UC Irvine
ICS Technical Reports

Title
UCI L_6 language manual

Permalink
https://escholarship.org/uc/item/7qv96782

Authors
Sowizral, Henry
Tadman, Frank

Publication Date
1974

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

UCI L^6 LANGUAGE MANUAL

by
Henry Sowizral
Frank Tadman

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

Department of Information and Computer Science
University of California, Irvine
TECHNICAL REPORT #50 -- June, 1974

TABLE OF CONTENTS

INTRODUCTION

L^6 stands for Bell Telephone Laboratories Low Level Linked List Language. UCI L^6 is an extended version of the L^6 implemented at Bell Telephone Laboratories. The extensions added make UCI L^6 a more powerful and convenient programming medium.

The major extensions to the original language are:

1) The user is allowed a close interactive control over the creation, execution, and modification of his program.

2) ELSE clauses have been added. These are executed only if the test fails.

3) Lines may now be typed in a freer format with no restriction on column usage.

4) L^6 programs may now be divided into modules called procedures which are independent of one another.

5) More primitives have been added to L^6. These make string manipulation, multidirectional branching, and extended stack operations much easier.

6) Both string and label variables have been added to allow dynamic specification of parameters to primitives.

7) The input and output facilities have been extended to allow multiple file access. Further, I/O can be done in any legal buffered data mode.

8) An extensive debugging facility has been added to allow the program to be interrupted. The state of the program can then be examined and execution continued.

9) Commands for creating and editing program have been added.

INTRODUCTION

This manual describes the basic L^6 language and the extensions to it. The manual is divided into twelve chapters. The first four provide an overview of the L^6 language and the data structures which can be constructed with it. The last chapters provide a more detailed description of the L^6 language, its operators, and its constructs. We suggest that those unfamiliar with L^6 read the first six chapters and chapter twelve. The other chapters describe features which are useful to more advanced programmers.

Several conventions are followed in the examples throughout this manual.

1) The ASCII control characters are written as an up arrow followed by a character. For example, control-C is represented by "↑C".

2) In examples of interaction with the computer, portions which are underlined were typed by the computer.

3) Text enclosed in square brackets is optional.

The L^6 interpreter was designed and implemented by Henry Sowizral and Frank Tadman as an independent study project. We thank Robert J. Bobrow for initiating and advising this project, and for the many hours he spent in discussing the design and implementation. We thank Richard Rubinstein for helping to sustain the project in professor Bobrow's absence. Allan Foodym provided many helpful comments and suggestions. Also we would like to acknowledge the senior project group (ICS 190B) which did an earlier implementation of L^6 on the Sigma-7 computer. Finally, this manual is based in large part upon the Sigma-7 L^6 language manual written by Robert Bobrow.

CHAPTER 1
GETTING STARTED WITH L^6

This chapter introduces the basic commands necessary to load and run an L^6 program online. Appendix A gives a deck setup for running L^6 in batch. If the user is at the monitor level and wishes to run an L^6 program he first must tell the monitor to run the L^6 interpreter by typing:

```
.R L6
```

L^6 then responds by typing its version number, the date, the time, then indicates its readiness to accept a command by typing a colon. Now the user tells L^6 which program he wants run. He does this via the LOAD command. This command takes as its arguments the name of the file that the program is saved on and optionally the name of the file on which the user wants the program and errors to be listed. The commands format is:

```
:LOAD [outfile=]infile
```

where "outfile" is optional and specifies the name of the file the user wants the listing on. Examples of "outfile" are:

```
LPT:
PARSER
PROG
```

The other argument "infile" is the name of the file that contains the program the user wishes to run.

Once the program has been loaded the next step is to start it running. To do this the user types:

```
:RUN procname
```

where "procname" is the name of the procedure that is to be executed. Whether or not the program runs successfully a dump of the status of the program is usually required. The command to do this is DUMP. It allows the user either to channel the dump to a file or to his terminal. Its format is:

```
:DUMP/FILE:outfile
```

If the "/FILE:outfile" is omitted then the dump is typed on the terminal, otherwise it goes to the file specified by "outfile".

Two other useful commands are LIST and EXIT. LIST permits the user to see the L^6 programs which are currently defined, while EXIT exits the user to the monitor.

A session could go as follows:

```
.R L6

UCI L6 2(5)              21:49:46          23-MAY-74

:LOAD TTT=TTT

NO ERRORS DETECTED
8P CORE USED

:RUN TRAVERSE

HALT AT LEVEL Ø
8P CORE USED

:DUMP/FILE:DUMP
:EXIT

EXIT

.
```

This user first ran the L^6 interpreter. After it responded he asked it to load all the L^6 procedures defined in a file called TTT.L6 and list the procedures with whatever errors there may be on a file named TTT.LST. He then ran the procedure named TRAVERSE. After it ran, a dump was requested on the file DUMP.LST. Finally, he returned to the monitor.

CHAPTER 2
DATA STRUCTURES

2.1 INTERNAL REPRESENTATION OF ELEMENTARY DATA

Internally, there is only one type of elementary data item in L^6, the field. A field is a sequence of n contiguous bits ($1 \leq n \leq 36$) lying within one machine word. Depending on the size of the field and the operation to be performed, a field may be used in various manners. Thus a field may contain:

a) a positive integer in binary form
($1 \leq n \leq 35$)

b) a signed integer in two's complement binary form
($n = 36$)

c) a sequence of 1, 2, 3, 4, or 5 characters in ASCII code
($n = 7, 14, 21, 28$ or 35)

d) a bit string or set of flags or logical variables
($1 \leq n \leq 36$)

e) a pointer to a word in user-allocated memory
($n = 18$, and the value of the 18 bit integer forms the address of the indicated word.)

2.2 CONSTANTS

There are several ways in which data can be represented in an L^6 program. Since all data is stored internally within fields, the external representation (or constant notation) is important primarily for programmer convenience and readability of programs. It is good programming practice to use notations that indicate the operations to be performed on the data.

2.2.1 NUMERIC CONSTANTS

DECIMAL INTEGERS

When data is to be used for arithmetic, it is most commonly represented in the form of an integer in decimal notation, with or without an optional sign. Some examples of the notation for integers are:

```
185      1234976      +169      -243      534258913
```

ARBITRARY BIT CONFIGURATIONS -- OCTAL REPRESENTATION

A convenient way for the user to represent arbitrary bit strings is as octal numbers. An octal number is written as a number sign (#) optionally followed by a sign (+ or -), followed by a sequence of from 1 to 12 characters from the set

Ø, 1, 2, 3, 4, 5, 6, 7

ANOTHER REPRESENTATION -- HEXADECIMAL REPRESENTATION

Another way to represent bit strings is using hexadecimal numbers. These are written as a period (.) followed by a sequence of 1 to 9 hexadecimal digits. These are any of the following characters

Ø, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

OTHER BASES -- NUMBERS OF BASES 2 - 1Ø

Numbers may be written in any base between two and ten by writing the base, followed by a number sign (#), and by an optionally signed integer. Some examples are:

```
2#1Ø111Ø11Ø1Ø11Ø11      8#2412      5#-432      1Ø#+1Ø
```

In an octal number (as in an hexadecimal number, but not a decimal number) each character represents a fixed number of bits (four for hexadecimal, three for octal).

OCTAL

Ø = ØØØ	1 = ØØ1	2 = Ø1Ø
3 = Ø11	4 = 1ØØ	5 = 1Ø1
6 = 11Ø	7 = 111	

Some examples of octal constants are:

```
#21   (= Ø1ØØØ1)
#5Ø23 (= 1Ø1ØØØØ1ØØ11)
#7777 (= 111111111111)
```

HEXADECIMAL

Ø = ØØØØ	1 = ØØØ1	2 = ØØ1Ø	3 = ØØ11
4 = Ø1ØØ	5 = Ø1Ø1	6 = Ø11Ø	7 = Ø111
8 = 1ØØØ	9 = 1ØØ1	A = 1Ø1Ø	B = 1Ø11
C = 11ØØ	D = 11Ø1	E = 111Ø	F = 1111

Some examples of hexadecimal constants are:

```
.11    (= ØØØ1ØØØ1)
.A13   (= 1Ø1ØØØØ1ØØ11)
.FFF   (= 111111111111)
.1ØAB2 (= ØØØ1ØØØØ1Ø1Ø1Ø11ØØ1Ø)
```

2.2.2 CHARACTER STRINGS

When data is to be used for output of alphanumeric characters, or other purposes where it is to be considered as a character string (e.g., when the character code for "Ø" is to be subtracted from the character code for a numeral to get the associated integer) the data is best represented as a character string. Character strings may be any sequence of one to five characters enclosed in either single quotes (´) or double quotes (").

Note that if a single quote is to be contained in a string the string should be delimited by double quotes. The delimiting character may also be inserted in a string by using two consecutive occurences of it. Some examples of character strings are:

```
"A"    ´A´    "AB*C"    ´Q3-A´    "AB´C"    "12A"
```

"""I´M""" and ´"I´´M"´ both represent the string "I´M".

LONG TEXT STRINGS FOR TITLES

To make it easier to print long titles or headings, two of the output operations (the text output operations, TOUT and FOUT) accept character strings of any length. For example:

```
"THIS IS AN EXAMPLE OF A LONG STRING FOR A HEADING"
```

SPECIAL CHARACTERS - THE EXCLAMATION POINT

An exclamation point (!) is interpreted specially in a text string. If it is followed immediately by one of the letters in the table below the exclamation point and the letter are replaced by one of the ASCII control characters shown below. This allows the use of special characters which can not be conveniently used in text strings. Two consecutive exclamation points are treated as a single exclamation point.

TEXT ARGUMENT	CONTROL CHARACTER	OCTAL VALUE	CHARACTER NAME
!C	↑M	15	carriage return
!L	↑J	12	linefeed
!F	↑L	14	formfeed
!T	↑I	11	tab
!V	↑K	13	vertical tab
!B	↑G	7	bell
!A	↑[	33	altmode (escape)
!U	↑]	35	home up
!D	↑\	35	home down
!S	↑↑	37	erase eof
!E	↑<	36	erase eol
!W	↑K	13	cursor down
!X	↑X	30	cursor forward
!Y	↑Y	31	cursor back
!Z	↑Z	32	cursor up
!!	!	41	exclamation point

Some examples are:

"THIS IS A LINE OF OUTPUT!C!L"

"!U!S"

"!!SURROUNDED BY SINGLE EXCLAMATION POINTS!!"

2.2.3 INTERNAL REPRESENTATION OF CONSTANTS

Although there are many external representations of data, all data is stored within L^6 as binary numbers. Therefore, once a constant is read in, it may be used for any purpose. For example, the constants below may be used interchangably.

```
"A"     #1Ø1     65     6#145     'A'     2#1ØØØØØ1
```

2.2.3 INTERNAL REPRESENTATION OF CONSTANTS

Although there are many external representations of data, all data is stored within L^6 as binary numbers. Therefore, once a constant is read in, it may be used for any purpose. For example, the constants below may be used interchangably.

```
"A"      #1Ø1      65      6#145      'A'      2#1ØØØØØ1
```

2.3 BLOCKS AND PLEX STRUCTURES

L^6 allows the user to build arbitrarily complicated data structures. The basic component of such structures is the block, which is a set of sequential words in a special area of memory, the user-allocated storage area. These blocks may be any size that will fit into the available storage. As far as the L^6 system is concerned, a block has no internal structure, and can be used to hold any data that the user desires to store in it.

The L^6 system provides the user with an automatic storage management facility which allows the user to request access to a block of any desired size, store data in the block that has been allocated, manipulate the data, and eventually return the allocated block to the storage manager when it is no longer needed. This allows the same area of storage to be used to hold different data structures at different times in the execution of the program. In general, the user conceptually decomposes a block into fields, each containing a meaningful piece of data. (It is possible to have overlapping fields.)

L^6 provides the user with convenient ways to extract data from fields within a block, to combine data from various fields and store the result in a given field. (See the section below on referencing data in fields.) It is up to the user to set up the data in a block as he needs it. Thus, for example, a block can be used to hold the elements of an array, stored in standard row or column major order, it can be partitioned (conceptually) into various fields of arbitrary length, it can hold a sequence of ASCII characters, etc.

One of the most important types of fields a block can contain is a field with a pointer to the origin (or middle) of another block. As indicated above, such a pointer field must be at least 18 bits long, and holds the relative address of a word in the user-allocated storage area. A field containing a pointer is often called a link field.

Such link fields allow the user to create complicated structures, called plex structures, containing many blocks linked together by pointers. Such structures can be used to represent linear lists of characters or numbers, graphs, circular lists, stacks, queues, etc.

2.4 STORING AND REFERING TO DATA ITEMS

L^6 has five places where data may be stored. The most important of these is a group of 26 registers called bugs. Using the bugs, the program may reference blocks in the user-allocated storage area. L^6 also has three stacks, the field contents stack, the field definition stack, and the subroutine return pushdown stack where the program may store data.

2.4.1 BASE DATA ITEMS -- BUGS

The basic data storage areas in L^6 are 26 registers, called bugs. These bugs are refered to by the single letter names:

A, B, C, D, E, F, G, H, I, J, K, L, M,
N, O, P, Q, R, S, T, U, V, W, X, Y, Z

Each of these registers can contain any 36 bit quantity. They can be used to hold numbers for arithmetic, characters for input and output, and most importantly, they can be used to hold pointers to blocks in the user-allocated storage area. Such pointers form the basis for refering to all data in the user-allocated storage area.

To refer to the 36 bit quantity contained in a bug, the user need only write the name of the bug in the correct place in the program. Thus, to refer to the contents of the bug Q, one writes Q.

2.4.2 REFERING TO DATA IN THE USER-ALLOCATED STORAGE AREA

To refer to a field in an allocated block, the user must specify a pointer and a field template. A pointer is an 18 bit quantity which gives the relative address of a word in the user allocated storage area. A field template indicates the position of a field relative to a pointer.

Most plex structures have pointer fields which contain null pointers. These are pointers which do not point to anything and serve to mark the "end" of the plex structure. The pointers Ø, 1, and 2 are reserved as null pointers. A block will never be allocated in the user-allocated storage area with these relative pointers. If more distinctive null pointers are needed those greater than or equal to #400000 may be used.

2.4.3 DEFINING FIELD TEMPLATES

At any given time the user can have up to 36 field templates defined.

The possible names for field templates are:

the 26 letters of the alphabet: A, B, ... , X, Y, Z
the ten digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

The user can change the field template associated with a field name at any time during the execution of his program, and can even define a field template on the basis of information computed by his program.

To specify a field template the user must give two pieces of information, an offset and a bit position specification. The offset indicates the word in which the field is to be found. The address of the word containing the field is obtained by adding the offset to the pointer. Thus, if the offset is 0, the field is in the word indicated by the pointer, if the offset is 5 then the field is in the fifth word after (i.e., with a higher address than) the pointer. The displacement of a field may be negative in which case the field refers to an address which is lower than the pointer.

To specify the position of the field within a word the user can either give a 36 bit mask which has 1's in those bit positions which are in the field and 0's elsewhere, or indicate the first and last bit positions in the field.

The leftmost bit in a word is refered to as bit 0, and the rightmost is bit 35. (The high order bit is bit 0, the low order bit is bit 35.)

Thus, to indicate a field of 5 bits, starting in the fifth bit position of the word, (bit 4) the user can give either the mask #036000000000 or the bit positions 4 8. If we let "-" stand for a bit position not in the field and "*" for a bit position in the field, we can "draw" the bit specification for the field as follows:

```
----*****---------------------------
```

An example of the way in which a user can define a field template is:

```
(3 D F 7 13)
```

which defines the field name F to be associated with the field template with offset 3 and specifying bits 7 through 13. Another example is:

```
(Ø D Q #77400)
```

which defines the field template named Q to have offset Ø and be in bits 21 through 27. Diagrams of the bit position specifications of the two fields are:

```
F = -------*******---------------------- and
Q = ---------------------*******--------
```

Equivalent definitions are:

```
(3 D F #776ØØØ) and (Ø D Q 21 27).
```

2.4.4 REFERING TO FIELDS -- BUG-FIELD STRINGS

In a program a field is referenced by means of a *bug-field string* which is a sequence of 2 to 63 characters. The first character is the name of a bug (which must contain a pointer to a location within an allocated block), and the remaining characters are the names of (currently defined) fields. All but the last field must be at least 18 bits long, since they must contain pointers.

Some examples are:
AX, QZ1, R13, M1RST5

But not:
A (only one character, just specifies a bug),
1X (the first character is not alphabetic, so it is not the name of a bug)

The meaning of a bug-field string is easy to see. For example, if the bug P contains a pointer and the field X is currently defined, then the bug-field string PX refers to the field whose position is specified by the pointer in P and the field template associated with X. In particular, say P points to the word with address #1Ø1 and X has been defined by (4 D X 18 35), then PX refers to the field in the rightmost 18 bits of the word with address #1Ø5. (That is, bits 18 through 35 in that word.) a diagram of the situation is:

```
X======>    ------------------------------------ #1Ø1
            ------------------------------------
            ------------------------------------
            ------------------------------------
            ------------------****************** #1Ø5
            ------------------------------------
```

Since the field specified by PX has 18 bits, it could contain a pointer to a word within another allocated block. In that case the bug-field string PXX would refer to the X field relative to the pointer in PX. Thus, if bits 18 to 35 of word #1Ø5 contain a pointer to word #215 then PXX would be the field in bits 18 through 35 of word #221. If field 5 were defined by (1 D 5 Ø 7) then PX5 would refer to bits Ø through 7 (the leftmost 8 bits) of word #216.

In general, if B is a bug and F, G, ... ,W are currently defined fields, each of at least 18 bits length, then BFG ... WQ refers to the field specified by the pointer in the field specified by BFG ... W and the template associated with Q.

With complicated plex structures, as in the diagram below, it is possible to refer to a given field in many ways, by finding alternative chains of pointers to guide the way to the field. In the diagram below, the same field (the A field in the first word of the second block) can be refered to as:

XA or WKA or WFFA or WF3KA

THE FIELDS DEFINED ARE:

F=2[2,2Ø]	BITS 2 - 2Ø,	DISPLACEMENT 2
A=-1[5,11]	BITS 5 - 11,	DISPLACEMENT -1
K=Ø[18,36]	BITS 18 - 36,	DISPLACEMENT Ø
3=Ø[1,18]	BITS 1 - 18,	DISPLACEMENT Ø

```
                                         K
                  -------------------------------------------
                  |                    |     #147           | #1ØØ
                  -------------------------------------------

                         A
                  -------------------------------------------
                  |      |//////|                           | #146
                  -------------------------------------------
X---------->      |                                         |
                  -------------------------------------------

                                         K
                  -------------------------------------------
W---------->      |                    |    #147            | #333
                  -------------------------------------------
                  |                                         |
                  -------------------------------------------
                  |  |     #456        |                    |
                  -------------------------------------------
                           F

                           3
                  -------------------------------------------
                  | |      #1ØØ          |                  | #456
                  -------------------------------------------
                  |                                         |
                  -------------------------------------------
                  |  |     #147        |                    |
                  -------------------------------------------
                           F
```

CHAPTER 3
TUPLE FORMATS

3.1 GENERAL DESCRIPTION, FORMAT AND EXAMPLES

In writing a program there are two types of common activities, testing data, and operating on data. In general, a test involves comparing the contents of a field with the contents of another field or with a literal (constant) in the program while an operation specifies some change to be made to the contents of some field, bug, pushdown stack or I/O device. In L^6, both tests and operations are written in the same format, with the distinction between the two types of action being made on the basis of where they occur in the L^6 statement. The basic syntactic unit which is used for both tests and operations is the tuple. A tuple must specify the action to be performed and the data on which it will act.

A tuple consists of a symbolic operation code and one or more data specifications, enclosed in parentheses. The data specifications can be constants, bug names, bug-field strings and sometimes labels. The operation code is always the second element of the tuple, and the elements are always seperated by at least one blank space.

The formats are:

```
(data-1 op) or
(data-1 op data-2) or
(data-1 op data-2 data-3) or
(data-1 op data-2 ... data-n) .
```

Some examples of tuples are:

(XA1 = 15) moves the integer 15 into the field XA1 when it is used as an operation tuple (when it is used as a test tuple, it compares the contents of the field XA1 with the integer 15.

(PQ + R) adds the contents of the bug R to the contents of the field PQ and stores the result in the field PQ.

(PX - 157) subtracts the integer 157 from the contents of the field PX and stores the result in PX.

(P GT 1Ø) requests a block of 1Ø words from the storage allocator, and stores a pointer to the first word of the allocated block in the bug P.

(RS FR) returns the block whose first word is pointed to by the field RS to the storage allocator (if RS does not point to the first word of an allocated block this causes an error).

3.2 CLASSIFIED LIST OF OPERATIONS AND TESTS

3.2.1 OPERATIONS

There are several basic classes of operations in L^6:

Storage Allocation -- GT to allocate a block, FR to free a block

Assignment -- ←, = or E for assignment, IC to interchange contents

Arithmetic -- + for addition, - for subtraction, * for multiplication, / for division

Bitwise Logical Operations -- & or A for Boolean and, ! or O for Boolean or, X for exclusive or, C for complement

Input/Output -- INIT for initializing input and output files, INS for inputting character (strings), OUTS and OUTF for outputting character strings, TOUT for outputting long text for titles, FOUT for outputting text and forcing the output buffer to be printed

Subroutine Call -- DO for transfering to a subroutine

Defining Field Templates -- D for defining fields

Pushdown Stack Instructions -- SFC for saving one or more data items on the field contents stack, RFC for restoring one or more fields from the field contents stack, SFD for saving one or more field definitions on the field definition stack, RFD for restoring one or more field definitions from the field definition stack

Bit Shifting and Counting -- L for left shift, R for right shift, COL for finding the position of the leftmost one within a field, CZL for finding the position of the leftmost zero, COR for finding the position of the rightmost one, CZR for finding the position of the rightmost zero, CO for counting number of ones in a field, CZ for counting number of zeroes

Incremental Dump -- DUMP for obtaining a dump of the contents of the user-allocated storage area while the program is running.

3.2.2 <u>TESTS</u>

There are a number of tests available, including:

Equality -- = for equality, # for inequality

Algebraic Comparison -- < for less than, > for greater than, <= for less than or equal, >= for greater than or equal, and R for inclusive range

Logical Tests -- O to test for corresponding one bits, Z to test for corresponding zero bits

3.3 COMMENTS ON OPERATIONS AND TUPLE FORMATS

As should be clear from the examples, the operator is always the second element of the tuple. Since the operation codes can be sequences of letters, it would be impossible to distinguish them from bug-field strings if they did not always occur in the same place in the tuple.

There must always be at least one space between consecutive elements of a tuple. Extra spaces may be inserted for readability wherever a single space is acceptable. Tabs, linefeeds, formfeeds, and vertical tabs are equivalent to spaces except inside character strings. There can be no spaces between the characters in a bug-field string.

Almost all operations that combine data store the result in the space occupied by the first argument. (The most notable exceptions are the extended GT operation, the RFC restore field contents operation, and the extended division tuple) Thus, the first argument of an operation tuple can be either a bug or a bug-field string, but not a constant. Remember, the original contents of the first argument are modified during arithmetic operations. The other arguments of a tuple are constants, bugs or bug-field strings.

3.4 OPERATIONS ON FIELDS OF UNEQUAL LENGTH

It is common to combine fields of unequal length in L^6 operations. Conceptually, the operations are performed on the two fields as if they were right justified in a 36 bit word, and the rightmost bits of the result are stored in the result field. Thus, if field FA has 17 bits and field PQR has 13 bits, the operation (PQR + FA) takes the 17 bits from FA and puts them in the rightmost 17 bits of a word, with the remaining bits zeroed, and puts the 13 bits of PQR in another word, with the remaining lefthand bits zeroed out, and adds the two quantities. The rightmost 13 bits of the resulting 17 (or 18) bit sum are stored in the field PQR, and the other 4 (or 5) significant bits are lost. Remember that a negative number takes 36 bits so the contents of any field shorter than 36 bits is treated as a positive number. In an assignment statement, (PQR = FA) results in PQR being filled with the rightmost 13 bits of the 17 bit field FA, the other bits being lost. (FA = PQR) results in the rightmost 13 bits of the field FA getting the contents of the field PQR, and the leftmost 4 bits of PQR being set to zero. As was noted above, constants can be regarded as being right-justified in a 36 bit field. Thus, if a field can contain enough bits to represent a constant the result is what you would expect.

3.5 THE FIELD CONTENTS STACK

The field contents stack provides the programmer with a convenient way to store the local information needed for a recursive subroutine. It is possible to store blocks of words on the stack, and to restore these same blocks. The contents of several fields may be saved at one time on the stack. There is only one field contents stack, and the elements stored on it are blocks of 36 bit words. Note, that as in all L^6 data transfers, if the fields are shorter than 36 bits, the quantity stored in the stack contains the field to be stored, right justified with zeroes on the left in a 36 bit word. The user should be careful, since the stack holds blocks of words. To save the bugs A, Q and the contents of the field PRX the user can write

```
    (3 SFC A Q PRX)
```

and to restore the same items he would use,

```
    (3 RFC A Q PRX)
```

Note that the items in a block are restored in the same order as they were placed in the block, and that it is only blocks which are stacked in a "last in, first out" fashion.

CHAPTER 4
THE L^6 PROGRAM

4.1 GENERAL FORMAT OF A PROGRAM

An L^6 program is a group of one or more procedures. Each procedure consists of a sequence of statements. Roughly speaking, a statement is a sequence of operations written as tuples, which may be performed unconditionally, or which may only be performed if some conditions specified by a set of test tuples are satisfied. Normally, control flows from one statement to the next statement in a sequence, and from left to right, tuple to tuple, within a statement. In order to allow the user to unconditionally or conditionally change the flow of control in his program any statement may have a label, and it is possible to specify that control is to be passed unconditionally or conditionally to the statement with a given label.

4.2 AN EXAMPLE PROGRAM

```
        PROCEDURE LINKLIST
/THIS L6 PROGRAM  READS IN A SEQUENCE  OF CHARACTERS FROM THE
/DISK FILE  LINK.DAT IN THE USER'S  DISK  AREA,  AND  FORMS A
/LINKED LIST OF THESE CHARACTERS.  IT THEN TRAVERSES THE LIST
/AND  PRINTS OUT THE  CHARACTERS,  FREEING THE  BLOCKS  AS IT
/GOES.  READING IS TERMINATED WHEN THE CHARACTER "." IS SEEN.
/END OF LIST IS INDICATED BY A Ø IN LINK THE FIELD.

/THERE ARE TWO FIELDS IN EACH BLOCK,  THE FIELD C IS AN 7 BIT
/FIELD FOR A CHARACTER, AND THE FIELD N IS THE LINK FIELD

/B POINTS TO THE START OF THE LIST, P IS A WORKING POINTER, C
/IS USED TO HOLD CHARACTERS TEMPORARILY ON INPUT


START   THEN ('LPT:' INIT 'LINK')       /SET UP I/O
        THEN(Ø D C Ø 6)(Ø D N 18 35)    /SET UP FIELD TEMPLATES
        THEN (P GT 1)(B = P)(PC INS 1)  /SET UP FIRST BLOCK
        THEN (C INS 1)                  /GET FIRST CHAR FOR LOOP

/THE  NEXT STATEMENT  IS  A  ONE  LINE  LOOP  WHICH  READS IN
/CHARACTERS AND PLACES THEM INTO BLOCKS UNTIL IT READS A "."

LOOP    IF (C # ".") THEN (PN GT 1)(P = PN)(PC = C)(C INS 1) LOOP
        THEN (PN = Ø)  (P = B)             /SETUP  P FOR OUTPUT LOOP

/THIS  LOOP  TRAVERSES  THE  LIST,  PRINTING  CHARACTERS  AND
/FREEING THE BLOCKS

OUTLP   IF (P = Ø) THEN HALT ELSE (PC OUTS 1)
        THEN (T = P)(P = PN)(T FR) OUTLP

        END
```

4.3 THE L^6 STATEMENT

4.3.1 LABELS, COMMENTS AND GENERAL FORMAT

An L^6 statement has a free format. A statement may be labeled. A label can be any sequence of from one to ten letters or digits, beginning with a letter, with the exception of the following reserved words:

IF	IFNONE	IFALL	IFANY
IFNALL	THEN	ELSE	PROCEDURE
PROC	END	LCLB	LCLF
EXTERNAL			

Labels defined in this manner are known as local labels. In the program above all non-labeled statements started in column 8. This was done only to improve readability, since L^6 would have been perfectly happy to have the statements start in any column. It is strongly suggested that the user develop his own standard format to improve readability, and stick to that format in typing his programs.

Comments can be placed in an L^6 program by preceding them with a slash (/). All characters on a line following a / are ignored by the parser. (Major exception: / is used as the division operator, and a / that occurs as the second element of a tuple does not cause subsequent characters to be ignored.)

4.3.2 UNCONDITIONAL OPERATIONS AND TRANSFERS

The simplest form of an L^6 statement is the unconditional statement which consists of the keyword THEN followed either by a sequence of operation tuples or a label or both. Some examples are:

```
THEN (P GT 1)(PC INS 1)

THEN LAB1

THEN (PR + 3)(P = PN) LAB1
```

When an unconditional statement is encountered the tuples within it are executed from left to right, and if there is a label control is then passed to the statement with that label. (If there are no tuples and only a label, control is passed directly to the statement with that label. If there is no transfer label, then after all the tuples have been executed control flows to the next statement in the program.)

The transfer labels HALT, DONE, FAIL and * have special significance. Transfering to the label HALT will cause the program to come to a halt. DONE and FAIL are special dummy labels used to return from subroutines and procedures. * is a special label useful for creating one-line loops.

Once execution begins on a sequence of tuples, all the tuples will be executed unless an error occurs, or there is a transfer tuple (GOTO or DO) in the sequence.

The unconditional statement is not only the simplest form of an L^6 statement, it is the basis for most of the other statements. The sequence of the keyword THEN followed by either a sequence of one or more tuples, a label or both, is refered to as a THEN clause.

4.3.3 CONDITIONAL OPERATIONS AND TRANSFERS

In a program of any complexity, there will be operations that are only to be performed when certain conditions are met, or transfers of control that are to be made only under certain conditions. L^6 provides several forms for such conditional expressions. The simplest form consists of the keyword IF followed by a test tuple, followed by a THEN clause. Some examples are:

```
IF (P = ".")THEN OUTPUT

IF (PC < 3) THEN (PC + 1)(P = PN)

IF (P # Ø) THEN (PC OUTS 1)(P = PN)LOOP
```

The sequence consisting of the keyword IF and the test tuple is called an IF clause. It is the simplest form of the IF clause, more complicated ones with multiple tests are shown below.

The ELSE clause

For these simple IF statements, the THEN clause is executed if the test is true, and if the test is not true the next statement in the sequence is executed. A useful generalization is the IF statement with an ELSE clause. An ELSE clause is like a THEN clause except that it starts with the keyword ELSE and can only occur after a THEN clause in a conditional statement. The ELSE clause is executed exactly like a THEN clause, except that it is only executed if the testing condition is not met. Some examples are:

```
IF (A = 3) THEN (P + 2)(A - P)(Q = QN) LAB3 ELSE (A = 1)

IF (P = Ø) THEN HALT ELSE (PC OUTS 1)(P = PN) LOOP
```

In the first case, the THEN clause is executed only if the contents of bug A is a 3, and the ELSE clause is executed if the contents of the bug A is not 3.

Multiple Tests in an IF clause -- IFALL, IFANY, IFNONE, IFNALL

Many times it is necessary to test several conditions to see if a line of code is to be executed. L^6 provides several alternative IF clauses with different keywords to control the execution of a line, depending on the conditions which must hold for the THEN clause to be executed.

The keyword IFALL indicates that the THEN clause is to be executed only if all the tests are true.

The keyword IFANY indicates that the THEN clause is to be executed only if at least one of the tests is true.

The keyword IFNONE indicates that the THEN clause is to be executed only if none of the tests are true.

The keyword IFNALL indicates that the THEN clause is to be executed only if at least one of the tests is false.

Some examples are:

```
IFALL (P # Ø)(PC # ".") THEN (PC OUTS 1) ELSE (PD OUTS 1)

IFANY (P = Ø)(PC ".") THEN (PD OUTS 1) ELSE (PC OUTS 1)
```

```
IFNONE (PN = Ø)(R = 3)(J > K) THEN (R + J)(P = PN)

IFNALL (Q = P)(PN = R) THEN (P = PN)(R = 2)
```

4.4 THE L^6 PROCEDURE

A procedure is a block of statements to be executed as a unit. Every procedure must be named. Procedures begin with a PROCEDURE statement of the form:

```
PROCEDURE name
```

where "name" is a label. This label becomes a new type of label known as an external label. External labels are unknown inside procedure definitions and an identical label may therefore be used as a local label in any procedure. (Exception: See the EXTERNAL statement in chapter 6).

CHAPTER 5
SOME USEFUL PROGRAMMING CONSTRUCTS

5.1 ONE LINE LOOPS

There is a special symbol * which can be used as a transfer label in either a THEN or an ELSE clause. The symbol * stands for the current line. Thus, it is possible to write convenient one-line loops in L^6.

For example:

```
IF (P # Ø) THEN (PC OUTS 1)(P = PN) *
```

This one line will print all the characters in the C fields of a linked list with link field N, stopping only when a Ø link field is found.

```
IF (P = Ø) THEN HALT ELSE (P FR PN) *
```

This line of code will free all the blocks in a linked list, and halt when it reaches a link field of Ø, having returned the last block.

5.2 SUBROUTINES

L^6 provides a minimal subroutine capability. It is possible to transfer control to a line of a program and to save the location from which the transfer took place. The subroutine transfer tuple is an operation tuple (and must occur in a THEN clause or an ELSE clause) and looks like:

```
(SUBR1 DO)

    or

(SUBR2 DO FAILEXIT)
```

where SUBR1 and SUBR2 are the labels of statements which are the beginning statements of subroutines. When the first form of DO is executed, the "address" of the next tuple in the clause (the one following the DO tuple) is pushed onto the return stack and control is passed to the statement whose label is the first argument of the DO tuple (this is a case where a sequence of letters as an argument to a tuple is interpreted as a label and not as a bug-field string). In the second form of the DO tuple, two addresses are put on the return stack, the address of the tuple following the DO tuple, and the address of the statement whose label is the second argument of the DO tuple.

If the label DONE is used as the transfer label of a THEN or an ELSE clause, the return stack is popped, and control passes to the address which was stored on the top of the stack.

If the label FAIL is used as the transfer label of a THEN or an ELSE clause, the stack is popped, and if there are two elements, control is passed to the second address, the one which came from the second argument of the DO tuple. This FAIL exit feature allows the programmer to write subroutines which test certain conditions and return to one location specified by the caller if some conditions hold, and return to another if the conditions do not hold. This is very often useful when external conditions might make it impossible for a subroutine to perform its assigned task, and it is necessary for the calling routine to know about it. This FAIL exit can also be used in many other ways. If the subroutine executes the label FAIL and the calling DO tuple does not have a FAIL address, the DONE exit is taken.

Since the location of the calling tuple is placed on a stack, it is quite easy to write recursive subroutines in L^6, as long as the programmer takes care to save internal variables before calling any subroutine which might call the calling routine recursively. The field contents stack makes such saving of internal registers quite convenient. Another way to write recursive programs is to write recursive procedures using the local bug and local field feature. This will be described in chapter 6.

5.3 THE EXTENDED GT AND FR OPERATIONS FOR STACKS

The extended form of the GT and FR operation make it quite easy to make a linked stack in L^6. To insert a block on the head of a stack pointed to by the bug P, with links in field L, the programmer simply writes:

```
(P GT 1 PL)
```

and to pop an item off the stack and return it to the free list one can write:

```
(P FR PN)
```

Note that since the contents of the first argument is saved before it is modified, and the field specified by the third argument is not found until after the first argument is modified, the extended GT operation results in the original value of P being stored in the link field of the block which has been obtained from the allocator, and which is now pointed to by the bug P. A similar juggling trick occurs in the extended FR operation.

CHAPTER 6
MULTIPLE PROCEDURES

6.1 GENERAL DESCRIPTION OF A PROCEDURE

A procedure is a group of statements of the form:

```
PROCEDURE name
statements
END
```

In this description "name" is a string of one to ten letters or digits beginning with a letter. The keyword PROC may be used in place of the word PROCEDURE in the first statement. "Statements" is any number of unconditional or conditional lines. Three other special types of statements may also be included. They are EXTERNAL, LCLB, and LCLF. These will be described later.

Any number of procedures may be defined, either by loading them from a file, or by writing them online using the editor commands.

The labels defined in a procedure are local to the procedure, that is, they are not defined outside of the procedure. This means that identical labels defined in several seperate procedures will not conflict. It also means that a procedure may not branch to a line in another procedure. For example, suppose there are two procedures defined as follows:

```
          PROCEDURE ONE
          THEN TWOSLABEL
DOUBLE    THEN DONE
          END

          PROCEDURE TWO
TWOSLABEL THEN HALT
DOUBLE    IF (X < 100) THEN (X + 1) DOUBLE
          END
```

If the command

```
:RUN ONE
```

is typed an undefined label error will occur since the definition of TWOSLABEL in procedure TWO is not recognized in procedure ONE. Also, the branch to DOUBLE in TWO branches back to the label DOUBLE in TWO and not to DOUBLE in procedure ONE. The definition of DOUBLE in ONE is not used.

6.2 CALLING A PROCEDURE -- EXTERNAL LABELS

A procedure may call another procedure (or itself) in the same way subroutines are called. For example, to call procedure PRINT the line

```
THEN (PRINT DO FAILEXIT) (C INS 1)
```

can be used. This causes procedure PRINT to be executed, beginning with the first line until either DONE or FAIL is encountered. Control returns (as with subroutines) to the INS tuple in case of a DONE or to the line with the label FAILEXIT in case of a FAIL. The label FAILEXIT must be defined as a local label in the procedure containing the calling DO tuple.

The names of procedures are not normally recognized within any procedure's statements. Therefore, a statement called the EXTERNAL statement is used. It has the form:

```
EXTERNAL proc-1,proc-2,proc-3, ... ,proc-n
```

The proc-1,proc-2,proc-3, ... ,proc-n is a list of procedure names. This statement causes these names to be interpreted as external rather than local labels. In the previous example, the procedure containing the DO tuple must contain the statement

```
EXTERNAL PRINT
```

to indicate that PRINT is the name of a procedure rather than a subroutine. EXTERNAL labels can not be defined at the beginning of a line, nor can they be used as branches, FAIL exits, or in the GOTO tuple. They are used only as the first argument of the DO tuple.

Procedures are called using the same stack which is used for subroutine calls. Therefore, when one of the labels DONE or FAIL is encountered it applies to the last DO tuple executed, whether it called a subroutine or a procedure. A DONE or a FAIL at level Ø (when no subroutine or procedure calls are in effect) acts exactly like a HALT, except that the message

```
        DONE AT LEVEL Ø

             or

        FAIL AT LEVEL Ø
```

is typed. This is so that procedures can be written which may be run either by being called by other procedures or by being executed by either the RUN or EXECUTE commands.

6.3 LOCAL BUGS AND FIELDS

When a procedure is called by a DO tuple, it is possible to specify bugs and field definitions which will be saved on the pushdown list along with the procedure call and restored when the procedure is exited by a DONE or FAIL. This is done with the LCLB or LCLF statements. They look like:

```
        LCLB bug-list
        LCLF field-list
```

EXAMPLES ARE:

```
        LCLF A,1,5
        LCLB A
        LCLB A,B,M,Z,R
```

This feature is especially useful for writing recursive procedures. It is also helpful for writing library routines which should not modify any bugs or fields except those used to pass values back to the calling routine.

6.4 A SAMPLE PROGRAM WITH MULTIPLE PROCEDURES

```
        PROCEDURE FACT

/ THIS IS THE MAIN PROCEDURE WHICH LISTS THE FACTORIALS
/ OF THE NUMBERS BETWEEN 1 AND 13 ON THE TELETYPE.

        EXTERNAL PRINTBUGC,FACTORIAL
        THEN ("TTY:" INIT "TTY:")           /SETUP I/O
        THEN (N = 1)                        /INITIALIZE A COUNTER
START   IF (N > 13) THEN ('!C!L' OUTF 2) HALT
        THEN ('!C!L' FOUT) (C ≤ N) (PRINTBUGC DO) ("     " TOUT)
        THEN (N + 1) (FACTORIAL DO)
        THEN (C SFC) (PRINTBUGC DO) (C RFC) START
        END

        PROCEDURE FACTORIAL

/ THIS PROCEDURE TAKES A NUMBER IN BUG N AND
/ COMPUTES N! IN BUG C.  THIS IS A RECURSIVE
/ PROCEDURE.

        EXTERNAL FACTORIAL
        LCLB N
        IF (N = 1) THEN (C ≤ 1) DONE
        THEN (N - 1) (FACTORIAL DO) (C * N) DONE
        END

        PROCEDURE PRINTBUGC

/ THIS PROCEDURE IS ALSO RECURSIVE. IT PRINTS
/ THE NUMBER IN BUG C IN DECIMAL.

        EXTERNAL PRINTBUGC
        LCLB B
PR      THEN (C / 10 B) (B ! "0")    /GET NEXT DIGIT
        IF (C # 0) THEN(PRINTBUGC DO) (B OUTS 1) DONE
        THEN (B OUTS 1) DONE
        END
```

6.5 LOCKED PROCEDURES

Inside the L^6 interpreter, both the source line and a special code called the object code which is made from the source line is stored. Only the object code is necessary to run the procedure. The source code is stored to allow procedures to be listed and modified. For this reason a procedure may be loaded without the source. This will typically save about half of the core which is otherwise needed to store the program. This is done by placing the word LOCK is square brackets at the end of the procedure statement. An example of this is:

```
PROCEDURE FACTORIAL [LOCK]
```

Since the procedure statement cannot be edited, a special command called the LOCK command is provided. LOCK takes a list of procedure statements as arguments. For example:

```
:LOCK FACTORIAL,FACT,PRINTBUGC
```

If no arguments are typed, all of the procedures are locked. This command has no effect on the procedure except that the option [LOCK] is added to the procedure statement. When the procedure is saved this will appear in the output file.

When a procedure with the option LOCK is loaded, it is loaded without the source. Therefore after loading the procedure it cannot be listed, edited, or saved.

CHAPTER 7
ADVANCED L^6 LANGUAGE FEATURES

7.1 LABEL VARIABLES

A bug-field string preceded by an at sign (@) may be used anywhere in place of a label. This construct is called a label variable. It is interpreted as a bug field string that contains a pointer to a two word block which contains the label one wishes to branch to. The two word block must contain the label left justified in seven bit ASCII padded with zero characters to ten characters. A zero character is a character composed of seven zero bits. For example assume bug B contains a pointer to a two word block that contains "PRINTER" left justified

```
                  -------------
B  ----------->  | P R I N T |
                  -------------
                 | E R       |
                  -------------
```

and assume the following statement is executed:

```
THEN (A = 2) @B
```

L^6 will first assign the value of 2 to bug A and then attempt to branch to PRINTER. The label variable may also be used in a DO tuple. For example if N is the field Ø[18,35] (displacement zero, bits 18-35) then

```
THEN (@BN DO)
```

will use the contents of the N field of the word pointed to by B as a pointer to a two word block that contains the label of the subroutine or procedure.

7.2 STRING VARIABLES

The other L^6 construct which involves bug-field strings is the string variable. L^6 recognizes as a string variable any bug-field string that is preceded by a dollar sign ($). String variables are similar to label variables in that the bug-field string is a pointer to a block of words. However, a string variable may be any number of words long. The block contains five ASCII characters left justified in each word and the string is terminated by a zero character (a byte of zeroes). String variables can be used in the INIT, OPENI, OPENO, FOUT, and TOUT tuples. String variables allow the user to specify output text and file names dynamically. For example:

```
     THEN($T FOUT)
```

will output the ASCII characters pointed to by bug T.

Assume that bug B points to the two word block that contains "TTY12:" left justified followed by a byte of zeroes.

```
     B  ---------->  -------------
                     | T T Y 1 2 |
                     -------------
                     | :         |
                     -------------
```

Now if the following expression were executed:

```
     THEN(2 OPENO $B)
```

then teletype twelve would be open for output on channel two.

CHAPTER 8
EXTENDED I/O

8.1 OPENO AND OPENI

The OPENO and OPENI tuples provide the user with the ability to do I/O to a multitude of devices. Among these are teletypes, disk, DECtape, paper tape reader and punch, card reader and punch, and others. These two tuples take as arguments the channel number, the device and file name, and optionally, the number of buffers and the data mode. The exact format is described in chapter twelve.

A channel number is a logical association between a specific file and a number. It is used to allow generality in the specification of input and output streams. For example, assume that we have a program to add a list of numbers together and output the sum and we wish to use this same program so that it would input its augends from both the teletype and a disk file. If each individual input or output tuple contained a reference to the file then a major rewrite of the program would be necessary every time we wished to change from the teletype to the disk file or vice versa. However, since channel numbers are used instead of the actual file names all that need be changed is the one tuple that makes the association between the file and the channel number. This is much easier to do. The channel number must be in the range Ø-1Ø. Channel Ø is the INIT tuple´s channel. This means that an OPENO (without optional arguments) using channel zero is equivalent to specifying the same device and filename in the output portion of an INIT tuple. Further, OPENI on channel Ø is analogous to the OPENO on channel Ø, however, it affects only the INIT´s input file specification. The other channels (1-1Ø) may not be used simultaneously for input and output as can channel Ø.

The file specification must be provided in DEC standard form:

```
"dev:filename.ext[ppn]<prot>"
```

dev: is the device name, filename.ext is the filename and the extension, [ppn] is the project programmer number, and <prot> is the protection of the file. When specifying an input file the protection code is ignored. It does, however, specify the protection to be associated with an output file.

The specification of the number of buffers is optional, although it is sometimes convenient to specify the number of buffers explicitly. A buffer is an area in memory that is used as an intermediate storage area. Usually this storage area holds values that have come from a device or are destined to be sent to some device. The reason for buffers is that devices are usually shared in a multiprogramming environment. This means that a program may not necessarily be able to do input or output exactly when it wants to. In fact it almost always has to wait in line to use resources. But, with buffers instead of having to wait in line to output just one character, the program may save up thirty or forty characters. This means the program will not have to wait in line as often and can therefore do whatever other tasks are required of it.

The last argument is also optional, but if the user wishes to specify the data mode he must also specify the number of buffers (the third argument). If the user does not want to specify the number of buffers explicitly then he may request a non-positve number of buffers. This will allocate a default number of buffers as specified by the monitor. The only legal data modes are the buffered mode (as defined in the _DECsystem1Ø Monitor Calls_ reference manual). These are:

CODE	NAME
#Ø	ASCII
#1	ASCII LINE
#1Ø	IMAGE
#13	IMAGE BINARY
#14	BINARY

If the same device is opened more than once in different modes then only the last specified mode is in effect. If no data mode is specified then ASCII is the mode assumed.

The last argument can actually be used to set the entire status word used in the opening of the file. This will allow such things as echo control and full character set for the teletype. Certain bits in the status word may cause an immediate end of file condition or the simulation of device errors. Therefore the programmer should be familiar with I/O programming before setting up his own status word.

8.2 EXTENDED INS, OUTF, OUTS, FOUT, AND TOUT

Each of the data transfer tuples can have an additional optional argument. This argument is the channel number. Its purpose is to specify the data stream to which the I/O operation should to be channeled. The channel number must be in the range Ø-1Ø and it corresponds directly to the file specified in the last OPENO or OPENI tuple that used the same channel number.

When an INS tuples reads more than one character, it shifts the previous character left seven bits and OR's in the next. If the byte size for the data mode is not seven bits, this may give useless input. Because image, image binary, and binary modes read data as thirty-six bit words, the second argument should be a one for these modes.

8.3 THE EOF TUPLE

The EOF tuple is used to sense an end of file condition. If an end of file condition has occured then the tuple is true, otherwise it is false. Its only argument is the channel number and it must be in the range Ø-1Ø. If the channel specified is an output channel the end of file condition will never be raised. If the channel selected is channel zero then only the input side of the INIT tuple is tested. This tuple is useful for checking to see if all of the data has been read in. If the end of file condition is raised you are assured that no more data is to be found in that file. This is not necessarily the case for a teletype because a control-Z may have been typed accidentally, thereby signalling an end of file prematurely.

Since an INS tuple can read up to five characters, it is possible that an end of file can occur before enough characters are received to satisfy the request. In this case L^6 supplies ASCII nulls (zero characters) at the right of the field to finish the input. However, any attempt to execute another INS tuple after an end of file occurs will result in an error message.

CHAPTER 9
THE L^6 INTERPRETER

9.1 TYPING IN COMMANDS

When L^6 is ready to accept a command it responds by typing a colon (:). Any of the commands which are described later in this chapter can then be input. When lines are typed in at the command level there are several control characters which can be used to correct typing errors. A rubout deletes the last character which has not already been deleted. This character is echoed in place of the rubout. Since L^6 handles its own rubouts, the characters deleted are not enclosed in backslashes as they are with the monitor. Typing a control U (↑U) deletes the entire command string and reprompts with the colon. Typing a control G (↑G) retypes the command string. This feature is useful when the input line has been cluttered by several rubouts.

Both commands and switches (words preceded by slashes which change the action of a command) can be abbreviated to any string which is nonambiguous. For example, PROCEDURES can be abbreviated to PROC. Using PRO as a command will result in an error message, since it could be either PROCEDURES or PROMPT.

9.2 LINE SPECIFICATIONS

Many of the commands take an argument which specifies a line or group of lines in the L^6 program. A line specification has the form:

```
procedure;line      or      procedure;line-1,line-2
```

The first of these specifies one line. The second one denotes a group of lines, those between line-1 and line-2 in the specified procedure.

One way of specifying lines is to give an absolute line number. The first line of a procedure is line 1 and the lines are numbered in ascending order from there. The line number of a lines may change as lines before it are inserted or deleted. A line may also be referenced by any label which it defines. A label may be modified by adding or subtracting a constant from it.

The dollar sign ($) may be used in place of a label to denote the last line of the procedure. Some examples should help to clarify this.

ONE;5	The fifth line of procedure ONE.
SCAN;1,10	The first through tenth lines of procedure SCAN.
REVERSE;GEAR	The line which defines label GEAR in procedure REVERSE. Produces an error if GEAR is not defined.
SYSTEM;CRASH-5	The fifth line before the line which defines the label CRASH in procedure SYSTEM.
TRAV;TREE+2,$-1	All lines between the second line after the line which defined TREE in TRAV and the line before the last line of the procedure TRAV.

The procedure name and the following semicolon may be omitted from the line specification. In this case the current procedure is assumed. The current procedure is not defined when the system is clear. When a procedure is specified in any command it becomes the current procedure. When a program is run the procedure in which execution stops becomes the new current procedure. For example:

```
:LOAD SAMPLE

NO ERRORS DETECTED
10P CORE USED


:LIST 5
? NO CURRENT PROCEDURE

:RUN MYPROG

HALT AT LEVEL 0
12P CORE USED
```

```
:LIST 5
      5:        THEN (A = 1)  /LINE 5 OF MYPROG

:PROCEDURES
     FACTOR           MYPROG           TRAVERSE

:PROCEDURE FACTOR

:LIST $
     12:       /THIS IS  THE LAST LINE OF FACTOR

:LIST MYPROG
? 'MYPROG' IS A UNDEFINED LABEL

:LIST MYPROG;

      1: /THIS IS THE PROCEDURE DEFINITION
      2: /OF MYPROG


:LIST TRAVERSE;$
    100:      /THIS IS THE LAST LINE OF TRAVERSE

:LIST 2
      2:        THEN (C GT 2) (V + 3) /THE SECOND LINE OF TRAVERSE
:EXIT

  EXIT
```

In the commands INSERT, DELETE, and LIST the list specification may be omitted. This has a different meaning in each case which is described with the command.

9.3 COMMANDS

The L^6 system provides a substantial number of commands to facilitate its use. They provide the user control over the execution, editing, and debugging of his L^6 program.

BREAKPOINT
CLEAR
CLOSE
CONTINUE
CORE
DDT
DELETE
DUMP
EDIT
ERRORS
EXECUTE
EXIT
HELP
INSERT
LABELS
LIST
LOAD
LOCK
PROCEDURES
PROMPT
REMOVE
RENAME
REPLACE
RUN
SAVE
STORE
UNDEFINED
UNLOCK

The commands and their descriptions follow.

BREAKPOINT --- This command provides the user with the capability to interrupt his program flow, examine the condition of his program, and then continue execution.

<u>Command</u> <u>format</u>: BREAKPOINT#brk proc;line:tuple

"Brk" is the number of the breakpoint to be assigned, and "proc" is the procedure name. "Line" is the line specification, and "tuple" is the tuple number. The format of the tuple number is "T", "E", or nothing followed by a number. "T" denotes the THEN clause and "E" denotes the ELSE clause. If neither is specified then "T" is assumed. If no specification is provided, i.e. everything is omitted, then the breakpoints currently in use are listed.

Examples:

B#4 FUN;5:T3	Breakpoint THEN tuple 3 of line 5 of subroutine FUN and use breakpoint number 4
B GEORGE;$:E1	Breakpoint ELSE tuple 1 of the last line in procedure GEORGE
B	List all the breakpoints

CLEAR --- This command re-initializes the L^6 system by freeing all accumulated core, deleting all opened files, resetting all I/O, and re-initializing the stacks. If the contents of files which currently open are to be saved, they should be closed. This can be done selectively, using the CLOSE tuples, or all files may be closed using the CLOSE command.

<u>Command</u> <u>format</u>: CLEAR num

The argument gives the amount of core to be assigned to the L^6 low segment. When the interpreter is started for the first time a default amount of core is used. Giving an argument on the CLEAR command changes this value for all subsequent CLEARs unless changed by a later CLEAR command with an argument. The CLEAR command does not affect the user core maximum unless the CLEAR has an argument which is larger than the present value. In this case the core maximum is set to a value higher than the new clear size.

The argument is a decimal integer optionally followed by "K" or "P" to denote either K (1Ø24 words) or pages (512 words). If neither "K" nor "P" is used "K" is assumed.

CLOSE --- This command closes all of the files which are currently open. It should be used before CLEAR if the contents of the currently open files are to be saved.

<u>Command</u> <u>format</u>: CLOSE

CONTINUE --- This command restarts the program at the place last interrupted by a breakpoint.

<u>Command</u> <u>format</u>: CONTINUE [num]

The only argument this command accepts is a numeric continuation count. It specifies the number of times to bypass the last breakpoint encountered without stopping.

CORE --- This command sets the user core maximum.

<u>Command</u> <u>format</u>: CORE [num]

"Num" is the maximum amount of core the low segment may contain. It is a decimal integer followed by "P" or "K". A "P" specifies pages and a "K" specifies thousands of words. If neither is present "K" is assumed. The command:

```
CORE 5K
```

will allow a maximum of 5K for the low segment.

If the core command is given without an argument, the amount of core in use is typed, along with the system and user core maximums (for more information see the description of core allocation in section 9.4).

DDT --- This command is used to enter DDT, DEC's debugger. This command is included for system maintenance.

Command format: DDT

DELETE --- This command deletes lines from a procedure.

Command format: DELETE line specification

The "line specification" selects one or a group of lines to be deleted. If more than five lines are to be deleted

```
ARE YOU SURE?
```

is typed. Typing Y causes the lines to be deleted. Any other character outputs

```
% NOT DONE
```

and no other action is taken. If the procedure name and semicolon is used, but no line numbers, the entire procedure is deleted.

The delete command may be used to edit a procedure after execution has been suspended at a breakpoint. However, there are several restrictions. If a line which contains a breakpoint is deleted, the breakpoint is removed and warning message is typed. If a line which contains a DO tuple that has a call on the return stack is deleted, the message

```
%STACK DAMAGE -- CAN'T CONTINUE
```

will be typed. This means that the execution cannot be continued by the CONTINUE command. If a LCLB or a LCLF is deleted and the procedure in which it occures is pendent on the stack then the message STACK DAMAGE is printed. Finally, if the line on which execution was suspended is deleted, the execution cannot be continued.

Examples:

DELETE ARROW;	deletes procedure ARROW
DELETE ARROW	deletes the line with the label ARROW from the current procedure
DELETE 5,$	deletes all lines after line five from the current procedure

DUMP --- This command allows the user to examine the current state of the program. By the use of switches the user may dump any or all of the following areas: the BUGS, CORE, FIELD CONTENTS STACK, FIELD DEFINITION STACK, FIELDS, I/O channels open, or a traceback of the subroutine calls.

Command format: DUMP/sl/s2 .../sn

/sl/s2.../sn are the switches that determine what areas if any are to be dumped.

Switch	Description
/ALL	Provide a complete dump of the state of the user´s program. (This is the default switch if none other is specified.)
/BUGS	Dump the bugs
/CHARACTERS	Provide a character dump with whatever else is being dumped.
/CORE	Dump the user-allocated storage area
/FCS	Dump the field contents stack.
/FDS	Dump the field definition stack.
/FIELDS	Dump the currently defined fields.
/FILE:filename	Dump onto the file specified.
/IO	Dump the currently open I/O channels.
/NARROW	Dump in a narrow for format. (For use with a terminal)
/NOCHARACTERS	Do not do a character dump
/NONE	Dump nothing.
/SPACE:n	Space the dump "n" number of times in between each line. (Ø < n < 1Ø)
/TRACE	Provide a traceback dump of the subroutine and procedure calls with local bugs and fields.
/WIDE	Dump in a wide format. (For use with the line printer)

EDIT --- The EDIT command opens the line specified for character editing. The same restrictions which apply to the DELETE command about lines which are suspended in execution also applies to editing lines. For information on character editing see chapter 10.

<u>Command</u> <u>format:</u> EDIT line specification

ERRORS --- This command allows the user to list all of the errors in a procedure.

<u>Command</u> <u>format:</u> ERRORS procedure

If the procedure name is omitted the current procedure is assumed.

EXECUTE --- This command allows the user to begin execution anywhere within his program.

<u>Command</u> <u>format:</u> EXECUTE proc;line:tuple

The arguments are the same as in the BREAKPOINT command. Where "proc" is the procedure name, "line" is the line number, and "tuple" is the tuple number preceded by E for an ELSE tuple or by either T or nothing for a THEN tuple. This command differs from the RUN command in that the user allocated storage area and the stacks are not cleared.

EXIT --- This command allows the user to leave the L^6 system. When it is executed all I/O is completed, the files are closed and kept, and the DECsystem monitor is given control.

<u>Command</u> <u>format:</u> EXIT

HELP --- This command provides helpful descriptions of the different commands in the system.

Command format: HELP [item]

"Item" is the name of the help text wanted. If no argument is provided a short description of the help command will be provided. HELP * lists all of the topics and commands available.

INSERT --- This allows the insertion of lines into procedures.

Command format: INSERT line specification

Lines are inserted after the line specified. When the INSERT command is typed L^6 responds by typing the line number of the line to be inserted followed by a colon. If just the procedure but no line number is given, the lines will be inserted at the beginning of the procedure. If the procedure is undefined this will define it. When a line followed by a carriage return is typed, the editor will respond with the next line number. To terminate the insert type ↑Z (control-Z, not up arrow Z). This will immediately return to the top level. Any partially typed line not terminated by a carriage return is discarded. Examples are:

```
:INSERT 5
      6: /INSERTED INTO THE CURRENT PROCEDURE
      7: /AS A COMMENT
      8:
      9: ↑Z

:INSERT NINE;$
    1ØØ:    /THIS LINE IS APPENDED TO PROCEDURE NINE
    1Ø1:↑Z

:INSERT TREE;
      1: THEN HERE /THIS BECOMES THE FIRST LINE OF TREE

:INSERT TREE
? 'TREE' IS AN UNDEFINED LABEL

:
```

LABELS --- This command lists all labels used in all the procedures. If the label is prefixed by an asterisk (*), it is an external label. Labels prefixed by a minus sign (-) are undefined.

Command format: LABELS

LIST --- This command causes lines to be listed either to a file or on the teletype.

Command format: LIST lines/s1/s2 .../sn

The argument "lines" is actually a list of line specifications seperated by ampersands (&). If the argument list is null then all procedures will be listed. The argument list is scanned from left to right. This implies that any explicit procedure name applies to any specifications to the right of the one with the explicit declaration.

Examples:

LIST 5&16	List lines 5 and 16 of the current procedure
LIST THIS;&THAT;	List procedure THIS and procedure THAT
LIST 5&Z1;5,10&16&Z2;	List line 5 of the current procedure, lines 5 through 10 and 16 of procedure Z1, and all of procedure Z2. The current procedure is changed to Z2.

The following table is a concise description of the switches available.

SWITCH	DESCRIPTION
/FILE:filename	List onto the file specified
/HEADERS	Procedures will be listed with a PROCEDURE and an END statement
/LABELS	All labels currently defined are listed after the statements specified in the LIST command
/NOHEADERS	Procedures will not be listed with a PROCEDURE and an END statement
/NOPAGES	The output will not be divided into pages
/PAGES	The output will be divided into pages with a header at the top of each page

If neither /HEADER nor /NOHEADERS is specified the default is /NOHEADER if the listing is to a teletype and /HEADERS if the /FILE switch is used. However, the default is always /HEADERS if the entire procedure is listed. If neither /PAGES nor /NOPAGES is specified, the default is /NOPAGES for the teletype and /PAGES if the /FILE switch is used.

Examples:

```
:LIST MAIN;1 & READ;4,5 /TITLES

          PROCEDURE MAIN
        1:THEN ('TTY:' INIT 'TTY:')
          END

          PROCEDURE READ
        4:THEN (C INS 1)
        5:IFANY (C = " ") (C = #11) THEN (C INS 1) *
          END


:LIST MAIN;1 & READ;4,5 /NOTITLES

        1:THEN ("TTY:" INIT "TTY:")

        4:THEN (C INS 1)
        5:IFANY (C = " ") (C = #11) THEN (C INS 1) *
```

LOAD --- This command loads and parses the user routines so that they may be executed by the L^6 system.

Command format: LOAD [outfile=]file-1,file-2, ... ,file-N/Sw

"Outfile" is the output file on which the listing and errors are written. If the output file is omitted then only the errors are outputted to the teletype. Also, if the extension is omitted, it will be assumed to be "LST".

"File-1,file-2, ..., file-N" are the files which are to be loaded consecutively. The LOAD command accepts complete file specification, so that FUN.L6K[3,4] is a legal file specification. If the extensions are omitted L6 is assumed.

The switch "sw" if present, may be either /LOCK or /NOLOCK. LOCK means that all procedures loaded are to be locked, even if they do not have the option LOCK in the procedure statement. NOLOCK loads all procedures without locking them regardless of whether the option LOCK appears in the procedure statement. The LOCK switch is especially useful in batch where the ability to edit and list the program is not necessary.

LOCK --- This command causes the option LOCK to appear in the procedure statement of one or all procedures.

<u>Command</u> <u>format:</u> LOCK [procedure]

The procedure specified is given the option LOCK. If the procedure name is omitted all procedures are locked. In this case L^6 will type

ARE YOU SURE?

Typing a Y followed by a carriage return locks all procedures, typing anything else returns L^6 to the command level with no other effect.

PROCEDURES --- If no argument is provided, this command lists all procedures which are either defined or referenced. Undefined procedures are preceded by a minus sign (-). If an argument is used the current procedure is changed to the procedure specified.

<u>Command</u> <u>format:</u> PROCEDURES [name]

PROMPT --- This command changes the prompt character associated with the user´s teletype input.

<u>Command</u> <u>format:</u> PROMPT character

Character is any special character. If no prompt is desired the command "PROMPT NONE" should be used.

REMOVE --- This command removes breakpoints.

<u>Command</u> <u>format:</u> REMOVE number

The only argument is the number of the breakpoint to be removed. If no breakpoint is specified the message:

```
     ARE YOU SURE?
```

Is printed. If Y is typed all the breakpoints will be removed, otherwise this command does nothing.

RENAME --- This command is used to change the name of a procedure.

<u>Command</u> <u>format</u>: RENAME newname←oldname

The procedure called "oldname" is renamed to "newname". An equal sign (=) may be substituted for the back arrow.

REPLACE --- This command in similar to INSERT except that lines are replaced beginning with the specified line. The line on which the ↑Z is typed in not affected.

<u>Command</u> <u>format</u>: REPLACE line-specification

EXAMPLE:

```
:REPLACE HASHCODE;3
      3:THEN (A ← B)  /REPLACES LINE 3 OF HASHCODE
      4:THEN (A ← Z) (A + 2) /REPLACES LINE 4
      5:↑Z
```

RUN --- This command first lists any undefined symbols. Then the user-allocated storage area and the stacks are cleared. Finally, the procedure specified is run. If no procedure name is specified the current procedure is assumed.

<u>Command</u> <u>Format</u>: RUN procedure

SAVE --- This command is exactly the same as LIST except that line numbers are not listed and the options /HEADERS and /NOPAGES are always in effect and may not be overridden. The SAVE command produces a listing which can later be read by the LOAD command.

<u>Command</u> <u>format</u>: SAVE lines/sl

STORE --- This command removes the L^6 high segment and then returns to the monitor. The low segment may then be saved using the SAVE or SSAVE monitor commands (for details see the section on core allocation).

<u>Command</u> <u>format</u>: STORE

UNDEFINED --- This command lists all undefined labels.

<u>Command</u> <u>format</u>: UNDEFINED

UNLOCK --- This command will remove the option LOCK from a procedure statement to which it has been added by the LOCK command. It has the same format as the LOCK command.

<u>Command</u> <u>format</u>: UNLOCK [procedure]

"Procedure" is the name of the procedure to be unlocked.

9.4 CORE ALLOCATION

There are two major divisions in the core area of a user. The first is the L^6 interpreter itself. It is stored in the high segment and presently occupies about 8K. It is sharable, that is, if several L^6 users are logged in at once, they all use the same high segment. The second area, the low segment contains the L^6 procedures, the user-allocated storage area, and the stacks.

The high segment is fixed in length. However, the low segment is variable. When the system is first started, the low segment is set to a default size (this is set at system assembly time). When the requirements of the program require more core than is currently present, more is obtained from the monitor and the size of the low segment is increased. This expansion process requires substantial overhead. Therefore, the CLEAR command can take an argument which is the amount of core to be assigned to the low segment. Once set, this new value is used until another CLEAR command with an argument is used.

Since the low segment is expanded automatically, it is possible for a program to expand until it reaches the limit set by the monitor. However, most programs do not need nearly this much core. To prevent "run-away" programs, that go into an infinite loop which allocate core, or have infinite recursion, the user may set a core limit on the size of the low segment data area. The command to do this is the CORE command with a numeric argument. When an attempt is made to allocate a block of core which would exceed this limit the message

```
% USER CORE MAXIMUM OF nnn EXCEEDED
```

is outputted, where nnn is the current user maximum. In batch, control returns to the command level. Online, however, the message:

```
ADDITIONAL CORE:
```

is typed. Typing a carriage return return control to the command level. Typing a number optionally followed by a "K" or a "P" adds that amount of core to the user maximum and continues the last command. If neither "P" or "K" is specified "K" is assumed.

Another maximum is the system core maximum. This is set by the monitor and gives the maximum amount of core which any job may obtain. When it is exceeded the message:

?SYSTEM CORE MAXIMUM OF nnn EXCEEDED

Sometimes, especially after extensive editing, storage in the interpreter becomes fragmented. Saving the program, clearing the interpreter, and reloading may solve the system maximum problem.

Typing the core command with no argument prints the amount of core used and the current maximums. For example:

```
:CORE

10P OF CORE USED
TOTAL CORE:      26P    [INCLUDING 16P HIGH]
USER MAXIMUM:    30P
SYSTEM MAXIMUM: 100P
```

This means that the low segment is 10 pages long (on the KA10 the size is printed in K). The total amount of core used is 26P, 16P by the L^6 interpreter itself. The user core maximum is 30P. This is the maximum size of the low segment only. The system core maximum is 100P. This includes both the interpreter and the low segment.

CHAPTER 10
CHARACTER EDITING

This chapter describes the character editor which is used to make changes within lines. It is called by the EDIT command as described in chapter 9. The editor will respond by typing

```
     <EDITING LINE nnn>
```

"nnn" is the number of the line being edited. The character editor then responds with a plus sign (+) to signal its readiness to accept input. The editor accepts a sequence of commands, each of which is one letter long. Some commands take one or more strings as arguments. A string consists of all the characters following the command up until the next altmode or carriage return. A line of input is terminated by a carriage return. Some commands also take numeric arguments. These proceed the command.

While a line is being edited, a pointer is kept to it. This can point to the beginning of the line, the end of the line, or between any two characters.

One useful command is the T command. This command types the line with a percent sign (%) at the current position of the pointer. The S command can have both a numeric and a string argument. Its format is:

```
     nSstring
```

This moves the pointer after the n'th occurence of "string". If "string" is not found an error message is printed and the pointer is moved to the beginning of the line.

The I command is another useful command. It has the form:

```
     Istring
```

This causes "string" to be inserted at the pointer. The pointer is left after the inserted string. The D command is used to delete characters. It has the form:

```
     nD
```

This deletes the n characters following the pointer if n is positive. If n is negative, the n characters before the pointer are deleted. An example of the use of these four commands is:

```
:LIST IOSET;1
        1: THEN ('LPT:' INIT 'PROG.DAT')
:EDIT 1
        <EDITING LINE 1>
+T
        1:%THEN ('LPT:' INIT 'PROG.DAT')
+S'
+T
        1:THEN ('%LPT:' INIT 'PROG.DAT')

+4DT
        1:THEN ('%' INIT 'PROG.DAT')

+ITTY:$T
        1:THEN ('TTY:%' INIT 'PROG.DAT')
+E


:
```

Notice that commands which do not take a string argument may be followed immediately by another command. Those which do must have an altmode (represented as a $, on some terminals it is called escape) to terminate the string.

There are three commands which search for a string (I, K, and R). If the search argument to any of these commands is null (has no characters), the string which will be used is the one used in the previous search command. A null string may not be used if no previous search has been done.

A complete list of the commands follows:

COMMAND	EFFECT
A	Move the poiner to the end of the line
nB	Move the pointer back n characters
nD	Delete the n characters after the pointer If n is negative, delete n characters before the pointer.
E	Exit from the character editor.
nF	Move the pointer forward n characters.
Istring	Insert "string" at the pointer. The pointer is left pointing after the inserted string.
J	Restores the line to its original state and reopens it for edit.
nKstring	Deletes the n´th occurence of "string". If n is omitted it is assumed to be one.
N	Moves to the next line in the procedure and open it for edit.
nRstring$string	Replaces the n´th occurence of the first string with the second.
nSstring	Move the pointer after the n´th occurence of "string".
T	Type the line with a % at the pointer
X	Restore the line being edited to its original state and return to the interpreter.

CHAPTER 11
DEBUGGING

Rarely does a program run correctly the first time. It almost always needs to be debugged. Since this topic is so important the rest of this chapter describes the commands provided to make the task of debugging easier.

Once all of the syntax errors have been found and the program loads correctly, the user is anxious to see if his program will execute correctly. He does this by typing the keyword "RUN" followed by the name of the procedure he wish to execute. Then L^6 begins execution of the named procedure and does not stop until either an error is encountered or a halt is executed. If an error occurs it is usually evident from the message what went wrong and the cause can many times be removed quickly. However, if the program terminates execution by encountering a HALT and it did not do what was expected of it, then the programmer has a problem on his hands. (It is at a time like this that the programmer wishes that he had never heard of the language he is programming in.)

The first thing that come to mind when trying to debug a program is to step through the program and examine what changes the execution of two or three tuples makes. This can be done by using the BREAKPOINT and CONTINUE commands. The BREAKPOINT command specifies a place in the program to stop execution and the CONTINUE command causes the program to continue execution from the last breakpoint.

The user has available twenty breakpoints. Every breakpoint has associated with it a number (between 1 and 2Ø). Through this number a breakpoint may be refered to uniquely. When a breakpoint is set and a breakpoint number is not specified the first free breakpoint will be assigned. If by chance all of the breakpoints are in use and the user wishes to set still another he may not unless he specifies the number of a breakpoint to reuse. This also provides the ability to change breakpoints. Breakpoints are removed by using the REMOVE command followed by the number of the breakpoint to be cancelled. If no argument is provided all of the breakpoints are removed.

The CONTINUE command continues the execution of the program from the last breakpoint. It takes as an optional argument a count. This count specifies the number of times to continue past the previous breakpoint without stopping. In effect, this allows the user to execute a loop a number of times and then stop. CONTINUE with no argument is equivalent to CONTINUE Ø. While execution is suspended because of a breakpoint any command can be used. However, some of the editor commands can cause either

```
    ?  STACK DAMAGE -- CAN'T CONTINUE
or
    ?  BREAKPOINT REMOVED
```

to be printed. These cases are described in the delete command in chapter 9.

Finally, the last command provided to make debugging easier is the EXECUTE command. This command allows the user to begin execution anywhere within any procedure. Its arguments are exactly the same as for the BREAKPOINT command.

When one is debugging a program it is usually necessary to execute some tuples which are not in the procedure being executed. This can be done through the use of direct execution. This is the execution of an actual unlabeled statement as a command. This statement is executed exactly as if it were a statement in whatever procedure is current. This means that whatever labels are defined in the current procedure may be used in the direct execution statement. If there is no current procedure then the only labels which may be used are the names of the procedures. These labels may only be used as external labels. Thus when there is no current procedure it appears that there was an implicit EXTERNAL statement with all of the names of the defined procedures as arguments to it.

CHAPTER 12
THE L^6 TUPLES

This chapter describes each operation and test in detail.

<u>Notations Used in Tuple Descriptions</u>

m	modified data area (bug name or bug-field string)
l	a literal constant (integer, character string, hexadecimal number)
c	contents of a field or bug used but not modified (bug name or bug-field string)
cl	contents of a field or bug, or literal constant used but not modified (literal, bug name or bug-field string)
int	positive integer
f	field name

The tuple descriptions are grouped according to the categories in section 3.2.

12.1 STORAGE ALLOCATION AND RELEASE

ALLOCATING A BLOCK:

Tuple Format:

```
(m GT cl)
(m-1 GT cl m-2)
```

Description:

The first format allocates a block whose size is determined by the value of "cl" and places a pointer to the first word of the block in "m".

The second format does several things. First, it saves the original contents of "m-1" in a special temporary register. It then allocates a block whose size is determined by the value of "cl" and puts a pointer to the first word of the block in "m-1". Finally, it places the original value of "m-1" in the position specified by "m-2" after "m-1" has been changed. Thus, it is possible to put the original value of "m-1" in the block which has just been allocated.

Examples:

(P GT 3) allocates a three word block and puts a pointer to the first word of the block in the bug P.

(QX GT MN) allocates a block whose size is given by the contents of the bug-field string MN, and puts a pointer to the block in the field specified by QX.

(P GT 2 PN) allocates a two word block, puts a pointer to the first word of the block in the bug P, and puts the original value of the bug P in the N field of the new block (as specified by PN).

RELEASING A BLOCK:

Tuple Format:

```
(m FR)
(m-1 FR m-2)
```

Description:

The first form of FR releases the block pointed to by "m". "m" must point to the first word of an allocated block or an error will occur. Although the contents of "m" are not modified by this tuple, the program must not reference the block after it has been freed.

The second form of FR does several things. It first saves the contents of "m-2" in a temporary register. It then releases the block pointed to by "m-1" (as in the simple FR tuple). Finally, it places the original contents of "m-2" in "m-1". Note that since the contents of "m-2" are obtained before the block is freed, this makes it possible to save a field from the released block and place it in "m-1". This extended form of the FR tuple, along with the extended form of the GT tuple make it convenient to build a linked pushdown list out of arbitrary size blocks.

Examples:

(P FR) frees the block pointed to by P.

(P FR PN) frees the block pointed to by P, but places the N field of the freed block in P.

12.2 ASSIGNMENT OPERATIONS

ASSIGNMENT:

Tuple Format:

```
(m ≤ cl)
(m = cl)
(m E cl)
```

Description:

All three of these tuples transfer the value of "cl" to "m". If the field "cl" is larger than "m" then only the rightmost bits (as many as will fit in "m") are transfered. If the field "cl" is smaller than "m" then "cl" is extended on the left with zeroes and then transfered.

INTERCHANGE OF CONTENTS:

Tuple Format:

```
(m-1 IC m-2)
```

Description:

This tuple causes the contents of "m-1" and "m-2" to be interchanged. If the two are fields of unequal size then the larger is truncated on the left as it is moved into the smaller, and the smaller is extended on the left with zeroes.

12.3 ARITHMETIC OPERATIONS

General note: In all arithmetic operations, the arguments to be combined are first transfered into 36 bit registers in right-justified positions (i.e., filled out on the left with zeroes if they are shorter than 36 bits). The two registers are combined as specified by the operation code, and the result is transfered into the modified argument (the first argument). If the first argument is shorter than 36 bits, only the rightmost bits of the result are transfered.

ADDITION:

Tuple Format:

```
(m + cl)
(m A cl)
```

Description:
The contents of "m" and the value of "cl" are added together and the result is stored in location "m".

SUBTRACTION:

Tuple Format:

```
(m - cl)
(m S cl)
```

Description:
The value of "cl" is subtracted from the contents of "m" and the result is stored in location "m".

MULTIPLICATION:

Tuple Format:

```
(m * cl)
(m M cl)
```

Description:

The contents of "m" is multiplied by the value of "cl" and the result (low order 36 bits of the product) is stored in the location "m".

DIVISION:

Tuple Format:

```
(m-1 / cl) (m-1 / cl m-2)
(m-1 V cl) (m-1 V cl m-2)
```

Description:

The contents of "m-1" is divided by the value of "cl" and the integer quotient is stored in the location "m". If "m-2" is present, the remainder of the division is stored in it. Note that this is integer division. For example, if X contains 9, then after executing (X / 2 Y) the bug X contains the integer 4, not the floating point number 4.5 or the fraction 4 1/2 or any other random value. The bug Y contains the integer 1. Division by zero does not result in an error, however it sets "m-2" to "m-1".

MODULO:

Tuple Format:

```
(m-1 MOD c1)
```

Description:

The contents of "m" is divided by the value of "c1". The remainder after this division is stored in "m". The result of the division is not saved. Note if "c1" is zero then this tuple does nothing.

12.4 BITWISE LOGICAL OPERATIONS

General comment:
The combination of arguments of different length with logical operations is done in essentially the same manner (with the same extensions and truncations) as arithmetic operations.

OR:

Tuple Format:

```
(m ! cl)
(m O cl)
(m SMP cl)
```

Description:
The contents of "m" is "ORed" with the value of "cl" and the result is stored in "m".

AND:

Tuple Format:

```
(m & cl)
(m N cl)
(m EXT cl) ---- (EXTract "m" bits of "cl")
```

Description:
The contents of "m" is "ANDed" with the value of "cl" and the results are stored in "m".

EXCLUSIVE OR:

Tuple Format:
```
(m X cl)
(m HAD cl) ---- (Half ADd "m" and "cl")
```

Description:
The contents of "m" is "EXCLUSIVE ORed" with the value of "cl" and the result is stored in "m".

COMPLEMENT:

Tuple Format:
```
(m C cl)
```

Description:
The value of "cl" is logically complemented (i.e. One bits are made zero, zero bits are made one -- be careful to remember truncation and left extension by zero) and the result is stored in "m".

12.5 INPUT/OUTPUT OPERATIONS

OPENING AND INITIALIZING FILES:

Tuple Format:

```
("outfile" INIT "infile")
```

Description:

This operation causes the file named "outfile" to be opened for output, and the file named "infile" to be opened for input. From the time this tuple is executed until another INIT tuple is executed, all data input to the program will come from file "infile", and all output will go to the file "outfile". (Exception: extra files can be opened using the OPENO and OPENI tuples.) this tuple can be used anywhere in a program to change the input and output files. The execution of this tuple closes the previously opened input and output files.

"Infile" must be the name of a previously existing file enclosed in quotes.

"Outfile" must be the name of a file (which may or not have previously existed) enclosed in quotes. If "outfile" is a previously existing file then that file will be overwritten with the new output and the previous contents of the file will be lost. If "outfile" is a new file name, the file will be created and output will go to the new file.

SPECIAL NOTE ON THE BUFFERING OF OUTPUT

L^6 buffers its output. Thus, the OUTS tuple and the TOUT tuple do not actually transmit data directly to the output file. Instead they place their argument character string at the end of an output buffer. The contents of this buffer are transmitted to the output file whenever:

1) the output buffer becomes filled during the execution of an output tuple,

2) the FOUT or OUTF (force output) tuples are used, which cause the current buffer to be output followed by the argument string, or

3) the output file is closed.

OPENING A FILE FOR OUTPUT

Tuple format:
```
     (cl-1 OPENO "filename" cl-2 cl-3)
```

Description: This tuple will cause the PDP-1Ø file named "filename" to be opened on channel "cl-1". The channel number must be between Ø and 1Ø inclusive. This tuple associates all output to channel "cl-1" with the file specified. If "cl-1" is zero then the output channel associated with the INIT tuple will be changed to "filename". The last two arguments "cl-2" and "cl-3" are optional. They specify the number of buffers to allocate and the data mode to use, respectively. If "cl-2" is zero then the default number of buffers (usually 2) is allocated. This allows one to specify the data mode without the need to explicate the number of buffers.

Examples:

(1 OPENO ´TTY:´) open the teletype for output to channel 1.

(5 OPENO "MTTT.NEW<Ø37>") open the file for output on channel five with a protection code of Ø37.

(B OPENO ´TTY7:´) open teletype seven for output on the channel specified by B.

(7 OPENO ´TTY:´ 4 #1Ø) open the teletype for output on channel seven with four buffers and in image mode

OPENING A FILE FOR INPUT

Tuple format:
```
     (cl-1 OPENI "filename" cl-2 cl-3)
```

Description: This tuple will cause the PDP-10 file named "filename" to be opened on channel "cl-1". The channel number must be between 0 and 10 inclusive. This tuples associates all input from channel "cl-1" with the file specified. If "cl-1" is zero the input channel associated with the INIT tuple is changed to "filename". The last two arguments are optional and specify the number of buffers and the data mode, respectively. If "cl-2" is zero then the default number of buffers (usually 2) are allocated. This allows one to specify the data mode without the need to explicate the number of buffers.

Examples:

(2 OPENI 'TTY5:') open teletype five for input on channel two.

(7 OPENI "DOC:BLLTN") open the file on channel seven for input.

(B OPENI "DSKB:INT.DAT" 5 #12) open the file on the channel specified by B for input with five buffers and in binary mode.

CLOSING A FILE

Tuple format:
(cl CLOSE)

Description:

This tuple will terminate all I/O activity with the file on channel "cl". It does this by forcing the last buffer and closing the file. The channel number must be between Ø and 1Ø inclusive. If the channel number is zero then both of the INIT files are closed.

Examples:

(4 CLOSE)	close channel four
(BFF CLOSE)	close the channel specified by BFF
(Ø CLOSE)	close the INIT channels

CHARACTER STRING OUTPUT:

Tuple Format:
(cl-1 OUTS cl-2 cl-3)

Description:
This tuple buffers several characters for output (see above). The number of characters is specified by the value of "cl-2". (This number must be either 1, 2, 3, 4, or 5.) The characters are specified by the value of "cl-1", and it is the rightmost "cl-2" characters that are transmitted; in a left to right sequence. The third argument "cl-3" is optional. If it is provided it denotes which channel the characters should be output to. If it is omitted the characters go to the INIT tuple's output file. Specifying channel zero as the output channel number is equivalent to omitting "cl-3".

Examples:

("ABC" OUTS 3) will buffer for output the three characters A, B and C, in that order.

("ABC" OUTS 2) will buffer for output the two characters A and B, in that order.

(BC OUTS 2) will buffer for output the rightmost two characters (14 bits) in the field specified by BC.

("XYZ" OUTS 2 1) will buffer the character X and Y for output to channel 1.

(LONG) TEXT STRING OUTPUT:

Tuple Format:
("text string" TOUT cl)

Description:
This tuple will buffer for output the entire long text string "text string". The last argument is optional. It specifies the channel which will recieve the buffering of the text.

FORCING OUTPUT OF TEXT FROM THE BUFFER

Tuple format:
(cl-1 OUTF cl-2 cl-3)

Description:

This tuple buffers several characters for output. The number of characters is specified by the value of "cl-2" ($1 \leq$ cl-2 ≤ 5). The characters are specified by "cl-1", and the rightmost "cl-2" characters are outputted, in a left to right manner. The third argument "cl-2" is the channel number to which the characters are to be output. If this number is omitted then the characters will be outputted to the INIT channel. Specifying channel zero is equivalent to omitting "cl-3".

FORCING OUTPUT OF TEXT FROM THE BUFFER:

Tuple Format:
("text string" FOUT cl)

Description:
This tuple places the long text string "text string" in the output buffer, and then forces the buffer to the device. The last argument "cl" is option. It specifies the channel number to which the text is to be outputted.

TEXT INPUT:

Tuple Format:

(m INS cl-1 cl-2)

Description:

This command reads in the number of characters specified by the value of "cl-1", and places them in the location "m", right-justified. The characters are read in from the file that is currently opened on the INIT channel. If "cl-2" is used then the input is from the file assigned to that channel. If "cl-2" is zero the input is from the INIT channel. (Note: $1 \leq$ cl-1 ≤ 5)

12.6 SUBROUTINE CALLS

The L^6 subroutine transfer strategy is described in detail in chapter five. Basically, L^6 provides a way for a tuple to transfer control to a labeled line. This type of transfer is known as a subroutine call. L^6 also provides a way of transfering control to a procedure, as described in chapter 1Ø. The "address" of the following tuple (or next line if there are no tuples after the DO tuple, or the label at the end of the clause if the calling tuple was the last tuple in a THEN or an ELSE clause) is saved. This "normal return address" is stored on the "return stack". If the label is external the local bugs and fields are also stored on the stack. Whenever the label DONE appears as a transfer label to be executed, the subroutine return pushdown stack is popped and control is passed to the "normal return address" which was just removed from the stack. If any bugs or fields were saved these are restored before control is returned. If the stack is empty the program will terminate.

Note that, in the case of subroutines, L^6 does not check to see that the user uses a DONE label to return from a statement which has been reached by a subroutine call. This is not "legal", but, it cannot be detected by L^6. The user can tell that this has happened by noticing the "level number" printed out in the dump or termination message when the program stops. That number tells how many return addresses are left on the subroutine stack (i.e., how many subroutines and procedures have been entered and not "returned from" when execution terminated). The user may look at a complete dump of the subroutine stack by doing a traceback dump.

The user can also supply an alternate return point for a subroutine, by means of the DO tuple with a FAIL exit. This tuple allows the user to specify a seperate FAIL exit to be placed on the subroutine return stack along with the "normal return address". Whenever the label FAIL appears as a transfer label the subroutine stack is popped. If the top of the stack had a FAIL exit then control is passed to the statement specified by the FAIL exit, otherwise control is returned as if a DONE label had been used. The FAIL exit must be a label local to the calling routine. It may also be *, HALT, DONE, or FAIL.

SUBROUTINE AND PROCEDURE TRANSFER:

Tuple Format:

```
(sublabel DO)
(sublabel DO failexit)
```

Description:

The first form of DO causes control to be transfered to the line with label "sublabel" unless "sublabel" appears in an EXTERNAL statement in the procedure containing the DO tuple. If "sublabel" is used in an EXTERNAL statement control is transfered to the procedure of that name. The normal return address is put on the stack, along with any local bugs and fields, as described above.

The second form of the DO tuple transfers control to the line with the label "sublabel", and establishes the line with label "failexit" as the fail exit on the stack.

12.7 <u>DEFINING FIELD TEMPLATES</u>

<u>FIELD TEMPLATE DEFINITIONS:</u>

Tuple Format:

```
    (cl-1 D f cl-2 cl-3)
    (cl-1 D f cl-2)
```

Description:

The first form of the D tuple defines the field template "f", with offset equal to the value of "cl-1" and with bit specifications given by the values of "cl-2" and "cl-3"

The second form of the D tuple defines the field template "f", with offset equal to the value of "cl-1" and with the bit specification given by the (right-justified) value of "cl-2" as a mask.

See the descriptions in chapter 2 for more detail.

12.8 PUSHDOWN STACK INSTRUCTIONS

There are two stacks in L^6 (aside from the return stack) which the user may use to store information. The user may save the definitions of field templates on the field definition stack and he may save field values on the field contents stack. Both of these stacks may actually have groups of field definitions or field values placed on them with a single save instruction. Thus it is simple to save several field definitions or values at one time and restore them later. One thing which must be remembered is that the user must remove the same number of items as he placed on the stack in one group, and in the same order. (The order of items in a group is not reversed when the group of items goes on the stack. The order of groups on the stack is the reverse of the order in which the groups were placed on the stack.)

SAVING FIELD VALUES (AND CONSTANTS):

Tuple Format:

```
(cl SFC)
(int SFC cl-1 cl-2 ... cl-int)
```

Description:

The first form of the SFC tuple is merely an abbreviated form of (1 SFC cl), so we describe the second form only. This command pushes onto the field contents stack a group of "int" values, from the items "cl-1", ..., "cl-int" (which may be constants, bug names or bug-field strings). L^6 will check that the number of arguments to the right of the SFC is greater than or equal to the value of "int" and if not it will give an error message. If there are more arguments than needed only the leftmost ones will be pushed. This allows the first argument to be a bug-field string which can specify the number of arguments to be pushed as the program is running.

Examples:

(3 SFC PX .ØAB -5) saves a group of three items, the contents of the field PX, the hexadecimal constant .ØAB and the (36 bit two's complement) integer -5.

(2 SFC X Y #11 Ø) saves only the contents of X and Y. #11 and Ø are not saved.

RESTORING VALUES SAVED ON THE FIELD CONTENTS STACK:

Tuple Format:

```
(m RFC)
(int RFC m-1 m-2 ... m-int)
```

Description:

The first form is an abbreviation for (1 RFC m), so we discuss only the second form. This tuple pops the top of the field contents stack. If the group at the top of the stack does not have at least the number of items in it specified by "int" an error occurs. Otherwise, the items in the group are stored in the locations "m-1", ... ,"m-int". Be sure to note that if the save operation was:

```
(int SFC cl-1 cl-2 ... cl-int)
```

and the restore operation is:

```
(int RFC m-1 m-2 ... m-int)
```

then the result is the same as the assignments:

```
(m-1 = cl-1) (m-2 = cl-2) ... (m-int = cl-int)
```

This statement is true both in terms of the ordering of the arguments and in terms of the truncation and extension of unequal size arguments in an assignment.

Examples:

(3 RFC P Q R) can be used to restore the bugs P, Q and R saved by (3 SFC P Q R).

SAVING FIELD DEFINITIONS:

Tuple Format:
(f SFD)
(int SFD f-1 f-2 ... F-int)

Description:
The tuple form (f SFD) is short for (1 SFD f), so we discuss only the second form. This tuple saves the field definitions associated with the field names "f-1", ..., "f-int" in a group on the field definition stack (or gives an error if too few arguments are present). It is similar to SFC.

RESTORING FIELD DEFINITIONS:

Tuple Format:
(f RFD)
(int RFD f-1 f-2 ... F-int)

Description:
This tuple is the companion tuple to SFD. Definitions. They work with SFD in the same way as it restores the fields or group of fields which were save by SFD. It works with SFD in the same way as RFC works with SFC (with respect to order of arguments).

12.9 BIT SHIFTING AND COUNTING

LEFT SHIFT:

Tuple Format:
(m L cl)

Description:
This tuple shifts the bits of the contents of "m" left by the number of positions specified by the value of "cl". Zeroes fill the vacated rightmost positions. The result is stored in the location "m".

RIGHT SHIFT:

Tuple Format:
(m R cl)

Description:
This tuple shifts the bits of the contents of "m" right by the number of positions specified by the value of "cl". Zeroes fill the vacated leftmost positions. The result is stored in the location "m".

POSITION OF LEFTMOST ONE BIT:

Tuple Format:
(m COL cl)

Description:
This tuple puts into "m" the position of the leftmost one bit of "cl". The position is given relative to the left boundary of the field, if the first bit of "cl" is one then "m" will be set to one.

Examples:
(A COL BC) If the field BC contains the bit string 2#ØØØØØØØØØØØ11Ø1ØØ1ØØ1Ø11 then after the tuple is executed bug A will be set to 12.

If the value in field BC is #241 (= 2#1Ø1ØØØØ1) then bug A is set to 1.

Note: If the value of the field BC is would have been zero (no one bits) then bug A would have been set to Ø.

POSITION OF LEFTMOST ZERO BIT:

Tuple Format:
(m CZL cl)

Description:
This tuple placed in "m" the position of the leftmost zero bit in the value of "cl". This operation is the same as COL, except that it looks for a zero instead of a one.

POSITION OF RIGHTMOST ONE BIT:

Tuple Format:
(m COR cl)

Description:
This tuple places in "m" the position of the rightmost one bit of the value of "cl", relative to the right end.

Examples:
(A COR BC) if the field BC contains the value #Ø4 (= 2#ØØØØØ1ØØ) then bug A is set to 3. If the field BC is all zeroes, then bug A is set to Ø the rightmost bit is position 1.

POSITION OF RIGHTMOST ZERO BIT:

Tuple Format:
(m CZR cl)

Description:
This tuple places in "m" the position of the rightmost zero bit of "cl", relative to the right end the rightmost bit is position 1.

COUNT NUMBER OF ONE BITS:

Tuple Format:
(m CO cl)

Description:
This tuple places in "m" the count of the number of one bits in the value of "cl".

COUNT NUMBER OF ZERO BITS:

Tuple Format:
(m CZ cl)

Description:
This tuple places in "m" the count of the number of zero bits in the value of "cl".

12.10 MISCELANEOUS TUPLES

INCREMENTAL DUMP TUPLE:

Tuple Format:
(cl-1 DUMP cl-2 cl-3)

Description:
This tuple causes a dump to be printed on the INIT file whenever the value of "cl-1" is greater than 0 and it uses that value as a spacing factor. Thus this tuple this can be used to cause a conditional dump. Such incremental dumps are of great value in debugging programs and in demonstrating the data structures that have been created at intermediate times during the execution of the program. The last two arguments are optional. They specify what is to be dumped and to which channel the dump is to go, respectively. "Cl-2" specifies what is to be dumped. This information is provided through the use of a bit mask. The following table enumerates the dumps available and the condition of the bits so that the dumps wanted will be specified.

FEATURE	BIT
BUGS	#20
CHARACTERS	#200
CORE	#40
FCS	#2
FDS	#4
FIELDS	#10
NARROW	#100
TRACEBACK	#1

Thus to dump traceback, and the bugs with characters in a wide format double spaced the dump tuple would appear as follows:

```
     (2 DUMP #221 4)
```

The last argument specifies the channel that the dump is to be outputted on. If omitted the output goes to the INIT file. If "cl-3" is zero the dump will also go to the INIT file.

To interpret the output of a dump, see Appendix B on "Reading a Dump".

INCREMENTING THE FIELD TEMPLATE:

Tuple Format:
(f IFLD)

Description:
This tuple increments the field definition. Incrementing a field causes it to mask off the next contiguous set of bits of the same size as the original template. For example, assume field H is defined to be a displacement of 1, bits Ø-6. Incrementing field H would redefine H as a displacement of 1, bits 7-11. Further increments would cause H to be redefined a 1[11,2Ø], 1[21,27], 1[28,35], 2[Ø,6]. Notice that attempting to increment a field so that it would have to mask outside a word causes the field to be set to the beginning of the next word. This tuple is very useful in setting up string and label variables.

CONDITIONAL MULTIDIRECTIONAL BRANCHING

Tuple Format:

```
(cl GOTO lab-1 lab-2 lab-3 lab-n)
```

Description:

This tuple provides a method of performing a multi-way branch in a line. The first argument of the tuple is an integer which specifies which label is to be used for the branch. If it is less than or equal to zero, or greater than the number of labels, the tuple has no effect. If however, the first argument is within range, control is transfered to the line with the label which is the cl′th left of the GOTO. If the selected label is undefined or external, control is transfered to the line which defines the last label in the GOTO statement. A error occurs if this label is external or undefined. This feature is very useful with label variables. For example, suppose a label name was read from the teletype and placed into a two word block by the program. If this was supposed to be a label in the procedure and it was misspelled then through the use of a GOTO tuple the program could regain control over a branch to an illegal label, thereby preventing the L^6 interpreter from issuing an error.

12.11 EQUALITY TESTS

EQUALITY TEST:

Tuple Format:

```
(cl-1 = cl-2)
(cl-1 E cl-2)
```

Description:

This tuple is true if the 36 bit register with the value of "cl-1" right-justified and filled with zeroes on the left is equal to the value of a second 36 bit register with "cl-2" right-justified filled with zeroes on the left.

This test is often used for fields of the same length, and in that case it is true exactly when the contents of the fields are identical.

INEQUALITY TEST:

Tuple Format:

```
(cl-1 # cl-2)
(cl-1 NE cl-2)
(cl-1 <> cl-2)
(cl-1 >< cl-2)
```

Description:

This tuple is true exactly when the tuple (cl-1 = cl-2) is false.

12.12 ALGEBRAIC COMPARISONS

GREATER THAN:

Tuple Format:

```
(cl-1 > cl-2)
(cl-1 G cl-2)
```

Description:

This test is true if the 36 bit register with the value of "cl-1" right-justified and filled on the left with zeroes is greater than the 36 bit register with the value of "cl-2" right-justified and filled on the left with zeroes. The comparison is done as 36 bit two´s complement numbers (with correct sign comparison).

LESS THAN OR EQUAL TO:

Tuple Format:

```
(cl-1 =< cl-2)
(cl-1 <= cl-2)
(cl-1 LE cl-2)
```

Description:

This test is true exactly when the test (cl-1 > cl-2) is false.

LESS THAN:

Tuple Format:

```
(cl-1 < cl-2)
(cl-1 L cl-2)
```

Description:
This test is true exactly when the test (cl-2 > cl-1) is true.

GREATER THAN OR EQUAL TO:

Tuple Format:

```
(cl-1 => cl-2)
(cl-1 >= cl-2)
(cl-1 GE cl-2)
```

Description:
This test is true exactly when the test (cl-2 > cl-1) is false.

INCLUSIVE RANGE TEST:

Tuple Format:

```
(cl-1 R cl-2 cl-3)
```

Description:
This test is true exactly when both (cl-2 <= cl-1) and (cl-1 <= cl-3) are true.

12.13 LOGICAL TESTS

SUBSET OF ONE BITS:

Tuple Format:
(cl-1 O cl-2)

Description:
This test is true exactly when all one bits in the value of "cl-1" have corresponding one bits in the value of "cl-2". (It is possible for this test to be true when the value of "cl-2" has more one bits than the value of "cl-1".)

Examples:
(5 O 7) is true.
(7 O 5) is false.

SUBSET OF ZERO BITS:

Tuple Format:
(cl-1 Z cl-2)

Description:
This test is true exactly when all zero bits in the value of "cl-1" have corresponding zero bits in the value of "cl-2". (It is possible for this test to be true when the value of "cl-2" has more zero bits than the value of "cl-1".)

Examples:
(5 Z 4) is true.
(4 Z 5) is false.

12.14 CHECKING FOR END OF FILE

CHECKING FOR THE END OF THE FILE:

Tuple Format:
    (cl EOF)

Description:
    This tuple checks to see if an end of file condition exists on the file on the channel specified by "cl". If end of file exists this tuple is true. If not, it is false. If "cl" is zero then the input side of the INIT tuple is checked.

APPENDIX A
SAMPLE CARD SETUP

This is a sample card setup for a typical L^6 program. It provides a maximum time which should be sufficient for small jobs.

```
$JOB name [p,p] /TIME:1:ØØ
$PASSWORD password
$DECK PROG.L6
     L6 procedures
$EOD
$DECK PROG.DAT
     data
$EOD
.R L6
:LOAD PROG<PROG/LOCK
:RUN procedurename
:DUMP/FILE:LPT:/SPACING:2
:EXIT
.DELETE PROG.L6,PROG.DAT

%ERR:
.REENTER
:DUMP/WIDE/FILE:LPT:
:EXIT
.DELETE PROG.L6,PROG.DAT
```

This setup creates two disk files PROG.L6 and PROG.DAT. The INIT tuple will be:

    THEN ("LPT:" INIT "PROG.DAT")

Of course any other filenames may be substituted for these two. The "name" in the job card is the name to be given to the job. It can be up to six characters in length. The "procedurename" is the name of the L^6 procedure which is to be executed. This is given on a procedure statement in the L^6 program.

APPENDIX B
READING A DUMP

L^6 has two ways to produce a dump, the DUMP tuple and the DUMP command. These are described in chapters twelve and nine respectively. The dump is divided into several parts. Each part is described below.

I/O DUMP

This dump lists all files which are currently open for program I/O. If the file status is nonzero it is listed below the file.

BUG DUMP

This dump lists the contents of all nonzero bugs. These are listed in octal. If requested, the bugs will also be listed in ASCII. The ASCII is listed in two columns. The first gives the five characters left justified in the bug (the five characters with their rightmost bits in positions 6, 13, 2Ø, 27, and 34). The second gives the five characters right justified in the bug (the five characters with the rightmost bits in positions (7, 14, 21, 28, and 35). For example, suppose bug A contains the string "RIGHT" right justified and bug B contains the string "LEFT " (with a trailing blank) left justified. The first line of the dump would look like:

```
A=245114362124   B=4621306521ØØ   *)$CD*LEFT *   *RIGHT...(@*
```

Sometimes the data was not intended to represent characters and therefore the dump will be meaningless. The ASCII character set contains many characters which are do not print, but may have other effects on the output device. Therefore these characters (#Ø-#37 and #175-#177) are replaced by a period (.) .

FIELD DUMP

All of the defined fields are printed by this dump. Examples of field definitions are:

A= 1[0,8]	777000000000	Field A has a displacement of 1 and includes bits 0 through 8. The 777000000000 is an octal mask. If converted to binary it will have ones in the bits of A and zeroes elsewhere.
4= -2[1,19]	377777600000	Field 4 has a displacement of -2 and contains bits 1 through 19.

CORE DUMP

This dump shows the user-allocated storage area. It has a format similar to the bug dump. In the dump will be one or more blocks. Between each block is a word containing information about the block. Both the right halfword (bits 18-35) of the word above and the left halfword (bits 0-17) of the word below the block contain the same information. The leftmost bit is zero if the block is free and one if the block is allocated. The rest of the bits give the length of the block plus one. For example, the following dump has three allocated blocks, a four word block at location #3, a five word block at location #23, and a ten word block at location #31. There is a free block of 10 words at location #20. Notice that groups of zero words are not included in the dump. An extra line is skipped when a group of words is omitted.

USER-ALLOCATED STORAGE AREA

```
000000  000000000002  000000000234    *.........N*  *..........*
000002  400000400005  405010647652    *@....A FOU*  *......A..**
000004  511012747644  421010246236    *R WORD BLO*  *$A/.$.A...*
000006  416260000000  400005000013    *CK...@.(..*  *.......P..*

000020  000000000000  000234000234    *.......`.N*  *.......@..*
000022  000013400006  405010644655    *..\..A FIV*  *..8...A..-*
000024  425012747644  421010246236    *E WORD BLO*  *.A/.$.A...*
000026  416260000000  000000000000    *CK........*  *..........*
000030  400006400013  405012442634    *@.4..A TEN*  *..h...A)..*
000032  202571751210  202051447606    * WORD BLOC*  *A/.%.A....*
000034  454000000000  000000000000    *K.........*  *..........*
000036  000000000000  000000000000    *..........*  *..........*
000040  000000000000  000000000000    *..........*  *..........*
000042  000000000000  400013000172    *.....@.X.=*  *........0.z*

000234  000021000021  000172400000    *.......T..*  *.......(..*
```

Blocks are zeroed when they are freed. This means that all free blocks should be zero with the exception of the last word in the block. Any data other than this word in a free block indicates that the program is using it. Also notice that a free block's last word is used by the allocator. If the program overwrites this word it may be impossible for the allocator to allocate a block when the program requests one. It will then print the message:

? POINTER DOES NOT POINT TO A BLOCK OR ALLOCATOR DATA DESTROYED

and stop the program. For these reasons the user should make sure that the program never accesses a block once it has been freed.

TRACEBACK DUMP

This dump prints a summary of all subroutine and procedure calls which have been made, but not terminated by DONE or FAIL. An example of an entry in this dump is:

```
        2: PROCEDURE YAWN CALLED BY
                  PROCEDURE BORED AT LINE 3 THEN TUPLE 2

LOCAL BUGS
        A=000000000010

LOCAL FIELDS
         3=  5[0,10]             6=    *** UNDEFINED ***

         1: PROCEDURE BORED CALLED BY
                 PROCEDURE LECTURER AT LINE 100 ELSE TUPLE 5
```

The local bugs and local fields listed are those values which were saved on the stack, i. e. the values which will be restored when the procedure is exited. The number preceding the colon is the level number. This is the number of nested procedure and subroutine calls which were in effect when this call was made.

FIELD CONTENTS STACK

The field contents stack is listed in blocks. Each block contains the values saved by one SFC tuple. The blocks are dumped in the order in which they will be restored. Therefore the values which will be restored by the next RFC tuple are dumped first.

FIELD DEFINITION STACK

The field definition stack is output in blocks similar to the field contents stack.

APPENDIX C
STORING THE LOW SEGMENT AS A CORE IMAGE

Sometimes it is convenient to store L^6 as a core image rather than reloading the program each time it is run. However, since the high segment is sharable it is not necessary to store a copy of it each time the low segment is saved. The STORE command causes the high segment to be removed and control to be returned to the monitor. The low segment may then be saved with the SAVE or SSAVE commands. For a description of these see DECsystem-10 Operating System Commands in DEC's software notebooks. Later when execution of the saved system is started, by either the ^R, RUN, or START commands, L^6 checks to see if a high segment is present. If not, it gets a copy from the system device and then starts it executing. Note, if L^6 is started and a high segment from the user's disk area is present, it will be used rather than a copy from the system device.

APPENDIX D
THE ASCII CHARACTER SET

OCTAL	ASCII	TEXT	OCTAL	ASCII	TEXT
ØØØ	null		Ø4Ø	space	
ØØ1	↑A		Ø41	!	!!
ØØ2	↑B		Ø42	"	
ØØ3	↑C		Ø43	#	
ØØ4	↑D		Ø44	$	
ØØ5	↑E		Ø45	%	
ØØ6	↑F		Ø46	&	
ØØ7	↑G(bell)	!B	Ø47	'	
Ø1Ø	↑H(backspace)		Ø5Ø	(	
Ø11	↑I(tab)	!T	Ø51	)	
Ø12	↑J(linefeed)	!L	Ø52	*	
Ø13	↑K(vert. tab)	!V	Ø53	+	
Ø14	↑L(formfeed)	!F	Ø54	,	
Ø15	↑M(car. ret.)	!C	Ø55	-	
Ø16	↑N		Ø56	.	
Ø17	↑O		Ø57	/	
Ø2Ø	↑P		Ø6Ø	Ø	
Ø21	↑Q(xon)		Ø61	1	
Ø22	↑R		Ø62	2	
Ø23	↑S(xoff)		Ø63	3	
Ø24	↑T		Ø64	4	
Ø25	↑U		Ø65	5	
Ø26	↑V		Ø66	6	
Ø27	↑W		Ø67	7	
Ø3Ø	↑X	!X	Ø7Ø	8	
Ø31	↑Y	!Y	Ø71	9	
Ø32	↑Z	!Z	Ø72	:	
Ø33	altmode	!A	Ø73	;	
Ø34	↑\	!D	Ø74	<	
Ø35	↑]	!U	Ø75	=	
Ø36	↑↑	!S	Ø76	>	
Ø37	↑←	!E	Ø77	?	

THE ASCII CHARACTER SET

OCTAL	ASCII	TEXT	OCTAL	ASCII	TEXT
1ØØ	@		14Ø	`	
1Ø1	A		141	a	
1Ø2	B		142	b	
1Ø3	C		143	c	
1Ø4	D		144	d	
1Ø5	E		145	e	
1Ø6	F		146	f	
1Ø7	G		147	g	
11Ø	H		15Ø	h	
111	I		151	i	
112	J		152	j	
113	K		153	k	
114	L		154	l	
115	M		155	m	
116	N		156	n	
117	O		157	o	
12Ø	P		161	p	
121	Q		162	q	
122	R		162	r	
123	S		163	s	
124	T		164	t	
125	U		165	u	
126	V		166	v	
127	W		167	w	
13Ø	X		17Ø	x	
131	Y		171	y	
132	Z		172	z	
133	[		173	{	
134	\		174	|	
135	]		175	}	
136	^		176	~	
137	_		177	rubout	