L6 MANUAL

Robert J. Bobrow

Technical Report #29 - 1973

1.00

CHAPTER II
THE L6 LANGUAGE

1. L6 Data Structures

1.1 ELEMENTARY DATA ITEMS

1.1.1 Internal Representation of Elementary Data

Internally, there is only one type of elementary data item in L6, the field. A field is a sequence of n contiguous bits ($1 \leq n \leq 32$) 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 represent:

a) a positive integer in binary form
($1 \leq n \leq 31$)
b) a signed integer in two's complement binary form
($n = 32$)
c) a sequence of 1, 2, 3 or 4 characters in EBCDIC code
($n = 8, 16, 24$ or 32)
d) a bit string or set of flags or logical variables
($1 \leq n \leq 32$)
e) a pointer to a word in user-allocated memory
($n = 16$, and the value of the 16 bit integer plus a fixed constant (called the "bias") form the address of the indicated word.)

1.1.2 Constant Notations - External Representation of Elementary Data

There are several ways in which data can be represented in an L6 program. Since all data is internally stored in the form of 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.

II - THE L6 LANGUAGE

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

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 "0" 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 four characters enclosed in either single quotes (') or double quotes ("). Note that if an apostrophe is to be contained in a string the string should be represented with double quotes. Some examples of character strings are:

"A", 'A', "AB*C", 'Q3-A', "AB'C", "12A"

Long Text Strings for Titles

To make it easier to print long titles or headings, one of the output operations (the text output operation, TOUT) accepts character strings of any length, represented by the string in quotes. For example:

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

Arbitrary Bit Configurations - Hexadecimal Representation

A convenient way for the user to represent arbitrary bit strings is as hexadecimal numbers. A hexadecimal number is written as a period (.) followed by a sequence of from 1 to 8 characters from the character set

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

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

```
0 = 0000,  1 = 0001,  2 = 0010,  3 = 0011,
4 = 0100,  5 = 0101,  6 = 0110,  7 = 0111,
8 = 1000,  9 = 1001,  A = 1010,  B = 1011,
C = 1100,  D = 1101,  E = 1110,  F = 1111
```

Some examples of hexadecimal constants are:

```
.11 (= 00010001), .A13 (= 101000010011),
.FFF (= 111111111111)
```

Hexadecimal numbers are often used to represent EBCDIC characters which cannot be conveniently typed directly in a character string, such as the "carriage return" (which is .0D).

1.1.3 Internal Representation of Constants

After constants have been read into L6, the user can consider them to be 32 bit fields (although there are actually several internal representations for constants in the L6 interpreted code). Negative integers are 32 bits in two's complement representation, positive integers are 32 bits, with the high order (leftmost) bits being 0, including the sign bit. Character strings are represented as right justified bit strings, with the leftmost 0, 8, 16 or 24 bits (for 4, 3, 2 or 1 character strings) being 0. Hexadecimal numbers are right justified with the leftmost bits 0.

Remember, internally all data is stored in a field, and the original notation for the constant that was used to create the data does not restrict the way in which the data in the field can be used. Even if a constant was input as a hexadecimal number it can still be used as text for output, in arithmetic operations, as a pointer, as well as being used for a bit string.

1.2 COMPOSITE DATA STRUCTURES IN L6 - BLOCKS AND PLEX STRUCTURES

L6 allows the user to build arbitrarily complicated data structures. The basic component of such structures is the block, which is a set of n sequential words in a special area of memory, the user allocated storage area. As far as the L6 system is concerned, a block has no internal structure, and can be used to hold any data that the user desires to store in n words (n may be any size that can fit in the available storage).

The L6 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.)

L6 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 EBCDIC 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 16 bits long, and holds the relative address of a word in the user allocated storage area. The actual address of the word pointed to may be obtained by adding a constant (the bias) to the pointer, giving a genuine machine address. 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, numbers, etc., to represent graphs, circular lists, stacks, queues, etc.

2. Storing and Referring to Data Items in L6

There are essentially three different areas where the user may store data in an L6 program. As indicated above, the user may store data in fields within blocks obtained from the storage allocator. There is also a pushdown stack, the field-contents stack, which may be used to save data, especially during the execution of recursive subroutines.

However, the most important data storage areas in L6 are a special set of 26 registers, called bugs, each of which can contain an arbitrary 32 bit data item. Although these bugs can be (and are) used to hold numbers for arithmetic, and character strings for input and output, their most important use is to hold pointers which provide the only way for the user to refer to the data stored in allocated blocks.

2.1 BASE DATA ITEMS - BUGS

The basic data storage areas in L6 are 26 registers, called bugs. These bugs are referred 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.

As mentioned above, each of these registers can contain any 32 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 referring to all data in the user allocated storage area.

To refer to the 32 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.

2.2 REFERRING 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 a 16 bit quantity which gives the relative address of a word in the user allocated storage area. (The actual address is obtained by adding a quantity called the bias to the pointer.) A field template indicates the position of a field relative to a pointer.

2.2.1 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, C,..., 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.

To specify the position of the field within a word the user can either give a 32 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 referred to as bit 0, and the rightmost is bit 31. (The high-order bit is bit 0, the low-order bit is bit 31.)

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 .0F800000 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:

```
(0 D Q .7F0)
```

which defines the field template named Q to have offset 0 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 .01FC0000) and (0 D Q 21 27).
```

2.2.2 Referring 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 31 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 (currrently defined) fields. All but the last field must be at least 16 bits long, since they must contain pointers.

Some examples are:

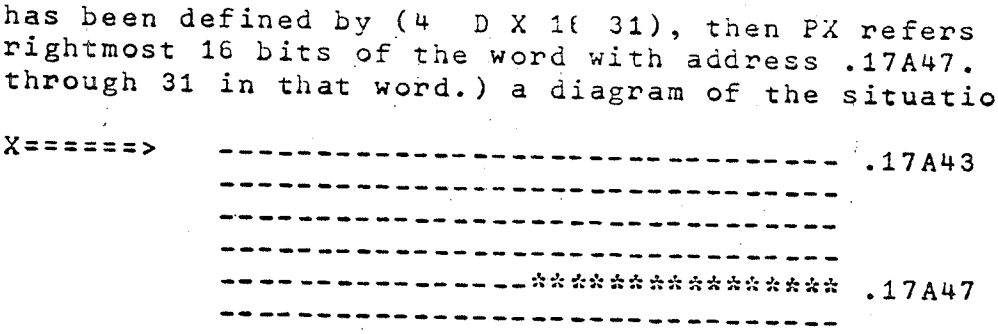
```
AX, QZ1, R13, M1RST1
```

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 .17A43 and X has been defined by (4 D X 16 31), then PX refers to the field in the

has been defined by (4 D X 16 31), then PX refers to the field in the rightmost 16 bits of the word with address .17A47. (That is, bits 16 through 31 in that word.) a diagram of the situation is:



Since the field specified by PX has 16 bits, it could contain a pointer to a word within another allocated block. In that case the field PXX would refer to the X field relative to the pointer in PX. Thus, if bits 16 to 31 of word .17A47 contain a pointer to word .17B30 (remember, the actual contents of the field PX would not contain .17B30, but would contain .17L30 minus the bias) then PXX would be the field in bits 16 through 31 of word .17B34. If field 5 were defined by (1 D 5 0 7) then PX5 would refer to bits 0 through 7 (the leftmost 8 bits) of word .17B31.

In general, if b is a lug and f, g, ...,w are currently defined fields, each of at least 16 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 (.

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 left field in the second word of the third block) can be referred to as:

XA or WKA or WFFA or WF3KA

3. Elementary Actions on Data - the TUPLE

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 (seeing if they are equal, which is greater, etc. ...). An operation generally specifies some change to be made to the contents of some field, bug, push-down stack or i/o device. In L6, both tests and operations are written in the same general format, with the distinction between the two types of action being made on the basis of where they occur in the L6 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.

In general, 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 separated by at least one blank space.

The general formats are:

(data1 op) or (data1 op data2) or (data1 op data2 data3) or (data1 op data2 ... 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 (it compares the contents of the field XA1 with the integer 15 when it is used as a test tuple)

(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 10) requests a block of size 10 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 L6:

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 - A for and, O for or, X for exclusive or, C for complement

Input/output - INIT for initializing input and output files, INS for inputting character (strings), OUTS 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 transferring 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 leftmost one position, CZL for finding leftmost zero position, COR for finding rightmost one position, CZR for finding rightmost zero position, CO for counting number of ones, CZ for counting number of zeroes

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

3.2.2 Tests

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 SOME GENERAL COMMENTS ON OPERATIONS AND TUPLE FORMATS IN L6

As should be clear from the examples, the operation code is always the second element of the tuple. Since the operation codes are 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. There can be no spaces between the characters in a bug/field string.

In L6, 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 and the RFC restore field contents operation.) Thus, in general, the first argument of a 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. In general, the other arguments of a tuple can either be constants, bugs or bug/field strings.

3.4 OPERATIONS ON FIELDS OF UNEQUAL LENGTH

It is common to combine fields of unequal length in L6 operations. Conceptually, the operations are performed on the two fields as if they were right justified in a 32 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 an accumulator, with the remaining bits zeroed, and puts the 13 bits of PQR in another accumulator, 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. 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 32 bit field. Thus, if a field can contain enough bits to represent the constant the result is what you would expect (remember that negative numbers require a full 32 bit field).

3.4 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 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 32 bit words. Note, that as in all L6 data transfers, if the fields are shorter than 32 bits, the quantity stored in the stack contains the field to be stored, right justified with zeroes on the left in a 32 bit word. The user should be careful, since the stack holds blocks of words. To save the registers A, Q and the field PRX the user can type

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

and to restore the same items he would type,

```
(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.

4. The L6 Program

4.1 GENERAL FORMAT OF A PROGRAM

An L6 program 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 the sequence, and from left to right, tuple to tuple, within the 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

```
/THIS L6 PROGRAM READS IN A SEQUENCE OF CHARACTERS 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 0 IN LINK FIELD.
/
/THERE ARE TWO FIELDS IN EACH BLOCK, THE FIELD C IS AN 8 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 ($ INIT $)(0 D C 0 7)(0 D N 16 31)  /SETUP FLD TEMPLATES ≥ IO
      THEN (P GT 1)(B = P)(PC INS 1)            /SETUP FIRST BLOCK
      THEN (C INS 1)                            /SETUP CHARACTER 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 (P = B)
/                                               /SETUP  P FOR OUTPUT LOOP
/THIS LOOP TRAVERSES THE LIST, PRINTING CHARACTERS AND FREEING BLOCKS
/
OUTLP IF (P = 0) THEN HALT ELSE (PC OUTS 1)(T = P)(P = PN)(T FR) OUTLP
```

4.3 THE L6 STATEMENT

4.3.1 Labels, Comments and General Format

In general an L6 statement is free form. If the statement has a label, the label must begin in the first character position. A label can be any sequence of from one to six characters, with the exception of the reserved words THEN, ELSE, DONE, FAIL, HALT. If the statement has no label the first character position must be blank. In the program above all non-labelled statements started in column 8. This was done only to improve readability, since L6 would have been perfectly happy to have the statements start anywhere after column 1 if they had no label. 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 L6 program by preceding them with a slash (/). All characters on a line following a / are ignored by the parser. Currently, / can be used only in column 1 or after a statement. (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 <THEN-clause>

The simplest form of an L6 statement is the unconditional statement which consists of the key-word 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. Transferring to the label HALT will cause the program to come to a halt. DONE and FAIL are special dummy labels used to return from a subroutine. * Is a special label useful for creating one-line loops.

Once execution begins on the sequence of tuples all the tuples will be executed unless there is an error, or unless there is a subroutine transfer tuple in the sequence, and the subroutine returns with a FAIL exit (see the section on subroutines).

The unconditional statement is not only the simplest form of an L6 statement, it is the basis for all other statements. The sequence of the keyword THEN followed by either a sequence of one or more tuples, a label or both, is referred to as a <THEN-clause>.

4.3.3 Conditional Operations and Transfers
The <IF-clause> and the <ELSE-clause>

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. L6 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 # 0) 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 are shown below with multiple tests.

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 an <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 = 0) 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. L6 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 indindicates that the <THEN-clause> is to be executed only if at least one of the tests is false.

Some examples are:

```
IFALL (P # 0)(PC # ".") THEN (PC OUTS 1) ELSE (PD OUTS 1)

IFANY (P = 0)(PC ".") THEN (PD OUTS 1) ELSE (PC OUTS 1)

IFNONE (PN = 0)(R = 3)(J > K) THEN (R + J)(P = PN)

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

5. Some Useful Programming Constructs in L6

5.1 ONE LINE LOOPS

There is a special symbol * which can be used as a transfer label in either a <THEN-clause> or an <ELSE-clause>. The symbol * stands for the current line. Thus, it is possible to write convenient one-line loops in L6.

For example:

```
IF (P # 0) 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 0 link field is found.

```
IF (P = 0) 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 0, having returned the last block.

5.2 SUBROUTINES IN L6

L6 provides a minimal subroutine capability (which will be expanded in later versions). 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 FALEXT)
```

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 subroutine return pushdown stack and control is passed to the statement whose label is the first

argument of the DO tuple (this is the only 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 subroutine 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-clause> or an <ELSE-clause>, the subroutine 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-clause> or an <ELSE-clause>, the subroutine 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. Note that the FAIL exit feature is the only way to avoid executing all the operation tuples in a <THEN-clause> or an <ELSE-clause> once the first tuple is executed.

Unfortunately, in the current version of L6 there is no way of having a separate set of bugs for each subroutine, or for having statement labels local to each subroutine. In general, information is passed to a subroutine by being placed in one or more bugs, and information is returned by being placed in bugs.

Since the location of the calling tuple is placed in a stack, it is quite easy to write recursive subroutines in L6, 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.

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 L6. 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.

L6 OPERATIONS AND TESTS

The following is a collection of descriptions including examples for all operations and tests implemented in version 1.5 of L6 on the Sigma-7.

Notations Used in Tuple Descriptions

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 mocified (literal, bug name or bug/field string)
int	positive integer
f	field name

The tuple descriptions are grouped according to the categories in section 3.2.1.

OPERATION TUPLES

1. STORAGE ALLOCATION AND RELEASE

ALLOCATING A BLOCK:

Tuple Format:

(m GT c1)
(m-1 GT c1 m-2)

Description:

The first format allocates a block whose size is determined by the value of "c1" 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 "c1" 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 field MN, and puts a pointer to the block in the field 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, "m" can no longer be used as a pointer since it no longer points to a word within an allocated block.

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 push-down 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.

2. ASSIGNMENT OPERATIONS

ASSIGNMENT:

Tuple Format:

(m ← c1)
(m = c1)
(m E c1)

Description:

All three of these tuples transfer the value of "c1" to the location "m". If the field "c1" is larger than "m" then only the rightmost bits (as many as will fit in "m") are transfered. If the field "c1" is smaller than "m" then "c1" 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 locations "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.

3. ARITHMETIC OPERATIONS

General note: In all arithmetic operations, the arguments to be combined are first transfered into 32 bit registers in right-justified positions (i.e., filled out on the left with zeroes if they are shorter than 32 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 32 bits, only the rightmost bits of the result are transfered.

ADDITION:

Tuple Format:

```
(m + c1)
(m A c1)
```

Description:

The contents of "m" and the value of "c1" are added together and the result is stored in location "m".

SUBTRACTION:

Tuple Format:

```
(m - c1)
(m S c1)
```

Description:

The value of "c1" 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" are multiplied by the value of "cl" and the result (low order 32 bits of the product) is stored in the location "m".

DIVISION:

Tuple Format:

(m / cl)
(m V cl)

Description:

The contents of "m" are divided by the value of "cl" and the integer quotient is stored in the location "m". Note that this is integer division. For example, if X contains 9, then after executing (X / 2) 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.

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 arithmetical operations.

OR:

Tuple Format:
(m O cl)
(m SMP cl)

Description:
The contents of "m" are "OR-ed" with the value of "cl" and the results are stored in the location "m".

AND:

Tuple Format:
(m N cl)
(m EXT cl) ---- (for EXTract the bits of "cl")

Description:
The contents of "m" are "AND-ed" with the value of "cl" and the results are stored in the location "m".

EXCLUSIVE OR:

Tuple Format:
(m X cl)
(m HAD cl) ---- (for Half-ADding "m" and "cl")

Description:

The contents of "m" are "EXCLUSIVE OR-ed" with the value of "c1" and the results are stored in the location "m'.

COMPLEMENT:

Tuple Format:

(m C c1)

Description:

The value of "c1" 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 the location "m".

5. INPUT/OUTPUT OPERATIONS

OPENING AND INITIALIZING FILES:

Tuple Format:

(outfile INIT infile)

Description:

This operation causes the Sigma-7 file named "outfile" to be opened for output, and the Sigma-7 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 file "outfile". 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 either be the name of a previously existing Sigma-7 file enclosed in quotes, or the symbol $. The symbol $ is interpreted to be the user's terminal console if the user is in interactive mode and the lineprinter if the user is in batch mode. (Obviously the symbol $ should not be used for the input file in batch mode. Lineprinter input?!?!)

"outfile" must either be the name of a Sigma-7 file (which may not have previously existed) enclosed in quotes, or the symbol $. 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. The symbol $ stands for the user's terminal console in interactive mode, and for the lineprinter in batch mode. (Thus $ is a perfectly reasonable output file in batch mode. Output previously sent to the lineprinter will not be destroyed by using $ as the output file.)

SPECIAL NOTE ON LINE TERMINATIONS IN FILES

The user deserves a word of caution on the use of carriage returns as line termination characters on the Sigma-7. There is some confusion in the UTS system about line termination characters. To wit:

1) there is (in general) no carriage return at the end of lines in files produced by reading a deck of cards from the card reader.
2) There are two commonly used characters for representing carriage returns which the user may "see" on input, (hexadecimal codes .0D and .15) but only one code (hexadecimal .0D) which will be recognized as a carriage return by the L6 system on output.

Thus, it is highly recommended that the user utilize some character other than a carriage return to denote the end of a line of input (e.g., a period, "." or a comma ","). It is also useful to write a standard input routine that skips over and discards all carriage return characters on input (it is necessary to check for both hexadecimal .0D and hexadecimal .15, even on input from the user's terminal).

SPECIAL NOTE ON THE BUFFERING OF OUTPUT

L6 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 either whenever:

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

2) a carriage return (hexadecimal .0D) is encountered in the output string, or

3) the FOUT (force output) tuple is used, which acts as if the argument string is terminated by a carriage return

CHARACTER STRING OUTPUT:

Tuple Format:

(c1-1 OUTS c1-2)

Description:

This tuple buffers several characters for output (see above). The number of characters is specified by the value of "c1-2". (This number must be either 1, 2, 3 or 4.) The characters are specified by the value of "c1-1", and it is the rightmost "c1-2" characters that are transmitted, in a left to right sequence.

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.

(.0D OUTS 1) will force out the current contents of the output buffer, followed by a carriage return (hexadecimal .0D).

(BC OUTS 2) will buffer for output the rightmost two characters (16 bits) in the field specified by BC.

(LONG) TEXT STRING OUTPUT:

Tuple Format:

("text string" TOUT)

Description:

This tuple will buffer for output the entire long text string "text string".

FORCING OUTPUT OF TEXT FROM THE BUFFER:

Tuple Format:

("text string" FOUT)

Description:

This tuple places the long text string "text string" in the output buffer, followed by a carriage return (hexadecimal .0D). This effectively forces out the entire contents of the buffer, then the long text string, and finally a carriage return.

TEXT INPUT:

Tuple Format:

(m INS c1)

Description:

This command reads in several characters, the number specified by the value of "c1", and places them in the location "m", right-justified. The characters are read in from the currently open data input file, as determined by the most recently executed INIT tuple. (Note: $1 \leq c1 \leq 4$)

6. SUBROUTINE CALLS

The L6 subroutine transfer strategy is described in detail in section II.5.2 (page II - 18). Basically, L6 provides a way for a tuple to transfer control to a labelled line, and to save the "address" of the following tuple (or next line if there are no tuples after the subroutine call, or the label at the end of the clause if the subroutine call tuple was the last tuple in a <THEN-clause> or an <ELSE-clause with a label). This "normal return address" is stored on the "subroutine return pushdown-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 popped off the stack. (If the subroutine stack is empty, an error message will result.) Note that L6 does not check that the user uses the 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 L6. 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 have been entered and not "returned from" when execution terminated).

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 separate 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 an error occurs.

SUBROUTINE 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 (an error occurs if no such line exists), with normal return address put on the subroutine stack 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 subroutine stack.

7. DEFINING FIELD TEMPLATES

FIELD TEMPLATE DEFINITIONS:

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 with name "f", with offset equal to the value of "cl-1" (which may be computed) and with bit specifications given by the values of "cl-2" and "cl-3" (which may also be computed).

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

See the descriptions in section II.2.2.1 (pages II - 6 and II - 7) for more detail.

8. PUSHDOWN STACK INSTRUCTIONS

There are two stacks in L6 (aside from the subroutine return stack) on which the user may 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 actually place groups of field definitions (field values) on the stack with a single push instruction. Thus it is simple to save several field definitions (field values) at one time and restore them later. The only confusing part is that the user must pop off the same number of items as he pushed on 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 size "int") of values, from the items "cl-1", ... , "Cl-int" (which may be constants, bug names or bug/field strings). L6 will check that the number of arguments to the right of the SFC is the same as the value of "int" and if not it will give an error message.

Examples:

(3 SFC PX .(AB -5) saves a group of three items, the contents of the field PX, the hexadecimal constant .0AB and the (32 bit two's complement) integer -5.

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 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 c1-1 c1-2 ... c1-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 = c1-1) (m-2 = c1-2) ... (m-int = c-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 registers 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. That 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 the wrong number of arguments is given). The operation works very similarly to SFC.

RESTORING FIELD DEFINITIONS:

Tuple Format:

(f RFD)

(int RFD f-1 f-2 ... f-int)

Description:

These tuples are the companion tuples for restoring field definitions. They work with SFD in the same way as RFC works with SFC (with respect to order of arguments).

9. BIT SHIFTING AND COUNTING

LEFT SHIFT:

Tuple Format:
(m L c1)

Description:
This tuple shifts the bits of the contents of "m" left by the number of positions specified by the value of "c1". Zeroes fill the vacated rightmost positions. The result is stored in the location "m".

RIGHT SHIFT:

Tuple Format:
(m R c1)

Description:
This tuple shifts the bits of the contents of "m" right by the number of positions specified by the value of "c1". Zeroes fill the vacated leftmost positions. The result is stored in the location "m".

POSITION OF LEFTMOST ONE BIT:

Tuple Format:
(m COL c1)

Description:
This tuple fills the location "m" with the position of the leftmost one bit in the value of "c1". The position is given relative to the left boundary of the field, with a value with a one in the leftmost

position resulting in "m" being set to 1.

Examples:

(A COL BC) If the field BC contains the bit string coded by .01A4B (= 00000001101001001011) then after the tuple the contents of bug A will be the integer 8.

If the value in field BC is coded by .B1 (= 10110001) then the bug A is set to 1.

If the value of the field "c1" is all zeroes (no one bits) then the bug A is set to 0.

POSITION OF LEFTMOST ZERO BIT:

Tuple Format:

(m CZL c1)

Description:

This tuple fills the location "m" with the position of the leftmost zero bit in the value of "c1". This operation is essentially the same as COL, except that it looks for zeroes instead of ones.

POSITION OF RIGHTMOST ONE BIT:

Tuple Format:

(m COR c1)

Description:

This tuple fills the location "m" with the position of the rightmost one bit of the value of "c1", relative to the right end.

Examples:

(A COR BC) if the field BC contains the value .04 (= 00000100) then the bug A is set to 3. If the field BC is all zeroes, then the bug A is set to 0. End (with the rightmost bit being called position 1).

POSITION OF RIGHTMOST ZERO BIT:

Tuple Format:
(m CZR c1)

Description:
This tuple fills the location "m" with the position of the rightmost zero bit of the value of "c1", relative to the right end (with the rightmost bit being called position 1).

COUNT NUMBER OF ONE BITS:

Tuple Format:
(m CO c1)

Description:
This tuple fills the location "m" with the count of the number of one bits in the value of "c1".

COUNT NUMBER OF ZERO BITS:

Tuple Format:
(m CZ c1)

Description:
This tuple fills the location "m" with the count of the number of zero bits in the value of "c1".

INCREMENTAL DUMP OF THE USER ALLOCATED STORAGE AREA

INCREMENTAL DUMP TUPLE:

Tuple Format:

(c1 DUMP)

Description:

This tuple causes a dump to be printed on the listing device whenever the value of "c1" is greater than 0. 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.

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

11. EQUALITY TESTS

EQUALITY TEST:

Tuple Format:

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

Description:

This tuple is true if the 32 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 32 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. ALGEBRAIC COMPARISONS

GREATER THAN:

Tuple Format:

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

Description:

This test is true if the 32 bit register with the value of "cl-1" right-justified and filled on the left with zeroes is greater than the 32 bit register with the value of "cl-2" right-justified and filled on the left with zeroes. The comparison is done as 32 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 the tests (cl-2 <= cl-1) and (cl-1 <= cl-3) are true.

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