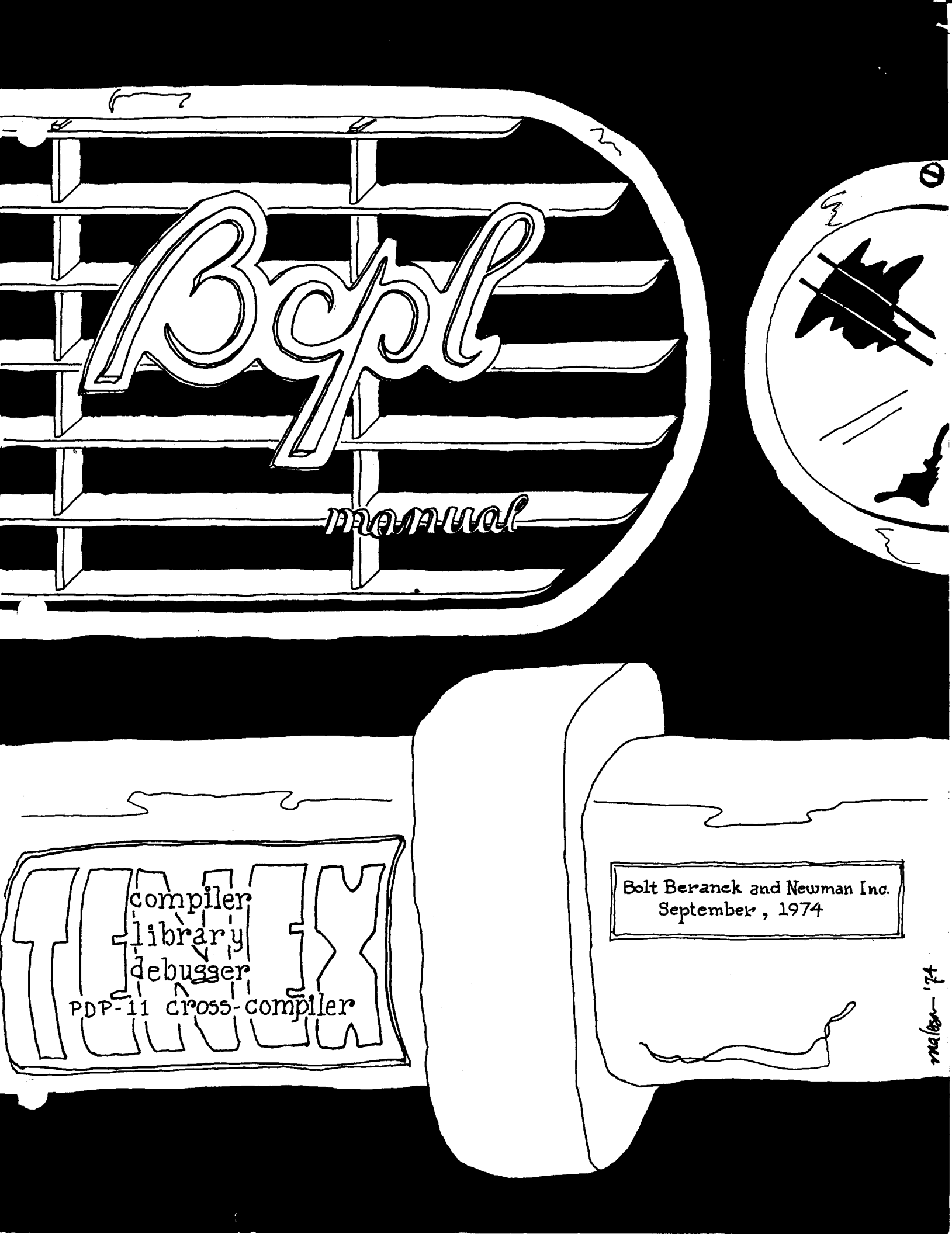

BCPL
manual
compiler
library
debugger
PDP-11 cross-compiler
Bolt Beranek and Newman Inc.
September, 1974
malcolm '74

The BCPL Reference Manual

This manual was originally written by Martin Richards at MIT Project MAC in 1964. It was revised by Henry Ancona at MIT Lincoln Laboratory in 1967, and by Paul Rovner, Jerry Burchfiel, and Jerry Wolf at BBN in 1973. The section on structures was originally written by Art Evans at MIT Lincoln Laboratory.

ABSTRACT

BCPL is a simple recursive programming language designed for compiler writing and system programming; it was derived from CPL (Combined Programming Language [1]) by removing those features of the language which make compilation difficult, namely, the type and mode matching rules and the variety of definition structures with their associated scope rules.

BCPL is a language which is readable, easy to learn and efficient. It is made self-consistent and easy to define accurately by an underlying structure based on a simple idealized object machine. The treatment of data types is unusual and it allows the power and convenience of a language with dynamically varying types and yet the efficiency of FORTRAN. BCPL has been used successfully to implement a number of languages and has proved to be a very useful tool for system programming. The BCPL compiler itself is written in BCPL and has been designed to be easy to transfer to other machines.

Table of Contents

Page

1. Acknowledgements

2. Introduction

3. Fundamental Concepts of BCPL
   - 3.1 The Object Machine
   - 3.2 Names, Variables, and Manifest Constants
   - 3.3 Addresses
   - 3.4 Simple Assignment
   - 3.5 The lv Operator
   - 3.6 The rv Operator
   - 3.7 Data Structures
   - 3.8 Data Types

4. Expressions
   - 4.1 Primary Expressions
     - 4.1.1 Numerical Constants
     - 4.1.2 Character Constants
     - 4.1.3 String Constants
     - 4.1.4 Names
     - 4.1.5 Boolean Constants
     - 4.1.6 Unspecified Initial Value
     - 4.1.7 Parenthesized Expressions
     - 4.1.8 valof Expressions
     - 4.1.9 Function Applications
     - 4.1.10 Vector Applications
     - 4.1.11 lv Expressions
     - 4.1.12 rv Expressions
     - 4.1.13 Half-word Extraction Expressions
     - 4.1.14 Quarter-word Extraction Expressions
     - 4.1.15 Structure References
   - 4.2 Arithmetic Expressions
   - 4.3 Relational Expressions
   - 4.4 Shift Expressions
   - 4.5 Logical Expressions
   - 4.6 Half-word Combination Expressions
   - 4.7 Conditional Expressions
   - 4.8 table and list Expressions
     - 4.8.1 Tables
     - 4.8.2 Lists
   - 4.9 selecton Expressions
   - 4.10 repname

5. Commands
5.1 Simple Assignment Commands
5.2 Assignment Commands
5.3 Routine Calls
5.4 Labelled Commands
5.5 goto Commands
5.6 if Commands
5.7 unless Commands
5.8 while Commands
5.9 until Commands
5.10 test Commands
5.11 Repeated Commands
5.12 for Commands
5.13 switchon Commands
5.14 loop, break, and endcase Commands
5.15 finish Commands
5.16 return Commands
5.17 resultis Commands
5.18 Sections and Blocks

6. Definitions
6.1 Scope Rules
6.2 Space Allocation and Extent of Variables
6.3 Externals
6.4 Globals
6.5 Statics
6.6 Manifests
6.7 Simple Variables
6.8 Vectors
6.9 Functions
6.10 Routines
6.11 Simultaneous Definitions

7. Structures
7.1 Introduction
7.2 Syntax
7.3 Semantics
7.4 Examples

REFERENCES

APPENDICES

A. BCPL Characteristics
- A.1 Reserved Words and Symbols
- A.2 The TENEX BCPL Character Set
- A.3 The BCPL Pre-processor:
  - Comments
  - Semi-colon and DO Insertion
  - PSEUDO Commands (e.g. GET)
- A.4 Subtle Features (for new users to watch out for)
- A.5 Operator Precedence

B. Usage of TENEX BCPL
- B.1 Typical source file organization
- B.2 Using the compiler
- B.3 Constructing a BCPL main program
- B.4 Routine and Function linkage conventions
- B.5 Utility programs
  - B.5.1 FMT.SAV
  - B.5.2 OCODE.SAV
  - B.5.3 PSYMB.SAV
  - B.5.4 PSAVE.SAV
  - B.5.5 CONC.SAV
- B.6 A Complete, Realistic, Working Example Program

C. Functions, Routines, and Special Static Variables in the TENEX BCPL Library
- C.1 I/O
  - C.1.1 I/O Streams
  - C.1.2 Character, Word, and String I/O
  - C.1.3 Integer and Floating Point I/O
  - C.1.4 ARPANET Interface
  - C.1.5 Formatted Output
- C.2 JSYS Interface
- C.3 Byte Manipulation
- C.4 String Manipulation and Number Conversion
- C.5 Error Handling
- C.6 Arrays
- C.7 Hash-coded Dictionary
- C.8 "Heap" Free Storage
- C.9 PSI Handling
- C.10 Miscellany

D. TENEX BCPL Maker's Guide

E. Debugging

F. PDP-11 BCPL
- F.1 Introduction
- F.2 PDP-11 Objects
- F.3 PDP-11 Operations
- F.4 PDP-11 Addressing
- F.5 The Stack Discipline

1. Acknowledgements

BCPL was originally designed and implemented by Martin Richards at MIT Project MAC. Many people have since contributed to the development of the language, compiler, utilities, and debugging system, as represented in the TENEX BCPL system. The language and compiler were extended to include structures, symbol tables, and various new language constructs by the group at MIT Lincoln Laboratory on the TX-2 computer, led by Art Evans. Carl Ellison (University of Utah Computer Science Department) built a TENEX BCPL bootstrap and brought an early version of the Lincoln compiler to Utah-TENEX through the ARPANET. He also built the first TENEX BCPL I/O library. Paul Rovner brought the Utah system to BBN-TENEX through the ARPANET, then used it to bootstrap an improved version of Lincoln's compiler. Victor Miller (BBN) helped to upgrade Ellison's code generators. The present TENEX BCPL system includes several packages of routines and functions, and several utility programs that were brought from Lincoln and modified for use on TENEX by Paul Rovner. Other utilities were contributed by Jerry Wolf, Ray Tomlinson, and Richard Schwartz at BBN. The TENEX BCPL debugger was designed and implemented by Jim Miller and Paul Rovner, with help by John Sybalsky. And Gail Hedtler of the TENEX group at BBN spent many hours puzzling through pencil scratched versions of this manual as she converted it to RUNOFF format and keyed it in.

2. Introduction

This document is designed to be an introduction to the BCPL programming language, a reference manual for it, and a user's manual for the BCPL programming system on TENEX. The description here of the BCPL programming language is independent of any particular implementation; features of the language which depend on the implementation (like the number of bits in a word) are pointed out (where appropriate) in the text.

The reserved words and symbols used in the language descriptions (and in the examples) are taken from the reserved words and symbols for TENEX BCPL. By convention, reserved words are composed of lower case characters only. Reserved words are underlined in this manual.

Section 3 (below) is an introduction to the philosophy of BCPL and to the key elements of the language. The next four sections describe in detail the form and meaning of the language constructs for expressions, commands, definitions, and structures, respectively. The appendices describe the TENEX BCPL programming system and how to use it. Also in the appendices are the list of reserved words and symbols, a description of the features of the pre-processor, and a description of the inevitable (but very few!) glitches out for which the new user should look. Attached as addenda to this document are the figures referred to in the text, and a chart of BCPL operator precedence relations.

The BCPL convention for program comments is that two adjacent "/" characters anywhere in the program identify the remainder of the line as arbitrary text, to be skipped over and ignored by the compiler.

The syntactic notation used is basically BNF with the following extensions:

(1) The symbols N, E, D, and C are used as shorthand for <name>, <expression>, <definition>, and <command>.

(2) The metalinguistic brackets '<' and '>' may be nested and thus used to group together more than one constituent sequence (which may contain alternatives). An integer subscript may be attached to the metalinguistic bracket '>' and used to specify repetition. If it is the integer n, then the sequence within the brackets must be repeated at least n times; if the integer is followed by a minus sign, then the sequence may be repeated at most n times or it may be absent.

3. Fundamental Concepts of BCPL

3.1 The Object Machine

BCPL has a simple underlying semantic structure which is built around an idealized object machine. This method of design was chosen in order to make BCPL easy to define accurately and to facilitate machine independence, which is one of the fundamental aims of the language.

The most important feature of the object machine is its memory store. This is represented diagrammatically in Figure 1. It consists of a set of numbered boxes (called "storage cells") arranged so that the numbers labelling adjacent cells differ by one. As will be seen later, this property is important.

Each storage cell holds a binary pattern called a "Value". All storage cells are of the same size and the length of Values is a constant of the implementation which is usually between 16 and 36 bits. A Value is the only kind of object which can be manipulated directly in BCPL. Every variable and expression in the language will always have a Value.

Values are used by the programmer to model abstract objects of many different kinds such as truth values, strings, arrays, and functions. There are a large number of basic operations on Values which have been provided in order to help the programmer model the transformation of his abstract objects. In particular, there are the usual arithmetic operations. These may be understood as operations which interpret their operands as integers, perform the integer arithmetic and convert the result back into the Value form; alternatively, one may think of them as operations which work directly on bit patterns and just happen to be useful for representing integers. This latter approach is closer to the BCPL philosophy. Although the BCPL programmer has direct access to the bits of a Value, the details of the binary representation used to represent integers are not defined and he would be losing machine independence if he performed non-numerical operations on Values he knows to represent integers.

An operation of fundamental importance in the object machine is that of indirection. This operation has one operand which is interpreted as an integer and it locates the storage cell which is labelled by this integer. This operation is assumed to be efficient and, as will be seen later, the programmer may invoke it from within BCPL using the rv operator.

3.2 Names, Variables, and Manifest Constants

A BCPL name (see 4.1.4) is associated either with a storage cell, in which case the name represents a "variable", or with a constant Value, in which case the name represents a "manifest constant". The Value of a BCPL variable is the Value contained in the cell; the term "variable" is used since this Value may be changed by an assignment command during execution. Variables are introduced by let and and declarations, the for command, formal parameter lists, and the static declaration. From the point of view of how storage cells are assigned to variables, there are two kinds of variables: "dynamic" and "static". A Dynamic variable is assigned a storage cell each time the program in which it is defined is executed. When this program finishes, the storage cell is reclaimed for use by other programs. A static variable is assigned its storage cell by the compiler, before program execution. This storage cell is uniquely associated with the static variable, and this association does not change during program execution.

A "manifest constant" is a name which is associated with a constant Value; this association takes place at compile time and remains the same throughout execution. Manifest constants are introduced by the manifest declaration and by the label declaration (see 5.4). There are many situations where manifest constants can be used to improve readability at no cost in run time efficiency, for example:

```
manifest { PI : 3.1415926}
```

3.3 Addresses

As previously stated, each storage cell is labelled by an integer; this integer is called the Address of the cell. Since a variable is associated with a storage cell, it must also be associated with an Address and one can usefully represent a variable diagrammatically as in Figure 2.

Within the machine an Address is represented by a binary bit pattern of the same size as a Value, and so a Value can represent an Address directly. Thus, a variable may have the Address of some storage cell as its Value. The programmer might think of this Value as a "pointer" to the storage cell.

3.4 Simple Assignment

The syntactic form of a simple assignment command is:

E1 := E2

where E1 is either a variable or some other expression which represents a storage cell (for example, see 4.1.10), and E2 is an arbitrary expression. Loosely, the meaning of the assignment is to evaluate E2 and store its Value in the storage cell referred to by E1. This process is shown diagrammatically in Figure 3.

Example:

X := 7

3.5 The lv Operator

As previously stated, an Address is represented by a binary bit pattern which is the same size as a Value. The lv operator provides the facility of accessing the Address of a storage cell.

The syntactic form of an lv expression is:

lv E

where E is an expression which represents a storage cell. The process of evaluation for the lv expression is shown diagrammatically in Figure 4. The Value of the lv Expression is the Address of the given storage cell.

3.6 The rv Operator

The rv operator is important in BCPL since it provides the underlying mechanism for manipulating data structures; it operates to yield the storage cell whose address is the Value of the operand.

The syntactic form of an rv expressions is as follows:

rv E

and its process of evaluation is shown diagrammatically in Figure 5. Note that rv (lv E) is identical to E (but only if E has an Address).

3.7 Data Structures

The considerable power and usefulness of the rv operator can be seen by considering Figure 6.

The diagram shows a possible interpretation of the Value of the expression

V + 3.

Some adjacent storage cells are shown; the top one has an Address which is the same bit pattern as the Value of V. One will recall that an Address is really an integer and that Addresses of adjacent cells differ by one, and thus the Value of (V + 3) is the same bit pattern as the Address of the bottom box shown in the diagram. If the operator rv is applied to (V + 3), then the contents of that cell will be accessed. Thus the expression:

rv (V + i)

acts very like a vector subscripting operation, since, as i varies from zero to three, the expression refers to the different elements of the set of four cells pointed to by V. V can be thought of as the vector and i as the integer subscript. The notion of a "vector" is a central one in BCPL. A Value which is used to address the first storage cell in a block of adjacent storage cells is said to define a "vector" of storage cells.

Since this facility is so useful, the following syntactic sugaring is provided:

E1!E2 or E1|E2 is equivalent to rv (E1 + E2)

A simple example of its use is the following command:

V|(i + 1) := V|i + U|i

One can see how the rv operation can be used in data structures by considering the following:

```
V|3 = rv (V + 3)      by definition
    = rv (3 + v)      since + is commutative
    = 3|V
```

Thus V|3 and 3|V are semantically equivalent; however, it is useful to attach different interpretations to them. We have already seen an interpretation of V|3 so let us consider the other expression. If we rewrite 3|V as Xpart|V where Xpart has value 3, we can now conveniently think of this expression as a selector (Xpart) applied to a structure (V).

By letting the elements of structures themselves be structures it is possible to construct compound data structures of arbitrary

complexity. Figure 7 shows a structure composed of integers and pointers.

3.8 Data Types

The unusual way in which BCPL treats data types is fundamental to its design, and thus some discussion of types is in order here. It is useful to introduce two classes:

(a) conceptual types
(b) internal types

The conceptual type of an expression is the kind of abstract object the programmer had in mind when he wrote the expression. It might be, for instance, a time in milliseconds, a weight in grams, a function to transform feet per second to miles per hour, or it might be a data structure representing a parse tree. It is, of course, impossible to enumerate all the possible conceptual types, and it is equally impossible to provide for all of them individually within a programming language. One standard practice when designing a language is to select from the conceptual types some basic ones and provide a suitable internal representation together with an adequate set of basic operations. The term "internal type" refers to any one of these basic types; the intent is that all the conceptual types can be modelled effectively using the internal types. A few of the internal types provided in a typical language, such as CPL, are listed below:

real
integer
label
integer function
(real, boolean) vector

Much of the flavor of BCPL is the result of the conscious design decision to provide only one internal type, namely, the binary bit pattern (or Value). In order to allow the programmer to model any conceptual type, a large set of useful primitive operations has been provided. For instance, the ordinary arithmetic operators + - * and / have been defined for Values in such a way as to model the integer operations directly. The six standard relational operators have been defined and a complete set of bit manipulating operations provided. All the operations provided are uniformly efficient and they have not been "over-defined". For instance, the effect of adding a number to a label, or a vector to a function is not defined even though it is possible for a programmer to cause it to take place.

The most important effects of designing BCPL in this way can be summarized as follows:

1. There is no need for type declarations in the language, since there is only one type of variable. This helps to make programs concise and also simplifies such linguistic problems as the handling of actual parameters and separate compilation.

2. BCPL has much of the power of a language with dynamically varying types and yet it retains the efficiency of a language (like FORTRAN) with manifest types; although the internal type of an expression is always known by the compiler, its conceptual type can never be, and, indeed, it may depend on the values of variables within the expression. For instance, the conceptual type of V|i may depend on the value of i. One should note that, in languages (such as ALGOL and CPL) where the elements of vectors must all have the same type, one needs some other linguistic device in order to handle more general data structures.

3. Since there is only one internal type, there can be no automatic type checking and it is possible to write nonsensical programs which the compiler will translate without complaint. This slight disadvantage is easily outweighed by the simplicity, power and efficiency that this treatment of types makes possible. Interpretations:

   (a) The Value of a variable of conceptual type vector is a storage cell-sized bit pattern (36 bits for TENEX BCPL) which is interpreted as the Address of the zeroth element of the vector. I.e., rv v and v|Ø represent the same storage cell.

   (b) The Address of the nth element of a vector v may be obtained by adding the integer n to v; thus lv v|n is equal to v + n.

   (c) The Value of a label is a bit pattern representing the program position of the labelled command.

   (d) The Value of a function or routine is a bit pattern representing the program position of the entry point of the function or routine (see 6.9 and 6.1Ø).

***Fig. 1

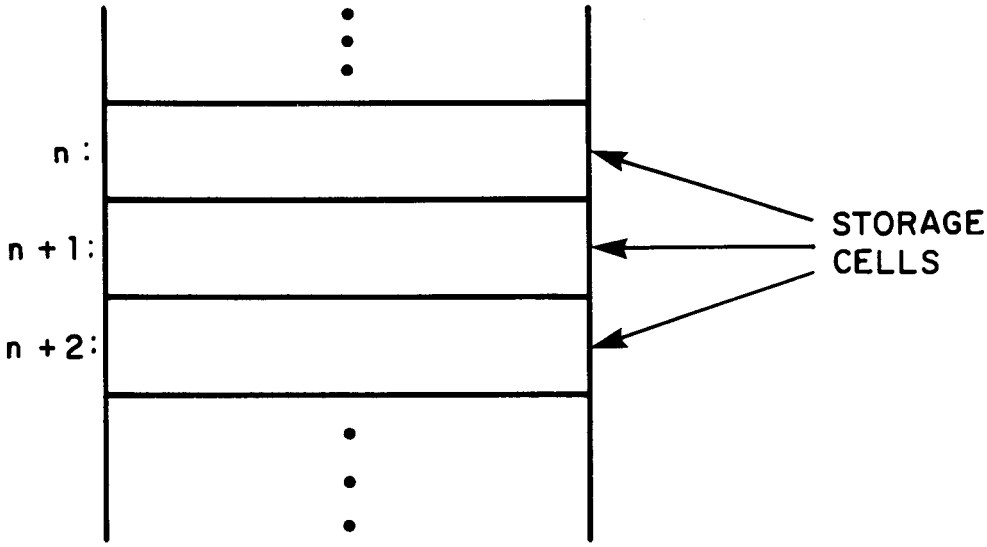


Figure 1: The Memory Store

***Fig. 2

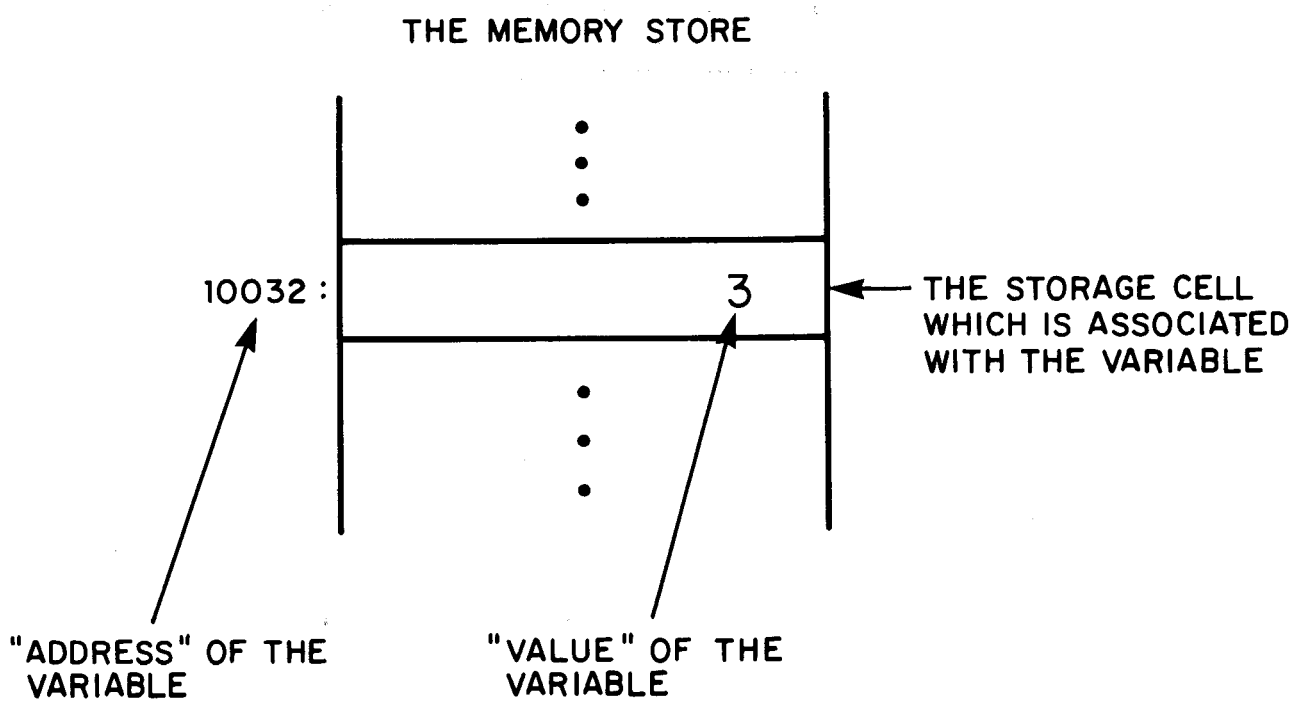


Figure 2: A Variable

***Fig 3

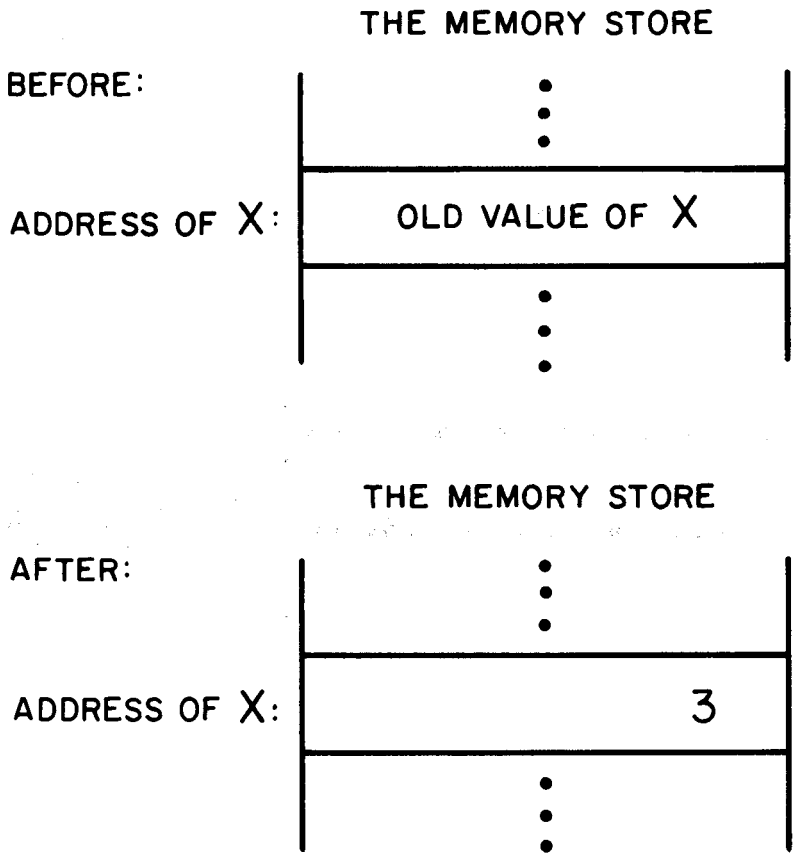
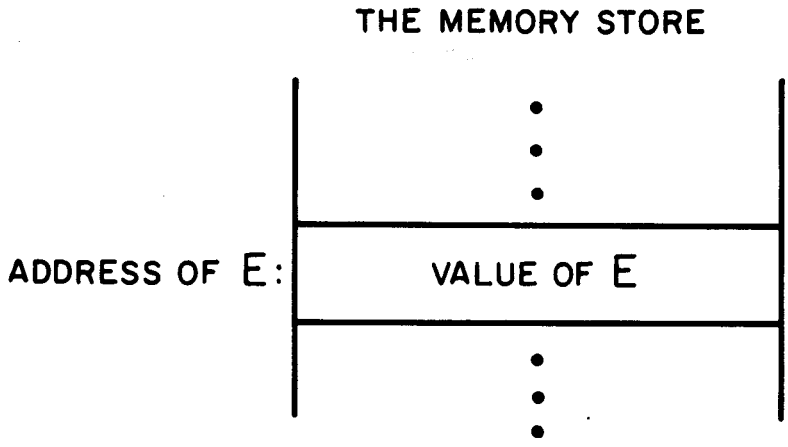


Figure 3: The Assignment Statement
(e.g. x:=3)

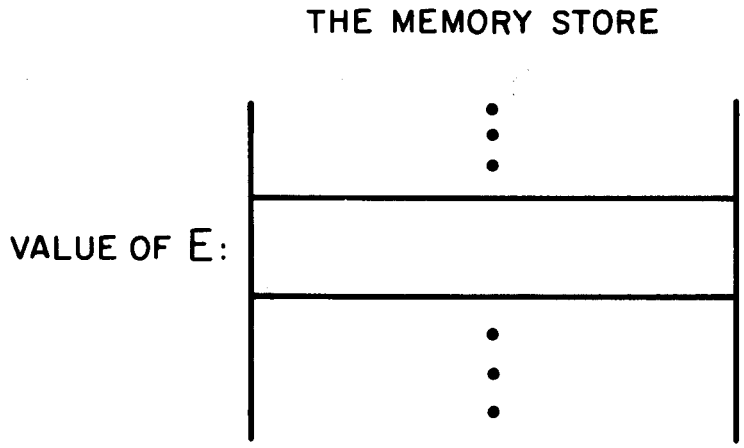
***Fig. 4



IF E REPRESENTS A STORAGE CELL, (lv E) IS AN EXPRESSION WHERE VALUE IS THE ADDRESS OF E. (lv E) IS AN EXPRESSION LIKE (X + 3): IT DOES NOT ITSELF HAVE AN ADDRESS.

Figure 4: The lv Operator

***Fig. 5



FOR THE EXPRESSION (rv E), THE VALUE OF E IS INTERPRETED AS THE ADDRESS OF (OR "POINTER TO") A STORAGE CELL. THE EXPRESSION (rv E) REPRESENTS THIS STORAGE CELL.

Figure 5: The rv Operator

***Fig. 6

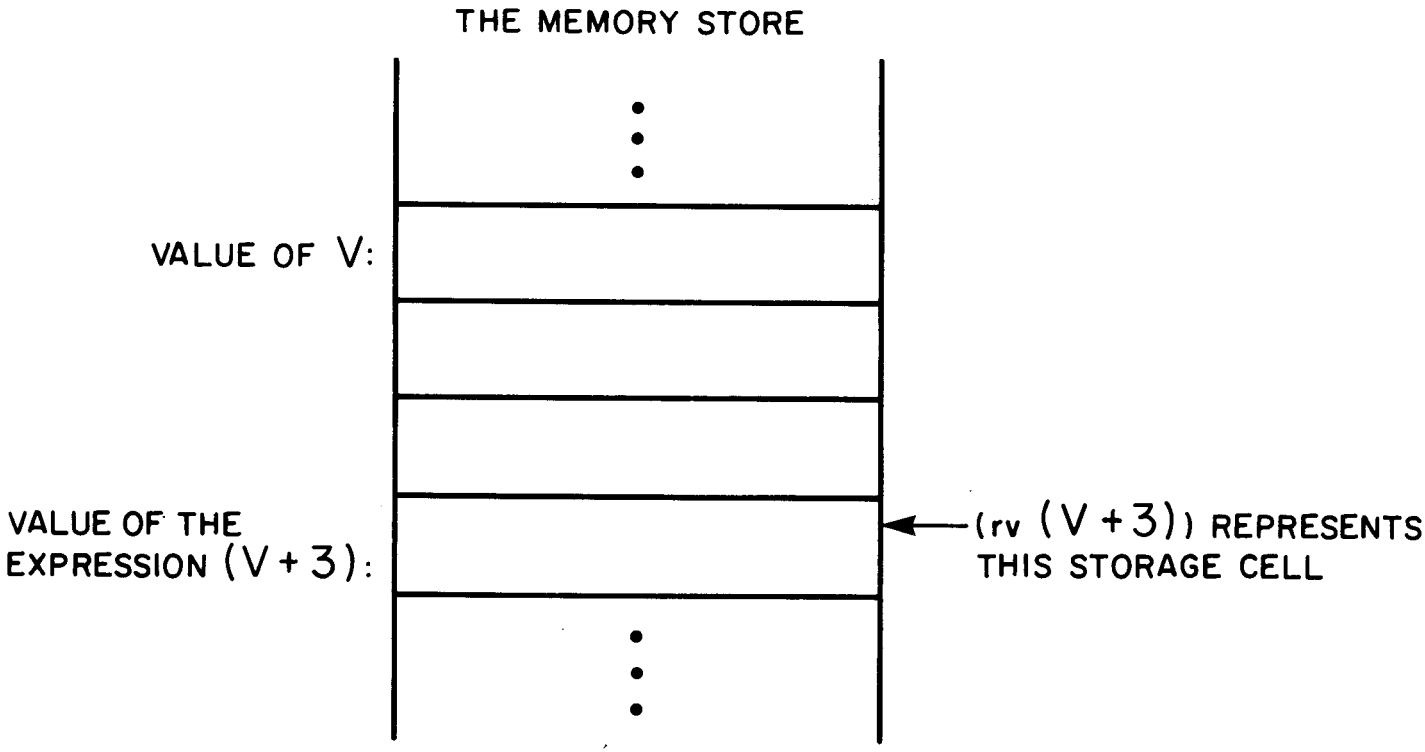


Figure 6: rv (V+3)

***Fig. 7

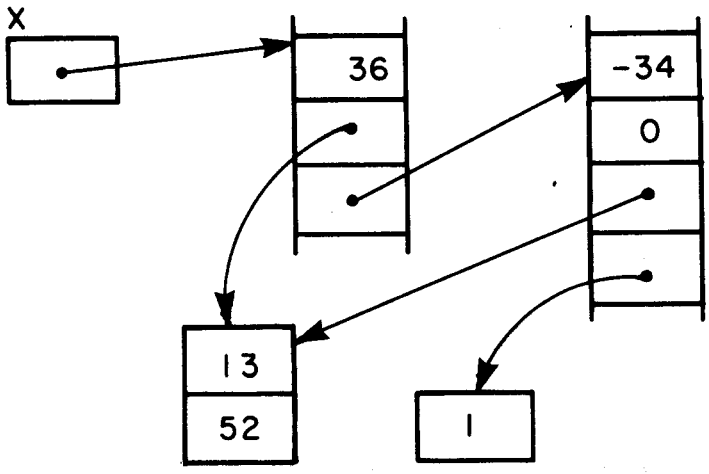


Figure 7: A Structure Composed of Integers and Pointers

4. Expressions

All BCPL expressions are described in this section. They are presented in syntactic classes in the order of decreasing binding power. The term "binding power" refers to the strength with which an operator binds its arguments. For example, the multiplication operator "binds more strongly" than the addition operator. The expression

```
E1*E2+E3
```

means

```
(E1*E2)+E3
```

not

```
E1*(E2+E3)
```

A concise presentation of the relative binding power of the BCPL operators is given in Appendix A.5.

4.1 Primary expressions

These are the most binding and most primitive expressions. They are:

numeric constants, character constants, string constants, names, Boolean constants, nil, bracketted expressions, valof expressions, function applications, vector applications, lv expressions, rv expressions, half and quarter word extraction expressions, and structure references.

4.1.1 Numerical Constants

Syntactic form:

```
    <decimal digit>1
or  #<octal digit>1
or  <decimal digit>1.<decimal digit>1
```

Semantics:

The sequence of digits is interpreted as a decimal integer in the first case, as a right justified octal number in the second, and as a floating point number in the third. In TENEX BCPL, other formats for floating point constants are allowed, per the FLIN JSYS [4].

4.1.2 Character Constants

Syntactic form:

$<character>

Semantics:

A character constant has an implementation dependent Value which is the bit pattern representation of the character; this is right justified and the remainder of the bits in the Value are zeros. See appendix A.2 for the list of TENEX characters and a description of the escape conventions for special characters.
TENEX example:

$a = #141 (ASCII value)

4.1.3 String Constants

Syntactic form:

"<character>1"
or '<character>1'

Example:

"Abc*n"

Semantics:

A string of characters delimited by " represents a BCPL string constant. On TENEX, this is represented as a vector, with the characters in successive words, packed four to a word, from left to right. The leftmost quarter of the zeroth word contains the number of characters in the string. This is limited to 511 (9 bits!). The first character is stored in the quarter which is second from the left in the zeroth word. Extra quarters in the last word of a string will be padded with zeros.

A string of characters delimited by ' is also represented as a BCPL vector: the string characters are packed in successive words of the vector, in ASCIZ string format.
Note that the escape conventions for special characters also hold in string constants.

4.1.4 Names

Syntactic form:

A name is a sequence of one or more (less than 24) characters from a restricted alphabet called the name character alphabet. The particular characters in this alphabet and the rules for recognizing the starts and ends of names are implementation dependent.

The TENEX name character alphabet contains the letters A....Z and a....z and the digits Ø....9. A name must start with a letter.

Semantics:

Two names are equal if they have the same sequence of name alphabet characters. A name may always be evaluated to yield a Value. If the name was declared to be a label or a manifest constant (see section 6.6) then the Value will be the same on every evaluation. If the name was declared in any other way then it is a variable and its Value may be changed dynamically by an assignment command.

4.1.5 Boolean Constants

Syntactic form:

true or false

Semantics:

The actual bit patterns which are the Values of true and false are implementation dependent. On TENEX, the Value of true is a bit pattern entirely composed of ones. The Value of false is zero. Note that true = not false

4.1.6 Unspecified Initial Value

Syntactic form:

nil

Example:

```
let x := nil
```

Semantics:

The Value of nil is undefined. Its purpose is to allow the user to not specify an initial value for a newly defined cell. In the example, the dynamic variable x is defined without an initial Value. nil may also be used in formal parameter lists, actual parameter lists, subword expressions, tables, lists and static declarations.

4.1.7 Parenthesized Expressions

Syntactic form:

(E)

Semantics:

Any expression may be enclosed in parentheses; this is used to specify grouping.

4.1.8 valof Expressions

Syntactic form:

valof <section or block>

Semantics:

A valof expression is evaluated by executing the section or block (see 5.18) until a resultis statement is encountered (see 5.17), which causes execution of the section or block to cease. The Value of the valof expression is the Value of the expression in the resultis command.

Example:

```
char:=valof
     { WriteS("Character:")   //Ask for a character
       resultis Readch()      //read and return it
     }
```

4.1.9 Function Applications

Syntactic form:

E1 (E2, E2, ... En)
where E1 is one of the primary expressions introduced above.

Semantics:

The function application is evaluated by evaluating the expressions E1, E2, ... En and assigning the Values of E2 ... En to the first n-1 formal parameters of the function whose Value is the Value of E1. This function is then entered. The result of the application is the Value of the expression in the function definition (see section 6.9).

4.1.10 Vector Applications (i.e. vector subscripting)

Syntactic form:

E1|E2
or E1!E2
where both E1 and E2 are primary expressions.

Semantics:

A Value of conceptual type "vector" is the Address of the zeroth storage cell in a block of adjacent storage cells. A vector application represents a storage cell. To obtain the Address of this cell, E1 and E2 are evaluated and summed. The Value of the vector application expression is the Value in this cell. E1 is often interpreted as the Value of a vector and E2 as the subscript. From the definition of lv expressions (section 4.1.11), the Address of an element of a vector may be obtained by evaluating the expression

```
lv E1|E2
```

The representations of Vectors, Addresses and integers is such that the following relations are true:

```
E1|E2 = rv (E1 + E2)

lv E1|E2 = E1 + E2
```

Note that

```
E1|E2|E3|E4
```

is calculated as

```
(((E1|E2)|E3 )|E4)
```

also,

```
E1|E2 = E2|E1
```

Function applications are more binding than vector applications, i.e.,

y|f(x) means y|(f(x)).

4.1.11 lv Expressions

Syntactic form:

lv E
where E is a primary expression

Semantics:

The Address of an expression which represents a storage cell may be obtained by applying the operator lv; it is only meaningful to apply lv to a vector application, an rv expression, or an identifier which is not a manifest constant. lv expressions are less binding than vector applications, e.g.,

(lv V|X) is (lv (V|X))

The Value of an lv expression is the Address of the specified storage cell. Examples of operands to the lv operator:

(a) A vector application.
The result is the Address of the element referenced (see section 4.1.10).

(b) An rv expression.
The result is the Value of the operand of rv.

(c) A name.
The result is the Address of the storage cell which is associated with the given name (this name must not represent a manifest constant).

4.1.12 rv Expressions

Syntactic form:

rv E
where E is a primary expression.

Semantics:

An rv expression represents a storage cell whose Address is the Value of the operand (TENEX implementation: The full width of E is used to compute the Address of the storage cell, particularly the indirect and index register fields[CAVEAT!!]). rv expressions are less binding than vector applications.

Examples:

```
rv #100000 := 7
```

stores the Value 7 into the storage cell whose Address is 100000 (octal).

```
rv v := 1 + rv v
```

increments the Value of the storage cell whose address is the Value of v.

```
v|0 := 1 + v|0
```

same as previous example.

4.1.13 Half-word Extraction Expressions

Syntactic form:

lh E
or rh E
or lhz E
or rhz E
where E is a primary expression.

Semantics:

The Value of a half-word extraction expression is the storage cell-sized bit pattern whose right half is the right half (for rh and rhz) or left half (for lh and lhz) of E, and whose left half is all zeros (if lhz or rhz) or sign extended (if lh or rh). The lh and rh half-word extraction expressons may appear on the left side of an assignment statement (see section 5.1). Half word extraction expressions are less binding than vector applications.

4.1.14 Quarter-word Extraction Expressions

Syntactic form:

q1 E
or q2 E
or q3 E
or q4 E
or q1z E
or q2z E
or q3z E
or q4z E
where E is a primary expression.

Semantics:

The Value of a quarter-word extraction expression is the storage cell-sized bit pattern whose rightmost quarter is the indicated quarter of E (q1 indicates the rightmost quarter, q4 the leftmost), and whose remainder is zero (if q1z, q2z, q3z, or q4z), or sign-extended (if q1, q2, q3, or q4). The q1, q2, q3, and q4 quarter word extraction expressions may appear on the left side of an assignment statement (see section 5.1). The binding power of quarter word extraction expressions is the same as that of half word extraction expressions.

4.1.15 Structure References

Structure references are primary expressions. See section 7.

4.2 Arithmetic Expressions

These expressions provide the standard integer and floating point operations of multiplication, division, remainder, addition and subtraction. They are less binding than the primary expressions.

Syntactic form:

```
    E1 * E2
or  E1 / E2
or  E1 rem E2
or  E1 + E2
or  E1 - E2
or  E1 %* E2
or  E1 %/ E2
or  E1 %+ E2
or  E1 %- E2
or  -E1
or  +E1
```

The operators %* %/ * / and rem are more binding than %+ %- + and - and associate to the right [i.e. E1/E2/E3=E1/(E2/E3)]. The operators %+ %- + and - associate to the left [i.e. E1-E2-E3=(E1-E2)-E3].

Semantics:

The integer operators interpret the Values of their operands as signed integers and yield integer results. The operator * denotes integer multiplication. The division operator / yields the correct result if E1 is divisible by E2; it is otherwise implementation dependent but the rounding error is never greater than 1. On TENEX, the result is obtained by an IDIV instruction, which truncates. The operator rem yields the remainder of

(E1 divided by E2)

its exact specification is implementation dependent. On TENEX, the result is obtained by the IDIV instruction. The operators + and - denote integer addition and subtraction. The four floating point operators interpret the Values of their operands as floating point numbers and perform the indicated operations. (Note: Automatic conversion between integer and floating point numbers does NOT occur in BCPL. It is the user's responsibility to use the correct operators).

4.3 Relational Expressions

A relational expression takes integer or floating point arguments and yields a Boolean Value to represent the truth of the relation.

Syntactic Form:

E1 <relop> E2 ... <relop> En
where <relop> ::= eq | ne | ls | gt | ge | le
and n > 1
The relational operators are less binding than the arithmetic operators.
(NOTE: lt and < are synonyms for ls ; gr and > are synonyms for gt ; = is a synonym for eq).

Semantics:

The Value of a relational expression is true if and only if all the individual relations are true. The Values of the expressions E1 ... En are interpreted as signed integers and the relational operators have their usual mathematical meanings. On TENEX, the floating point number representation is such that this interpretation is also correct for floating point relational expressions. Note that the Value of an expression such as x=true is implementation dependent.

4.4 Shift Expressions

The shift operations allow one to shift a binary bit pattern to the left or right by a specified number of bit positions.

Syntactic form:

E1 lshift E2
or E1 rshift E2 s2
or E1 lscale E2
or E1 rscale E2

E2 is any primary or arithmetic expression and E1 is any shift, relational, arithmetic or primary expression. Thus the shift operators are less binding than the relations on the left and more binding than those on the right.

Semantics:

The shift operations are logical operations; the scale operations are arithmetic operations. The Value of E1 is interpreted as a logical bit pattern and that of E2 as an integer. The result of E1 lshift E2 is the bit pattern E1 logically shifted to the left by E2 bit positions. If E2 is negative, shifting occurs to the right. E1 rshift E2 is as for lshift but shifts in the opposite direction. For the shift operators, vacated bit positions are filled with zeros and the result is zero if E2 is greater than the number of bits in a machine word. For the scale operators, the result is:

E1 times (2 to the power E2).

4.5 Logical Expressions

These expressions allow one to manipulate bits of a Value directly. They can be used in conjunction with the shift operators to pack and unpack data. The standard BCPL representations of true and false are chosen so that the logical operators may also be used on Boolean data.

Syntactic form:

~ E1 (also not E1)
or E1 & E2
or E1 \ E2
or E1 eqv E2
or E1 neqv E2

The not operator is most binding; then, in decreasing order of binding power are:

&, \, eqv, neqv

All the logical operators are less binding than the shift operators.

Semantics:

The operands of all the logical operators are interpreted as binary bit patterns of ones and zeros. The application of the not operator yields the logical negation of its operand (bit-by-bit complement). The Value of the application of any other logical operator is a bit pattern whose nth bit depends only on the nth bits of the operands and can be determined by the following table:

The Values of the nth bits	Operator &	\	eqv	neqv
both ones	1	1	1	0
both zeros	0	0	1	0
otherwise	0	1	0	1

4.6 Half-word Combination Expressions

Syntactic form:

E1,,E2
where E1 and E2 may be any logical expression or expressions of greater binding power

Example:

E3,, E4 & E5
parses as E3,, (E4 & E5)

Semantics:

E1,,E2
produces a storage cell-sized Value (36 bits for TENEX BCPL) whose left half is the same as the right half of E1, and whose right half is the same as the right half of E2.

4.7 Conditional Expressions

A conditional expression allows for conditional evaluation of one of two expressions.

Syntactic form:

E1 -> E2, E3
or E1 => E2, E3
where E1, E2, and E3 may be any subword expressions or expressions of greater binding power. E2 and E3 may, in addition be conditional expressions.

Semantics:

The Value of the conditional expression is the Value of E2 or E3 depending on whether the Value of E1 represents true or false respectively. In either case only one alternative is evaluated. If the Value of E1 does not represent either true or false then the Value of the conditional expression is undefined.

4.8 table and list Expressions

These represent the two ways of creating initialized vectors: the table expression causes a vector to be built and initialized at compile time in static storage; the list expression causes a vector to be built and initialized at run time in dynamic storage.

4.8.1 Tables

Syntactic form:

table E0, E1, ... En
where all the expressions are more binding than comma.

Semantics:

A table is a static vector whose elements are initialized prior to execution to the Values of the expressions E0 to En; these expressions must have Values which can be computed at compile time. The Value of a table is a pointer to its zeroth element. The elements of a table may include tables, vectors, or strings.

Example:

```
static{ x := table 1,2,3}
```

4.8.2 Lists

Syntactic form:

list E0, E1, ... En
The initial values E0, E1, ... En can be any expressions.

Semantics:

The expressions are evaluated and stored in the list at the time the list expression is evaluated. The storage is allocated dynamically as for vectors.

```
   let L := list E0, E1, ... En
```

is equivalent to

```
   let L := vec n
   L|0, L|1, ... L|n := E0, E1, ... En
```

4.9 selecton Expressions

Syntactic form:

selecton E into { <list of cases> }
where each "case" in the <list of cases> is a label of the form:

    case <constant>:

or

    case <constant1> to <constant2>:

or

    default:

followed by an expression to evaluate.

Semantics:

As for the switchon command (section 5.13), E is evaluated, and the indicated "case" is selected. The Value of the selecton expression is the Value of the expression which follows the selected case label. If none of the case labels are applicable, the default label is selected. The default label should not be omitted.

4.10 repname

This is a mechanism for a shorthand description of a list of elements which are seperated by commas. It is useful in table and list expressions (4.8), argument lists (4.1.9 and 5.3), formal parameter definition lists (6.9 and 6.10), and on the right hand side of the assignment command (5.2). The form

    <expression or name> repname <number>

may be used in place of a list element in any of these contexts. It means the same as if you had written out <number> list elements, each identical to <expression or name>.
For example,

    let foo(x, nil repname 40) be ...

is a convenient way to define a routine (see 6.10) which can have up to 41 parameters, and in which you want to deal with a vector of their Values, rather than each parameter by a unique name. The "NumbArgs" function in TENEX BCPL (see appendix C.10) can be used to determine how many parameters were given in a call on "foo", and the expression

    lv x

can be used as a "vector", whose elements (starting from the zeroth) are the Values of the parameters to "foo".

5. Commands

5.1 Simple Assignment Commands

Syntactic form:

E1 := E2

Semantics:

E1 may either be the name of a variable, a vector application, an rv expression, a half word extraction expression, a quarter word extraction expression, or a structure reference, and its effect is as follows:

(a) If E1 is the name of a variable: The assignment replaces the Value of the variable with the Value of E2.

(b) If E1 is a vector application: The Value of the storage cell referenced by E1 is changed to be the Value of E2.

(c) If E1 is an rv expression (indirect addressing):

the Value of the operand of rv is interpreted as an Address; the Value of E2 then is stored in the storage cell having this Address.

(d) If E1 is a half word expression (only rh and lh are allowed):

```
        rh E3 := E4
    is syntactic sugar for
        E3 := lh E3,, E4
```

and

```
        lh E3 := E4
    is syntactic sugar for
        E3 := E4,, E3
```

(See section 4.6)

(e) If E1 is a quarter word extraction expression (only q1, q2, q3, and q4 are allowed),

```
        q2 E3 := E4
```

causes quarter 2 of E3 to be replaced by quarter 1 of E4. Quarters are numbered from right to left. See section 4.1.14.

(f) For assignment to structure references, see section 7.

5.2 Assignment Commands

Syntactic form:

L1, L2, ... Ln := R1, R2, ... Rn

Semantics:

The semantics of the assignment command is defined in terms of the simple assignment command; the command given above is semantically equivalent to the following sequence, except that the order in which the simple assignments are done is not specified.

```
L1 := R1
L2 := R2
...
Ln := Rn
```

5.3 Routine Calls

Syntactic form:

E1 (E2, E3, ... En)

where E1 is a name or a parenthesized expression.

Semantics:

The above command is executed by assigning the Values of E2, E3, ... En to the first n - 1 formal parameters of the routine whose Value is the Value of E1; this routine is then entered. The execution of this command is complete when the execution of the routine body is complete (see 6.9 and 6.10).

5.4 Labelled Commands

Syntactic form:

N: C

where N is a name.

Semantics:

This declares a manifest constant which is associated with name N; its scope (see 6.1) is the smallest textually enclosing routine or function body (see 6.9 and 6.10) or block (see 5.18) and its Value is a bit pattern representing the program positon of the command C.

5.5 goto Commands

Syntactic form:

goto E

Semantics:

E is evaluated to yield a Value, then execution is resumed at the statement whose label has the same Value.

5.6 if Commands

Syntactic form:

if E do C

also:

if E then C

Semantics:

The Value of E is interpreted as a Boolean Value. See section 4.1.5 for the representation of Boolean Values. If E is true, C is executed. If E is false, C is not executed. If the Value of E represents neither true nor false then the effect is implementation dependent. If E is evaluable at compile time, then C is compiled without any run time check of E if E is true; no code is compiled if E is false. Similar appropriate compile-time analysis is done for other commands.

5.7 unless Commands

Syntactic form:

unless E do C

Semantics:

For a boolean expression E, this statement is exactly equivalent to the following:

```
if not (E) do C
```

5.8 while Commands

Syntactic form:

while E do C

Semantics:

This is equivalent to the following sequence:

```
   goto L
M: C
L: if E then goto M
```

where L and M represent internally generated names.

5.9 until Commands

Syntactic form:

```
until E do C
```

Semantics:

This statement is equivalent to:

```
while not (E) do C
```

5.10 test Commands

Syntactic form:

```
test E then C1 or C2
```

also:

```
test E ifso C1 ifnot C2
```

also:

```
test E ifnot C2 ifso C1
```

Semantics:

This statement is equivalent to the following sequence:

```
    if not (E) goto L
    C1
    goto M
L:  C2
M:
```

where L and M represent internally generated names.

5.11 Repeated Commands

Syntactic form:

```
C repeat
```

or

```
C repeatwhile E
```

or

```
C repeatuntil E
```

Where C is any command other than an if, unless, until, while, test, or for command.

Semantics:

C repeat is equivalent to:

```
L: C
   goto L
```

C repeatwhile E is equivalent to:

```
L: C
   if E then goto L
```

C repeatuntil E is equivalent to:

```
L: C
   if not (E) then goto L
```

where L represents an internally generated name.

the repeatwhile command differs from the while command in that the repeatwhile loop test is performed after executing the body of code at least once. The same relation exists between repeatuntil and until.

5.12 for Commands

Syntactic form:

```
   for N := E1 to E2 do C
```

also:

```
   for N := E1 by E3 to E2 do C
```

also:

```
   for N := E1 to E2 by E3 do C
```

where N is a name. NOTE: step may be used as a synonym for by.

Semantics:

The above statement is equivalent to:

```
{ let N := E1
  let END := E2
  until N gr END do
  { C
  N := N + E3 }
}
```

"by E3" or "step E3" is optional; the increment is assumed to be 1 if not specified. END represents an internally generated name. If specified, the expression E3 must be evaluable at compile time. Note that the for command is an implicit block (see 5.18 and 6.1)

5.13 switchon Commands

Syntactic form:

switchon E into { <list of cases> }

where each "case" in the <list of cases> is a "case label" followed by a sequence of commands. A "case label" has the form:

case <constant>:

or

case <constant1> to <constant2>:

or

default:

NOTE: If you want to inject a declaration into the sequence of commands in a "case", then make a block (see 5.18) out of the declaration and the relevant sub-sequence of commands [CAVEAT!].

Semantics:

The expression is first evaluated and if a case exists which has a constant with the same arithmetic Value then execution is resumed at that label; otherwise, if there is a default label then execution is continued from there, and if there is not, execution is resumed just after the end of the switchon command.

The switch is implemented as a direct switch, a sequential search or a tree search depending on the number and range of the case constants.
The case label

case E1 to E2:

is equivalent to

case E1: case (E1+1): case (E1+2): ... case E2:

where E2 must not be less than E1.
NOTE: branchon is a synonym for switchon.

5.14 loop, break, and endcase Commands

Syntactic form:

```
loop

break

endcase
```

Examples:

1.
```
for i := 1 to v|0 do
{ let x := v|i
  if x = 0 then loop
  - - -
  - - -
L1:
}
```

2.
```
until j = 0 do
{ if A > CaseK|j then break
  CaseK|(j+1) := CaseK|j
  CaseL|(j+1) := CaseL|j
  j := j - 1
}
L2:
```

3.
```
switchon Op into
{ case SWITCHON: Transwitch (x) ; endcase
  case SEQ:   Trans (x|1) ; endcase
  default:    Trans(x|2) ; endcase
}
L3:
```

Semantics:

The loop command causes a jump to a program point just inside the smallest enclosing loop, so that the end condition is tested and the loop repeated as required. In a for command the loop command also causes the index to be incremented before the test is made (as usual). In the first example, this is the program point labelled L1. Execution of the break command causes a jump to the point just after the smallest textually enclosing loop introduced by one of the following reserved words: until, while, repeat, repeatwhile, repeatuntil and for. In the second example, this is the program point labelled L2. The endcase command causes a jump to the program point just after the smallest textually enclosing switchon command. In the third example, this is the program point labelled L3.

5.15 finish Command

Syntactic form:

finish

Semantics:

This causes the execution of the program to cease (HALTF on TENEX).

5.16 return Commands

Syntactic form:

return

Semantics:

This causes the execution of the smallest enclosing routine body (see 6.10) to cease and return.

5.17 resultis Commands

Syntactic form:

resultis E

Semantics:

This causes execution of the smallest enclosing valof expression (see 4.1.8) to cease and return the value of E.

5.18 Sections and Blocks

Syntactic form:

{ <command or declaration>1<;<command or declaration>>0 }

(Note: The semicolon can be omitted between commands that appear on separate lines (see Appendix A.3). Square brackets may be used in place of curly brackets if desired. A matched pair of brackets may be given the name IMMEDIATELY ADJACENT to the right of the bracket. This is useful for documentation and error checking. Unless you intend to do this, brackets should be followed by space, tab, or carriage return (CAVEAT!!).)

Semantics:

A "section" is a sequence of BCPL commands that is enclosed in brackets (brackets are called "section brackets" in BCPL). Labels declared inside a section may be referenced from outside the section; e.g., the program is allowed to jump (goto) into the body of an if command. A "block"is a section in which there are declarations. Labels declared inside a block may not be referenced from outside the block. A section or block is executed by performing the declarations (if any) and commands in sequence. Within a block, the scope of the definee of a declaration is the region of program consisting of the declaration itself, and the succeeding declarations and commands.

6. Definitions

6.1 Scope Rules

The SCOPE of a name N is the textual region of program throughout which N refers to the same "data item" (either a variable or a manifest constant). Every occurrence (i.e. use) of a name must be in the scope of a declaration of the same name.

There are three kinds of declaration:

(1) Each element of the formal parameter list of a function or routine: its scope is the function or routine body (see 6.9 and 6.10).

(2) A label set by colon in a block: its scope is the block.

(3) Each declaration in a block: its scope is the region of program consisting of the declaration itself and the succeeding declarations and commands of the block.

Two data items are said to be declared at the same level of definition if they were declared in the same formal parameter list, as labels of the same block, or in the same declarations.

There are three semantic restrictions concerning scope rules, namely:

(a) Two data items with the same name may not be declared at the same level of definition.

(b) If a name N is used but not declared within the body of a function or routine, then it must either be a declared manifest constant or a static variable: that is, it must have been declared as external, global, an explicit function or routine, or a static. This restriction on functions and routines has been imposed in order to achieve a very efficient recursive call. In terms of the implementation, this restriction states that either the Value or the Address of every "free variable" of a function or routine is known prior to execution.

Note that the following program is illegal:

```
let a,b := 1,2
let f(x) := a*x + b
```

However, it may be corrected as follows:

```
static { a := 1; b := 2 }
let f(x) := a*x + b
```

(c) A label set by colon may not occur within the scope of a data item with the same name if that data item was

declared within the scope of the label and was not an external or global.

Declarations are permitted intermixed with statements. The rule is that a declaration may follow any semicolon, or may follow any sequence of labels which follows a semicolon. The scope of such a declaration is to the end of the smallest enclosing section or block. Note that it is not the case that a declaration may appear anywhere that a label may. (For example, an arm of a conditional may be labelled but it may not be a declaration.) Since a declaration introduces a block, it follows that labels and case labels that appear after it are not accessible from outside it (CAVEAT!).

6.2 Space Allocation and Extent of Variables

The EXTENT of a variable is the time through which it exists and has an Address. Throughout the extent of a variable, its Address remains constant and its Value is changed only by assignment.

In BCPL, variables can be divided into two classes,

(1) Static variables:

Those variables whose extent lasts as long as the program execution time. Every static variable must have been declared either in a function or routine definition, or in an external, global, or static declaration. For static variables that are initialized to tables or vecs, the space for the table or vec has the same extent as the static variable.

(2) Dynamic variables:

Those variables whose extent is limited; the extent of a dynamic variable starts when its declaration is executed and continues until execution leaves the scope of the declaration. Every dynamic variable must be declared either by a simple declaration, a vector declaration or as a formal parameter.

6.3 Externals

Syntactic form:

external { <name> <;<name>>Ø}

Semantics:

The external declaration declares a set of names (6 character length limit in TENEX BCPL for each such name) to be used in common by separately compiled programs. For each such name, exactly one program must declare the name as a function, routine, or static variable. Within the program

where the name is defined, it must also appear in an external declaration. Within a program where the name is used, it must appear in an external declaration. The programs that use the name should be loaded with the program that defines the name, otherwise the loader will complain.

6.4 Globals

Syntactic form:

global { <name>:<number> <;<name>:<number>>0}

(Note: "colon equals" (:=) may be used in place of : in global, static, and manifest declarations)

Semantics:

Globals are very similar to externals, except that numbers are used to identify them. Global numbers are to be allocated by the user. In TENEX BCPL, he may use numbers between #400 and #1377. The numbers between 0 and #377 and between #1400 and #1777 are reserved for the libraries. Globals exist in TENEX BCPL in addition to externals only because the number of characters in the name of an external on TENEX is limited to 6; the number of characters in a global name is the same as for any BCPL name (less than 24).

6.5 Statics

Syntactic form:

static { <name>:<constant>
<; <name>:<constant> >0 }

Semantics:

This declares each name to have an initial Value equal to the Value of the specified constant expression. Expressions composed of constants and the operators

+ - * / $ " ´ table vec ,, rem

are allowable. When used in this context, vec denotes a static vector.

6.6 Manifests

Syntactic form:

manifest { <name> : <constant>
<;<name> : <constant> >Ø }

Semantics:

This declares each name to be a manifest constant with a Value equal to the Value of the specified constant expression. The meaning of a program would remain unchanged if all occurrences of manifest named constants were textually replaced by their corresponding Values.

6.7 Simple Variables

Syntactic form:

let N1, N2,..., Nn:=E1, E2,..., En

Semantics:

Dynamic variables with names N1 ... Nn are first declared, but not initialized, and then the following assignment command is executed

N1, N2,..., Nn:=E1, E2,..., En

6.8 Vectors

Syntactic form:

let N1, N2,..., Nn := vec E1, vec E2,..., vec En

where the Ni are names.

Semantics:

Processing is similar to 6.7, above. The Ei's must be expressions which can be evaluated at compile time. Each of these defines the maximum allowable subscript of the corresponding vector. The minimum subscript is always zero. The initial Value of each Ni is the Address of the zeroth element of the vector; the Ni are dynamic variables. The vector subscripting operation is described in section 4.1.1Ø.

6.9 Functions

Syntactic form:

let N(<list of names, separated by commas>):=E

where N is a name.

Semantics:

This defines a function and a static variable with name N whose conceptual type is "function". The

static variable N has its Value initialized (prior to execution of the program) to the memory location of the start of the compiled code for the "function body" (E). Syntactically, E can be any expression. N defines an external if it is in the scope of an external declaration for N, or a global if it is in the scope of a global declaration for N. The names in the name list (if any) are called formal parameters and their scope is the function body (E). Each is a variable which is initialized to the Value of the corresponding parameter in the call (see 4.1.9). The extent of a formal parameter lasts from the moment of its initialization in a call until the time when the evaluation of the body is complete. The Value of the function application expression is the Value of E. All functions and routines may be defined and used recursively. Function applications are described in section 4.1.9.

6.10 Routines

Syntactic form:

let N(<list of names, separated by commas>) be C

where N is a name.

Semantics:

This defines a routine with name N. A routine declaration is like a function declaration except that the body of a routine is a command and therefore its application may not be used as an expression. A routine should therefore only be called in the context of a command. A function may be called either as an expression or as a command. Routine calls are described in section 5.3.

6.11 Simultaneous Definitions

Syntactic form:

let D <and D>0

NOTE: ...and let... is allowed

Semantics:

All the declarations are effectively executed simultaneously and all the defined variables have the same scope which includes the simultaneous definition itself; a set of mutually recursive functions and routines may thus be declared.

7. Structures

7.1 Introduction

An important problem in programming has to do with accessing and changing subfields of structured data. Here the term "structured data" refers to any collection of data -- that is, of bits -- which has some structure meaningful to one or more programs. As a simple example, a compiler is concerned with the instruction format of the object computer. Specifically, on the PDP-10, the 36-bit instruction word is divided into bit fields as follows (from left to right)...

op	9	operation code
ac	4	accumulator spec
d	1	indirect (defer) bit
x	4	index register specification
ad	18	address

Here for each field we give a one or two character name, the width of the field in bits, and its function (which is irrelevant to the present discussion). Now consider a compiler written to compile code for this machine. If jj is a variable containing the index register desired, the command

```
w := (w & #777760777777) \ ((jj & #17) lshift 18)
```

might be used to set the index part of w. If y is a pointer to an instruction, then the command

```
rv y := (rv y & #777760777777) \ ((jj & #17) lshift 18)
```

might be used instead. It would clearly be desirable to be able to program this operation in a more transparent manner. It is this sort of problem that the structure definition facility described in this section helps to alleviate.

Let us continue with the above example. In the syntax about to be described, the instruction format given above might be described by the following structure declaration:

```
structure
  { instruction
    { op     bit 9 //operation code
      ac     bit 4 //accumulator spec
      d      bit 1 //indirect address
      x      bit 4 //index register spec
      ad     bit 18 //address
    }
  }
```

This declaration defines the name "instruction" as being associated with a structure, the structure being composed of fields of

bits as shown. The dot is used to indicate sub-structure, so that

```
instruction.x
```

refers to the x-part (that is, the index part) of an instruction. We can then refer to the index part of the word pointed to by y as

```
y >> instruction.x
```

The mark ">>", which may be read "right lump", has been selected because of its resemblance to a pointer. It indicates that we are concerned with a subfield of a word pointed to. If instead w were the actual word in question instead of being a pointer to that word, we might write

```
w << instruction.x
```

(the "<<" may be read as "left lump"). The statements given earlier might then be written as

```
w << instruction.x := jj

y >> instruction.x := jj
```

respectively. These forms are more readable. Similarly, the "indirect" bit of the instruction pointed to by p could be set to one by executing

```
p >> instruction.d := 1
```

The structure facility in BCPL permits convenient access to and changing of subfields of Values. An important advantage of the facility is that the description of data bases can be separated (in seperate "get" files fetched by the compiler--see Appendix A.3) from the code that manipulates them. The idea is to specify the "shape" of a data item -- its representation as a bit pattern in memory. The shape of a data item is, in general, distinct from its use.

7.2 Syntax

Three structure constructs are included in the language -- the <structure declaration>, the <structure reference>, and the size expression. The first may be used wherever a declaration may be and serves to declare that a particular name references a particular structure, or shape. The second may be used wherever a primary expression may be used and serves to access a structured item. The last may be used to compute (as a constant expression) the size in bits of a structure.

BNF syntax follows, using the usual notation. Names of syntactic classes are enclosed in angle brackets, the vertical bar "|" is the meta-linguistic OR, and ":::=" means "is defined to be". All other characters stand for themselves.

Syntax for structure declarations:

```
<structure declaration>  ::=
        structure { <sd-list> }
<sd-list> ::=
        <sd-item>  |  <sd-item> ; <sd-list>
<sd-item> ::=
        <sd-term> | <sd-term> overlay <sd-item>
<sd-term> ::=
        <name> <replicator> <declarator> <size>
        <declarator> <size>
        fill <declarator>
        <name> <replicator> ; { <sd-list> }
        { <sd-list>}
<replicator> ::=
        ^ <constant expression>
        ^ <constant expression> ^ <constant expression>
        <empty>
<declarator> ::=
        bit | bitn | bitb | byte | byten | char | word
<size> ::=
        <constant expression> | <empty>
```

Syntax for structure references:

```
<structure reference> ::=
        <expression> >> <sri>
        <expression> << <sri>
<sri> ::=
        <src> | <src> . <sri>
<src> ::=
        <name> | <name> ^ <expression>
```

In addition, add to the definition of <constant expression> the possibility

```
        size <sri>
```

The syntactic categories left undefined in this syntax are

<expression>	any expression
<constant expression>	any expression whose Value can be deduced at compile time
<name>	any name

7.3 Semantics

A <structure declaration> is a list of <sd-item>s, separated by semicolons. (The semicolons missing from the examples shown throughout this document would be inserted automatically by the compiler.) Each <sd-item> is one or more <sd-term>s, separated by overlays. Ignoring this possibility for the moment, assume an <sd-item> to be an <sd-term>. The interest comes in an <sd-term>, of

which there are five flavors. As an example of the first, consider

x bit 5

this specifies a field named "x" which is 5 bits wide. The <sd-term>

y^3 bit 7

specifies three replications of field "y", each replication being 7 bits wide. The "up arrow" indicates a structure subscripting operator, used in both declarations and references. The three instances of y would be referred to as y^1, y^2 and y^3. Note that the first has "subscript" 1, as opposed to the usual BCPL subscription convention in which the first item has "subscript" zero. The difference between structure subscripting and regular BCPL subscripting is emphasized by using different characters. Now consider the declaration

z^0^2 bit 7

This uses as much space as the previous example, but the fields are to be referred to as z^0, z^1 and z^2. We see then that if the replicator is absent, it is taken as one. If one value is given (as for y above), it is taken as the upper limit with the lower limit taken as one. If two fields are given, they are taken as the lower and upper limits, respectively.

Consider now the classes <declarator> and <size>. The keywords bit, byte and word are self-explanatory, although the number of bits in a byte and of bytes in a word are of course implementation dependent. On TENEX, both char and byte represent 9-bit fields. In most implementations char would be the same as byte but they may be different. (Even if they are identical, the programmer may find it convenient to think of some items as char and some as byte.) The declarators bitn and byten refer to numeric fields. When referenced, they are taken as signed quantities and treated as appropriate in the implementation. For example, in a computer using one's complement or two's complement arithmetic the leftmost bit of the field would be copied into all bits of the word to the left of the field. The declarator bitb signifies a Boolean field and is permitted only for fields of width one. Accessing such a field yields either true or false.

An <sd-term> such as <declarator> <size> may be used to leave an unnamed field of the given width. Such fields may straddle word boundaries, even though named fields may not.

The <sd-term>s fill byte and fill word represent fields wide enough to go to the next byte boundary or word boundary, respectively. No name is associated with these, as they are used only to insure that the next field starts on an appropriate boundary. Since a subfield may not extend over a word boundary, this is frequently necessary. Use of fill frequently permits a given declaration to be used on

different computers with different word lengths.

Note carefully the restriction alluded to above: a named field is not permitted to extend over a word boundary. Thus the declaration

structure { a { b bit 27 ; c bit 27 } }

is improper on a machine with a 36-bit word, since a.c extends over a word boundary. Also illegal is

structure { a^2 bit 27 }

since a^2 is bad. There is no restriction about extending over byte boundaries, so the declaration

structure { a { b bit 5 ; c byte } }

is correct. Another way to look at this is that the <sd-term> byte is synonymous with the <sd-term> bit 9 (on TENEX), and

structure { a { b bit 5 ; c bit 9 } }

is clearly acceptable. Also acceptable is

structure { a char 3; char 2; b char 3 }

on a computer with 4 char's per word, since the field that straddles the word boundary is unnamed.

The <size> specifies the width of the field, in units of the <declarator>. If missing it is taken as one. Thus byte 3 refers to a field three bytes wide. The <size> may be any expression that can be evaluated at compile time.

All that remains to be explained is the keyword overlay. Two <sd-term>s separated by overlay are to occupy the same storage. Consider, for example, the declaration

structure { a { b byte 2; c byte 2 overlay cn byten 2 } }

The reference x<<a.b refers to the left half of x, and x<<a.c refers to x's right half. (This example assumes four bytes per word.) However, x<<a.cn interprets x's right half as a numeric quantity, so that it would be accessed with sign extension. That is, c and cn refer to the same part of the structure. Consider another example, in which we assume four 9-bit bytes per word:

structure { a { b byte 2; c^6 bit 3 overlay d^3 bit 6 } }

Here the right half of the word is to be regarded as either three 6-bit fields or six 3-bit fields.

7.4 Examples

Following are some examples of <structure declaration>s, with comments on their effect. The examples assume a 36-bit word with four 9-bit bytes per word, as on TENEX.

A string in BCPL on TENEX is stored four characters per word, with the length (in characters) stored in the first (leftmost) byte position. (The BCPL convention for structures is that byte positions are counted from left to right.) Then a declaration for such a structure is

```
structure { string { n byte; c^511 char }}
```

With this declaration, the length in bytes of string x may be referred to as x>>string.n, and the 4-th character of x as x>>string.c^4. The number 511 in the declaration comes from the fact that the maximum string length must be storable in 9 bits. The maximum length of a string in words is given by the expression

```
(size string)/36
```

(the parentheses are not needed.) This expression has value 128 (given the above declaration of string on TENEX) and is capable of being evaluated at compile time, so that

```
let v = vec (size string) / 36
```

is permissable. Note that the structure declaration of string would be less useful had n been declared to be a numeric field with byten rather than byte. In that case the left-most bit would be interpreted as a sign bit, so the possible values storable in a 9-bit field would be from -256 to 255. Since negative string lengths seem uninteresting, declaring the field to be a byte field gives the more useful range of 0 to 511.

Sometimes it is convenient to store strings one character per word. A useful format is to put the length in the zero-th word of a vector and the characters in successive words. Routines for unpacking and packing strings are then like this:

```
let unpackstring(s, v) be    // unpack string s into vector v
{ v|0 := s >> string.n       // the length of the string
  for k := 1 to v|0 do v|k := s >> string.c^k }

and packstring(v, s) be      // pack vector v into string s
{   s >> string.n := v|0
   for k := 1 to v|0 do s>>string.c^k := v|k }
```

Note the rather pleasant symmetry between these two routines.

Here is a routine that reverses the bits of a word:

```
let reverse(x) := valof
{  structure { b^0^35 bit 1 } // 36 1-bit items
   let t := 0
   for n := 0 to 35 do t << b^n := x << b^(35-n)
   resultis t }
```

The value of the function is the bit-reverse of its input.

REFERENCES

[1] Strachey, C. (Editor) "CPL Working Papers" a technical report, London Institute of Computer Science and the University Mathematical Laboratory, Cambridge (1966).

[2] Richards, M. "The BCPL Reference Manual", Project MAC Memo M-352-1, M.I.T. Cambridge, Mass. (Feb. 1968).

[3] Richards, M. "BCPL: A tool for Compiler Writing and System Programming", 1969 Spring Joint Computer Conference.

[4] The TENEX JSYS Manual

APPENDICES

A. BCPL Characteristics

A.1 Reserved Words and Symbols

The reserved words and symbols of BCPL are implementation dependent: they depend on the character set that is available. To simplify the transfer of BCPL from one machine to another, a set of "canonical symbols" has been developed. Each implementation of the BCPL compiler has a preprocessor which translates the reserved words and symbols for that implementation into the canonical symbols. The "canonical representation" of a BCPL program consists of a sequence of canonical symbols.

The names of the canonical symbols are given below together with corresponding examples of how they are represented in TENEX BCPL. The list of words and symbols under 'TENEX Form' includes the list of TENEX BCPL reserved words and symbols.

Canonical Symbol	TENEX Form	Described in Section
number	1Ø3 #777 3.56	4.1.1
name	abc i H2	4.1.4
stringconst	'xyz*n' "p"	4.1.3
charconst	$a $3	4.1.2
true	true	4.1.5
false	false	4.1.5
nil	nil	4.1.6
valof	valof	4.1.8
lv	lv	3.1, 4.1.11
rv	rv	3.1, 4.1.12
lh	lh	4.1.13
rh	rh	4.1.13
lhz	lhz	4.1.13
rhz	rhz	4.1.13
mult	*	4.2
div	/	4.2
rem	rem	4.2
plus	+	4.2
minus	-	4.2
fplus	%+	4.2
fminus	%-	4.2
fmult	%*	4.2
fdiv	%/	4.2
eq	eq or =	4.3
get	get	A.3
size	size	7.2
offset	offset	7.2
ne	ne	4.3
ls	ls or < or lt	4.3
gt	gt or > or gr	4.3

ge	ge	4.3
le	le	4.3
not	not or ~	4.5
lshift	lshift	4.4
rshift	rshift	4.4
lscale	lscale	4.4
rscale	rscale	4.4
logand	logand or &	4.5
logor	logor or \	4.5
eqv	eqv	4.5
neqv	neqv or xor	4.5
cond	=> or ->	4.7
comma	,	4.7
table	table	4.8.1
list	list	4.8.2
repname	repname	4.10
and	and	6.10
ass	:=	5.1
goto	goto	5.5
resultis	resultis	5.17
colon	:	5.4
test	test	5.10
ifso	ifso	5.10
ifnot	ifnot	5.10
for	for	5.12
if	if	5.6
unless	unless	5.7
while	while	5.8
until	until	5.9
repeat	repeat	5.11
repeatwhile	repeatwhile	5.11
repeatuntil	repeatuntil	5.11
loop	loop	5.14
break	break	5.14
return	return	5.16
finish	finish	5.15
switchon	switchon	5.13
branchon	branchon	5.13
selecton	selecton	4.9
case	case	5.13,4.9
default	default	5.13,4.9
endcase	endcase	5.14
let	let	6.2
manifest	manifest	6.5
static	static	6.4
external	external	6.3
global	global	6.11
be	be	7.2
sectbra	{ or [	5.18
sectket	} or]	5.18
rbra	(	4.1.7
rket	)	4.1.7
structure	structure	7

char	char	7
fill	fill	7
word	word	7
overlay	overlay	7
bit	bit	7
bitb	bitb	7
bitn	bitn	7
byte	byte	7
byten	byten	7
semicolon	;	5
into	into	5.13
to	to	5.12
by	by or step	5.12
do	do or then	5.12
or	or	5.10
vec	vec	6.7
vecap	| or !	4.1.10
uplump	^	7.2
leftlump	<<	7.2
rightlump	>>	7.2
dot	.	7.2

A.2 The TENEX BCPL Character Set

Code(Octal)	Char	Usage in BCPL source program
000	^@	Null ... Ignored as if it weren't there
001,006	^A,^F	Illegal
007	^G	Bell ... Illegal
010	^H	Backspace ... Illegal
011	^I	Tab ... Ignorable, like space
012	^J	Line feed ... Taken as "End Of Line"
013	^K	Vertical tab ... Illegal
014	^L	Form Feed ... Ignorable
015	^M	Carriage Return ... Ignorable
016,031	^N,^Y	Illegal
032	^Z	[Terminate input stream to the compiler] EOF
033	Altmode	Illegal
034,036	^\,^],^^	Illegal
037	^_	TENEX EOL, BCPL "*n", taken as "End Of Line"
040	Space	Ignorable
041	!	VECAP
042	"	Quote Character for BCPL strings
043	#	Octal number prefix
044	$	Character constant "quote"
045	%	Floating-point operation prefix %+ %- %/ %*
046	&	Logical AND operator
047	'	Quote character for ASCIZ strings
050	(	Expression parenthesis
051	)	Expression parenthesis
052	*	Integer multiply operator
053	+	Integer add operator

054	,	COMMA
055	-	Integer subtract operator
056	.	Structure operator and decimal point for floating point numbers
057	/	Integer divide operator
060,071	0,9	Digits
072	:	COLON (for labels)
073	;	SEMICOLON
074	<	LS operator
075	=	EQ operator
076	>	GR operator
077	?	Illegal
100	@	If /U, character or word upper case escape char otherwise, ignored as if it weren't there (see B.2)
101,132	A,Z	Uppercase letters (mapped to lower case, if /U)
133	[	Optional SECTBRA
134	\	Logical OR operator
135	]	Optional SECTKET
136	^	UPLUMP (structure operator)
137	_	part of the name character alphabet
140	`	(grave) Illegal
141,172	a,z	Lowercase letters
173	{	SECTBRA
174	|	VECAP
175	}	SECTKET
176	~	Logical NOT operator
177	Rubout	Illegal (BCPL "*r")

Escape conventions for non-printing characters and control characters in character and string constants are defined for TENEX BCPL.

Example:

```
$*s   represents space
$**   represents *
$^a   represents control a
$*^   represents ^
```

A complete description of these conventions follows:

^x where x in { [,\,],^,a, ... ,z } => that control character

*n => code 37, TENEX EOL (new line)
*r => code 177, Rubout
*s => code 40, Space
*t => code 11, Tab
*b => code 10, Backspace
*p => code 14, "Page", form feed

```
*f => code 14, form feed
*v => code 13, vertical tab
*<Three Octal digits> => octal escape
*c => code 15, carriage return
*l => code 12, line feed
*^ => code 136, ^
*" => code 42, "
*' => code 47, '
** => code 52, *
*$ => altmode
*a => code 100, @
*e => code 777, end of stream
*d => code 0, @ (NULL, dummy character)
```

A.3 The BCPL Preprocessor

The Preprocessor is the name of the part of the BCPL compiler which transforms the raw source text of a program into canonical symbols. The conventions in the TENEX version are as follows:

(a) A name is any sequence of upper or lower case letters and digits, starting with a letter, which is not a reserved word. The character immediately following a name may not be a letter or digit. A name may be no longer than 23 characters. All reserved words are strings of two or more lowercase letters.

(b) User's comments may be included in a program between a double slash '//' and the end of the line. Example:

```
let Factorial(n) := valof
 { // This function returns the factorial of
   //   its argument.

   if n = 1 do resultis 1
   resultis n * Factorial(n-1)
 }
```

(c) For documentation purposes, section brackets may be tagged with a name or integer. CAVEAT: Section bracket tags are detected as a name or integer which is immediately adjacent to the bracket. Thus, section brackets which are not tagged must be separated from a following letter or digit by space, tab, or carriage return.

(d) The canonical symbol semicolon is inserted by the preprocessor between pairs of canonical symbols if they appear on different lines and if the first is from the set of canonical symbols which may end a command or definition, namely:

```
break return finish repeat rket endcase loop nil
sectket name stringconst number true false charconst
```

and the second is from the set of canonical symbols which may start a command, namely:

```
test for if unless until while goto resultis
case default break return finish sectbra
switchon endcase loop selecton branchon
charconst not lhz rhz number strinconst
rbra valof rv name rh lh ql q2 q3 q4
```

(e) The canonical symbol "do" is inserted by the preprocessor between pairs of canonical symbols if they appear on the same line and if the first is from the set of canonical symbols which may end an expression, namely:

```
rket sectket name number
stringconst true false charconst nil
```

and the second is from the set of canonical symbols which must start a command, namely:

```
test for if unless until while goto
resultis endcase loop
case default break return finish switchon branchon
```

(f) A directive of the form:

```
get <specifier>
```

may be used anywhere in a BCPL program; it directs the compiler to replace the directive with the file (of text) referred to by the specifier. The form of the specifier is a string constant (the file name: see the example program in Appendix B).

(g) Pseudo commands
There are three special commands to the lexical analysis component of the compiler. These should be prefixed by two colons on a new line, and ended by carriage return.

1. ::reserve <word>,<word>,...
   This causes the indicated word(s) to be marked as "reserved" in the compiler's dictionary. This is useful when new reserved words are added to the language; existing programs that use such as variables should not be made obsolete by such changes to the language. Users who know about such changes and want to use the new features can do so by using the reserve command, which has effect until the end of the program, or until the next unreserve command for the word(s).

2. ::unreserve <word>,<word>,...
   Marks the indicated reserved word(s) as unreserved. Has effect until the end of the program, or until the

next reserve command for the word(s).

3. ::synonym <word1> <word2>
For use only with reserved words. Makes <word1> a synonym to the reserved word <word2>. If <word2> is marked as unreserved, <word1> will nevertheless be marked as reserved.

A.4 Subtle Features for note by new users (CAVEAT)

1. Upper and lower case alphabetic characters are distinct, unless you use the /U switch in the compiler's command line. In particular, the "Start" routine should be so spelled.

2. A section bracket ({ and }) should be followed by space, tab, or carriage return unless you mean to label it (see 5.18).

3. If inside of a switchon case body you desire to use let, you should enclose the case body in section brackets (see 5.13).

4. The Value of a string or a vector is a pointer to the zeroth cell of the string or vector.

5. The left half of the Value of a routine, function, or label is the JRST op-code.

6. Especially for FORTRAN Devotees: Arguments are passed "by Value" in BCPL ; iN particular, a routine can't change the Value of a variable that is passed as an argument.

A.5 Operator Precedence

BCPL Operator Precedence:

There are 2 numbers associated with each (binary) operator, to determine both its binding power with despect to operators to its left, and its binding power with respect to operators to its right.

In what follows, operator classes are indicated by angle brackets.

CLASSES

<eqv>::= eqv|neqv|equiv|nequiv

<lshift>::= lshift|rshift|lscale|rscale

<=>::= ne|<|>|le|ge|%<|%>|%=

<+>::= +|-|%+|%-

<*>::= *|/|rem|%*|%/

<repname>::= repname|repval

<lh>::= lv|rv
lh≠|rh≠|lh|rh
q1≠|q1
q2≠|q2
q3≠|q3
q4≠|q4

<list>::= list|table

<· means keep scanning to the right (i.e. call Rexp recursively)

·> means reduce (i.e. return from Rexp)

Example:

```
a + b lshift 5 | c & d
```

from the chart,

```
+ ·> lshift;   =>  (a+b)  lshift....

lshift ·> |;   =>  ((a+b) lshift 5)...

| <·   &   ;   ·>  ...(c & d)
```

and

```
((a+b) lshift 5) | (c & d)
```

		<eqv.>	/	&	<lshift>	<=>	<+>	<*>	vecap	<repname>	comma	cond.	;;	other [e.g. do]
↓ binops	<eqv.> (22,22)	>	<	<	<	<	<	<	<	>	>	>	>	>
	\ (23,22)	>	<	<	<	<	<	<	<	>	>	>	>	>
	& (24,23)	>	>	<	<	<	<	<	<	>	>	>	>	>
	<lshift> (25,30)	>	>	>	>	>	<	<	<	>	>	>	>	>
	<=> (30,30)	>	>	>	>	>	<	<	<	>	>	>	>	>
	<+> (34,34)	>	>	>	>	>	>	<	<	>	>	>	>	>
	<*> (35,34)	>	>	>	>	>	>	<	<	>	>	>	>	>
	vecap (40,40)	>	>	>	>	>	>	>	>	>	>	>	>	>
	<repname> (12,12)	<	<	<	<	<	<	<	<	>	>	<	<	>
	comma (12,11)	<	<	<	<	<	<	<	<	<	<	<	<	>
	cond. (13,12)	<	<	<	<	<	<	<	<	>	>	<	<	>
	;; (14,13)	<	<	<	<	<	<	<	<	>	>	>	<	>
↓ unops	<lh> (0,35)	>	>	>	>	>	>	>	<	>	>	>	>	>
	vec (0,30)	>	>	>	>	>	<	<	<	>	>	>	>	>
	not (0,25)	>	>	>	>	<	<	<	<	>	>	>	>	>
	<list> (0,11)	<	<	<	<	<	<	<	<	<	<	<	<	>

$[op(N1,N2)]$

rule: ... op1 ... op2 ...

$op1_{N2} < op2_{N1} \rightarrow$ op1 < op2 else op1 > op2

B. Usage of TENEX BCPL

B.1 Typical Source File Organization

By convention, the file name extension for BCPL source files is .BCP . A BCPL source file normally starts with a comment which describes the contents of the file. Following this, there are usually declarations of externals, globals, statics, and manifests which are to be in effect during the compilation of the source file. For convenience, collections of standard declarations are often kept in separate text files (example: GLOBAL declarations for the I/O library are contained in <BCPL>HEAD.BCP); the "get" command tells the compiler to insert the text from a specified file into the source, and behave as if this text were a part of the source. Thus, the "declaration portion" of the BCPL source file often contains "get" commands. See the example program, below.

B.2 Using the Compiler

The BCPL compiler is the subsystem named "BCPL.SAV". Its primary job is to translate a BCPL source file into a TENEX .REL file. Other jobs that it does include the generation of a MACRO listing of the program, and the generation of a specially-formatted symbol table for the program. Typing "?" will cause the compiler to explain its command line format. The CCL subsystem will select the BCPL compiler for compilation of .BCP files.

B.3 Constructing a BCPL Main Program

Compile a BCPL source program which contains the definition of a routine named "Start", with "Start" declared as GLOBAL #1 (as in <BCPL>HEAD.BCP). Then load the resulting .REL file (and any others). The BCPL library (<SUBSYS>BCPLIB.REL) will be searched automatically. Starting the resulting core image will cause the routine named "Start" to be called. Returning from "Start" will cause the program to terminate (HALTF).

B.4 Routine and Function Linkage Conventions

1. Calling Sequence

```
                call:
...             ;CODE TO STUFF ARGS
                ;AND NUMBER OF ARGS
                ;INTO THE NEW STACK FRAME
ADDI 16,n       ;MOVE THE STACK POINTER
                ;TO THE NEW STACK FRAME
JSP 1,subr      ;CALL THE ROUTINE OR FUNCTION
SUBI 16,n       ;MOVE THE STACK POINTER
                ;BACK TO THE OLD STACK
                ;FRAME
```

at routine or function entry:

```
MOVEM 1,Ø (16) ;SAVE THE RETURN POINTER
               ;IN THE CURRENT STACK FRAME
```

Routine or function return:

```
JRST 2,@Ø(16) ;RESTORE FLAGS AND RETURN
```

2. On entry to a routine or function, the number of arguments is expected to be in 1(16)

3. The first argument is expected to be in 2(16)
   The kth argument is expected to be in k+1(16)

4. Only register 16 is expected to be preserved across a routine or function invocation.

5. n is the size of the current stack frame.
   This is equal to
   2+
   number of args declared in the currently active routine or function +
   number of registers for local variables on the stack in the currently active routine or function when the call is made.

6. subr is the address of a register which contains a JRST instruction to the first instruction of the routine or function being called (In BCPL, the Value of a label, routine, or function is such a JRST instruction).

7. The Value of a function call is returned in AC1.

B.5 Utility Programs

a. <BCPL>FMT.SAV

This is a program which formats a BCPL source file. WARNING: the source file should have no syntactic errors. The program is experimental; there are pathological cases which cause it to mess up. Use it at your peril. (We find it quite useful).

b. <BCPL>OCODE.SAV

This is a program which prints out the compiler's intermediate output code for the idealized object machine in a human-readable form. The program requires a .O file as its input; the compiler will produce this if the /O switch is specified for the compilation.

c. <BCPL>PSYMB.SAV

This program prints out the symbol table (i.e. the .S file) in a human-readable form. Its output is to the file named <pgm name>.SYMTAB.

d. <BCPL>PSAVE.SAV

This program prints out useful information about an SSAVE'd file in which there are BCPL .REL files.

e. <BCPL>CONC.SAV

This program generates a concordance (CREF) for one or more BCPL source files.

B.6. A Complete, Realistic, Working Example Program

A programming example which demonstrates a simple application of recursion is known as the "8 Queens Problem". The problem requires the placement of eight queens on a standard 8x8 chessboard in a configuration such that no queen threatens any other queen.

The recursive solution to this problem is a function which:

1. Assumes that on entry, there is a queen in each column to the left, but there are no conflicts.

2. Tries to place a queen in each row of the current column, failing when it conflicts with any of the queens in previous columns.

3. For each success above, calls itself recursively to iterate over the next column to the right. This will discover all non-conflicting configurations to the right (printing them as solutions) before returning.

The argument of the function is the current column. The data structures needed for bookkeeping are vectors indicating that some queen is already placed in a particular row, a particular upward-diagonal, or a particular downward-diagonal, so the attempt to place another queen in the same row/updiagonal/downdiagonal results in a conflict. In addition, a solution vector is needed for type-out: this specifies which row the queen is in for each column.

The following example illustrates the use of comments, the labeling of section brackets, the use of the get declaration to include other files in the compilation, the static declaration to allocate storage, routine definitions, and the use of several library functions (WriteN, WriteS) for typeout. The block structure, the long identifiers, (up to 23 upper/lower case characters) and the mnemonic operators all contribute to program readability.

```
// Solution of 8 Queens Problem
get "<BCPL>HEAD.BCP"
get "<BCPL>UTILHEAD.BCP"          // Link to I/O subroutine Library
static
{ Solutions: nil                  // Total number of legal solutions
  Row:       vec 7                // Array to remember the col
                                  // the Queen is in for each row 0-7
  Horiz:     vec 7                // True if a queen is in
                                  // the Horizontal row 0-7
  Updiag:    vec 14               // True if a Queen is in
                                  // the Upward Diagonal 0-14
  Downdiag:  vec 14               // True if a Queen is in
                                  // the Downward Diagonal 0-14
 }
let Start() be
{st
  for I:=0 to 7                    // Each Horiz row is empty
  do Horiz|I:=false
  for I:=0 to 14                   // And each diagonal is empty
  do { Updiag|I:=false; Downdiag|I:=false }
  Solutions:=0                     // No solutions yet
  Queens(0)                       // Types out all solutions
  WriteS("*n Number of Solutions= ")
  WriteN(Solutions)               // Summary
 }st
and Queens(Col) be // There are no conflicts in previous columns
{qn
  let Updiag2,Downdiag2:=Updiag+7-Col,Downdiag+Col
  for N:=0 to 7 do // Try to put a Queen in each row of this column
  unless (Horiz|N  Updiag2|N  Downdiag2|N) do
        // No conflict with queens to left
  { Row|Col:=N                     // Remember where for typeout
    test Col=7                    // Check for all done
    ifso
    { WriteS("*n")                // Legal Solution
      for Col:=0 to 7 do { WriteN(Row|Col); WriteS("*s") }
      Solutions:=Solutions+1
     }
    ifnot
    { Horiz|N:=true                // Place a Queen there
      Updiag2|N:=true
      Downdiag2|N:=true
      Queens(Col+1)               // Find all legal configs
                                  // in cols to the right
      Horiz|N:=false               // Now remove Queen
      Updiag2|N:=false
      Downdiag2|N:=false
     }
   }
 }qn
```

APPENDIX C. Functions, Routines, and Special Static Variables in the TENEX BCPL Library

C.1 I/O

C.1.1 I/O Streams

There are two kinds of "BCPL I/O streams": JFN streams and function streams. JFN streams are simply TENEX JFN's. Function streams are functions which are specified by the user to be either a source of bytes (for input) or a sink (for output). We are working on adding "string streams". The global declarations are in <BCPL>HEAD.BCP and <BCPL>UTILHEAD.BCP.

FindInput(Desc,bytesize)
CreateOutput(Desc,bytesize)

bytesize: (optional argument) assumed to be 7 (bits) if not specified.

Desc:

-f: use function or routine (f) for I/O. For each character operation, f will be called. The first arg to f will be the byte size. The char will be the second arg (output only).

Ø: primary input/output JFN

1: ask user for string at run time

2: expect string at run time from primary input file, but don't prompt the user.

s: s is a string ; do GTJFN

The Values of FindInput and CreateOutput are zero if an error occurs, JFN's for the opened files for cases Ø, 1, 2, and s, and a 36 bit number for case -f as follows:

q4 = -1 if read, -2 if write
q3 = bytesize
right half = rhz(-f)

EndRead(stream,lefthalf)
EndWrite(stream,lefthalf)
These "close" the specified "stream". In both EndRead and EndWrite, the second argument is optional. If present, it is used as the left half of AC1 in the CLOSF call (for JFN streams). If absent, Ø is used.

CLOSF(jfn) does just that.

INPUT
The default input stream (used by PBIN, for example) (initialized to the primary input stream for this process)

OUTPUT
The default output stream (used by PBOUT, for example) (initialized to the primary output stream for this process)

EofFlg
A static variable which is set by BIN, PBIN, Readch, and SIN. Set to true if an EOF is encountered while reading bytes, set to false otherwise.

rfptr(jfn)
returns the byte pointer ala jsys RFPTR or a negative number (the negative error number from the JSYS)

sfptr(jfn,byte ptr)
sets the byte pointer, ala jsys SFPTR. The first byte in a file has byte pointer=Ø.
Value: a negative number (the negative of the JSYS error number) if the JSYS fails, zero otherwise.

IsCharInput(input stream)
returns true if there is another input character available on the specified input stream. (uses SIBE for JFN streams)

EchoMode(boolean)
turns on or off keyboard echoing, ala the argument (true => on).

C.1.2 Character, Word, and String I/O

The global declarations are in <BCPL>HEAD.BCP and <BCPL>UTILHEAD.BCP.

BIN(stream)
Read a byte from the specified input stream.

Value: the byte read.
Note that if BIN (or PBIN or Readch or SIN) reads past End Of File, the character code returned is #777, and EofFlg is set true, otherwise EofFlg is set false.

PBIN()
Read a byte from the primary input stream.
Value: the byte read. Note: PBIN() is equivalent to BIN(INPUT)

BOUT(stream,Byte)
Write a byte on a specified output stream.

PBOUT(Byte)
Write a byte on the primary output stream.
PBOUT(Byte) is equivalent to BOUT(OUTPUT,Byte)

Readch(stream, lv Ch) is equivalent to
Ch := BIN(stream)

Writech(stream, Ch) (same as BOUT),

For Readch and Writech, if there is only one argument, it is assumed to be a character to either read from INPUT or write on OUTPUT.

SIN(stream,TENEX string ptr, bytecount, termbyte)
bytecount:
Ø => Ø byte terminates
>Ø => bytecount
termbyte: optional argument ; present => use it for terminating byte, if it occurs before the byte count is exhausted.

Value: if bytecount > Ø , Value is the number of bytes transferred. Otherwise, Value is the revised TENEX string pointer.

SOUT(stream,TENEX string ptr, bytecount, termbyte)
same as SIN, but for output

WriteS(String) or WriteS(stream,String).
Write a BCPL string. Former case uses primary output stream (OUTPUT).

ReadWord(instream, strng, chlv, skipbool, termstring)
reads a word from the specified stream (instream) as delimited by "terminator characters" (as specified by the 5th argument) into the specified buffer. Editting via ^A, ^Q, and ^R is implemented.
If only one arg, INPUT is used as the instream, and the specified arg is used as "strng".
If lh strng < Ø, the word goes into the buffer pointed to by rh strng in unpacked BCPL string format:
[[(rh strng)|Ø=# chars ; (rh strng)|i=iTH char]]
Otherwise, the word goes into strng in the packed BCPL string format. The (optional) third argument specifies the Address of a variable into which to store the character which terminated the word. If four or more arguments are given, and the fourth is false, then ReadWord returns whenever it reads a terminator. Otherwise, Readword skips over word-initial terminators.
The (optional) fifth argument is a string specifying the set of terminator characters. If this argument is absent, the function IsTerminator() is used.
The Value of a call on ReadWord is (rhz strng).

IsTerminator(ch)
returns a Boolean: true if the char is *s *t *c *l or *n
false otherwise.

C.1.3 Integer and Floating Point I/O

The global declarations are in <BCPL>HEAD.BCP and <BCPL>UTILHEAD.BCP.

WriteN(number)
writes the (integer) "number" (in decimal) on the primary output stream.
WriteN(stream,number) does it to the specified stream.

WriteOct(...)
similar to WriteN, but radix 8

WriteR(stream,n,AC3)
Write a single precision floating point number
1 arg: n: OUTPUT assumed as stream and Ø assumed as AC3 to FLOUT
2 args: Ø assumed as AC3 to FLOUT
Value: error code (for ERSTR) if an error is detected; otherwise Value is -1.
A more elaborate formatted output facility is described in C.1.5.

ReadN(instream)
calls ReadWord, (with skipbool = true) then TxtToInt. If the argument is missing, INPUT is used.

C.1.4 Network Interface

CreateNetDialogue
FindNetInput
CreateNetOutput
LISTENING
NETLOCALSOCKET
NETINOPENF2
NETOUTOPENF2
NETWAITTIME
NETPOLLTIME
NETWAITFLAG
LocalSocket
NetStatus
EndNetDialogue
InitNetLibrary

This section describes a package of subroutines for doing ARPA Network I/O from BCPL programs, or from programs which can interface to the BCPL subroutine calling conventions. The (BCPL) source file for the subroutine definitions is <BCPL>NL.BCP. The BCPL head file with the GLOBAL declarations is <BCPL>NLHEAD.BCP. The REL file to be loaded with your calling program is <BCPL>NETLIB.REL. The subroutines work and have been used to implement several programs which use facilities at Lincoln TX-2 and transfer files both ways between BBN-TENEX and TX-2. Note that reference is made in the documentation to "error numbers" which are returned from the subroutines when they fail for some reason. These aren't described here.

SUBROUTINES:

I. CreateNetDialogue

CreateNetDialogue(foreign host, outstream-lv, instream-lv, localsocket, foreign socket)

foreign host: either a BCPL string (the host name) in the right half and -1 in the left half, or the host number. If left half is negative, then right half is taken as a pointer to a BCPL string.

outstreamlv and instreamlv: addresses of storage cells in which the two new stream identifiers are to be stored.

localsocket: a local socket number (must be even). This argument is optional. If it is missing or negative, one is made up.

foreign socket: 1 (for logger) assumed if this argument is missing.

Returns Ø if successful, error number otherwise. If successful, 8-bit send and receive TTY connections

are opened to the given foreign host.

II. CreateNetOutput and FindNetInput

Arguments (two options):

A. For normal connections:

(bytesize, foreign host, foreign socket, local socket, OPENF ac2, waittime, polltime)

1. bytesize

Either 8, 32, or 36 (BITS)

2. foreign host

Either a BCPL string (the host name) in the rh and -1 in the lh, or a host number.

3. foreign socket

An absolute foreign socket number.

The remaining arguments are optional. Casual or novice users can probably ignore them and the subsequent discussion under "For normal connections":

If an optional argument is omitted, the indicated global variable is used in its place:

1. local socket default variable: NETLOCALSOCKET

If specified, its form is to be

[directory number or Ø or -1],,[relative local socket number or -1]

-1 in LH means job relative
Ø in LH means local directory relative
>Ø in LH means other directory relative
-1 in RH means relative local socket number = 1Ø*JFN

5. OPENF ac2

default variable: Either NETINOPENF2 or NETOUTOPENF2, for input and output, respectively.

If specified, its form is to be as described in the TENEX JSYS Manual.

NOTE: The value of AC2 when OPENF is called by CreateNetOutput or FindNetInput includes the value of the first argument (bytesize). This is shoved

into the appropriate bits of the fifth argument if it is specified, or into the value of the appropriate OPENF2 global variable. The "data mode" bits allow one to specify whether and how to buffer messages (for efficient network utilization), and whether to wait for matching RFC or CLS. Note that NETWAITFLAG is NOT used to determine these bits. For more information, see the JSYS Manual.

6. waittime

   default variable: NETWAITTIME This is a number of milliseconds to wait before giving up the attempt to establish the connection.

7. polltime default variable: NETPOLLTIME This is the number of milliseconds to pause between attempts to establish the connection.

B. For LISTENING connections:

(bytesize, LISTENING, localsocket, OPENF ac2, waittime, polltime)

This is for opening a LISTENING connection.

1. The bytesize is either 8, 32, or 36 (BITS).

2. The second argument should be the MANIFEST constant named LISTENING.

The remaining arguments are optional, and are treated as described above.

Notes

1. Both functions return either a BCPL stream descriptor (>Ø), or a negative error number if they fail.

2. Normally, these functions wait until the connection gets established, as per the appropriate arguments or defaults. The functions may be caused to return immediately by setting the global variable named NETWAITFLAG to false. It is then the user's responsibility to check the status of the network stream before doing any I/O (see NetStatus). An example is the opening of a LISTENING connection. If NETWAITFLAG is false, the waittime and polltime arguments are meaningless.

Global Variables

1. NETLOCALSOCKET(initially [0,,-1])

2. NETINOPENF2(initially 6 -> bits 6 thru 9 (data mode: immediate return. see the TENEX JSYS Manual.) 10 -> bits 19 thru 22 (10 octal) (direction of connection))
3. NETOUTOPENF2(initially same as above, except 4 -> bits 19 thru 22)
4. NETWAITTIME(initially 20000) i.e. 20 seconds
5. NETPOLLTIME(initially 5000) i.e. 5 seconds
6. NETWAITFLAG(initially true)

Defined Constant (manifest) LISTENING := -1 a number distinct from any foreign host number, or from (-1,, BCPLstringptr)

III. LocalSocket

LocalSocket(network stream) returns the absolute local socket number for the specified stream if it succeeds, or the negative of the CVSKT JSYS error number if it fails.

IV. NetStatus

NetStatus(network stream,vector) returns status information ala JSYS 145 (GDSTS) in the vector. vector|1 is the connection state (see the document on the TENEX ARPANET SOFTWARE INTERFACE) vector|2 is the connection byte size vector|3 is the foreign host number vector|4 is the foreign socket number NetStatus returns its second argument as its value.

V. EndNetDialogue

EndNetDialogue(output stream, input stream) CLOSF the two JFN's, and wait for them to close

VI. InitNetLibrary

InitNetLibrary() initialize the global statics to their default values.

C.1.5 Formatted Output

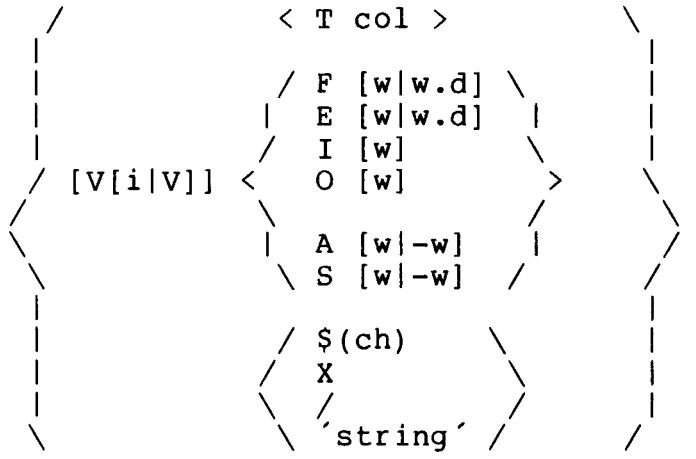
```
TypeF(formatstring,arg1,arg2,....)
PrintF(formatstring,arg1,arg2,....)
PWriteF(formatstring,arg1,arg2,....)
EndPrint()
WriteF(stream,formatstring,arg1,arg2,arg3,....)
OutputF(stream,formatstring,argvec,nvalues)
```

The global declarations are in <BCPL>HEAD.BCP.

These routines are intended to be similar to FORTRAN WRITE statements. Stream is as usual in BCPL I/O. In TypeF, #101 is assumed for the output stream. PWriteF assumes the static OUTPUT as the stream (like PBOUT). In PrintF, a printer stream is opened the first time it is called, (and is remembered in a local static). This stays open until all files are shut, or a reset is done, or the JFN is closed, or the printer stream is closed using Endprint(). PrintF is meant to mirror the operation of the FORTRAN PRINT statement. All of these routines call OutputF which takes as its arguments, the stream, the format string, a vector of values (the first in argvec|1) and the number of values (including do loop iteration values). OutputF can be called directly, if so desired.

The format string is a BCPL string. It is similar in form to the FORTRAN format statement with some deletions, some additions, and some modifications. The basic form of a single command field is as follows:

<FMT>::=



where upper case letters indicate those specific characters and lower case letters are used to indicate integer numbers. Symbols or subfields enclosed in [] are optional. Larger brackets means "applies to". A "|" between two optional subfields means "or". Using "<FMT>" for the basic command, the following ways are available for combining them into a Total Format ("<TFMT>"):

<TFMT>::=

[n] <FMT>
<TFMT>,<TFMT>
[n] (<TFMT>)

Where the format would not be ambiguous without the comma between fields it is not necessary, but it should be remembered that spaces are not delimiters. It is assumed that the reader is familiar with FORTRAN FORMAT statements and only differences will be discussed. (for a general description of FORTRAN FORMAT fields, see the brief description at the end of this section.)

T field: Will cause the next output to start at the column specified. The first column is column 1. This is accomplished by outputting spaces or a carriage return and spaces as is necessary.

F field: The free format floating point output (when w.d is not given) uses the BCPL routine WriteR which uses the FLOUT JSYS with ac3=Ø. It leaves no spaces. If w Is specified, and .d is not, then -1 is assumed. (d=-1 is taken to mean no decimal point, but at present this doesn't work and it works as if d = Ø)

E field: The exponent will always be given as E+- and two digits. There is, at present, no control over the exponent, so that the entire w columns will be filled (with the possible exception of the first if the number is positive). This may at some date be changed so that there is one and only one integer digit given, with spaces to the left. Free E format is E14.7 format or: -i.ddddddE+ee or *si.ddddddE-ee

I,O field: Integer (radix 1Ø and 8, respectively). There are no spaces with Free format.

ASCIZ strings (A field) and BCPL strings (S field) are significantly different from FORTRAN. If width is not specified, the entire string is output.
w-width can be positive or negative. If abs(w) is less than the length of the string, then the first abs(w) characters of the string are output. If +w is greater than the length, the entire string is output, right-justified to w columns, filled to the left with spaces. If -w is greater than the length, the string is output left-justified, and padded to the right with spaces to make up w columns.

' - literal field: Any literal string of BCPL characters terminated by a single quote ('). To include a single quote in the string, use two successive single quotes(''). To include a double quote use (*")

$ - single character field: The next BCPL character in the compiled string will be taken as a literal and output. (special case of a literal field)

X - field: Output a space. (special case of single character field)

/ - field: Outputs an EOL. (special case of single character field)

V - vector field: If a V precedes a data field type, then instead of using the value in the argument list as a value, it is used as a pointer to a block of values. If i is specified, the i successive values will be output with the following field specification. The default is 1. This counts as one value output with respect to the format statement and argument list. If instead of an integer i, the next character is an upper or lower case V, this specifies a "variable vector length": the value after the pointer value is taken as an integer and used as i (using up one of the arguments, of course).

Iteration - Matching parentheses can be nested around parts of the format string. If an integer is specified, this has the effect of writing out the enclosed part n times. If a positive integer precedes a field with no intervening left parenthesis, there are implied parentheses around that one field.

Termination rules - A comma (or right parenthesis-comma) is expected after all fields, so that a typical properly formed string would be:

"2(I5, 2x,f),/,3x,s"

However, it is generally allowed for the comma to be omitted except where it must serve as a delimiter between two numbers. Therefore

"2(I5, 2xf)/3xs"

would accomplish the same goal. Notice that the comma between the 5 and the 2 is necessary, otherwise, it would be interpreted as

"2(I52, x,f),/,3x,s"

even though there is a space. Commas (or spaces) may be found to make the format string more readable, but this is left up to the user.

General rules - All parentheses must be matched. All spaces are ignored (except within literals) and therefore are not valid as delimiters. All field type specifications and the V can be upper or lower case.

Error Handling - In the case of column overflow, free format is used. Errors in the format string are detected at run time. An error message will be output to the primary output stream (OUTPUT), followed by the words "Format Error", followed by the compiled string with "*^*" inserted immediately following the character that is thought to be wrong. The program then finishes (HALTF). Typing CONTINUE to the EXEC will cause execution to continue immediately after the call to WriteF. (This will soon be changed to use the ERRSET facility in BCPL).

Execution - continues until:

a) The format string is exhausted. If there are still more

values left in the argument list, then processing will continue at the last left parenthesis on the same level, if there are any. If the last non-space is not a right parenthesis, then execution will continue from the beginning of the format string.
The following examples show where execution will continue if there are more values:

```
"F7.4, I5,2(3V2i2,1x, 2(s1Ø,/))"

"F7.4,i5, 2(3Vvi2,3x,2(s-1Ø, /) ), I "

" x, 2(I5, ´ Here´´s one*n´), t2Ø, (f1Ø,I2)"
```

b) A field requiring a value is encountered, and the argument list has been exhausted. One value is used each time a value-taking specification is executed in the expanded string. The vector field iteration counts as 1 field no matter how many values are used from the value block. If the variable vector length is used, then an extra value is used to get the number of values to be output from the block.

Examples:
I) If there were two vectors of data x and y and a program were comparing different values of the two arrays, then the call to WriteF and the corresponding output for 2 executions might look like this:

```
WriteF(stream, "´Test Case No. ´,I,/, 2(I5,f6.Ø,/),/", i, j,x|j,
 k,y|k)
```

Output

```
Test Case No. 9
   24 3454.
  142  42Ø.

Test Case No. 1Ø
  1Ø5-7523.
    7  -45.
```

Here is what the following formatstring would produce with the same data:

```
"´*nComparison *# ´,i2, (t2Ø,i,t26,f,/)"
```

Output

```
Comparison #  9      24     3454.
                     142    420.

Comparison # 10      105    -7523.
                     7      -45.
```

II) In order to output a vector or part of vector, which would be done in FORTRAN by:

```
        WRITE(1,10) (X(I), I = K,L)
10      FORMAT(1X, F6.0)
```

Output

```
  34.5
 -20.7
2473.0
3650.0
 .
 .
etc.
```

Using WriteF this would be:

```
WriteF(stream,"x,Vvf6.1,/", x+k, k-1+1)
```

III) To put out two vectors and the subscript simultaneously:

FORTRAN

```
       WRITE(1,10) (I,X(I),I,Y(I), I = 5,10)
10     FORMAT(' X(',I2,') =',F7.4,2X,'Y(',I2,') =',F6.2)
```

Output

```
X( 9) = 0.3425  Y( 9) = 12.45
X(10) = 0.8739  Y(10) = -7.02
```

Using WriteF this would be:

```
for i := 5 to 10 do
 WriteF(stream,"'x(',i2,') =',f7.4,2x,'y(',i2,') =',f6.2,/",i, x|i, i,
 y|i)
```

Relevent FORTRAN format rules.

C.2 JSYS Interface

The global declaration is in <BCPL>HEAD.BCP.

JSYS(JsysNumber, InputACs, OutputACs)
Perform a JSYS call.
JsysNumber: just that. The file <BCPL>JSHEAD.BCP has a set of manifest declarations for the JSYS names.

InputACs: A vector (of at least 10 cells) having the values of the input AC's to the JSYS.
v|1 equals AC1, v|2 equals AC2, etc.

OutputACs: A vector (of at least 10 cells) for the values of the output AC's from the JSYS.

JSYS(n,v) is equivalent to JSYS(n,v,v)
JSYS(n) allowed also (some JSYS's don't require parameters).

Value: the number of instructions skipped plus one.

NOTE: There is a "get" file of manifest declarations for JSYS names: <BCPL>JSHEAD.BCP, in which each JSYS is named by prefixing "js" to the JSYS name (e.g. jsGTJFN).

C.3 Byte Manipulation

The global declarations are in <BCPL>HEAD.BCP.

POINT(Size,Location,RightmostBit)
Construct a PDP-10 byte pointer.

Size: the byte size (number of bits in a byte)

Location: the Address of the cell containing the byte

RightmostBit: the bit position in the cell of the right-most bit in the byte. Bits in a cell are numbered from 0 to 35, from left to right. This argument may be absent; if so, it is assumed that you mean the first (leftmost) byte in the indicated word (Location). POINT is meant to be used like the POINT Pseudo-op in MACRO-10.

LDB(BytePtr)
Extract a byte. Value is the byte.

DPB(Byte, BytePtr)
Deposit the specified byte.

ILDB(BytePtrLV)
Increment the byte pointer and then extract a byte. Value is the byte.

BytePtrLV: The Address of a cell which contains the byte pointer.

IDPB(Byte, BytePtrLV)
Increment the byte pointer and then deposit the specified byte.

IBP(BytePtrLV)
Increment the byte pointer.

C.4 String Manipulation and Number Conversion

Packstring
Unpackstring
StringToASCIZ
ASCIZToString
Egstr
TxtToInt
findsubstr
scanuntil
changesubstr
scanpastst
pullch
putch
addch
append
inttotxt
inttoocttxt

The global declarations are in <BCPL>HEAD.BCP, <BCPL>UTILHEAD.BCP, and <BCPL>STRINGHEAD.BCP.

Conventions:

Characters within a string are numbered starting at 1. the routines assume adequate storage for strings...e.g. in append, the output string is assumed large enough to hold the result. BCPL strings can be no bigger than 511 characters. Beware: NO checking for string buffer overflow or more than 511 characters is done.

Conversion routines for packed and unpacked BCPL strings:
Vector|0 is character count [[unpacked format]]
so is q4 (String|0) [[packed format]]
Vector|n is the nth character in the string [[unpacked format]]

Packstring
Packstring(Vector,String) returns the string

Unpackstring
Unpackstring(String,Vector)

Conversion routines for BCPL and ASCIZ strings:

StringToASCIZ
StringToASCIZ(BcplString,VectorForASCIZString) returns the vector.

ASCIZToString
ASCIZToString(ASCIZString,VectorforBcplString) returns the vector.

Eqstr

Eqstr(string1,string2) returns true if the two BCPL strings are equal, false otherwise

TxtToInt(string)

This subroutine converts the indicated text string into a number. Leading spaces or tabs are NOT allowed. The string may be prefixed by a minus sign or a # character or by both. The minus sign means negative, and the # character means octal (default case is decimal). If the lh of the argument is negative, the right half is used as the pointer to a vector, and the string is input unpacked (see Unpackstring).

findsubstr(str,substr,slv,elv,scn)

Find the indicated substring (substr) in the indicated string (str)- start searching at character number scn. If the substring is found, return the character number of its first character. in rv slv, and the character number of its last character in rv elv, and resultis true, else resultis false.

scanuntil(line,buffer,chnlv,c1,c2,c3, ...)

Search for one of (up to) 18 characters in a string starting at the indicated character position (rv chnlv). If "buffer" is non-zero, it is taken as a vector in which a BCPL string having the scanned characters is to be constructed. Resultis true if one of the indicated characters is found, false otherwise. rv chnlv is the character number of the located termination char. If none is found, rv chnlv is unchanged.

changesubstr(str,srchsubstr,newsubstr)

Change all instances of the indicated substring (srchsubstr) in the indicated string (str) to the indicated new substring (newsubstr). Resultis a ptr to the changed string (newsubstr).

scanpastst(line,chnlv)

Search the indicated string (line), starting at the indicated character,(rv chnlv). Return the character number of the first non-space or tab character in rv chnlv. It is assumed that "line" is a string which has at least one non-(space or tab) character.

pullch(txt,cp)

This fn returns the element (a character) at the specified character position (cp) within the specified string (txt).

putch(ch,txt,cp)

This procedure replaces the character at the specified character position (cp) within the specified string (txt) with the specified character (ch).

addch(ch, txt)

Append the specified character (ch) to the specified string (txt).

append(t1,t2,t3)

Append the string t2 to the string t1 and store result in the string t3. Any two or all three specified strings can be identical. The third argument is returned as the value of the function call.

inttotxt(i,txt)

inttoocttxt(i,txt)

These functions convert the indicated number into a text string in the indicated vector. "inttotxt" generates decimal equivalent, "inttooctxt" generates octal equivalent (i.e. preceded by a # character). If required, a minus sign appears. The second argument (txt) is returned as the value of the function call.

float(x)
convert the specified integer (x) to a floating point number. Value: this number.

fix(x)
truncate the specified floating point number (x). Value: the integer result.

fixr(x)
same as fix, but with round-off instead of truncation

C.5 Error Handling

The global declarations are in <BCPL>HEAD.BCP and <BCPL>UTILHEAD.BCP.

ErrSet(severity,resultlv,function,arg1,arg2,...)
A call on ErrSet establishes a point in your program's environment to which to return if an error occurs (see ERROR, below) during the call on the indicated function.

severity: a number to indicate how "severe" an error needs to be to cause a return to this ErrSet call.

resultlv: The Address of a variable into which to store the Value of the function call if the function returns without inducing an error. If this argument is zero, it is assumed that you don't want the Value of the function call.

function: The function to call

arg1,arg2,...: The arguments to the function.

The result of the call on ErrSet is true if the function call succeeded, and equal to the severity of the error otherwise.

ERROR(n)
A call on ERROR induces an unsuccessful return from the most recent call on ErrSet such that n is less than or equal to the severity of the ErrSet. The Value of the ErrSet call will be n.

Level()
Returns value of stackpointer for current environment. This is needed (for example) for LongJump and LongDebrk (in PSI pkg.).

LongJump(label,level)
Loads level into stackpointer (AC 16), then jumps to label.

ERSTR(ErrorNumber) or ERSTR(stream,errornumber)
Uses the ERSTR jsys. Arguments are handled like WriteN.

Help(string)
prints out the string, and a help message, and then HALTF's. CONTINUE will cause Help to return.

C.6 BCPL Array Package

Dimension

sub

ArrayCheck

The global declarations are in <BCPL>UTILHEAD.BCP.

A function (sub) is provided to compute a vector subscript for a multidimensional "array," given the subscripts and a "dope vector" describing the dimensions of the array. A function (Dimension) is also provided for forming dope vectors. Arrays are indexed so that for consecutive words in the vector, the 1st subscript varies most rapidly, as in FORTRAN. There is a limit of 10 on the number of dimensions.
Array bound checking is performed given either of two conditions:

1. Global variable ArrayCheck has the Value true (initially false).
2. The zeroth element of the dope vector is minus the number of dimensions. This can be accomplished by giving a negative first argument to the function "Dimension".

The function Dimension forms a dope vector. Arguments:

dopevec = a vector which will be stuffed with the dimension information. Must be of length 2*ndimensions+3
If dopevec is given as -vector, then the dope vector will be flagged to cause array bound checks to be done.

There follow pairs of minimum and maximum subscript values for as many dimensions as the array is to have.

The resulting dope vector has the following form:

```
ndims,     min1,     1,     min2,     (max1-min1+1),     min3,
(max1-min1+1)*(max2-min2+1)              ,...,              Nil,
(max1-min1+1)*...*(maxN-minN+1)
```

The function sub computes the vector element given the dope vector and the array subscript(s).

For example:

```
let foo,foodim:=vec 121,vec 7
Dimension(foodim,0,10,-5,5)
for i:=0 to 10 do for j:=-5 to 5 do
{ foo|sub(foodim,i,j):=i,,j}
```

C.7 Hash-Coded Dictionary

DictGetFree
DictRetFree
TBP
InitDict
RestoreDict
Enter
Find
NextDictEntry

The BCPL dictionary package is a collection of subroutines which are used to construct and maintain a hash-coded dictionary. This dictionary provides a mechanism for relating a specified BCPL string to a "dictionary entry" (via Enter and Find). This entry is a block of adjacent cells which has two parts: a header of at least one cell, followed (in memory) by enough cells to store the string. The right-most quarter of the zeroth cell of the header is used to store the number of cells in the header.

The "get" file of GLOBAL and EXTERNAL declarations for the dictionary package is <BCPL>DICTHEAD.BCP. The BCPL library contains the dictionary package; it will be loaded automatically with your program if you need it.

The dictionary package deals with relative pointers. The idea is that a dictionary may reside anywhere in memory. Indeed, the program can deal with several dictionaries, each residing in a different portion of memory, by specifying the base address for the relative pointers in the new dictionary of interest when a switch between dictionaries is made (via RestoreDict). Accordingly, when the user constructs and initializes a new dictionary (via InitDict), he specifies its base address; if he doesn't want to deal with relative pointers, the base address should be zero.

The dictionary package uses a free storage allocation mechanism which must be provided by the user. This leaves the user free to define his own free storage strategy, and decide from whence free storage cometh. There is a standard free storage allocation package for TENEX BCPL, described in section C.8 below.

The free storage mechanism for the dictionary is two subroutines, referenced as GLOBALs by the dictionary package, and expected to be defined by the user and loaded along with his program:

DictGetFree(n)
    should return a pointer to a block of n registers

DictRetFree(pointer)
    should put the indicated block of registers back into the free storage pool

The subroutines which are provided by the dictionary package are presented below:

InitDict(hashsize,offset)
purpose: create and initialize a new dictionary

hashsize: size of the primary hash table. For best results, this should be roughly 50% bigger than the number of entries expected.
offset: base address for dictionary pointers

Value: a relative pointer to the primary hash table (called "hashstart" below). (Note: InitDict will call DictGetFree to allocate storage for this table).

RestoreDict(hashsize,offset,hashstart)
purpose: reset the dictionary package to consider an old (previously initialized) dictionary. This is useful when dealing with more than one dictionary, when you want to switch between them.

hashstart: a relative pointer to the primary hash table

Value: hashsize.

Enter(wrd,address,datalength)
purpose: enter a given word in the dictionary
wrd: a pointer to a BCPL string

address: the address of a storage cell into which to store a relative pointer to the dictionary entry.

datalength: The number of cells to allocate (except for the rightmost quarter of the zeroth cell) for the header of the dictionary entry.

The Value is a Boolean:
true => it was found to be already entered
false => it was not found to be already entered.

Find(wrd,address)
purpose: find the dictionary entry for a given word

wrd,address: same as for Enter

The Value is the same as for Enter.

Delete(wrd,address)
purpose: delete the dictionary entry for a given word

wrd,address: as above

Value:
true => it was found and deleted
false => it was not found

NOTE: even though a pointer to the entry is returned in the specified storage cell, the storage for the entry will have been reclaimed (i.e. a call will have been made on DictRetFree) by the time Delete returns.

NextDictEntry(firstblv,entrylv)
purpose: find all the dictionary entries, one at a time, in an undefined order

firstblv: the Address of a storage cell which should contain true for the first call on NextDictEntry, and false for subsequent calls. NextDictEntry will set the Value of the storage cell to false before it returns the first time.

entrylv: the Address of a storage cell into which to store a relative pointer to the next dictionary entry. Meaningful only if the result of the call on NextDictEntry is true.

Value: true if there is a next entry, false otherwise.

Example: ["base" has the dictionary base address as its value]

```
// print out all entries in a dictionary
{ let b:=true
  let entry:=nil
  while NextDictEntry(lv b, lv entry) do
   { WriteS(base+entry+qlz base|entry)
     PBOUT($*n)
   }
}
```

(The following discussion is for users who want to replace the dictionary package's hash-coding algorithm with their own).

The dictionary subroutines hash-code the input BCPL string to yield an address into the "primary hash table". This table is marked to indicate (for each entry) whether it is full, and, ifso, where an overflow ("secondary") hash table is for that entry. The subroutine which does the hash coding is named TBP, is referenced as an EXTERNAL by the dictionary package, and a standard version of it is part of the BCPL library. The user is free to define his own TBP subroutine, and load it with his program. For a user who wants to do this, the specs for TBP are presented below:

TBP()
returns an address relative to the start of the hash table whose length is the value of GLOBAL #352, where a pointer to the input string is the value of GLOBAL #351, and where the value of GLOBAL #350 is the number of registers used to hold the string, minus 1. This equals: (q4 GL351|0)/4

C.8 Free Storage Allocation

GetBlock
RetBlock
ResetFreeStore
GetStorageSpace

This package is meant to manage memory allocation; the user specifies large regions of memory (via GetStorageSpace) that are to be carved up into blocks (via GetBlock) in response to his subsequent requests. No "garbage collection" is done; i.e. it is the user's responsibility to explicitly release blocks of storage when they are no longer needed (via RetBlock).

The BCPL free storage package consists of three subroutines:

1. GetBlock(n)
   returns a pointer to a block of n storage cells

2. RetBlock(pointer)
   reclaims a free storage block which was previously allocated by GetBlock

3. ResetFreeStore()
   This re-initializes the free storage routines.

The "get" file of GLOBAL declarations for the free storage package is

<BCPL>FREESTOREHEAD.BCP

The BCPL library contains the free storage package; it will be loaded automatically with your program if you need it.

The free storage package expects that the user will provide a subroutine for allocating big chunks of memory for its use:

4. GetStorageSpace(npages)
   Value should be a pointer to a new chunk of memory of the specified number of pages. Currently, the npages argument will be 16.

C.9 PSI System Interface

PSICHN
PSILEV
PSIPC
FNTBL
PSICHØ
PSISetCh
PSIOn
PSIOff
PSIClear
PSIChEnb
PSIChDis
PSIChInit
IsPSIChEnb
FreeTICh
ATI
DTI

The software support for use of the TENEX pseudo-interrupt system from a BCPL environment comes in two pieces. The first, named BPSI, is a machine language component which is necessary for PSI operation. The second, named PSI, is a set of BCPL functions and subroutines which interface to the pseudo-interrupt system JSYS's. Both packages reside in the BCPL library, and are loaded automatically when a BCPL program in which they are referenced is loaded. Such a program should "get" <BCPL>PSIHEAD.BCP. For a detailed discussion of the TENEX PSI system, see Chapter 5 of the TENEX JSYS Manual.

BPSI: the basic machine language package

The basic operation of a pseudo-interrupt is: When an interrupt occurs, a table is consulted for an address to which control is transferred. The section of program so selected saves any AC's it needs, carries out its processing, then either restores AC's and debreaks to the previous environment, or initializes any AC's it needs and debreaks to an arbitrary address. In a BCPL environment, this needs to get translated to: When an interrupt occurs, consult a table for the name of a BCPL subroutine, and cause it to be run. If it returns, restore to the previous environment. Also provide a means by which the interrupt can be debreaked (debroken?) and control transferred to some arbitrary label. These services are provided by the BPSI package.

BPSI contains the following, which are declared external in <BCPL>PSIHEAD.BCP:

a. PSICHN - PSICHN|i, i=Ø,1,...,35 is the PSI channel table.

b. PSILEV - PSILEV|i, i=1,2,3 is the PSI level table.

c. PSIPC - PSIPC|i, i=1,2,3 holds the PC for level i.

d. FNTBL - FNTBL|i, i=Ø,1,...,35, contains a JRST to the subroutine to be executed upon interrupt on channel i.

e. PSICHØ - An interrupt on channel i should transfer to location 4*i + lv PSICHØ.

In order to set up an interrupt on channel C, at level L, to execute routine R, do the following:

1. Set PSICHN|C to L,,4*C + lv PSICHØ.

2. Set FNTBL|C to R.

3. Enable PSI channel C using the AIC JSYS. If not already done, set up the entire PSI system with the SIR JSYS (AC2 = lv PSILEV|1,,lv PSICHN|Ø) and enable it with EIR.

When the interrupt occurs, the routine R will be called with three arguments:

1. Level of interrupt

2. Channel of interrupt

3. Lv of the PC storage word for this interrupt.

To return from the interrupt, the routine R merely returns. Of course, it can not return a value. To transfer to some label, with which is associated a level (stack pointer), call the routine LongDebrk(pclv,label,level), where:

pclv = Lv of the PC storage word

label = The label to be transferred to

level = The level associated with the label (may be obtained from the function Level() in <BCPL>UTIL.)

WARNING: The present implementation of BPSI contains the following fudge: when the interrupt subroutine is run, a new stack frame is made for it by adding 50000 to the existing one. This obviously will fail in cases where the current stack frame is very large. Beware of this until the BCPL code generator is modified to provide a solution to this potential bug. You have been warned!

PSI: the user routines

PSI contains a number of functions and subroutines which act as an interface between the user and the JSYS's which control the pseudo-interrupt system. Use of this package is not necessary for pseudo-interrupt usage, as is BPSI. The information given above is sufficient for the user to implement direct calls to the appropriate JSYS's. Although the functions and routines described here are intended to cover the most common types of pseudo-interrupt usage, they are not exhaustive or completely general. In order to realize the full flexibility of the TENEX pseudo-interrupt system, the user may have to supplement them with other direct JSYS calls. In particular these routines do not give access to the JSYS's RWM, SIRCM, RIRCM, STIW, or RTIW.

The default, where appropriate, is for all pseudo-interrupt operations to refer to the current fork. However, PSISetCh, PSIOn, PSIOff, IsPSIOn, PSIChEnb, PSIChDis, PSIChInit, and IsPSIChEnb can take a fork handle as an optional first argument in addition to the arguments shown below.

PSISetCh(level,channel,routine) - set up for an interrupt on the given level and channel, to dispatch to the given routine. Also enable the channel.

PSIOn() - declares the level and channel tables and enables the pseudo-interrupt system.

PSIOff() - disables the pseudo-interrupt system.

PSIClear() - clears all interrupts in process and all waiting interrupts.

PSIChEnb(channel1,channel2,...) - enables channel(s) given by the argument(s).

PSIChDis(channel1,channel2,...) - disables channel(s) given by the argument(s).

PSIChInit(channel1,channel2,...) - initiates interrupt(s) on the channel(s) given by the argument(s).

boolean:=IsPSIChEnb(channel) - returns true if the channel has been enabled.

value:=FreeTICh() - returns the number of a free channel which may be used for a terminal interrupt. terminates with a message if none is free.

ATI(character,channel) - assigns the character to cause interrupts on the given channel and sets the terminal interrupt word (nondeferred) for that character. Does not enable the channel. Terminates with a message if character is not a valid

terminal interrupt character.

DTI(character) - breaks the assignment of character to whatever channel it was assigned to.

Example program:

```
get "<BCPL>HEAD"
get "<BCPL>UTILHEAD"
get "<BCPL>PSIHEAD"

manifest{ rubout:= 177}
static { savedlabel:=nil; savedlevel:=nil}

let Start() be
{ OUTPUT:= 101
  let foo:=$A-1
  let rubchnl:=FreeTICh()  //Find a free channel for rubout int.
  ATI(rubout,rubchnl)  //Assign rubout to it
  PSISetCh(1,rubchnl,rubint)  //Level 1, that channel, to rubint
  PSIOn()
  savedlabel:=here
  savedlevel:=Level()
here:
  Writech($*n)  //Ridiculous program just
  foo:=foo+1  // to have something to do
  for i:=1 to 72 do  // til user types rubouts to
  { Writech(foo)  // show that PSI's work!
    Wait(2000)
  }
  goto here
}

and rubint(l,c,lvpc) be  //rubout causes this routine
{ WriteS("*sXXX")  // to be run
  LongDebrk(lvpc,savedlabel,savedlevel)
}
```

C.10 Miscellany

The global declarations are in <BCPL>HEAD.BCP and <BCPL>UTILHEAD.BCP.

InitACs
InitACs is a vector containing the initial AC's
InitACs|0 := AC0 on start-up ...
in addition, InitACs|16 := base of the runtime stack

ACCall(subr,v,v)
Similar to JSYS, except first argument (subr)is the Value of a subroutine which expects to be called via

```
PUSHJ 17,subr
```

and expects arguments in AC's

TLOCK(lv LockWord)
Does an "AOSE" (add 1 and skip if result equals zero) on LockWord, and returns false if the AOSE skips, true otherwise.

blt(from,to,last,safe)
from, to, and last are Addresses of:
the zero'th word of the source block
the zero'th word of the destination block, and
the last (N.B. NOT last+1) word of the destination block, respectively.
The fourth argument (safe) is optional. If present, a check is made for overlapped blt and the right thing is done if necessary

NumbArgs()
no arguments. The Value returned is the number of arguments in the most recent call on the function or routine which called NumbArgs.

Wait(number of milliseconds)

MakeDate(v,d)
builds a BCPL string of the specified date and time (d) (using the ODTIM JSYS)in the specified string (v). If d is missing, the current date and time is used. This also happens if d is zero. MakeDate returns its first argument.

Date()
returns the current date and time ala TENEX. (a 36 bit number).

min(arg1,arg2,arg3,...)
Returns the minimum of the arguments. Assumes args are either all integer or all floating point.

max(arg1,arg2,arg3,...)
as above, but maximum.

FORTRAN-BCPL Interface

FArg(type,arglv)

Returns a Value of the proper form for an argument to a FORTRAN function or subroutine (i.e. #320B9 + typeB12 + arglvB35). Argument types (cf. DEC FORTRAN manual, chapter 9) are:

0	integer	4	octal
1	(unused)	5	hollerith
2	real	6	double precision
3	logical	7	complex

Many FORTRAN routines do not check the type or presence of the #320B9, but they are necessary for others. FORTRAN arguments are passed by Address, so arglv must be an Address, not a Value. This means:

1. strings and vectors are passed as "themselves".
2. simple variables are passed by Address.
3. constants can't be passed. You must define a variable with the value of the constant, then pass its Address.

FCall(globalval, argvec, acc1lv)

Calls a FORTRAN function or subroutine whose entry address (which must be declared external) is given by globalval.

argvec|0 = number of arguments
argvec|1 = first argument, a la FArg
(etc.)
acc1lv is an optional argument, the Address of a variable into which to stuff the second word of a returned double precision or complex Value (which FORTRAN returns in AC1).
Example:

```
manifest { real:=2}
external { AMAX1}
let fargs:=list 2, FArg(real, lv a), FArg(real, lv b)
let amax:=FCall(lv AMAX1, fargs)
```

Note: FArg and FCall apply to the old DEC FORTRAN system (F40), not the new one (FORTRAN10).

Hint: For programs which use FCall often, the following function, which does not allow the use of the 3rd argument to FCall, may be useful:

```
let FFCall(routinelv, nil repname 30) := valof
{  let x := routinelv
```

```
    routinelv := NumbArgs()-1
    resultis FCall(x, lv routinelv)
}
```

This lets the example call above look like:

```
let amax := FFCall(lv AMAX1, FArg(real, lv  a),  FArg(real,  lv
b))
```

BCALL is a FORTRAN-callable function which allows a FORTRAN program to call a BCPL function or subroutine, providing:

1. The BCPL function's use of stack space does not exceed the fixed amount provided within BCALL (currently 5000 words)

2. The BCPL function accepts all arguments as Addresses, which is what FORTRAN delivers.

3. BCALL is not used recursively: FORTRAN world calling BCPL through BCALL, which calls more FORTRAN things through FCall, which calls still more BCPL stuff through BCALL... (this restriction will eventually be removed).

Usage:

```
CALL BCALL(BPROC,ARG1,ARG2,...)
REAL=BCALL(BPROC,ARG1,ARG2,...)
INTEGR=IBCALL(BPROC,ARG1,ARG2,...)
```

where:

BPROC=BCPL procedure, declared in an EXTERNAL statement
ARG1, etc.=arguments to BPROC.

Naturally the only BCPL procedures which can be called by name directly are those which have external declarations. Those declared as globals have to be called as GL147, etc.

D. TENEX BCPL Maker's Guide

There should be a <BCPL> directory and a <XBCPL> directory. The former is for the library sources, and HEAD files, and initialized binary data structures for the compiler (e.g. the initial dictionary and symbol table). The latter is for the compiler source files.

The compiler is
<SUBSYS>BCPL.SAV
The BCPL library is
<SUBSYS>BCPLIB.REL

To make a compiler: load <XBCPL>XBCPL.REL and ssave it on
<SUBSYS>BCPL.SAV

<XBCPL>XBCPL.REL is a FUDGE2 file which is composed of the following .REL files:

```
HANDCD        (hand coded stuff -- for efficiency)
NLEX1  (lexical analysis)
NLEX2
NMAIN1 (The main program and error handling stuff)
NMAIN2
NMAIN3
CAE0   (builds the parse tree)
CAE1
CAE2
CAE3
CAE4
TRN0   (translate the parse tree to 0CODE)
TRN1
TRN2
TRN3
TRN4
TRN5
TRN6
TRN7
TRN8
TRN9
CG0    (translate the 0CODE to a .REL file)
CG1
CG2
CG3
CG4
CG5
```

The following files should be in the <XBCPL> directory:

```
        BCPLERRORS.DOC  (text of compiler error messages --
                         specially formatted)
        CAE0.BCP
        CAE1.BCP
        CAE2.BCP
        CAE3.BCP
        CAE4.BCP
        CG0.BCP
        CG1.BCP
        CG2.BCP
        CG3.BCP
        CG4.BCP
        CG5.BCP
        HANDCD.MAC
        HEADBCPL.BCP      (head files used by the compiler sources)
        HEADCAE.BCP
        HEADCAECON.BCP
        HEADCG.BCP
        HEADJIMREAD.BCP
        HEADLEX.BCP
        HEADLEXCON.BCP
        HEADMAIN.BCP
        HEADSYMB.BCP
        HEADTRN.BCP
        HEADTRNCON.BCP
        INITDICT          (text file for compiler dictionary --
                           specially formatted)
        MERMSG.SAV
        MERMSG.BCP        (program to build ERRMSGS.BIN;1)
        MKCPDC.SAV        (program to build COMPDICT.SAV)
        MKCPDC.BCP
        NLEX1.BCP
        NLEX2.BCP
        NMAIN1.BCP
        NMAIN2.BCP
        NMAIN3.BCP
        OCODE.BCP
        OCODE.SAV         (utility program to convert the binary
                                 0CODE file to text)
        OCODETXT.BCP      ("get" file for 0CODE.BCP)
        OPCSPLNGS.BCP     (a "get" file of MACRO instruction names --

                           used by CG)
        PDPOPS.BCP        (a "get" file of PDP-10 0PCODES --
                           used by CG)
        TREE.BCP          (a "get" file of tree node names --
                            used by MAIN)
        TRN0.BCP
        TRN1.BCP
        TRN2.BCP
        TRN3.BCP
        TRN4.BCP
```

```
TRN5.BCP
TRN6.BCP
TRN7.BCP
TRN8.BCP
TRN9.BCP
XBCPL.REL          (the compiler REL file)
```

To make XBCPL.REL, assemble HANDCD.MAC, and compile (using <SUBSYS>BCPL.SAV) the BCPL programs, then use FUDGE2 to construct the big .REL file with /A.

The <BCPL> directory should have the following files:

```
BCALL.MAC     (FORTRAN-BCPL interface)
BCFAST.MAC    (hand coded library functions and routines)
BCMAIN.MAC    (the main program -- required if START is
                defined)
BCPLIB.REL    (the BCPL library -- a FUDGE2 file)
BDICT.BCP     (part of the hash coded dictionary package)
BPSI.MAC      (part of the PSI package)
COMPDICT.SAV  (the initial compiler symbol table -- binary)
CONC.BCP      (the CONC utility)
CONC.SAV      (make by compiling, loading, and SSAVing
                CONC.BCP)
DICT.REL      (a FUDGE2 file composed of BDICT.REL
               and TBP.REL)
DICTHEAD.BCP
ERRMSGS.BIN   (the compiler error messages -- binary)
ERRSET.MAC    (error handling package part)
FMT.SAV       (the BCPL source file formatting utility)
FMT1.BCP
FMT2.BCP
FMT3.BCP
FMT4.BCP
FMT5.BCP
FMT6.BCP
FMT7.BCP
FMT.CMD       (list of files to compile and load to make
                FMT.SAV)
FREESTOREHEAD.BCP
FRESTR.BCP    (the free storage package)
HEAD.BCP
HEADFMT.BCP
IOFMT.BCP     (the formatted I/O package)
IOLIBE.S
IOLIBE.BCP    (I/O package)
JSHEAD.BCP    (JSYS names and manifest definitions)
MANUAL.DOC      (the BCPL manual)
NETLIB.REL      (a FUDGE2 file composed of NL.REL
               and XNTLIB.REL)
NL.BCP        (part of the network interface package)
NLHEAD.BCP
PSAVE.SAV     (the PSAVE utility -- make by compiling and
              loading PSAVE.BCP)
PSAVE.BCP     (prints useful information about a SSAV file)
PSI.BCP       (the rest of the PSI package)
PSIHEAD.BCP
PSYMB.SAV
PSYMB.BCP     (the PSYMB utility)
STRING.BCP    (the string package)
STRING.REL
STRINGHEAD.BCP
TBP.MAC       (the rest of the hash-coded dictionary package)
UTIL.BCP      (goodies in the library)
UTIL.S
```

UTILHEAD.BCP
XNTLIB.MAC (the rest of the network interface package)

The compiler uses the following files at compile time:
    <BCPL>ERRMSGS.BIN;1
    <BCPL>COMPDICT.SAV

To make ERRMSGS.BIN;1
    Delete <BCPL>ERRMSGS.BIN;1 and expunge it. Connect to the <XBCPL> directory.
    Compile, load, and run MERMSG.BCP
        It will use <XBCPL>BCPLERRORS.DOC to create a new
            <BCPL>ERRMSGS.BIN;1
            This file is used to print error messages.

To make <BCPL>COMPDICT.SAV
    Connect to the <XBCPL> directory.
    Compile, load, and run <XBCPL>MKCPDC.BCP
        It will ask for the name of an input file.
        Give it: INITDICT<carriage return>.
        It will print out the octal address of the start of the dictionary, and the end. (start: #260000)
        Do an SSAV from 260 to 261 (or to the last used page) onto
            <BCPL>COMPDICT.SAV.
        This file is used during compiler initialization.

To make OCODE.SAV:
    Compile, load, and SSAV OCODE.BCP. The compilation will use OCODETXT.BCP as a "get" file.
    When run, OCODE.SAV will ask for a program root name. It assumes a .O file for that program exists (binary ocode file from the compiler...generated if you use the /O switch in the compilation) and makes a text file with the extension .OCODE

To make PSYMB.SAV:
    Compile, load, and SSAV it onto PSYMB.SAV
    This program generates a readable (text) version of the binary symbol table file (<rootname>.S file).

To make BCPLIB.REL
    Use FUDGE2 to make a library of the following .REL files, in the order specified:

        BCMAIN
        IOFMT
        FRESTR
        BDICT
        TBP
        PSI
        BPSI
        UTIL
        IOLIBE
        BCFAST
        BCALL
        ERRSET

APPENDIX E: Debugging

The BDDT subsystem on TENEX is used to examine and control the execution of BCPL programs. Two interesting features of BDDT are the isolation of the command language from the peculiarities of the machine on which the user's program runs, and the interface between BDDT and the user's program. Both of these features were designed to allow BDDT to run on TENEX and debug a BCPL program which runs either locally or on another machine on the ARPANET. BDDT is currently available on TENEX for debugging BCPL programs that run locally, and will soon be available for debugging BCPL programs that run on a remote TENEX or on a remote PDP-11.

E.1 How to invoke BDDT: The EXEC BDDT Command

The TENEX EXEC language has a BDDT command, similar to the IDDT command, which calls BDDT to debug a program. This command can be used in two ways: either before running a program, or after it has been interrupted by control-C. The first way is to tell the EXEC

```
@RESET
@BDDT
```

This will start running BDDT, which will ask for the name of a program to debug. The extension (if not given) is assumed to be .SAV. A variation on this method is to do a RESET, then GET the program (using the EXEC GET command), and then type BDDT. The second way is to give the BDDT command to the EXEC after stopping your program with Control-C. This will start BDDT, with the program loaded, and will act exactly as if BDDT had been used to run the program.

When entered, BDDT types out "!BDDT!", and attempts to load the symbol tables created by the BCPL compiler for each of the BCPL files in the program. In order to find these files, BDDT uses the symbol table created by the LOADER to find the name of the .REL files that were loaded. The symbol files are assumed to have a .S extension, and only the first 6 letters are used in the file name. If the BCPL file had more than 6 letters in its name BDDT may not be able to find its symbols. Symbols are not loaded for any part of the program for which the .S file is not found on the CONNECTed directory.

Once BDDT has been run using the EXEC BDDT command, you can Quit (see E.2.5 below) at any time and later (as long as you don't destroy your program) issue the BDDT command. Instead of loading a new version of BDDT, the EXEC will re-start the old version, so that the symbol files and breakpoints will not be lost (just like with IDDT). If you want to load a new version of BDDT you should Quit BDDT and use the NO BDDT command to the EXEC (you will thereby lose any breakpoints you may have set using BDDT).

The EXEC IDDT and BDDT commands can be used jointly. Typing BDDT to the EXEC will cause control to go to BDDT, while typing IDDT to the EXEC will cause control to go to IDDT. In this way, both debuggers

can be applied to one program. Since it is not possible to run both IDDT and BDDT at once, only the one that most recently started or continued the program will set its breakpoints in the program. Be sure to exit from IDDT to the EXEC using ";h", and from BDDT to the EXEC using the "quit" command.

E.2 BDDT Commands

E.2.1 Commands to Control Program Execution

There are 10 commands which can be used to control the execution of a program from BDDT. These commands are Start, Break, Unbreak, Continue, Goto, Retfrom, Set.BDDT.Break.Character, TRet, Watch, and Unwatch.

BDDT has a special "break character" which is used to stop your program when it is running. Typing this character (which is initially RUBOUT, but can be changed using the Set.BDDT.break.character command) stops your program as soon as it reaches a point where BDDT thinks that the stack is safe (see section E.6 below), and prints out the state of your program. It then returns control to BDDT. The break character is useful for stopping a program when it has "run-away" and is not encountering any breakpoints, or it is hung for some reason, or for stopping the program to examine some variables.

In addition, both RUBOUT and the break character can be used at any time to abort a command to BDDT. In most cases, this will instantly abort the command, and return to BDDT command level. When you are giving a subcommand, however, it will abort only the subcommand, not the top-level command.

Start -- This command starts your program at its normal start address. It is useful for starting a debugging session, and also for re-starting your program.

Break -- This command allows you to specify a point in your program at which to suspend its execution and give control to the debugger. Since it is essential that the stack be "safe" before control is given to BDDT, the debugger will simulate your program until it thinks the stack is safe (see section E.6). The break command must be followed by an SCD (Source Command Descriptor, see section E.3), which is used to determine where to place the breakpoint. An SCD indicates a point in the program, either in terms of the source text or as an Address. Break has 6 subcommands, which may be used if the SCD is terminated with a comma. The subcommands are:

NameIt -- This subcommand is followed by the name to be associated with this breakpoint. If this subcommand is not given, a default name will be created and printed. Default names for breakpoints are "BPT" followed by a small number.

Do -- This subcommand is followed by either a list of BDDT commands (enclosed in curly-braces) to be executed when the

breakpoint is hit, or the name of an action (see below) to call when the breakpoint is hit. If a list of commands is used, a default name for this action will be created and printed. Default names for actions are "ACT" followed by a small number. Action routines are useful for tracing the program flow, for automatically examining the contents of variables at certain points in the program, or for doing simple patching.

If -- This subcommand is followed by a BCPL expression (see section E.2.2). If the Value of the expression is true, the breakpoint will occur; if the Value of expression is false the breakpoint will not occur.

Count -- This subcommand is followed by a number (n). The debugger will take control when the breakpoint has been already passed n times. Thus Count 1 would cause the program to pass the breakpoint once, and then stop the next time and give control to the debugger.

ListAll -- This subcommand will make BDDT type out the address, action name, condition, and count associated with this breakpoint when you have finished giving subcommands.

No -- This subcommand is followed by one of the keywords If, Condition, Action, Do. This subcommand causes the corresponding previously executed subcommand to be ignored. (For convenience, "Condition" can be used instead of "If", and "Action" can be used instead of "Do"). For example, typing "No Condition" will cause any previous If subcommand to be ignored.

An example of a Break command is:

```
*Break Start>2,
**If DoneProcessing
**Do {NextVal/
OldVal/
OldVal:=NextVal
} (named: ACT1)
**ListAll
**
```

When a breakpoint is encountered while executing the program, the following sequence of events occurs: 1) The If expression is checked. If it is false, the program proceeds and does not turn over control to BDDT; 2) If the condition is true the Count is checked. If it is not zero, 1 is subtracted from it and the program continues; if it is zero, the Action is performed (if there is one). When the action terminates, or if there is no action, control of BDDT is turned over to the teletype.

Unbreak -- This command removes a breakpoint that was previously set. The command is followed either by a breakpoint name, or a carriage return to remove all breakpoints.

Continue -- Continues the suspended program, either from a breakpoint, or from where the BDDT break-character was typed.

Goto -- Followed by an SCD (see section E.3). The program continues from that SCD. This is similar to Continue, but allows you to specify where to resume execution in your program. For example, if you don't want to execute a piece of code, you could put a breakpoint at the beginning of it and have BDDT perform a Goto to just after it as part of the action of the breakpoint.

RetFrom -- This is followed by an SD (Stack Descriptor, see section E.4). An SD specifies a particular function or routine invocation in the current state of the program. RetFrom causes the program to return from the indicated function or routine. It can optionally be followed by the keyword "With" and an expression, in which case the value of the expression is returned as the value of the function call.

Set.BDDT.Break.Character -- This command changes the character that is used to suspend the running program and give control to BDDT, and also to cancel BDDT commands. It is initially set to rubout. The break character can be set to any control-character, or rubout, except for control-A, C, R, and Q.

TRet -- This command is followed by an SD (see section E.4). It is the same as the Break command, except that it causes a break when the program returns from the specified function or routine invocation. TRet's automatically remove themselves after they occur. The same subcommands that exist for the Break command can be used with TRet.

Watch -- This command is followed by a list of eitner complete SCDs, names of functions or routines, or names of REL files followed by ":". It causes these SCDs, routines, or all routines and functions defined in the files, respectively, to be "watched". Normally, this means that when the routine or function is called, a message is printed out on the teletype, and when the routine or function is exited its result is printed. There are 4 subcommands to Watch, which can be given if the last SCD or file is terminated with a comma. They are:

Header -- This subcommand means that when the watch occurs, a message should be printed indicating that the watch has occurred.

Trace -- This subcommand can optionally be followed by a number. It means that when the watch occurs a trace of that many frames should be printed.

Result -- This subcommand is followed by a type-out mode ($A, $C, etc., see secton E.2.2). When the routine in which the watch occurs exits, the result is printed in the mode specified.

Special -- This (unimplemented) subcommand is used to specify the names of variables that are to be printed out when the watch occurs.

No -- The subcommand can be followed by any of the above subcommands, and the effect is to negate their effect. Thus "No Header" will make the watch not print out the message that it has occured.

Old -- This subcommand is used to undo a subcommand that may have been given by mistake. For example, "Old Header" will restore the watch to either No Header or Header, depending on what it was before this watch command was given.

Default subcommands are Header, Result $O, and Trace Ø.

For example, the command:

```
*watch Write:,Main:Start,DonePrinting,
**Result $S
**
```

would cause a heading to be printed whenever any function or routine in the file Write, the function Start (in the file Main), or the routine DonePrinting is entered. Whenever one of these functions or routines is exitted, the result is printed in string mode.

UnWatch -- This command is followed by a list like the one in the watch command. It removes any watches that may have been set on those locations. If no list is given, all Watches are removed.

E.2.2 Commands to Examine Program Status

These commands allow you to examine the BCPL source code, the compiled machine code, the values of variables, and the nest of function and routine invocations.

Expressions are any valid BCPL expression, including, for example variables, manifests, and the result of a "valof" statement. The current implementation of BDDT allows only constants, variables, variable|constant, variable|variable, and constant|constant for expressions ("|" may be replaced by "!").

Since all variables in BCPL are scoped, BDDT simulates the scoping rules. In order to be more convenient, however, BDDT recognizes a sort of "dynamic scope" made up of the sequence of routines that have been called. In order to find the value of a variable, BDDT first examines the symbol table of the routine or function that was being executed just before BDDT was entered. If the variable is not found, a list of globals and externals is checked. If it is still not found, BDDT "backs up" one stack frame and searches again. If, after backing up all the way to the top of the stack, the symbol still isn't found, a list of all the unique statics defined in

all the symbol tables that are loaded is searched. If all this fails, an error message ("?") is printed.

Since it is sometimes necessary to examine a variable that is defined several stack frames back, but is re-defined as a new local variable in a later function, BDDT has a special "Attention" command that can be used to make symbol searches start somewhere other than at the top of the stack. See section E.2.5.

<expression>/ -- This causes the Value of the expression to be printed. Since slash is also the BCPL division operator, any expression containing division must be enclosed in parentheses.

<expression>= -- This prints the Address of the expression in octal. It has the same effect as
"lv <expression>/".
If the expression doesn't have an Address, a question mark is printed. Since equals-sign (=) is a BCPL operator, expressions using it as an operator must be enclosed in parentheses.

<expression>@ -- This has the same effect as
"rv <expression>/".

. -- The symbol "." can be used to mean the last thing examined by either / or @. Thus, typing ./ will print the Value of the thing just examined, .= will type the Address of the thing just examined, and .@ will print the "rv" of the Value of the thing just examined. This latter is useful for following a chain of pointers.

line-feed, Control-H (backspace) and "^" -- These commands can be used any time after examining a Value with either "/" or "@". They mean "examine the Value of the next storage cell," where the next storage cell means the one following the cell represented by "." if the command is linefeed, or the one preceding the cell represented by "." if the command is "^" or backspace. The number of cells to move forward or back depends on the type-out mode. For example, on the PDP-11 in instruction type-out ($I) mode, linefeed may print the next word, or the one after that, depending on whether the instruction is one, two, or three words long.

Several commands that examine the Values of expressions, or change Values of expressions can be placed on one line. Line-feed and "^" can be used until you type a carriage-return. For example:

```
*Arg1/     #10     $d     8     .:=17(*L)
#1001/    #15     Arg2/     1073(*C)
*Table/     #400000     @     #150     $s
 HELP!!*nI'm*sdead.*n
```

$A, $C, $D, $F, $I, $O, $S -- These commands change the print-out mode to ASCIZ string (A), Character constant (C), Decimal integer (D), Decimal floating point (F), Instruction (I), Octal integer (O), and BCPL string format (S). If the command has a single dollar-sign (as above), the effect lasts until the command line is ended with

carriage-return. If the command has a double-dollar-sign (e.g. $$A), the effect lasts until another double-dollar-sign command is given. If any of these commands is given after an expression nas been examined, but before a carriage-return, the Value printed is re-printed in the new mode. The default print-out mode is $D. (In order to make tne typing easier, and to be more compatible with DDT, alt-mode and escape can be used instead of "$", and will print out as "$").

Trace -- This command prints the current nest of function and routine invocations, their arguments, and other useful information. Trace can optionally be followed by a number of frames to trace back, and the trace will stop after printing that many frames.

Print -- This command is followed by an SCD and prints the source text at that point in the program. It can be followed by the key words Up or Down (or both) followed by a number of lines before and after the line to be printed. This command is very useful for locating a program position in the source text. In order to use this command, the .BCP file in which this SCD is defined must be in your CONNECTed directory.

For example:

```
*Print Main:Start[let T := 6]<1 up 2 down 2
    and Start() be
     {
==>       let T:= 6
        for i:= 1 to T do
        { A := valof
```

">" and "<" -- These commands are identical to typing "Print %>1" and "Print %<1" respectively.

Address -- This command is followed by an SCD. It prints the SCD as an (octal) Address. Like the Print command, it is useful for relating source text to compiled code, and is helpful if it is necessary to switch from BDDT to IDDT.

Code -- This command is followed by an SCD. It prints out tne compiled code for tne given SCD.

E.2.3 Commands to Change Program Status

Anytime a location can be examined, an assignment can be made. This is done using the charcters ":=" for the assignment operator. If there is nothing to the left of the assignment operator, "." is assumed. (The assignment operator is currently either "_", or ":=").

For example:

```
*X/      #10      :=15(*C)
```

but not:

```
*#100/      64      :=5
```

(since it is illegal in BCPL to say #100:=5)

E.2.4 A Command to Call a Function or Routine

The Call command is used to call a routine or function. The command is followed by the name of the function or routine, an open paren, an argument list (each argument can be an expression), and a close paren, and terminated with a carriage return. The routine or function is called, and the returned result is printed. If a breakpoint is hit during execution of the call, it will be handled normally by the debugger, and the Continue command will continue the called routine or function. When the routine or function finishes, the continue command will resume the program where it stopped before the Call command was given.

E.2.5 Commands to control BDDT

Short.Commands -- This command stops BDDT from completing the printing of a command that is prematurely terminated with space or carriage return.

Long.Commands -- This command returns to automatic printing command completion mode. BDDT is initially in Long.Commands mode.

Quit -- Stops BDDT and exits to the EXEC. BDDT can be continued by typing either Continue or BDDT to the EXEC. Quit should always be used to leave BDDT, not Control-C, since Quit removes any breakpoints that may be set and changes the teletype echoing to what it was when your program last stopped.

Reset -- This re-initializes some internal BDDT information. Mainly, this command has the effect of "forgetting" about ever having run the program. It has no effect on either actions or breakpoints, breakpoints will still be set in the program, and associated with actions. The only thing which is changed is information about which actions it may have been executing previously, and which breakpoints had been encountered. After you execute a Reset command, you cannot Continue your program until you have given a Start command.

Get -- This command asks for the name of a program to load. It effectively re-starts the debugging session by reloading the program and its symbol files.

Do -- This command is followed by either an action name or a list of BDDT commands enclosed in curly braces. It can be followed by the keyword "if" and an expression. The commands are executed if the expression is true.

Example: Do { Trace } if Done
(where Done is a variable defined in the program)

List -- This must be followed by one of the keywords "Breakpoints" or "Actions". It lists all the defined actions or breakpoints, and their contents or subcommands.

Edit.Breakpoint -- This command is followed by a breakpoint name. It allows you to enter any of the subcommands to the break command, and the additional subcommand "Old", which is like "No", except that it restores the parameter to the Value it had before you gave the Edit command.

Edit.Action -- This command, which is not implemented yet, will be used to allow you to create or edit action routines. The specifications are not yet clear, but it will probably invoke TECO to allow you to edit the routines.

Load.Symbols -- This command is followed by the name of one of the .REL files in your program, and the file name where the symbols for that program can be found. It causes BDDT to load those symbols. This is useful for obtaining symbols from programs which were not compiled on your CONNECTed directory, or which do not have the extension ".S". It types a warning if a symbol file was already loaded for this file. If the rel file was not compiled by BCPL this command should not be used to load symbols for it.

Attention -- This command is followed by an SD. It makes all subsequent symbol searches begin at the specified SD.

Unwind -- This command is followed by an SD. It is the same as a RetFrom command, except that instead of continuing the program, BDDT retains control. It is useful for examining variables several frames back on the stack before returning from the routine. "attention. . .retfram" is equivalent to "unwind. . .continue".

E.3 Source Command Descriptors (SCDs)

SCDs are used to identify program points, either as memory addresses or as "pointers" to the BCPL source text. They are designed to be independent of the machine for which the code was written. An SCD has several parts, each of which is described below, along with the required punctuation. The parts must occur in the order given below, although some of the parts may be missing.

An example of a complete SCD is:

```
ACTS:ReadAction[let T :=]<3+2
```

The REL file name -- This part is terminated with a colon (:) if it is present. It is the name of the RELfile in which this SCD is defined. If only one static exists with the function or routine name, the REL file name can be omitted (this latter feature is not implemented yet).

The Function or Routine name -- This is the only required part of the SCD. It is the name of the BCPL function or routine in which the SCD is defined.

Search Strings -- A string to be searched for in the BCPL code can be specified. This is done by enclosing the exact string in

square brackets ("[" and "]"). More than one string can be searched for. After the search, the SCD points to the first command which begins after the start of the search string. When using the search feature it is wise to use the Print command to make sure that the correct SCD has been found. The .BCP file in which the SCD is defined must be in your CONNECTed directory for search strings to be used.

Number of Commands -- There is a loosely defined idea of a command in BCPL. For example, the if <expression> construct is a command, as is the do <statement> construct. It takes some playing around with BDDT to get a feel for what a command really is. You can specify the number of commands before or after the part of the SCD you have already typed by preceeding this number with > or <. For example, the second command in the routine Start would have the SCD "Start>2".

Number of Machine words -- You can also specify the exact number of words before or after what you have typed by preceding this number by either + or -. For example, the second word past the beginning of the first command in the routine Start would have the SCD "Start>1+2".

% can be used in the routine or function field to represent the most recent SCD used in a Print command, or the SCD at which the program was stopped. (This latter is not implemented yet.) Example: the SCD for the command following the one just printed could be represented by %>1.

In addition to the "standard" SCD described above, a number can be used for an SCD, in which case it stands for the SCD which occupies that Address in core. For example, if Start begins at location #140 (octal), then #142 when used as an SCD is the same as "Start+2".

E.4 Stack Descriptors (SDs)

SDs are used to refer to a particular invocation of a function or routine in the run-time stack. There are two forms of SDs, either a number (n), meaning the nth preceeding function or routine (the current one is designated as 0, the one before it is 1, and so on), or as the name of the function or routine followed by "-n" where "n" is used to indicate which call on the (recursive) function or routine is meant.

If "-n" is omitted, the most recent call is meant (i.e. n is assumed to be zero). If n is 1, the call before the most recent is meant. Example: Foo-2 means the invocation of Foo two previous to the most recent invocation of Foo.

E.5 Actions

Actions are lists of BDDT commands which are stored for use, and can be executed when a breakpoint is encountered. Actions can be created in three ways, by using the Do subcommand to Break, the Do command, or the Edit Action command.

To create an action, type in the list of commands, exactly as you would to BDDT, but enclosed in curly braces ("{" and "}"). BDDT will print out a name which you can use to refer to the action.
Example:

```
*Break Start>2,
**Do {T/
A:=15
Trace
continue
} (named: ACTØØ)
**
```

This creates an action named ACTØØ.
Afterwards, you can type

```
Do ACTØØ if BitSetBool
```

(Where BitSetBool is one of your program's variables).

E.6 Where BDDT will Stop Your Program

Since BDDT uses the stack for almost every command, it is important that the stack be "safe" before you enter BDDT. In order to ensure this, BDDT examines where the program has stopped, and simulates the program until the stack is "safe". This means that you cannot have BDDT stop your program inside the sequence of instructions for calling a routine or function.

E.7 BDDT conventions

There are four major conventions used in BDDT, a convention for inputting numbers, a convention for editting commands, a case convention, and a set of conventions for file names. Each of these is described below.

All numbers input to BDDT are assumed to be in decimal unless preceded by "#". For example, "15" means 15 decimal, whereas "#15" means 15 octal (13 decimal). For output, BDDT will always indicate octal numbers by preceding them with "#", except when printing in instruction mode, where all numbers are octal, to be consistent with DDT and IDDT. BDDT initially prints values in decimal.

BDDT has 3 reserved characters that are used for editting input. They are control-A, which will delete one character at a time until the beginning of this field (if you try to erase characters before the field, you will get a bell). Control-R will retype the entire line as typed so far. Control-Q will cancel the entire current field, and then retype the line.

Commands to BDDT may be in either upper or lower case, or a mixture. Variable names, however, must be typed exactly as they appeared in the program, with capitals as required. A future version of BDDT will provide for easier debugging of programs from all upper-case terminals. Currently, if a program was compiled with the "/U" switch on the compiler, all its symbols must be typed to BDDT in lower case.

When BDDT searches for symbol files, it assumes that the file will have the ".S" extension. The Load.Symbols command may be used to load other files. Source files are assumed to have a ".BCP" extension.

APPENDIX F: PDP-11 BCPL

F.1 INTRODUCTION

In 1973, Ray Tomlinson and Jerry Burchfiel of BBN adapted the TENEX BCPL compiler to generate code for the PDP-11. This effort turned out to be a two pass learning experience: the first version was a modified code generator driven by standard OCODE. Upon evaluation, this approach was shown to be unsuitable for production of efficient code because of the postfix nature of OCODE: expressions must be computed before any information is available about the disposition of the results. This makes it impossible to compute values in the location where they will be stored, to optimize use of the registers and stack, etc. The net result was impressively poor code.

The second attempt (and current implementation) eliminated the use of OCODE, instead generating code by subroutine calls while walking the AE (applicative expression) tree, where all disposition information is readily available. This approach results in excellent code for expression evaluation: recursive functions do register and stack allocation by examination of the AE tree, so results are computed into the location where they will be needed.

Evaluation of code generated indicates that cross-compiled PDP-11 BCPL code takes about 50% more core storage than equivalent functions hand-coded in assembly language by a shrewd and creative programmer. This difference results almost solely from the stack discipline observed by BCPL: a new stack frame must be created and arguments pushed into it for each function call. The creative programmer, on the other hand, tends to twiddle bits and then jump into the middle of some routine. This saves code but results in debugging nightmares. Structured programming costs core memory, but pays for itself by making debugging and program modification straightforward.

An additional observation worth making is that the process of creating a high-level language compiler for a machine is an excellent way to evaluate the machine architecture: a machine which does not gracefully support the constructs of nigh level languages is a poorly designed machine.

For example, the stack of the PDP-11 operates backwards, building towards lower addresses. This was undoubtedly considered a clever storage allocation trick by some hardware designer, but presented problems for BCPL: calculating and pushing successive elements of a dynamic vector into the stack causes them to appear in backwards order in the address space: the machine's address arithmetic cannot be used to access the vector elements.

On the other hand, the reversed byte order (byte 0 is the low-order byte of a word, byte 1 is the high-order byte) brought howls of anguish from the ARPA PDP-10 community, and the ILLIAC Project even made hardware modificatons to their PDP-11s to reverse the byte

ordering. However, the reversed bytes caused no addressing problems, inconsistencies, or even structure reference difficulties in the PDP-11 BCPL compiler. This was merely a problem of preference, not a true implementation difficulty.

A number of other addressing difficulties which we discovered are described in section F.4 below.

F.2 PDP-11 OBJECTS

The objects handled by PDP-11 BCPL are bytes, words, and structure fields. The use of the structure facility for databases maximizes the machine-independence of programs (permitting use of code on either the PDP-10 or PDP-11), as long as no structure field is required to be longer than 16 bits (a field may not cross a word boundary).

Structure fields are allocated from right-to-left (bit 0 to bit 15) so that, for example, successive byte-sized fields will fall into successive byte addresses. Structure references are compiled to put as much burden on the PDP-11 address arithmetic as possible. For example, byte instructions are compiled to reference byte fields.

A PDP-11 character is represented as an 8-bit byte. Strings can be no more than 255 characters long (in successive byte addresses), since the string length must be stored in the first byte.

F.3 PDP-11 OPERATIONS

Some of the operators of TENEX BCPL have not been implemented in PDP-11 BCPL. They are:

1. Floating point operators %*, %1, %+, %-. Our PDP-11 has no floating point hardware. However, the addition of these operators would be quite simple for machines with the floating hardware.
2. Quarter word operators q1, q2, q3, q4, q1z, q2z, q3z, q4z. No application for these operators was apparent.

A restricted set of operators are defined for byte manipulation. This approach makes available the power of the PDP-11, which deals with both bytes and words, but violates the spirit of BCPL, which is that all objects remain typeless. As a compromise, the BCPL syntax has not been changed, but it describes both bytes and words: whether an object is a byte or a word must be inferred from syntactic context.

To be specific, a byte object is either a byte primitive or a byte expression. In the BNF definitions below, W is a word object.

```
<byte primitive>:=  rh W lh W                (compiles MOVB)
                    -<byte primitive>        (compiles NEGB)
                    not<byte primitive>      (compiles COMB)
                    <byte primitive>+1       (compiles INCB)
```

```
                    <byte primitive>-1     (compiles DECB)
                    <byte primitive>*2     (compiles ASLB)
                    <byte primitive>/2     (compiles ASRB)

<byte expression>:=  <byte primitive>
     <byte primitive>&<byte expression>     (compiles BICB)
     <byte primitive>\<byte expression>     (compiles BISB)
```

All other operations are fullword operations which operate on word objects. Byte objects are automatically converted to word objects as needed in expressions by sign extension.

F.4 PDP-11 ADDRESSING

We already mentioned one addressing problem, the backwards stack. Elements of a "list" declaration are computed and pushed onto the stack in reverse order so that they may be referenced as a vector using the PDP-11's address arithmetic.

A more serious problem is the inherently schizophrenic behavior of the PDP-11: most operatons (ADD, SUB, MUL, DIV) require fullword operands; however, these operands must be referenced using byte addresses. Inconsistencies in specified operations can only be detected by the hardware at runtime: a bus address trap is triggered when an odd byte address is generated during a word operation.

This problem would not be so serious if the high level language did not support address arithmetic specifications. However, BCPL permits the user to manipulate pointers (rv operator) and specify address arithmetic (vector application, e.g. Foo|i). Either of these can cause a word transfer with an odd byte address, resulting in a fatal runtime error.

This error can only be avoided by extreme caution on the part of the programmer (remember to index by 2 when scanning through a word array) or by avoiding use of rv and vector applicatons through exclusive use of the structure facility. Vector declarations are also a source of confusion: for byte addressing consistency, vec(n) allocates bytes from Ø to n. This is a drastic example of machine dependency, as vec(n) means words Ø to n in TENEX BCPL.

The conclusion to be drawn here is that byte addressing of word quantities was a very dubious choice on the part of the PDP-11 machine designers.

One other point should be mentioned with respect to addressing in PDP-11 BCPL. A new type of entity called a "load-time constant" has been created for purposes of efficiency. Both labels and static vectors are load-time constants; they cannot be manifest constants because the compiler cannot determine their address value until the program has been loaded. In TENEX BCPL, they are variables, which always causes an extra level of indirection when they are referenced.

For example, in TENEX BCPL, goto L causes an indirect transfer through a variable cell L which holds the address of the desired destination. Similarly, reference to a static vector requires explicit addition of values to find the desired address, instead of using the machine's address arithmetic. Load time constants eliminate this extra level of indirection, and permit references to these elements through indexed addressing.

For example, a static array can be made a load time constant by a declaration:

```
manifest {Foo: vec 100}
```

and elements of Foo will be referenced by indexed addressing.

At some point in the future, it is likely that TENEX BCPL will also define labels and static vectors as load time constants, making it impossible to dynamically redefine a label or static array (a highly suspect practice).

A related form is the rv of a number, which is permitted as a manifest constant in PDP-11 BCPL. For example:

```
manifest{PS:=rv#177776}//Program status word
PS:=PS \#300  //Set processor priority to 6
```

This construction permits convenient access to processor and device registers.

F.5 THE STACK DISCIPLINE

Recursive calls on all functions and routines are supported by a stack discipline which creates a new frame on the top of the stack for each function invocation. In the PDP-11, two pointers are maintained to define the current frame: the framebase pointer points to the beginning of the frame (where the return PC is saved), and the stack pointer points to the current top of the stack, which moves as temporary results are pushed onto and popped from the stack. When a hardware interrupt occurs, the processor state information is pushed onto this same stack.

As mentioned before, the 0th word of the frame base holds the return PC. The next word holds the number of arguments supplied by the caller. This permits functions to be called with a variable number of arguments, a facility particularly useful for library functions which supply reasonable defaults for all arguments not suplied by the caller. The calling arguments are pushed into successive stack words, followed by local variables and temporaries up to the top of the stack.

The calling procedure is thus:

```
    Caller: Push hole for return PC
            Push number of calling args
```

```
        Push calling args
        Increment framebase ptr to return PC hole
        Call routine
        Decrement Framebase ptr by amount it was incremented
```

and the routine itself is:

```
Entry: Put return PC into hole at base of frame
       Adjust Stacktop ptr to number of defined args
 Exit: Copy framebase ptr into stacktop ptr
       Subroutine return - restore return PC
```

This arrangment permits calls with a variable number of arguments while maintaining a "covered stack", in which no valid information is ever beyond the stack pointer, where it could be smashed by interrupts.

INDEX

Allocation 46
and 49
Assignment 37
bit 53
bitb 53
bitn 53
Blocks 44
Boolean 23
branchon 41
break 42
by 40
byte 53
byten 53
case 35, 41
char 53
default 35, 41
do 38, 40
endcase 42
eq 30
eqv 31
Extent 46
external 46
fill 53
for 40, 42
form 43
Functions 48
ge 30
global 47
goto 38
gt 30
Half-word 33
if 38
ifnot 39
ifso 39
label 37
le 30
let 48
lh 27
lhz 27
loop 42
ls 30
lscale 30
lsnift 30
lv 10, 26
manifest 48
ne 30
neqv 31
nil 24
not 31
or 39
overlay 54

ql 28
qlz 28
q2 28
q2z 28
q3 28
q3z 28
q4 28
q4z 28
Relational 30
rem 29
repeat 42
repeatuntil 39, 42
repeatwhile 42
repeatwhile 39
repeat 39
repname 35
reserve 63
resultis 43
return 43
rh 27
rhz 27
Routine 37
Routines 49
rscale 30
rshift 30
rv 10, 27
Scope 45
Sections 44
selecton 35
Shift 30
size 55
static 47
step 40
Structures 50
switchon 41
synonym 64
test 39
then 38, 39
to 40
unless 38
unreserve 63
until 39, 42
valof 24
Variables 48
vec 47, 48
Vectors 48
while 38, 42
word 53

"left lump" 51
"right lump" 51
"subscript" 53
"up arrow" 53

%* 29
%+ 29
%- 29
%/ 29

& 31

* 29

+ 29

- 29
-> 33

/ 29

:= 36

<< 51

>> 51

Address 9
Address 26
Arithmetic 29
assignment 10

brackets 44

CAVEAT 41, 44, 46, 62, 64
CAVEAT 27
character 22
CONC 70
conditional 33
constant 9

Dynamic 46

example 70
expression 21

floating 29
FMT 69
function 25

get 63

half-word 27

integer 29

JFN 72

list 34

Manifest 9

name 9, 23

OCODE 69

parentheses 24
PSAVE 70
PSYMB 70

quarter-word 28

Static 46
string 22

table 34

Value 8
variable 9
vector 11, 25, 34

\ 31

^ 53