AD 699 928

CSLx (x=6,7) A PROGRAMMER'S MANUAL TO THE USE AND UNDERSTANDING OF A LOW-LEVEL LINKED LIST STRUCTURE LANGUAGE

W. Jack Bouknight

Illinois University
Urbana, Illinois

30 November 1969

AD699928

CSL$_x$
(x = 6, 7)

A PROGRAMMER'S MANUAL TO THE USE AND UNDERSTANDING OF A LOW-LEVEL LINKED LIST STRUCTURE LANGUAGE

by

W. Jack Bouknight

This work was supported in whole by the Joint Services Electronics Program (U.S. Army, U.S. Navy, and U.S. Air Force) under Contract DAAB 07-67-C-0199.

CSLx

(x = 6, 7)

A PROGRAMMER'S MANUAL TO THE USE AND UNDERSTANDING

OF A LOW-LEVEL LINKED LIST STRUCTURE LANGUAGE

by

W. Jack Bouknight

November 30, 1969

ACKNOWLEDGEMENTS

Special thanks goes to Sandra Bowles who typed the final draft of this manual. Were it not for her knowledge of publication preparation, the result would surely have been less satisfactory.

As it is, time was not available to completely perfect the presentation and the reader will have to bear with us as best he can. In fact, a special vote of appreciation must go to those hardy pioneering souls who have been using the CSLx system for lo these many years with nothing to guide them but direct communication with the author. Were it not for their helpful suggestions and criticism, this language and this manual for its use might never have come to pass.

Jack Bcuknight
November 30, 1969

TABLE OF CONTENTS

CHAPTER 1. Introduction

In keeping with the effort to upgrade the CSL computer system in software, a need was recognized in the summer of 1967 for some type of list structure manipulation language which could be implemented on the CDC 1604 and integrated into the new CSL computer operating system ILLSYS.

During the summer of 1967, the author was introduced to L^6 (Bell Laboratories Low-Level Linked List Language) which was developed at Bell Labs by K. C. Knowlton. We reproduce some of the introductory comments by Mr. Knowlton from his article describing the L^6 system.[1]

> Bell Telephone Laboratories Low-Level Linked List Language (L^6, pronounced "L-six") contains many of the facilities which underlie such list processors as IPL[2], LISP[3], COMIT[4] and SNOBOL[5], but it permits the user 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 linked data structures....
>
> ....Important features of L^6 are: the availability of several sizes of storage blocks, a flexible means of specifying within them fields containing data or pointers to other blocks, a wide range of logical and arithmetic operations on field contents, and an instruction format in which remote data is referenced by concatenating the names of fields containing the succession of pointers leading to this data....
>
> ....L^6 data structures are made by fetching from a storage allocator blocks of many sizes, and linking them by pointers which are planted in fields which the programmer himself defines....Relative sizes of blocks go as powers of 2; thus the storage allocator can easily split large blocks of free storage into smaller ones and, conversely, can easily fit pieces back together to reconstitute large blocks if and when their parts are simultaneously free....
>
> ....In general, L^6 is useful where storage allocation is microscopic and dynamic or where the programmer wants the pattern of pointers among data items to correspond closely to the physical or logical structure of the objects with which his program deals (electronic circuits, communication

networks, strings of text, parsed sentences and formulas, search trees) as in simulation, game playing, symbol manipulation, information retrieval and graph manipulation. It can also serve as the means for implementing quickly, and in a relatively machine-independent way, higher-level list languages which contain more powerful operations for specific problem areas.

The CSLx (x = 6,7) system is the result of implementing the basic concepts of the L^6 language on the CDC 1604 computer system under the control of the Illar System (ILLSYS) developed by the Computer Group at CSL. The CSLx language is a superset of the original L^6 language and includes the following features:

-- two methods of storage allocation
-- direct coupling to FORTRAN system functions and subroutines
-- contains facility for embedding machine language statements (ILLAR language)
-- floating point arithmetic
-- user defined pushdown-popup stacks
-- generalized format I/O statements
-- computed control transfer statements
-- pseudo-subscripted field declarations
-- DO operations with arguments

The organization of this manual is somewhat like the structure of a tree. The entire work requires a good foundation of knowledge of the basic precepts of linked-list storage systems. Chapter 2 gives a brief initial development of strings, storage blocks and pointers. Chapter 3 discusses the basic syntax of the language and gives the formats of the statements, operations and programs in the language.

The trunk of the tree is made up of the operations of the CSLx language. These include storage control (Chapter 4), data manipulation (Chapters 5 - 6).

Extending from the trunk of the tree are the branches which correspond to operations statements of the CSLx language. These include control of program flow (Chapters 7 - 8) and decision-making statements (Chapter 9). Programmer controlled push-pop data stacks and basic I/O statements complete the manual (Chapters 10 - 11) followed by some sample programs (Chapter 12).

This manual is a compromise between an outline and a textbook. It is assumed that programming experience has been acquired by the reader, not necessarily with list-structures. We make no attempt to treat list-structures themselves beyond a brief look at linked lists since CSLx is a general blocked-storage system. If the reader needs further information about the ILLSYS operating system on the CSL 1604 computer, he should consult with members of the Computer Group.

CHAPTER 2. List Structures, Blocks, Fields, Bugs and Pointers

Section 2.1 Overview of Data Storage Elements

The general method of data storage used in computer memories for mathematical programs is the array structure. An array is a block of contiguous memory locations (words) where the lowest word is labeled with the name of the array and individual elements of the array are obtained by specifying a subscript (index) which when appended to the array name uniquely designates the desired element. As an example, assume an array ALPHA exists. The tenth element of ALPHA would be specified bv ALPHA(9) where the array begins at ALPHA(Ø).

Relationships between elements in an array are specified by operations on the indices of the elements. Suppose ALPHA contains x,y pairs of cartesian coordinates of some curve. An x coordinate lies in element ALPHA(I) and the y coordinate lies in element ALPHA(I+1). Thus, given the index I of some x coordinate in ALPHA, the index for the associated y coordinate is I+1. Furthermore, given the index I of the x coordinate of some point on the curve, the index of the x coordinate of the next point is I+2.

Many cases of data storage arise where the relationships between data elements or blocks of data elements are not conveniently specified in terms of operations on indices c linear arrays. To satisfy this need for a more general data linking, list-structures (strings) were developed.

The key defining feature of list-structures is an element called the link. Relationships between blocks of data are specified in the manner in which the blocks are linked together. What is a link? An illustration if we may.

Suppose we have three (3) sets of cartesian coordinates, x_1y_1, x_2y_2, and x_3y_3. Each coordinate is contained in one computer word and the y coordinate lies in word m+1 where the x coordinate lies in word m.

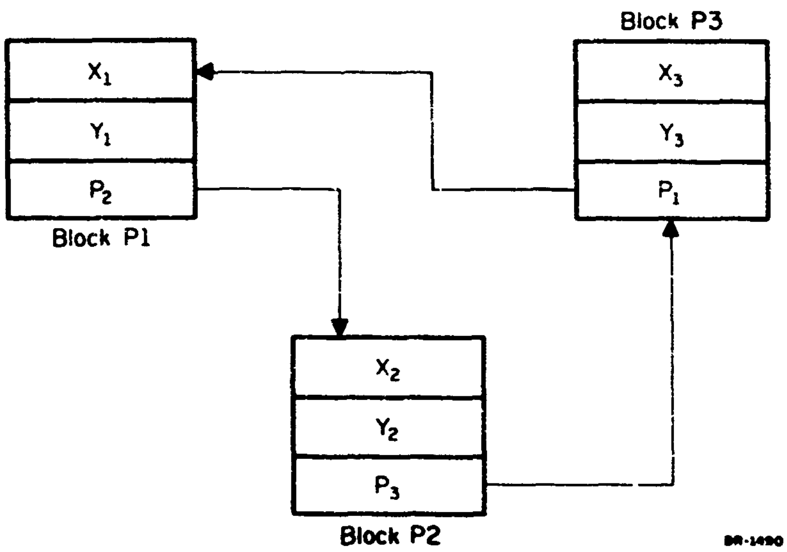
Let us append one more computer word to each coordinate pair to form a block of three (3) words. This extra word will be used to hold a link for use in "stringing" the blocks together into a list-structure.

Assume that the coordinate pairs lie in computer word blocks beginning at locations P1, P2 and P3 (called the base addresses of the blocks). Let us define a pointer as the contents of a computer word which contains the computer representation of the base address of some coordinate block.

Now let us place a pointer in the third word of each block as follows:

x_1y_1 block third word holds P2
x_2y_2 block third word holds P3
x_3y_3 block third word holds P1

We now can state that the third word of the x_1y_1 block contains a pointer to the x_2y_2 block, etc. We pictorially represent out data in the figure below. By knowing which block we are looking at in any instant in time, we can search the third word of the block for a pointer to another block. This concept states the link between the two blocks.



Linked List of Three Blocks
Figure 2.1

Note that the three coordinate blocks may lie in non-contiguous sections of the computer memory. This is the inherent power of the list structure when combined with the ability of using the links to specify relationships between data blocks in storage analogously to the relationships in the actual conceptial data.

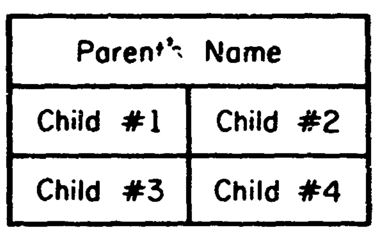
The actual representation of a pointer in a computer system with 32,768 words of memory would be a 15-bit binary address of the base address of some block of words. Typically, the same computer will have a word size of N bits where N > 15. Thus, we are wasting N-15 bits of the third word of each coordinate block in the above example. We can solve the problem of wasted space by a concept of subdivision of a word into elements called fields.

Fields are usually defined in a global manner relative to block base addresses. They are also specified as all bits in a computer word delimited by a left-most bit and a right-most bit. For instance, suppose we define field POINT as the third word of any block, and consisting of the $(n-16)^{th}$ bit through the $(N-1)^{st}$ bit in the word. Thus, the pointer in the P1 block would be found in element P1(POINT).

Note that the form of the descriptor of the desired element is analgous to the array subscript notation. Because pointer search routines actually trace the pointers in the description to reach the desired data element, there is no reason why successive pointer "strings" cannot be used. By starting at block P1, the pointer in block P2 can be addressed by the descriptor P1(POINT(POINT)). The search starts at block P1 and its field POINT. Because P1(POINT) is not the end of the "string", the field P1(POINT) is accessed for the pointer P2 and the search continues at block P2.

At this point the original descriptor has in effect been reduced to P2(POINT) and since this terminates the "string," the field POINT is accessed for a data element, the pointer to P3. We are now free to define the reamining (N-15) bits of word 3 as some other data or pointer field if we desire.

A quick example of a conceptual data structure that is easily stored in a computer memory in list-structure form is a family tree. Using a block and field structure:



DR-1491

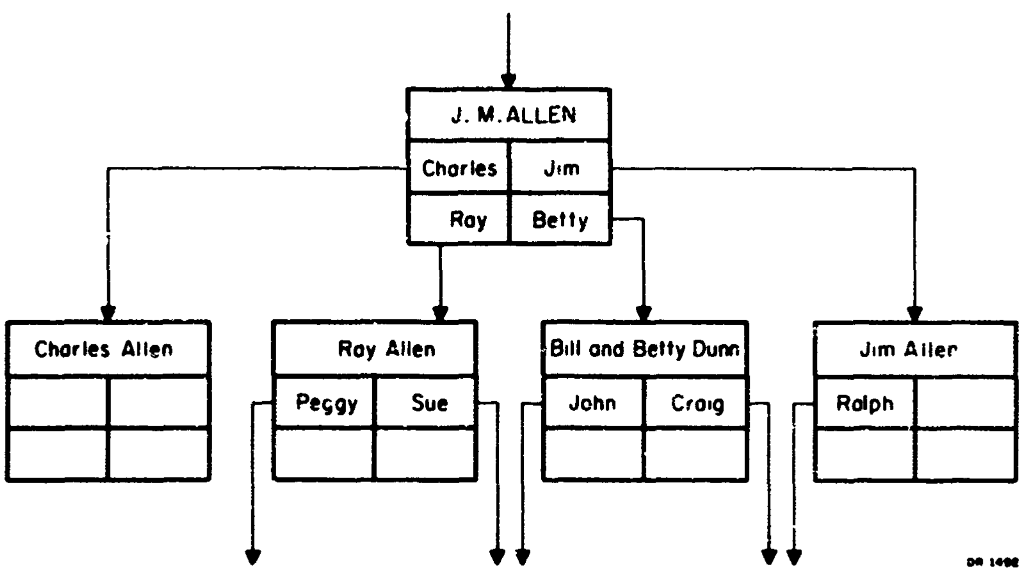
Block with Five (5) Fields
Figure 2.2

we might arrive at the structure in Figure 2.3.

Each pair of parents is indicated by a block in the structure. The first word of the block holds the name of the parents. Each field CHILD #x holds a pointer to the resulting block defined for that child.

We will not discuss the basic theory or operations on list-structures any further at this point. Examples of their usage will be given later with particular emphasis on how they may be handled using the CSLx language.

We begin at this point to elaborate on the CSLx system, its syntax and usage.



Family Tree Structure
Figure 2.3

Section 2.2 Storage Blocks

The basic element of storage in the CSLx system is the block. A block may contain 2^N words in the CSL6 system (N = Ø-7) and 1-32 words in the CSL7 system. The words in a block are numbered contiguously from zero (Ø) to N-1 for a block of N words. For purposes of discussion, we will adopt the notation N-block when we discuss a block where N is the number of words in the block.*

The global storage area (GSA) is defined at program loading time as the "free storage" area bounded at the low end by the system location MEMEND and at the upper end by location COMNBEG. MEMEND is the first location above the end of the user's program and subroutines. COMNBEG is the lowest location of COMMON as defined in the user's program and subroutines.

Control of the use of the GSA is performed by two system routines: L6STORAG or L7STORAG. The GSA is partitioned into blocks and strung together in lists called the unused N-blocks lists (UBL_N). The user must initially instruct the storage allocator (SA) (either L6STORAG or L7STORAG) as to the maximum size block which will be needed by his program. Storage is then partitioned into as many maximum size blocks as is possible. Then the remaining storage is partitioned into the next smaller size of block. This continues until all of GSA is partitioned into blocks.

All of the 1-blocks are then strung together in the unused 1-block list (UBL_1). All of the 2-blocks are placed in the UBL_2. This process continues up to the M-blocks where M is the maximum size block to be needed.

During the execution of the user's program, requests are made to the storage allocator (SA) for blocks from the GSA. If such a block is available, the program receives from the SA a pointer which enables it to work with the requested block. Pointers are 15-bit quantities and therefore, require that fields where they are held are large enough to hold at least 15-bits of information. Further discussion of pointers must await a description of fields and bugs which are described later in this chapter.

What happens if no block of the requested size is immediately available from the GSA? For this occurrence, separate actions are taken in the CSL6 and CSL7 systems. We discuss the CSL6 system operation first.

Suppose the program requested an N-block from storage. Since the UBL_N is empty, the SA searches the higher UBL_i for the next UBL_i which is not empty. If UBL_k contains a K-block, the following occurs.

*Original notation in Bell Labs report.[1]

Suppose $N = 2^j$ and $K = 2^i$. Then the K-block is divided into two L-blocks where $L = 2^{i-1}$. These L-blocks are placed in UBL_L. If $N \neq L$, then K is set equal to L and the division process is repeated with one of the blocks from UBL_L.

When N = L, a block of the requested size is now available and the SA completes its operation by passing the pointer of the requested block to the program. The remaining half of the divided block is left in UBL_N.

The possibility exists that no UBL_i above UBL_N contains a block. In other words, there are no other unused blocks in the GSA that are larger than the requested size block. When this condition occurs, the SA performs a "garbage collection" operation. "Garbage collection" consists of recombining smaller blocks into larger blocks until all possible pairs have been recombined. A complete recombination is performed on all blocks smaller than the requested size starting with 1-blocks and working up. If, after "garbage collecting" is complete, there is still no block of the requested size, then a system error message results informing the user that no more unused blocks of the requested size exist and a return is made to the system monitor (CSLMCS).

In the CSL7 system, the same procedure of dividing larger blocks into smaller blocks is used to produce a block of the required size. Suppose the program is requesting an N-block. The SA finds that it has no N-block but it does have a K-block (K = N+4 for purposes of discussion). The SA will simply divide the K-block into an N-block and a 4-block. The 4-block will be added to UBL_4 and the pointer for the N-block will be returned to the program.

No recombination is allowed in the CSL7 system. The reason for this will be explained later. Because of no recombination, the SA must declare no more unused blocks if it cannot find some UBL_i higher than UBL_N with at least one i-block available.

Section 2.3 Fields

The basic element of storage for data and pointers is the field. Fields fall into two (2) categories: Fullword Fields (FWF) and Variable Length Fields (VLF). A fullword field (FWF) is one 1604 word, i.e., 48-bits in length. Variable length fields (VLF) may be any length from 1 to 48 bits long and may reside in any portion of a 1604 word.

VLF are designated internal to storage blocks while FWF are separately defined in the user program or his subroutines. Literals are FWF and are defined in each user program or subroutine. Literals may be read from but not written into during program execution. FWF defined internal to a given program or subroutine for use as a data storage location are called internal fields (IF). FWF used in a program or subroutine for data storage but defined externally in some other program or subroutine are called external fields (EF). The various means by which FWF are designated in a program or subroutine will be detailed in a later section.

Let us turn our attention for the present to a discussion of variable length fields (VLF). The VLF and the block structure are the basic attributes of CSL6 and CSL7 that give the languages their power and utility.

Recall that a block is a contiguous set of 1604 words and pointed to by a pointer. Figure 2.4 shows the schematic of a 4-block. The divisions within the block are called VLF. They may lie anywhere within the block, they may overlap one another, and they may even coincide in some cases.

Two different arrangements are shown: one for a CSL6 block and one for a CSL7 block. Two areas in the CSL6 block ① and one area in the CSL7 block ② are crossed out. These areas are:

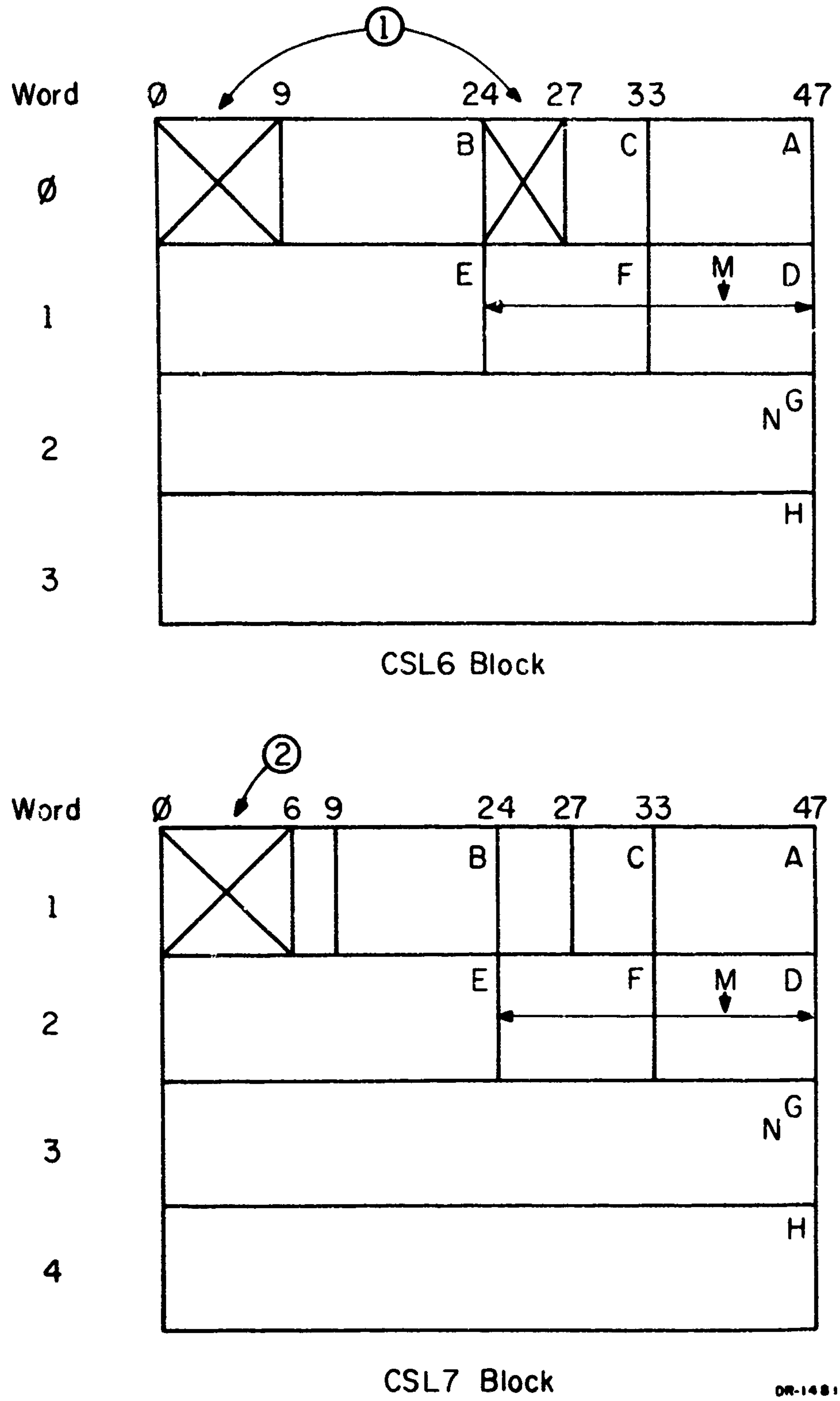
①	Word Ø	Bits Ø- 8
	Word Ø	Bits 24-26
②	Word Ø	Bits Ø- 5

CSLx system information is kept in these areas and therefore, the user is not allowed to assign VLF covering these areas.

A VLF is defined by specifying three (3) parameters:

1. Word bias
2. Left bit boundary
3. Right bit boundary

A listing of the parameters of the VLF in Figure 2.4 will best illustrate their meanings. The VLF letter name (which may be A-Z, Ø-9) is shown in the upper right hand corner of each VLF area.



A 4-Block in CSL6 and CSL7
Figure 2.4

Field	Word Number	Left Bit	Right Bit
A	Ø	33	47
B	Ø	9	23
C	Ø	27	32
D	1	33	47
E	1	9	23
F	1	24	32
G	2	Ø	47
H	3	Ø	47
M	1	24	47
N	2	Ø	47

Note that VLF G and N coincide and VLF M overlaps F and D.

*The user must remember the following rule concerning VLF specifications: every VLF specification applies to every block in use by the user program or subroutine. The word bias parameter is relative to the beginning of any block and when taken together with the pointer to a block, the result is a unique address in the 1604 memory.

Section 2.4 "Bugs"

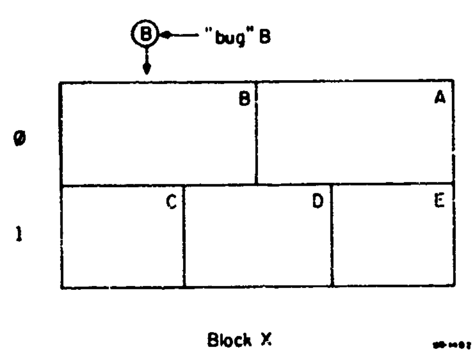
An electronic computer is usually designed with one or more full word registers where data manipulation may occur. In the CSLx systems, 26 registers, referred to as "bugs," have been set aside for use as data registers or pointer registers. These registers, hereafter referred simply to as "bugs," are actual 1604 memory locations, not hardware registers, but the use is the same.

The notation "bug" comes from the original Bell Laboratory L^6 Report[1]. Linked-list structures can be likened to beads on a string. Since "bugs" hold pointers to blocks which may reside on the string, the blocks are referenced through the "bug" depending on where the "bug" points, or for the analogy, where the "bug" sits. Moving pointers up and down the list corresponds to the "bug" crawling up and down the string.

As a general descriptive convention, a "bug" is indicated pointing to a block as shown in Figure 2.5. "Bug" B holds the pointer to block X.

Any one of the 26 "bugs" may also be used for data manipulation. A "bug" is 48-bits in length and therefore, falls into the FWF category. They are referenced in the CSLx program with a field string field descriptor which will be described in Section 3.2.1.2.

"Bugs" are automatically set up by the CSLx compilers in the user's MAIN program. Each "bug" is also made as an entry point. Therefore, all subroutines reference "bugs" as external symbols and allow a single set of system "bugs" to suffice for all the user's program and subroutines. Obviously, this means that not more than one MAIN program may be loaded into memory at a given time.



A 2-Block Pointed to by a "Bug"
Figure 2.5

CHAPTER 3. The Basic Syntax and Format of Statements and Programs in the CSLx System

Section 3.1 Overview

Syntax descriptions of the basic elements of the CSLx language are our first order of business (Sec. 3.2 - 3.5). The discussion will then advance to combinations of the basic elements into the various statement forms (Sec. 3.6 - 3.7). Finally, we describe the form of the programs and subroutines (Sec. 3.8 - 3.9).

Section 3.2 Basic Data Descriptions

Two classes of syntax elements describe data. The data descriptor is the notation for describing some field in core storage whether it be a full word field (FWF) or a variable length field (VLF). Literals describe explicit forms of data such as numbers or hollerith character strings.

Section 3.2.1 Data Descriptors

There are three (3) types of data descriptors: internal full word fields (IF), external full word fields (EF) and field strings which reference VLF.

Section 3.2.1.1 Internal and External Full Word Fields

Internal full word fields (IF) are used for reference to FWF which are defined internally to the program or subroutine where the reference is made. Any FWF is labeled with up to eight (8) BCD characters under the label convention of the ILLAR language.[7] The data descriptor has the form:

/XXXX

where XXXX is the label attached to the FWF. A form of pseudo-subscripting is allowed on IF's. A pseudo-subscripted IF has the form:

/XXXX±(exp)

where (exp) is an arithmetic expression made up of the operators + and/or - and literals and/or other data descriptors. Section 3.2.1.3 discusses pseudo-subscripting further.

External full word fields (EF) follow the same conventions as for IF except that the referenced FWF is defined in a program or subroutine other than the one in which the reference is made. The form of the data descriptor is:

*YYYY

Pseudo-subscripting is also allowed in the same manner as for IF.

Section 3.2.1.2 Variable Length Fields and Field Strings

A variable length field (VLF) is referenced through a field string data descriptor. A field string describes a string of pointers which eventually point to the destination field where the desired data is to be found or stored. Recall that a pointer denotes the Øth word of some n-block in memory. The notation for discussion is shown in Figure 3.1. The basic format for a VLF field string is as follows:

BTT...TR

Each of B, T and R are single characters. B designates one of the 26 "bugs" (A-Z). T and R designate field names (A-Z, Ø-9).

The "bug" B and each T designate where pointers are to be found. R is a field to be referenced, either for fetching or storing of data or a pointer. To get to R, a "trace" is made in the following manner:

> The "bug" contains a pointer to some block. If there are no T in the field string, then R lies in the block "pointed to" by the "bug." If there are one or more T in the field string, then the first T field lies in the block pointed to by the "bug" and contains a new pointer to a block. Each successive T field lies in the block pointed to previously and contains a pointer to a block. The R field lies in the block "pointed to" by the last T field and can be referenced from there.

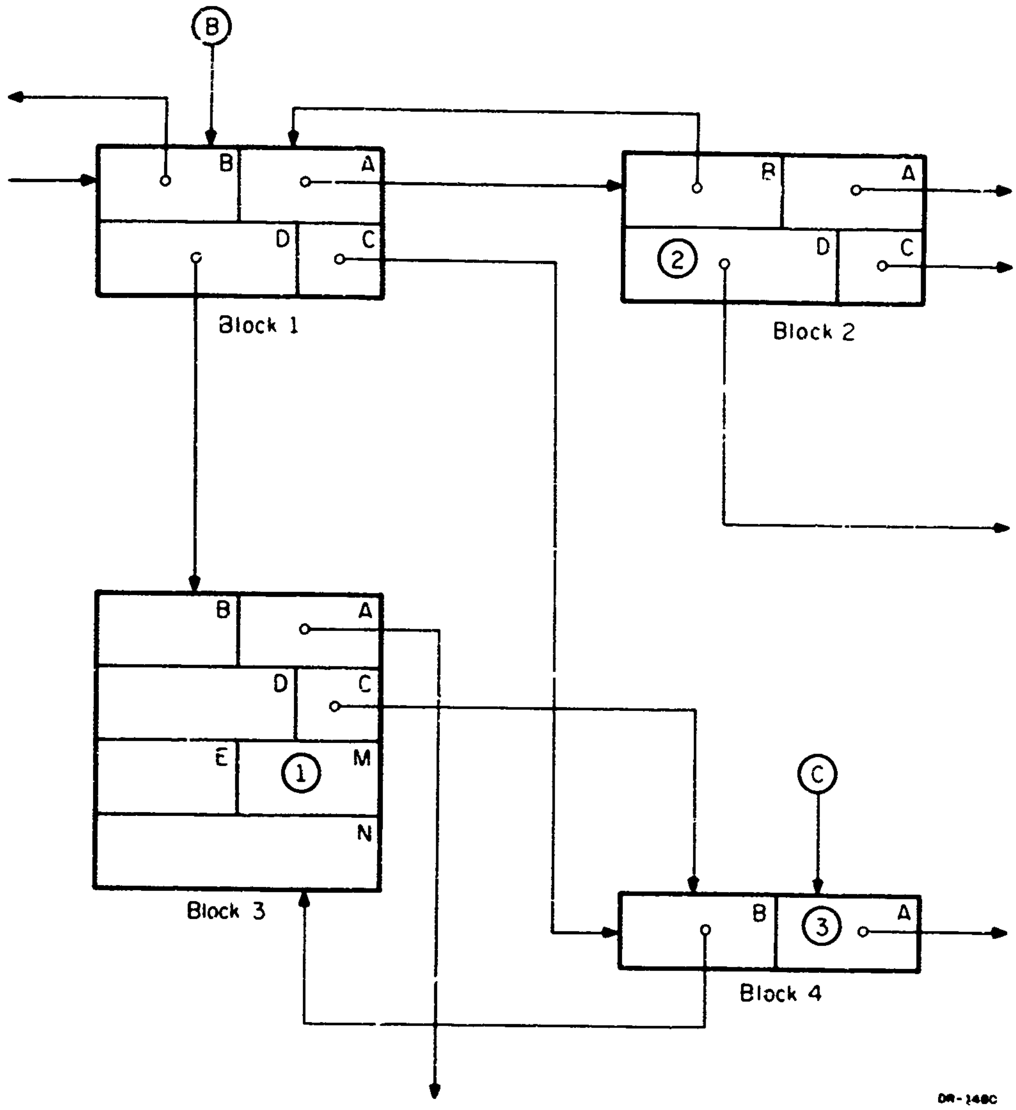
A special case of the VLF field string occurs when only one alphabetic character appears in the string. There are, therefore, no T's and no R. Thus, the indicated "bug" is to be referenced directly for fetching or storing.

Let us illustrate using Figure 3.1. There are three (3) VLF singled out and numbered as ① ② and ③ VLF ① may be referenced in one of the following ways.

BCM	(1)
BCBM	(2)
CBM	(3)
BDCBM	(4)

Let's look at field string (1). "Bug" B "points" to block 1 whose C field "points" to block 2 whose M field is the desired field for reference. (2) states that "bug" B "points" to block 1 whose C field "points" to block 4 whose B field "points" to block 3 which contains the M field. The reader should now be able to follow the "trace" to arrive at the desired M field by any of the indicated paths. For VLF ② only one path can be taken:

BAD



A Typical Linked-List Structure
Figure 3.1

While for VLF (3) the following paths may be taken:

BCA
CA
EDCA

The reader is encouraged to plot the paths from either of the "bugs" B and C to any of the VLF's for further practice and understanding of the VLF referencing algorithm.

Section 3.2.1.3 Pseudo-Subscripting of Internal and External FWF

In order to allow completely general compatibility between the CSLx list-structure system and the more common array-structured systems, FORTRAN and ILLAR, some form of subscripting in linear arrays is necessary.

In the CSLx system, an IF or EF may be treated as a linear array and indexed in a pseudo-subscriptive manner by use of a data descriptor of the following form:

/field ± index1 ± index2 ±......± indexN
*field ± index1 ± index2 ±......± indexN

field is the label assigned to the referenced EF or IF which becomes the zeroth element of the array field.

The string of indexI elements separated by + or - forms an arithmetic expression which when evaluated provides the bias used to index the array field. The elements indexI may be any form of data descriptor or decimal/octal literals.

Some examples will further illustrate:

/BUFFER+10B	internal field - octal literal index
/BUF+25	internal field - decimal literal index
/LIST-/INDEXCT	internal field - internal field index
/STRING-BAD	internal field - string index (field)
/BUFR+*EXTINDEX	internal field - external field index
*BUFA+345B	external field - octal literal index
*BUFB-21	external field - decimal literal index
*BUFC-/TINDEX	external field - internal field index
*BUFRA+DART	external field - string index (field)
*BUFL-BUFEXT	external field - external field index

3.2.1.4 Literals

Literal data descriptors explicitly define data during an operation. There are four types of literal elements allowed in CSLx programs:

1. octal
2. decimal
3. floating point
4. hollerith

We choose not to discuss each type of data in detail because the literal conventions for CSLx programs are identical to the conventions of the ILLAR language system. The ILLAR system manual may be referred to by the reader to clarify his questions.

The type of literal allowed in a given situation varies greatly and s best explained when necessary.

Section 3.3 Basic Operation Unit

For compatibility reasons, the same form for the basic operation unit as used in L^6 is maintained in the CSLx systems. The format is as follows:

(a,op,b,c,d)

op is a series of characters denoting some available operation in CSLx and perhaps one of its modes of operation. As an example, the operation code AO denotes the Addition operation with the operands assumed to be Octal integers.

a, b, c and d, some of which may not be present, designate fields where operands may be fetched or results stored during the course of the operation. A complete description of the arrangements for all of the possible operations will be made in the appropriate section dealing with each operation. We present here for illustration several BOU's simply to show form as they might appear in a CSLx program:

```
(A,E,1)
(/TIME,A,1/CLOCK)
(*DATE,A,-6,/BUFFER+INDEX)
```

In the CSLx system. spaces are ignored except internal to BOU's. There they are counted necessarily because of the possible inclusion in a hollerith literal.

Section 3.4 Test Unit

The test unit (TU) is a special operation unit which makes a test between two items and produces a "vote" of yes or no for a result. Test units are allowed only as part of a test statement (Section 3.7).

The format of the TU is:

(a,t,b)

a and b are data descriptors or literals. t is some relationship (e.g., ≥, >, ≠, etc.). The TU determines if atb is true, yes or no. The yes or no "vote" is used to make a test statement decision during the execution of the CSLx program.

Further discussion of the relational test operators will be made in Chapter 9.

Section 3.5 "goto" Elements

The most basic form of control transfer allowed in the CSLx language is the "goto." A "goto" is simply the label of the statement to which control is to be transferred. This operation is the equivalent of the GO TO statement in FORTRAN. However, only the label of the destination statement is needed.

There are several CSLx system "goto" elements which are reserved for special purposes and therefore, the user may not use them as statement labels:

EXIT	
DONE	(Section 7.2)
FAIL	(Section 7.2)

The DONE and FAIL "goto" elements are connected with subroutine calling operations and are explained in the indicated sections. The EXIT "goto" will cause a transfer of control to the END statement of the program for a subsequent exit to the calling program.

Section 3.6 Program Statement Elements

In a CSLx program, there are two classes of statements: declarative and executable. The declarative statement performs non-executable operations such as storage space definition, global space linkage, program definition, etc. All other statements are called executable statements because they compile operations which are executed only at run time.

For purposes of outline, we choose to list the types of statements at this time but we defer any elaboration until the appropriate section. The declarative statements are:

GLOBAL	(Section 4.3.2)
DEFINE	(Section 4.3.1)
ENTRY	(Section 4.3.1)
DO ENTRY	(Section 7.2)
CALL ENTRY	(Section 7.3)
EXTERNAL	(Section 4.3.1)

Under the heading of executable statements, we have three classes: composite, test, and primary statements. We will discuss in detail the makeup of the composite statement in a moment. The test statement discussion is reserved until Section 3.7. For now, we simply list the members of the primary statement class and give the definition of the class as those statements whose formats are specifically related to their individual functions:

INPUT	(Section 11.2)
OUTPUT	(Section 11.2)
ENDIO	(Section 11.4)
TRANSFER	(Section 8.2)
SWITCH	(Section 8.2)
POPUP	(Section 4.4)
PUSHDOWN	(Section 4.4)
CALL	(Section 7.3)
DEFSTACK	(Section 10.2)
STACK	(Section 10.3)
UNSTACK	(Section 10.4)

The statement most used in a CSLx program is the composite statement. The name of the class is derived from the fact that the statement is made up of a composite of basic operation units (BOU), "goto," and sometimes ended with a primary statement used as a unit.

The arrangement or presence of any or all of the three types of units in a composite statement is governed by the following rules:

1. A composite statement ends after a "goto" or primary statement unit.

2. A composite statement may contain as many BOU elements as desired.

3. Only one "goto" or primary statement unit may appear in a composite statement.

After the reader has read later sections and studied the sample programs, the form of permissible composite statements will be more apparent.

Section 3.7 Test Statements

Test statements are provided for conditional transfer of control in a CSLx program during the execution run. One or more test units (TU) are executed and their "votes" tallied. Action is then taken based on the "vote" according to the type of test statement.

There are four (4) basic test statements covering the four (4) possible outcomes of "voting" tabulations:

IFALL
IFNONE
IFANY
IFNALL

Two shorthand test statements:

IF
NOT

are allowed. IF functions as IFALL and NOT functions as IFNONE with the restriction that only one (1) test unit be used in either case.

The general format of the test statement consists of four (4) parts:

1. LABEL
2. TYPE
3. IF computation
4. Result computation

The LABEL is a standard statement label. TYPE is one of the six (6) mnemonics specified above. The IF computation is a string of one or more TU's (except for IF and NOT). The result computation may be of two forms.

1. a "goto"
2. a composite statement proceeded by the key word THEN

As we give a brief explanation of the four basic test statements, we will also illustrate to clarify the actual source record form.

```
IFALL (A,L,3)  (B,G,Z) THEN (A,L,B) EXIT
IFALL (K,E,3)  (J,N,4) BITE
```

The IFALL statement transfers to the next consecutive statement if any TU "votes" no.

IFNONE (A,G,1) (A,I,8) BADTAPE

The IFNONE statement transfers control to the next consecutive statement if any TU "votes" yes.

IFANY (A,L,2) (B,L,3) THEN (A,E,B)

The IFANY statement transfers control to the next consecutive statement if all TU's "vote" no.

IFNALL (G,G,H) (H,G,I) THEN (I,E,G) OUT

The IFNALL statement transfers control to the next consecutive statement if all TU's "vote" yes.

Note that we have essentially stated the action taken by these test statements in the reverse manner. This is intended to require the reader to do some thinking about the operation of test statements. A good fundamental understanding reduces programming errors and reversed decision-making is among the most common ones.

Section 3.8 Source Language Formats in CSLx Programs

There are three types of source language records in a CSLx program:

1. Comment
2. CSLx source
3. ILLAR source

All three classes of records are ten (10) words long in BCD format.

Section 3.8.1 Comment Source Records

The comment source record class contains just three records:

1. Comment record
2. CSL6 switch record
3. ILLAR switch record

Comment records contain an asterisk (*) in col. 1 with columns 2-80 available for user typed material. Comment records are not compiled but are listed on both the CSLx source listing and when requested, the subsequent ILLAR listing of the compiled program.

The CSLx system has the facility for programmer selection of either CSLx language or ILLAR machine language internal to any CSLx program. To accomplish a switch, either of the switch records:

--ILLAR	col. 1-7
--CSL6	col. 1-6

is placed in the program. All following records up to the next switch record or the end of the program will be treated as of the type of language selected. Even though the length of records in either language is the same, note that comment records assume the tab information of the language selected.

Section 3.8.2 CSLx Source Records

These are four fields in the CSLx Source Record:

1.	LABEL	col. 1-8
2.	CHAIN	col. 9
3.	STATEMENT	col. 10-72
4.	USER	col. 73-80

Tab information is present in the ILLSYS system to allow tab operations to column 10 and 73. Moving to column 9 requires eight (8) spaces.

The LABEL field serves two purposes: 1) to provide a means for statement referencing during EDIT operations on the CSLx program and 2) to provide symbolic references for transfers of control inside or out of the CSLx program. The convention for statement labels is that established for the ILLAR assembly language and we repeat the convention briefly for completeness.

Labels must be left-justified in the field and are restricted to eight (8) BCD characters or less. All of the alphabet and numeric characters may be used in labels subject to some restrictions described below. In addition, two special characters, the period "." and asterisk "*" may be used with the following restrictions: an asterisk may end a label but should not appear within it. A period may not begin a label but may appear within it or at the end.

The following restrictions on symbols beginning with numeric characters are necessary to avoid conflicts with the convention on literals:

1. A single digit number may not be followed immediately by one of the letters f, p or h.
2. Any combination of numeric characters may not be followed immediately by one of the letters b, d, or e.

For illustration, we list here some of the acceptable and not-acceptable forms of labels.

Acceptable	Not Acceptable
a	(a)
a1	*abc
abcdefgh	read*a
231mn	a+b
read*	1b
a.b	.a
a..c	..twofive

The STATEMENT field holds all CSLx statements. Although the field is only 63 characters long, extra long statements can be placed in the STATEMENT fields of successive source records by placing a non-blank character in the CHAIN field of all records in the "chain" but the first. Note that a chain is broken by the next source record with a blank CHAIN field or a comment class record. Labels placed on "chained" records (col. 9 non-blank) will be ignored.

The USER field is simply an eight (8) character field which is reproduced on the CSLx source listing only and can be used in any way desired.

Section 3.8.3 ILLAR Source Records

The conventions of the ILLAR system are well written up in the ILLAR manual. For further details, the reader should contact the system librarian.

Section 3.9 Program Descriptions

A program written in the CSLx (x = 6,7) language system may take one of three forms:

1. Main program
2. Subprogram
3. Subroutine

Each program begins with a header record and ends with the END record. The END record contains END in columns 10-12 and blanks in the remaining columns. The END card may be labeled if the user wishes.

Each of the three program classes is identified by a unique header record:

1. Main programs - PROGRAM
2. Subprograms - SUBPROGRAM
3. Subroutine - SUBROUTINE

The descriptive word begins in column 10 of the header record. The descriptive word is followed by a space and then the program name, up to eight (8) BCD characters.

If arguments are present for the program, they are listed by label on the header record following the name, enclosed in parentheses. and separated by commas. The following are some examples of header records:

```
PROGRAM TEST
SUBPROGRAM TESTER(A,TIME)
SUBROUTINE CLOCK(ARG)
```

To initialize the ILLSYS system to read CSLx format records, a CSL6 language directive should be placed just prior to the header record. The language directive is a record containing --CSL6 in columns 1-6 of the record followed by blanks in the remaining columns.

A program set is a collection of programs which are placed in consecutive order on some input medium to be read and compiled in contiguous order. A program set begins with the first header record read from the medium and ends with a FINIS record. The FINIS record contains FINIS in columns 10-14 with blanks in the remaining columns. In accordance with ILLSYS conventions, two (2) end-of-file records are written after the FINIS record on the medium.

A program set may contain any number and arrangement of programs from the three (3) classes of CSLx programs with the following single exception:

THERE MAY BE ONLY <u>ONE</u> (1) MAIN PROGRAM IN A PROGRAM SET.

Further flexibility in programming is provided by allowing the intermixing of CSLx system programs and ILLAR system programs in the same <u>program set</u>. The user may also store his source records in SQUOZE BCD format which allows a condensing factor of 5 or 6 in the length of the <u>program set</u> on the input medium.

The following is an illustration of a representative <u>program set</u>.

```
--CSL6
          PROGRAM MAIN
          .
          .
          .
          END
--END
--CSL6
          SUBPROGRAM ROUTINE1(ARG1,ARG2)
          .
          .
          .
          END
--END
--ILLAR
          IDENT               ILLAR6
          .
          .
          .
          END
--END
--CSL6
          SUBROUTINE SUB1
          .
          .
          .
          END
--END
--ILLAR
          FINIS
--END
--END
```

CHAPTER 4. Storage Allocation, Field Definition and Manipulation

Section 4.1 Overview

The first operation which must be performed when a CSLx program is executed is to set up available storage in a block structure format. The next operation usually performed is to define the fields which will be used in the blocks. The name of the rest of the game is manipulation of data stored in the fields of various blocks.

The first two topics of this chapter will be presented in detail. The third will be only a beginning since manipulation covers many areas (later chapters). The types of manipulation which will be discussed in this chapter are data - independent such as pushdown-popup in stacks, field interchange, etc.

Since we begin in this chapter to show exact formats of statements and operation units, we will also begin the practice of giving an example in detail for each new disclosure.

Section 4.2 Storage Allocation

In Chapter 2, we explained the two methods of storage allocation available to the use of the CSLx system.

The method used in CSL6 is the fast storage allocation developed by K. C. Knowlton.[6] This method of storage allocation allows for complete recombination of smaller "free" blocks if possible and therefore, allows greatest flexible usage of storage. The penalty paid is in the power of 2 size of blocks.

In the CSL7 system, the flexibility of variable size is allowed at the expense of recombination which somewhat reduces flexibility of storage. The main reason for developing the CSL7 type of storage allocation was due, however, to a need on the part of some users to cut down on the amount of permanent system information attached to each block.

In the CSL6 system, three types of system tags are attached to each and every block obtained from the storage allocator routine (L6STORAG). The first tag is the FREE/INUSE flag and occupies bit Ø of word Ø in every block:

set to Ø if FREE
set to 1 if INUSE

This flag is used by the system debugging routines (Section 4.5) during dump operations.

The second tag attached to each block is the size of the block specified as a power of 2. This tag is placed in bits 24-26 of word Ø. The system uses this tag to identify block size and an operation has been provided for the user which enables him to also read this tag (Section 4.4.1).

The third tag, located in bits 1-8 or word Ø, is storage allocator information. This tag is used during recombination.

In developing the CSL7 storage allocation and block scheme, the third tag is eliminated and the second tag expanded to hold five (5) bits of information, i.e., the actual number of words in the block. The FREE/INUSE flag still lies in bit Ø of word Ø while the count tag has been moved to bits 1-5 of word Ø. Thus, only six (6) bits of system information are used in the CSL7 system as opposed to twelve (12) in the CSL6 system.

The CSLx user is protected from violating the system areas of word Ø as long as he stays in the CSLx language. As soon as he moves into ILLAR, it becomes his responsibility to protect against violations. During the first year of usage of the CSLx system, this has not become a problem.

Section 4.2.1 Storage Allocation Setup Unit

The first execution statement in a CSLx MAIN program should contain a storage allocation setup BOU. This requirement applies only to the MAIN program in a program set.

(SS,d) The Storage Setup operation unit initializes the storage allocation routine and causes all available storage to be dismembered into blocks, the largest of which is specified by d, a positive decimal integer.

In the CSL6 system, d is taken to be either a power of 2 with a maximum of $128(2^7)$ and minimum of 4. In the CSL7 system, d is any integer from 4 to 32.

The (SS,d) BOU also causes all field definitions to be cleared out and all stacks to be cleared. Thus, this operation effectively initializes the user's program and the CSLx system.

Example: (SS,4)

This BOU initializes the storage allocator to partition all available storage into N-blcoks with a maximum value of N = 4.

Section 4.3 Definition of Fields

Recall that there are two (2) classes of fields in the CSLx system: full word fields (FWF) and variable length fields (VLF). The methods of definition of these two (2) classes of fields are completely different and as such, will be explained in separate sections.

Section 4.3.1 Definition of Full Word Fields (FWF)

Since a FWF is of fixed length (48 bits), the user must simply define the label to be attached and whether the field is internal or external. The simplest of these is the external field (EF) and therefore, we will discuss it first.

Briefly stated, the use of an EF data descriptor:

*XXXX

is sufficient to cause the necessary information to be compiled stating that XXXX is a FWF external to the current program.

Situations sometimes arise where the user desires to explicitly declare some labels for EF. The EXTERNAL declarative statement provides this ability:

EXTERNAL,LABEL1,....LABELX

The EXTERNAL statement may appear any place in the CSLx program. Defining the EF may also occur in an ILLAR section of code. Since this is a departure from compiler control, the user assumes all risks.

Example: EXTERNAL, CSLMCS, TAPBINOT

The FWF CSLMCS and TAPBINOT are defined as external to the current CSLx program.

Defining the internal field (IF) is a bit more precise as follows: each IF must be explicitly defined. The definition process is handled through the DEFINE declarative statement:

DEFINE, LABLE1, LABEL2,...,LABELN

An expansion of the capability exists to allow the labels to define arrays by specifying the size of the array in enclosing parentheses:

DEFINE, ALPHA (20)

The DEFINE statement may appear at any point in a CSLx program.

Example: DEFINE, ALPHA, LONG, TWO (2Ø)

FWF labeled ALPHA and LONG will be set aside in the program. A twenty (2Ø) word array labeled TWO will also be set aside.

The ENTRY declarative statement is provided to allow a user to flag selected FWF in one CSLx program to be referenced as EF in another program:

ENTRY,LABEL1,...,LABELN

The labels of the ENTRY statement may refer to arrays in which case, no size parameter is used and the zeroth location of the array is the actual global entry point.

Example: ENTRY, ALPHA, LONG, TWO

Assume that this statement appears in the same program as the previous example. Thus, programs outside this CSLx program may refer to the FWF ALPHA and LONG and also to the array TWO.

Section 4.3.2 Definition of Variable Length Fields (VLF)

The definition of VLF in the CSLx system is a dynamic operation which occurs during execution of the program. A definition may occur at any place and time in any program.

There are three (3) attributes in a field definition:

1. Word position in a block. Counting begins at zero (Ø).
2. Leftmost bit position of the word.
3. Rightmost bit position of the word.

Fields may not overlap word boundaries. Fields may overlap or coincide with other fields. A field definition must occur prior to the first use of that field in a CSLx program. Otherwise, a compiler diagnostic will occur.

Bit positions in the word are numbered Ø to 47 moving from left to right. Due to the organization of the 1604 computer, three fields compile operations which are faster than the general field definitions:

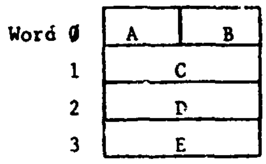
1. bits Ø to 47 - full word field

2. bits 9 to 23 - upper address of 1604 word

3. bits 33 to 47 - lower address of 1604 word

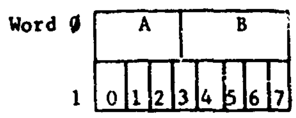
It is to the user's advantage if he can use these arrangements where possible.

Example I:



Field	Word	Left Bit	Right Bit
A	Ø	9	23
B	Ø	33	47
C	1	Ø	47
D	2	Ø	47
E	3	Ø	47

Example II:



Field	Word	Left Bit	Right Bit
A	Ø	9	23
B	Ø	33	47
C	1	Ø	47
Ø	1	Ø	5
1	1	6	11
2	1	12	17
3	1	18	23
4	1	24	29
5	1	30	35
6	1	36	41
7	1	42	47

Fields are defined by using the following BOU:

(w,Df,l,r) This BOU causes a definition of field f to be made at this point in the program during execution. f is a single letter, A-Z or Ø-9.

The fields w, l, and r may be either positive integers or data descriptors of fields where a positive integer can be found. w is the word position of field f in all blocks. l is the leftmost bit position of f and r is the rightmost bit position of f

Error messages occur for illegal values of w, l, and r and if f is not a legal field character name. To aid in debugging, legal values are assumed for w, l and r where necessary as follows:

error	assumption
$\underline{w} < Ø$	$\underline{w} = Ø$
$\underline{w} > 127$	$\underline{w} = Ø$
$\underline{l} < Ø$	$\underline{l} = Ø$
$\underline{l} > 47$	$\underline{l} = 47$
$\underline{r} < Ø$	$\underline{r} = Ø$
$\underline{r} > 47$	$\underline{r} = 47$
$\underline{l} > \underline{r}$	$\underline{l} = Ø$, $\underline{r} = 47$

In CSL6, if $\underline{f}$ covers bits Ø-8 and/or 24-26 and $\underline{w} = Ø$, $\underline{w}$ is set to 1.
In CSL7, if $\underline{f}$ covers bits Ø-5 and $\underline{w} = Ø$, $\underline{w}$ is set to 1.

For examples of the field definition BOU's, we list the BOU's for previous examples I and II below:

Example I:

1. (Ø,DA,9,23)
2. (Ø,DB,33,47)
3. (1,DC,Ø,47)
4. (2,DD,Ø,47)
5. (3,DE,Ø,47)

Example II:

1. (Ø,DA,9,23)
2. (Ø,DB,33,47)
3. (1,DC,Ø,47)
4. (1,DØ,Ø,5)
5. (1,D1,6,11)
6. (1,D2,12,17)
7. (1,D3,18,23)
8. (1,D4,24,39)
9. (1,D5,30,35)
10. (1,D6,36,41)
11. (1,D7,42,47)

Provision is made to allow the <u>field</u> definitions made in one program of a <u>program set</u> to be used in other programs of the <u>set</u>. The <u>fields</u> are specified in the GLOBAL <u>declarative statement</u>:

GLOBAL,a,b,...,z

The a, b,...,z are single letters, A-Z.

There are 3 cases concerning the occurrence of the GLOBAL statement in a program.

Case 1. Field definition - no GLOBAL statement.

The defined field is internal to the associated program and cannot be referenced from the outside.

Case 2. No field definition - GLOBAL statement.

The referenced field is defined in the associated program with external labels so that all field processing routines for that field are located outside the program and linkages are made by the ILLSYS loader.

Case 3. Field definition - GLOBAL statement.

The referenced field is defined in the associated program and each field processor routine for the referenced field is assigned as an ENTRY point. This allows both internal and external routines to reference a given set of field processor routines.

The importance of these cases is that only one definition point for a given field may be allowed to be GLOBAL in nature. Otherwise, there will be more than one set of field processing routines for some field and the system will be unable to handle this ambiguous loading situation.

Incorporated into the CSLx auxiliary systems are pushdown stacks which retain entries containing all necessary information for the definition of some field at a later date with a previous field definition. Field definitions may also be passed to and from subroutines by this means.

(S,FD,f) (R,FD,g)	The user Saves (pushdown) the current definition of field f and Redefines (popup) field g with the last entry pushed into the pushdown stack. f and g are field names, A-Z. Entries are placed in a stack on a last-in-first-out basis.

Section 4.4 Block and Field Manipulation Operations

We begin at this point to discuss manipulation operations in the CSLx system. Our concern in this section is with the data-independent operations (we stretch the point a little when we deal with pointers) which we divide into two (2) classes:

1. Block operations
2. Field operations

Section 4.4.1 Block Manipulation Operations

The first two (2) BOU's we discuss are concerned with communication with one or the other of the CSLx storage allocator routines (L6STORAG or L7STORAG).

(a,GT,b)
(a,GT,b,c)

Blocks of storage are obtained from the storage allocator with this operation.

In the CSL6 system, b is either a positive integer denoting the number of words in the desired block or a data descriptor of a field where such an integer resides. b should be a power of 2 but if it is not, the next higher power of 2 will be assumed up to a maximum of 128 words.

In the CSL7 system, b is the same as in the CSL6 system except that values run from 1 to 32 and no assumptions are made. In either system, b ≤ ∅ causes an error return to the system (ILLSYS).

Upon completion of the call to the storage allocator, the pointer to the requested block is placed in field a. If c is present, the contents of field a prior to the storage allocator call are placed in field c. New blocks, when obtained from the storage allocator, are completely cleared to zeros.

Example: (A,GT,4)

When complete, "bug" A will hold the pointer to some 4-block which is initialized to all zeros.

Example: (A,GT,4,AB)

Assume field B is fifteen (15) bits long and also that "bug" A holds a pointer to block N. After the operation is complete, "bug" A will point to a new 4-block and field B of the new block will hold a pointer to block N.

(a,FR,Ø)
(a,FR,b)

Blocks of storage are "freed" or returned to the storage allocator by this BOU when they are no longer in use.

a is a field which points to the block of storage to be "freed." If b is present (not Ø), then when the block freeing operation is completed, the contents of field b are placed in field a.

Example: (A,FR,AB)

Assume "bug" A points to block M and field AB holds a pointer to block N. After completion of this operation, block M will be placed in some UBL in free storage and "bug" A will hold a pointer to block N.

The facility for duplicating blocks exists in the next BOU.

(a,DP,b)

Field b points to a block in storage. A new block of storage of the same number of words is obtained from the storage allocator and the contents of the first block are copied into the new block. A pointer to the new block is placed in field a.

Example: (A,DP,C)

Assume that "bug" C holds a pointer to some N-block M. After the operation is complete, a new N-block K will be present containing the exact same contents as block M and "bug" A will hold a pointer to block K.

In order to maximize the amount of information stored in a block, the user is allowed to access the size tag for a block.

(a,BS,b)

This operation allows the user to monitor the sizes of blocks that he is working with. b is a data descriptor of a field which holds a pointer to some block of storage. The BOU obtains the size of that block of storage and places it in field a.

Example: (A,BS,C)

Assume "bug" C holds a pointer to a K-block. After completion of the operation, "bug" A will hold the integer K.

Section 4.4.2 Field Manipulation Operations

We begin our discussions of field manipulation operations with the pointer copying BOU.

(a,P,b) This BOU causes the pointer contained in the field designated by b to be copied into the field designated by a. All fields which will contain pointers must be at least fifteen (15) bits wide.

Example: (A,P,AB)

Assume field AB to hold a pointer to block K. After completion of the operation, "bug" A will hold a pointer to block K. The field AB will be undisturbed.

We inherited the following shorthand notation for the pointer copying BOU from the original L^6 language.

(a,b) A special 2-element form exists to aid in scanning down strings. The 2-element form produces the same operation as if the second data descriptor were a concatenation of a and b.

Example: (A,B)

This BOU produces the same result as the previous example: (A,P,AB).

For copying of all other forms of field contents, the field copy BOU is used.

(a,E,b) b may be either a signed decimal integer or a data descriptor. The contents of field b are copied into the field designated by a.

Example: (A,E,-23)

After completion of the operation, "bug" A will contain -23_{10}.

(a,EO,c) — c may be either a signed octal literal or a data descriptor. The contents of field c are copied into field a.

Example: (B,EO,77)

After completion of the operation, "bug" B will contain 77_8.

(a,EH,d) — d is a string of up to 8 BCD characters, right justified, zeros left with spaces counted, which will be copied into field a.

Example: (H,EH,HOLLRITH)

After completion of the operation, "bug" H will contain the BCD string HOLLRITH.

(a,EF,e) — e may be either a floating point literal conforming to ILLAR language specifications or a data descriptor. The contents of field e will be copied into field a.

Example: (G,FF,22.3E10)

After completion of the operation, "bug" G will contain 22.3×10^{10}.

The CSLx system provides a BOU for exchanging the contents of two fields.

(a,IC,b) — The contents of the field designated by a are Inter-Changed with the contents of the field designated by b.

Example: (AC,IC,AB)

Assume field AC = 10_{10} and field AB = 24_{10}. After completion of the operation, field AC will contain 24_{10} and field AB will contain 10_{10}.

Incorporated into the CSLx auxiliary systems is a pushdown stack which will hold the contents of specified fields in the user's program. An example of such usage would be saving and restoring the contents of a "bug" during execution of a subroutine.

(S,FC,a)
(R,RC,b) — The user may Save (pushdown) the contents of field a or he may Restore (popup) the contents of field b.

Example: (S,FC,A)
(R,FC,B)

Assume "bug" A holds the number 62_{10}. The f. st BOU "pushes" the 62_{10} into the stack. The contents of "bug" A will be undisturbed.

The second BOU will "pop" the 62_{10} out of the stack and store it in "bug" B.

Two statements are provided to aid the CSL6 system programmer in providing multiple pushdown and popup operations on the system field contents stack. The format of the PUSHDOWN primary statement is:

```
PUSHDOWN,ABC,CD,1Ø,77b,-10.0
```

The elements of the statement are separated by commas "," and may be either data descriptors or literals (octal, decimal, or floating point, but not hollerith).

The format of the POPUP primary statement is:

```
POPUP,B,EF,GH,Z
```

The elements of the statement are also separated by commas"," but they may be data descriptors only. Note that the order in which field contents are "popped" out of the stack is the reverse of the order in which they were "pushed" into the stack.

```
Example:   PUSHDOWN,A,B,C,
           POPUP,C,B,A
```

After both statements are executed, "bugs" A, B, and C will contain their original contents.

The CSLx system also provides the facility for allowing the user to define and operate his own pushdown-popup data stacks. These operations will be discussed in Chapter 10.

Section 4.5 Special Debugging Aids - STATE and DUMP

Because the storage design of the CSLx system is so different from the standard memory array, two BOU elements have been provided which will dump required information about the status of the user's CSLx program.

(DO,STATE) The (DO,STATE) operation unit causes the following information to be output on the line printer.

1. Name of program and record number of "do" operation unit.
2. Time since execution of program began.
3. All current field definitions.
4. Contents of field contents pushdown stack.
5. Contents of subroutine calls pushdown stack.
6. Count of blocks in free storage by size.
7. Contents of all bugs.

(DO,DUMP) The (DO,DUMP) operation unit causes the following information to be output on the line printer:

1. All information provided by the (DO,STATE) operation unit.
2. Memory contents.
   a. Pointers of strings of free storage by block size.
   b. Contents of all occupied storage blocks in octal.

Neither dump will affect the interval clock.

Both options output a message to the console typewriter requesting the user to type a carriage return (CR) to allow the computer to continue execution. When control is returned from either BOU, execution will begin on the next executable statement or unit following the BOU.

Chapter 5. Logical Operations on Data

Section 5.1 Overview

Logical data operations fall into three classes:

1. Bit manipulation
2. Shifts
3. Count and position detection

The first class includes the complement operation (Section 5.2), OR (5.3), Exclusive OR (5.4), AND (5.5) and field substitution (5.6). The second class contains the left (5.7) and right (5.8) shifts. The third class contains the bit counting (5.9) and the bit-locating (5.10) operations.

Section 5.2 The Complement Operation

The bit complement BOU fetches the contents of a field or literal, complements by bit, and stores the result in a second field. Because of the ones-complement integer arithmetic and the biased exponent floating point arithmetic of the 1604 computer, the complement operation also may serve as the negation operation.

(a,C,b) — b may be either a signed octal integer or a data descriptor. The contents of field b are complemented on the way to being placed in field a.

Example: (ABC,C,53)

Suppose that field C is a 6-bit field. Then the octal integer 53_8 would be complemented to 24_8 and stored in VLF ABC.

(a,CD,b) — This form is the same as above except that b may be a signed decimal integer or a data descriptor.

Example: (/TIME,CD,460)

The decimal integer 460_{10} is negated to -460_{10} and stored in field /TIME

(a,CH,b) — b is interpreted to be a string of up to 8 BCD characters, right-justified, zeros left with spaces counted. All other considerations apply as with the preceding two forms.

Example: (*EXH,CH,J B)

The hollerith literal J B(412062_8) is complemented to ≡ ≤(365715_8) and stored in field *EXH.

(a,CF,b) — This form is the same as the first two except that b may be either a floating point literal conforming to the ILLAR language specifications or a data descriptor. All other considerations are the same as with the preceding three forms.

Example: (ABF,CF,-10.23)

10.23 will be stored in field ABF.

Section 5.3 Logical OR Operation

The logical OR data operation operates in a bit-wise manner according to the following truth table:

a	b	result
Ø	Ø	Ø
Ø	1	1
1	Ø	1
1	1	1

(a,O,b)
(a,O,b,c)

b may be either a signed octal integer or a data descriptor. The contents of field b are logically Ored with the contents of field a. The result is copied into field c if it is present. Otherwise, the result returns to field a.

Example: (ABE,O,4ØB)

Assume field ABE contains $32Ø_8$. After completion of the operation, field ABE will contain $36Ø_8$.

Example: (ABE,O,4ØB,C)

Assume field ABE = $32Ø_8$. After completion of the operation, "bug C" will contain $36Ø_8$. Field ABE will be unaffected.

(a,OH,b)
(a,OH,b,c)

In this format, b is interpreted to be a string of up to 8 BCD characters, right justified, zeros left with spaces counted. All other considerations are the same as for the preceding form.

Example: (D,OH, - 1 - - - - -)

Assume "bug" D holds the octal constant 2020002062464642_8. This is the hollerith literal - - ; - BOOK. After completion, "bug" D will contain - - 1 - BOOK.

Section 5.4 Exclusive OR Data Operation

The Exclusive OR operation handles data in a bit-wise manner according to the following truth table:

a	b	result
Ø	Ø	Ø
Ø	1	1
1	Ø	1
1	1	Ø

(a,X,b)
(a,X,b,c)

b may be either a signed octal integer or a data descriptor. The contents of field b are exclusively Ored with the contents of field a. The result is copied into field c if it is present. Otherwise, the result returns to field a.

Example: (ACE,X,17ØB)

Assume field ACE to contain $34Ø_8$. After the operation is complete, field ACE will contain $23Ø_8$.

(a,XH,b)
(a,XH,b,c)

In this format, b is interpreted to be a string of up to 8 BCD characters, right justified, zeros left with spaces counted. All other considerations are the same as for the preceding form.

Example: (/TEST,XH,FREE)

If field /TEST contains the Hollerith constant FREE, then after completion of the operation, field /TEST will be zero.

Section 5.5 The Logical AND Data Operation

The logical AND operation handles data in a bit-wise manner according to the following truth table:

a	b	result
Ø	Ø	Ø
Ø	1	Ø
1	Ø	Ø
1	1	1

(a,N,b)
(a,N,b,c)

b may be either a signed octal integer or a data descriptor. The contents of field b are logically aNded with the contents of field a. The result is copied into field c if it is present. Otherwise, the result returns to field a.

Example: (/RES,N,777B)

Assume field /RES holds 37477_8. After completion, field /RES will contain 477_8.

(a,NH,b)
(a,NH,b,c)

In this format, b is interpreted to be a string of up to 8 BCD characters, right justified, zeros left with spaces counted. All other considerations are the same as for the preceding form.

Example: (NAME,NH,TWO)

Assume field *NAME holds $Ø Ø77_8$. After completion, field *NAME will hold $23ØØ46_8$. (TWO = 232646_8).

Section 5.6 Logical Substitution Operation

The logical substitution operation operates upon data in a bit-wise manner according to the following truth table:

a	b	m	result
x	y	Ø	x
x	y	1	y

(a,U,b,m)
(a,U,b,m,c)

This operation unit allows selective substitution (insertion) of any portion of a field with another field.

a is a data description whose contents will be substituted for. m is either a signed octal integer or data descriptor which provides a mask through which the substitution will be made. Each 1-bit in the mask means that the corresponding bit in field a will be substituted for. m is right justified with zeros left.

b is either a signed octal integer or a data descriptor which provides the data to be substituted into a. If c is present, the new field contents after substitution will be placed in field c. Otherwise, the result will be returned to field a.

Example: (A,U,77B,CBA)

Assume "bug" A holds the hollerith literal FIELD= . Assume field CBA holds the BCD number 6. After completion of the substitution operation, "bug" A will contain FIELD= 6.

(a,UH,b,m)
(a,UH,b,m,c)

This form is also the same as the first form except that b is interpreted to be a string of up to 8 BCD characters right justified, zeros left with spaces counted.

Example: (A,UH,77b,6)

This example is the same as the one above except that the BCD character 6 is explicitly stated as a hollerith literal.

Section 5.7 Logical Left Shift Operation

The logical left shift operation allows information from one field to be shifted in the left direction into another field.

(a,L,b)

b may be either a positive decimal integer or a data descriptor. The content of field b is the number of bit positions which field a is shifted to the left. This 3-element form specifies that zeros are shifted in from the right. The result is placed back in field a.

Example: (A,L,2)

Assume "bug" A to hold the number 15_{10}. After completion of the shift, "bug" A will hold 60_{10}. If we express the numbers in octal, 17_8 becomes 74_8.

(a,L,b,c)
(a,L,b,c,d)

b again specifies where the shift count is found. c may be a signed octal integer or a data descriptor. The field or literal specified by c is positioned prior to the shifting operation such that the left edge of c is next to the right edge of field a. The result after shifting is placed in field d if it is present. Otherwise the result is returned to field a.

Example: (A,L,6,ACD)

Assume field ADC to be six (6) bits long. Assume also that field ADC holds the BCD character + and "bug" A holds the string ALPHA. After the shift, "bug" A will hold the string ALPHA+. Field ADC is undisturbed.

(a,LH,b,c)
(a,LH,b,c,d)

c is interpreted to be a string of up to 8 BCD characters, right-justified, zeros left with spaces counted. All other considerations are the same as the previous form.

Example: (A,LH,6,+)

This example produces the same result as the example above for the second case where "bug" A contains the string ALPHA.

Section 5.8 Logical Right Shift Operation

The logical right shift operation allows information from one field to be shifted in the right direction into another field.

(a,R,b)

b may be either a positive decimal integer or a data descriptor. The content of field b is the number of bit positions which field a is shifted to the right. This 3-element form specifies that zeros are shifted in from the left. The result is placed back in field a.

Example: (A,R,4)

Assume "bug" A holds the number 1024_{10}(2000_8). After the shift is completed, "bug" A will hold 64_{10}(100_8).

(a,R,b,c)
(a,R,b,c,d)

b again specifies where the shift count is found. c may be a signed octal integer or a data descriptor. The field or literal specified by c is positioned prior to the shifting operation such that the right edge of c is next to the left edge of field a. The field width of literals is assumed to be the same as field a. The result after shifting is placed in field d if it is present. Otherwise, the result is returned to field a.

Example: (AC,R,6,A)

Assume field AC is six (6) bits wide. Assume "bug" A holds the string ALPHA+. After the shift, field AC will contain the character +. "bug" A will not be disturbed.

(a,RH,b,c)
(a,RH,b,c,d)

c is interpreted to be a string of up to 8 BCD characters, right-justified, zeros left with spaces counted. All other considerations are the same as the previous form.

Exampel: (AC,RH,6,+)

This operation unit produces the same result as the example above.

Section 5.9 Bit Counting

(a,OS,b)

The field designated by b has its one bits counted and the count is placed in the field designated by a. If no bits of the type required are present, the count is set to zero (Ø).

Example: (A,OS,BC)

Assume field BC holds the octal number 1Ø3463$_8$. After completion of the bit count, "bug" A will hold 8$_{1Ø}$.

(a,ZS,b)

The field designated by b has its zero bits counted and the count is placed in the field designated by a. If no bits of the type required are present, the count is set to zero(Ø).

Example: (A,ZS,BC)

Assume field BC is eighteen (18) bits wide and contains the octal number 1Ø3463$_8$. After completion of bit counting, "bug" A will hold the count of 1Ø$_{1Ø}$.

Section 5.10 Bit Position Detection Operation

The bit-position detection operation units determine the position of the leftmost or rightmost zero or one bit in the field designated by b. Positions are counted as the i^{th} position in the field, not the word in which the field resides. Positions number from 1 up, left and right. If no bit of the type designated exists in the field, the position information is set to zero (0). When the operation is completed, the position count will be placed in the field designated by a. In the following examples, assume that field BC is twenty-four (24) bits wide and contains the number 14061375_8.

(a,LO,b)

This operation detects the position of the leftmost one bit in field b. The position count is placed in field a.

Example: (A,LO,BC)

When complete, "bug" A will contain 3_{10}.

(a,LZ,b)

This operation detects the position of the leftmost zero bit in field b. The position count is placed in field a.

Example: (A,LZ,BC)

When complete, "bug" will contain 1_{10}.

(a,RO,b)

This operation detects the position of the rightmost one bit in field b. The position count is left in field a.

Example: (A,RO,BC)

When complete, "bug" A will contain 1_{10}.

(a,RZ,b)

This operation detects the position of the rightmost zero bit in field b. The position count is left in field a.

Example: (A,RZ,BC)

When complete "bug" A will contain 2_{10}.

Chapter 6. Mathematical Operations

Section 6.1 Overview

The CSLx system provides the standard set of mathematical operations usually found in computer languages with the exception of exponentiation. They are:

1. Addition (6.2)
2. Subtraction (6.3)
3. Multiplication (6.4)
4. Division (6.5)

In addition, conversion from fixed-point to floating-point and vice versa is provided.[6.6] An absolute value function is provided for either type.[6.7]

The type of mathematical operation, fixed or floating-point, is stated by the postfix on the opcode. Floating-point operations always have the postfix letter F attached.

Because the 1604 computer word is forty-eight (48) bits long, arithmetic operations on VLF require that the field be expanded to forty-eight (48) bits. This is accomplished by extending the leftmost bit in the field to the left until forty-eight (48) bits are achieved. Thus, the leftmost bit in a field holding an arithmetic quantity is treated as the sign bit of the field.

Note that sign extension dictates that integers in fields live in the range (-2^{N-1}) to $(2^{N-1}-1)$ where the width of the field is N bits. This sign extension feature does not apply anywhere else in the CSLx system.

Section 6.2 Addition Operation

(a,A,b)
(a,A,b,c)

b may be either a signed decimal integer or a data descriptor. The contents of fields b and a are added as integers. The result is copied into field c if it is present. Otherwise, the result is returned to field a.

Example: (/SLOT,A,10)

Assume /SLOT = 20_{10}. After addition, /SLOT = 30_{10}.

(a,AO,b)
(a,AO,b,c)

The operations are the same as the above form except that b may be either a signed octal integer or a data descriptor.

Example: (/SLOT,AO 12)

Assume /SLOT = 24_8. After addition, /SLOT = 36_8.

(a,AF,b)
(a,AF,b,c)

b may be either a floating-point literal conforming to the ILLAR language specifications or a data descriptor. The contents of field b are added to field a in floating-point format. The result is copied into field c if it is present. Otherwise the result is returned to field a.

Example: (/SLOT,AF,10.0)

Assume /SLOT = 20.0. After floating-point addition, /SLOT = 30.0.

Section 6.3 Subtraction Operation

(a,S,b)
(a,S,b,c)

b may be either a signed decimal integer or a data descriptor. The contents of field b are subtracted from field a. The result is copied into field c if it is present. Otherwise, the result is returned to field a.

Example: (/SLOT,S,1Ø)

Assume /SLOT = $2Ø_{1Ø}$. After subtraction, /SLOT = $1Ø_{1Ø}$.

(a,SO,b)
(a,SO,b,c)

The operations are the same as the above form except that b may be either a signed octal integer or a data descriptor.

Example: (/SLOT,SO,12)

Assume /SLOT = 24_8. After subtraction, /SLOT = 12_8.

(a,SF,b)
(a,SF,b,c)

b may be either a floating-point literal conforming to the ILLAR language specifications or a data descriptor. The contents of field b are subtracted from field a in floating-point format. The result is copied into field c if it is present. Otherwise the result is returned to field a.

Example: (/SLOT,SF,1Ø.Ø)

Assume /SLOT = 2Ø.Ø. After floating-point subtraction, /SLOT = 1Ø.Ø.

Section 6.4 Multiplication Operation

(a,M,b)
(a,M,b,c)

b may be either a signed decimal integer or a data descriptor. The contents of fields b and a are multiplied as integers. The result is copied into field c if it is present. Otherwise, the result is returned to field a.

Example: (/SLOT,M,10)

Assume /SLOT = 20_{10}. After multiplication, /SLOT = 200_{10}.

(a,MO,b)
(a,MO,b,c)

The operations are the same as the above form except that b may be either a signed octal integer or a data descriptor.

Example: (/SLOT,MO,12)

Assume /SLOT = 24_8. After multiplication, /SLOT = 310_8.

(a,MF,b)
(a,MF,b,c)

b may be either a floating-point literal conforming to the ILLAR language specifications or a data descriptor. The contents of field b are multiplied with field a in floating-point format. The result is copied into field c if it is present. Otherwise the result is returned to field a.

Example: (/SLOT,MF,10.0)

Assume /SLOT = 20.0. After floating-point multiplication, /SLOT = 200.0.

Section 6.5 Division Operation

In all cases of divide operations, the CSLx system will compile a check for a divisor of zero. When an attempt to divide by zero occurs during the execution of the program, an error message will appear and control will be transferred to the operating system (ILLSYS).

(a,V,b)
(a,V,b,c)

b may be either a signed decimal integer or a data descriptor. The contents of field a are divided by field b as integers. The result is copied into field c if it is present. Otherwise, the result is returned to field a.

Example: (/SLOT,V,1Ø)

Assume /SLOT = $2Ø_{1Ø}$. After division, /SLOT = $2_{1Ø}$.

(a,VO,b)
(a,VO,b,c)

The operations are the same as the above form except that b may be either a signed octal integer or a data descriptor.

Example: (/SLOT,VO,12)

Assume /SLOT = 24_8. After division, /SLOT = 2.

(a,VF,b)
(a,VF,b,c)

b may be either a floating-point literal conforming to the ILLAR language specifications or a data descriptor. The contents of field a are divided by field b in floating-point format. The result is copied into field c if it is present. Otherwise the result is returned to field a.

Example: (/SLOT,VF,1Ø.Ø)

Assume /SLOT = 2Ø.Ø. After floating-point division, /SLOT = 2.Ø.

Section 6.6 Data Format Conversion

(a,FX,b)

b is a data descriptor of a field assumed to hold a floating-point format data word. The BOU converts the floating-point word to fixed-point format and places the result in the field designated by a.

Example: (AC,FX,B)

Assume field AC to be six (6) bits in length. Assume also that "bug" B contains the number 24.65. After completion of the operation, field AC will contain 24_{10}.

(a,FL,b)

This operation is complementary to the above form. The contents of field b are assumed to be in fixed-point format. The BOU converts the fixed-point word to floating-point format and places the result in the field designated by a.

Example: (AD,FL,AC)

Assume field AD to be forty-eight (48) bits in length and field AC to be eight (8) bits in length. Assume field AC to contain the number -17_{10}. After the operation is complete, field AD will contain the number -17.00.

Section 6.7 Absolute Value Function

(a,ABSV,b)

The absolute value of the contents of field b is placed in field a. a and b are both data descriptors. If field b is a VLF, sign extension will be performed before taking the absolute value.

Example: (A,ABSV,A)

Assume "bug" A to hold -24.6. After completion of the operation, "bug" A will hold +24.6.

Chapter 7. Subprograms, Subroutines and Functions

Section 7.1 Overview

In the ILLSYS system, calling sequences in the ILLAR and FORTRAN language systems obey what we will call the FORTRAN type calling sequence:

1. A return jump (1604 code) instruction is made to the entry point of the subroutine or function.
2. Only one call may be made to a given subroutine or function at a time.
3. Argument transfers are made by passing the address of the argument instead of the argument.

In the CSLx system, a new type of subroutine calling sequence called the DO type entry is provided:

1. A direct transfer is made to the entry point.
2. Calls to routines are recursive, that is, the return addresses are kept in a last-in-first-out pushdown stack.
3. Argument transfers follow the FORTRAN convention.
4. Two types of exit from the called routine are provided: standard and error exit.

The consequences of the first rule are that any statement or group of statements in a CSLx program may be treated as a subroutine. The second rule increases the flexibility of a subroutine by allowing it to call itself. Rule four provides for exits based on unusual conditions.

In this chapter, we discuss both the DO type calling sequence (Section 7.2) and the FORTRAN type calling sequence (7.3). A special form of the FORTRAN type calling sequence, the FUNCTION subroutine call is treated in Section 7.4.

Section 7.2 SUBPROGRAM Operations

The BOU used to drive DO type subroutines in the CSLx system is, of course, the DO BOU.

(DO,label)
(f,DO,label)

label is the name of the subroutine to be executed. This is a program label which may appear at any place in a CSLx program. The BOU causes an internal label pointer to the next BOU or statement after the DO BOU to be pushed down into the system subroutine call stack. This entry in the stack may be executed by a DONE "goto" as will be explained later.

If f is present, it is interpreted as a label to which a return from the subroutine may be made by a FAIL "goto" as will be explained later.

The action of the DO BOU after pushdown is to transfer control in the CSLx program to the "called" routine.

Either label or f may be treated as external to the CSLx program where the DO BOU is present by prefixing the label with an astrisk (*).

Example: (DO,COUNT)

After the proper return address is pushed down into the subroutine call stack, control will be transferred directly to the routine COUNT. No "fail" exit will be allowed from COUNT.

Example: (*CSLMCS,DO,DRIVE)

After the proper return address and the external "fail" label CSLMCS have been pushed down in the subroutine call stack, control will be transferred to the routine DRIVE.

This form of the DO BOU does not allow for argument transfers. A special case called the DOARG BOU is provided for this purpose.

(DOARG,label,list)

label and f are the same type of labels as described above for DO BOU's. The distinction is made by the use of DOARG instead of DO as the opcode.

The arguments are specified in the list. The list is made up of data descriptors or literals separated by commas "," and terminated by the ")" of the operation unit. No hollerith literals may be placed in the list.

Example: (DOARG,TIME,47B)

The routine TIME is driven with the argument 47_8.

Example: (ENDFILE,DOARG,READTAPE,32Ø32B,/BUFFER,1Ø)

The routine READTAPE is driven with the arguments $32Ø32_8$, BUFFER and $1Ø_{1Ø}$. The "fail" exit label ENDFILE is also provided.

Two system defined "goto" elements provide the means of return from DO type subroutines.

DONE

The encountering of a DONE "goto" causes essentially a subroutine type return transfer of program control. The transfer point is obtained by a popup of one element from the subroutine call stack. If no element exists, an error return will be made to ILLSYS.

A DONE "goto" terminates the statement in which it occurs. The "goto" also compiles an end to any input/output (I/O) operation area that may be in force at that point (Chapter 11). This I/O end operation is executed before the transfer of the "goto."

FAIL

The encountering of a FAIL "goto" causes an error return transfer from a subroutine. The transfer point is obtained by a popup of one element from the subroutine call stack. If no element exists, an error return is made to ILLSYS. An error message and return to ILLSYS will be made if no FAIL entry is found in the element popped from the stack.

Note that each element from the subroutine call stack may contain both DONE and FAIL transfer points.

A FAIL "goto" terminates the statement in which it occurs. The "goto" also compiles an end to any I/O operation area that may be in force at that point (Chapter 11). This I/O end operation will be executed before the transfer of the "goto."

Examples of usage will be made in Chapter 12 where we intend to give CSLx programming examples.

Facility for entering a program or subroutine at some entry point other than at the header card by use of a DO or DOARG BOU is provided by the DO ENTRY declarative statement. The statement format is as follows:

```
label     DO ENTRY
label     DO ENTRY,NOPREAMBLE
```

A DO ENTRY point may be declared in either a PROGRAM, SUBROUTINE or SUBPROGRAM at any point desired. The first form will cause parameter setting operations when entered if there are arguments specified in the header record. The second form will cause parameter setting operations to be ignored for that entry point.

The label attached to a DO ENTRY statement will be tagged as a global entry point which can be accessed from programs outside the program where the entry point is defined. The DO ENTRY point may only be accessed by either a DO or a DOARG BOU operation. Exit from the section of code headed by the DO ENTRY statement must be performed by either the DONE or FAIL "goto" operations. This requirement is also met by the END statement of a SUBPROGRAM program.

Section 7.3 Fortran type Subroutine Operations

The calling sequence for a FORTRAN type subroutine is specified by the CALL primary statement The format of the CALL primary statement is as follows:

CALL, NAME(list)

NAME is the name of the routine to be called. NAME is always an external program label (no "*" required).

The "," must be present to separate CALL from NAME. The "list" may or may not be present. The format of the "list" is simply a string of data descriptors, literals (no hollerith) or program labels. Two-way transfers of information via any one element of the "list" is possible for all element forms except field strings. The user must be responsible for not destroying literal arguments through return transfer usage.

If the list is present. it must be enclosed by "(" and ")". If only the "(" and ")" are present, the calling sequence will establish that the last "list" used in a CALL to routine NAME is used for this CALL.

We remind the user that only one type of return is allowed from FORTRAN type subroutine. Control will be returned to the next CSLx statement after the CALL statement.

Example: CALL,TIME

This is the simple form with no arguments. The routine TIME is executed and control returned to the next CSLx statement.

Example:

```
CALL,NAME1(A,ABC,/BC,*TIME,10,77b,10.4)
CALL,NAME1()
```

The first CALL to NAME1 also carries with it the arguments:

1. "bug" A
2. field ABC
3. Internal FWF BC
4. External FWF TIME
5. integer number 10
6. octal number 77
7. floating-point number 10.4

The second CALL to NAME1 causes the same arguments of the first CALL statement to be used as NAME1 is executed. This form executes a little faster as no argument address planting needs to be performed.

Examples of usage will be given in Chapter 12 where we intend to give CSLx programming examples.

Facility for entering a program or subroutine at some entry point other than at the header card by use of a FORTRAN type calling sequence is provided by the CALL ENTRY <u>declarative</u> statement. The statement format is as follows:

```
label    CALL  ENTRY
label    CALL  ENTRY,NOPREAMBLE
```

A CALL ENTRY point may be declared in either a PROGRAM or SUBROUTINE at any point. CALL ENTRY statements <u>may not</u> be used in SUBPROGRAM programs. The first form of the statement will cause parameter setting operations when entered if there are arguments specified in the header record. The second form will cause parameter setting operations to be ignored for that entry point.

The <u>label</u> attached to a CALL ENTRY statement will be tagged as a global entry point which can be accessed from programs outside the program where the entry point is defined. The CALL ENTRY point may <u>only</u> be accessed by a FORTRAN type calling sequence. Exit from the program entered at the CALL ENTRY statement <u>must</u> be through the END statement of the associated program or subroutine.

Section 7.4 Fortran type Functions

A special version of the FORTRAN type calling sequence routine exists and is called a FUNCTION routine. The calling sequence is the same as a FORTRAN type subroutine but the return of the result of execution is made by leaving the one (1) word result in the 1604 computer main arithmetic register.

The CSL system provides the FUNC BOU which allows the calling of a FUNCTION routine and placement of the execution result in some field for further processing by the CSLx program.

(a,FUNC,name,list) — The name of the FUNCTION routine is name and will always be defined as an external label (no "*" needed).

a is a data descriptor where the result of the FUNCTION will be placed upon completion of its operations. list is an arguments list constructed in the same manner as in the DOARG BOU. The arguments are determined by the FUNCTION routine's requirements.

Example: (A,FUNC,SQRT,4.0)

The SQRT of 4.0 is computed and returned to "bug" A upon completion of the operation.

Appendix B contains the necessary forms to allow usage of all the standard FORTRAN system functions.

Chapter 8. Control Transfer Operations

Section 8.1 Overview

We previously discussed the "goto" in Section 3.5 for use in effecting unconditional transfers of control between segments of CSLx programs.

Section 8.2 discusses the "assigned" TRANSFER primary statement and Section 8.3 discusses the "computed" TRANSFER primary statement. These two statements are analogous to the "assigned" and "computed" GO TO statements in the FORTRAN language system. Both statements provide dynamic control transfers during execution of a CSLx program.

Section 8.2 "Assigned" TRANSFER Operation

The format of the "assigned" TRANSFER primary statement is as follows:

TRANSFER (aa)

aa is the up to eight (8) BCD character statement label attached to a transfer "goto" variable. This label must not be used for any other purpose in the CSLx program where it occurs.

Since aa is in effect a special type of data word, we use a special primary statement to change the value of aa:

SWITCH, aa, bb

The SWITCH primary statement sets the contents of transfer "goto" variable aa to the statement label bb. When the TRANSFER (aa) statement is executed, program flow is transferred to the CSLx program statement labeled bb. An error return is made to ILLSYS if no assignment has been made to aa.

External program labels may be used provided they are prefixed by an asterisk (*) or declared as external FWF.

Example: SWITCH,ALPHA,ENDI
TRANSFER(ALPHA)

When execution of the TRANSFER statement occurs, control will be transferred to the statement labeled ENDI.

Section 8.3 "Computed" Transfer Operations

The "computed" TRANSFER primary statement achieves dynamic transfer control by sampling the value of some designated integer field. The general format is:

TRANSFER (ℓ1, ℓ2,...,ℓn) index

The list ℓ1, ℓ2,...,ℓn is made up of statement labels, each of which may be internal or external to the current CSLx program. External labels must either be declared as external FWF or be prefixed with an asterisk (*).

Index is a data descriptor for some field where an integer number in the range of $-\infty$ to N-1 where there are N labels in the list. If the contents of index are negative, the TRANSFER statement is not executed. Program execution continues at the next program statement. If the contents of index are $\geq$N, then an error will be declared and control transferred to the ILLSYS monitor CSLMCS. Otherwise, control will be transferred to statement ℓ_{index}.

Example: TRANSFER(UP,DOWN,OUT)I

If "bug" I = Ø, control transfers to the statement labeled UP. If "bug" I = 1, control goes to statement DOWN. No transfer occurs in "bug" I contains a negative number.

There is a short form of the "computed" TRANSFER statement that allows a binary choice of control transfer:

TRANSFER(label) index

If field index contains a positive number, control will go to statement labeled label.

Example:

```
          TRANSFER(S1)I
          TRANSFER(BETA)J
ALPHA     ....
          ....
          .
          .
          ....
S1        TRANSFER(ALPHA)J
BETA      ....
          ....
          .
          .
          ....
```

The above sequence of statements solves the following truth table:

I	J	Transfer to statement
-	-	ALPHA
-	+	BETA
+	-	BETA
+	+	ALPHA

Chapter 9. Relational Test Operations

Section 9.1 Overview

In this chapter, we will discuss the relational test operation units (TU) which are used in decision statements (Section 3.7). The first TU discussed is the pointer equality TU (Section 9.2). The second TU allows checking of block size (9.3).

Next to be discussed are four (4) mathematical relationship TU's:

1.	Equality	(Section 9.4)
2.	Inequality	(Section 9.5)
3.	Greater Than	(Section 9.6)
4.	Less than or equal	(Section 9.7)

The last two TU's are logical in nature and test for patterns of ones (9.8) or zeros (9.9).

The reader will note that the opcode fields of the TU may be the same as those of some BOU's. The distinction is made simply upon the condition that the TU must appear in a decision statement after the statement mnemonic.

Section 9.2 Pointer Equality Test

A special test unit (TU) is provided for checking equality between pointers. This lends itself to clarification of the language when being read and also protects against possible error due to the design of the 1604 computer (some pointers might not appear equal even though in fact they were).

(a,P,b)

a and b must be data descriptors. The fields contain pointers which are compared and if equal, the TU registers a "yes" vote. Otherwise, the TU says "no."

Example: (A,P,BC)

Assume "bug" A and field BC hold pointers. If the pointer in "bug" A points to the same block as the pointer in field BC, a "yes" vote will be recorded.

Section 9.3 Block Size Test

(a,BS,b)•.

a is a data descriptor of a field that contains a pointer. The size of the block which the pointer references is compared to the contents of field b and if equal, a "yes" vote is recorded. Otherwise, a "no" vote is taken by the TU.

b may be either a positive decimal integer or a data descriptor. Successful values of the contents of field b are powers of 2 (max 128) in CSL6 and 1 to 32 in CSL7.

Example:	(AT,BS,8)	(CSL6)
	(AT,BS,13)	(CSL7)

Assume field AT holds a pointer to 16-block K. Both TU's will register "no" votes.

Section 9.4 Data Equality Test

This TU compiles a vote on the mathematical equality of two (2) data items.

(a,E,b)

a is a data descriptor. b may be a signed decimal integer or a data descriptor. The contents of field a are compared to the contents of field b and if equal, the TU votes "yes." Otherwise, the TU votes "no."

Example: (/ACT,E,-22)

Assume field /ACT = -20_{10}. The TU will vote "no."

(a,EO,b)

b may be either a signed octal integer or a data descriptor. All other considerations are the same as the previous form.

Example: (/ACT,EO,-24)

Assume field /ACT = -20_{10}. The TU will vote "yes."

(a,EH,b)

b is interpreted to be a string of up to 8 BCD characters, right-justified, zeros left with spaces counted. All other considerations are the same as for the two previous forms:

Example: (A,EH,TIME)

Assume "bug" A holds the string CLOCK. The TU will vote "no".

(a,EF,b)

b may be either a floating point literal conforming to the ILLAR language specifications or a data descriptor. All other considerations are the same as for the three previous forms.

Example: (D,EF,26.145)

Assume "bug" D holds the number 26.1451. The TU will vote "no".

Section 9.5 Data Inequality Test

This TU compiles a vote on the mathematical inequality between two (2) data items.

(a,N,b)

a is a data descriptor. b may be a signed decimal integer or a data descriptor. The contents of field a are compared to the contents of field b and if not equal, the TU votes "yes." Otherwise, the TU votes "no."

Example: (/ACT,N,-22)

Assume field /ACT = -20_{10}. The TU will vote "yes."

(a,NO,b)

b may be either a signed octal integer or a data descriptor. All other considerations are the same as the previous form.

Example: (/ACT,NO,-24)

Assume field /ACT = -20_{10}. The TU will vote "no."

(a,NH,b)

b is interpreted to be a string of up to 8 BCD characters, right-justified, zeros left with spaces counted. All other considerations are the same as for the two previous forms.

Example: (A,NH,TIME)

Assume "bug" A holds the string CLOCK. The TU will vote "yes".

(a,NF.b)

b may be either a floating point literal conforming to the ILIAR language specifications or a data descriptor. All other considerations are the same as for the three previous forms.

Example: (D,NF,26.145)

Assume "bug" D holds the number 26.1451. The TU will vote "yes".

Section 9.6 Greater Than Test

This TU compiles a vote on whether one data item is mathematically greater than another data item.

(a,G,b)

a is a data descriptor. b may be a signed decimal integer or a data descriptor. The contents of field a are compared to the contents of field b and if a > b, the TU votes "yes." Otherwise, the TU votes "no."

Example: (/ACT,G,-22)

Assume field /ACT = $-2\emptyset_{1\emptyset}$. The TU will vote "yes".

(a,GO,b)

b may be either a signed octal integer or a data descriptor. All other considerations are the same as the previous form.

Example: (/ACT,GO,-24)

Assume field /ACT = $-2\emptyset_{1\emptyset}$. The TU will vote "no."

(a,GH,b)

b is interpreted to be a string of up to 8 BCD characters, right-justified, zeros left with spaces counted. All other considerations are the same as for the two previous forms.

Example: (A,GH,TIME)

Assume "bug" A holds the string CLOCK. The TU will vote "yes".

(a,GF,b)

b may be either a floating point literal conforming to the ILLAR language specifications or a data descriptor. All other considerations are the same as for the three previous forms.

Example: (D,GF,26.145)

Assume "bug" D holds the number 26.1451. The TU will vote "yes".

Section 9.7 Less Than or Equal Test

This TU compiles a vote on whether one data item is mathematically less than or equal to another data item.

(a,L,b)

a is a data descriptor. b may be a signed decimal integer or a data descriptor. The contents of field a are compared to the contents of field b and if a ≤ b, then TU votes "yes." Otherwise, the TU votes "no."

Example: (/ACT,L,-22)

Assume field /ACT = -20_{10}. The TU will vote "no."

(a,LO,b)

b may be either a signed octal integer or a data description. All other considerations are the same as the previous form.

Example: (/ACT,LO,-24)

Assume field /ACT = -20_{10}. The TU will vote "yes".

(a,LH,b)

b is interpreted to be a string of up to 8 BCD characters, right-justified, zeros left with spaces counted. All other considerations are the same as for the two previous forms.

Example: (A,LH,TIME)

Assume "bug" A holds the string CLOCK. The TU will vote "no".

(a,LF,b)

b may be either a floating point literal conforming to the ILLAR language specifications or a data descriptor. All other considerations are the same as for the three previous forms.

Example: (D,LF,26.145)

Assume "bug" D holds the number 23.1451. The TU will vote "no".

Section 9.8 Ones Pattern Test

This TU compiles a vote on whether the pattern of one-bits in one data item is included in another data item.

(a,O,b)

a is a data descriptor. b may be a signed octal integer or a data descriptor. A "yes" vote is registered by the TU if a has one bits in all of the positions that b has one bits. Otherwise, a "no" vote is recorded.

Example: (B,O,146)

Assume "bug" B holds the number 340146_8. The TU will vote "yes".

(a,OH,b)

b is interpreted to be a string of up to 8 BCD characters, right-justified, zeros left with spaces counted. All other considerations are the same as the previous form.

Example: (H,OH,ED)

Assume "bug" H holds the string TRIED. The TU will vote "yes."

Section 9.9 Zeros Pattern Test

This TU compiles a vote on whether the pattern of zero-bits in one data item is included in another data item.

(a,Z,b)

a is a data descriptor. b may be a signed octal integer or a data designation. A "yes" vote is registered by the TU if field a has zero bits in all of the positions that b has zero bits. Otherwise, a "no" vote is recorded.

Example: (K,Z,401)

Assume "bug" K holds the number 107301. The TU will vote "no".

(a,ZH,b)

b is interpreted to be a string of up to 8 BCD characters, right-justified, zeros left with spaces counted. All other considerations are the same as the previous form.

Example: (P,ZH,TRIED)

Assume "bug" P holds the string ED. The TU will vote "yes".

Chapter 1Ø. User Pushdown-Popup Data Stacks

Section 1Ø.1 Overview

The CSLx system provides automatically one (1) data pushdown-popup stack where data may be temporarily stored. Further capability for this type of operation is provided in the user defined stack system.

The user performs three operations concerning his own defined stacks:

1. definition by labelling (Section 1Ø.2)
2. pushdown operations (Section 1Ø.3)
3. popup operations (Section 1Ø.4)

A maximum of fifty (5Ø) user stacks may be defined.

The lengths of the stacks are bounded only by the limits of unused memory and the number of "free" blocks available from the storage allocator.

Section 10.2 Definition of a User Stack

The CSLx user "defines" a user stack by assigning a label as follows with the DEFSTACK primary statement:

DEFSTACK, stack1, stack2,..........., stackN

stack1, ... , stackN are BCD program labels of up to 8 characters by which the stacks will be referenced. All user stacks will be open-ended to the limit of available core storage. That is, as a stack needs to be extended, it will be by adding one more block of storage. As stacks are emptied, their "freed" sections (storage blocks) will be returned to the storage allocator for use elsewhere.

User stack "definition" is not global in nature. This dictates that the same "definition" for a given user stack must be given in every program of a program set where that user stack will be used. During execution of a program set, all "definitions" of a given user stack will refer to the exact same stack in memory.

Example: DEFSTACK, ALPHA, BETA

User stacks ALPHA and BETA may now be referenced.

Note: In order for proper initial setup of the user stack system to occur, at least one (1) stack must be defined in the PROGRAM of a program set.

Section 1Ø.3 Pushdown Operation on a User Stack

Data may be pushed down into a user stack by using the STACK primary statement:

STACK, stakname, list

stakname is a label previously attached to one of the user stacks. A compiler error will result if the stack has not been "defined". list is a list of data descriptors or literals (no hollerith) similar to the lists for the PUSHDOWN statement. The elements of the list specify fields which the user desires to pushdown in the indicated stack.

Example: STACK, BETA, 1, 77B, 1Ø.83, A, /TIME

The top five (5) items in user stack BETA will be, in order:

1. contents of field /TIME
2. contents of "bug" A
3. literal 1Ø.83
4. literal 77_8
5. literal 1

Section 1Ø.4 Popup Operations on a User Stack

Data is popped up out of a user stack by using the UNSTACK primary statement:

UNSTACK, stakname, emptyext, list

stakname is the name of a previously "defined" user stack from which the user desires to remove data (popup). If the stack has not been previously defined, a compiler error will result. If the stack is empty, control of the user program will be transferred to emptyext which must be a statement label. The user may find out how many elements of the list were filled prior to the emptyext by accessing the filled count in either external fields L6STKCT (CSL6) or L7STKCT (CSL7). list is a list of data descriptors where the user desires the data being popped up from the stack to be stored.

The emptyext transfer point and the filled count locations make the user stacks somewhat more flexible than the system supplied data stack. This advantage is offset by the fact that user stack operations are slower than system stack operations.

Example: UNSTACK, BETA, ERROR, A, B, C, D, E

Assume user stack BETA was loaded by the STACK statement in Section 1Ø.3. Then, when all operations are complete:

1. "bug" A holds the contents of field /TIME
2. "bug" B holds the former contents of "bug" A
3. "bug" C holds 1Ø.83
4. "bug" D holds 77_8
5. "bug" E holds 1

Chapter 11. Input/Output of SCD Information with Format Conversion

Section 11.1 Overview

The CSLx system provides statements for format controlled I/O operations only. All other forms of input/output may be used by appropriate CALL statements to the ILLSYS input/output routines.

The use of the I/O statements is broken down into three phases:

1. Initialization
2. Data fetch or storage
3. Termination

The three (3) phases all apply to either input or output.

The statements for initialization are described in Section 11.2 followed by the data fetch and store BOU's (11.3). Section 11.4 ends the discussion by describing the termination phase.

The CSLx user should familiarize himself with the FORTRAN language system FORMAT statement. The FORMAT statement for the CSLx system is identical and therefore, the user is directed to the FORTRAN manual for detailed information.

Section 11.2 Initialization of Input/Output Operations

If the reader is familiar with the FORTRAN language, he will remember that I/O occurs within single statement where all data items and the controlling format are tied together in a common specification. In the CSLx system, a great deal more flexibility is achieved by separately specifying format and data items.

Each and every FORMAT statement controls what we will call an <u>input/output area</u> <u>(I/O area)</u>. The <u>I/O area</u> begins with either an INPUT or an OUTPUT <u>primary statement</u> and ends when properly terminated (Section 11.4). Also associated with each <u>I/O area</u> is an input or output medium.

The formats of the INPUT and OUTPUT <u>primary statements</u> are as follows:

INPUT, Imedium, format, end#I
OUTPUT, Omedium, format end#O

All three arguments: <u>Imedium(Omedium)</u>, <u>format</u> and <u>end#I(end#O)</u> are <u>statement label</u> in form. <u>format</u> refers to the controlling FORMAT statement.

<u>Imedium (Omedium)</u> may represent one of two ways for specifying an input(output) medium. The first way is an explicit statement of the type of input(output) unit.

For input:

1. PAPER TAPE	F	paper tape reader
PT		(flexowriters)
2. TYPEWRITER	T	console typewriter
3. MAG TAPE x		x magnetic tape (BCD mode)
(x = 2, 3, ..., 8)		
4. TELETYPE	Y	paper tape reader
TTY		(teletype)

For output:

1. PAPER TAPE	F	paper tape punch
PT		(flexowriter)
2. TYPEWRITER	T	console typewriter
3. MAG TAPE x		x magnetic tape (BCD mode)
4. PRINTER	P	printer-format control
PRINTER Q	Q	no format control
PRINTER O	O	no line count
5. TELETYPE	Y	paper tape punch
TTY		(teletype)

The second way in which Imedium (Omedium) may be specified is as a data descriptor of a field which contains the single character logical unit code as indicated in the center column above. The code is right-justified with zeros left in the field. This second specification is assumed by the CSLx compiler if Imedium (Omedium) is not one of the above labels.

The end#I (end#O) parameter is used for termination of the I/O area and will be discussed in Section 11.4.

The CSLx programmer must remember that an input I/O area may not overlap an output I/O area. A compiler diagnostic will occur if this happens. Any errors occurring in a FORMAT statement will in all probability not be found until execution time.

Section 11.3 Data Fetch and Store In An I/O Operation

Data is transmitted to and from the I/O medium in units corresponding to the areas in memory where the data items were found or will be stored. Since the area of storage in the CSLx system is the field (or literal), we move data in or out in terms of the fields from which they were fetched or to where they will be stored.

Section 11.3.1 Data Storage During An Input Operation

A special BOU called the TAKE BOU is used during an input operation.

(TAKE, a)	a is only a data descriptor. One unit of data is taken from the input medium and stored in field a. The format of the data item is determined by the FORMAT statement controlling the I/O area where the BOU is found.

Inside the I/O area, almost any CSLx operations may be performed. The user must not attempt certain operations as follows:

1. No transfers into an I/O area except to the INPUT statement.
2. No transfers out of the I/O area without properly terminating I/O operations (see Section 11.4)

Let us present a short example of an input I/O area in a CSLx program.

```
READ      INPUT, PT, FORMIN, END1
          (TAKE,A) (TAKE,B)
END1      (C,E,Ø) (TAKE,D) ENDIO
FORMIN    FORMAT (I2, F7.4, R4)
```

Assume the following data record is read from the paper tape reader in Flexowriter Code:

1229.6873CSL6

After completion of the input operation, the "bugs" have the following contents:

"bug" A = $12_{1Ø}$
"bug" B = 29.6873
"bug" C = Ø
"bug" D = CSL6

Section 11.3.2 Fetching Data During An Output Operation

A special BOU called the FEED BOU is used during an output operation.

(FEED, a) — a may be either a data descriptor or a literal (no Hollerith). One unit of data is taken from field a and delivered to the output medium. The format of the data item is determined by the FORMAT statement controlling the I/O area where the BOU is found.

Inside the I/O area, any CSLx operations may be performed subject to the same restrictions as for the input I/O area.

We present here a short example of an output I/O area from a CSLx program.

```
WRITE     OUTPUT, PRINTER, FORMOUT, END2
          (FEED,A) (FEED,B)
END2      (FEED,C) ENDIO
FORMOUT   FORMAT (1X, I2, 2X, I1, 2X, F7.4)
```

Assuming that "bugs" A, B, and C were set up by the input I/O area in the example of Section 11.3.1, the following line will appear on the printer when the output is complete:

```
12  0  29.6873
```

Section 11.4 I/O Area Termination

The CSLx user has two (2) types of I/O area termination to be aware of: compiler and execution. Compiler termination of an I/O area deliniates the end of the CSLx source records to be read by the compiler and included in a specific I/O area. Execution termination must occur at all points where control will be transferred out of an I/O area.

Compiler termination of an I/O area occurs at the end of the CSLx statement labeled end#I (for input) or end#O (for output). end#I and end#O are the last arguments of the INPUT and OUTPUT primary statements.

Execution termination of an I/O area occurs when the ENDIO primary statement is encountered. The ENDIO statement consists only of the character string ENDIO. Additionally, the DONE, FAIL and EXIT system "goto" units will create execution termination operations just prior to control transfer.

For examples of the usage of the ENDIO statement, see both of the examples of Sections 11.3.1 and 11.3.2.

Chapter 12. Sample Programs

Section 12.1 Overview

We present in this chapter two (2) sample programs written in the CSL6 language which illustrate some of the basic operations performed in the CSLx system. Either program will run in the CSL7 system without any modifications.

Each program is presented first in its complete listing format as it appeared on the line printer followed by a discussion of how the program operates step-by-step. For reference, each line of the program is numbered sequentially and referred to as line 23 for example.

```
--CSL6
          PROGRAM SORTNUMS
*
*         SAMPLE PROGRAM TO FORM A LIST OF INTEGERS IN A STRING,
*              SORT THE STRING AND ELIMINATE DUPLICATES, AND THEN
*              LIST THE CONTENTS.
*
          (SS,4)
FIELDA    (Ø,DA,33,47)
FIELDB    (Ø,DB,9,23)
FIELDN    (1,DN,Ø,47)
*
*         INITIALIZE LIST WITH ZERO LIST ELEMENT
*
INIT      (A,GT,2)(L,P,A)
*
*         READ LOOP - WHEN NEGATIVE INTEGER APPEARS, SKIP TO INITIAL
*              PRINTLOOP
*
READLOOP  INPUT,PT,INFORM,ENDI
INFORM    FORMAT(I4)
ENDI      (TAKE,N) ENDIO
*
*         ADD NEW NUMBER TO LIST
*
ADDNUM    (A,GT,2,AB)(ABA,P,A)(AN,E,N)
          IF (N,L,-1) PRINT1
          READLOOP
*
*         PRINT INITIAL CONTENTS OF LIST IN ORDER RECEIVED
*
PRINT1    (A,P,LA)
PRINT11   OUTPUT,PRINTER,OUTFORM,ENDO
OUTFORM   FORMAT(1X,I4)
ENDO      (FEED,AN) ENDIO
          (A,A)
          IF (AA,N,Ø) PRINT11
*
*         SORT LIST - SORT ROUTINE TAKEN FROM ORIGINAL L6 WRITEUP
*
SORT      (A,P,LA)
SORT1     IF (AA,E,Ø)  PRINT2
SORT2     IF (AN,E,ABN) THEN (ABA,P,AF)(A,FR,AA)SORT1
```

```
         IF (AN,L,ABN) THEN (AN,IC,ABN)(A,B) SORT2
         (A,A) SORT1
*
*        PRINT LIST IN SORTED ORDER AND RETURN BLOCKS TO FREE STORAGE
*
PRINT2   (A,P,LA)(C,E,Ø)
END3     OUTPUT,PRINTER,OUTF2,END3
OUTF2    FORMAT(1H )
PRINT22  OUTPUT,PRINTER,OUTF1,END4
```



```
OUTF1    FORMAT(1X,6HENTRY,I4,3H = ,I4)
END4     (C,A,1)(FEED,C)(FEED,AN) ENDIO
         (A,A)
         IF (AA,N,Ø) PRINT22
         END
```

```
CSL6 OF     5/20/69              CSL6          COMPILATION DATE      5/26/69
                                                                              PAGE 1
     --END
     --ILLAR
              FINIS
   2
  81
 104
  32
 512
   6
   2
   1
  48

ENTRY        1 =     1
ENTRY        2 =     2
ENTRY        3 =     6
ENTRY        4 =    32
ENTRY        5 =    48
ENTRY        6 =    81
ENTRY        7 =   104
ENTRY        8 =   512
ELAPSED TIME   0   0   2.4
```

Section 12.2 Sample Program to Sort A String of Integers

Program SORTNUMS reads a file of records from paper tape, each record containing an integer number. Each integer is placed in a <u>block</u> in core, and <u>all blocks</u> are linked together on a string with both forward and backward <u>pointers</u>.

When input is completed, the <u>blocks</u> of the string are arranged in ascending order of their integer contents. The final result is listed along with the input data.

The output as shown for program SORTNUMS is exactly as it would appear on the 1604 line printer. Had there been any error messages during compilation, they would have occurred immediately following the record in error.

The program is begun on lines 1-6 where storage is set up with a maximum block size of 4. <u>Fields</u> A, B and N are defined on lines 7-9.

"Bug" L will be loaded with a <u>pointer</u> to the first <u>block</u> in the string containing the input data. Thus, in line 13, we get one <u>2-block</u>, point to it with "bug" A and set the same <u>pointer</u> in "bug" L.

Line 18 begins the input I/O area where the integer data string will be read in. The input medium is PT (flexcode paper tape) and the format is specified in statement INFORM (line 19). The I/O area will end on statement ENDI (line 20). Each time lines 18-20 are executed, one integer value is read in and placed in "bug" N. The end of I/O operations is signalled by ENDIO on line 20.

Each new entry to the string is processed in lines 24-26. First, a <u>2-block</u> is obtained and linked back to the last <u>block</u> on the string. Then, the last <u>block</u> is linked forward to the new <u>block</u>. Finally, the integer read in is placed in <u>field</u> N of the new <u>block</u> (line 24). If the integer read in is negative, this signals the end of the input data. Control will transfer to the initial print loop PRINT1 (line 25). Otherwise, we return for a new read operation (line 26).

Printout begins by setting "bug" A to point to the first <u>block</u> in the string which contains data. Note that the actual first <u>block</u> on the string is a dummy <u>block</u> used to initialize the string (line 30). Then we start an output I/O area for the PRINTER (line printer) controlled by format statement OUTFORM (lines 31-32). The end of the I/O area occurs on statement ENDO (line 33).

Each time through lines 31-33, one item of data, taken from <u>field</u> AN, is printed. "Bug" A is then advanced to the next <u>block</u> (line 34). At this point, we make a test on whether we have reached the end of the string or not.

Two tests could be made. We could test the next field AN in the string to see if it is negative. We could also test the forward pointer of the next block to see if it is zero (Ø). Remember that all blocks obtained from the storage allocator are zeroed in all fields. Thus, the last block on the string will have zero (Ø) in its A field (forward pointer).

We test the end of the string via the second test described above (line 35). If field AA is not zero (Ø), control returns to PRINT11 to print another value.

When the list is printed, sorting begins as the list pointer is initialized in "bug" A (line 39). Next comes the test for end of the string (line 4Ø) where control will advance to the second print routine PRINT2 when sorting is complete.

Line 41 determines if the integer in the current block (pointed to by "bug" A) is equal to the value in the previous block. If so, the current block is dropped from the string by linking the previous block to the next block on either side of the current block. Then the current block is returned to storage with "bug" A set to point to the next block on the string. Control then returns to the end test.

Line 42 now tests the relationship between the value in the current block and the value in the previous block. If the current value is less than or equal to the previous value, the values of the two blocks are interchanged. Then "bug" A is moved back to the previous block. This enables push back of smallest values before larger ones. Control then returns to the equality test SORT2. If no interchange is needed, "bug" A is moved down the string (line 43) and the end test performed again.

After sorting, the printer is spaced (line 48-49) and the list is output in the format (line 47, 5Ø-54):

ENTRY N = Value

When the printing is complete, the program ends and control returns to ILLSYS.

On page L-4 the actual listing of the output from SORTNUMS is shown. The time of execution was 2.4 seconds. The entire listing of compilation and execution is shown exactly as it would appear on the line printer.

```
--CSL6
          PROGRAM CHARCT
*
*               THIS ROUTINE READS BCD SOURCE RECORDS AND COUNTS
*         FREQUENCY OF CHARACTERS, SIZES OF WORDS AND COMPUTES
*         THE AVERAGE SIZE WORD.
*
*
          (SS,8)
          (A,E,Ø)
LOOP1     IF (A,N,64) THEN (/COUNT+A,E,Ø) (/SIZE+A,E,Ø) (A,A,1) LOOP1
          (/AVG,E,Ø) (/WORDCT,E,Ø)
*
*         INITIALIZE READ ROUTINE
*
          (ERROR1,DO,*INITREAD)
*
*         READ 1 CHARACTER AND COUNT
*
READ      (ENDREAD,DO,*READCHAR) (/COUNT+C,A,1)
          IF (C,NH, ) THEN (/WORDCT,A,1) READ
*
*         SPACE REQUIRES COUNT ON WORD SIZE
*
SPACES    IF (/WORDCT,G,3Ø) THEN (/WORDCT,E,3Ø)
          (/SIZE+/WORDCT,A,1) (/WORDCT,E,Ø)
          READ
*
*
*         PRINT RESULTS
*
*
*         OUTPUT CHARACTER FREQUENCY COUNTS
*
ENDREAD   OUTPUT,PRINTER,F1,ENDREAD
F1        FORMAT(16H1CHAR        CT  )
          ENDIO
          OUTPUT,PRINTER,F2,E1
F2        FORMAT(1H ,2X,R1,I1Ø)
          (A,E,1)
E1B       IF (A,EH, )E1A
          (FEED,A)(FEED,/COUNT+A)
E1A       (A,A,1)
          IF (A,L,6Ø) E1B
E1        ENDIO
```

```
*
*         OUTPUT COUNTS OF WORD SIZE
*
          (A,E,1)
E2        OUTPUT,PRINTER,F2,E2
F3        FORMAT(13HØSIZE     CT  )
          OUTPUT,PRINTER,F6,E3


CSL6 OF  5/2Ø/69                CHARCT        COMPILATION DATE        7/1/69                    PAGE 2
F6        FORMAT(2X,I2,4X,I5)
E3A       (FEED,A)(FEED,/SIZE+A)(A,A,1)
          IF (A,L,3Ø) E3A
E3        ENDIO
*
*         COMPUTE AVERAGE SIZE OF WORD
*
          (A,E,1)(C,E,Ø)
E4        (A,M,/SIZE+A,B)(/AVG,A,B)(C,A,/SIZE+A)(A,A,1)
          IF (A,L,3Ø) E4
          (A,FL,C)(/AVG,FL,/AVG)(/AVG,VF,A)
*
*         OUTPUT AVERAGE SIZE OF WORDS
*
          OUTPUT,PRINTER,F4,E5
F4        FORMAT(25HØAVERAGE SIZE OF WORD IS,F12.6)
E5        (FEED,AVG) ENDIO
          EXIT
*
*         ERROR MESSAGE FOR PREMATURE EOF
*
ERROR1    OUTPUT,PRINTER,ER1,ERROR1
ER1       FORMAT(15H1EOF READ FIRST)
          EXIT
*
*         STORAGE
*
          DEFINE,COUNT(64),SIZE(64),AVG,WORDCT
          END
```

```
 --END
 --CSL6
           SUBPROGRAM READCHAR
 *
 *         READ BCD SOURCE RECORDS AND DELTVER ONE CHARACTER PER
 *         CALL - FAILEXIT TAKEN IF END-OF-FILE IS READ FROM
 *         TAPE 4
 *         DRIVE *INITREAD* BEFORE FIRST CALL TO *READCHAR*
 *         CHARACTER WILL BE LEFT IN BUG C
 *
 *
REDO       (/INDEX,A,1)
           IF (/INDEX,L,8Ø) THEN (C,E,/LIST+/INDEX) DONE
           (FAIL2,DO,REREAD) REDO
 *
 *         READ NEW CARD AND BREAK DOWN INTO CHARACTER LIST
 *
REREAD     CALL,READBCD(4,/BUFFER,1Ø,77B)
           IF (/BUFFER,EH,--END    ) FAIL
           CALL,DCDBCDIN(/FORM1,/BUFFER,8Ø)
           (/INDEX,E,1)
REREAD1    CALL,WRDBCDIN(/LIST+/INDEX)
           (/INDFX,A,1)
           IF (/INDEX,L,8Ø) REREAD1
           (/INDEX,E,Ø) CALL,ENDBC'IN
           DONE
 *
 *         INITIAL ENTRY FOR SETUP
 *
INITREAD   DO ENTRY,NO PREAMBLE
           (FAIL2,DO,REREAD) EXIT
FAIL2      FAIL
 *
 *         STORAGE
 *
           DEFINE,LIST(8Ø),BUFFER(1Ø),INDEX
FORM1      FORMAT(8ØR1)
           END
```

```
CSL6 OF  5/2Ø/69                CSL6        COMPILATION DATE        7/1/69                          PAGE 1

    --END
    --ILLAR
            FINIS

    CHAR    CT
      1     35
      2      9
      3     11
      4      8
      5      5
      6      9
      7      Ø
      8      1
      9      1
      Ø     15
      =      Ø
      ≠      Ø
      ≤      Ø
      %      Ø
      [      Ø
      /     2Ø
      S     4Ø
      T     7Ø
      U     37
      V     12
      W     14
      X      5
      Y      2
      Z     15
      ]      Ø
      ,    1Ø3
      (     47
      →      Ø
      ¯      Ø
      ^      Ø
      -      2
      J      Ø
      K      Ø
      L     1Ø
      M     12
      N     4Ø
      O     7Ø
      P     24
      Q      3
```

R	9Ø
•	Ø
$	Ø
*	39
↑	Ø
↓	Ø
>	Ø
+	8
A	89
B	5
C	43
D	47
E	126
F	45
G	15
H	18
I	5Ø
<	Ø
.	2
(	47

SIZE	CT
1	41
2	28
3	14
4	23
5	16
6	15
7	18
8	6
9	6
1Ø	2
11	3
12	Ø
13	4
14	6
15	2
16	Ø
17	Ø
18	1
19	2
2Ø	4
21	1
22	1
23	1
24	Ø

(COLUMN CONT.)

25	2
26	Ø
27	Ø
28	Ø
29	1
3Ø	3

AVERAGE SIZE OF WORD IS 6.Ø95ØØØ
ELAPSED TIME Ø Ø 15.8

Section 12.3 Sample Program to Read BCD Records and Determine Frequency of Character Usage and Average Length of Word

Program CHARCT drives subroutine READCHAR to break down data on BCD source file records in order to find the frequency of character usage, determine the length of words on the records, and to calculate the average length of words in the file. This routine uses no linked storage but does contain pseudo-subscripting and extensive input-output operations.

The output as shown is exactly as it would appear on the 1604 line printer. Had there been any error messages during compilation, they would have occurred immediately following the record in error.

The operation begins in lines 7 - 1Ø. Storage is initialized because the system push-down stacks will require linked blocks even though they are not used in the program. Two arrays, COUNT and SIZE, are cleared to zero (Ø) counts. The initial values of average length AVG and total number of words counted WORDCT are also cleared to zero (Ø).

In line 14, an initial call is made to READCHAR to cause the first look-ahead read-in of a BCD source file record. The input file is always assumed to be on logical unit 4. We will leave the discussion of READCHAR until later in order to not interrupt the flow of the program listing.

The read-in of each character is performed on line 17. READCHAR produces one character in "bug" C. When the end of the input file is reached, an early exit is made to statement ENDREAD (line 33). Normally, the count for the input character is incremented by one (line 17). Additionally, the character is checked for being a space (2ØB). If it is not a space, WORDCT is incremented (line 18) to count the number of characters for the current word and the loop is repeated.

If a space character is read, then a word boundary has been reached (line 23). The size of word is limited to 3Ø characters maximum to prevent spill-over in memory (line 23). The appropriate word size counter is incremented (line 24) and the characters per word counter is reset to zero (Ø) (line 24). Control then returns to read a new character.

When the input file is exhausted, the character usage counts are listed. Output is initialized (lines 33-35) by labelling the printout. The count lines for the printout are started at line 36. An index for the printout is kept in "bug" A. Note that the printout for the character space (2ØB) is skipped on line 39. For all other characters up to $6Ø_{1Ø}$ (74B), first the index and then the count are fed to the output statement (line 4Ø). The index is checked, and if the end is reached, the output loop is left. Otherwise, the next count is processed (lines 41-43).

The second part of the printout is initialized with a heading (lines 47-49) and an index in "bug" A is set to one (1). Again, a feed output loop is set up in line 5Ø and the loop entered at line 52. The index of the word size count is output, followed by the total count for that size. When all counts have been printed (lines 52-54), the third part of the printout is prepared.

Line 58 initializes an index in "bug" A to one (1) and a summing register, "bug" C, to zero (Ø). The list of word sizes will now be summed for total size and total words. The loop starts in line 59 where the total count for the current index is multiplied by the current size and summed in AVG. The total count of words is summed in "bug" C. Then the index is advanced.

When the loop is complete, the average size or word is computed (line 61). Lines 65-67 output a statement and the calculated average. Control returns to the calling routine at this point (line 68).

A small section of code resides in lines 72-74 where an error message will result if the input data file contains no data. Note also in line 78 that the two arrays, SIZE and COUNT, and two variables, AVG and WORDCT, are explicitly defined.

The second part of the program is the SUBPROGRAM READCHAR which has two entry points. INITREAD provides an initializing step to read in the first record from the source data file and set up the character unpacking routines for operation. READCHAR causes one character to be read from the current source record. If the record is empty, a new record is read. When the end of the file is reached, the special "fail" exit is taken.

Character unpacking operations begin at REDO (line 9) where the current character index is incremented. If the index is less than 81, then the next character is taken from the sequential characters LIST (line 1Ø). Otherwise, a new record is called for (line 11) and control returned for the first character in that record (line 9). The "fail" exit will be taken by the REREAD section of code at the end of the file.

The next record from the input file is obtained beginning at REREAD (line 15). A 1Ø-word BCD record is read into BUFFER. If the first eight (8) characters of the next record are --END , then the end of the input file has been reached. Thus, the "fail" exit will be taken (line 16).

The breakdown of the input record is accomplished by use of the system DECODE routine which is initialized in line 17 to decode 8Ø characters in R1 format from BUFFER. 8Ø characters are planted in LIST in lines 18-21. The DECODE input operation is terminated and the current character index set to zero (Ø) in line 22. Control then returns to the calling section of code.

Lines 27-29 contain the initializing code. The REREAD section is called in order to set up the first record or detect an empty input file. The program ends with a definition of variable INDEX and the two arrays, BUFFER and LIST.

APPENDIX A

Error Messages

In this appendix we list the various error messages produced by the CSLx compiler. Each message is listed as it will be printed and may be followed by a clarification statement if necessary.

A.1 Statement Breakdown

ILLEGAL USE OF ,

ILLEGAL USE OF)

Generated by misplaced commas and right parentheses.

A.2 Field Designations

x FIRST CHAR NOT BUG

x is not A-Z for "bug"

a(oo) IS AN ILLEGAL FIELD CHARACTER

a is not 0-9, A-Z. oo is octal value.

FIELD a UNDEFINED

INTERNAL FIELD SHORT

Only "/" appears as field designation.

EXTERNAL FIELD SHORT

Only "*" appears as field designation.

A.3 Operation Block Processing

SINGLE OPERATION BLOCK

NO FIELD AFTER (

First field in block is missing. Also may mean space after "(".

ILLEGAL SEPARATOR

Only "," is legal separator.

MISSING 2nd FIELD

MISSING 3rd FIELD

MISSING 4th FIELD

MISSING 5th FIELD

Field missing between "," and ")"

OPERATION BLOCK TOO LONG

Block has more than 5 fields

INCOMPLETE STATEMENT

")" probably missing in last operation block of statement.

TEST BLOCK NOT 3 FIELDS

ILLEGAL FIELD OPERATOR aa(oooo)

Not allowed operation. aa is field operator and oooo is octal equivalent.

ILLEGAL OPERATION IN IF COMPUTATION aa(oooo)

Not allowed test operation. aa is operation code and oooo is the octal equivalent.

A.4 Unknown Data at End of Statements

UNKNOWN DATA AT END OF xxxxxxxx STATEMENT

UNKNOWN DATA AFTER xxxxxxxx

xxxxxxxx operations must be at end of the statement they occur in.

xxxxxxxx is one of

1. EXIT
2. FAIL
3. DONE
4. a "goto"
5. ENDIO
6. INPUT
7. OUTPUT
8. TRANSFER
9. SWITCH

A.5 IF and NOT Statements

ONLY ONE TEST ALLOWED IN IF STATEMENT

ONLY ONE TEST ALLOWED IN NOT STATEMENT

A.6 OUTPUT Statement

ILLEGAL FORMAT FOR OUTPUT STATEMENT

INCOMPLETE OUTPUT STATEMENT

ATTEMPTED TO START OUTPUT STATEMENT INSIDE INPUT STATEMENT AREA

ATTEMPTED TO START OUTPUT STATEMENT INSIDE OUTPUT STATEMENT AREA

A.7 INPUT Statement

ILLEGAL FORMAT FOR INPUT STATEMENT

INCOMPLETE INPUT STATEMENT

ATTEMPTED TO START INPUT STATEMENT INSIDE OUTPUT STATEMENT AREA

ATTEMPTED TO START INPUT STATEMENT INSIDE INPUT STATEMENT AREA

A.8 TRANSFER Statement

ILLEGAL TRANSFER STATEMENT FORMAT

INCOMPLETE TRANSFER STATEMENT

MISSING INDEX FIELD ON TRANSFER STATEMENT

ILLEGAL INDEX FIELD FORM

A.9 SWITCH Statement

ILLEGAL FORMAT FOR SWITCH STATEMENT

INCOMPLETE SWITCH STATEMENT

A.10 GLOBAL Statement

INCOMPLETE GLOBAL STATEMENT

"Bug" name missing after last ","

ILLEGAL SEPARATOR IN GLOBAL STATEMENT

Only "," is legal separator

ILLEGAL BUG NAME IN GLOBAL STATEMENT a

a is not a "bug" A-Z

NON SINGLE CHAR FIELD IN GLOBAL STATEMENT a

a contains more than 1 character

A.11 POPUP Statement

INCOMPLETE POPUP STATEMENT

Missing field after last ","

ILLEGAL SEPARATOR IN POPUP STATEMENT

Only "," is legal separator

ILLEGAL FIELD IN POPUP STATEMENT

Not a legal field designation or may be a literal

A.12 PUSHDOWN Statement

INCOMPLETE PUSHDOWN STATEMENT

Missing field after last ","

ILLEGAL SEPARATOR IN PUSHDOWN STATEMENT

Only "," is legal separator

ILLEGAL FIELD IN PUSHDOWN STATEMENT

Not a legal field designation

A.13 DEFINE Statement

INCOMPLETE DEFINE STATEMENT

Missing label after last ","

ILLEGAL SEPARATOR IN DEFINE STATEMENT

Only "," is legal separator

ILLEGAL LABEL IN DEFINE STATEMENT

Label does not conform to ILLAR label conventions

A.14 CALL Statement

INCOMPLETE CALL STATEMENT

Probable missing argument in call list and/or missing ")"

ILLEGAL SEPARATOR IN CALL STATEMENT

Comma must separate CALL from subroutine name

ILLEGAL FORMAT OF CALL OBJECT NAME

Subroutine name does not follow ILLAR program name convention.

A.15 ENTRY Statement

INCOMPLETE ENTRY STATEMENT

Missing label after last ","

ILLEGAL SEPARATOR IN ENTRY STATEMENT

Only "," is legal separator

ILLEGAL LABEL FORMAT IN ENTRY STATEMENT

Label does not conform to ILLAR label convention

A.16 DO ENTRY Statement

NO LABEL FOR ENTRY POINT

A.17 CALL ENTRY Statement

NO LABEL FOR ENTRY POINT

NO CALL ENTRY ALLOWED IN SUBPROGRAMS

A.18 EXTERNAL Statement

INCOMPLETE EXTERNAL STATEMENT

Missing label after last ","

ILLEGAL SEPARATOR IN EXTERNAL STATEMENT

Only "," is legal separator

ILLEGAL LABEL IN EXTERNAL STATEMENT

Label does not conform to ILLAR label convention

A.19 DEFSTACK Statement

INCOMPLETE DEFSTACK STATEMENT

Missing name after last ","

ILLEGAL SEPARATOR IN DEFSTACK STATEMENT

Only "," is legal separator

ILLEGAL NAME IN DEFSTACK STATEMENT

Names must conform to ILLAR label convention

STACK NAME x TOO LONG

Name x contains more than 8 characters

50 STACKS USED UP

Only 50 user stacks may be defined

STACK a IS DOUBLY DEFINED

A.2Ø STACK Statement

INCOMPLETE STACK STATEMENT

Premature end of statement after STACK or missing field designation after last ","

ILLEGAL SEPARATOR IN STACK STATEMENT

Only "," is legal separator

MISSING STACK NAME

Missing stack name or name does not conform to ILLAR label convention

STACK NAME TOO LONG

Stack name contains more than 8 characters

UNDEFINED USER STACK

ILLEGAL FIELD IN STACK LIST

Some field position contains an illegal field designation

A.21 UNSTACK Statement

INCOMPLETE UNSTACK STATEMENT

Statement ends prematurely after UNSTACK or missing field after last ","

ILLEGAL SEPARATOR IN UNSTACK STATEMENT

Only "," is legal separator

MISSING FAILEXIT IN UNSTACK STATEMENT

Missing FAILEXIT label or label does not conform to ILLAR label convention

FAILEXIT LABEL TOO LONG

FAILEXIT label contains more than 8 characters

UNDEFINED USER STACK

MISSING STACK NAME

Missing stack name or name does not conform to ILLAR label convention

STACK NAME TOO LONG

Stack name contains more than 8 characters

ILLEGAL FIELD IN STACK LIST

Some field position contains an illegal field designation

A.22 Hollerith Literals

HOLLERITH LITERAL OVER 8 CHARACTERS

A.23 Block Duplication Operation

ATTEMPTING TO DUPLICATE INTERNAL FIELD

ATTEMPTING TO DUPLICATE EXTERNAL FIELD

A.24 Field Contents and Field Definition Stack Operation

ILLEGAL FIELD DEFINITION OPERATION

First field in operation block is not S or R

ILLEGAL FIELD CONTENTS OPERATION

First field in operation block is not S or R

A.25 FEED Operation

FEED NOT ALLOWED IN IF COMPUTATION

FEED NOT PRIMED BY OUTPUT STATEMENT

A.26 TAKE Operation

TAKE NOT ALLOWED IN IF COMPUTATION

TAKE NOT PRECEDED BY INPUT STATEMENT

A.27 Storage Setup

IF STATEMENT STORAGE SETUP

ILLEGAL BLOCK SIZE ARGUMENT

Argument must be positive integer literal

A.28 Substitution Operation

SUBSTITUTION OPERATION HAS ONLY 3 FIELDS

A.29 Compilation of Argument Lists for CALL and DOARG Statements

INCOMPLETE LIST

Missing field after last ","

ILLEGAL FIELD IN LIST

Some field position contains an illegal field designation

ILLEGAL SEPARATOR IN LIST

Only "," is legal separator

ILLEGAL BUG CHAR IN LIST

Single character field is not A-Z or literal

A.30 Header Card

ARGUMENTS ERROR

Premature end on PROGRAM, SUBROUTINE, SUBPROGRAM cards with arguments before ")" found

A.31 FORMAT Statement

MISSING) FOR FORMAT

MISSING (FOR FORMAT

APPENDIX B

Proper Formats for Driving FORTRAN Language Function Subroutines

In order to maintain good compatability between language systems in the ILLAR system (ILLSYS), several special operation codes or statements have been included in each language to allow driving of function subroutines peculiar to the other system languages. In the CSLx language system, the FUNC BOU is provided to enable the use of FORTRAN language implicit function subroutines. The function subroutines are peculiar in that their result (always a single result) is returned to the calling program in the main accumulator of the 1604 computer. The FUNC BOU allows these subroutines to be called and then to place their result in some designated <u>field</u>.

Below we have listed proper forms of BOU's for driving most of the standard FORTRAN function subroutines. <u>Field A</u>, any type of <u>field designator</u>, is the <u>field</u> where the returned result will be placed. Arguments X, X1, and X2 may be any <u>field descriptor</u> or <u>literal</u> (no hollerith) as required by the function subroutine. For further descriptions and details about any particular subroutine, the reader is advised to see the ILLAR system librarian.

BOU Format	Operation of Function Subroutine
(A,FUNC,ABSF,X)	Absolute value of X in floating-point
(A,FUNC,INTF,X)	Truncation of integer part in floating-point
(A,FUNC,MODF,X1,X2)	X1 taken modulo X2 in floating-point
(A,FUNC,XMODF,X1,X2)	X1 taken modulo X2 in fixed-point
(A,FUNC,SINF,X)	Sine of X radians
(A,FUNC,COSF,X)	Cosine of X radians
(A,FUNC,TANF,X)	Tangent of X radians
(A,FUNC,ASINF,X)	Arcsine of X in radians
(A,FUNC,ACOSF,X)	Arccosine of X in radians
(A,FUNC,ATANF,X)	Arc tangent of X in radians
(A,FUNC,TANHF,X)	Hyperbolic tangent of X radians
(A,FUNC,SQRTF,X)	Square root of X in floating-point
(A,FUNC,LOGF,X)	Natural log of X in floating-point
(A,FUNC,EXPF,X)	e to the X^{th} power in floating-point
(A,FUNC,SIGNF,X1,X2)	Sign of X1 times X2 in floating-point
(A,FUNC,XSIGNF,X1,X2)	Sign of X1 times X2 in fixed-point
(A,FUNC,PWRRR,X1,X2)	$X1^{X2}$ in floating-point
(A,FUNC,PWRII,X1,X2)	$X1^{X2}$ in fixed-point
(A,FUNC,PWRRI,X1,X2)	$X1^{X2}$, X1 in floating-point, X2 in fixed-point
(A,FUNC,PWRIR,X1,X2)	$X1^{X2}$, X1 in fixed-point, X2 in floating-point
(A,FUNC,RANF,X)	Random number generator, X = +, then result is fixed-point; x = -, then result is floating-point

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author) University of Illinois, Coordinated Science Laboratory, Urbana, Illinois 61801	2a. REPORT SECURITY CLASSIFICATION
	2b. GROUP

3. REPORT TITLE

CSLx (x=6,7) A PROGRAMMER'S MANUAL TO THE USE AND UNDERSTANDING OF A LOW-LEVEL LINKED LIST STRUCTURE LANGUAGE

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

5. AUTHOR(S) (First name, middle initial, last name)

BO'.NIGHT, W. Jack

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
November, 1969	112	-

8a. CONTRACT OR GRANT NO. DAAB-07-67-C-0199	9a. ORIGINATOR'S REPORT NUMBER(S) R-446
b. PROJECT NO.	
c	9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)
d	

10. DISTRIBUTION STATEMENT

This document has been approved for public release and sale; its distribution is unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
	Joint Services Electronics Program thru U. S. Army Electronics Command Fort Monmouth, New Jersey 07703

13. ABSTRACT

This report is a programmer's manual for usage and understanding of a low-level linked list processing language operating on the Control Data 1604 computer. The basis for the language is L^6, a language developed at Bell Laboratories by K. C. Knowton.

The manual describes all of the operation codes, statements and procedures for using the language. In addition, a brief discussion is given on linked-list storage schemes and how they are handled.

DD FORM 1 NOV 65 1473

Security Classification

Security Classification

14 KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
List-processing						
Computer Language						
Programming						
Linked-list Storage Scheme						

Security Classification