operators

EFF	Mnemonic	Description
1	ABS	!S := abs !S
2	NEG	!S := neg !S
3	NOT	!S := not !S
4	RV	!S := ! (!S)
5	TRUE	S := S + 1; !S := -1
6	FALSE	S := S + 1; !S := 0

Table 1-2 Binary Operators

EFF	Mnemonic	Description
7	MULT	Binary operator: S := S - 1; S!0 := S!0 * S!1
8	DIV	As above, for /
9	REM	As above, for rem
10	PLUS	As above, for +
11	MINUS	As above, for -
12	EQ	As above, for =
13	NE	As above, for ≠
14	LS	As above, for <
15	GR	As above, for >
16	GE	As above, for ≥
17	LE	As above, for ≤
18	LSHIFT	As above, for >>
19	RSHIFT	As above, for <<
20	LOGAND	As above, for &
21	LOGOR	As above, for |
22	EQV	As above, for eqv
23	NEQV	As above, for neqv

Table 1-3 Transfer Operations

EFF	Mnemonic	Description
24	FINISH	Halt BCODE machine.
25	RTRN	Return from subroutine: S := P - 1 (reset stack top) C := P!1 (return address) P := P!0 (old stack frame)
26	FNRN	Return from function: T := !S (set return value) S := P - 1) C := P!1) Like RTRN. P := P!0)
27	GOTO	Indirect jump (unconditional): C := !S S := S - 1
28	SWITCHB	This instruction performs a multi-way branch depending on the value on top of the stack. The instruction format is: SWITCHB n D C(1) L(1) : : C(n) L(n) where n is the number of cases to choose from, and C(1) L(1) ... C(n) L(n) are the n corresponding value-label pairs. D is the default label. The branch proceeds as follows: If value C(i) equals the value on the stack top, control goes to address L(i). If none of the values match, control goes to address D. Note that, since a binary search is used, the values C(1) ... C(n) must be in ascending order.
29	SWITCHX	This instruction performs a multi-way branch using an indexed jump. The format of the instruction is: SWITCHX min max D L(min) ... L(max) where min and max are the minimum and maximum values, respectively, that the following jump table can handle. L(min) ... L(max) are the max-min+1 labels defining the jump table. D is the default label. The instruction works as follows: Suppose the contents of the stack top is X. Then, if min $\leq$ X $\leq$ max, control goes to address L(X). Otherwise, control goes to address D.

Table 1-4 Miscellaneous Operations

EFF	Mnemonic	Description
30	STIND	Store indirect: S := S - 2 !(S!2) := S!1
31	PUSHT	Push T register. S := S + 1 !S := T
32	GETBYTE	Fetch byte from memory. S := S - 1 S!0 := getbyte(S!0, S!1)
33	PUTBYTE	Store byte into memory. S := S - 3 putbyte(S!1, S!2, S!3)
34	SLCTAP	Extract field (see SLCTST). S := S - 1 S!0 := (S!1) of (S!0)
35	SLCTST	Deposit field. S := S - 3 (S!2) of (S!1) := S!3

Conceptually, the SLCTAP and SLCTST operations both operate on (1) the address of a vector containing the field and (2) a field selector that defines the string of bits to be extracted or deposited; this string may not cross word boundaries. The selector has three components: the size (number of bits in the field), the shift (number of right shifts needed to right justify the field in a word), and the offset (number of words from the beginning of the vector to the word containing the field). These components are packed into a single word for the SLCTAP and SLCTST operations; the method of packing may vary from machine to machine.

36	MOD	BCPL supports constructs like "var +:= 1", where a binary operator is combined with an assignment. Accordingly, BCODE supplies the MOD and MODST operations, so that, for the above construct, one need only push the address of var, followed by the number 1; then the sequence

MOD PLUS MODST

will perform the required addition combined with assignment. In detail, MOD replaces the stacked address of var with its contents and saves the address in register T; PLUS leaves the sum on the stack top; and MODST stores this sum in the

location saved in register T.

In other words, MOD performs:

```
T := !(S - 1)
!(S-1) := !T
```

and MODST performs:

```
!T := !S
S := S - 1
```

37 MODST Store result of modified instruction. (See MOD.)

2. The BCODE Code Generator

As should be apparent from the description of the BCODE machine, there is a very close correspondence between OCODE and BCODE. In addition, examination of the code for the BCPL compiler reveals that it has been divided into three logical parts: the parser, the translator and the code generator. The parser utilizes recursive descent techniques to construct the parse tree. The translator walks the parse tree to generate the intermediate OCODE. This OCODE is translated into IBM 370 code by the code generator.

To allow the code generator to operate on OCODE, it is made available in an encoded form in an OCODE buffer. To produce BCODE, the code generator for the 370 has been replaced by a BCODE code geneartor. The BCODE code generator exists in two parts: one which is machine independent (section BCGEN), and another which is machine dependent (section OBJHPGEN.) In general, the machine dependent part defines the loader format for a particular macgine. If the machine independent linker/loader (described in sections 3 and 4) were used, the entire code generator would be machine independent (with the wordsize being the only parameter.) The code generator currently generates HP assembler source, however. The mapping between OCODE and BCODE is described in Appendix 1. The following appendix (appendix 2) gives instructions on how to use the BCODE code generator under MTS.

2.1. The Code Compaction Problem

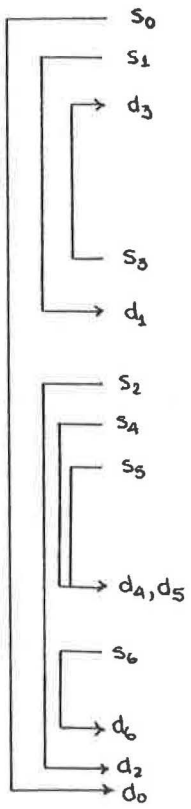
One of our primary objectives in undertaking this project was to transport large programs such as the BCPL compiler onto a mini-computer. The major problem in such endeavors is that one invar-

iably runs short of memory space. Therefore, in all aspects of the system, the generation of dense code has always been the over-riding concern. (As an example, we note that the BCODE code generator emits a SWITCHB or a SWITCHX instruction depending on which is more space efficient.)

One technique available for code compaction is the massaging of generated code to change direct addresses into relative addresses. This requires the base machine to support relative addressing; the BCODE machine does. Relative addressing is available on various commercially available machines such as the PDP/11, the Nova series, and the HP3000, as well as the virtual Intcode machine. For the latter, a scheme is described in [P] whereby an assembly language source is compacted by scanning the code for instructions with label references. For each label reference, the relative displacement between the instruction and the referenced label is computed. If this displacement value is small enough to fit into the address field of a single word instruction, the instruction is changed from a double word instruction using an absolute address to a single word instruction using a relative address. The freed word is marked appropriately as a hole, and a subsequent pass made over the code to delete all holes.

Not only does the above method require the code to be scanned at least twice for each compaction pass, but in general, several compaction passes are required before all possible compaction is achieved. This is because it is very likely that a given compaction pass will make label references which were previously too far apart sufficiently close for "relativising."

It seems intuitively clear that at least with the simple data structures presently used for storing of code, the iterative nature of the compaction process is unavoidable. If we consider a less general branching structure than that which is assumed above, a possibly better algorithm than the brute-force approach outlined above emerges. In particular, let us assume that all branches are nested, as shown below pictorially



where an s_i is the label of an instruction referencing label d_i. (Note that a barnch or a jump in the following discussing refers to any memory reference instruction, that is, an instruction with a label operand.) Now we define a function displ(s,d) which has as its value the optimal displacement between an instruction labelled s and a location d which it references. Let us define an inactive code segment to be one which has no label reference instructions within it. Then, if a code segment bounded by the

addresses [s,d] is inactive, the optimal displacement is clearly |d-s| address units. Otherwise, the optimal displacement is simply the sum of the optimal displacements of all branches nested one level down augmented by the size of any inactive regions between these branches. In the example above, we see that

$$\text{displ}(s_0,d_0) = (s_1-s_0) + \text{displ}(s_1,d_1) + (s_2-d_1) + \text{displ}(s_2,d_2) + (d_0-d_2)$$

We note that it is a simple matter to actually generate object code as the assembly source is being scanned by the displ function. Hence, assuming nested jumps, it is possible to achieve complete compaction of code in a single pass over the assembly code.

In reality, we note that many jumps will not be nested. If we process such a structure by partitioning all the (potentially unnested) jumps into classes of nested jumps, and then applying the above compaction method to each nested class in succesion, complete compaction is still achievable. Since the compaction of a set of nested jumps is achievable in linear time, intuition suggests that fewer iterations will be required by this recursive compaction technique than would be with the brute-force method.

In the above description, note that we tacitly assumed the ability to locate all jumps nested below a given jump. In practice, the time required to construct such a data structure may render it impractical. The implications of these and other aspects of the method have not fully been considered, it presently being only at the exploratory stage. Nonethless, it seems to be a promising approach to the difficult problem of efficient code compaction.

3.1. Linker Format

A BCODE object module may be viewed as a stream of <opcode,operand> pairs where opcode is itself the pair <operator,listsize>. operator defines the type of operation to be performed; listsize determines the length of the operand list to which the operator is applied. The operand list length is in units of operand records (to be defined below).

Since the same linker operator is typically applied to several contiguous operands, we can extend the definition of an operand to that of a list of operand records each of which has the same format. An operand record can extend over an arbitrary number of words; the only restriction is that each such record must be aligned on word boundaries.

To maintain a high degree of portability, the input to the linker is defined in terms of words. Variations in word length do not alter the basic linker instructions; however, a word must possess the following properties:

- Simple BCODE instructions[1] must fit in one word.
- an opcode (i.e., <operator,listsize> pair) should fit in one word, with operator occupying the left half of the word and listsize the right half.
- the number of characters (i.e., bytes) per word should be a positive integer.
- a BCPL data object should fit in one word.
- a string's length should be encodeable in a character.

From the above, it should be evident that the only parameter required in defining the linker for a specific machine are the number of bytes per word

---

[1] A simple BCODE instruction is one which fits into one BCODE machine word.

and the number of bits per word. In the following description they are denoted by the constants BYTESPERWORD and BITSPERWORD.

3.2. Linker Instructions

The basic purpose of a linker is to resolve external addresses. To this end, some declaration commands are needed to define external references and entry points. Although the linker does not modify the relocation flags, it nonetheless needs to look at instructions defining the actual object data, along with their relocation flags. This information is passed on to the loader for its use. The instructions needed to effect the above functions are now described.

The instructions are defined in their mnemonic form only. In an actual loader the data will be encoded as bit patterns, of course. The following mappings occur between the notational device and the actual bit patterns:

n — the value of listsize (see section 1). It is stored in the right half of the first word of the command. (The left half of the word contains the operator).

ss_i — is a string of arbitrary length. It is represented as a vector of characters $c_0\ c_1\ c_2\ \ldots\ c_k$ where k represent the string length. Note that c_0 is initialized to k. An example is the string "hello" which is encoded as

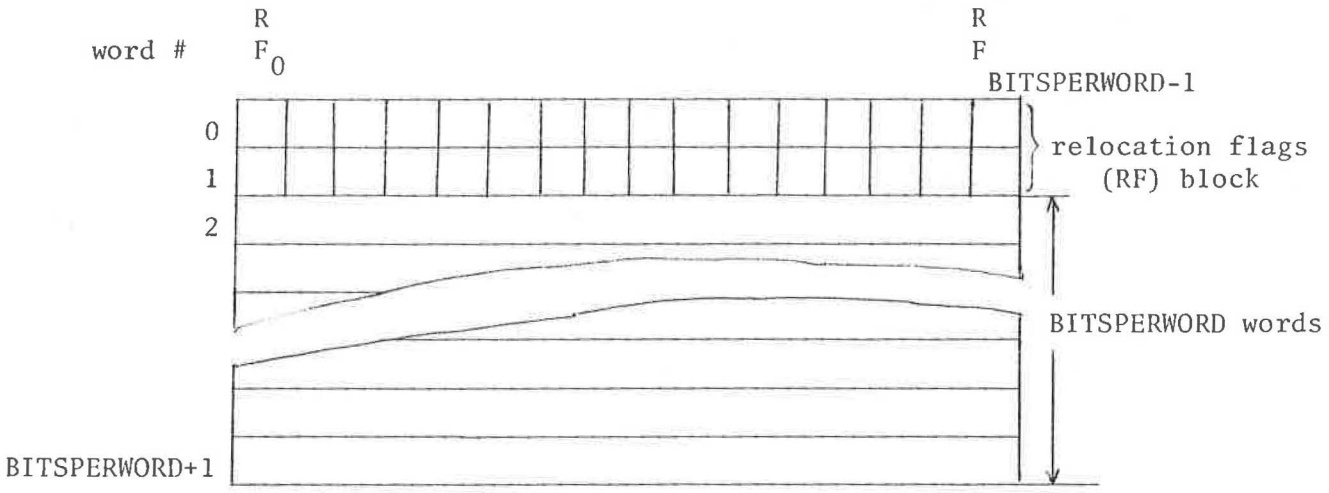
c_0	c_1	c_2	c_3	c_4	c_5
5	'h'	'e'	'l'	'l'	'o'

A string of length k occupies $\lceil k/\text{BYTESPERWORD} \rceil$ words.

The string data is left justified (i.e., with any unused bytes at the right hand side).

$\underline{L_i}$ defines a label. It occupies the next available word.

$\underline{cb_i}$ defines a code block. A code block has the following form:



Each $\underline{cb_i}$ therefore contains a maximum of BITSPERWORD words of object code. Since there are BITSPERWORD bits in each word, each word of object can be assigned a 2-bit relocation flag. The relocation flags have the following definitions:

Flag value	Meaning
0	Word unused
1	Absolute
2	Normal relocatable
3	External reference

An external reference flag indicates that the corresponding code word contains an external *unresolved) address. The word is initialized to the index into the external symbol

references in the ordering defined by the external symbol declarations (see section 3.2.2.).

addr defines a relative address in the object module being linked. It occupies the next available word.

val defines a value which can be stored in a word. It is stored in the next available word.

3.2.1. Entry declarations

Entry declarations are of the form

$$\text{ENTRY} \quad n \quad ss_1 \quad L_1 \quad ss_2 L_2 \quad \ldots \quad ss_n L_n$$

where ss_i is the string naming the entry and L_i is the corresponding address. The ENTRY command is encoded as 1.

3.2.2. External declarations

External declarations take the form

$$\text{EXTERNAL} \quad n \quad ss_1 \quad L_1 \quad ss_2 L_2 \quad \ldots \quad ss_n L_n$$

where ss_i is the string naming the external value referenced and L_i is a pointer to a list of locations which use the external.

The EXTERNAL declaration is encoded as 2.

3.2.3. Code Segments

Object code is specified by the use of the CODE command as follows:

CODE n cb_1 cb_2 ... cb_n

Note that although relocatable flag information is not destroyed by the linker, most normal relocatable values are modified when linking together several segments. Also note that the relocation flag stating 'unused' is used to fill up any extra cells which are unused in a code block. The empty cells do not actually exist in an object module's image; their non-existence is detectable via the relocation flags.

CODE is encoded as 3.

3.2.4. Patch declarations

As symbols in an ENTRY list are processed, various external symbols become resolvable. In general, the external references may have occurred very far back in the pair of object modules being linked. To avoid having to store the entire object module in main memory, the PATCH command is defined. Its function is to direct the loader to perform a patch to memory while loading. The format of the command is

PATCH addr val

where the relative address addr is modified by loading the value val into it.

Note that all PATCH commands are executed by the loader. The value of the PATCH operands addr and val may change however as the relative displacements of instructions change as new object modules are linked.

The PATCH directive is encoded as 4.

3.2.5. Miscellaneous control commands

Various forms of linker/loader directives may potentially exist (e.g. for execution time debugging control). One necessary in a basic linker is a directive to specify the end of an object module. It is specified by

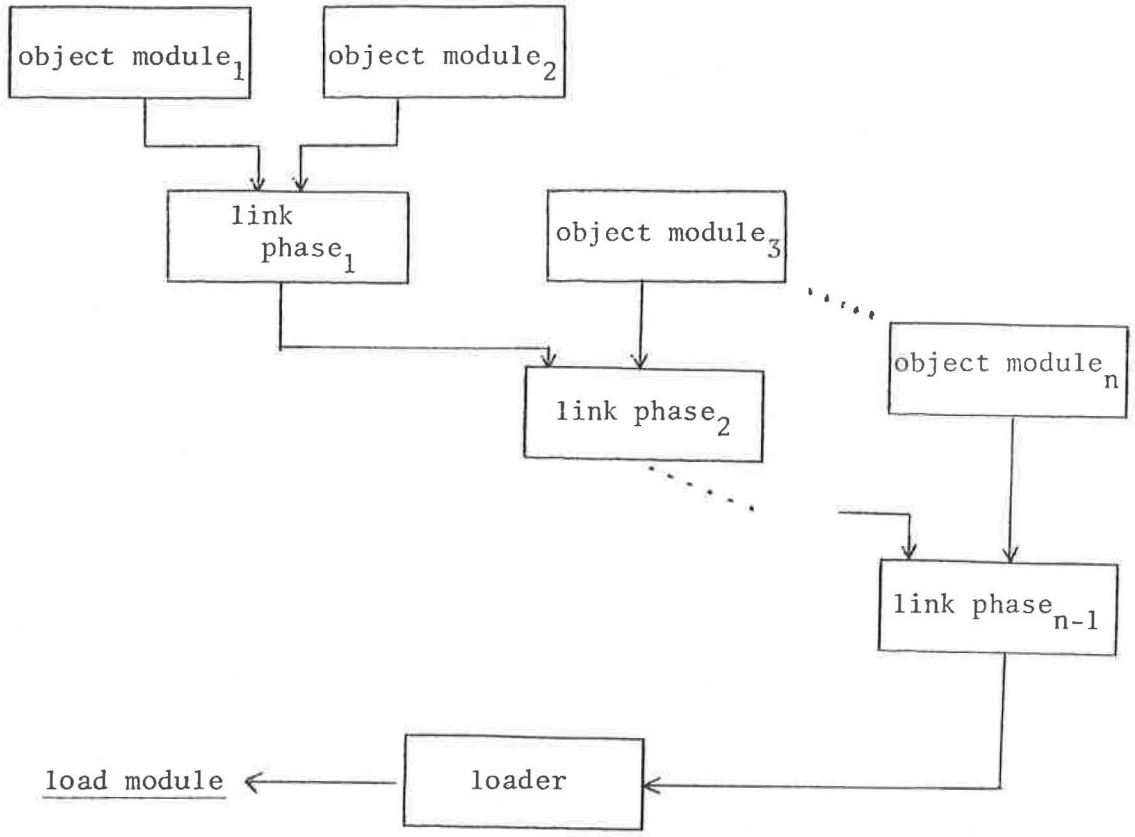
END.

The encoded value of END is 5.

4. The BCODE Loader

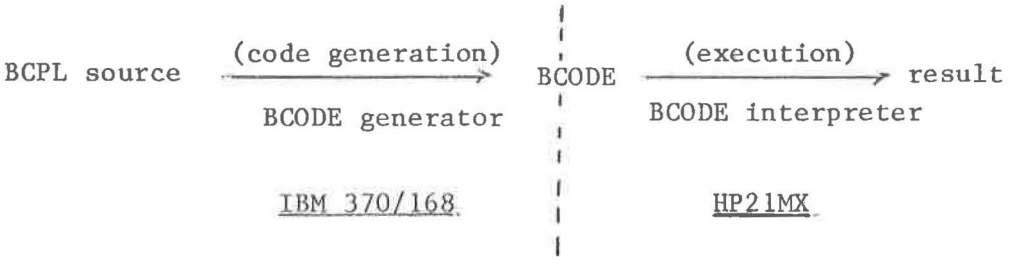
Although the BCODE linker (see section 3) and loader have not been implemented, their design is nevertheless presented because they form an integral part of any machine independent BCODE system. Since the loader is simply the final phase of object module processing after all linking, this section was to contain a description of how this phase was implemented. The loader not having been implemented, this section is empty.

We note, however, how the loader fits into the overall link/load process by the following block diagram:



5. An Interpreter for the BCODE Machine.

With the exception of a few features (most noteably external symbols), an interpreter for the BCODE machine has been implemented on the HP21MX system in microcodes. The implementation is straightforward. A number of BCPL programs have been executed on the HP machine via



Our data show that the percentage gain in space reduction as compared to the same BCPL source being translated to OCODE and then to HP Assembler codes (the currently standard procedure to execute BCPL programs on the HP) ranges from 10% to 40% depending on the characteristics of the programs.

References:

[P] Peck, J.E.L., V.S. Manis, W.E. Webb, Code Compaction for Minicomputers with Intcode and Minicode, Technical Report 75-02 Computer Science, UBC, 1975.

[R] Richards, M., The BCPL Programming Manual, Computer Science Technical Manual 75-10, UBC, 1975.

[W] Wong, Kenny W.O., Introduction to OCODE, Computer Science, UBC 1976.

Appendix 1. OCODE to BCODE Translation.

This appendix defines the OCODE to BCODE translation process. Each OCODE instruction is listed along with its BCODE equivalent. In some cases, a sequence of OCODE instructions translates into a single BCODE equivalent; this sequence is listed together.

The notation to be used is defined below:

n	Some (non-negative) integer.
Ln	Label as used in LAB L42 ... JUMP L42 the notation L(1), L(2), ..., L(k) indicates that k labels are required.
q	Global cell number (for example, q=37 in "LG 37"). The notation q(1), q(2), ..., q(k) indicates that k globals are required.
ss	A string given in one of two forms. Form 1 is a number n (length of string) followed by n integers. Each integer is the decimal representation of a single character in the string. The second form also begins with a number n, but the length is followed by n characters enclosed in double quotes. For example, the string "ABC" can be represented as 3 65 66 67 (in ASCII) or as 3 "ABC" The second form is clearly preferable as it is machine independent.
sz sh of	Three integers (size, shift, offset) used to define a selector for SLCTAP and SLCTST.

OCODE Stmt	BCODE	Comments
ABS	ABS	Unary operator: abs
COM		BCPL statement delimiter
DATALAB Ln		Label definition (for data)
DEBUG		??? (Not implemented)
DIV	DIV	Binary operator: div
EQ	EQ	Binary operator: eq
EQV	EQV	Binary operator: eqv
END		End of program
ENDBLOCK n ss(1) ... ss(n)		Marks end of blocks whose names are given by ss(1) ... ss(n)
ENDPROC ss		Marks end of procedure ss
ENTLABS ...	(see right)	Define entry points. OCODE format is: ENTLABS n ss(1) L(1) ss(2) L(2) : : ss(n) L(n) where 'n' is the number of symbols and 'ss(i) L(i)' give the name (in string form) and the value (a label) of symbol i. Note: If a label is 'I0', its corresponding symbol is external.
ENTRY ...	(see right)	Indicate the start of a procedure. OCODE format is: ENTRY n L c(1) ... c(n) where 'L' is the label defining the start of the procedure, and 'c(1) ... c(n)' are the n characters of the procedure's name.
FINISH	FINISH	Halt BCODE machine

OCODE Stmt	BCODE	Comments
FIX		??? (Not implemented)
FNAP n	CALL n-1 PUSHT	Function call; PUSHT pushes the function result back cnto the stack. Note that FNAP translates into two BCODE instructions.
FNRN	FNRN	Return from function
GBYTE	GETBYTE	Binary operator: %
GE	GE	Binary operator: >=
GLOBAL ...		Define globals. OCODE format is: GLOBAL n q(1) L(1) q(2) L(2) : : q(n) L(n) This OCODE instruction specifies that each qlobal q(i) is to be initialized to label L(i).
GOTO	GOTO	Indirect jump (destination address is cn top of stack)
GR	GR	Binary operator: >
INCLUDE		??? (Not implemented)
ITEMC 'ch'		Define a cell initialized to character 'ch'
ITEMF ss		??? (Not implemented)
ITEML Ln		Define a cell initialized to the address of label Ln
ITEMN n		Define a cell initialized to integer n
JF Ln	JF Ln	Jump if false (top of stack tested and popped off)
JT Ln	JT Ln	Jump if not false (see JF)
JUMP Ln	JUMP Ln	Unconditional jump (stack undisturbed)

LAB Ln		Define label (see also DATALAB)
LC 'ch'	L n	load character n = binary value of 'ch'
LE	LE	Binary operator: <=
LF ss		??? (Not implemented)
LG q	LIG q	Load contents of global cell
LL Ln	LI a	Load contents of labelled cell a = address of label
LLG q	LG q	Load address of global
LLP n	LP n	Load address of cell n on the current stack frame
LLX ss	L a	Load address of external cell
LN n	L n	Load number
LP n	LIP n	Load cell n on current stack frame
LOGAND	LOGAND	Binary operator: &
LOGOR	LOGOR	Binary operator: |
LS	LS	Binary operator: <
LSHIFT	LSHIFT	Binary operator: <<
LSTR ss		Load address of string a = address of ss
LX ss	LI a	Load contents of external cell where string "ss" specifies the name of the external symbol and "a" is that symbol's address.
MINUS	MINUS	Binary operator: -
MOD op	MOD op MODST	Binary operator 'op' is modified so that it will store the result back into the first operand. Note that three BCODE instructions are needed.
MODSLCT		Not used by BCPL compiler

OCODE Stmt	BCODE	Comments
NE	NE	Binary operator: ¬=
NEG	NEG	Unary operator: -
NEQV	NEQV	Unary operator: neqv
NEEDS ss		???
NOT	NOT	Unary operator: ¬
PARAMETER		??? (Not implemented)
BYTEADDR PBYTE	PUTBYTE	Store byte. Note that PUTBYTE is a ternary operator; the sequence BYTEADDR PBYTE translates into a single BCODE instruction.
PLUS	PLUS	Binary operator: +
RES Ln	JUMP Ln	Unconditional jump to end of valof block
REM	REM	Binary operator: rem
RSHIFT	RSHIFT	Binary operator: >>
RSTACK n		Not used in BCODE machine
RTAP n	CALL n-1	Routine call
RTRN	RTRN	Return from routine
RV	RV	Unary operator: rv
SAVE n	SETS n-1	Set stack pointer
SECTION ss		Start of new program section
SG g	SG g	Store into global
SL Ln	S a	Store into labelled location a = address of label
SLCTAP sz sh of	L f SLCTAP	Extract field (f = {of,sz,sh} packed). Note that two BCODE instructions are required.
SLCTST sz sh of	L f SLCTST	Deposit field (like SLCTAP)
SP n	SP n	Pop the top element off the stack and store it at index n

		in the current stack frame.
STACK n	SETS n-1	Set stack pointer to location n-1 in current stack frame
STARTBLOCK		Marks start of block
STIND	STIND	Store indirect (destination address is on top of stack)
STRING ss		??? (Define string?)
STORE		Instructs code generator to flush all temporary values to core
SWITCHON	(see right)	Multi-way branch. OCODE format is: SWITCHON n D C(1) L(1) C(2) L(2) : : C(n) L(n) where 'n' is the number of cases, 'D' is the default label, and 'C(i) L(i)' is the value-label pair for case i. The BCODE machine supports switching via a binary search (SWITCHB) or an indexed jump (SWITCHX).
SX ss	S a	Store into external cell a = address of external symbol ss

No floating point operators (of format #xxx) are implemented.

Appendix 2: Using the BCODE Code Generator Under MTS

The BCPL compiler generating BCODE is invoked via the MTS command

```
$RUN YYC4:BCPL [SCARDS=sourcefile] [SPRINT=printfile]
               [0=ocodefile] 1=objfile PAR=par
```

where sourcefile is the name of the BCPL source file

printfile is the name of the file to receive the program listing

ocodefile is the name of the file to which the OCODE is to be punched (if desired)

objfile is the name of the file to which the HP Assembler source for the BCODE is to be emitted

par is the standard parameters string supported by the BCPL-V compiler (see [R]). At a minimum PAR=A/L must be specified.

The number of BCODE instructions generated is printed on SPRINT. The BCODE file (objfile) may be transferred to paper tape via the system program *PUNCHJOB.