NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

*1 MANUAL

by

Allen Newell

Jay Earley

Fred Haney

Carnegie Institute of Technology
Pittsburgh, Pennsylvania
June 26, 1967

This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (SD-146) and is monitored by the Air Force Office of Scientific Research.

JUN 6 '72

HUNT LIBRARY
CARNEGIE-MELLON UNIVERSITY

*1 MANUAL

*1 is a set of system 360 macros [1] designed to perform the basic tasks that appear in many list processing and systems programming applications. It is based on L6, a list processing language designed by K. Knowlton of the Bell Laboratories [2]. According to Knowlton, the purpose of L6 was to permit users to "get much closer to machine code in order to write faster running programs, to use storage more efficiently, and to build a wider variety of data structures." The goals of *1 are approximately the same. As implemented at Carnegie-Mellon University, *1 contains most of the important facilities of L6, except the available space mechanism. This was omitted purposely because it or other available space schemes can be programmed rather easily in *1. *1 also contains a number of additions to L6: block operations, dynamic bounds on fields and blocks, the meta-language.

One objective in the design of *1 has been to give the programmer control over the code that is produced. This is accomplished by making the internal workings available to the user and by allowing him to alter any of the internal macros for his own use either directly or by using the meta-language.

*1 provides two basic data structures, the block and field, which may be compounded into arbitrarily complex high-level data structures. Data within these structures is referenced by indexing or by chains of pointers which lead to desired storage locations. The language also includes a basic set of arithmetic and logic operations, data comparisons and tests, control operations, and input/output operations.

Data Structures

A block is a set of contiguous words or bytes which are to be treated as a unit. Two types of blocks may be defined. A base block is an actual set of words or bytes in memory; it is declared by giving the address of its origin and its length. A block, on the other hand, is not a specific piece of memory, but rather a type of structure, whose origin is specified at run time when it is used. The declarations are as follows:

BLOCK		block (origin and length in words)
BBLOCK	base	block (origin and length in words)
YBLOCK		block (origin and length in bytes)
YBBLOCK	base	block (origin and length in bytes)

Note that a Y indicates that data is given in bytes rather than words. A base block declaration defines a block by specifying its name, extent, and the address of the origin [leftmost byte]. A block declaration specifies the name and extent of a block, but the origin must be specified at run time in relation to an outer block.

Examples:

```
BBLOCK B,(LOC,8)
YBLOCK C,(4,40)
```

Base block B contains 8 words with origin at location LOC. C is a block of 40 bytes with origin at the number 4 byte of an implied outer block.

The other data structure of *1 is the field. A field is a set of contiguous bits within a full word. Like a block, a field may have explicit or implicit origins. Its origin is always specified in words.

Examples: BFIELD D,PLACE

FIELD C,0,(18,31)

The base field D is a full-word at location PLACE. Field C is bits 18-31, inclusive, of the beginning word of some implied outer block. Whenever C is used, its origin must be specified by the context. All numbering of bytes, bits and words starts at 0.

Definitions*

<IDENTIFIER> ::= <LETTER>|<IDENTIFIER><LETTER>|<IDENTIFIER><DIGIT>

<QUANTITY> ::= <FIELD SEQUENCE>|$<ASSEMBLER EXPRESSION>|<ASSEMBLER EXPRESSION>

<ORIGIN> ::= <QUANTITY>

<SIZE> ::= <QUANTITY>

<BLOCK SPECIFIER> ::= BLOCK|BBLOCK|YBLOCK|YBBLOCK

<BLOCK DEFINITION> ::= <BLOCK SPECIFIER> <IDENTIFIER>,(<ORIGIN>,<SIZE>)

<NUMBER> ::= <DIGIT SEQUENCE>|<SETA SYMBOL>

<LOW BIT> ::= <NUMBER>

<HIGH BIT> ::= <NUMBER>

<FIELD DEFINITION> ::= FIELD <IDENTIFIER>,<ORIGIN>,(<LOW BIT>,<HIGH BIT>)|
BFIELD <IDENTIFIER>,<LOCATION>

Examples:

BLOCK B,(3,5)

BBLOCK DM(LOC,XYZ)

FIELD A,1,(18,31)

BFIELD B,LOC

* Within a section we first define the syntax using BNF [3], and follow it with a more informal description.

The syntactic form <ASSEMBLER EXPRESSION> represents an assembler constant, any expression which is a legal operand of a SETA instruction, or an assembler symbol,as defined by the System 360 Assembler Language Manual. Usual these will be decimal numbers or assembler names which have been assigned locations in the assembly. The $ may be used to resolve ambiguity (see Operations). If a quantity is to be used as an address and it is less than a full address then it is as if it were extended with 0's on the left.

The origin of a block or field and the size of a block may be specified either statically or dynamically. If it is specified statically by an assembler expression, this value is fixed for the assembly. If it is specified dynamically by a field sequence F (see below), the origin of the block will be the current contents of field F at the time that the block is accessed. This means that these sizes and locations can be changed at run time. Since this is implemented using index registers, it is ready-made for use in accessing tables and arrays. The origins of base blocks and base fields may not be specified dynamically.

A field consists of the low bit, the high bit, and all bits between. A base field is always one full-word. We soon expect to allow the location of a base field or base block to be a general register or registers. This will allow *1 to produce much better code and will also aid in communication between *1 and assembler language code.

<u>Sequences</u>

```
<ELEMENTARY IDENTIFIER> ::= <LETTER>|<ELEMENTARY IDENTIFIER><DIGIT>
<NAME> ::= [<IDENTIFIER>]|<ELEMENTARY IDENTIFIER>
<FIELD NAME> ::= <NAME>
```

```
<BASEFIELD NAME> ::= <NAME>

<FIELD SEQUENCE> ::= <SEQUENCE><FIELD NAME>|<BASEFIELD NAME>

<BLOCK NAME> ::= <NAME>

<BASEBLOCK NAME> ::= <NAME>

<BLOCK SEQUENCES> ::= <SEQUENCE><BLOCK NAME>|<BASEBLOCK NAME>

<SEQUENCE> ::= <FIELD SEQUENCE>|<BLOCK SEQUENCE>
```

A sequence is a chain of block and field names which leads to a block or field. The first name in the sequence and only the first must be a base field or base block name. Each field is assumed to contain a pointer to a location in memory. An implied block is assumed to exist there with that location as its 0th word.

The value of the sequence is calculated in the following way: First a block is accessed. If the sequence starts with a base field we access the implied block which that points to; if it starts with a base block we access that block. If the next name is a block name B, we then access the B sub-block of that block. If the next name is a field name F, we access the F field of the block and then access the implied block pointed to by that field. In either case we now have a block again, so the process is repeated. This continues through the sequence until the last name gives us either a block or field as a value. In the case that a field is last, we do not access its implied block.

A field sequence points to a field (<FIELD NAME>) within a block (<SEQUENCE>). A block sequence points to a sub-block (<BLOCK NAME>) of an outer block (<SEQUENCE>).

Examples:

BFIELD W, ...

BBLOCK C, (...,5)

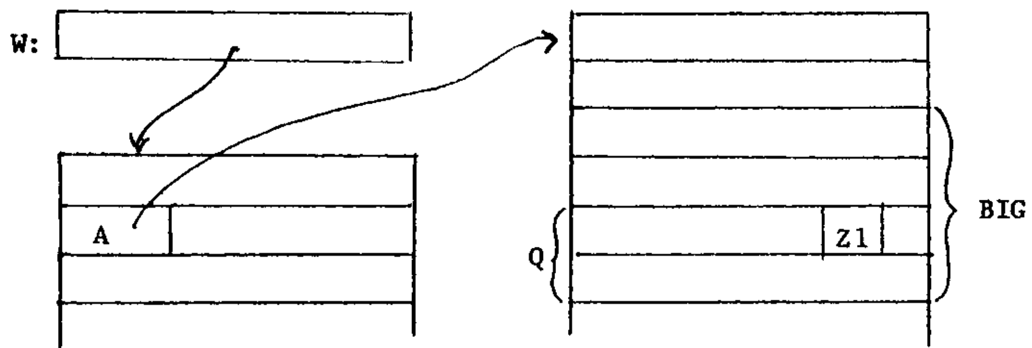
FIELD A, 1, (0,15)

FIELD Z1, O, (21,26)

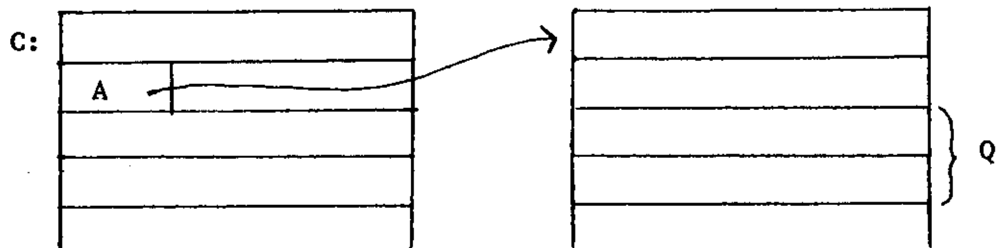
BLOCK BIG, (2,4)

BLOCK Q, (2,2)

To calculate WA[BIG]QZ1 we go through the following chain:



To calculate CAQ we do:



Note that one may concatenate single letter names (or single letters followed by digit strings) to form sequences without punctuation. Other names, however, require [ ] around them.

Operations

<SHIFT OPERATOR> ::= →→ | ←←

<SHIFT OPERATION> ::= (<SHIFT OPERATOR> <QUANTITY>,<FIELD SEQUENCE>)

<COMPLEMENT OPERATION> ::= (¬,<FIELD SEQUENCE>)

<FIELD OPERATOR> ::= +|*|-|/|MOD|∧|∨|X∨

<FIELD OPERATION> ::= (<FIELD SEQUENCE>,<FIELD OPERATOR>, <QUANTITY>)

<EITHER> ::= <BLOCK SEQUENCE>|<QUANTITY>

<STORE OPERATION> ::= (<SEQUENCE>,←,<EITHER>)

<POINTER OPERATION> ::= (<FIELD SEQUENCE>,←o,<EITHER>)

The operators have the following meanings:

OPERATOR	MEANING
←	Logical move or store
+	Add
-	Subtract
*	Multiply
/	Divide
MOD	Remainder
∧	Logical and
∨	Logical or
X∨	Logical exclusive or
¬	Logical complement
→→	Shift right
←←	Shift left
←o	Store pointer

When a field operation is performed the field sequence and quantity are combined according to the field operator and the result replaces the contents of the field specified by the field sequence. The arithmetic operations will work correctly for non-negative integers and for negative integers which are kept in full word fields.

The store operation will store a constant, or the contents of the field or block on the right, into the field or block on the left. The information is stored right justified in a field and right justified beginning at the last word or byte of a block. The rest of the field or block is cleared. If a block or quantity is stored into a block or field which is too small, it may clobber contiguous information. The convention of not making a check for any kind of error holds throughout the language, so that a minimal amount of code will be produced. Any user who wants these checks can insert them using the meta-language.

In a pointer operation it is assumed that the right hand operand points to a location in memory. It causes the left field to point to that location. When the right operand is a field sequence, the pointer operation is the same as a store. When the right operand is a block sequence, it causes the left field to point to the beginning of the block.

The MOD operator produces the remainder upon division of the field sequence by the quantity.

Shift operations cause the contents of the field specified by the field sequence to be shifted the number of bits specified by the quantity. 0's are supplied when bits are vacated by the shift, but any non-zero bits which are

made to overlap other fields within the same word may cause trouble can be checked using the meta-language.

Examples:

(ABF,+,4)	adds 4 to ABF.
(ABF,V,AZF)	unites the contents of AZF into ABF.
(¬,ABCFG)	complements ABCFG.
(→ → 3,AB)	shifts the contents of AB 3 bits to the right.

Note: If the sequence of characters that specifies a quantity has a valid interpretation as a sequence, then it is treated as a sequence. Assembler expressions which might be misconstrued as sequences must be preceded by $. Thus if NAME is an assembler label, and N is a base block or base field and A, M, and E are blocks or fields then

(W,←,NAME) treats NAME as a sequence

while (W,←,$NAME) treats it as an assembler label.

Input/Output and Data Conversion

<I/O DECLARATION> ::= INPUT<BASEBLOCK NAME>|OUTPUT<BASEBLOCK NAME>

<I/O OPERATION> ::= IN<BASEBLOCK NAME>|OUT<BASEBLOCK NAME>

<CONVERSION OP> ::= B←D|D←B

<CONVERSION OPERATION> ::= (<FIELD SEQUENCE>,<CONVERSION OP>,<FIELD SEQUENCE>)

Any base block may be declared as an input area, an output area, or both. Its declaration must occur before it is used in an I/O operation. The IN operation causes a card to be read and stored in the specified base block. The OUT operation causes the contents of the specified base block to be printed on

the system print device. As with other operations, no check is made to see that the I/O area is of the right size.

The B←D operation converts the contents of the right-hand field from the zoned decimal format to binary format and stores the result in the left-hand field. D←B converts the right field from binary to zoned decimal and stores the result in the left-hand field. (Note that B←D is a single symbol and is not (B,←,D) which is a store operation.)

Tests

<RELATION> ::= >|<|≤|≥

<ARITHMETIC TEST> ::= (<QUANTITY>,<RELATION>,<QUANTITY>)

<EQUALS> ::= =|≠

<POINTS> ::= =o|≠o

<BASIC TEST> ::= (<SEQUENCE>,<EQUALS>,<EITHER>)|
(<FIELD SEQUENCE>,<POINTS>,<SEQUENCE>)|
(<SEQUENCE>)|(¬<SEQUENCE>)

<EITHER> ::= <BLOCK SEQUENCE>|<QUANTITY>

Tests only have meaning when they occur in test statements (see the following section).

The equality test checks to see if the contents of two fields or blocks are the same. In case they are not of the same size, the smaller one is padded on the left with zeroes. This is analogous to the store operation (i.e., after storing one field or block into a larger one, an equality test on them will yield true).

A pointer test =o is true when the field sequence and sequence designate the same storage location.

The logical test (<SEQUENCE>) is true if the designated field or block is non-zero. The ¬ symbol indicates reversal of the sense of the test.

Branching and the Composition of a Program

<CONTROL OPERATION> ::= (GOTO,<QUANTITY>)

<OPERATION> ::= <FIELD OPERATION>|<SHIFT OPERATION>|<COMPLEMENT OPERATION>|
<STORE OPERATION>|<POINTER OPERATION>|
<CONTROL OPERATION>|<I/O OPERATION>|<CONVERSION OPERATION>

<OPERATION SEQUENCE> ::= <OPERATION>|<OPERATION SEQUENCE>,<OPERATION>

<IF WORD> ::= IF|IFANY

<TEST SEQUENCE> ::= <TEST SEQUENCE>,<TEST>|<TEST>

<TEST STATEMENT> ::= <IF WORD><TEST SEQUENCE>,THEN,<OPERATION SEQUENCE>

<DEFINITION> ::= <BLOCK DEFINITION>|<FIELD DEFINITION>

<STATEMENT> ::= <DEFINITION>|<TEST STATEMENT>|DO<OPERATION SEQUENCE>

<LABELED STATEMENT> ::= <IDENTIFIER><STATEMENT>|<STATEMENT>

Examples:

```
LABEL1  IFANY   (ABF,=,ABE),(ABF,=o,G),THEN,(ABH,←BIG),(GOTO,LABEL2)
        DO      (X,+,3),(Y,*,5),(ABCF,←,AG)
```

The control operation, GOTO, specifies an unconditional branch to the storage location designated by the operand. Normally this will be the identifier of a labeled statement, but this address may be computed, since a field sequence may be used as the operand.

All operations of the operation sequence of an IF statement are executed if the logical value of every test of the test sequence is true. Otherwise control passes to the next statement in the program. The operations specified by an IFANY statement are executed if at least one test of the test sequence is true. Since negatives of all tests exist, "IF NONE" and "IF NOT ALL" are not needed.

If an operation sequence is preceded by DO, the operations in the sequence are performed unconditionally from left to right.

Abbreviations

The *1 instructions described in the preceding sections will perform many of the simple tasks that are common to list processing or systems programming applications. Other more complex tasks can be carried out by macros written in terms of *1 instructions. For example the bookkeeping involved in dynamic allocation of work areas can be coded easily in *1. The operation of inserting or deleting a block in a linked list is another example. Important jobs like these will be included in *1 as library macros. New ones will be added to the library whenever they are in sufficient demand. The following abbreviations are in the implementation of *1 at CMU. Each <... OPERATION> is to be added as another alternative to <OPERATION>.

<PUSH> ::= ↓<FIELD SEQUENCE><FIELD NAME>

<POP> ::= ↑<FIELD SEQUENCE><FIELD NAME>

<LEFT SIDE> ::= <PUSH>|<FIELD SEQUENCE>

<RIGHT SIDE> ::= <POP>|<QUANTITY>

<PUSH-POP OPERATION> ::= (<LEFT SIDE>,←o,<RIGHT SIDE>)

PUSH or POP, operates on a list of blocks which are linked by a specified field. The field sequence portion of a PUSH or POP points to a block in the list. The field name portion indicates which field links the blocks together. The POP operation deletes the specified block from the linked list and causes the field designated by the left side portion to point to the deleted block. For example, the POP operation

(W,←o,↑XB)

accomplishes the following:

(W,←o,X),(X,←o,XB)

The PUSH operation inserts a specified block into the linked list at a prescribed point. The right side is a field which points to the block to be inserted. The left side is a PUSH whose field sequence specifies at which point in the list the block is to be inserted. The field name portion of the PUSH specifies at which point in the list the block is to be inserted. The field name portion of the PUSH specifies the line field. The effect of

(↓WCC,←o,X)

is

(XC,←o,WC),(WC,←o,X)

Currently, PUSH and POP cannot be combined in the same operation, but we hope to correct this. (Normally (↓XA,←o,↑YA) has the effect of (YA,←o,X),(X←oY), (Y,←o,YA) but this requires a temporary store to obtain what is really wanted.)

<SUBROUTINE OPERATION> ::= (CALL,<QUANTITY>)|(RETURN)

HUNT LIBRARY
CARNEGIE-MELLON UNIVERSITY

The CALL and RETURN abbreviations provide a subroutine mechanism. CALL saves the address of the operation following the CALL and transfers control to the designated subroutine. Addresses are saved in a stack so that recursive calls of subroutines are allowed. RETURN occurs in the subroutine; it causes a transfer to the last address saved by a CALL and the popping of the stack. A subroutine requires no special declaration and its body need not be contiguous. It is merely entered by a CALL and left by a RETURN. In fact, the same program label may be used for both subroutine calls and GOTO's.

```
(CALL,LAB)
```

is an abbreviation for

```
DO ([MS],+,1)*
DO ([MS][F0],←o,next)
DO (GOTO,LAB)
```

where next is the address of the next instruction,

```
FIELD F0,0,(0,31),
```

and MS is a base field which is a pointer to the mark stack. The storage for this stack must be set up by the programmer and MS must be initialized.

```
(RETURN)
```

is an abbreviation for

```
DO ([MS],-,1)
DO (GOTO,[MS][F1])
```

where

```
FIELD F1,1,(0,31)
```

---

* MS, F0, and F1 must be preceded by a $, see Implementation.

Implementation and the Meta-Language

The current implementation of *1 consists of about 70 macros. Only a few of their names occur in the *1 language described in the preceding sections. When one of these macros occurs in a program a sequence of nested macro calls produces the code required to perform the required task. It is not unusual for a simple *1 operation; e.g., the ← operation, to produce 15 or more macro calls. Most of these calls are for *1 internal macros; that is, those whose names do not appear in the *1 syntax. However, there is no reason, in general, why a program should not call one of the underlying macros if it serves the programmer's purpose. An effort has been made to make these internal features available to users. They are described in detail in a separate paper, *1 Internal Specifications.

Included in *1 is a language - the meta-language - which may be used to regulate or alter the code produced by *1 macros. Users who want to use this feature of *1 may want to acquaint themselves with the internal operation of the language, by using the internal specs and/or a listing of the macros. The meta-language will be explained in a subsequent version of the manual.

Register Allocation

The instructions produced by *1 macros make extensive use of general registers for indexing, masking fields, etc. For this reason it is necessary to follow special procedures when using registers in assembly language within a *1 program. The *1 compiler does its own register allocation. Therefore, when the programmer wants to use a new register himself, he must either ask *1 to give him a register to use or he must take one which *1 is not using and tell

1 not to use the register. This communication is done with five internal macros: GREG, RREGS, SAVE, RET, and GPAIR. The general method of operation of these macros is as follows: A single tag (i.e., entry in a vector) is associated with each general register. Each tag may be 0, 1, or 2. At any time during the compilation of a program, a 0 indicates that the register is not in use, a 1 that it is in temporary use by *1, and a 2 that it is in permanent use either by *1 or the programmer. The values of these tags are manipulated as follows:

GREG	Changes the first 0 tag to 1
GPAIR	Changes the first consecutive even-odd pair of 0's to 1's.
RREGS	Changes all 1 tags to 0.
SAVE R	Sets the tag of R to 2.
RET R	Sets the tag of R to 0.

GREG requires no parameter. It returns the number of the first available register by means of the SETA symbol &R. All registers must be obtained via &R. Hence, if more than one register is required the user must assign successive values of R to different SETA symbols.

This procedure is illustrated below:

```
GBLA    &R
LCLA    &S
GREG
SETA    &R
GREG
instructions involving &S and &R
```

* The names of all internal macros are preceded by an X, and the names of all global SET symbols, fields, and blocks used by *1 are preceded by a $ even though they are not written that way in this manual.

GPAIR requires no parameter. It returns in &R the number of the first available pair of registers such that the first is even numbered.

RREGS returns all registers which have not been protected by the SAVE macro to the store of available registers.

SAVE requires one parameter. It protects the specified register from the RREGS macro.

RET requires one parameter. It returns its register to the store of available registers.

Note: RREGS is called automatically at the end of each *1 <OPERATION>, so any register which needs to be safe over an operation must be protected by SAVE.

Example

This is a simple *1 program to find the last element on a list. The list is pointed to by LIST and linked by L. It has a 0 in the link field of the last element. T will act as a temporary and will point to the result when it is finished.

```
            BFIELD    LIST,...
            BFIELD    T,...
            FIELD     L,...,(...,...)
            DO        (T,←o,LIST)
BACK        IF        (TL,=,0),THEN,(GOTO,OUT)
            DO        (T,←o,TL),(GOTO,BACK)
OUT
```

References

[1] IBM System/360 Disk and Tape Operating Systems Assembler Language Specification, Form C24-3413-3.

[2] Knowlton, Kenneth C., "A programmer's description of L6," Communications of the ACM, August, 1966.

[3] Naur, Peter, (ed.) "Revised report on ALGOL 60," Communications of the ACM, January, 1963.