NELIAC

Peter Robohm, LTJG USNR   •   2 November 1964   •   Primer and Programming Guide

U.S. Navy Electronics Laboratory • San Diego, California 92152 • A Bureau of Ships Laboratory

PREFACE

This Primer and Programming Guide has been extracted from the
documentation of the interim Command Ship Data System (CSDS) to
permit a wider distribution of the specifications for the NELIAC
language. The interim CSDS was a project completed under the
technical supervision of the Navy Electronics Laboratory for the
Bureau of Ships, Navy Department.

CONTENTS

Note

The prefix "B" in the pagination of this manual was required for another, separate issue of a User's Guide for NELIAC. The "B" has no significance in this edition and may be ignored.

ILLUSTRATIONS

TABLES

APPLICATION OF NELIAC

1. INTRODUCTION

INTENT

This part of the User's Guide is intended to serve as both an introduction and a programmer's working-guide to the computer compiling language known as NELIAC -- the Navy Electronics Laboratory International Algorithmic Compiler. The NELIAC vocabulary and phraseology required of a programmer for communication with a computer will be presented along with the background information necessary for a complete understanding of NELIAC applications.

It was the original intent for NELIAC, to function independently of any particular machine. Implicit in this design requirement was the understanding that a programmer would be able to compile any algorithm written in the NELIAC language on a large number of digital machines with only a very few concessions to individual machine characteristics. In the discussion which follows, reference to machines will be avoided, whenever possible, for it is hoped that this guide, while serving principally as a key to understanding the micro-programs of the Interim Command Ship Data System (as implemented on the AN/USQ-20) , will be universally applicable to the comprehension and composition of NELIAC programs.

Another purpose conceived for and incorporated into NELIAC was that it lend itself to the digital computer solution of scientific

problems. Thus the examples and problems presented herein will be of the type requiring algebraic or scientific expressions for their solution.

ORGANIZATION

The order of presenting the material in this manual might be considered somewhat unorthodox. However, an attempt has been made to group NELIAC elements of comparable characteristics in order to provide a continuity of ideas. The acorn-to-oak principle also has been utilized: from the most basic of the NELIAC elements -- the name -- grow the large, many-faced program devices of later chapters.

In addition to the textural presentation of each subject, two summary tables are included where appropriate. In the first table the reader will find a synopsis of the ideas just presented, in order to give the new programmer a concise review and the experienced hand a concise reference. The second table is devoted to a number of examples, some legal and some illegal (and why), which illustrate the material that has just been presented.

At the end of the NELIAC portion of this manual, in section 9, two "case study" problems are presented with full statements, solutions, and discussions. The second problem, a little more involved than the first, provides a ground on which to parade the more sophisticated elements of NELIAC.

BACKGROUND INFORMATION

The only prior knowledge in the field assumed of the reader is a basic understanding of computers and the concept of a stored program. Several good references covering these topics are available from manufacturers or libraries.

Engineers and scientists in first using the digital computer recognized its rapid and accurate handling of large, complex problems involving vast amounts of data. As early as 1956, digital computer manufacturers and other groups came to realize the need for languages which could be used with some facility after a minimum of programmer training. In other words, a language form other than machine code was required.

From this realization have come several "automatic programming" systems consisting of language-processor pairs. The systems are labeled "automatic" because the computer itself seems to be responsible for the programming task. This is not absolutely correct, as shall be seen.

These languages are characterized by their algebraic notation for mathematical solutions and by the use of English phrases for program control. They are often referred to as procedure- or problem-oriented languages (POL), a title which reflects their great usefulness in the solution of scientific problems.

The function of the processors is to translate the algebraic notation and English phrases into machine code for subsequent execution. A more detailed look at this translation process follows.

In the step numbered 1 in figure B1-1, the processor, in machine code, is read into computer storage. In the step numbered 2, the source program, a procedure written in the system language by the user to solve his problem, is presented to the processor residing in core. Since the source program is not written in machine code, but in a more sophisticated language, the processor must be capable of translating the one into the other. In so doing, the processor calls upon generators of machine language appliable to every source program syntactical form. The object program is a collection of machine instructions created by the generators from the source program. The collection process is known as compilation, and it is from this that the processor or translator has become a "compiler."
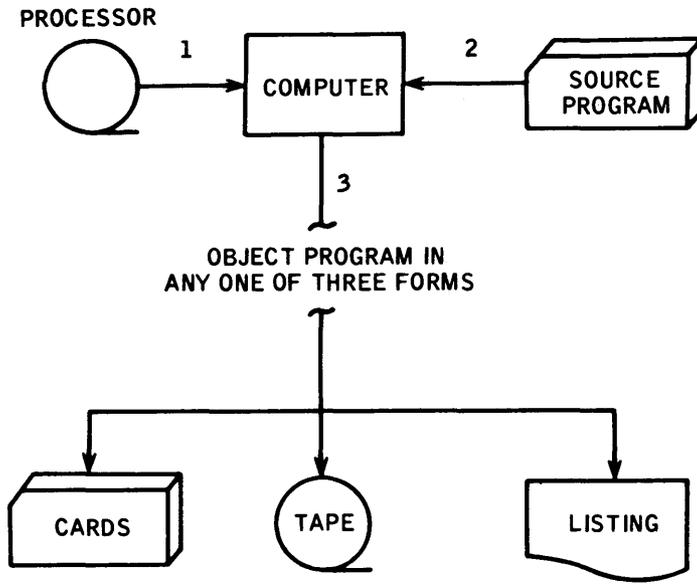
Figure B1-1. The compilation procedure.

As each source program statement is encountered, the compiler generates multiple object-program instructions in machine code. The object program is the assemblage of these instructions. Step 3 illustrates the fact that the object program may be returned to the programmer in several forms: cards, tape, or a hard-copy listing.

Execution, or the actual running of the problem on the computer, is a separate and distinct operation. Once the object program has been created, it may at any later time be read into core, data supplied to it, the computation prescribed undertaken, and the results presented to the problem originator.

This is the meaning of the term "automatic programming." Computer programming is automatic in the sense that, in presenting the processor with a source program in a problem or procedure oriented language, the creation of a program in an executable form (i.e., in machine code) is left to the compiler.

Thus, the automatic programming system is comprised of two parts: a language and a processor. The "language" part is of primary concern in the following sections of this manual.

NELIAC

The NELIAC language and compiler were conceived and developed by a group of U. S. Navy Officers and civilians at the U. S. Navy Electronics Laboratory in San Diego, California. Due to the fact that it is procedure-oriented and machine independent, its development has included an implementation on many different computers and a use in a large number of locations for varied scientific purposes.

NELIAC is a dialect of ALGOL 58, the Algorithmic Language established in 1958 by an international forum as a standard for expressing computer algorithms for scientific problems. One of the administrative problems of NELIAC has been standardization; in spite of machine independence, there are difficulties in implementing all facets of the expanding NELIAC on all machines. There is no provision for input/output in the original NELIAC, in line with the concept of ALGOL 58, to permit dissociation with any reference machine. In the compilers since developed for specific computers, routines have been devised for the machine-related functions and have been included in the separate NELIAC's.

To further classify it, NELIAC is a "one-pass self-compiler." "One-pass" means that there is no intermediary conversion of source program language to assembly language; translation of NELIAC is made directly to machine code. This eliminates multiple considerations of the program in core and consequently saves compilation time. "One-pass" also means that machine code is generated as each source program statement is encountered; there is no backtracking to pick up information.

The term "self-compiler" has much more significance. NELIAC is written in its own language, and as such is capable of

compiling itself. Consider figure B1-1 again. If the source pro-
gram is replaced by an updated NELIAC processor which is to be
compiled, the old processor is read into core, the new NELIAC is
presented to it for compilation, and the output or object program
is a revised NELIAC compiler.

There are interesting ramifications to this unique capability.
First, the hard-copy listing of the source program is never out of
date since changes are made directly to the system at the NELIAC
language level, and the entire compiler is recompiled to incorpo-
rate the changes. Secondly, the implementation of NELIAC on
other computers is more readily achieved because of the higher
level language which describes the system processor; routines for
compilation of features that are machine independent are already
written; only the machine dependent features, such as input/out-
put, need to be programmed in machine code.

A misinterpretation that should be clarified is the statement
that NELIAC is self-documenting. The basis for saying that
NELIAC is self-documenting is that the NELIAC processor is
written in the NELIAC language and that the listing of such a pro-
cessor provides an instruction-by- instruction documentation for
the compiler. The area of fallacy lies in the argument that such
a listing is totally self-documenting. Unless the programmer goes
to great pains to write comments within the body of the program,
there should be accompanying definitions for the terms used as an
explanation for the directions of program flow in order that any
programmer may understand the intentions of the original author.
Such measures prevent the necessity of guesswork to understand
flow and term meanings. Hence, it is strongly advocated that any
program written should be accompanied by appropriate documen-
tation. Later examples will illustrate the need for such descrip-
tive efforts.

2. THE NELIAC PROGRAM

APPROACH

As often as not, a man sitting down to a problem which he must solve by himself will use an intuitive approach in which the steps to solution are arrived at subconsciously and without calculation. However, when that man intends to utilize a digital computer for problem solution, he is forced to examine, plan, and delimit his approach very precisely.

Initially, he must understand the problem sufficiently to state accurately the given conditions and the unknowns to be found. If he cannot do this, no amount of data manipulation will provide the needed answers.

As a second step, he must describe the problem mathematically, reducing the original definition to formulae or equations, thereby isolating the pertinent information. If his problem is not reducible to simple algebra, he must recognize the applicability of some numerical analysis scheme.

The third step requires an understanding of digital computers and thus the services of a programmer who must propose an algorithm or procedure for a computer solution of the problem. This will probably involve a detailed picture or block diagram of the algorithm as well as the instruction-by-instruction specification of the computer program. For the computer to accomplish

the proposed calculations, the programmer might specify the algorithm in machine code, a language comprised of numbers formatted into instructions. To do this, he must know the machine in detail, a knowledge acquired through special training in the subject. His final act of preparation for execution is to assign memory locations to data storage, an elaborate task.

After loading the program successfully, the stage is set for testing the program. Data are applied to the instructions, and the programmer soon knows if his routine is correctly written. If not, he must go back, diagnose the troubles, and repair the errors. When he is successful, the results are returned to the problem's originator. This is the fourth step.

The last step is an evaluation of the answers by the originator; if he is satisfied that the program achieved the goal, the algorithm and program should be documented. If he isn't satisfied, the programmer initiates step four again.

The reader should understand that since the computer cannot exercise common sense (should the programmer incorrectly specify the data or instruct the computer badly), the chances for success on the first few tries are slight indeed. Consequently, the programmer frequently repeats the entire approach several times.

POL

If the programmer uses a procedure-oriented language (POL) instead of machine code while going through his approach, he takes advantage of a number of features which simplify his task.

The first advantage of a POL is that it reduces the requirement for a detailed knowledge of digital computers and frees the programmer to concentrate on other steps in the problem approach. In using the POL, the algorithm may be written in a language akin to English. Little training is required to become versed in such a language; in fact, when so desired, the scientist and the programmer may be the same person.

Secondly, the compiler considers names and operations as written by the programmer. It does all the assignment of addresses to data and instructions and translates the algebraic expressions and control statements to machine code.

Thirdly, the language allows comparison between quantities to decide logic flow; when commanded to do so, it compiles the statements necessary to create control for multiple executions of a series of instructions; and it incorporates such facilities as sub-routines and functions which allow the programmer to specify an algorithm once and call on (jump to) it from different portions of his program.

The fourth advantage is that program changes are more simply made when a routine is in error. The language is easier to read than machine code; when errors are determined, they are readily located and the necessary corrections can be quickly incorporated in the program with a compilation.

PROGRAM STRUCTURE

An algorithm written for the solution of a problem using the NELIAC language is called a program. When a program reaches such proportions that its size does not permit the processor to compile all of it simultaneously, the programmer must break the program logically into smaller parts called segments or flowcharts. For large programs it is usually desirable to separate programs logically into smaller parts for ease of error-checking. The NELIAC processor has the capability of individually compiling these flowcharts and then considering the whole program for grammatical errors. The flowchart concept also is useful when a number of programmers are working on separate segments of a much larger system, in that grammar checks may be made on individual flow charts before assembling the over-all program. For the programmer familiar with other compiling systems, the term "flowchart" must not be confused with the block diagram or flow diagram concept used to pictorially outline program logic flow.

FLOWCHARTS

In constructing a program, five primary flowchart types are available for use by the programmer at his discretion; they are: (a) the flowchart which incorporates data storage allocation and program logic, called the "process" flowchart; (b) the "declarative" flowchart, which provides for the introduction of machine dependent routines through a unique definition and call arrangement; (c) the "executive" flowchart, an indicator to the program entrance; (d) the "correction" flowchart, the segment intended to replace, before compiling, a flowchart presently a part of the program; and (e) the "edited" flowchart, the output of a formatting run by the compiler on any of the other four flowchart types.

Process Flowchart

The process flowchart (figure B2-1), introduced to the compiler with a control number of 5, is made up of two parts: the dimensioning and the program logic. The fact that either part or both parts are included in the flowchart is dictated by need and not by language rules.

The dimensioning part might better be called the "definition," since with dimensioning the programmer informs the compiler that areas are to be reserved by name for specific data, as well as defining the mode (indicating whether the data are to be integral or decimal fractions), the predetermined numerical values (if any), and the type of computer word storage to be used (whether full or partial word). Output formats and messages are also defined in dimensioning.

The program logic is the portion of the flowchart or program in which is found the algebraic approach, statement by statement, to the problem's numerical solution. Program control, directing the computer in a path of execution, subroutines, and functions to do special repeated programmed tasks, and other features that will be explained later, are also to be found in the program logic.

```
5 (COMMENT ' ' PROCESS FLOWCHART)_____ CONTROL NUMBER
A, BOX1 = 4, CZJT,_____ DIMENSIONING
DIAGONAL (4) = 1, 2, 3, 4_____ DIMENSIONING
$_____ END OF DIMENSIONING
TITLE ' '_____ PROGRAM LOGIC
ZONE CLEAR,_____ PROGRAM LOGIC
ZONE CLEAR ' ' BEGIN_____ PROGRAM LOGIC
    A + BOX1 =) CJZT,_____ PROGRAM LOGIC
    ($ CLEAR CELLS LS CJZT GR ,_____ PROGRAM LOGIC
    LS DIAGONAL GR , $), END_____ PROGRAM LOGIC
.. _____ END OF FLOWCHART
```

Figure B2-1. Process flowchart, example.

Declaration Flowchart

The declaration flowchart (figure B2-2), indicated by a 6
(control number), enables the programmer to assign a NELIAC
name to an absolute (machine) address, to define input/output
functions, and to define desired machine code procedures. The
purpose of the procedure written in machine code is to avoid the
necessity of programming a required routine several times in
NELIAC; a hand-coded procedure is often more efficient as well.

```
6 (COMMENT ' ' DECLARATION FLOWCHART)_____ CONTROL NUMBER
CLEAR CELLS = LS MACHINE (70100 OCT 0K) GR _____ DEFINITION
            LS MACHINE (16030 OCT 0K) GR _____ DEFINITION
.. _____ END OF FLOWCHART
```

Figure B2-2. Declaration flowchart, example.

Executive Flowchart

Through the use of the executive flowchart (figure B2-3), indicated by control number 9, the programmer provides an entrance (indicates the first instruction to be executed during actual operation) to his program. Another reason for using this type of segment is in the field of program debugging (error resolving). With such a tool the programmer can force a flowchart execution sequence and cause diagnostic checks to be made without altering his process flowchart.

The executive flowchart has all the dimensioning and program logic capabilities of the process flowchart. In the absence of such a flowchart, the first executable process flowchart becomes the entrance.

```
9 (COMMENT ' ' EXECUTIVE FLOWCHART)_____ CONTROL NUMBER
$_____ END OF DIMENSIONING
ZONE CLEAR, ($ CALL NELOS LS , $)_____ PROGRAM LOGIC
.._____ END OF FLOWCHART
```

Figure B2-3. Executive flowchart, example.

Correction Flowchart

A correction flowchart (figure B2-4), having a control number of 8, serves to modify an existing flowchart by deleting the old and substituting the entire corrected flowchart specified as the next flowchart.

```
8 (COMMENT ' ' CORRECTION FLOWCHART)_____ CONTROL NUMBER
15_____ NUMBER OF FLOWCHART TO BE REPLACED
- - - - -_____ FLOWCHART SEPARATOR
5 (COMMENT ' 'CORRECTED PROCESS FLOWCHART #15)_ CONTROL NUMBER
$_____ END OF DIMENSIONING
.._____ END OF FLOWCHART
```

Figure B2-4. Correction flowchart, example.

Edited Flowchart

An edited flowchart (figure B2-5), which is requested by a programmer (i.e., is not automatic), is a formatted, somewhat diagnostically checked output flowchart which will come out in any of three forms depending upon the portion of the NELIAC processor called: printed copy, cards, or magnetic tape. The primary use of the edited flowchart is in helping the programmer find grammatical errors through use of the formatted (to conform with logic flow intentions) output to search for errors in procedure specification. An edited flowchart output from any of the other four flowchart types may be requested. The edited flowchart, still at the NELIAC language level, must nonetheless be compiled before execution.

```
Ø (COMMENT ' ' EDIT PROCESS FLOWCHART) _____ CONTROL NUMBER
A (1ØØ), X _____ DIMENSIONING
$ _____ END OF DIMENSIONING
1 =) J, 19 =) X, _____ PROGRAM LOGIC
REPEAT ' ' _____ PROGRAM LOGIC
A ($ J $) + X =) A ($ J+5Ø $), J+1 =) J, _____ PROGRAM LOGIC
J = 12 ' ' STOP.  REPEAT. _____ PROGRAM LOGIC
STOP ' '_____ PROGRAM LOGIC
. . _____ END OF FLOWCHART
```

Figure B2-5. Edit flowchart, example.

3. NAMES

GRAMMAR OF NAMES

Neliac Symbol Set

There are 62 characters in the NELIAC symbol set, including all the numbers 0 through 9, the entire alphabet, and some 26 special characters. Letters and numbers together are used in the construction of names, numbers alone are used for data, and the special characters serve primarily as punctuation and operation symbols.

No differentiation is made between upper and lower case letters by the NELIAC compiler, but the programmer may use both cases interchangeably for greater readability. In other words, the NELIAC names, "EXCHANGE," "exchange," and "ExcHanGE" are equivalent. This text for its examples, uses all capital letters since in a card system no differentiation between upper and lower cases is possible.

The early versions of NELIAC were built with the Friden Flexowriter as an input/output device. Some of the punctuation characters of the symbol set are not available on card keypunches and high speed printers and, consequently, concessions have been made in the card NELIAC system to these character restrictions. The conversions necessary are shown in figure B3-1.

FLEX	CARD	FLEX	CARD	FLEX	CARD	FLEX	CARD
;	$	}	END	=	EQ or =	≥	GQ
:	"	x	*	≠	NQ	∩	AND
[	($	→	=)	<	LS	∪	OR
]	$)	↑	**	>	GR	8	OCT
{	BEGIN	|	'	≤	LQ	, . ( ) + - / INTERCHANGEABLE	

SYMBOLS

SYSTEMS

Figure B3-1. NELIAC character set conversion.

Name Formation

A NELIAC name is any combination of letters and numbers so long as the first character is a letter. Blanks imbedded in the name are ignored by the NELIAC processor, enabling the programmer to make his long names meaningful by spacing. The first fifteen nonblank characters constitute a name; characters beyond those fifteen are not compiled, but a programmer may still refer to the long name elsewhere in his program.

Index Register Variables

The first exception to these rules of formation are the index register variables, which are identified by the single letters I, J, K, L, M, N. (This does not prevent the use of names starting with one of these letters.) These six variables are not available for use as NELIAC names, but are employed as counters within the program logic. Each letter is associated with the machine index registers B1 through B6, respectively; use of an index register for counting generates a more efficient object program than if other NELIAC names are utilized for the same purpose. An index register is only half a word long (AN/USQ-20: 15 bits). The purpose of the index register variables will become more apparent later in the text. Other single letters, however, are valid NELIAC names.

Operators and Comparison Symbols

The only other exception to the general rule is the use of names which are identical to the card-NELIAC-language operators and comparison symbols. These include BEGIN, END, EQ, NQ, LS, GR, LQ, GQ, AND, OR, and OCT. Wherever used, a space (or blank card column) must precede and follow the operator or symbol. The programmer must not use these letter combinations for purposes other than those for which they were intended; otherwise, he may anticipate unexpected compilation results.

Following are a synopsis of the formation of names (table B3-1) and a list of examples of names (table B3-2).

TABLE B3-1. NAME FORMATION CONSTITUENTS

a. A combination of letters and numbers
b. First character must be alphabetic
c. Imbedded blanks have no significance
d. Maximum significant length = 15 characters; longer name possible
e. Single letters; except I, J, K, L, M, N, may be used; exceptions reserved for register variables
f. Operator and comparison symbols may not be used

TABLE B3-2. EXAMPLES OF NAMES

a.	A,	legal; single letter
b.	BOX 1,	legal; letters, imbedded blank, number
c.	NONSENSICAL PHRASE	legal; only first 15 considered
d.	cummings	legal; lower case permissible
e.	1C74	illegal; starts with number
f.	C2$41	illegal; contains symbol
g.	I	illegal; index register variable
h.	BEGIN	illegal; operator
i.	LS	illegal; comparison symbol

The Purposes of a Name

A name serves as an identifier or tag within the bounds of the algorithm. When the programmer uses a name in his program, it is linked to an address or a series of addresses in computer storage by the processor during compilation. So far as the programmer is concerned, however, this area in core continues to be addressed mnemonically.

If a name is defined in the dimensioning portion of a flow-chart, it is called a noun. Nouns provide a tag for storage which will contain variable (ever-changing) information or data.

Names assigned to routines in program logic are known as verbs, since they are associated with the action portion of the flow-chart. If a programmer wishes to jump from some point in his program to a particular routine, he indicates by specification of a verb and punctuation that the jump to a particular location is intended.

Meaningful Names

The programmer is urged to use NELIAC names which represent the literal meaning of his data or logic intentions. For example, in handling data concerning rocket fuel, the name ROCKET FUEL would be a sensible choice. Or, a routine to clear an area of computer core to zeros might be identified more easily by calling it ZONE CLEAR.

An effort in this direction pays rich rewards. The amount of documentation for a well-designed algorithm is greatly reduced by an appropriate choice of names.

NAME PRECEDENCE

The "name list" is a table maintained by the NELIAC processor as an inventory of names contained in a NELIAC program. As new names are encountered, they are added to the list.

There are three distinct times when the processor deletes or purges names from the name list: at the completion of a function or subroutine definition, at the end of a flowchart, and after the last flowchart or upon program completion. The timing of such purges implies that some names can be more localized than others, which is the case. When it is considered that the size and importance of subroutines, flowcharts, and programs increase in that order, this may be more clear.

The purges prevent the name list from becoming too long. As a result, additional subroutines, functions, and flowcharts can be processed in any given compiling run. More names can be temporarily stored, assigned addresses, and purged from the name list before list overflow than would be the case if all entries were additive and never subtractive. Another outcome of this differentiation or precedence among names is that a purged name list is a shorter table to search during compilation. This represents a sizable savings in processing time, since a search of the list is usually made several times for each NELIAC statement compiled.

All NELIAC names are divided into three classes: global names, local names, and function or subroutine names. The global class consists of those names which mnemonically tag addresses referred to throughout the program. The local names are peculiar to only one flowchart of the program. The subroutine or function names are used only in the definition and call of these specialized routines. Further discussion of these classes follows.

Global Name

A global name may be referenced from any flowchart in the program. It is sometimes designated as a "permanent" name because throughout the compiling run the name is never purged from the processor's name list. The programmer must be careful to avoid defining the name for more than one purpose as this sort of error results in an unsuccessful compilation.

Local Name

The local class of name is distinguished by its unique formation: one character of the name within the first 15 nonblank characters is an absolute sign (flex system) or apostrophe (card system); L'OCAL NAME is an example. This is the exception to NELIAC rules concerning the inclusion of symbols other than letters and numbers in name formation. When defining local nouns, the apostrophe is necessary only in the dimensioning portion of the flowchart. For local verbs, their first occurrence in the program logic serves as an indicator to the compiler. Additional specification of this class by use of the apostrophe is unnecessary and represents wasted time.

The local name can only be called or referred to from within the flowchart which contains its definition. In other words, a local name is uniquely identified with a specific flowchart.

Otherwise known as a temporary name, the local noun or verb is purged from the name list when the processor encounters a double period during compilation. This prevents double address definitions for the same name and reduces ambiguity.

Subroutine or Function Name

This class of names is available for reference only during the definition of the subroutine or function. Upon encountering the right brace or END as punctuation, signalling definition completion, the need for these names terminates, and they are deleted from the name list. More information is presented on this class of names in the discussion of subroutines and functions.

Joint Use

All three NELIAC name classes may be used within the same program; furthermore, because of the way the NELIAC processor handles names, a programmer may use the identical name for all three purposes in one program. For example:

```
5 (COMMENT ' ' FIRST PROCESS FLOWCHART)
AJAX, a'jax, BAKER, b'aker, Z
$
2=) ajax + baker =) Z,
Z= 2 ' ' GO (ajax, baker $ Z) $$ STOP.
GO (Ajax, Baker $ z) ' '
    BEGIN Ajax + Baker =) z, END ,
STOP ' '
. .
5 (COMMENT ' ' SECOND PROCESS FLOWCHART)
$
8 =) AJAX + BAKER =) Z
. .
```

The example has been formatted to point out precedence; ordinarily, with the exception of the local name definition (which requires an apostrophe), all names are written to look the same. In the flowcharts above, AJAX represents a global name, a'jax is a local name, and Ajax is a name of functional precedence. The first two are defined in the dimensioning portion of the initial flowchart; the last is prescribed in the function definition.

To illustrate the result of name precedence in the example program, when any of the defined names are used in the program logic of either flowchart, the use or absence of capital letters indicates their precedence. In the first flowchart where AJAX, a'jax and Ajax are all defined, employment of that four letter name within the program logic but external to the function definition results in local name precedence, here indicated by lower case letters. Use of the name in the second flowchart invokes global precedence since a local name of the same spelling has not been defined there.

NOUNS

Noun Usage

As indicated earlier, a noun is one of two forms a name may take. The NELIAC programmer uses a noun to reserve computer storage for data he anticipates inputting, manipulating and outputting. He specifies his storage intentions in the dimensioning portion of the flowchart. The specification may include one or more of the following parameters to fully describe the programmer intentions: the mode; the size of storage to be allocated; the form of the array if more than one value is to be stored simultaneously; any initial values; the signs of the values; the input and output formats of variable data fields; and any partial computer word storage. Each of these topics will be considered in this section.

All nouns must be defined somewhere within the boundaries of the NELIAC program. For maximum object program efficiency, nouns should be specified before their use. Only named full computer word storage (and a type of noun to be considered later -- a literal) may be defined after a reference in program logic.

It should be noted that index registers, although not names or nouns, are defined automatically by the NELIAC processor, and further specification is not required.

Mode

All computational efforts are divided into two types: fixed point and floating point algebra. The basic differences between the two are discussed in the following paragraphs.

FIXED POINT ALGEBRA

When fixed point algebra is used to express data for computation, the programmer does so with the knowledge that he is manipulating integers or whole numbers, and that no fractions are involved. This method of data handling is similar to the way in which people count, number pages, score a basketball game, and do many other daily arithmetic operations.

So far as fixed point algebra concerns programmers, these numbers are treated as if there is a hypothetical radix point to the far right of all significant numbers (e.g., we imagine 236 to be 236.). Throughout our calculations with these numbers, that radix point (decimal or octal) will not move from its right-hand position; in this sense, these numbers have a "fixed point."

Addition, subtraction, and multiplication with fixed point numbers present no problem; however, division does. When dividing one integral number by another, and the dividend is not an exact multiple of the divisor (dividend/divisor = quotient), the dividend is reduced to a nonzero remainder. In fixed point algebra, this remainder is ignored, and although the quotient may not be the completely correct answer, it is accepted as being close enough for use with the algorithm. Another way of saying that the remainder is ignored is to employ the word "truncation"; any portion of the dividend remaining after integral division is truncated or cut off.

Another simple trick is "rounding" an answer to the nearest whole number, and NELIAC fixed point algebra ignores this trick, too.

Some sample fixed-point-algebra calculations follow:

Addition: 21 + 42 = 63
Subtraction: 967 - 822 = 145
Multiplication: 85 x 131 = 11135
Division: 5/2 = 2; 11/3 + 22/3 = 1∅

FLOATING POINT ALGEBRA

When one divides a pie among guests, computes an income tax, or buys a partial share of stock, fractions play an important role in each calculation. Since numerical analysts have chosen to do so and because the digital computer is so designed, these fractions are expressed decimally when using a computer.

When, using pencil and paper, a person solves an arithmetic problem having decimal fractions, experience dictates the proper positioning of the decimal point in the answers. For computers, experience comes from programming; consequently, computer designers have resorted to the use of engineering notation for floating point numbers internally in order to properly accomplish the desired calculations and positioning. During calculations, intermediate results are shifted to maintain a consistent form.

Numbers in ordinary floating point notation are expressed as a whole. Engineering notation for a floating point number consists of three parts, a mantissa, a base and a characteristic.

Ordinary notation: - 1234.56789∅
Engineering notation: - .123456789 x   10   +4
                        ─────────────   ──   ─────
                           mantissa   base characteristic

A mantissa contains the significant digits of the number, with the decimal point left-justified (to the left of the most significant digit). This is a normalized mantissa. A base is the number system under consideration; in this example the decimal system is used, hence the base equals 10. In the octal system,

base equals 8; in the binary system, base equals 2. A character-
istic is that power to which the base is raised to give a multipli-
cand, which when multiplied by the mantissa, results in the float-
ing point number of ordinary notation. Both notations, ordinary
and engineering, are used in NELIAC.

The rules of truncation and rounding for fixed point algebra
may or may not apply to floating point calculations, although not
entirely in the same sense as in the fixed point mode. Truncation
occurs when the space allotted to a mantissa or characteristic is
exceeded; no heed is paid to significance of truncated digits. (Rules
for preventing improper truncation will be presented later).
Rounding in any case never occurs. Consider $3.0 \div 16.5 =$
$0.18181818...$ This calculation if delegated to machine solution
would be carried out until truncation, and if the last digit were a
one, and the next digit beyond allotted mantissa space were to be
an eight, the last digit would remain a one.

COMPARATIVE MERITS OF MODE

There are advantages for and against each mode of algebra.
Fixed point algebra requires less core space; only one computer
word is reserved for each number. All indexing is done in this
mode since fractions are meaningless for such a purpose. A dis-
advantage of the fixed point mode is that it is in general inaccurate
for nonintegral calculations; scaling (multiplying or dividing by
the radix) of operands may be used to achieve more precision, as
shown in the following example:

To accomplish a division of 15 by 64 with two places of
accuracy, (a) scale 15 up by 1∅∅∅ = 15∅∅∅; (b) integrally divide
15∅∅∅ by 64 = 234; (c) for two places of accuracy, add 5, then
truncate the last digit, leaving 23. This is the answer in the
fixed point mode, taking into account that it is 1∅∅ times too large.
(If the calculation had been done in floating point, the answer
would be ∅.23.)

As for floating point algebra, it requires more core space; for each value, a word must be set aside for the mantissa and one for the characteristic. However, as mentioned in the preceding example, this mode allows greater accuracy, without the requirement of scaling. Another point in its favor is the fact that it is far more useful in expressing scientific data and problems, which is why we have a NELIAC, anyway.

Constants

When a value is invariable during all executions of the program, it is a constant. A NELIAC programmer will encounter a constant in two forms. The first form is the initial value, a constant assigned in dimensioning or early in the program logic to preset a noun to a specific number, such as H = <u>47</u>, or -<u>23</u> =) R,. The value of the noun may be modified by program logic, but when an entrance is made to the beginning of the program (as in a subsequent execution), the initial value again presets the noun. The second form is the expression operand, a number which is part of a NELIAC expression and is not changed during execution, such as A * <u>3</u> =) B.

Constants in NELIAC may be written in either mode: fixed point or floating point.

FIXED POINT CONSTANTS

A fixed point constant is an association of numbers in either the decimal or the octal number system. This mode of constants is stored in one computer word ($3\emptyset$ bits in the AN/USQ-20). Its magnitude range decimally is $\pm 536,87\emptyset, 911$ (upper bit is reserved for a sign, and $2^{29}-1$ possible nonzero values as well); octally the range is $\pm$ 37777 77777. For numbers larger than the maximum

specifications, the programmer may expect trouble; the minimum consequence would be truncation of the least significant (most right-hand, nonzero) digits. The octal specification must include the octal sign: OCT (space OCT space; e.g., 213 OCT). In the absence of the octal sign, the constant is assumed to be decimal.

Fixed point constants must be integral in nature; no fractions are allowed, and consequently no radix point. Constants involve only numbers, signs, and blanks (which are ignored by the compiler). No alphabetic or special symbol information may be considered part of a number. The sign of a fixed point constant is assumed to be positive if the sign is physically absent in the specification. Otherwise, it may be positive or negative, as indicated.

Following are a synopsis (Table B3-3) and a list of examples (Table B3-4) of fixed point constants.

TABLE B3-3. CONSTITUENTS OF FIXED POINT CONSTANTS

a.	Numbers and blanks only; no letters or special characters (other than ± or OCT)
b.	Whole number; no fractions or radix point
c.	Sign may be + or −; absence of sign indicates positive
d.	May be decimal or octal; OCT (space OCT space) after number indicates octal; no indicator implies decimal
e.	Magnitudes: ±536870911 decimal
	±37777 77777 octal

TABLE B3-4. EXAMPLES OF FIXED POINT CONSTANTS

a.	-500000000	legal; negative decimal constant
b.	0	legal; zero
c.	497	legal; positive decimal constant
d.	+17 OCT	legal; positive octal constant
e.	-626 OCT	legal; negative octal constant
f.	-500,000,000	illegal; special symbols
g.	-600000000	illegal; exceeds magnitude allowed
h.	-43B7	illegal; letters not permitted
i.	+27.	illegal; decimal point
j.	123459 OCT	illegal; not an octal number

FLOATING POINT CONSTANTS

All fractional values are expressed in NELIAC by the floating point constant. As discussed earlier, numbers of this mode require two computer words for storage, one word each for the characteristic and mantissa. The range in magnitude of the characteristic is ± 268,435,455 (sign bit, overflow bit and 2^{28}-1 possible nonzero values). The range of the mantissa is ± 536, 87∅,911, with the storage oriented identically as that for an equivalent fixed point number. NELIAC is capable of handling only decimal floating point numbers.

Floating point numbers, as we noted previously, are written in two forms: the ordinary notation and the engineering notation. Each form in NELIAC has its idiosyncrasies; both have common elements. These common elements will be covered first.

Floating point constants involve only numbers, signs, blanks (which are ignored by the compiler), and decimal points (optional in the dimensioning specification of engineering notation for whole numbers). Special symbols and letters may not be part of a constant. Unsigned constants are considered positive; signed numbers are considered as indicated. All numbers less than one must have a zero before the decimal point.

In the ordinary notation, the decimal point is mandatory. The NELIAC specification for this notation consists of only a mantissa: a sign, an integer or integers, a decimal point, and a decimal fraction, if necessary (e.g., -∅∅27351694∅∅). The significant digits (27351694) may not exceed the magnitude range of the mantissa. This notation may be used in both the dimensioning and the program logic portions of the NELIAC flowchart.

In the engineering notation, constants are comprised of a mantissa and a characteristic. The general NELIAC form for this notation is "± MANTISSA * ± CHARACTERISTIC," where the mantissa consists of integral and fractional parts separated by a decimal point.

When specifying whole numbers, a decimal point is not mandatory. For example, the number +24.∅ may be expressed in NELIAC in many ways, several of which follow:

a. +24.∅*∅
b. +24*∅
c. +24∅*-1
d. +24∅.∅*-1
e. +∅.24*2
f. 24*∅

The first case ("a") is the basic form. Case "b" is identical to "a" except for the absence of the decimal point; the omission is permitted because the number is integral (no fractional part). Cases "c," "d," and "e" illustrate the fact NELIAC specifications of such a number need not be consistent with regards to decimal point justification. A programmer may locate the decimal point anywhere in his mantissa providing his characteristic is changed accordingly so as to not modify the given value. Internally, however, the decimal point of the floating point constant is still justified fully to the right or left (depending upon machine characteristics). The last case ("f") illustrates the omission of sign from the second case.

The significant digits of the mantissa may not exceed the maximum magnitude. A similar restriction is placed upon the characteristic. The engineering notation may be used in the dimensioning portion of a flowchart only.

Following are a synopsis (table B3-5) and a list of examples (table B3-6) of floating point constants.

TABLE B3-5. CONSTITUENTS OF FLOATING POINT CONSTANTS

a. Numbers and blanks only; no letters or special characters (other than ±, *, or .)
b. Ordinary notation, mantissa: sign, integers, decimal point and decimal fractions
c. Engineering notation, mantissa, and characteristic: sign, integer (s); decimal point and decimal fraction of mantissa may be omitted in specification of whole numbers
d. Signs of mantissa and characteristic may be + or −; absence of sign indicates positive
e. Decimal floating point constants only
f. Magnitude range:
+536,87Ø,911.Ø x 10 ± 268,435,455
$\underbrace{\qquad\qquad\qquad}$ $\underbrace{\qquad\qquad\qquad}$
mantissa   characteristic
g. Constants less than one require a zero before decimal

TABLE B3-6. EXAMPLES OF FLOATING POINT CONSTANTS

a.	Ø.Ø, Ø.Ø*Ø, +Ø.Ø, +Ø*Ø	legal; zeros
b.	5.23, 52.3*-1, 0.523*1, 0.523*+1, +5.23*Ø,	legal; variety
c.	-5ØØØØØØØØ*- 1ØØØØØØØØ,	legal; small constant
d.	-Ø.46325786, Ø.ØØØØØØØØØØØ4	legal; fractions
e.	Ø	illegal; fixed point number
f.	-1,ØØØ.Ø	illegal; special symbol
g.	2.3*-75Ø236429	illegal; characteristic too large
h.	-4.Ø*7.6	illegal; characteristic is floating point
i.	7.7*77 OCT	illegal; octal floating point
j.	-.693	illegal; missing leading zero
k.	+46.7-8	illegal; missing asterisk

Variables, Whole Word

A variable is an area in core set aside to maintain data of a varying or nonconstant nature. The values assigned to a variable may change during execution or between executions of a particular program.

There are two patterns of storage provided by the NELIAC language: whole word and partial word. A discussion of partial word storage will come later. Whole word storage implies that the smallest unit of storage is the computer word (30 bits in the AN/USQ-2∅). As mentioned, the fixed point mode requires one word per constant. The floating point mode, however, requires two words to express a similar value. In both modes, no matter how small or how large the value, the number is stored in some multiple of the word.

A whole word variable, then, is a noun used to tag one or more words in core reserved for storage of fixed point or floating point data.

The mode of a variable is specified by punctuation in the dimensioning section of the NELIAC flowchart. If a properly formed NELIAC noun is immediately followed by a comma, the punctuation indicates that this variable is fixed point. Index register variables, I through N, may be dimensioned in a flowchart as fixed point variables, but such dimensioning serves no purpose since the NELIAC processor automatically provides definition for these variables. A noun employed for fixed point whole word storage may be referenced in program logic before being dimensioned elsewhere in the program.

If a NELIAC noun is immediately followed by a period, this indicates that the variable is reserved for floating point data.

Following are a synopsis (table B3-7) and a list of examples (table B3-8) of whole word variables.

TABLE B3-7. CONSTITUENTS OF WHOLE WORD VARIABLES

a.	Variables are identified by properly formed NELIAC nouns
b.	Fixed point whole word nouns need not be dimensioned before use; must be dimensioned somewhere in program
c.	Fixed point variables: noun followed immediately by comma
d.	Floating point variables: noun followed immediately by period
e.	Index registers should not be dimensioned

TABLE B3-8. EXAMPLES OF WHOLE WORD VARIABLES

a.	ABCZ,	legal; fixed point variable
b.	ABCZ.	legal; floating point variable
c.	3LTD	illegal; improper NELIAC noun
d.	I.	illegal; register variables should not be dimensioned, definitely not as floating point

CHANGING MODES OF VARIABLES

When a variable is dimensioned as floating point, and a fixed point constant is moved to the area in core reserved for that variable, the number is converted to a normalized floating point constant before storing it. The converse holds true for the floating point constant intended to be stored in a fixed point variable. However, any fractional portion of the mantissa is truncated before storage. For example:

a. +86.793 stored in a fixed point whole word variable becomes +86

b. -126 stored in a floating point whole word variable becomes the equivalent in core of -$\emptyset$.126*3

DIMENSIONING WHOLE WORD VARIABLES

In the foregoing, information necessary for a preliminary discussion of dimensioning has been presented. The purpose for the next brief discourse is to help tie together constants and variables as used in dimensioning, and to introduce further topics which could not be discussed without such prior discourse.

As stated previously, dimensioning is to inform the compiler that areas are to be reserved by name for specific data, as well as to define the mode, initial values (if any), and the storage pattern for whole or partial words. Dimensioning then, serves two purposes: it assists the compiler in storage allocation, and it assists the programmer by putting initial values in specific reserved areas.

By dimensioning a noun in NELIAC, the programmer indicates a need for storage. The amount of storage that is reserved depends upon that follows the noun. In the absence of an initial value, a comma indicates fixed point, and one computer word is reserved for one constant; a period indicates floating point, and two computer words are reserved for one constant. In either case, the value stored by the compiler in the variable is exactly zero.

To store a nonzero initial value, an equal sign and the value and a comma follow the noun. For example:

SHARKS = +47.67*-2,
JEM = -114,
PART = 1∅9.83,

No matter what the mode of the initial value, not that it is followed by a comma. The mode of the variable is determined by that of the initial value. Once the variable is dimensioned, it is not possible in program logic to change its mode; any value stored later in that variable will conform to the mode already specified.

Several fixed or floating point nouns may share the same variable definition. For example, with the following dimensioning statement, the fixed point variables A, B, and C may all jointly occupy the same computer word:

A ' ' B ' ' C,

All are preset to zero. The floating point variables XRAY, YAN-KEE, and ZULU are defined at the same locations by the following:

XRAY ' ' YANKEE ' ' ZULU = -1.47,

Each is preset to the initial value. The double apostrophe acts as the connector in any multiple variable specification.

Bitfield Algebra

The format of data storage in digital computers is determined by a system called a code. All computer logic circuitry is based on one code or another. In general, the differentiation between machine types is the manner in which instructions and data are handled.

There are several different codes in common use. The most widely used are the character (binary coded decimal or alphanumeric) and the binary codes. The lowest denominator of storage in a character machine that may be addressed is the character itself, while in a binary computer the data are manipulated in full word increments. Most binary computers offer in their machine language the additional capability of handling data one bit at a time.

Few procedure oriented languages for binary digital computers offer the programmer the capability of handling the contents of computer words a bit at a time; NELIAC, however, is atypical in this respect. The significance of this capability will be discussed in the next paragraphs.

PACKING

When small positive numbers are stored in whole computer words, zeros are employed to fill the remaining binary digits. Consequently, multiple entries of low magnitude data require large storage areas primarily full of zeros. To utilize the computer core efficiently, a method must be available for the careful programmer to pack and unpack large amounts of data in a small amount of core.

NELIAC provides this packing and unpacking capability. Packing involves the storage of data side-by-side, with as little insignificant information as possible (preferably none) stored with the important data. For example, if three variables are to be dimensioned, and each variable has a significant (nonzero) length less than half a word, the programmer might consider making a composite number out of these values; such a packing should require a maximum of two whole words. Consider the case of A = 5, B = 146, C = 2479. Since the numbers are stored internally in the octal number system, before creating a composite value these should be converted: A = 5, B = 222 OCT , C = 4657 OCT . The composite value might be G = 4657ØØ2225 OCT . Note that this value is considerably less than the maximum of 77777 77777 OCT .

UNPACKING

As might be expected, unpacking involves the capability of selecting certain contiguous bits within the specified word and assigning them to another variable for additional manipulation or output. This operation would follow when a programmer for purposes of efficiency had packed many variables into a few and was now ready for separation of the values into their respective variables.

As an example, take G (defined above under "Packing") and, is program logic, divide the word. This requires the use of the partial word variable to be discussed shortly but which for immediate purposes must be introduced now. If G = 4657ØØ2225 OCT and the word size is 3Ø bits, each octal digit takes 3 bits, the bit positions being numbered Ø through 29 in a right to left direction. Consequently, to obtain 4657 OCT to restore C, the statement is made that G (18 =) 29) =) C. This commands the computer to take bits 18 through 29 of G and store them in C. To restore A and B, G (3=) 11) =) B, and G (Ø=) 2) =) A would be written.

DISCUSSION

Operations in program logic which involve bitfield algebra generate more machine language instructions than the equivalent whole word operation. In short, there is what is known as a "tradeoff"--efficiency in storage vs. efficiency in instructions. The tradeoff becomes advantageous to the programmer when there are long lists or tables of small values to be defined. (These elements are discussed later within the confines of this section.)

Certain computers (including the AN/USQ-2Ø) have the capability of handling half-words with the same facility as whole words. Small numbers, then, may be stored two to a register, representing a reduction by half in dimensioned storage. The machine code instructions generated from program logic to handle these half words are no less efficient than the equivalent instructions to handle whole words. Therefore, no tradeoff in half-word algebra is necessary. Where it can be used, this form of data handling is advantageous.

Variables, Partial Word

The term "partial word storage" defines the fact that the smallest unit of storage is something less than a full computer word: a bit, a series of bits, or a half word. With this facility,

the programmer need only reserve a portion of a register for his variable; i.e., he may store several variables within another variable. He may also select certain contiguous bits from a full word variable for separate manipulation. Several examples of these tools have been discussed in the previous paragraphs. Now the mechanics of partial word variables will be examined.

An important aspect of bitfield algebra that the programmer must keep foremost in mind is that any partial word data are of necessity positive fixed point constants. A number stored in or extracted from a partial word must be assumed to be an unsigned integer. The assumption holds since only with sign extension (moving the sign bit with the constant) can a negative number be properly signed, and only with two full words (not a partial word) may a floating point number be represented.

All partial word variables must be defined before use. This rule necessitates dimensioning the variable somewhere in the program prior to involving it in a logic expression. When employing a partial word variable in an expression its formation is VARI-ABLE (LOWEST BIT =) HIGHEST BIT); e.g., ASIAN (14 =) 26). When specifying a single bit, the lowest and highest bit are identical; e.g., ASIAN (14 =) 14). The symbol "=)" between bit specifications means "through." Bitfields must be within a single word; there can be no partial word variables which cross the imaginary line dividing computer registers. Implicit in this restriction is the statement that a smaller number always precedes a larger number in partial word specifications.

The programmer must insure sufficient space in his bit-field definitions for the data he intends to store. When an attempt is made to store a number too large for the definition, a part of the number (the most significant or leftmost digits) will be truncated.

Partial word variables may be dimensioned by enclosing the bitfield definitions between BEGIN and END punctuation symbols. A maximum of one computer word is dimensioned between symbols. For example:

BEGIN ECHO (23 =) 27), END ,

Examination of this particular example reveals the fact that an entire word of zeros will be reserved for ECHO whether or not a bitfield specification is made in the variable definition.

If, however, a clever programmer wishes to use several variables in one word, he may so define them with the understanding that bitfields of variables may overlap. To illustrate:

BEGIN ECHO (23 =) 27), XYLD (24 =) 28),
ANTE (∅ =) 4), END ,

He has packed three partial word variables in one word.

Multiple variable specifications in dimensioning incorporating partial word variables are possible as in the following example:

ALPHA ' ' BANG ' ' BEGIN CHECK (23 =) 29), END ,

Here CHECK is defined as the 23rd through 29th bits of the coincident variables ALPHA and BANG. All three variables are considered to be defined by this one statement. No multiple variable specification is allowed within the punctuation BEGIN and END; i.e., A ' ' BEGIN B ' ' C (∅ =) 1), END , is not permitted.

It is also possible to define several different partial words within ALPHA and BANG provided that the bitfield specifications are enclosed between BEGIN and END. For example:

ALPHA ' ' BANG ' ' BEGIN C (∅ =) 4), D (3 =) 9),
E (23 =) 23), END ,

Notice in this example that if more than one bit of information is stored in either ALPHA or BANG, it is not possible to retrieve more information than that from the variable E. Note, also, that storing a value in D does not imply it is retrievable from variable C (although it is from ALPHA or BANG).

Consider a multiple variable specification like the following:

B ' ' BEGIN A (6 =) 11), END ,

In the following, A is defined as bits six through eleven of a word named B:
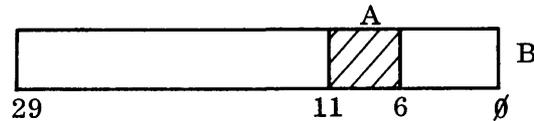
If the variable A is involved in an expression,

$$A \; (\emptyset \; =) \; 2) \; =) \; D,$$

bits zero through two of A are being stored in variable D; this is the equivalent of storing bits six through eight of B.

Index registers also may be treated as partial word variables, although they may not be so defined in dimensioning. In program logic, the last five bits of index register five might be stored in a variable HOLD:

$$M \; (1\emptyset \; =) \; 14) \; =) \; HOLD,$$

It is emphasized again that a partial word variable must be a valid NELIAC noun.

Following are a synopsis (table B3-9) and a list of examples (table B3-10) of partial word variables.

TABLE B3-9. CONSTITUENTS OF PARTIAL WORD VARIABLES

a. Variables are identified by properly formed NELIAC nouns
b. Only positive fixed point data may be stored or extracted
c. Must be defined before use in a flowchart
d. Bitfields specified are contained in a single word
e. Bitfield: VARIABLE (LOW BIT =) HIGH BIT)
f. Single bits: VARIABLE (BIT =) BIT), as A (1$\emptyset$ =) 1$\emptyset$)
g. Multiple partial word specifications possible, but variables containing bitfields are enclosed by BEGIN and END
h. Index registers may be handled as partial word variables

a. BEGIN D35K (23 =) 23), END ,	legal; single bit, dimensioning
b. BEGIN TEST (∅ =) 27) = 146, END ,	legal; 28 bits, dimensioning
c. A ' ' B ' ' BEGIN C (19 =) 26), END ,	legal; multiple variables, dimensioning
d. I (8 =) 14) =) ALPHA (13 =) 19),	legal; partial word phrase, program logic
e. 847 =) XYZ (2∅ =) 29),	legal; partial word phrase, program logic
f. 9DELT (14 =) 15),	illegal; improper NELIAC noun
g. A'ZAHJ (15 =) 9),	illegal; bits limits in wrong order
h. 14.∅7 =) HELP (1 =) 13),	illegal; no floating point data allowed
i. -14.∅7 =) HELP (1 =) 13),	illegal; no negative data allowed
j. A ' ' B ' ' C (19 =) 26),	illegal; BEGIN and END missing
k. I (17 =) 26) =) ALPHA (13 =) 19),	illegal; index register variables are only 15 bits long: ∅ =) 15

Subscripts

To understand the reasons for subscripting, consider an equation with several terms:

$$AX^6 + BX^5 + GX^4 + LX^3 + CX^2 + SX + D = 14$$

The coefficients (A, B, G, L, C, S, and D) are purposely mixed alphabetically to illustrate a point: if a single letter with ascending subscripts was used instead, there would be less confusion as to which coefficient goes with which term:

$$A_{\emptyset} X^6 + A_1 X^5 + A_2 X^4 + A_3 X^3 + A_4 X^2 + A_5 X + A_6 = 14$$

In this equation, only one name, A, would be necessary to identify a particular item -- the coefficients.

If this argument is extended to a matrix, the reasons are even clearer, for now the confusion in row and column coefficients can be eliminated:

<div style="text-align:center">COLUMNS   COLUMNS</div>

$$
\begin{matrix}
\text{R} \\ \text{O} \\ \text{W} \\ \text{S}
\end{matrix}
\begin{bmatrix}
Z & K & M & D \\
N & G & U & A \\
X & D & B & R \\
S & O & Y & Q
\end{bmatrix}
\quad \text{becomes} \quad
\begin{matrix}
\text{R} \\ \text{O} \\ \text{W} \\ \text{S}
\end{matrix}
\begin{bmatrix}
A_{\emptyset,\emptyset} & A_{\emptyset,1} & A_{\emptyset,2} & A_{\emptyset,3} \\
A_{1,\emptyset} & A_{1,1} & A_{1,2} & A_{1,3} \\
A_{2,\emptyset} & A_{2,1} & A_{2,2} & A_{2,3} \\
A_{3,\emptyset} & A_{3,1} & A_{3,2} & A_{3,3}
\end{bmatrix}
$$

The first subscripted digit after the name is the row and the second digit is the column of the matrix in which one would expect to find the coefficient. The range of row and column numbers begins with zero.

The expectation that a variable may be assigned several values, all of which are to be retained concurrently in core, forces the programmer to provide storage for these constants. This type of storage will involve the use of a variable name and subscripts to distinctively identify each element of the value list.

Subscripted variables serve the programmer as an indexing tool; in an array of similar numbers we can index or point directly to a particular value. If we ask for $A_{\emptyset}$, we have indexed to the zero element of an array named A.

SUBSCRIPT FORMS

Presently available input/output equipment for digital computers does not provide the resources for handling numbers which are above or below the printed line. As a consequence, no provision for this has been built into the NELIAC language. NELIAC subscripts are written on the same line as the variable but are enclosed in punctuation to set them apart, as in "A ($ 4 $)."

Subscripts are encountered in both parts of a NELIAC flowchart, dimensioning and program logic. Consistent with the definition of dimensioning, a subscript associated with a variable in dimensioning indicates the number of values a programmer expects to assign to that variable. VARIABLE (7). indicates that seven floating point numbers will be stored in a core area (in this case 14 words) identified by the name VARIABLE.

A subscript associated with a variable in program logic is employed to select a particular value from among several assigned to the variable and involves this value in some programmer-specified algebraic manipulation. VARIABLE (7) in program logic indicates that the contents of VARIABLE$_7$ are currently being considered mathematically in a NELIAC statement. Subscripts used without variables (in program logic) are discussed in Section 5 under the topic "Indirect Addressing."

To dimension a variable that will have length of seven values, a programmer writes in NELIAC "PERS (7)." However, when reference is made in program logic to the first element of the list, the address is "PERS ($\emptyset$$)" and the seventh element is "PERS (6)." Notice that the subscripts assigned internally run from zero to one less than the length specified in dimensioning. This mental conversion (of length to subscript) must be borne in mind when subscripts are used.

A subscript takes many forms depending upon its intended use. In dimensioning, where a subscript specifies a list length, it is a constant; any other form would be ambiguous since the compiler depends upon this subscript for storage reservation. The subscript follows the variable and is enclosed in parentheses, as in "OBOE(4)."

In program logic, one refers to an element in a list with any one of the following subscript forms: constants, nouns, index register variables, or index register variables plus or minus a constant.

If a constant is used in dimensioning or program logic as a subscript, it must be an unsigned fixed point number less than or equal in magnitude to 77777 OCT or 32767 decimal. Fixed point subscripts are employed since floating point numbers are neither desirable nor necessary for use as subscripts. For example $X_{1.3}$ is not generally a meaningful notation, and $X_{3.0}$ is redundant when X_3 is sufficient. The floating point mode is therefore not implemented.

An important point to remember is that the subscript itself does not have any bearing on the mode of the subscripted variable. The fact that subscripts must be fixed point does not affect a fixed or a floating point variable in the determination of the variable mode.

Nouns used for subscripts imply that the contents of the named computer word will be used as the actual subscript. The numerical subscript will change as the value assigned to the noun is changed in program logic. When the subscript is referenced by name, the value contained in the noun at the instant of consideration is the subscript for that execution. If the noun APPLE has been given the value 4, BASKET ($ APPLE $) would refer to BASKET (4) or the fifth element of the list named BASKET.

In the introduction to index register variables under "Grammar of Names" at the beginning of this section, it was indicated that the use of these special nouns for some jobs generated more efficient code than if other nouns were used for the same purpose. This statement holds for subscripts; wherever possible, the index register variables, I, J, K, L, M, and N should be employed as indices.

If an index register variable and a constant are used jointly as a subscript they are separated by a plus or a minus sign. The sum of the contents of the index register and the constant must not exceed the magnitude limits of the constant alone -- 77777 OCT or 32767 decimal. In format, the index register variable must always precede the constant in a subscript.

Under no circumstances may a subscript be subscripted. This means that any noun used as a subscript must itself be

unsubscripted. Index register variables cannot be subscripted.

Following are a synopsis (table B3-11) and a list of examples (table B3-12) of subscripts.

TABLE B3-11. CONSTITUENTS OF SUBSCRIPTS

a.	Dimensioning: (SUBSCRIPT)
b.	Program logic: ($ SUBSCRIPT $)
c.	Forms:
	1) Dimensioning: constant only
	2) Program Logic: constant, noun, index register, index register ± constant
d.	Constant: unsigned (positive) fixed point number ≤ 77777 OCT or 32767 decimal
e.	Noun: valid NELIAC name; contents of variable is actual subscript, ≤77777 OCT or 32767 decimal
f.	Index register variable: I, J, K, L, M, N; contents of index register is actual subscript
g.	Index register variable ± constant: variable must precede constant; sum of contents of index register and constant must be ≤ 77777 OCT or 32767 decimal
h.	Subscripts may not be subscripted

TABLE B3-12. EXAMPLES OF SUBSCRIPTS

a.	(3∅ OCT), (1), (32767)	legal; list lengths (dimensioning)
b.	($ 45 $), ($ I $),	legal; subscript for program logic
c.	($ HOLOCAUST $)	legal; subscript for program logic
d.	($ I+42 OCT), ($ M-3 $)	legal; subscripts for program logic, (variable ± constant form)
e.	(I), (HOLOCAUST)	illegal; constants only for dimensioning
f.	(4∅∅∅∅)	illegal; constant too large
g.	($ 4+I $)	illegal; variable and constant reversed
h.	($ A ($ I $) $)	illegal; subscripts may not be sub-scripted
i.	($ -36.9 $)	illegal; subscript must be unsigned and fixed point

SUBSCRIPTED VARIABLES

Consider a list of numbers. If they are stored in a computer for referencing in the course of program logic, they must be assigned to a variable. To individually select or index individual elements of this list, the name of the list and the subscript (position of the element) are specified. In doing so, a subscripted variable has been used.

A series of constants is usually referred to as a list if it has one dimension and as an array or table if it has two dimensions. NELIAC has the capability of handling either possibility, with one subscript for lists and two subscripts for arrays or tables.

A list in dimensioning is specified with a variable and a constant in parentheses, denoting list length. When addressing a particular list element in program logic, a variable and a subscript enclosed in a combination of parentheses and dollar signs (as indicated in the previous discussion of subscripts) are required.

An array or table in dimensioning requires an asterisk between subscripts to give a row and column specification. The asterisk is used in NELIAC as a multiplication sign; the product of the two subscripts will indicate to the programmer the size of storage he is reserving. The product must not exceed the maximum core size of the computer.

One element of an array in program logic is addressed in a fashion similar to an element of a list. Two subscripts are required instead of one, and they are separated by a comma. The format otherwise is identical. A doubly dimensioned variable cannot be used before definition.

Samples of the forms, in order, are:

a. List, dimensioning: A(14),
b. List, program logic: A ($ 13 $) =) B,
c. Array, dimensioning: A (4 * 9),
d. Array, program logic: A ($ 3,7 $) =) B,

The programmer is cautioned to avoid the obvious pitfall of expressing "A_3" as "A3"; such a representation is a NELIAC noun entirely separate from the original variable, A. Another possible pitfall is the double subscript when specifying the variable name of a singly dimensioned list. Both will cause results that for all purposes are errors.

If a noun is used as a subscripted variable, it represents a collection of values and may not be used without a subscript except in two cases. The first case is its use in an input/output statement when it is desired by the programmer to transfer the entire list or array. The second case occurs when the variable is found in an arithmetic statement in which case it is interpreted as if it were the first element of the list or array as in VARIABLE ($ Ø $).

Following are a synopsis (table B3-13) and a list of examples (table B3-14) of subscripted variables.

TABLE B3-13. CONSTITUENTS OF SUBSCRIPTED VARIABLES

a. Valid NELIAC name followed by appropriate subscripts
b. List, dimensioning: VARIABLE (SUBSCRIPT)
c. List, program logic: VARIABLE ($ SUBSCRIPT $)
d. Array, dimensioning: VARIABLE (SUBSCRIPT 1 * SUBSCRIPT 2); SUBSCRIPT 1 = row, SUBSCRIPT 2 = column
e. Array, program logic: VARIABLE ($ SUBSCRIPT 1, SUBSCRIPT 2 $); SUBSCRIPT 1 = row, SUBSCRIPT 2 = column
f. Noun defined as subscripted variable used without subscripts specifies whole list or array in I/O, or first element in list or array in program logic

TABLE B3-14. EXAMPLES OF SUBSCRIPTED VARIABLES

a.	BOA (4),	legal; 4 element list (dimensioning)
b.	KP (4 * 7),	legal; 28 element list dimensioning
c.	MAN (8 * 13 OCT),	legal; 12∅ element list (dimensioning)
d.	JOB ($ XRAY $) =) A,	legal; variable in program logic
e.	D ($ I+7 $) =) A,	legal; composite subscript (program logic)
f.	AFOGE ($ 13, 9 $) =) A,	legal; double subscript (program logic)
g.	N'ICK (I),	illegal; list length must be constant
h.	I (NICK)	illegal; subscripted register variable
i.	GZ ($ J, K, $) =) A,	illegal; second comma is improper
j.	PORFO LB (6 * 9 * 3),	illegal; more than two dimensions
k.	MAN5	illegal; not a subscripted variable

A programmer may desire to preset some or all of the elements of an array or list before program execution. As in single value variables, NELIAC makes provision for initial values.

In a many-element list or array, if no initial values are specified, all elements are preset to zero by the compiler. The mode of such a variable is determined by these punctuations: a period following the variable definition denotes floating point, and all elements are reserved two words each; a comma following the variable implies fixed point, and all elements are reserved one word each.

When nonzero initial values are given, the mode of the variable is dictated by the mode of the first element, and all succeeding initial values must be written to match that mode if an error in dimensioning is to be avoided. In addition, assuming a properly punctuated constant specification, any elements left blank are assumed to be zero.

Some examples of subscripted variables with initial values follow:

a.	B(14).	Denotes fourteen floating point elements in a list named B; all are preset to zero.
b.	R(4 * 3),	Reserves twelve words in core for twelve fixed point variable R elements; all are preset to zero.
c.	ANCHOR (4) = 6, 2, 14, -237,	Defines four initial values for the fixed point variable ANCHOR.
d.	ZY (5) = $\emptyset$*$\emptyset$, 8.$\emptyset$, -$\emptyset$.$\emptyset\emptyset$3, 4762.$\emptyset$, -15.62* -13,	Five floating point values preset the variable ZY; note that the absence of a first value would have caused ZY to be defined as fixed point and an erroneous definition of all other values would have followed.
e.	NOBE(3*2) = 6, , 2,-1, ,9,	Two fixed point zeroes are implied through punctuation; elements are stored sequentially by rows: 6,$\emptyset$, 2,-1, $\emptyset$,9,

In an extension of the above discussion, if a variable is dimensioned as having seven values, and only the first four are specified, as in A(7) = 1,2,3,4, the remaining values are assumed to be zero; no further punctuation is necessary.

When subscripting partial word variables, the rules for these variables are a summation of restrictions applied to partial words and subscripted variables. A programmer who wishes to store a 4 in bits nine through eleven of the tenth element of variable DACHS does so by specifying the following:

$$4 =) \text{DACHS} (\$ 9 \$) (9 =) 11,$$

To dimension forty elements with a bitfield specification of bits six through eight of a variable ENTROPY, the correct form is:

BEGIN ENTROPY (6 =) 8), END (4∅),

If these bitfield elements were to be part of another variable THERMO, the definition would look like this:

THERMO ' ' BEGIN ENTROPY (6 =) 8), END (4∅),

This is interpreted to mean that there is a list named THERMO, forty words long, and, in each element, bits six through eight may also be referred to as ENTROPY ($ n $), where n = ∅ through 39, and THERMO ($ n $) refers to the full word element; THERMO ($ n $) (6 =) 8) would be equivalent to ENTROPY ($ n $). ENTROPY ($ n $) (∅ =) 1) would specify bits six and seven of THERMO ($ n $).

Moving THERMO ($ n $) (3 =) 7) to BIX3 would cause bits zero through four of BIX3 to be reset to the contents of the THERMO bitfield and any remaining bits of BIX3 to be reset to ∅. (Refer to figure B3-2 and to the preceding paragraphs under "Variables, Partial Word.")
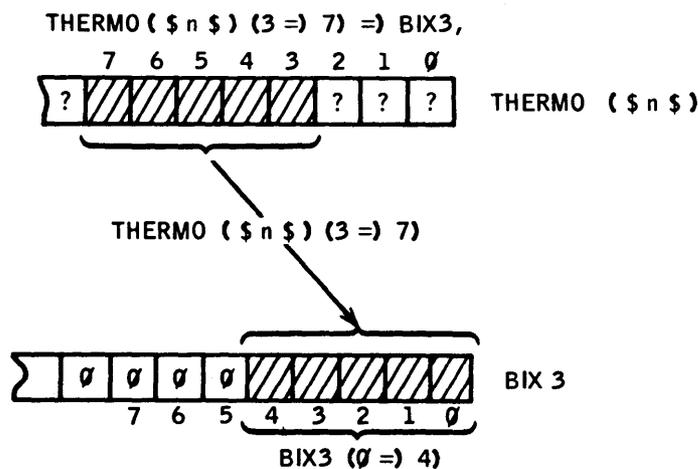
Figure B3-2. Partial word transfer.

MATRICES

A matrix is the general term applied to a group (defined as ≥ 1) of elements assigned to a variable. The group is always considered to be in rectangular form, row by column. A 1-by-1 matrix is a single element; a 1-by-N matrix has one row, N columns, or N elements; an M-by-1 matrix has M rows, one column, or M elements; and an M-by-N matrix has M rows, N columns, or M times N elements.

Under previous definitions, a list is a 1-by-N matrix and an array is an M-by-N matrix. One may consider a variable subscripted in dimensioning without initial values as a "normal" matrix. A "preset" matrix is a subscripted variable with one or more dimensioned initial values. A "congruent" matrix, often referred to as a congruent table, is a matrix known by more than one noun. In other words, a multiple variable specification may be applied to the same area in core and the area addressed by several names. The following examples explore the possibilities provided by these matrix types:

a. A '' B '' C,

A congruent, normal matrix with one element, addressable by A or B or C.

b. A(2) '' B, C,

A congruent, normal matrix; matrix A has two elements, B and C; matrices B and C are each composed of a single element.

c. A(10) '' B(6), C(3), D,

A congruent, normal matrix; matrix A has three partition matrices, B, C, and D, with 10 elements; matrix B has 6 elements, matrix C has 3 elements, and matrix D has a single element.

d. A(7) '' B(4), C(3) '' D(2), E,

A congruent, normal matrix; matrix A has two partition matrices, B and C; matrix C has two partition matrices, D and E; the number of elements of each matrix is indicated by subscript.

e. A ' ' B ' ' C = 1,

A congruent, preset matrix; A, B, and C are single element matrices, all equal to 1, all occupying the same single word in core.

f. A(7) ' ' B(4) = 1, 3, 5, 7, C(3) ' ' D(2) = 2, 4, E = 6,

A congruent, preset matrix similar to example d; A ($ 3 $) = B ($ 3 $) = 7 and A ($ 5 $) = C ($ 1 $) = D ($ 1 $) = 4.

g. A ' ' B ' ' BEGIN C(6 =) 11), END ,

A congruent, normal matrix with bits 6 through 11 of matrices A and B known as the matrix C; all matrices have single elements.

h. A(7) ' ' B(4) = 1, 3, 5, 7, C(3) ' ' BEGIN D(14 =) 17), END (2) = 2, 4, E = 6,

A congruent, preset matrix similar to example f except matrix D consists of bits 14 through 17 of the first two elements of matrix C which have initial values of 2 and 4, respectively.

i. A(2 * 3) ' ' B(3 * 2) = 1.$\emptyset$, 2.$\emptyset$, 3.$\emptyset$, 4.$\emptyset$, 5.$\emptyset$, 6.$\emptyset$,

A congruent, preset matrix; the third element may be referenced by either A($ $\emptyset$, 2 $) or B($ 1, $\emptyset$ $).

j. A(2 * 3) ' ' B(3 * 2) ' ' C(7 * 1) = 1, 2, 3, 4, 5, 6, 7,

A congruent, preset matrix; note that matrix C is dimensioned as having one more element than A or B; reference to C($ 6, $\emptyset$ $) indexes to the first word beyond both A and B but is the last element of C; this is an acceptable dimensioning statement, much as A($ 6 $) refers to the sixth word beyond the single word defined by A(1); A($ 2, $\emptyset$ $) and B($ 3, $\emptyset$ $) both contain a 7; A($ 2, 1 $) would refer to the first word beyond all three dimensioned matrices.

The double apostrophe signals congruency -- the left-hand matrix is said to be congruent to the right-hand matrix. Both matrices occupy an identical area in core if the subscripts indicate equivalence in area reserved. If one subscripted variable occupies more core than another, overlap where possible is accomplished by the compiler.

Dimensioning

The format and purposes of dimensioning will now be examined. Examples will be used to illustrate the topics of dimensioning already discussed. Although some elements of NELIAC theoretically should be discussed before dismissing dimensioning, for better understanding and text organization they have been left to later sections.

Dimensioning is in part a concession to the needs of the compiler. The programmer, as the only constituent of the problem-solving process understanding in advance the storage requirements of his program, must specify all he knows about his variables before they are used in the program logic. The compiler has no way of guessing the programmer's intentions; in this sense, the programmer must accommodate the language.

Dimensioning, as well, accommodates the programmer. It provides (a) a vehicle for defining initial values, (b) formats for data inputs and outputs, (c) switches for directing program flow, and (d) diagnostic messages (of which (b), (c), and (d) are the topics reserved for the later discussion).

Dimensioning in any flowchart or program must precede the program logic because of the concessions to the compiler just enumerated. The first NELIAC character in a program is the control number, and it indicates to the NELIAC compiler the type of flowchart the programmer has written. The last character in dimensioning is the dollar sign, which is a flag indicating to the

compiler that all preceding is dimensioning and all following is program logic. Examine the following typical dimensioning examples:

a. 5   Control number.

    A,   Fixed point whole word variable.

    B.   Floating point whole word variable.

    C = 1,   Fixed point whole word variable preset to 1.

    D = 4.∅,   Floating point whole word variable preset to 4.∅.

    $ Flag.

    (COMMENT '' PROGRAM LOGIC)..

b. 5

    BEGIN EXTRA (9 =) 15),   Partial word variable defined equal to ∅.
      END ,

    BEGIN J1 (∅ =) 1∅),   Several partial word variables defined within one computer word.
    J2 (∅ =) 7), J3 (8 =) 1∅),
    J4 (9 =) 1∅), END ,

    F '' G '' BEGIN   Multiple variable specifications incorporating a half word variable.
    H (∅ =) 14), END ,

    $

    (COMMENT '' PROGRAM LOGIC)..

c. 5

    PAX (3) = 1, -147, ,   Preset fixed point list.

    PIX (5) = -31.72,   Preset floating point list.
    9.2, -∅.∅∅3, + 85.6 * 6

    POX (2 * 2) = 18, -7, + 3,   Preset fixed point array.

    Q ''   Congruent, preset, single element matrix.
    LONG WAY TO SAY MA-
      TRIX = 1,

    R (7) ''   Congruent, preset matrix of seven elements.
    S (4) = 1, 3, 5, 7,
    T (3) ''
    BEGIN ULTIMATE (14 =) 17),
    END (2) = 2, 4, V12 = 6,

    $

    (COMMENT '' PROGRAM LOGIC)..

Note that the individual specifications may be written to fill a line or strung out over several lines. The ability of the NELIAC compiler to ignore insignificant blanks permits the programmer this freedom.

VERBS

Verb Usage

In addition to the noun form, a name may take the form of a verb. A verb is the name that a programmer assigns to a flowchart routine (several lines of NELIAC), function, or subroutine which enables him to reference a line or routine within his program. Examination of a simple problem will serve to explain this ability.

Suppose a tally must be kept of automobiles stopped by a red light in a particular direction at an intersection, and when this line becomes seven autos long the light must change to green to allow seven cars to go by after which it must change back to red again, etc.

The solution may be represented diagrammatically in a form similar to figure B3-3. The arrows indicate the direction of logic flow in the block diagram. START is the entry point, and all of the rectangular blocks contain arithmetic or process type instructions. The oblong block contains a question: "IS NUMBER OF AUTOS = 7?" The decision offers two alternatives: a true (YES) and a false (NO); if false, flow returns to the previous block; if true, flow moves forward to the next block.
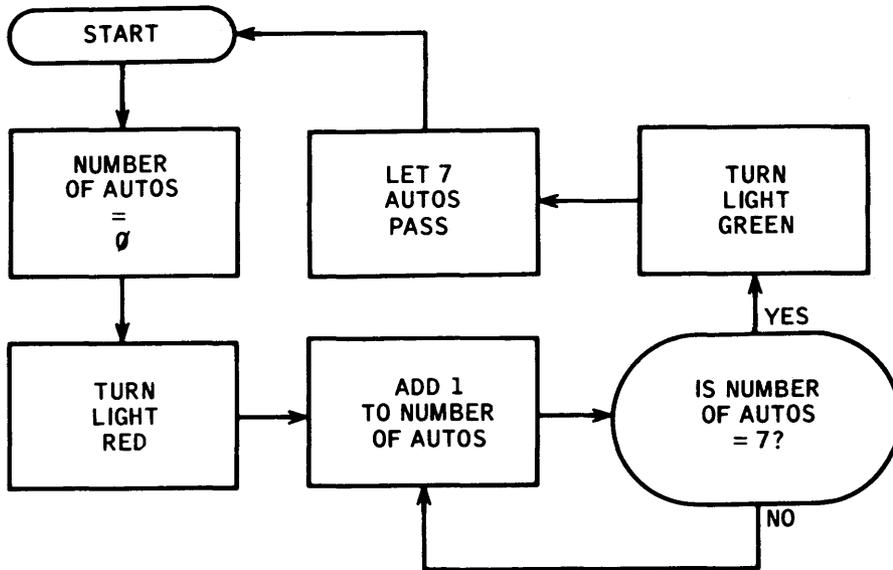
Figure B3-3. Block diagram solution of traffic problem.

When NELIAC is used to satisfy the requirements of the problem, the block diagram may be followed quite closely as in the following example:

```
5
NUMBER OF AUTOS,
RED = 1, GREEN = ∅,
LIGHT,
$
start ''
∅ =) NUMBER OF AUTOS,
RED =) LIGHT,
no ''
NUMBER OF AUTOS + 1 =) NUMBER OF AUTOS,
NUMBER OF AUTOS = 7 :  YES.    NO.
yes ''
GREEN =) LIGHT,
NUMBER OF AUTOS - 7 =) NUMBER OF AUTOS,
START.

. .
```

Notice that the words "start," "no," and "yes" -- the verbs-- are in lowercase letters. Notice also that without these verbs there would be no point of reference to which to proceed after an alternative (NUMBER OF AUTOS = 7: YES. NO.) or at the end of the program (START.). The NELIAC language does not provide arrows per se as the block diagram does; in order to provide a logical flow from program segment to program segment, the NEL- IAC language equivalent of arrows -- the verbs -- are used.

In the formation of a verb, all of the rules pertaining to the NELIAC name apply. A name may not be a noun and a verb simultaneously. Name precedence is especially important when the same verb is used in several flowcharts and subroutines of one program.

A verb is used only in program logic since it is a tag referenced in actual program execution. A verb does not appear in dimensioning since it is considered to be defined by that which immediately follows its double apostrophe. The verb generates no instructions; like the noun, it is placed on a list where it is assigned an address, and this address becomes the location of the first executable instruction of the routine it identifies.

If a verb is used at the very beginning of a flowchart, the entire flowchart is identified by that name. If a verb is used at the very end of a flowchart (verb followed by two periods) it denotes the stopping point of the program. Note that the preceding NELIAC example pertaining to the traffic problem had no stopping point; the solution was a continuous loop.

Program Logic

The organization of this text dictates a short discussion of program logic at this point, before its parts are discussed in greater detail.

All of that which follows the dollar sign of dimensioning and precedes the double period which denotes the end of the flowchart is program logic. Program logic is made up of arithmetic operations of all types, subroutines, functions, control logic, and some other programming tools remaining to be discussed.

Program logic is the algorithm of the programmer in quest of a solution to his problem. The compiler accommodates the programmer by enabling him to specify his algorithm in algebraic formulae and in near-English statements.

In the remaining sections, the components of program logic will be examined.

4. ARITHMETIC OPERATIONS

INTRODUCTION

Previous sections have covered the necessary preliminaries to writing a computer program. This section introduces the actual process of preparing a program which will command the computer in mathematical and near-English language to manipulate data in the manner specified. As the topics concerning program logic are covered, it will be seen why a problem-oriented, algebraic language provides all the tools needed to solve a scientific problem no matter how complex.

NELIAC requires a programmer to be as precise in his specifications as with any mathematical regimen. For instance, in the area of arithmetic precedence, a wide difference can develop between the arithmetic operation as intended by the programmer, and the expression as written, translated, and executed. Neither the computer nor the compiler can differentiate between the intention and the operation as actually programmed.

Programming is an exacting task. But the job can be made easier by learning the rules thoroughly and applying them consistently, and these next sections have been prepared with this end in mind.

As most everyone at some time or other has learned, the quantities of interest in an algebraic expression are called

operands and the symbols that connect them are known as operators. An operand or an assemblage of operands and operators forms an expression, and one expression set equal to another expression constitutes an equation.

NELIAC uses essentially identical terminology: the operands are the constants and variables of all forms; the operators (with concessions to the card handling hardware) are the familiar addition, subtraction, multiplication, division, and exponentiation symbols; and the expressions are the endless permutations and combinations of the operands and operators.

In NELIAC, however, the sense of equality between two expressions is avoided. This evasion is offset by provision in the language for an operation more powerful than and inclusive of the equation: the "store" operation. The result of a series of calculations in a NELIAC expression is stored in an area of core reserved by a variable. Instead of an equation, NELIAC uses the term "statement." There is a small but important difference: implicit in the store operation is the ability to replace an old value stored in the variable with a new one, allowing the calculation of the new value to be completed without creating a condition of inequality.

For example, in some areas of NELIAC programming a statement of the form "SUM + 1 =) SUM," is very useful. This statement says, in essence, take the old value assigned to the variable SUM, add 1 to it, and store the new value back in SUM (replacing the old value, of course). If SUM were preset to zero, upon execution of this statement the left-hand operation would yield a total of 1, and SUM would be reset to the value 1. Notice that if this had been an equation instead of a statement, at completion of the execution of the operation the left-hand expression (value of 2) and the right-hand variable (value of 1) would now be equal--an impossibility.

The term "equation," therefore, is avoided in the discussion of NELIAC program logic. An expression and a variable joined by the store symbol constitute the "statement."

EXPRESSIONS

Symbols and Their Use

As already mentioned, expressions are an aggregate of algebraic parts known as operands and operators. Operands as used in a NELIAC expression are simply the variables and constants the formation and grammar of which were investigated in the preceding section 3. Operators, or operation symbols, are now introduced formally. These symbols and their meanings are shown in table B4-1.

TABLE B4-1. NELIAC ARITHMETIC OPERATION SYMBOLS

SYMBOL	TITLE	OPERATION
+	Plus	Addition
-	Minus	Subtraction
*	Asterisk	Multiplication
/	Slant	Division
*2** /2**	Scale-up Scale-down	Exponentiation

The remainder of this section will define the arithmetic operations of NELIAC in order to dispel any confusion between the contents of a variable and its address. A shorthand notation--c( ), which cannot be used in a NELIAC program--is employed in the specification of an operation's meaning. c( ) refers to the contents of the variable at the instant considered.

The addition A + B is defined as the contents of the computer word(s) identified by the variable A added to the contents of the

computer word(s) identifed by the variable B. In shorthand this is c(A) + c(B). The subtraction, A - B, means c(A) - c(B).

The asterisk and slant--multiplication and division, respectively -- are concessions to the card handling equipment associated with the digital computer. No commonly used symbol for multiplication is distinctive enough on the keypunch, and consequently the asterisk has been substituted. Because of the restriction that all NELIAC statements must be written on the line and not above or below it, divisions have been reformatted and the ÷ symbol is replaced by the slant. A * B is defined as c(A) · c(B); A/B is interpreted to mean $\frac{c(A)}{c(B)}$ or c(A) ÷ c(B).

Exponentiation in NELIAC is a little more difficult to understand. Since the compiler has been designed primarily for a binary computer, exponentiation is thought of in terms of shifting the contents of a variable so many binary digits to the left or right (to the left makes it a larger number; to the right, a smaller one). It can also be thought of as scaling, a subject covered in section 3 in the discussion of mode. Scaling up by a power of two is equivalent to multiplying the operand by 2^n; scaling down by a power of two is equivalent to the operation "operand ÷ 2^n."

Because of the binary influence, exponentiation in NELIAC may be carried out only in powers of two. The expression A * 2 ** B means the contents of A are shifted left (increased in magnitude) c(B) binary digits. That is, c(A) are scaled up by $2^{c(B)}$, or c(A) · $2^{c(B)}$. The expression A / 2 ** B is the converse; the contents of A are shifted right (decreased in magnitude) c(B) binary digits, or c(A) are scaled down by $2^{c(B)}$, or c(A) ÷ $2^{c(B)}$.

Formation of Expressions

The spectrum of possible operand and operator combinations is bounded only by the inventiveness and needs of the programmer. There are, however, some basic rules which pertain to all expressions, and these should be well understood before even the first program is written.

A single operand expression is often employed to preset a partial word variable, to store the contents of an index register somewhere in core, or to duplicate the results of a calculation in another computer word with a separate name.

The operand may be a single NELIAC constant or a single NELIAC variable. If the operand is a variable, the variable may be whole word, half word, or partial word, subscripted or nonsubscripted. No negative constant may be considered an expression by itself. To employ such a constant for any purpose, a leading zero must be supplied; e.g., $\emptyset.\emptyset$ -3.2 =) DO. The mode of the constant or whole word variable may be fixed point or floating point; for half word and partial word variables the mode must be fixed point. The use of a single index register variable as an expresion is fully valid.

The complexity of an expression may be increased by including several variables and constants and introducing arithmetic symbols to indicate intended computation.

With complexity comes the possibility that the programmer will err in an area which to now may have seemed unimportant: mode. There is a rule of NELIAC which prohibits the mixing of modes in an expression. This type of error occurs in an expression when a floating point variable or constant and a fixed point variable or constant are joined by an operator; e.g., I + 3.127, or APRON / 3, when APRON has been dimensioned as a floating point variable.

There are several exceptions to the ordinary rules of mathematical notation. First, in ordinary arithmetic AB can be interpreted as A · B or A times B. However, in NELIAC, AB can never be equivalent to A * B; if multiplication is intended, the appropriate symbol may not be omitted. Such an omission forces the NELIAC processor to interpret AB as a new name.

Second, in common usage expressions like A ÷ B · C and A ÷ B ÷ C are considered ambiguous. Such expressions are permitted in NELIAC and they are interpreted as follows:

A / B * C is the equivalent of (A / B) * C

A * B / C is the equivalent of (A * B) / C

A / B / C is the equivalent of (A / B) / C

Third, it is impossible to correctly write in NELIAC two operators consecutively. For example, the expression A * -B, where a programmer intended to change the sign of B before multiplying by A, may be perfectly good algebraically, but is unacceptable grammatically in NELIAC. To change the indicated expression to proper form, the programmer needs only to enclose the -B in parentheses; i.e., A * (-B).

Irrespective of any of the rules of arithmetic precedence, the use of parentheses in NELIAC gives the enclosed operation precedence over the surrounding operations in an expression. A + B * 2 ** C / D might rightfully be interpreted in mathematical circles as $A + B \cdot 2^C$ all divided by D; if the programmer had intended $(A + B) \cdot 2^{C/D}$, he should have written in NELIAC, (A + B) * 2 ** (C / D). This change forces the computation of the addition and the division before any consideration of exponentiation. Parentheses, then, provide a grouping symbol, a means of treating the enclosed operators and operands together as a whole. The symbol itself may be used frequently for it generates no code. In fact the variable ENEMY may be well-enclosed in parentheses as (((((ENEMY))))); such a use only causes a lengthened compilation time while the processor wades through the chaff. Note that parentheses always come in pairs. Another important consideration is that the parenthesis, in addition to the subscript and the exponent, as shall be seen, has no effect on the mode of the arithmetic operation it surrounds.

Rules regarding the use of exponentiation are fairly straightforward. When scaling or shifting in NELIAC, the operand following the double asterisk must be a positive fixed point constant, or a whole word, half word, or index register variable containing a positive fixed point number. The reason for this restriction is

obvious when one considers the impossibility of shifting the contents of some variable a fraction of a binary digit. (The positive requirement on the exponent is made because of the fact that positive and negative exponents are already designated by the appropriate exponentiation symbol: scale-up or scale-down.) The "leading" operand, which precedes the scale-up or scale-down symbol, must be a fixed point variable or a parenthesized operation of the fixed mode.

Following are a synopsis (table B4-2) and a list of examples (table B4-3) of expressions:

TABLE B4-2. CONSTITUENTS OF EXPRESSIONS

1. Format:
   a. Single operands or several operands separated by operators
   b. Single operands: Positive constants or whole word variables, fixed or floating point, subscripted or nonsubscripted; half word or partial word variables, fixed point, subscripted or nonsubscripted; index register variables
   c. Several operands: Any combination of the above operands and appropriate operators

2. Error-prone areas:
   a. Operands: Cannot mix modes within an expression
   b. Operators: Cannot be presumed; must be specified
   c. Operators: May not be written consecutively

3. Parentheses: Used as a grouping symbol to force arithmetic precedence; can be used in pairs freely

4. Mode:
   a. Parentheses: No effect on mode of enclosed operation
   b. Exponent: No effect on mode of leading operand in exponentiation

5. Exponentiation:
   a. Leading operand: Fixed point variable or a fixed point parenthesized operation
   b. Exponent: Positive fixed point constant or whole word, half word or index register variable containing positive fixed point numbers

a.	3, + 14∅6.29, + ∅.∅2 * 7, ∅.∅ - 7.461,	legal; single constant expressions
b.	TRAC, L, MU'CK ($ 6,5 $),	legal; single variable expressions
c.	ITSELF (8 =) 23), ST (15 =) 29),	legal; partial word expressions
d.	ENTRY + 44, I + ZERO * (9 - UPTURN),	legal; fixed point expressions
e.	-3.∅79 * AZJ ($ 6 $), 1.∅ - GLE * 2 ** 7,	legal; floating point expressions
f.	HERBS + HOMINY/OIL - (VOX / 2 ** PZP),	legal; either fixed or floating point depending on operands
g.	I * 2.∅,	illegal; mixed mode, I must be fixed point
h.	1∅ * - A,	illegal; consecutive operators
i.	(A * (X) + 1.∅/(Y + 3.∅),	illegal; missing right parenthesis
j.	7DRAFT	illegal; incorrect name or missing operator
k.	(1.8∅2 * G) / 2 ** K	illegal; leading operand is floating point
l.	- 12.8	illegal; zero must precede minus sign

Arithmetic Precedence

The operations contained within a NELIAC expression are executed in an order prescribed by three conventions of precedence: the parenthetical convention, the hierarchy table, and the left-to-right rule. These conventions are in order; an operation falling within the realm of parenthetical precedence is executed before an operation governed only by the left-to-right rule.

The left-to-right rule is the most basic of the conventions.
In the absence of parentheses and in the case of identical operations within an expression (as in a string of simple additions), the expression is executed operation-by-operation from the leftmost operand to the rightmost operand. There can be no more basic a convention and yet have expressions make any sense.

If the operations are unlike, but the expression again contains no parentheses, the hierarchy table (table B4-4) determines precedence of execution. The hierachy is composed of the five types of operations: addition, subtraction, multiplication, division, and exponentiation.

TABLE B4-4. HIERARCHY TABLE

OPERATION	PRECEDENCE
Exponentiation	High
Multiplication, division	Middle
Addition, subtraction	Low

To interpret table B4-4, an example will be considered. For illustration purposes intermediate results will be stored in elements of a list identified by TEMP, and the final result in the variable ANSWER.

The expression:

$$A + B - C * D / E * F + G * 2 ** 4$$

Order of execution:

1. G * 2 ** 4 =) TEMP ($ 0 $), (exponentiation before all else)

2. C * D =) TEMP ($ 1 $), (in a string of multiplications and divisions, which are of equal precedence, the left-to-right rule applies)

3. TEMP ($ 1 $) / E =) TEMP ($ 2 $),

4. TEMP ($ 2 $) * F =) TEMP ($ 3 $),

5. A + B =) TEMP ($ 4 $), (next lowest precedence;
    several operations of equal precedence, so left-to-
    right rule prevails again)

6. TEMP ($ 4 $) - TEMP ($ 3 $) =) TEMP ($ 5 $),

7. TEMP ($ 5 $) + TEMP ($ 0 $) =) ANSWER,

The most powerful of the precedence schemes is the paren-
thetical convention. Any operation or series of operations enclosed
between parentheses will be executed before an operation governed
only by the other two conventions. For example, in the expression

$$A * (B - C) / 2 ** D,$$

the subtraction, exponentiation, and multiplication are carried out
in that order.

It is in the area of arithmetic precedence that the differences
between the expression as written and the expression as intended
occur. If NELIAC programs are written to conform to the three
conventions, there are no differences in meaning at execution time.

ARITHMETIC STATEMENTS

NELIAC grammar prescribes that every series of arithmetic
operations must end with the storing of the computed result in a
variable of some type. The only exception is found in the condition-
al transfer which involves alternatives to be chosen on the basis of
the calculation; no storage is necessary for such transfers. A dis-
cussion of conditional transfers is contained in section 5.

Store Symbol

This symbol has in effect been introduced in several places, specifically in partial word variables (indicating the connector "through"), and generally in various examples. The symbol is composed of an equal sign followed by a right parenthesis: =). It is intentionally designed to approximate the right-directed arrow of the Flexowriter NELIAC system.

The consequence of the store symbol between an expression (no matter how complex) on the left and a variable on the right is the storage of the result of the arithmetic operations indicated within the expression into the computer word(s) identified by the varible.

Formation of Statements

An expression becomes a statement with the addition of the store operator and operand. The operand may be any NELIAC variable dimensioned large enough to contain the answer. In the event the result overflows the variable, it is truncated in a manner dictated by the variable type: most significant digits truncated for fixed point, and least significant digits missing for floating point.

The store operator may be included within the body of an expression, permitting the storage of intermediate results. For example

$$A + B =) C * D / E =) F,$$

causes the sum $A + B$ to replace the contents of C and the final result to be stored in F. Notice that there are three expressions joined by the store symbols and two store operands, C and F. C plays a double role: an operand in an expression and a store operand. There is no practicable limit to the number of included store symbols.

The store operator may also be written several times at the end of a statement, indicating a multiple store operation, as in:

THIRD + PERSON =) HE =) SHE =) IT,

This results in the saving of the machine code necessary to store the result separately in all three operands.

A statement is terminated by a comma or its equivalent. Normally the punctuation immediately follows the store operand, but in the case of a multiple store operation, the last operand is followed by the comma. The word "equivalent" is used to indicate that in operations other than arithmetic, such as control, the punctuation indicating the end of a statement is not the comma (which will be explained in section 5).

The final rule of formation concerns itself with mode. Allied with, but contrary to the rule forbidding the mixing of modes in expressions, a programmer may mix modes in a statement, provided that each side of the store symbol is of only one mode. Consider an expression of the fixed point mode and a floating point store operand. The store operation consists of two parts: conversion of the fixed point result to a floating point notation, and storage. In a statement with a floating point expression and a fixed point variable to the right of the store symbol, conversion again is necessary. This time the fractional portion of the number is cut off, and the integral number stored. To illustrate this, two statement examples are given:

    a.   XQ and FLPTV are dimensioned as floating point variables.

    b.   S2U and LINK are fixed point.

        Statement 1: S2U / LINK =) XQ,
          If S2U equals 9 and LINK equals -7, XQ = -1.∅ after execution.

        Statement 2: XQ * FLPTV =) LINK,
          If XQ equals ∅.∅∅2 and FLPTV equals + 783.4 * 2, LINK = 156 after execution.

Following are a synopsis (table B4-5) and a list of examples (table B4-6) of arithmetic statements.

TABLE B4-5. CONSTITUENTS OF ARITHMETIC STATEMENTS

a. Valid NELIAC expression
b. One or more store operators
c. Store operands, one per store operator
d. Any included expressions
e. Terminal comma

EXPRESSION =) STORE OPERAND ($ ∅ $)
             =) STORE OPERAND ($ 1 $)
                 + INCLUDED EXPRESSION
             =) STORE OPERAND ($ n $),

f. Pitfalls:
Arithmetic overflow (result larger than space reserved by store operand) and/or wrong mode (unintended)

TABLE B4-6. EXAMPLES OF ARITHMETIC STATEMENTS

a.	YZERO =) RESULT,	legal; two single variables
b.	BSQUARED - 4 * A * C =) TERM,	legal; expression and operand
c.	5 =) I =) L + 3 =) M,	legal; multiple store and included expression
d.	X ($ 1 $) / 4 =) A, =) B,	illegal; comma after last operand only
e.	-723.15 =) LAP	illegal; zero must precede minus sign
f.	86215 =) M,	illegal; arithmetic overflow (M being an index register 15 bits long)

5. CONTROL OPERATIONS

INTRODUCTION

Most digital computers are sequential by nature; that is, they are designed to execute machine language instructions in consecutive addresses in core. It is the function of the control section of computer circuitry to oversee sequential operations. An execution of this type will begin at a specified entry point and proceed through all of core unless otherwise directed.

It is not always the intention of the programmer that execution be sequential, and provisions are made in the manufacturer's machine instruction set for leaving the mainstream of the program either temporarily or altogether in favor of another program segment. These departures from sequential operations also are an executive function of the control section.

This section concerns the NELIAC language equivalents of those machine instructions which permit nonsequential operations.

UNCONDITIONAL TRANSFER

Jump

When the programmer wishes to depart from the mainline program regardless of consequences and to neglect all machine conditions attendant to previous execution, an unconditional transfer may be used to move program execution to another location in core. Such a transfer involves no qualitative or quantitative tests; the jump is made and operations resumed in a new program segment.

The period, a mark of punctuation, is employed in the NEL-IAC language to indicate an unconditional transfer or jump command. The location of the NELIAC statement next executed is designated by the verb specified just before the period. The verb in the jump command must not be part of an arithmetic operation. The instruction, ENTER. causes an unconditional jump to the verb with the same name. Within the definition of the verb, the first executable operation is then performed.

This form of unconditional transfer is also known as a "direct" jump, in the sense that there are no intervening operations. It is also characterized by the fact that the execution consists of a single transfer.

The following example -- repeated from section 3 -- contains three verbs and three unconditional transfers: "start," "no," and "yes." Study this example again with the added knowledge of jumps.

```
5
NUMBER OF AUTOS,
RED = 1, GREEN = ∅,
LIGHT,
$
start ' '
∅ = ) NUMBER OF AUTOS,
RED = ) LIGHT,
no ' '
NUMBER OF AUTOS + 1 = ) NUMBER OF AUTOS,
NUMBER OF AUTOS = 7  :  YES.  NO.
yes ' '
GREEN = ) LIGHT,
NUMBER OF AUTOS - 7 = ) NUMBER OF AUTOS,
START.

. .
```

Return Jump

The NELIAC language provides another unconditional transfer, this one of a slightly different nature. The return jump, as it is called, is actually two jumps in one instruction. Before the execution of the first jump, the address of the last command performed is noted, and control is transferred to a special program segment known as a subroutine. A subroutine provides a set of operations applicable to computational requirements at several points in the program (further explained in section 6). Upon completion of the operations specified within the confines of the subroutine, another jump is executed back to the area of the departure point and specifically to the address of the next executable instruction (a location based upon the address noted) in that portion of the program.

To indicate a return jump to a subroutine, the name of that subroutine followed by a comma is written into the program. The subroutine name must not be part of an arithmetic operation. Consider the following flowchart:

```
5
A, B, C, X, Y, Z
$
A + B =) C, COMPUTE, WHOA.
COMPUTE ' ' BEGIN A + C =) X * B =) Y =) Z, END ,
WHOA ' '
    . .
```

In this flowchart the return jump command "COMPUTE," forces the computer to execute those operations between the BEGIN and END punctuation symbols and return to the command "WHOA." The latter is an unconditional transfer to the corresponding verb at the end of the program; an instruction to halt execution is implied by the definition of WHOA.

CONDITIONAL TRANSFER

The jump and return jump allows the computer no opportunity to exhibit its logical powers; the matter is predetermined and the command unconditional. However, the next level of sophistication to be studied -- the conditional transfer -- does provide this opportunity.

In brief, the conditional transfer presents the computer with a comparison statement to evaluate and forces it to make a logical choice between two alternatives, a "yes" or true option and a "no" or false option. The alternative chosen specifies the location of subsequent program execution. At no time will both alternatives be selected simultaneously.

Symbols

The remainder of the NELIAC character set is shown in figure B5-1 with a list of usages and meanings. (See figure B3-1 for other symbols.) Note that either form of the symbol meaning "equal to" is acceptable and that all symbols must be set off from operands by spaces. Caution is advised for those who use the Boolean characters without some understanding of Boolean algebra.

Formation

The structure of the conditional transfer may be expressed as follows:

COMPARISON STATEMENT '' ALTERNATIVE 1 $
ALTERNATIVE 2 $

The comparison statement is like an arithmetic statement except that the store operator is absent, and its place is taken by any of six comparison symbols. There are two basic forms of comparison which may be written in NELIAC: 1) A ? B, where the question mark is one of the symbols EQ, NQ, LS, GR, LQ, or GQ; and 2) A LS B LS C, a statement with two LS symbols permitting a "between limits" comparison. A, B, and C are intended to represent arithmetic expressions.

	CHARACTER	USAGE	MEANING
COMPARISON SYMBOLS	EQ or =	A = B	A EQUAL TO B
	NQ	A NQ B	A NOT EQUAL TO B
	LS	A LS B	A LESS THAN B
	GR	A GR B	A GREATER THAN B
	LQ	A LQ B	A LESS THAN OR EQUAL TO B
	GQ	A GQ B	A GREATER THAN OR EQUAL TO B
CONNECTORS	AND	A AND B	A ∩ B (BOOLEAN)
	OR	A OR B	A ∪ B (BOOLEAN)

Figure B5-1. NELIAC comparison symbols and connectors.

In the first form, A and B, as expressions, may be combinations of arithmetic operators and operands of almost indefinite length. Each expression must be of a single mode, and the modes of A and B must be identical but may be either floating or fixed point. All of the rules of arithmetic expression formation are applicable.

The second form is more limited in one sense; expressions A, B, and C are restricted to the fixed point mode. In its use, A LS B LS C makes up for its inflexibility in formation because this one comparison does the work of two. It provides the capability of checking whether or not the value represented by expression B lies between the values of A and C, where A is the lower limit and C the upper limit.

Boolean connectors are employed to join from two to sixteen comparison statements into one large comparison statement. Depending upon the sense of the connectors, the large comparison statement must as a whole be true in order to select the first alternative; otherwise, it is false, and alternative 2 is chosen. Connectors of one type or the other may be utilized in each statement; a combination of ANDs and ORs is an illegal formation. The individual statements of the large comparison statement may be of either mode.

Boolean connectors serve an important purpose in that they provide expanded decision capabilities. The comparison statement

A LQ B AND C NQ D

asks if A is less than or equal to B and if C is not equal to D; both comparisons must be true for alternative 1 to be chosen. If either or both is false, alternative 2 is selected.

Consider another comparison statement:

A GQ G OR L EQ 3

This asks if A is greater than or equal to G or if L is equal to 3, or both. In other words, if either comparison or both is true, control is transferred to alternative 1. If both comparisons are false, the execution moves to alternative 2.

Following are a synopsis (table B5-1) and a list of examples (table B5-2) of comparison statements.

TABLE B5-1. CONSTITUENTS OF COMPARISON STATEMENTS

Forms:
a. A ? B, where ? is EQ, NQ, LS, GR, LQ, or GQ
b. A LS B LS C

Particulars:
a. In forms a and b, preceding, A, B, and C are arithmetic expressions of indefinite length
b. In form a, A and B must be of a single mode, either floating or fixed point
c. In form b, A, B, and C must all be fixed point expressions

Boolean connectors:
a. Join two to sixteen comparison statements
b. Either ANDs or ORs may be used, but not a combination

TABLE B5-2. EXAMPLES OF COMPARISON STATEMENTS

a.	DIV3 * 14.3 LQ FALL	legal; form a, floating point
b.	LOW LS RISK LS 9 + HIGH	legal; form b
c.	1/3 GQ HOP * (3 + 5 * D)	legal; form a, fixed point
d.	ANTE LS 5∅	legal; forms a and b joined by
	AND ODDS GR ∅.5∅	Boolean connectors
	+ JFACTOR	
	AND I LS N LS M	
e.	K GQ -13.6∅2 + BLU	illegal; mixed mode
f.	NORMAL LS 96.4 LS HOT	illegal; floating point mode
g.	AB OR KANGAT	illegal; improperly used Boolean connector
h.	FLOW EQ 6.2	illegal; combination of AND
	AND B EQ -4	and OR
	OR DAP GQ X48	

In the structure of the conditional transfer, the comparison statement is followed by a double apostrophe signalling the beginning of the alternatives.

Alternative 1 and alternative 2 follow identical rules of formation; if an alternative is described, the discussion concerns both. An alternative is that set of control statements or combination of arithmetic and control statements to which the computer moves after evaluation of the comparison statement. All instructions found in an alternative are executed. In the absence of any instructions or in the absence of a direct jump command as the last instruction of the alternative, the first operable NELIAC statement beyond the conditional transfer becomes the next instruction executed. In other words, it is not necessary to specify a direct jump in an alternative if sequential operations are intended.

Alternative 1 is the "yes" or true option and alternative 2 is the "no" or false option. The alternative selected depends upon the evaluation of the comparison statement during program execution.

In general, a dollar sign terminates an alternative. Within the body of an alternative, a return jump is specified with a verb and comma and, after the return jump, execution continues inside the alternative. However, when a return jump to a subroutine is desired at the end of an alternative, the dollar sign serves as the punctuation necessary to indicate such a transfer; i.e., the comma is not used. In this case, after the subroutine, execution will return to the first operable NELIAC statement beyond the conditional transfer.

If a direct jump is specified in an alternative, it is interpreted as the last executable instruction in that alternative. In other words, the dollar sign is replaced as the terminal punctuation for that alternative by the period associated with the jump. Care must be exercised with the use of direct jumps in conditional transfers to prevent the unintentional completion of alternatives.

Following are a synopsis (table B5-3) and a list of examples (table B5-4) of alternatives.

TABLE B5-3. CONSTITUENTS OF ALTERNATIVES

Forms:
a.   Blank
b.   Control statements
c.   Control and arithmetic statements

Particulars:
a.   If form a alternative is chosen, control is transferred
     to first NELIAC statement beyond conditional transfer;
     $ terminates alternative
b.   If form b or c alternative is chosen, and last instruction
     of alternative is return jump, after execution of sub-
     routine, control is transferred to first NELIAC state-
     ment beyond conditional transfer; comma for that return
     jump not needed; $ terminates alternative and replaces
     comma as well
c.   If form b or c alternative is chosen and it contains
     direct jump, the jump becomes last executable in-
     struction of alternative; period (.) terminates alter-
     native

TABLE B5-4. EXAMPLES OF ALTERNATIVES

a.	A = B ' ' $ STOP $	legal; form a, form b
b.	MIX GQ Y ' ' $ X + Z =) G, T.	legal; form a, form c
c.	A = B ' ' 47 $ STOP $	illegal; 47 cannot be a verb
d.	REP LS I LS YOYO ' ' REST + 1 =) F $ WOW. $	illegal; extraneous punctuation in alternative 2

Nested Conditional Transfers

It is often advantageous to substitute an entire conditional
transfer within an alternative of another conditional transfer. In
effect, we can make a decision within a decision within a decision
... etc. The fact that the subsequent conditional transfer is total-
ly enclosed within the confines of an alternative to a preceding com-
parison statement has generated the adjective "nested."

To illustrate the formation of the nested decision, the following conditional transfer is given:

DAY = CLEAR '' ^{①} TEMPERATURE = WARM '' ^{②}
GET UP, GO PICNICKING \$ ^{②T}
GET UP, GO SKIING \$ ^{②F} GO HOME. ^{①T}
STAY IN BED \$ ^{①F}

(Note: "T" = true option; "F" = false option)

The outer comparison statement, identified by the superscript $①$ after the double apostrophe, has two alternatives and superscripts $①T$ and $①F$ (true and false options for decision $①$). Note that the second conditional transfer (superscript $②$, $②T$, and $②F$) is explicitly specified within conditional transfer $①$.

If DAY does in fact equal CLEAR, a second decision is necessary: does TEMPERATURE equal WARM? If it does, the alternative GET UP, GO PICNICKING is selected; if it doesn't, the false alternative GET UP, GO SKIING is chosen. In either case, the direct jump GO HOME. is executed afterwards. Under the circumstances in which DAY is not equal to CLEAR, the false alternative STAY IN BED is executed without consideration of the second comparison statement.

The superscript notation, although not an allowable grammatical form of NELIAC, is often used by programmers to mentally check the nesting of conditional transfers before the source program is punched into cards. Two rules of thumb should be employed in conjunction with the extra-NELIAC notation: first, alternatives always come in pairs (i.e., there must be two symbols of alternative termination -- period or dollar sign -- for each double apostrophe); second, nested conditional transfers must be totally enclosed in one alternative or the other of the more inclusive conditional transfers.

A maximum of fifteen conditional transfers may be nested. One may anticipate the complexity and occasional obscurity that accompanies the nesting. For the purposes of understanding and better organization, a programmer may choose to enclose, between the BEGIN and END punctuation symbols, entire conditional transfers when they are a part of or the whole alternative of a preceding comparison statement. Only nested decisions may be so written.

One special rule applies to this added punctuation: if the nested conditional transfer is written so that the last instruction before the END is an unconditional transfer outside both alternatives (as GO HOME. in the last example), the verb specified must be followed by a comma or period, regardless of the fact that a dollar sign outside the END actually terminates the more inclusive alternative.

The following example illustrates a nested conditional transfer using BEGIN and END punctuation symbols:

DAY = CLEAR ' ' BEGIN TEMPERATURE = WARM ' '
                GET UP, GO PICNICKING $
                GET UP, GO SKIING $ GO HOME. END $
                STAY IN BED $

To see the comparative effects of the conditional transfer with Boolean connectors and the nested conditional transfer, refer to figures B5-2 and B5-3. A glance at the two figures shows the double comparison with Boolean connectors is the simpler in scope of the two. With Boolean connectors note that only two options are available; in the alternative ONEORNEITHER. it is impossible to ascertain which comparison held true or if both were false.

Nested conditional transfers are more powerful. Nesting results in three alternatives; two of the three are firm indicators. The third alternative eliminates the early comparison but cannot rule on the second; another comparison is necessary to decide on the appropriate action.

INDIRECT TRANSFER

Indirect Addressing

A noun or verb used in an arithmetic operation will cause the contents of the word(s) specified to be involved in the appropriate calculations. During execution of this operation, addresses are referenced directly and the data contained therein withdrawn and manipulated.

```
A NQ B AND D NQ E "
      BOTH.
      ONEORNEITHER.
```
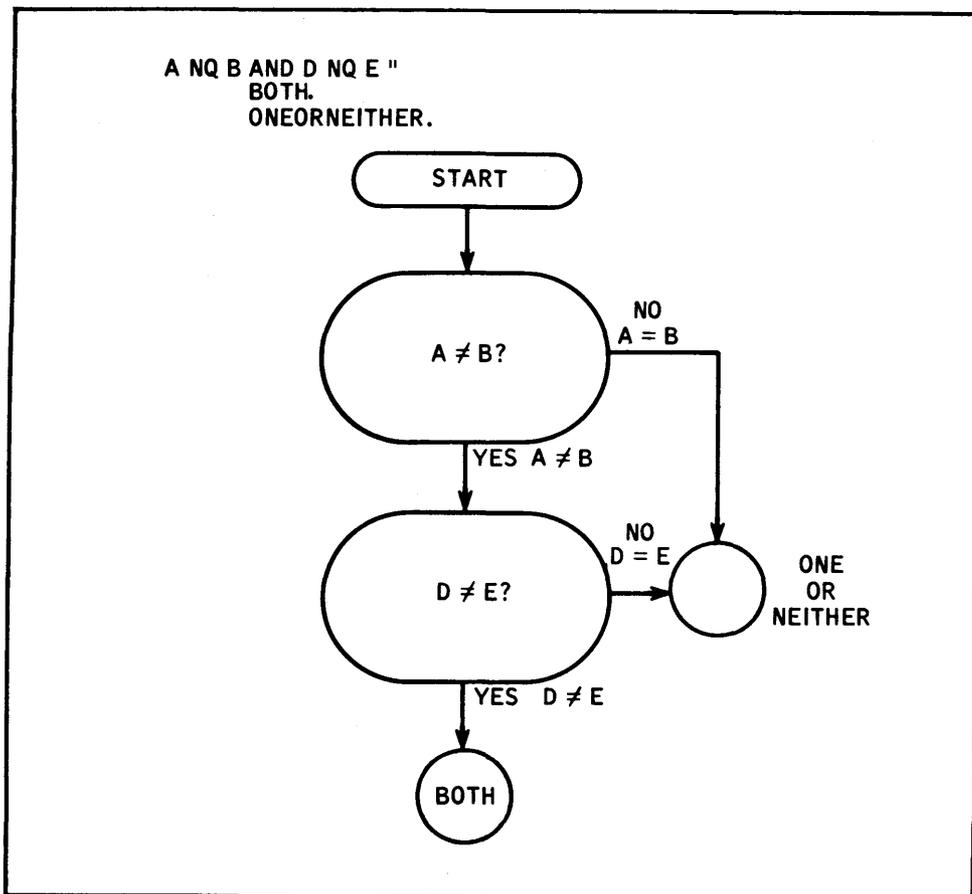
Figure B5-2. Conditional transfer with Boolean connectors.

A control statement of the unconditional transfer form will, upon execution, command the control portion of the computer to jump to the address specified as part of the instruction. Again, the location is referenced directly.

Indirect addressing, as one might expect, is more complex than direct addressing. The address specified in an instruction which implies indirect addressing causes the computer, first, to reference the given location and extract another address contained therein and, second, to involve the information (data or instructions) found at the new location in arithmetic or control operations.

Variables written in program logic as subscripts (enclosed in proper punctuation) such as ($ VARIABLE $), are treated as operands in themselves. This NELIAC form, when incorporated in the

```
A NQ B ' ' D NQ E ' '
        BOTH.  FIRST NOT SECOND.  $
        NOT FIRST.
```
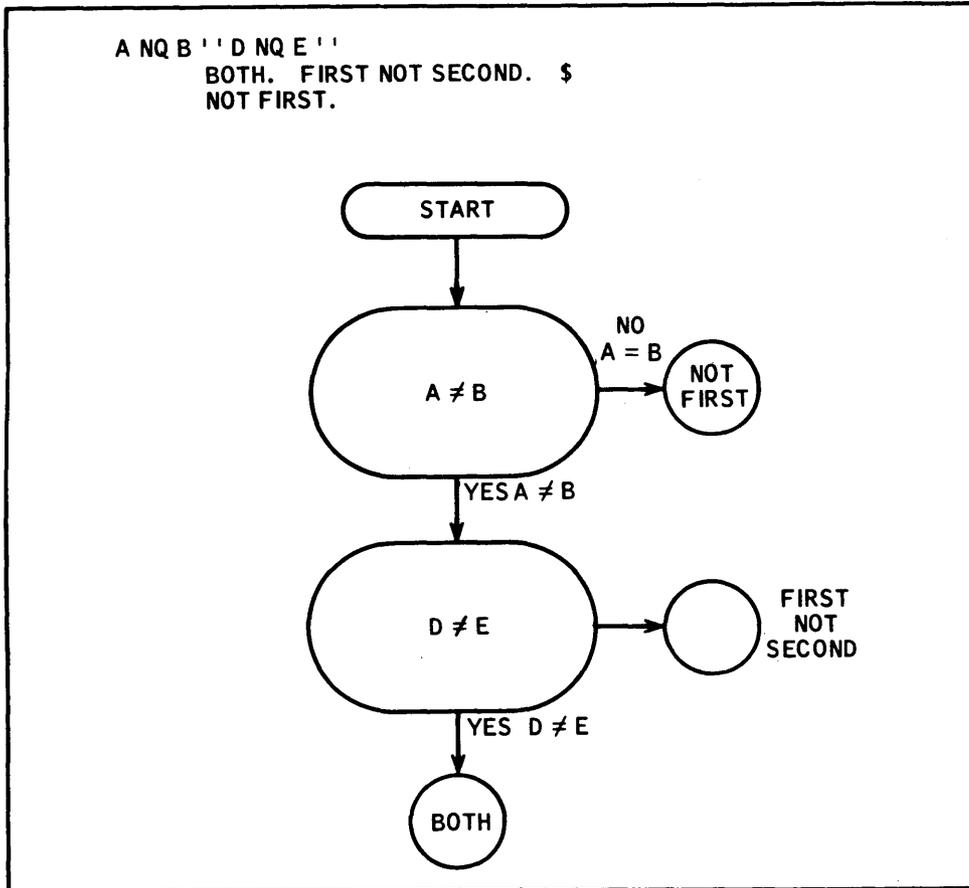
Figure B5-3. Nested conditional transfer.

appropriate instruction environment, infers indirect addressing.
For example, the command ($ CONNECT $). is an indirect trans-
fer; during execution, the computer is to look at the contents of
the variable CONNECT and use the information stored there as the
address to which control is transferred. If CONNECT had con-
tained the address assigned to the variable KEY, ($ CONNECT $).
would have been equivalent to executing the direct jump, KEY.

Note that an unsigned constant as a subscript does not imply
indirect addressing. It is interpreted, rather, as a request for
the contents of a particular word in core. The arithmetic state-
ment ($ J $) + ($ 47 $) =) ($ A $), when executed, would cause the
contents of the address specified in J to be added to the contents of
computer word numbered 47 decimal (or 57 octal) and the sum
stored in the address contained in the variable, A.

Some interesting sidelines are created by the subscript form, and are mentioned only briefly for interest's sake. First, in dimensioning, the statement ABC = BEGIN XYZ END , assigns to the variable ABC an initial value equal to address of the variable XYZ. Second, in dimensioning, the statement A = BEGIN 4$\emptyset\emptyset\emptyset\emptyset$ OCT END , is illegal; one may not specify a particular machine address for a variable since this is a function of the compiler (see section 7). Third, in program logic, only a single level of indirect addressing is allowed; i.e., it is not possible in NELIAC to request the contents of the address specified as the contents of the address specified as the contents of a variable. Fourth, the form ($ v ± c $) may be used (v = fixed whole or half word variable, c = constant); in general, it is defined as the contents of the location identified by the variable plus or minus so many words.

Then why have indirect addressing? The answer is that this feature allows the programmer to change the address contained in a variable like CONNECT from time to time, creating in effect a modifiable, multidirectional switch. The use of this switching feature, through address modification, saves many NELIAC programming steps and consequently execution time.

Switches

In the conditional transfer a bidirectional programming device was uncovered. Based on a comparison statement, a true or a false alternative path was taken. Because conditional transfers are lengthy and sometimes clumsy (when nested), and because a compact list of routine names is often required for specification of many different paths of execution, switches are incorporated in NELIAC.

There are basically two forms a NELIAC switch may take: a noun switch (often called an address switch) and a verb switch (known also as a jump table). The more important of the two is the verb switch, and immediate emphasis will be placed on it.

A verb switch may be defined in either the dimensioning or the program logic portion of a flowchart. It is composed of an identifying name for the switch, and two or more entries in a list or array. These entries are the verbs associated with routines or subroutines. In dimensioning, the verb switch looks like one of these:

a. SWITCH = BEGIN A, B, C, D, END ,

b. QUEUE (4 * 3) = BEGIN

ALFA, ROMEO, JULIET, HOTEL,
ROMEO, PAPA, NOVEMBER, QUEBEC,
BRAVO, CHARLIE, ECHO, WHISKEY,

END ,

If written in program logic, the verb switch must have a format similar to the following:

COUNT ' ' ONE. TWO, THREE, ETC.

No two-dimensional switch may be incorporated in program logic.

When a verb is specified in a switch, there must be a corresponding routine or subroutine with that name; the routine it identifies serves as defintion for the verb. Punctuation separating entries in the list or array may be either the period or the comma; their function in verb switch definitions is as a separator and consequently no differentiation is made between them.

A subscripted verb, followed by a period or comma, in program logic constitutes the command necessary to operate the switch. QUEUE ($ 1, 1 $). in the sample switches will instruct that a direct jump be made to the routine PAPA. COUNT ($ 3 $), forces a return jump to the subroutine named ETC. No execution actually takes place in the verb switch itself. The switch may even be thought of as being passive, since all it does is provide an address indirectly to a jump command. Whether the jump is direct or return is determined by the command punctuation. If a return jump to a verb is specified, that verb must be defined by a subroutine.

The noun switch is a list of names, and both the names and the noun switch must be defined in dimensioning. The noun switch cannot be differentiated in appearance from the verb switch except that it is never written in program logic.

Each noun in the switch represents a machine address. Using an entry of the noun switch one may modify the starting address of a table of values, as in the following example:

```
5
SOLE (2),
CHANGE = BEGIN SOLE END ,
$
CHANGE ($ ∅ $) + 1 =) CHANGE ($ ∅ $),
..
```

This sort of programming task might be required when another list (defined in another flowchart) would overlay the first element of the list named SOLE should this provision not be made. The address of a flowchart to be loaded, while one is already resident, could be added to the resident segment's length to provide a loading address. This sort of information would originate from a noun switch.

Noun switches may also be composed of literal addresses. Literals are lines of English defined in dimensioning which are output on a hard copy device at the request of the programmer. Their primary use is in debugging (making flowcharts error-free) and in printing headings. Noun switches allow the programmer to specify the literals for output in any order he chooses. More will be said about literals in section 8.

ITERATIVE PROCEDURES

Introduction

There are three approaches to programming an algorithm containing a group of instructions which must be repeated several

times. To illustrate the several methods, consider a simple problem; the first five integers are to be stored in a list dimensioned as five elements in length and identified by the variable NUMBER.

The first approach is to write the instructions over and over again in the flowchart until the requisite number of executions is provided for. The straightforward approach to the solution appears below in routine form:

.

.

.

1 =) NUMBER ($ Ø $),
2 =) NUMBER ($ 1 $),
3 =) NUMBER ($ 2 $),
4 =) NUMBER ($ 3 $),
5 =) NUMBER ($ 4 $),

.

.

.

The second approach incorporates the utilization of the conditional transfer to test if sufficient executions have been completed. Note that the important instruction, the storage of the value into the list, is written only once; all of the other instructions are the "master" or the controlling instructions. The second approach follows:

.

.

.

Ø =) I,
ADD ' ' I + 1 =) I,
I =) NUMBER ($ I-1 $),
I $\neq$ 5 ' ' ADD. GO ON.
GO ON ' '

.

.

.

Using NELIAC loop control to replace the master instructions of the second approach constitutes approach number three:

.

.

.

I = 1 (1) 5 BEGIN I =) NUMBER ($ I-1 $), END ,

.

.

.

The number of instructions to accomplish the required computation (including control) decreases from the straightforward to the loop-control approach: five instructions in the first, or straight forward, approach; four in the second, or master-instruction, approach; and one in the third, or loop-control, approach. As the sophistication increases, more of the counting and control function is maintained by programming tools rather than by the programmer. The example of the straight forward approach was written for five executions of the storage instructions; the program length required to store the first 1øø or 1øøø integers would be so immense as to nearly defeat the purpose of using a procedure-oriented language. The other two approaches would need only minor changes and would involve no additional NELIAC instructions.

In spite of the outward efficiency in coding the loop control rather than the straightforward approach, consider again the familiar tradeoff in digital computers: time vs. space. The store instruction, written repeatedly, sans loop control, occupies a larger area of core than the single instruction in either of the other approaches. This space ratio becomes more unfavorable for the repetitive specification as required iterations become more numerous.

On the other hand, the time required to execute the loop control, added to the routine execution time, all multiplied by n iterations, exceeds the length of execution time of the straight-forward approach. The second approach is no less time-consuming.

A decision must be made, and the approach favoring the loop control wins the nod. The reason is simple. In digital

computers space is often more valuable than time. (This does not, however, dictate a policy; the use of one approach over another should be decided by the programming application.)

In the second, or master-instruction approach, the register variable I was preset to $\emptyset$, and in a routine identified by the verb ADD, immediately reset to 1. This value of I was stored in NUMBER ($ $\emptyset$ $). Then a test to see if I $\neq$ 5 was performed. Until the loop was executed five times, the true alternative ADD. was chosen; once the inequality was false, control dropped out of the loop to the verb GO ON. Note that the direct jump to ADD caused I to be incremented, a new value to be stored in the next sequential element of NUMBER, and caused a return to the conditional transfer.

The third, or loop-control, approach reads as follows: take the value 1 and store it in I; execute all instructions found between BEGIN and END; test to see if I is equal to five; if so, drop to next instruction; if not, go back, increment I by 1 (the parenthesized constant) and proceed as before in executing, testing, etc.

Format

The general form of an iterative procedure is

LOOP CONTROL BEGIN PROCEDURE, END ,

where LOOP CONTROL is defined as

INDEX = START (INCREMENT/DECREMENT) STOP

and PROCEDURE is a routine composed of arithmetic and control operations which are to be repeated the number of times designated in LOOP CONTROL.

INDEX may be any of the index register variables, I, J, K, L, M, N. It serves the LOOP CONTROL by providing a register

to act as a counter. During execution of the iterative procedure, the variable specified as an index will contain the value of the count related to each execution and, as shown in the example of the third, or loop-control, approach under the immediately preceding "Introduction" heading, this portion of loop control may be involved in the computation of the PROCEDURE.

START is the beginning value of LOOP CONTROL; it may take any of seven forms:

a. Positive fixed point constant, including zero.

b. Fixed point whole word variable.

c. Fixed point half word variable.

d. Fixed point whole word variable subscripted.

e. Fixed point half word variable subscripted.

f. Index register variable.

g. Index register variable ± fixed point constant (must be in order specified).

INCREMENT/DECREMENT is a mnemonic name for the parenthesized fixed point constant found in LOOP CONTROL. In a situation where the INDEX is to increase in a positive direction from the START limit to the STOP limit, the INCREMENT is positive. To count in a negative direction from START to STOP when START is some positive integer and STOP is zero, the DECREMENT is positive. For a nonzero STOP in LOOP CONTROL, to create a decrementing count, the DECREMENT must be negative.

STOP has the same rules of formation that govern START.

Following are a synopsis (table B5-5) and a list of examples (table B5-6) of iterative procedures.

TABLE B5-5. CONSTITUENTS OF ITERATIVE PROCEDURES

Formats
a. Iterative procedure:
   LOOP CONTROL BEGIN PROCEDURE, END ,
b. LOOP CONTROL:
   INDEX = START (INCREMENT/DECREMENT) STOP
c. PROCEDURE:
   Any routine composed of arithmetic and control
   operations

Particulars
a. INDEX: Any of index register variable, I, J, K,
   L, M, N
b. START: Any of seven forms:
   (1) Positive fixed point constant, including zero
   (2) Fixed point whole word variable, subscripted or
   (3) Not
   (4) Fixed point half word variable, subscripted or
   (5) Not
   (6) Index register variable
   (7) Index register variable ± fixed point constant
       (in order specified)
c. INCREMENT/DECREMENT: Parenthesized fixed
   point constant:
   (1) START ≤ STOP: INCREMENT positive
   (2) START > STOP, STOP = $\emptyset$:
           DECREMENT positive
   (3) START > STOP, STOP ≠ $\emptyset$:
           DECREMENT negative
d. STOP: Same rules as START

TABLE B5-6. EXAMPLES OF ITERATIVE PROCEDURES

a. I = $\emptyset$ (1) 5 BEGIN I =) J, END ,   legal; incrementing loop
b. I = 5 (1) $\emptyset$ BEGIN I =) J, END ,   legal; decrementing loop,
                                        STOP = $\emptyset$
c. I = 6 (-1) 3 BEGIN I =) J, END ,   legal; decrementing loop,
                                        STOP ≠ $\emptyset$

d. J = HARP ($ 5 $) (7) LUTE	legal LOOP CONTROL; whole word variables (one subscripted)
e. M = I - 6 (4) I + 26	legal LOOP CONTROL; index register variables ± fixed point constants
f. L = 98 (-1) PAL (15 =) 29)	legal LOOP CONTROL; half word variable
g. INDEX = N (1) L BEGIN B, END ,	illegal; INDEX is a mnemonic device for describing formats; only index register variables may be used
h. K = 1 (PANG) G BEGIN B, END ,	illegal; INCREMENT/ DECREMENT must be fixed point constant
i. J = 4 (1) 2	illegal LOOP CONTROL; in this loop, INCREMENT/ DECREMENT must be negative to decrement 4 to 2
j. N = I (2) 5 + K	illegal LOOP CONTROL; STOP has constant and variable reversed

Operations

The purpose of this subsection is to familiarize the programmer or programmer-initiate with the inner workings of loop control and with the programming possibilities of this tool.

LOOP CONTROL is a most important consideration when establishing an iterative procedure in NELIAC. The rules of formation have been reviewed; actual operations now will be

described. To simplify the discussion, the steps of execution are enumerated:

1. Set the INDEX = START.

2. Execute the instructions contained in the PROCEDURE.

3. Test if INDEX = STOP.

4. INDEX = STOP: Set INDEX to zero, take first sequential instruction beyond END punctuation.

5. INDEX ≠ STOP: Add INCREMENT/DECREMENT to INDEX and return to step 2.

Note that the STOP or end point of the loop control must be reached exactly; i.e., a multiple of the INCREMENT/DECREMENT added to START must exactly equal STOP. If STOP never equals INDEX, the PROCEDURE will continue to be executed indefinitely.

LOOP CONTROL written as either a decrementing or an incrementing loop with the START and STOP limits interchanged from one case to the other causes no difference in the number of PROCEDURE executions. For example, I = ∅ (1) 15 BEGIN PROCEDURE END , controls 16 executions of the PROCEDURE; I = 15 (1) ∅ BEGIN PROCEDURE END, also forces 16 executions. The reason for one LOOP CONTROL rather than another is the use of the INDEX within PROCEDURE; it is sometimes the desire of a programmer to use a decrementing count in PROCEDURE calculations rather than an incrementing INDEX.

START and STOP limits in LOOP CONTROL must not be altered during iterative executions of the PROCEDURE. The result of such an operation is the deterioration of LOOP CONTROL and the possibility of program failure.

START and STOP limits which are equal will allow a single execution of the iterative procedure since the test for equality is made subsequent to the execution.

Control may be transferred at any time from an iterative procedure. If it occurs at the normal STOP limit, the INDEX is set to zero. If, however, a transfer is made before that point,

the instantaneous INDEX value is saved for later reference, and
will maintain its status unless control is resumed within the loop
or if some arithmetic statement changes the INDEX.

Entire iterative procedures may be enclosed within other
iterative procedures, as in the following example:

I = 1 (2) 5 BEGIN J = 4 (-1) 2 BEGIN I + J =) K, END , END ,

The instruction length of the PROCEDURE, i.e. from BEGIN to
END , is known as "scope." The general rule for nesting iterative
procedures is that the scope of an enclosed iterative procedure
must lie wholly within the scope of the outer iterative procedure(s).
This is illustrated in figure B5-4, whereas figure B5-5 shows an
illegal configuration of iterative procedures. In figure B5-5, the
nested loops are illegal because the scope of iterative procedure
#2 extends beyond that of procedure #1.
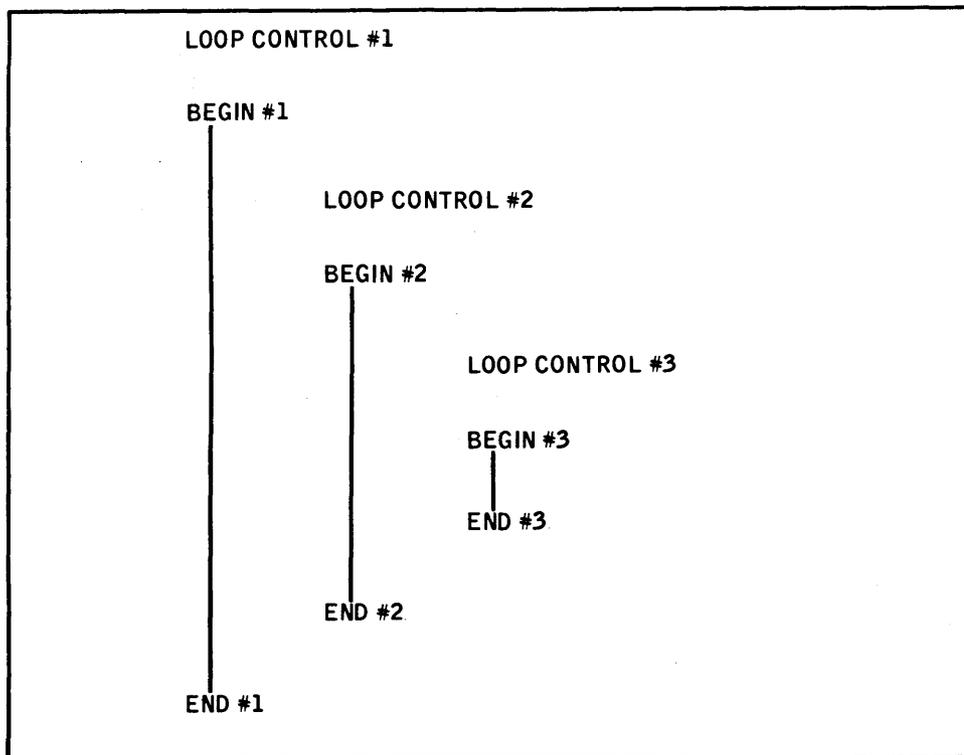
Figure B5-4. Legal nested iterative procedures.
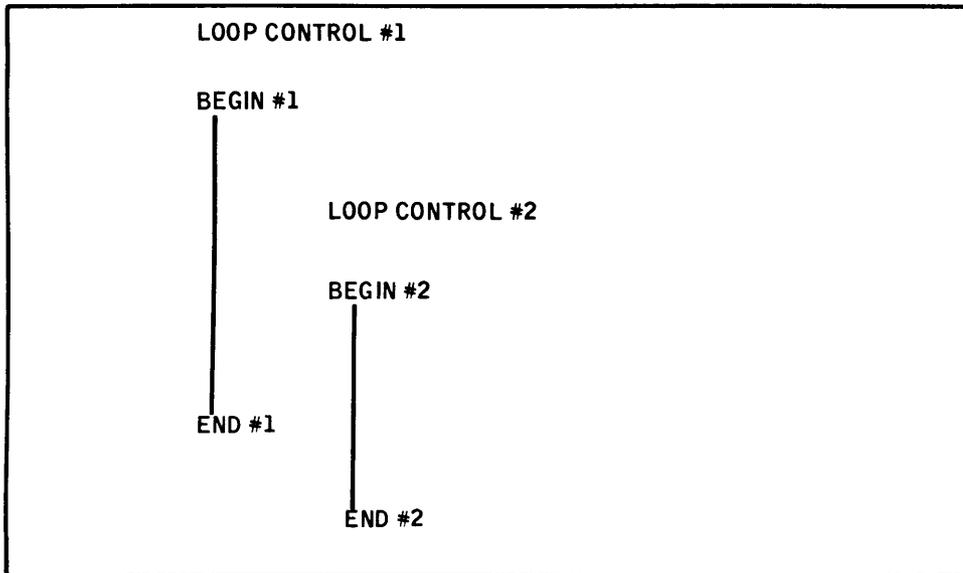
Figure B5-5. Illegal nested iterative procedures.

Another illegal operation is the transfer of control into an iterative procedure from a point outside its scope. For example, a transfer from loop #3 into loop #1 of figure B5-4 might be legally carried out, but the converse would not be permitted. The purpose of this restriction is to prevent program failure or the deterioration of loop control.

Iterative procedures may be nested six deep without any extra programming; six nested loops would utilize every index register variable available. Any attempt to reuse one of these variables before its iteration is complete, without making special provision to save the current value assigned to it, will cause problems in the loop with the modified index. The only extra programming necessary to prevent these troubles is to temporarily store the outer index value in a fixed point variable while the index is reused and then to reinstate this value before operations in the outer loop are continued. For example:

```
        .
        .
        .
I = 1 (3) 4 BEGIN I =) TEMP STORE,
I = 12 (1) Ø BEGIN A + B =) C, END ,
TEMP STORE =) I, END ,
        .
        .
        .
```

6. SUBROUTINES AND FUNCTIONS

INTRODUCTION

A routine is a series of instructions written to match the program requirements and expressed in a specific sequence. When these requirements dictate that the identical routine must be incorporated several times in the algorithm, the source program reflects the inclusion of the indistinguishable instruction sets.

NELIAC provides the programmer with the subroutine and the function to avoid the necessity for multiple programming and flowcharts with their lengthy composition and compilation times. Subroutines and functions are routines written (or defined) once within the confines of the program and referenced (or called) as necessary from any point in the program logic of any flowchart. They are classified as closed routines--the machine code equivalent to these source program segments is inserted only once in the object program in spite of the number of references made in program logic.

In principles of operation there are several degrees of similarity between the subroutine and the function. Each is a self-contained group of commands which will manipulate data and/or cause control operations to take place. Each is defined and called in program logic; definitions and calling sequences may look almost identical. Each may be defined or called from within another subroutine or function.

The differences and similarities of these routines are discussed in the following subsections.

SUBROUTINES

Definition

To create a subroutine from a routine, a programmer needs only to enclose the routine between the BEGIN and END punctuation and assign it a name. In other words, the format is

VERB ' ' BEGIN ROUTINE END ,

A subroutine normally is written toward the end of a program to avoid the necessity of branching around it, since it is not executable unless called by name in a return jump instruction. If its importance or size warrants the separation, the subroutine definition may occupy a flowchart by itself (no dimensioning is needed, provided that all names used in the routine are properly defined elsewhere).

A subroutine definition may include the description of functions or other subroutines. Any verb defined by the included routines is considered of "subroutine or function" precedence (see section 3 for explanation) and therefore may duplicate any other name outside its own routine without compiling errors. Other verbs and all nouns (operands in arithmetic or control statements) called or operated upon within the routine must be externally defined.

Likewise, the definition may incorporate definitions of and calls on any functions or subroutines in the program other than itself.

Calls

The uniquely proper manner for subroutine entry is via the subroutine call; unless such a call is made, the subroutine is non-executable. An unconditional transfer of control in the form of a return jump is the correct form for subroutine calls. Such a call may be written at any point in the program. During execution, the call will force transfer to the subroutine and cause operation of all executable instructions therein contained and an automatic return transfer to the departure point.

A subroutine may be left via an included direct jump at any time; if no return to an inner point in the subroutine is anticipated. If temporary departure from the subroutine, such as a return jump to another subroutine, is desired, the call is the only exit which provides a reentry to that particular routine at the departure point. Calls made within the subroutine on other subroutines defined within the confines of the outer subroutine are legal because of name precedence.

Following are a synopsis (table B6-1) and a list of examples (table B6-2) of subroutines.

TABLE B6-1. CONSTITUENTS OF SUBROUTINES

Definition:
a. Form:
   VERB ' ' BEGIN ROUTINE END ,
b. VERB is any name unique to the more inclusive subroutines; if the subroutine is not nested, VERB is a name unique only to the program
c. ROUTINE is any combination of arithmetic and control statements; other subroutines and functions may be defined or called within ROUTINE

Call:
a. Form: ordinary return jump--SUBROUTINE NAME,
b. Only entry to subroutine
c. May call any other subroutine or function; such a call provides a reentry point
d. Direct jump from subroutine provides no reentry point

TABLE B6-2. EXAMPLES OF SUBROUTINES

a.	TRIVIA ' ' BEGIN A + B =) C, END ,	legal; definition
b.	REDO ' ' BEGIN I * 4 =) J < 14 ' ' TRIVIA$ PART. END ,	legal; definition includes call on subroutine TRIVIA
c.	CRANK ' ' BEGIN CHUG ' ' BEGIN SEAR ' ' BEGIN D, END , END , END ,	legal; definition of nested subroutines; included call on subroutine D
d.	FORT ' ' BEGIN ADDAGAIN, GEO/ GLOSS * 2 ** 9 =) STORE, ADDAGAIN ' ' BEGIN STORE + 1 =) STORE, END , END ,	legal; definition includes call on subroutine nested within itself
e.	BAD ' ' BEGIN BAD, END ,	illegal; definition includes call on subroutine itself
f.	ADDAGAIN, FORT ' ' BEGIN ADDAGAIN, GEO/GLOSS * 2 ** 9 =) STORE, ADDAGAIN ' ' BEGIN STORE + 1 =) STORE, END , END ,	illegal; call external to subroutine on name of subroutine precedence

FUNCTIONS

Discussion

     Subroutines and functions are self-contained groups of commands designed to manipulate data and/or cause control operations to take place. In other words, they are assigned the same basic task, and the programmer is left to choose the proper one for his needs.

In calculus, a parameter is a quantity to which arbitrary values may be assigned. If a dependency is established between two parameters, x and y, it may be said that the value assigned to parameter y is some function of the value given to parameter x or vice versa. In the shorthand of calculus, this dependency is written as

$$y = f(x)$$

The value of the function, $f(x)$, when $x = c$, is $f(c)$.

The names written as a part of the NELIAC function definition are also called parameters in that they characterize the names or values that will be substituted during function execution. The parametric function, as written in the definition, provides the guidelines for name substitution in the call and execution.

A representative NELIAC function definition is written as follows:

P (X, Y, Z $ D) ' ' BEGIN X + Y * Z =) D, END ,

The function P has three input parameters--X, Y, and Z-- and one output parameter--D. The definition consists of the routine enclosed by the BEGIN and END. A function call from within program logic

P (A, B, C $ E),

would cause the following routine to be calculated:

A + B * C =) E,

Note the facility with which the parametric values in the function use can be changed. The function, then, in contrast with the subroutine, manipulates the data in and performs control operations on the names supplied by the call, and not on those names written in the definition. This facility permits the programmer to write a generalized routine and to use the function at several points in the program logic without intermediate store operations to replace the contents of subroutine names.

Definition

Like the subroutine, the function is defined only once in a program, for the calls to it provide the necessary linkages. The function definition is usually placed among the last routines of a flowchart or program in order to isolate it from executable routines and to avoid interruption of sequential operation of the program.

The function definition may be considered a flowchart in itself. The definition may be broken into two parts, one easily identified as the dimensioning portion, the other as the program logic.

Dimensioning in the function definition comes directly after the name of the function (which is a verb, since it provides an entry point for a procedure), before the double apostrophe (signalling the beginning of the routine), and enclosed between a left and a right parenthesis. Within the parentheses the dummy parameters for the function are dimensioned. All parameters in a function definition are of the "subroutine or function" precedence, and consequently the identical NELIAC names may be used as nouns and verbs elsewhere in the program.

Dimensioning in the function definition allows the name forms discussed in section 3--nouns, subscripted nouns, nouns involved in bitfield algebra, modal specification (fixed or floating point), assignment of initial values, and combinations of the above. The nouns are the dummy parameters of the function.

The dimensioning portion of the definition in general looks like

NAME (INPUT PARAMETERS $ OUTPUT PARAMETERS)''

NAME is a verb, as previously mentioned, and the routine to the right of the double apostrophe serves to define it. INPUT PARAMETERS is a list of all those dummy parameters that represent values calculated previous to the call on the function. The list may be as few as one input parameter or as many as desired. OUTPUT PARAMETERS is a list of values calculated within the function's

routine from the input parameters. This list may be nonexistent, in which case the parenthesized parameters are written without the dollar sign, since there is no need to separate IN from OUT. Again, the list may be as long as necessary.

Parameters in either list must include the mode the programmer intends to assign to each. If a parameter is fixed point, it is separated from its fellows by a comma (the last comma before the dollar sign or right parenthesis is superfluous). If, on the other hand, the parameter is written as floating point, a period follows it no matter what its position in either list.

The input and output parameters must be specified in the definition in the same sequence that they are found in the program logic. A closer look at this dimensioning might find

VERB (INPUT PARAM1, INPUT PARAM2 $ OUTPUT PARAM1.)"

The program logic portion of the function definition, which follows the double apostrophe, may contain any of the operations one might find in a normal flowchart: arithmetic and control operations, definitions of and calls on subroutines and other functions, etc. Calls on subroutines and functions defined within the outer function are legal. Variable operands exclusive of definitions and calls are either input and output parameters dimensioned by the function definition, or variables defined elsewhere in the program (external to the function definition).

A function definition, like that of a subroutine, may include the description of subroutines and/or other functions. Any verb defined by the included routines is considered of "subroutine or function" precedence and therefore may duplicate any other name outside its own routine without compiling errors.

Under the topic of function calls, we will come to the use of the function in an arithmetic statement. For most occasions of such a use it is necessary to insure that intermediate results in the statement, derived from execution of the function, are stored in the computer's accumulator (a register employed as a storage place for totals outside normal core storage). To provide this insurance, the program logic of the function definition should

normally include as a last instruction the storage of the output parameter back into itself, as in the following example:

F (X, Y; X) ' ' BEGIN X + Y =) X, X =) X, END ,

In general, then, the function definition has the following form:

VERB (PARAMETERS) ' ' BEGIN ROUTINE END ,

Calls

Without the requisite call, the NELIAC function is not executable because there is no other programmable means for entry into the routine. In addition, the function call provides a list of variables for substitution in the slots reserved by the dummy parameters.

The function call may take two forms. The first is the function as an operand in an arithmetic statement, and the second is as a return jump to the function alone.

The first form is the more sophisticated. It is equivalent to a return jump to the function plus the additional arithmetic statement incorporating the function's output. The following example illustrates the difference between the two forms:

First form: FUNC (A, B $ C) + D =) E,

Second form: FUNC (A, B $ C),

Additional statement: C + D =) E,

It is emphasized that in the first form the function definition must include provision for storage of the function's output in the accumulator. Bitfield operands and some arithmetic calculations involving "add-to-store" operations demand this provision.

Since the function name has been identified as a verb, and since only a noun may serve as an operand in an arithmetic statement, the question arises as to how a function call can be incorporated in arithmetic operations. The answer is twofold. First, only a function with a single output parameter may be used in an arithmetic statement. Second, the inclusion of the function implies that its parametric numeric output will be manipulated in some manner, and since the output is assigned to a noun the rule of NELIAC operands has not been violated. (This argument may seem strained, but it is valid.)

The mode of the function call in an arithmetic statement is determined by the mode of the output parameter. As a result, the entire mode of the expression is dictated by this same parameter.

The second form of the function call resembles the lefthand portion of the function definition--a verb and one or more parenthesized variables. The call is punctuated by a comma indicating a return jump command. Following is an example of the general form:

VERB (INPUT PARAMS $ OUTPUT PARAMS),

There are rules of NELIAC grammar common to both forms of the function call which must be observed.

The function call may have any of the following parameter name forms: a noun, subscripted or not; a noun, with bit limits, subscripted or not; or a verb. In the case of the verb, an unconditional transfer of control within the function's program logic involving the verb must be of the return variety to insure proper stowage of the function's output.

The parameters may be of mixed modes. That is, the input parameters of both the function definition and call may be of a different mode than the output parameters. However, it is mandatory that corresponding parameters (e. g., input parameter 1 of definition and input parameter 1 of call, etc.) be of identical mode. No provision is made for conversion of one mode to the other during the function call which precedes execution.

The parameters must agree in order and number; thus, if two input parameters and three output parameters are defined for the function, two input parameters and three output parameters must be supplied in the function call. If a call is required which deletes the use of a parameter defined in the function, the call must be written with a space between commas to indicate that the parameter's absence is intentional. Otherwise, the parameters are right-justified upon compilation, and the absence of a parameter might transmit an input parameter of the call to an output parameter of a definition, rendering the entire operation meaningless.

All parameters used in the function call are of necessity defined somewhere within the program confines. No attempt need be made to indicate the mode of the call parameters. Punctuation serves only to separate parameters. Regardless of the punctuation used in the call, the dimensioned mode of the call parameters will be used.

Following are a synopsis (table B6-3) and a list of examples (table B6-4) of functions.

TABLE B6-3. CONSTITUENTS OF FUNCTIONS

Definition:

a. Form:
    VERB (PARAMETERS) ' ' BEGIN ROUTINE END ,

b. VERB is any name unique to the more inclusive subroutines; if the function is not nested, VERB is a name unique only to the program

c. PARAMETERS composed of two lists, INPUT and OUTPUT separated by $; dummy parameters to be replaced by other variables specified in function call; INPUT for variables calculated before function call; OUTPUT is provision for results from execution; INPUT parameters: minimum = 1, maximum boundless; OUTPUT parameters: minimum = $\emptyset$, maximum boundless; if no OUTPUT, eliminate $; commas separate parameters if fixed point, periods if floating point; punctuation before $ and) not needed unless last parameter is floating point; PARAMETERS must be specified in same order used in ROUTINE; dimensioning in PARAMETERS may include any name form possible in flowchart dimensioning

d. ROUTINE similar to flowchart program logic; arithmetic and control operations, definitions of and calls on other subroutines and functions, are possible

Call:

a. Forms:

    (1) ...VERB (PARAMETERS)...

    (2) VERB (PARAMETERS),

b. Form (1): Function as operand of arithmetic statement

c. Form (2): Return jump to function

d. General form:
    VERB (INPUT PARAMS $ OUTPUT PARAMS),

e. PARAMETERS may be nouns, subscripted or not, partial or full word; or verbs; mixed modes in call, but each call parameter must be same mode as corresponding definition parameter; call parameters must agree in order and number with definition parameters; all are defined elsewhere in program; commas or periods serve only as separators; omitted parameter must be provided for with dummy comma; form(1) call: no more than one output parameter; form(2) call: no output, no $ or output parameter

TABLE B6-4. EXAMPLES OF FUNCTIONS

a.	SERIES (X, Y $ Z) ' ' BEGIN X + Y / X - X / Y =) Z, P, END ,	legal; function definition
b.	SERIES (A, B $ CHU'M),	legal; form (2) call
c.	HOP + 4.37 * BLEU - TRIM / SERIES (T, SO $ B6) =) N,	legal; form (1) call
d.	BAD CALL (43, A4),	illegal; constant may not be used in call
e.	DENT + LOG (P, Q $ R, S) =) THUD,	illegal; call in routine may not have > 1 output parameter
f.	OHGAD ($ IAG'O) ' ' BEGIN 3 =) IAG'O, END,	illegal; must have $\geq$ 1 input parameter

7. DECLARATIONS

MACHINE DEPENDENCY

A procedure oriented language (POL), as defined in section 2, is a programming tool which incorporates the use of algebraic notation for mathematical solutions, near-English phrases for program control, and other sophisticated capabilities which place the language on a level high above machine code. The POL has been developed for the scientist, engineer, or mathematician who desires to write his own programs and lacks the detailed training required of a machine language programmer.

NELIAC, as the language under discussion, satisfies the criteria for a procedure oriented language. Its syntax is organized so as to be largely machine independent. Routines written in NELIAC may be compiled on any of several digital computers with little compensation for individual machine characteristics.

Some programs, of necessity, must be subject to computer idiosyncrasies. These are programs written in the areas of input/output, often-used and machine dependent algorithms, and address assignment.

Input/Output

Communications between a digital computer and its peripheral equipment (hardware external to the central processing unit such as a printer, card reader, magnetic tape unit, etc.) constitute the programming area known as input/output. The central processing unit, or mainframe, is the reference equipment in such exchanges. Input describes communications from an external device to the mainframe, and output implies the reverse.

Input/output configuration normally changes from one computer installation to another, and communications programming changes sympathetically. Therefore, one might expect for each manufacturer, model, variation in equipment, etc., that the section of a compiler devoted to communications would be wholly machine dependent; this is the case.

Specialized Algorithms

The digital computer manufacturers offer two classes of hardware: generalized and specialized. A generalized machine is one designed for multipurpose uses, suitable for many unspecified jobs. A specialized computer, on the other hand, is constructed to accomplish only the task for which it was designed.

As in computer hardware, there is generalization and specialization in compilers. Specialization provides efficiency in the source-to-object-program translation and in the machine code generated by the translation. The immediate effects of such efficiency are savings in time for compilation and execution and a compression of core space required to contain the object program.

A generalized compiler by its nature must be capable of handling a wide scope of syntactical combinations. Provisions for generalities cause some inefficiencies in object program code reflected as wasted time and space.

Any routine that is machine dependent or specialized, written with the intention that it will be frequently used, and processed by a generalized compiler, is doomed to some inefficiencies at best. Other means for incorporating the required algorithm in the object program must be found or devised.

Address Assignment

In medium-to-large-scale computers, certain addressable registers are reserved for particular purposes. The contents of these registers at all times reflect the status of the function they serve. For example, a binary clock may be an integral part of the computer design. There will be a word in core assigned to the clock, and a request for the contents of that word will yield the clock's numeric reading in binary.

Likewise, a location in core which is the entry point to a specialized routine must be addressable. A name must be given to the routine (so that it may be called from some external point in the program) and concurrently pinned to a particular machine address.

To utilize the clock or provide the entry point, the compiler or some adjunct of it must provide the facilities for addressing these specific registers.

NELIAC accommodates all of these three areas with the "declaration," a powerful machine-dependent programming device.

The declaration is a member of the subroutine family. It is subject to definition and call as is the subroutine and function. But unlike the latter pair, the declaration is an "open" routine-- the appropriate machine code is inserted in the object program each time the declaration call is used. Where calls to NELIAC subroutines or functions are made, return jumps are generated to the routine named. For "closed" routines, the machine code routines are included just once in the object program regardless of the number of calls made upon them.

The declaration definition is a routine written essentially in machine code. It may reside on a compiling system tape awaiting a call into the object program, or it may be entered as part of the source program.

If the routine is a resident of the compiling system, it is known as a "system declaration." Declarations so designated are of sufficient importance to the NELIAC user to be already fully programmed. They are simply called from the system tape wherever needed; no reprogramming is necessary. Input/output, some specialized routines, and some address assignments are included among system declarations.

If the declaration definition is entered as part of the source program, the programmer uses the declaration flowchart (control number of 6, for which see "Flowcharts" in section 2). A definition of this type must be written by the programmer, although such a project should not be undertaken unless space and time are at a high premium. The declaration flowchart is normally reserved for the specialized routine or address assignment.

The general form of a declaration definition is

VERB = ROUTINE,

where VERB is any previously unused NELIAC name, and ROUTINE is an arithmetic and control operation program similar in purpose to a flowchart's program logic. It does not, however, look like the latter. Further discussion is reserved for the next subsections.

The declaration call is similar to the function call. It may be used anywhere in program logic, subject to the restriction that a declaration cannot be called before it is defined. The call involves the specification of parameters or operands to fill the slots provided in the definition. The same form of call is used whether a declaration is a system resident or not.

The declaration call on the sample definition, given above, may be represented in general form by the following

($ VERB LS OPERAND1 GR , ... LS OPERANDn GR , $),

VERB for the definition and call must be identical in order to compile the proper pair together. The operands are constants and variables to be incorporated in the machinations of the declaration's ROUTINE. They are written in the precise order required by the ROUTINE.

The application of declarations to each of the three areas (input/output, specialized routines, address assignments) will be described in detail in the ensuing subsections. For the remaining discussion concerning declarations, all machine references will be made to the digital computer utilized to implement the Command Ship Data System, the Remington Rand Univac AN/USQ-20, about which some knowledge is assumed on the part of the reader. Additional explanation will be offered where it is felt necessary to do so.

CATEGORIES

The routine of a declaration, like any other arithmetic or control operation, is composed of operators and operands. Declaration operators (shortened to "declarators"), written in the declaration definition, indicate the machine code intentions of the programmer. These declarators are English words or phrases punctuated in a manner to provide, when necessary, a niche for operand substitution by the declaration call. The combination of the definition and call is required for generation of the machine language inserted in the object program, because operands and operators both must be specified to write full instructions.

Declarations may be separated into three categories. The category of declaration used is dependent upon the requirements of the algorithm and is a decision made by the programmer. In turn, the category determines in what form the appropriate declarator is to receive the operand requisite to its part in the computation.

Category I

This declaration category depends upon the call as written in program logic to provide the operands and input/output "sense" (direction) to generate machine code. Category I declarations are characterized by the punctuation symbols LS and GR which enclose the declarators of the declaration definition and the operands of the program logic call.

Declarators of this category are always written in the definition in the following manner:

LS DECLARATOR GR ,

Operands provided by the call are enclosed in LS and GR , unless the operand is to be an input parameter (a variable assigned to an area in core to be filled by reading data into the computer from some external device). In this case, the punctuation symbols are reversed, as

GR OPERAND LS ,

Further discussion of the declaration for input/output purposes is presented later in this section.

The declarators listed below belong in Category I:

LS EXTERNAL FUNCTION GR ,
LS RELEASE INTERRUPT LOCKOUT GR ,
LS JUMP ACTIVE GR ,
LS BUFFER GR ,
LS MONITOR BUFFER GR ,
LS GENERATE BUFFER CONTROL WORD GR ,
LS DELAY GR ,
LS SET INTERNAL INTERRUPT ENTRANCE GR ,
LS SET EXTERNAL INTERRUPT ENTRANCE GR ,

Some of these are applicable to other categories; all are defined in the subsection of this section called "Declarators."

Category II

The formation of machine code from declarations of this category is the antithesis of Category I. That is, the operators and required operands are both specified in the declaration definition, no dependence is placed upon the call except the sense of the LS or GR punctuation, an LS followed by a comma in the call indicates that the corresponding operand in the definition should be regarded in the output sense, and a GR followed by a comma indicates the operand is an input variable.

Category II definitions are characterized by declarators enclosed in parentheses. Where an operand is required by the declarator, a decimal or octal number, enclosed by a separate set of parentheses, is located between the declarator and its right parenthesis as in the following:

(DECLARATOR (OPERAND)),

The declarators listed below belong in Category II:

(EXTERNAL FUNCTION (OPERAND)),
(RELEASE INTERRUPT LOCKOUT),
(JUMP ACTIVE),
(TERMINATE BUFFER),
(DELAY (OPERAND)),
(MACHINE CODE (OPERAND)),

The use and meaning of the presence or absence of the OPERAND will be explained later in the subsection, "Declarators."

Category III

Definitions of Category III declarations contain declarators and operands. Calls from program logic provide additional or modifying operands and the input/output sense.

For the declarator EXTERNAL FUNCTION, a four word (three instructions and a temporary storage word) program is tacked onto the otherwise appropriate machine code for the purpose of adding the definition operand to the contents of the corresponding modifying operand of the call at execution time.

In the case of MACHINE CODE, the operand addition is carried out at compilation time and the sum inserted directly into the instruction operands of the machine equivalent to the declaration routine.

All operands used in the definition, regardless of declarator, must be fixed point numbers. Call operands may be fixed point constants or variables, including index register variables. The definition takes the following format:

NAME = (CHANNEL #) LS DECLARATOR (OPERAND) GR ,

while the call (input sense) looks like

($ NAME LS MODIFYING OPERAND GR , $),

Only two declarators belong to the Category III:

LS EXTERNAL FUNCTION (OPERAND) GR ,
LS MACHINE CODE (OPERAND) GR ,

Following are a synopsis (table B7-1) and a list of examples (table B7-2) of categories.

TABLE B7-1.   CONSTITUENTS OF CATEGORIES

Category I
a. Definition depends upon call for operands and I/O sense
b. Definition form: NAME = (CHANNEL #) LS DECLARATOR$_1$ GR , ... LS DECLARATOR$_n$ GR ,

c. Call form: all purposes except input:
   ($ NAME LS OPERAND GR , $),; input:
   ($ NAME GR OPERAND LS , $),
d. Applicable declarators:
   LS EXTERNAL FUNCTION GR ,
   LS RELEASE INTERRUPT LOCKOUT GR ,
   LS JUMP ACTIVE GR ,
   LS BUFFER GR ,
   LS MONITOR BUFFER GR ,
   LS GENERATE BUFFER CONTROL WORD GR ,
   LS DELAY GR ,
   LS SET INTERNAL INTERRUPT ENTRANCE GR ,
   LS SET EXTERNAL INTERRUPT ENTRANCE GR ,

Category II
a. Definition depends upon call for I/O sense only; operands are
   self-contained
b. Definition form: NAME = (CHANNEL #)
   $(DECLARATOR_1$ (OPERAND)), $(DECLARATOR_2)$, . . .
   $(DECLARATOR_n$ (OPERAND)),
c. Call form: ($ NAME LS , GR , GR , LS , . . . etc. $) (LS:
   operand to be treated in output sense; GR: operand to be
   treated in input sense)
d. Applicable declarators:
   (EXTERNAL FUNCTION (OPERAND)),
   (RELEASE INTERRUPT LOCKOUT),
   (JUMP ACTIVE),
   (TERMINATE BUFFER),
   (DELAY (OPERAND)),
   (MACHINE CODE (OPERAND)),

Category III
a. Definition depends upon call for I/O sense and modifying
   operands; operands are self-contained
b. Definition form: NAME = (CHANNEL #)
   LS $DECLARATOR_1$ (OPERAND) GR , . . .
   LS $DECLARATOR_n$ (OPERAND) GR ,

TABLE B7-1. (CONT)

c. Call form: all purposes except input:
   ($ NAME LS OPERAND GR , $),; input:
   ($ NAME GR OPERAND LS , $),
d. Applicable declarators:
   LS EXTERNAL FUNCTION (OPERAND) GR ,
   LS MACHINE CODE (OPERAND) GR ,

TABLE B7-2. EXAMPLES OF CATEGORIES

a. PUNCH CARD 1 = (1∅)   legal; Category I definition
   LS MONITOR BUFFER GR ,
b. ZU INT RELEASE =   legal; Category II definition
   (MACHINE
   (6∅∅∅∅ OCT ∅)) ,
c. RELEASE = LS MACHINE   legal; Category III definition
   (6∅11∅ OCT ∅) GR ,
d. ($ JB TRAP LS D GR ,   legal; Categories I and III call
   GR PFF LS , $),
e. ($ CALL NELOS LS , $),   legal; Category II call

DECLARATORS

As the operators of the declaration routine, declarators are
the symbolic phrases which generate the machine language func-
tion or operation code, whereas the operands of the definition and

call are responsible for completion of the instruction format. The function code f of a machine instruction is the fundamental command to the computer's logic.

The discussion which follows is an exposition of the purpose(s) each declarator serves.

Release Interrupt Lockout

RELEASE INTERRUPT LOCKOUT is a declarator which may be used in Category I or Category II declarations.

The machine function code $f = 6\emptyset$ is generated for each specification of this declarator. This machine instruction has two extraordinary uses besides its intention as a jump command; if the branch designator j is equal to a zero, the command to clear the interrupt lockout mode (established by the interrupt lockout instruction $73\emptyset3\emptyset\emptyset\emptyset\emptyset\emptyset$) will be created, no jump will actually transpire, and the next sequential instruction will be executed; if $j = 1$, the interrupt lockout will be cleared, and a jump to the operand specified by the call will follow the release.

Employing RELEASE INTERRUPT LOCKOUT in a Category I declaration causes j to be set to a 1. In a Category II delcaration, the arithmetic jump command generated has a $j = \emptyset$, and no operand is required.

Jump Active

Declarations of Category I or Category II may employ this operator to jump to a specified address if a particular channel is active.

If a Category I declaration is defined, the machine function code is determined by the I/O sense of the call from program logic. In the case of an operand surrounded by punctuation indicating the input sense, the function code $f = 62$ is generated; for the output case, the instruction becomes a 63 function code. For either function code, the operand of the call is inserted as the operand of the object program.

If a Category II declaration is defined, the declaration JUMP ACTIVE and the I/O sense of the corresponding LS or GR punctuation of the call will combine to designate the proper function code. If punctuation is LS , the instruction will be $f = 63$; if GR , f will equal 62. Note that JUMP ACTIVE does not require an operand in a Category II definition. The operand substituted will be the address of the instruction itself.

The JUMP ACTIVE declaration forces a check to see if the communications channel $\hat{j}$ is actively transmitting information either in or out. If the channel is found to be active, the operand in the generated instruction becomes the address transferred to. If it is inactive, the next sequential instruction is executed.

External Function

This declarator is the only one of eleven to belong to all three categories.

EXTERNAL FUNCTION generates one of two machine function codes--$f = 13$ or $f = 17$--depending upon the I/O sense of the punctuation in the active statement. In the input sense, $f = 17$ is generated, and the information from the communication channel $\hat{j}$ will be stored in the operand formed. In the output sense, $f = 13$ is generated, and the contents of the operand specified will be transmitted out on channel $\hat{j}$ to the peripheral equipment connected thereto. A channel number must be specified by the declaration definition.

The sole purpose of this declarator is to provide a means for control communications between peripheral equipment and the mainframe. For $f = 13$, a word will be transmitted out on the indicated channel, forcing the external device to accomplish some function such as turning itself on, enabling an output mode, etc. In the opposite direction, an interrupt may be generated by a piece of peripheral equipment containing an indication that it has some control information to give to the computer, such as the fact that it has just completed the rewind of a magnetic tape unit. The only way the computer can assess the meaning of the interrupt is to draw it in core for examination; this store operation is the purpose of the $f = 17$ instruction.

As a Category I declarator, EXTERNAL FUNCTION relies upon the call for operand and input/output sense. Defined in a Category II declaration, the declarator is written with an operand; it requires the input/output sense of the program logic call. Category III means that EXTERNAL FUNCTION receives a modifying operand and the intended direction from the call to be compiled along with its self-contained operand.

Generate Buffer Control Word

The terms "buffer" and "buffer control word" introduce some new topics which require the following preliminary discussion before proceeding to the subject itself of this subsection.

The AN/USQ-2∅, or mainframe, and its peripheral equipment communicate in a buffered mode. This means that normal mainframe operations may take place simultaneously with input/output functions. For a program involving much communications, the buffered mode represents a great saving in time.

A buffer of the AN/USQ-2∅ is that area of core specified by the buffer control word (BCW), a single word in memory that contains the upper and lower limits of the area. Designating certain locations as a buffer identifies them with the input/output process.

This designation marks the area with a purpose in addition to its normal function as programmable storage. That is, the buffer may be filled or transmitted by an input/output program, the contents of the buffer may be manipulated by program dynamically during communications, or the buffer area may be used for purposes unallied with input/output.

The opposite, however, is not true. Data transmitted externally may not be taken from just anywhere in core; data to be communicated must be provided for with a BCW.

Communications proceed at a word-by-word pace in either the input or output direction. How the information is assembled is determined by the direction of transmission and the equipment receiving the data. For example, the printer (strictly an output device) must print 120 columns simultaneously; since one word provides only 5 characters, the printer must wait for 24 transmissions before it can assemble and print a line. A card reader, on the other hand, as an input device, reads only enough information from a card to fill the buffer specified; if the buffer is longer than that necessary to contain 80 columns of information (16 words), subsequent cards are read to satisfy the buffer. The magnetic tape unit requires only a variable length buffer. In the output direction, the length of a buffer in general specifies the length of a record on tape. In the input direction, the information from a record in words up to the length defined in the BCW is stored in the buffer. If too much information is contained in the record for the buffer, an interrupt is sent to the computer indicating the fact; if too little or just enough data to fill the buffer is received from the record, an interrupt is generated by the magnetic tape unit denoting the sensing of the end of a record. Any space in the buffer unfilled by the input operation maintains its previous contents.

Communications do not commence until the device addressed is ready to accept the information. When that equipment signals its acceptance, communications between the buffer and the device begin. Meanwhile, the mainframe continues to perform its functions oblivious of the communications. The computer is able to handle control and arithmetic operations simultaneously with input/output because the logic for each is separate.

Once initiated, the mechanically programmed section of computer logic governing the buffer mode proceeds to input or output data, without additional prodding, at a speed determined by the external device (because the mainframe is so fast). Buffer guidelines are provided by the limits and length of the BCW.

The lower half of the BCW contains the five-digit starting address; transmission of information continues without interruption until the data contained in the address in the upper half of the BCW have been moved. At such time, the buffer is said to be terminated, and the input/output processing ceases until begun again by appropriate instruction.

The length of the buffer used in communications is a decision of the programmer. It is necessary, however, to consider the fixed length requirements of the peripheral devices when communicating with them--for example, the printer buffer should be some multiple of 24 words in length, and that of a card punch a multiple of 16 words. Otherwise, a partially filled buffer will cause unexpected results following the next communications.

Now, to discuss the declarator , GENERATE BUFFER CONTROL WORD, itself--it serves a small but important purpose: the storing of the buffer control word in the Q-register (a nonaddressable 30-bit arithmetic register) for access by the input/output processor. The latter takes the BCW from the Q-register and stores it in the specially wired address for control of buffered communications.

GENERATE BUFFER CONTROL WORD is a Category I declarator dependent upon the declaration call for substitution of an operand. The punctuation enclosing the operand is ignored as the declarator acts independently of input/output sense.

The storing of the BCW in the Q-register is but the end product of this declarator. Depending upon the operand of the declaration call, a translation of some form must transpire before the BCW is established.

To input or output an entire list, the call operand takes the form

LS LIST GR ,

GENERATE BUFFER CONTROL WORD obtains the lower and upper limits of LIST and creates a BCW by placing the upper limit in the upper half of the BCW and the lower limit in the lower half.

To input or output some continuous portion of a list, "running subscripts" are used. These are integers, register variables, or fixed point whole or half word variables separated by a store operand such as 3 =) 7 , A =) PAB , I =) J. Running subscripts modify the address of the first list entry to provide the addresses of the desired segment. For example, LIST ($ 3 =) 7 $) tags LIST ($ 3 $) through LIST ($ 7 $) for communications. The declarator converts the subscripts to machine addresses and generates the BCW from them.

To communicate to or from some list whose BCW is already contained in an address, the form

LS ($ ADDRESS $) GR ,

is used. The contents of that location, without checking for validity, will be treated as the BCW.

Buffer

This is a Category I-only declarator.

Since all input/output communications involve the use of buffering for data transmission, it is not surprising to find declarators which provide the means for initiating buffers. To initiate a buffer implies that transmission of data into or from the locations specified in the BCW should commence.

For information transfer in the mainframe to peripheral equipment direction, the $f = 74$ command is generated by the declarator BUFFER. Input buffer initiation is accomplished by the $f = 73$ command. The instruction generated is dictated by the sense of the operand in the call from program logic.

The channel designator $\hat{j}$ is required to indicate which channel is to be employed for buffer operations. The channel number of the declaration provides this designator.

BUFFER has all the capabilities of generating a BCW from a program logic call that the declarator GENERATE BUFFER CONTROL WORD has. The call operand may take any of these forms: LIST, LIST ($ MODIFIER 1 =) MODIFIER 2 $), ($ LIST $). (See GENERATE BUFFER CONTROL WORD, immediately preceding.)

Set Internal Interrupt Entrance

This declarator provides the capability to store a return jump at the special address or "entrance" queried by the MONI-TOR BUFFER declarator. Termination of a buffer generates an interrupt which halts all processing and causes the entrance to be queried, and any instruction there executed. The precise address is determined by the sense of the call operand and the channel which is to be active during the communications.

Before explaining the SET INTERNAL INTERRUPT ENTRANCE, the term "interrupt" should be defined. An interrupt is a coded message generated by the occurrence of some input/output phenomenon; its purpose is the intervention of program execution which has proceeded in parallel (also known as "asynchronous operations") during communications, and the provision of equipment status information for the input/output processor (a programmer-specified algorithm). The interrupt is one word in length, and the contents of certain bits are interpreted by the processor. Status information might include the fact that a magnetic tape unit had written a record properly or that the tape rewind previously requested had been terminated.

Two types of interrupts are accepted by the input/output processor--external and internal. The external interrupt is created by an input/output event in a device external to the

mainframe. The preceding examples concerning status information are of this type. An internal interrupt is generated by the input/output section of computer logic when a monitor buffer, previously initiated, is terminated after transmission of the specified block of data. This form of interrupt is accepted by the input/output processor as well.

After interpretation of an interrupt and execution of any associated interrupt program, control is returned to the main program at the address abandoned due to the interruption.

SET INTERNAL INTERRUPT ENTRANCE is a Category I declarator. In addition to the generation of the return jump, sufficient other machine code is inserted into the object program to store the return jump instruction in the appropriate entrance, as determined by the sense of the corresponding declaration call operand and the channel number of the definition.

Monitor Buffer

The difference between BUFFER and MONITOR BUFFER lies in the direction of program flow after execution of the declarator.

BUFFER causes a channel to become active and the BCW is stored at an address specially wired for access by the mechanized input/output program (discussed earlier). Communications then are begun and continue until terminated. If more input/output jobs are to be accomplished, the input/output section processes them all in turn. When the job queue is empty, this portion of the computer becomes idle. Meanwhile, parallel data processing has been accomplished. The next instruction to be executed after the last buffer termination is the one presently being executed by the arithmetic and control logic in parallel with the communications.

MONITOR BUFFER goes through the same steps as BUFFER. However, it goes one step further by allowing the programmer to

place an instruction at another specially wired address. The next instruction to be executed after buffer termination will be the one found at the special address; an internal interrupt causes all processing except that instruction to cease. In most cases the address will contain a return jump to some programmer-specified routine which will perform one or several functions to "tidy up" or to the input/output section such as for turning off the external device. After execution of the subroutine, control will be transferred to the address of the next sequential instruction of the mainstream program before it was interrupted.

To initiate a monitored buffer in the input direction, an $f = 75$ is generated by the declarator while the operand is supplied by the declaration call. The function code $f = 76$ applies to a monitor buffer in the output sense. MONITOR BUFFER is valid only in Category I.

The channel number provided by the declaration definition supplies the necessary information to complete the instruction. The channel designator $\hat{j}$ indicates the channel to be made active during buffered communications; its value is supplied by the programmer designation of channel number.

MONITOR BUFFER has all the capabilities of generating a BCW from a program logic call as has the declarator GENERATE BUFFER CONTROL WORD. The call operand may take any of these forms: LIST, LIST ($ MODIFIER 1 =) MODIFIER 2 $), ($ LIST $). (See preceding GENERATE BUFFER CONTROL WORD.)

Set External Interrupt Entrance

Specially wired entrances are provided for external interrupts as well. The particular location referenced as an interrupt entrance will be defined by the input/output sense of the communications which led to the interrupt and the channel which was active during data transmission.

When an external device experiences a prescribed event of significance to the computational process, it sends an interrupt to the mainframe. The portion of the input/output processor that is to handle the interrupt is pinpointed by a return jump contained in the appropriate external interrupt entrance. SET EXTERNAL INTERRUPT ENTRANCE generates a return jump at the entrance specified by the declaration definition and call--input/output sense and operand address (for the return jump) from the call, and the channel number from the definition.

SET EXTERNAL INTERRUPT ENTRANCE is a Category I declarator, as well. Like its corollary, SET INTERNAL INTERRUPT ENTRANCE, this declarator generates the machine code necessary to store the return jump at the proper address upon program execution.

Terminate Buffer

In monitor buffered operations, when the contents of the entire buffer as defined by the BCW have been outputted or the buffer has been filled by a peripheral device, an internal interrupt is generated by input/output logic notifying the interrupt program that the buffer has been terminated.

However, it may become necessary to prematurely terminate a buffer. TERMINATE BUFFER incorporated in a declaration provides for such an eventuality. As a Category II declarator, it does not depend on the definition for anything except the channel number and only upon the call for the sense of communications, and, in addition, it does not require an operand.

At execution, the declarator will force a simulated buffer termination, but no internal interrupt will occur. The buffer will be terminated on the input channel by the $f = 66$ machine function code. In the output direction, the $f = 67$ function code generated by the declarator will cause completion of buffered communications on the channel designated.

Delay

DELAY causes the inclusion of two machine language instructions and the creation of a one-instruction loop. Upon execution, it effectively provides sufficient delay between other declarators to permit electromechanical devices to match the electronic communications rate. The first instruction ($f = 12$) is relied upon to store a count in one of the index registers (in this case B7). The second instruction is an indexed jump command ($f = 72$) to its own compiled address.

DELAY is either a Category I or II declarator. If it is defined in a Category I declaration, the operand of the declaration call becomes the count. If it is in a Category II declaration, the count is a self-contained operand.

At execution, the count is stored in B7; the indexed jump acts as a "do nothing" command by jumping repeatedly to itself, each time decrementing the count in B7 by one. When B7 is reduced to zero, the next sequential instruction is executed.

The delay is equivalent to the execution time of the indexed jump instruction (8 microseconds) multiplied by the operand value (count). Input/output sense of the operand is ignored.

Machine Code

The declarator, MACHINE CODE, is the most versatile of all eleven declarators, even though restricted to Categories II and III. Its purpose is to provide the programmer with the ability to include any machine instruction of the AN/USQ-2∅ repertoire in his source program.

In Category II, the declarator is written in a manner to include the entire operand, and, since the operand is the desired machine language instruction, it is incorporated verbatim into the

object program. This, of course, dictates that the operand as written in the definition be composed entirely of numbers. The indicated sense is always output.

The operand is always divided into two halves. The first half contains the function code (two digits) and three designators (each one digit); the second half is the instruction operand which may contain an address or some data (one to five digits).

One degree of flexibility exists in the Category II declaration involving MACHINE CODE. If the letter "L" is written to the immediate right of the declaration operand's second half (this is the only exception to the rule of wholly numeric operands in Category II declaration definitions), it serves as a signal to the compiler that an addition is to be performed before generation of machine code. The compiled address of the machine language instruction is added to the operand contained in the half which precedes the "L", and the sum is substituted as the second half of the now finished machine code instruction. For example, the declarator (MACHINE CODE (61ØØØ OCT 77776 OCT L)), might be compiled at address 42315_8, in which case the instruction as inserted in the object program would look like 61ØØØ42314 since 77776 acts as a minus one. In the positive direction, (MACHINE CODE (61ØØØ OCT 13 OCT L)), at a hypothetical address 26727_8 would generate a machine instruction 61ØØØ26742.

It is emphasized that this facility to specify a relative address in machine code should not be disregarded. In the first example, the programmer wished to jump to the instruction just passed; in the second example, the jump was to the thirteenth (octal) address beyond the generated instruction.

As a Category III declarator, MACHINE CODE incorporates an operand in the definition and relies upon the call to provide an added operand. At compilation time, the corresponding operands are summed and the result becomes the operand of the machine language instruction. The indicated sense is always output.

A single degree of flexibility also is possible in Category III MACHINE CODE declarations, but before explaining this it is necessary to first talk about another machine quantity.

One of the three designators mentioned previously as components of the first half of a machine instruction is the operand interpretation designator "k." As the name would indicate, different values assigned to "k" cause the computer to interpret the instruction operand in different ways. For example, $k = 1$ forces consideration of only the lower half of the contents at the operand address; $k = 3$ causes the whole of the operand's contents to be considered. Values for "k" range from $\emptyset$ to 7.

The NELIAC compiler assigns a "k" of zero to all verbs because they are addresses and used solely as entry points. Full word nouns on the other hand are given a $k = 3$ because the entire contents are a candidate for further processing.

When half-word algebra was indicated as preferable to other partial word algebra, the reason was the k-designator. Half-word nouns can be manipulated with no more difficulty than whole word nouns because $k = 1$ handles the lower half and $k = 2$ the upper half of nouns.

If a Category III MACHINE CODE declaration is written with a "K" to the immediate right of the second half of the definition operand (same position as the "L" of Category II), whatever the k-designator of the embryo declaration machine instruction in the definition, the k-designator of the declaration call operand suppresses and replaces it. For example, if the call operand is a verb, the machine code instruction inserted in the object program has a k-designator of zero; for a noun dimensioned as a full word, k becomes a three in the object program.

This flexibility is more than a protection device. If properly used, an address or the contents thereof need never be improperly referenced.

Regardless of the category declaration used, this declarator is the most demanding of all with regards to understanding the reference machine. However, it also provides the facility of writing an algorithm in machine code for purposes of efficiency in space and time since all function codes may be used, and each is transferred almost verbatim to the object program (without extra instructions to store and retrieve operands, etc., that one might

expect from the translation of near-English phrases to machine code).

Also irrespective of category line is the fact that input/output sense is completely ignored in the MACHINE CODE declaration. The output sense is normally assumed for appearance alone.

Following are a synopsis (table B7-3) and a list of examples (table B7-4) of declarators.

TABLE B7-3. CONSTITUENTS OF DECLARATORS.

RELEASE INTERRUPT LOCKOUT
a. Category I or II
b. Purpose: clear interrupt lockout mode
c. Category I: $f = 6\emptyset$, $j = 1$, clear lockout, jump to operand
d. Category II: $f = 6\emptyset$, $j = \emptyset$, clear lockout, no jump, next sequential instruction
e. No channel number specification necessary

JUMP ACTIVE
a. Category I or II
b. Purpose: jump to specified address if channel is active
c. Input: $f = 62$, Category I: input channel active, jump to operand; Category II: input channel active, jump to own address
d. Output: $f = 63$, Category I: output channel active, jump to operand; Category II: output channel active, jump to own address
e. If channel inactive, next sequential instruction
f. Channel number required in definition

EXTERNAL FUNCTION
a. Category I, II, or III
b. Purpose: control communications
c. Input: $f = 17$, interrupt store instruction
d. Output: $f = 13$, external equipment function command
e. Channel number required in definition

GENERATE BUFFER CONTROL WORD
a. Category I
b. Purpose: establish BCW in Q-register; required for buffer or monitor buffer operations
c. $f = 1\emptyset$, store BCW in Q
d. Operand forms: LIST, LIST ($ MODIFIER 1 =) MODIFIER 2 $), ($ LIST $); operand converted to BCW
e. No channel number specification necessary

BUFFER
a. Category I
b. Purpose: initiate data transmission; controlled by BCW
c. Input: $f = 73$, initiate buffered input
d. Output: $f = 74$, initiate buffered output
e. Upon buffer termination, no definitive action
f. Channel number required in definition

SET INTERNAL INTERRUPT ENTRANCE
a. Category I
b. Purpose: store return jump at internal interrupt entrance
c. Three instructions: (1) enter Q-register ($f = 1\emptyset$) with contents of next address, and skip around next address; (2) contents: return jump instruction; (instruction operand furnished by call); (3) store contents of Q-register ($f = 14$) at internal interrupt entrance
d. Entrance determined by input/output sense (call) and channel number (definition)

MONITOR BUFFER
a. Category I
b. Purpose: initiate data transmission; controlled by BCW
c. Input: $f = 75$, initiate buffered input
d. Output: $f = 76$, initiate buffered output
e. Upon buffer termination, jump to internal interrupt entrance
f. Channel number required in definition

SET EXTERNAL INTERRUPT ENTRANCE
a. Category I

b. Purpose: store return jump at external interrupt entrance
c. Three instructions: see SET INTERNAL INTERRUPT ENTRANCE
d. Entrance determined by input/output sense (call) and channel number (definition)

TERMINATE BUFFER
a. Category II
b. Purpose: manual buffer termination
c. Input: $f = 66$, terminate input buffer
d. Output: $f = 67$, terminate output buffer
e. No internal interrupt generated
f. Channel number required in definition

DELAY
a. Category I or II
b. Purpose: provide delay between input/output operations to allow external devices to match communication flow
c. Two instructions: (1) enter B7 index register ($f = 12$) with count; (2) indexed jump command ($f = 72$) to own compiled address (number of executions = count)
d. When B7 = $\emptyset$, next sequential instruction
e. No channel number specification necessary

MACHINE CODE
a. Category II or III
b. Purpose: include any machine language instruction in object program
c. Operand is instruction; separated in two halves: (1) function code and designators; (2) instruction operand; each half numeric, except in extended form
d. Category II extended form: "L" after definition operand; generates instruction operand = sum of instruction's compiled address and definition operand
e. Category III extended form: "K" after definition operand; causes suppression of k-designator of definition operand and replacement with k-designator of call operand (verb: $k = \emptyset$, half word noun: $k = 1$ or 2, full word noun: $k = 3$

f. Channel number not normally required in definition (depends on operand)

g. Input/output sense of call ignored

TABLE B7-4. EXAMPLES OF DECLARATORS

a-1.	RIL1 = LS RELEASE INTERRUPT LOCKOUT GR ,	legal; Category I definition
a-2.	($ RIL1 LS JUMP ADDRESS GR , $),	and call
b-1.	RIL2 = (RELEASE INTERRUPT LOCKOUT),	legal; Category II definition
b-2.	($ RIL2 LS , $),	and call
c-1.	JA1 = (5) LS JUMP ACTIVE GR ,	legal; Category I definition
c-2.	($ JA1 GR JUMP ADDRESS LS , $),	and call (input)
d-1.	JA2 = (5) (JUMP ACTIVE),	legal; Category II definition
d-2.	($ JA2 LS , $),	and call (output)
e-1.	EF1 = (5) LS EXTERNAL FUNCTION GR ,	legal; Category I definition
e-2.	($ EF1 GR 1∅ OCT LS , $),	and call (input)
f-1.	EF2 = (5) (EXTERNAL FUNCTION (4∅ OCT)),	legal; Category II definition
f-2.	($ EF2 LS , $),	and call (output)
g-1.	EF3 = (5) LS EXTERNAL FUNCTION (4∅ OCT) GR ,	legal; Category III definition
g-2.	($ EF3 GR 1∅ OCT LS , $),	and call (input)
h-1.	GBCW = LS GENERATE BUFFER CONTROL WORD GR ,	legal; Category I definition
h-2.	($ GBCW LS LIST GR , $),	operand form 1 call, or

h-3.	($ GBCW LS LIST ($ 4 =) 7 $) GR , $),	operand form 2 call, or
h-4.	($ GBCW LS ($ LIST $) GR , $),	operand form 3 call
i-1.	BUF = (5) LS BUFFER GR ,	legal; Category I definition
i-2.	($ BUF GR LIST LS , $),	and call (input)
j-1.	SIIE = (5) LS SET INTERNAL INTERRUPT ENTRANCE GR ,	legal; Category I definition
j-2.	($ SIIE GR RETURN JUMP ADDRESS LS , $),	and call (input)
k-1.	MBUF = (5) LS MONITOR BUFFER GR ,	legal; Category I definition
k-2.	($ MBUF LS LIST GR , $),	and call (output)
l-1.	SEIE = (5) LS SET EXTERNAL INTERRUPT ENTRANCE GR ,	legal; Category I definition
l-2.	($ SEIE LS RETURN JUMP ADDRESS GR , $),	and call (output)
m-1.	TBUF = (5) (TERMINATE BUFFER),	legal; Category II definition
m-2.	($ TBUF GR , $),	and call (input)
n-1.	DEL1 = LS DELAY GR ,	legal; Category I definition
n-2.	($ DEL1 LS COUNT GR , $)	and call
o-1.	DEL2 = (DELAY (4$\emptyset\emptyset\emptyset$ OCT)),	legal; Category II definition
o-2.	($ DEL2 LS , $),	and call
p-1.	MC1 = (MACHINE CODE (61$\emptyset\emptyset\emptyset$ OCT 14$\emptyset\emptyset\emptyset$ OCT)),	legal; Category II definition
p-2.	($ MC1 LS , $),	and call
q-1.	MC2 = (MACHINE CODE (61$\emptyset\emptyset\emptyset$ OCT 54 OCT L)),	legal; Category II extended; definition
q-2.	($ MC2 LS , $),	and call
r-1.	MC3 = LS MACHINE CODE (1$\emptyset\emptyset\emptyset\emptyset$ OCT $\emptyset$) GR ,	legal; Category III definition
r-2.	($ MC3 LS 49 GR , $),	and call
s-1.	MC4 = LS MACHINE CODE (1$\emptyset\emptyset\emptyset\emptyset$ OCT $\emptyset$K) GR ,	legal; Category III extended; definition
s-2.	($ MC4 LS FULL WORD NOUN GR , $),	and call

DEFINITION AND CALL

This subsection summarizes the information on declarations that has been presented in this section. Specialized algorithm and address assignment types of declarations will be emphasized because they are far more apt to be written than the input/output type.

Definition

The function of system declarations should be reiterated at this point. The system declaration is reserved for the more difficult input/output routines and for those specialized algorithms and address assignments which are of sufficiently wide application to merit writing them once and making them available to all system programmers. For these people, system declarations are well enough documented to reduce the use of declarations to copying the proper call into the source program at the correct places and to supplying the proper operands for the declaration call. Note well that the use of system declarations does not imply the writing of the declaration definitions since these are supplied by the system automatically when called.

For purposes of efficiency or to remedy deficiencies in other portions of the language, the declaration definition as part of the source program may be utilized. If such is the case, it will be necessary for the programmer to know some of the finer details of declaration definition.

The definition must be written as part of a declaration flow-chart (section 2). These flowcharts may be located anywhere in the program provided they precede the particular declaration call in the program logic of some flowchart. It is therefore not surprising to see the declaration flowchart(s) written before all other flowcharts in the program.

As previously stated, the general form of a declaration definition is

VERB = ROUTINE ,

where VERB is any previously unused NELIAC name, and ROU-TINE is an arithmetic and control operation segment, similar in purpose, if not likeness, to a flowchart's program logic. The routine of a declaration definition is composed of operators and operands. The operators, called declarators, are always in-cluded in the definition. The operands, however, are written either in the definition or the call or both, depending upon the category that the programmer selects for his declarators.

Normally, ROUTINE is composed of several declarators, the appropriate operands and any necessary punctuation. The definition may contain declarators of mixed categories; e.g., MACHINE CODE of Category II, EXTERNAL FUNCTION of Cate-gory I, DELAY of Category II, and MACHINE CODE of Category III may all be in the same definition. Incorporating these declara-tors, a definition might be

TROUBLE = (MACHINE CODE (11Ø3Ø OCT Ø L)),
          LS EXTERNAL FUNCTION GR ,
          (DELAY (4ØØ OCT)),
          LS MACHINE CODE (61ØØØ OCT Ø) GR ,

(No attempt at creating a meaningful declaration has been made.)

Since all declarators except MACHINE CODE are primarily input/output oriented, it is not surprising to find that specialized algorithms rely upon this particular declarator. To illustrate the form for definitions of specialized algorithms, the following system declaration has been chosen for analysis:

SEARCH NOT BETWEEN = LS MACHINE CODE
                     (11Ø3Ø OCT ØK) GR ,
                     LS MACHINE CODE
                     (1ØØ3Ø OCT ØK) GR ,
                     (MACHINE CODE (21ØØØ
                     OCT 1)),

LS MACHINE CODE
(7Ø23Ø OCT ØK) GR ,
LS MACHINE CODE
(Ø4537 OCT 77776 OCT
ØK) GR ,
LS MACHINE CODE
(61ØØØ OCT ØK) GR ,
LS MACHINE CODE
(1673Ø OCT ØK) GR ,

Since the MACHINE CODE declarator contains the major portion of the machine language instruction as its operand, one may expect just seven machine language instructions to be generated from the above definition. Six of the seven declarators are enclosed in punctuation indicating Category III, and therefore the programmer would be expected to supply six operands in the declaration call. The seventh operand is self-contained.

The six operand meanings are as follows: first declarator, "lower argument"; second, "upper argument"; third, "list length to search"; fourth, "name of list"; fifth, "no find entry"; and sixth, "find index." The machine language routine inserted in the object program for this definition will read:

a.   Enter the A-register with the lower argument.

b.   Enter the Q-register with the upper argument.

c.   Subtract one from the contents of the A-register and store the result in the A-register.

d.   Execute the next instruction a number of times equal to the list length to search; decrement the instruction operand of the next instruction by one upon each execution; enter the list length into B7.

e.   Compare: skip the next instruction if the contents of the entry being examined are greater than the contents of the Q-register, or less than or equal to the contents of the A-register. The address of the first entry of the list to be examined is equal to the address identified by

the name of the list, plus the length of the list minus one (i. e. , the search begins at the end of the list and terminates at the beginning).

f.   Jump unconditionally to the "no find" routine entry.

g.   Store the contents of the index counter (B7) in the location identified as "find index."

To summarize the purpose of the declaration: the declaration causes a list of a known length to be searched for values between an upper and a lower limit. If a find is made at any point, the search is abandoned with the index count saved for purposes of later reference. If no find is made, the declaration is exited and a jump to a routine implying "no find" is made.

With the tools given, little additional explanation other than the understanding of machine instructions is necessary. Several points of punctuation come to mind: the OCT or octal symbol must be written whenever the numeric portion of an operand is to be interpreted by the compiler in the octal number system; each declarator (and operand, if required) is separated from the succeeding declarators by a comma; and any additional punctuation is determined by the category selected.

A hybrid category has not been classified with the other three. This "fourth" category is not a true declaration type, but is instead a specification of the order in which a series of declarations are to be executed. All declarations specified must have been previously defined. The specification itself is a definition and is given a name. The call for this declaration from program logic must provide the necessary operands for all the designated declarations and in the order they are to be executed. A sample definition might be:

NAME 1 = (4) LS EXTERNAL FUNCTION GR , (DELAY
    (OPERAND 2)),
NAME 2 = (6) LS BUFFER GR , LS MACHINE CODE
    (OPERAND 4a)) GR ,
NAME = NAME 1, NAME 2,

Sample call:

($ NAME LS OPERAND 1 GR , LS , LS OPERAND 3 GR ,
LS OPERAND 4b GR , $),

Although this example is strictly input/output oriented, the
same formation could be used with the specialized algorithms.

In connection with the example, the channel number of each
declaration applies to all declarators in that declaration which
require a channel number. If a different channel is to be specified,
then a separate declaration must be written (note that NAME 1 used
channel 4 and NAME 2 used channel 6; if the declarators in NAME
2 had referenced channel 4 instead of 6, the two declarations could
have been merged into one, but not necessarily).

The address assignment form of declaration is unique. It
enables the programmer to assign a name to a particular machine
address. Any nonrelocatable routine (program compiled at a given
address) introduced into core simultaneously with a NELIAC pro-
gram is executable only if the machine address is given a NELIAC
name.

The format of such a declaration definition is

VARIABLE ' ' K   ADDRESS,

VARIABLE is any NELIAC name; K is the operand interpretation
designator (see MACHINE CODE in preceding subsection on
declarators); ADDRESS is an octal or decimal machine address.
Each address assignment is followed by a comma. Consider the
declaration which allows reference to the specially wired clock
register at address $\emptyset\emptyset\emptyset36$ octal:

CLOCK ' ' 3   36 OCT ,

As before, the k-designator reflects the address usage: $\emptyset$ for
verbs, 3 for full word nouns, etc.

Address assignment declarations need no calls; they are the
exception to the rule, and may be considered self-calling.

Calls

The declaration programming device comes in two parts, mentioned previously: the definition and the call. For input/output and specialized algorithms they are inseparable--one cannot be used without the other. Both are needed to generate the machine code.

The call is found in the program logic of any process flow-chart. Depending upon the categories of the declarators involved in the definition, the call may provide the input/output sense and/or operands to complete the machine language instructions. The operands of a call must line up with the spaces left for them in the definition; they must be in the order specified and in number equal to the requirements.

Consider, if you will, a call on the system declaration, SEARCH NOT BETWEEN, illustrated previously:

```
($  SEARCH NOT BETWEEN  LS 4ØØ OCT GR ,
                        LS 45Ø OCT GR ,
                        LS 47 OCT GR ,
                        LS DATA BANK GR ,
                        LS NOFIND GR ,
                        LS CONTINUE GR ,  $),
```

The requirements were filled; $4ØØ_8$ became the lower argument, $45Ø_8$ the upper argument, 47_8 the list length to search, DATA-BANK the name of the list, NOFIND the "no find" entry, and CONTINUE the "find index." Note that they were equal in number to the definition needs and in the order prescribed; there was no call operand (or space left blank) for the operand already specified in the definition.

The call from the program logic causes the compiler to generate a machine language routine from the function codes (disguised as declarators) of the definition and the addresses and values of the call. Machine instructions are inserted in the object program each time the call is written.

To prevent the inclusion of the same machine instructions every time the identical declaration is referenced, the programmer may enclose the declaration call between BEGIN and END punctuation, making it a closed subroutine. The subroutine may have the same name as the declaration. Any subroutine call of this form simply generates a return jump to the declaration routine which is included once in the object program.

The general form for a declaration call is

($ VERB LS OPERAND 1 GR , GR , GR OPERAND 2 LS , ... , $),

An input operand is specified GR ... LS, and an output operand is vice versa. In a call the sense frequently determines the function code of the machine instruction generated, but where no sense is required by the declarator the output sense is assumed. In a call input and output operands may be mixed. Each operand is enclosed by GR and LS and commas separate operands. Each declaration call begins with the combination ($ VERB LS or ($ VERB GR .

Operands may be of four forms: address variables (verbs), input operands (register variables, whole or half word nouns), output operands (whole or half words) and buffer operands (see discussion of GENERATE BUFFER CONTROL WORD under "Declarators" in this section). Note that index register variables should not be used as operands into which values are to be stored; for example, "find index" in the SEARCH NOT BETWEEN declaration may not be stored in an index register.

8. COMMENTS, ABSOLUTE CODE, AND WRITE PACKAGE

COMMENTS

The purpose of COMMENT statements is to provide the programmer with a means for writing into his NELIAC algorithm any alphanumeric information he considers necessary for the understanding of the dimensioning and program logic. When written in the prescribed format, the COMMENT is ignored by the compiler; it appears only in the input medium (cards, tape, etc.) and on the hard copy listing of the source program. A COMMENT will not be printed at execution time.

The correct form for the statement is

(COMMENT ' ' this is a comment)

Between the double apostrophe and the right parenthesis, any of the NELIAC symbols (except another right parenthesis, of course) may be used to increase program clarity. The statement may be written anywhere in the algorithm, as in the following example:

```
5
(COMMENT ' ' DIMENSIONING 123 * / + - END)
A, B, C $
(COMMENT ' ' PROGRAM LOGIC)
A + B =) C,
(COMMENT ' ' END OF PROGRAM)
. .
```

ABSOLUTE CODE

This element of the NELIAC language is an archaic leftover from the predeclaration days. Although it is still acceptable by most, if not all compilers, the absolute code--commonly referred to as "crutch code" because early compilers were incomplete and had to "lean" on the machine language to provide full capabilities-- is now replaced by the declaration.

The absolute code is very similar to the operand of the MACHINE CODE declarator. There are five octal digits corresponding to the function code f, the designators j, k, and b, followed by the octal sign OCT and the instruction operand Y which may be numerical or a noun, subscripted or not. (Subscripting is limited to constants and index register variables.) The numerical Y has one or more digits which are assumed to be decimal unless indicated as octal. Absolute code may be inserted in the flowchart at any point in the program logic and requires no external punctuation except a comma to set it off from any succeeding instructions. All instructions in the repertoire of the reference computer may be implemented in the NELIAC source program. Following is an example of absolute code:

```
5
A, B, C, D (4),
$
1 =) A =) B,
A + B =) C,
1ØØ3Ø OCT C,
14Ø3Ø OCT D ($ 3 $),
D ($ 3 $) =) A =) B,
1ØØ3Ø OCT B,
26ØØØ OCT ØØ436 OCT,

..
```

WRITE PACKAGE

Declarations and absolute code provide the only means for input communications to the computer in the NELIAC language. However, an additional output programming device--the "write package"--is available for generation of hard copy information.

Basically, the package has two output capabilities: the title literal and the formatted literal. The title literal is a message written by the programmer for use as a heading or title for the anticipated output or as a means for generating error messages. This literal, defined in the dimensioning portion of the flowchart, reflects the precise information requested for output. It has the general form

($ TITLE LITERAL ' ' LS HEADING GR $),

TITLE LITERAL may be any previously unused NELIAC name. HEADING may contain any alphanumeric characters or symbols (except the apostrophe, double asterisk, /, LS , or GR) that the programmer wishes to employ as a title. It is written between the LS and GR punctuation combination. As mentioned previously, the literal is defined in the flowchart dimensioning.

To print the title literal, the programmer uses the statement

WRITE (TITLE LITERAL),

at that point in the program logic where he intends to output the HEADING. TITLE LITERAL therefore acts the same as a noun switch entry; calling the noun from the program logic causes the message defined to be outputted on the high speed printer.

The formatted literal is the more general of the two, since it incorporates the facilities of the title literal. It is formatted in the sense that this literal allows the programmer to output the contents of variables listed in the program logic call according to the format specified in the literal definition in flowchart dimensioning.

The formatted literal definition is an image of the form in which the programmer wishes to output any alphanumeric messages and numerical results. This literal affords, as well, the opportunity to control the printer with three special symbols.

As before, the literal is given a name to identify it for call. Any messages, formatted data output specifications, and control symbols are included between the double apostrophe and the $) combination, as in the following example:

($ FORMATTED LITERAL ' ' DATA FORMAT,
CONTROL SYMBOLS, LS MESSAGES GR $),

DATA FORMAT provides the following data images:

a.   888... (Contents of the variable named in the call outputted in octal notation.)

b.   ∅∅∅... (Contents of the variable named in the call outputted in fixed point decimal notation.)

c.   XXX... (Contents of the variable named in the call outputted in alphanumeric.)

In all of the above images, the number of characters written in the format dictates the number of spaces on the printed line that the compiler will reserve for the contents of the call variable. The programmer must insure that the proper format length is used to prevent data truncation upon output. If the outputted data does not fill the field reserved, blank spaces will be inserted by the printer. For numeric output a space for a sign must be provided.

d.   ∅∅.∅∅∅... (Contents of the variable named in the call outputted in floating point decimal notation; the number of zeros preceding the decimal point indicates the length of integral portion of the floating point data plus sign that the programmer anticipates outputting; the length of the zeros to the right of the decimal point specifies the degree of fractional accuracy desired.)

e. Ø̸Ø̸.Ø̸Ø̸Ø̸ * Ø̸Ø̸ ... (Contents of the variable named in the call outputted in the engineering notation of floating point decimal numbers; the characters to the left of the asterisk act in a fashion similar to the preceding specification; the zeros to the asterisk's right are the exponent length that the programmer will allow; this format is generally used when the magnitude of results is unknown.)

CONTROL SYMBOLS cause the printer to execute a top of form, or spacing control, or a line skip:

a. ** (Generates a top of form command to the printer.)

b. ' N ' (Generates spacing control--N spaces on a line are inserted in the line output; N is a decimal fixed point number.)

c. / (Generates a line skip, equivalent in theory to a carriage return.)

MESSAGES allow the programmer to insert the equivalent of a title literal within a formatted literal. There is no extraordinary restriction placed on the MESSAGES because of their inclusion in the formatted literal.

The write package provides in addition the capability to repeat any of the data formats, control symbols, or messages in any literal output. The repeat format is

($ NAME ' ' (M ' ' FORMATTED LITERAL) $),

NAME is the noun by which the literal may be referenced. M is a fixed point decimal number which specifies how many times the FORMATTED LITERAL will be repeatedly printed. For example:

.
.
.

(COMMENT ' ' DIMENSIONING)
($ OUTPUT ' ' '2Ø̸' (3 ' ' '8' Ø̸Ø̸) / **
        LS POUNDS PER INCH GR $),

.
.
.

OUTPUT is the literal name; '2∅' indicates twenty spaces;
(3 '' '8' ∅∅) causes eight spaces followed by two fixed point decimal
digits to be repeated three times; / is a line skip and ** a top of
form; "POUNDS PER INCH" is printed on the top of a new form.

Calling the literal with the following statement commands the
output of the contents of the three variables requested.

    .
    .
    .

(COMMENT '' PROGRAM LOGIC)
WRITE (OUTPUT, X, Y, Z),

    .
    .
    .

The result of such a call is shown in figure B8-1.
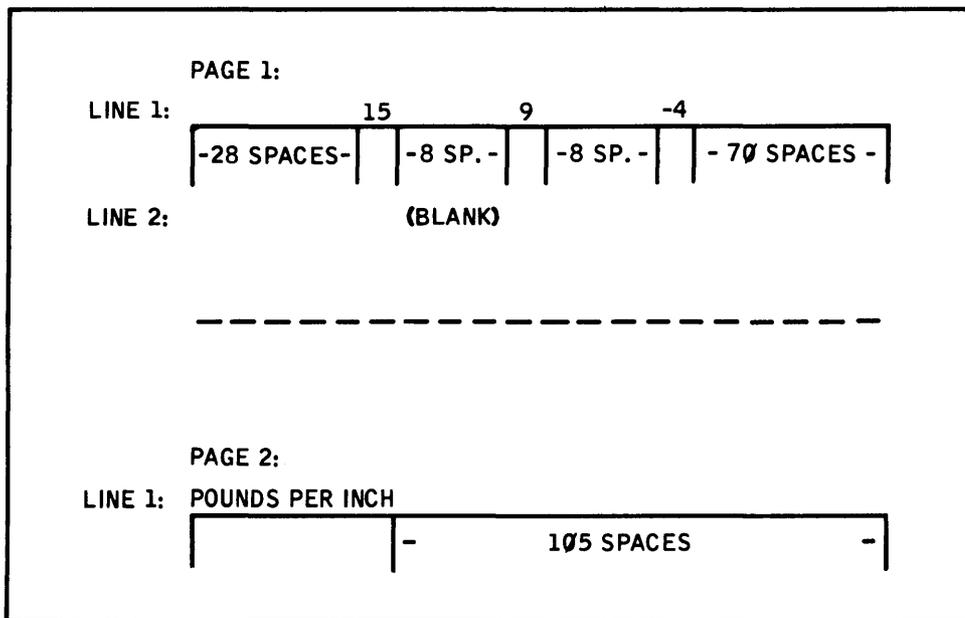

Figure B8-1. Formatted literal output.

9. CASE STUDIES

PROBLEM ONE

Statement

Three simultaneous equations with three unknowns--x, y, and z--are presented for solution. The values of the unknowns are to be found to the nearest integer.

The equations are:

$$X + 2Y = 28$$
$$9X + 5Y + 6Z = 215$$
$$7X + 8Y + 3Z = 181$$

Problem Discussion

Any three simultaneous equations with three unknowns may be represented in the following form:

$$A_0 X + B_0 Y + C_0 Z = D_0$$

$$A_1 X + B_1 Y + C_1 Z = D_1$$

$$A_2 X + B_2 Y + C_2 Z = D_2$$

As any algebraist knows, to find an unknown in terms of known quantities (such as coefficients), the other unknowns in the equations must be eliminated through algebraic manipulation. Once one unknown is found, the other unknowns may be determined through substitution of the new "known" into some intermediate equations.

The equations for solution, then, are:

$$Z = \frac{(D_0 A_1 - D_1 A_0)(B_1 A_2 - B_2 A_1) - (D_1 A_2 - D_2 A_1)(B_0 A_1 - B_1 A_0)}{(C_0 A_1 - C_1 A_0)(B_1 A_2 - B_2 A_1) - (C_1 A_2 - C_2 A_1)(B_0 A_1 - B_1 A_0)}$$

$$Y = \frac{D_0 A_1 - D_1 A_0 - Z(C_0 A_1 - C_1 A_0)}{B_0 A_1 - B_1 A_0}$$

$$X = \frac{D_0 - B_0 Y - C_0 Z}{A_0}$$

There is nothing very complicated about this problem. The reader can already anticipate a fairly simple approach and solution.

To aid in the numerical solution, the constants of the given equations are assigned to the hypothetical coefficients:

$$A_0 = 1, \ B_0 = 2, \ C_0 = 0, \ D_0 = 28$$

$$A_1 = 9, \ B_1 = 5, \ C_1 = 6, \ D_1 = 215$$

$$A_2 = 7, \ B_2 = 8, \ C_2 = 3, \ D_2 = 181$$

Flowchart

The solution presented here (figure B9-1) is one of a large number which could be written to achieve the correct answer. The attack made here is the straightforward approach. Wherever possible, repeated computations are eliminated by preprocessing the coefficients. The algebra is reduced through the use of common terms. The output is printed by means of the write package.

```
5
A(3) EQ 1, 9, 7,
B(3) EQ 2, 5, 8,
C(3) EQ Ø, 6, 3,
D(3) EQ 28, 215, 181,
TERM(6),
($HEADING' ' ** '25' LS OUTPUT FROM NELIAC PROBLEM ONE GR ///
            '29' LS X GR '9' LS Y GR '9' LS Z GR // $),
($OUTPUT' ' '2Ø' (3' ' '8' ØØ) / ** $),
X, Y, Z
$
COMPUTE THE ANSWERS' '
D($Ø$) * A($1$) - D($1$) * A($Ø$) =) TERM($Ø$),
C($Ø$) * A($1$) - C($1$) * A($Ø$) =) TERM($1$),
B($1$) * A($2$) - B($2$) * A ($1$)=) TERM($2$),
D($1$) * A($2$) - D($2$) * A($1$) =) TERM($3$),
C($1$) * A($2$) - C($2$) * A($1$) =) TERM($4$),
B($Ø$) * A($1$) - B($1$) * A($Ø$) =) TERM($5$),
(TERM($Ø$) * TERM($2$) - TERM($3$) * TERM($5$) ) /
(TERM($1$) * TERM($2$) - TERM($4$) * TERM($5$) ) =) Z,
(TERM($Ø$) - Z * TERM($1$) ) / TERM($5$) =) Y,
(D($Ø$) - B($Ø$) * Y - C($Ø$) * Z) / A($Ø$) =) X,
WRITE(HEADING), WRITE(OUTPUT, X, Y, Z),
..
```

Figure B9-1. NELIAC problem one, flowchart.

Flowchart Discussion

This step-by-step assessment of the flowchart is intended to provide a confirmation of the rules developed in the text and to suggest ideas for programmers who may be undecided on directions to take.

a.   5 (Control number; informs compiler that this is to be a process flowchart.)

b.   A(3) EQ 1, 9, 7, (First dimensioning statement; presets the list called A with three fixed point constants.)

c.   B(3) EQ 2, 5, 8, (List B preset with three fixed point constants.)

d.   C(3) EQ ∅, 6, 3, (List C preset with three fixed point constants.)

e.   D(3) EQ 28, 215, 181, (List D preset with three fixed point constants.)

f.   TERM(6), (List called TERM dimensioned as six elements in length; all elements preset to zero.)

g.   ($HEADING' ' ** '25' LS OUTPUT FROM NELIAC PROBLEM ONE GR /// '29' LS X GR '9' LS Y GR '9' LS Z GR // $), (Literal; identified by an address variable, in this case HEADING; double asterisk indicates a top of form command; all numbers enclosed in apostrophes are spacing specifications; any information contained between the LS and GR punctuation is to be outputted on the printer; a slash indicates a line skip.)

h.   ($OUTPUT' ' '2∅' (3' ' '8' ∅∅) / ** $), (Another literal, this one named OUTPUT; twenty spaces; parenthesized specification indicates that information will be outputted via the literal in a format given by the specification; here the 3 indicates that everything following the double absolute sign is to be repeated three times-- eight spaces and a two decimal integer result; this is followed by a line skip and a top of form operator.)

i. X, Y, Z (Three fixed point nouns, all preset to zero.)

j. $ (End of dimensioning, beginning of program logic.)

k. COMPUTE THE ANSWERS'' (Verb; name of the flowchart.)

l. D($Ø$) * A(1) - D(1) * A($Ø$) =) TERM($Ø$), (First line of program logic; computes $(D_0) \cdot (A_1) - (D_1) \cdot (A_0)$ and stores the result in a temporary storage word, $TERM_0$.)

m. C($Ø$) * A(1) - C(1) * A($Ø$) =) TERM (1), (Computes $(C_0) \cdot (A_1) - (C_1) \cdot (A_0)$ and stores the result in a temporary storage word, $TERM_1$.)

n-1. B(1) * A(2) - B(2) * A(1) =) TERM(2),

n-2. D(1) * A(2) - D(2) * A(1) =) TERM(3),

n-3. C(1) * A(2) - C(2) * A(1) =) TERM(4),

n-4. B($Ø$) * A(1) - B(1) * A($Ø$) =) TERM(5), (Computation of $TERM_2$ through $TERM_5$.)

o. (TERM($Ø$) * TERM(2) - TERM(3) * TERM(5)) / (TERM(1) * TERM(2) - TERM(4) * TERM(5)) =) Z, (Computation of the first unknown Z based on the values at the temporary storage locations.)

p. (TERM($Ø$) - Z * TERM(1)) / TERM(5) =) Y, (Computation of the unknown Y based on the temporary storage values and the parameter Z.)

q. (D($Ø$) - B($Ø$) * Y - C($Ø$) * Z) / A($Ø$) =) X, (Computes $(D_0 - (B_0) (Y) - (C_0) (Z))$ divided by A_0 which solves for the last unknown X in terms of the other parameters Y and Z and the equation coefficients.)

r. WRITE(HEADING), WRITE(OUTPUT, X, Y, Z) (Outputs the title literal, the formatted literal, and the three unknowns.) The printed output appears as:

X	Y	Z
14	7	9

s.   .. (Flowchart termination.)

PROBLEM TWO

Statement

This problem concerns a picture window with two parallel glass panes of differing compositions and widths separated by an air space (figure B9-2).
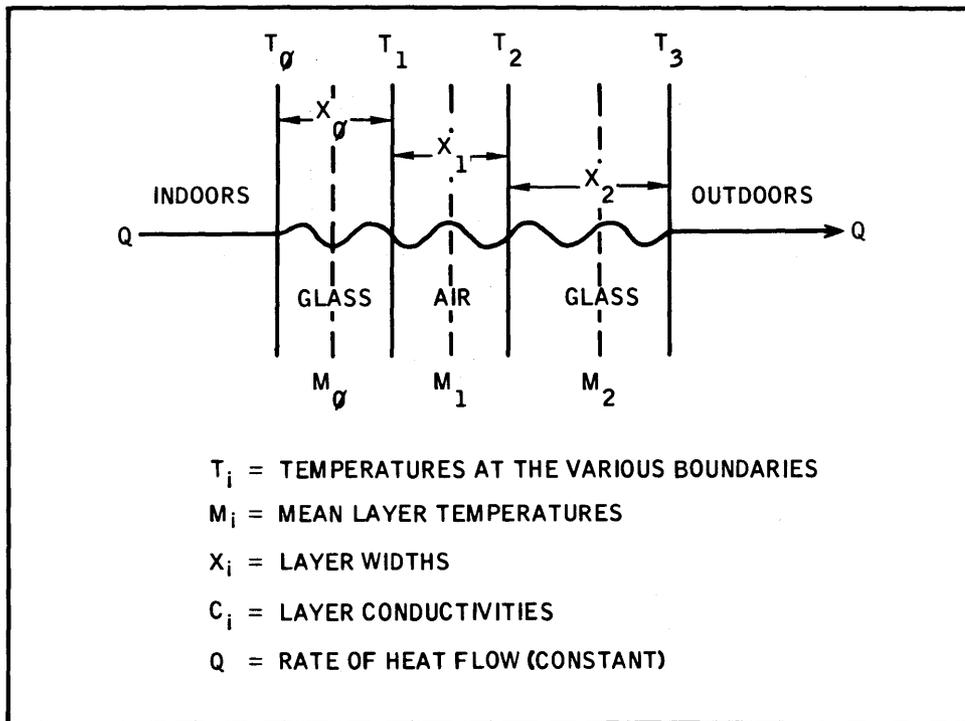

Figure B9-2. NELIAC problem two, diagram.

The indoor and outdoor temperatures, and the layer widths are known; the conductivity of each layer is assumed to be a linear function of the layer's mean temperature. The time is some night during winter when heat flows in the indoor-to-outdoor direction through the glass.

The rate of heat flow and the internal temperatures at the air/glass boundary are to be found and outputted on the high speed printer.

Problem Discussion

This problem is considerably more difficult than the first, but not beyond the reach of a new programmer. There are two reading approaches possible: first, consideration of all aspects: theory, formulae, and fabrication of an algorithm for solution (this is the long way); or, second, use of the algorithm provided toward the end of this discussion. It is, of course, up to the reader which course he takes; the discussion will encompass both approaches.

Since the conductivities (or inverse resistance to heat flow) C_i are assumed to be linearly dependent upon the mean layer temperatures M_i, in general one may state that

$$C_i = A_i \, M_i + B_i \qquad (1)$$

In turn, the mean temperatures M_i are found by averaging the temperatures T_i at the boundaries of each layer:

$$M_i = \frac{T_i + T_{i+1}}{2} \qquad (2)$$

The rate of heat flow Q is the quotient found by dividing the indoor-outdoor temperature differential $(T_0 - T_3)$ by the sum of the quotients formed in the division of each layer thickness X_i by its conductivity C_i:

$$Q = \frac{T_{\emptyset} - T_3}{\left(\frac{X_{\emptyset}}{C_{\emptyset}}\right) + \left(\frac{X_1}{C_1}\right) + \left(\frac{X_2}{C_2}\right)} \tag{3}$$

The known quantities in the equation above are too few: $T_{\emptyset}$, T_3, $X_{\emptyset}$, X_1, and X_2. Somehow the conductivities C_i must be ascertained in order to solve for Q.

The rate of heat flow is known to be constant; if equations are written for the Q across each layer, the following equalities are obtained:

$$Q = \frac{T_{\emptyset} - T_1}{\left(\frac{X_{\emptyset}}{C_{\emptyset}}\right)} = \frac{T_1 - T_2}{\left(\frac{X_1}{C_1}\right)} = \frac{T_2 - T_3}{\left(\frac{X_2}{C_2}\right)} \tag{4a}$$

or, by multiplying the denominator up,

$$Q = \frac{C_{\emptyset}(T_{\emptyset} - T_1)}{X_{\emptyset}} = \frac{C_1(T_1 - T_2)}{X_1} = \frac{C_2(T_2 - T_3)}{X_2} \tag{4b}$$

Substituting the right-hand term of equation (1) for Q, and dropping the middle term of equations (4a) and (4b) which have only one known quantity, the following is obtained:

$$\frac{T_{\emptyset} - T_3}{\left(\frac{X_{\emptyset}}{C_{\emptyset}}\right) + \left(\frac{X_1}{C_1}\right) + \left(\frac{X_2}{C_2}\right)} = \frac{C_2(T_2 - T_3)}{X_2} = \frac{C_{\emptyset}(T_{\emptyset} - T_1)}{X_{\emptyset}} \tag{4c}$$

Solving for the intermediate temperatures T_2 and T_3, equations (5a) and (5b) evolve:

$$T_2 = T_3 + \frac{\left(\frac{X_2}{C_2}\right)(T_{\emptyset} - T_3)}{\left(\frac{X_{\emptyset}}{C_{\emptyset}}\right) + \left(\frac{X_1}{C_1}\right) + \left(\frac{X_2}{C_2}\right)} \tag{5a}$$

$$T_1 = T_\emptyset + \frac{\left(\dfrac{X_\emptyset}{C_\emptyset}\right)(T_\emptyset - T_3)}{\left(\dfrac{X_\emptyset}{C_\emptyset}\right) + \left(\dfrac{X_1}{C_1}\right) + \left(\dfrac{X_2}{C_2}\right)} \tag{5b}$$

In equations (5a) and (5b), note that all unknowns save C_i, the conductivities of each layer, have been eliminated. Reviewing equations (1) and (2), it becomes evident that C_i is dependent upon T_i; conversely, in equations (5a) and (5b), the dependency is reversed. Hence, the conclusion is that there are too many unknowns for so few equations. Consequently, the approach to solution must incorporate an iterative scheme.

The algorithm begins with an initial guess at the interval boundary temperatures T_1 and T_2. Thereafter, the solution proceeds as follows:

   a.   Solve for the mean layer temperatures M_i using
        equation (2).

   b.   Solve for the layer conductivities C_i using equation (1).

   c.   Solve for the rate of heat flow Q using equation (3).

   d.   Solve for a corrected set of internal temperatures T_1
        and T_2 using equations (5a) and (5b).

   e.   Repeat steps a. through d. until the rate of heat flow Q,
        as determined in step c., is approximately equal for two
        successive calculations. When that occurs, the problem
        stands solved.

Indoor temperature $T_\emptyset$ and outdoor temperature T_3 will be $7\emptyset$ and $\emptyset$ degrees, respectively. The widths $X_\emptyset$, X_1, and X_2 will be $\emptyset.25$, $\emptyset.2$, and $\emptyset.15$ inch, in that order. The arbitrary constants A_i and B_i are given as follows:

$$A_\emptyset = .\emptyset\emptyset25, \quad A_1 = .\emptyset\emptyset\emptyset28, \quad A_2 = .\emptyset\emptyset2$$

$$B_\emptyset = .\emptyset419, \quad B_1 = .\emptyset\emptyset36, \quad B_2 = .\emptyset4\emptyset7$$

ORIGINAL

Flowchart

As in the case of the Problem One example, this solution (figure B9-3) is only representative of many which could have been written. The approach is largely straightforward, but a loop is fashioned from entry points and a conditional transfer in order to force repeated execution of the steps outlined in the preceding subsection. Because of the floating point specification of the known data, this flowchart differs in mode from the last. Output again is printed by the write package.

```
5

T(4) EQ 7Ø.Ø, 9Ø.Ø, 8.Ø, Ø.Ø,

A(3) EQ Ø.ØØ25, Ø.ØØØ28, Ø.ØØ2,

B(3) EQ Ø.Ø419, Ø.ØØ36, Ø.Ø4Ø7,

C(3).

MN(3).

X(3) EQ Ø.25, Ø.2Ø, Ø.15,

SAVE. RATE. TEMP 1. TEMP 2. RATE OF HEAT FLOW. ,

($HEAD' ' /// '1Ø' LS INTERMEDIATE RATE OF HEAT FLOW GR /// $),

($HEADING' ' ** '25' LS RATE OF HEAT FLOW IN DEGREES/INCH GR /// $),

($OUTPUT' ' LS RATE OF HEAT FLOW EQ GR '1' ØØ.ØØØØØ /

LS TEMP 1 EQ GR '1' ØØ.ØØØØØ /

LS TEMP 2 EQ GR '1' ØØ.ØØØØØ / $) $

RATE ROUTINE ' '

(T ($Ø$) + T($1$) )/2.Ø =) MN($Ø$),

(T ($1$) + T($2$) )/2.Ø =) MN($1$),

(T ($2$) + T($3$) )/2.Ø =) MN($2$),

A($Ø$) * MN($Ø$) + B($Ø$) =) C($Ø$),

A($1$) * MN($1$) + B($1$) =) C($1$),

A($2$) * MN($2$) + B($2$) =) C($2$),

(T ($Ø$) - T($3$) )/( (X ($Ø$) / C($Ø$) ) + (X ($1$) / C($1$) ) + (X ($2$) /

C($2$) ) ) =) RATE,
```

Figure B9-3. NELIAC problem two, flowchart.

```
RATE - SAVE LS 0.00001 ' ' SAVE - RATE LS 0.00001 ' ' STOP . $ $ $

( (X ($2$) / C($2$) ) * (T ($0$) - T($3$) ) / ( (X ($0$) / C($0$) ) +

(X ($1$) / C($1$) ) + (X ($2$) / C($2$) ) ) ) + T($3$) =) T($2$),

T($0$) - ( (X ($0$) / C($0$) ) * (T ($0$) - T($3$) ) / ( (X ($0$) / C($0$) )

+ (X ($1$) / C($1$) ) + (X ($2$) / C($2$) ) ) ) =) T($1$),

T($1$) =) TEMP 1, T($2$) =) TEMP 2, RATE =) RATE OF HEAT FLOW,

WRITE(HEAD), WRITE(OUTPUT, RATE OF HEAT FLOW, TEMP 1, TEMP 2, ),

RATE =) SAVE,

RATE ROUTINE.

STOP' '

T($1$) =) TEMP 1, T($2$) =) TEMP 2, RATE =) RATE OF HEAT FLOW,

WRITE(HEADING), WRITE(OUTPUT, RATE OF HEAT FLOW, TEMP 1, TEMP 2),

. .
```

Figure B9-3. (Continued)

Flowchart Discussion

Many programming techniques in this flowchart are similar to those used in the solution of the previous Problem One. Although the techniques themselves are listed, discussions of these techniques are not repeated here; refer to the "Flowchart Discussion" for Problem One if this information is required.

a. 5

b. T(4) EQ 70.0, 90.0, 8.0, 0.0,

c. A(3) EQ 0.00025, 0.00028, 0.002,

d. B(3) EQ 0.0419, 0.0036, 0.0407,

e. C(3). (Floating point mode established; three zeros stored.)

f. MN(3).

g. X(3) EQ 0.25, 0.20, 0.15,

h. SAVE. RATE. TEMP 1. TEMP 2, RATE OF HEAT FLOW. ,

i. ($HEAD'' /// '1∅' LS INTERMEDIATE RATE OF HEAT FLOW GR /// $),

j. ($HEADING'' ** '25' LS RATE OF HEAT FLOW IN DEGREES/INCH GR /// $),

k. ($OUTPUT'' LS RATE OF HEAT FLOW EQ GR '1' ∅∅.∅∅∅∅∅ / LS TEMP 1 EQ GR '1' ∅∅.∅∅∅∅∅ / LS TEMP 2 EQ GR '1' ∅∅.∅∅∅∅∅ / $) $ (The ∅∅.∅∅∅∅∅ format specification informs the compiler that a maximum positive integral number of 99 or a maximum negative integral number of –9 is anticipated; five decimal places of accuracy are requested.)

l. RATE ROUTINE'' (Entry point for the iterative procedure.)

m-1. (T ($∅$) + T(1))/2.∅ =) MN($∅$),

m-2. (T (1) + T(2))/2.∅ =) MN(1),

m-3. (T (2) + T(3))/2.∅ =) MN(2),

m-4. A($∅$) * MN($∅$) + B($∅$) =) C($∅$),

m-5. A(1) * MN(1) + B(1) =) C(1),

m-6. A(2) * MN(2) + B(2) =) C(2), (Generation of mean temperatures and conductivities.)

n. (T ($∅$) – T(3))/((X ($∅$) / C($∅$)) + (X (1) / C(1)) + (X (2) / C(2))) =) RATE, (Computation of Q.)

o. RATE – SAVE LS ∅.∅∅∅∅1'' SAVE – RATE LS ∅.∅∅∅∅1 '' STOP . $ $ $ (Comparison statement which checks to see if the new and previous values of the rate of heat flow are nearly equal, within a tolerance of 1 x 1∅$^{-5}$; true alternative causes jump to end of routine; false or partially false alternatives all cause regeneration of the intermediate boundary temperatures and an intermediate printout.)

p.   ((X (2) / C(2)) * (T ($∅$) - T(3))/((X ($∅$) / C($∅$)) + (X (1) / C(1)) + (X (2) / C(2)))) + T(3) =) T(2),

q.   T($∅$) - ((X ($∅$) / C($∅$)) * (T ($∅$) - T(3)) / ((X ($∅$) / C($∅$)) + (X (1) / C(1)) + (X (2) / C(2)))) =) T(1),

r.   T(1) =) TEMP 1, T(2) =) TEMP 2, RATE =) RATE OF HEAT FLOW,

s.   WRITE(HEAD), WRITE(OUTPUT, RATE OF HEAT FLOW, TEMP 1, TEMP 2,),

t.   RATE =) SAVE,

u.   RATE ROUTINE. (Direct jump to the entry point for the iterative procedure.)

v.   STOP' ' (End of routine; entry point.)

w.   T(1) =) TEMP 1, T(2) =) TEMP 2, RATE =) RATE OF HEAT FLOW,

x.   WRITE(HEADING), WRITE(OUTPUT, RATE OF HEAT FLOW, TEMP 1, TEMP 2),

y.   ..

A sample intermediate solution follows:

    INTERMEDIATE RATE OF HEAT FLOW

    RATE OF HEAT FLOW = 4.46972

    TEMP 1 = 65.38∅6∅

    TEMP 2 = 13.76712

The rate of heat flow, correct to five decimal places, is given in the final printout:

RATE OF HEAT FLOW IN DEGREES/INCH

RATE OF HEAT FLOW = 3.88173

TEMP 1 = 65.4$\emptyset$417

TEMP 2 = 11.21554

A number of improvements could have been made. For example, the generation of the mean temperatures and conductivities could have been relegated to functions, and the iterative scheme reduced from four instructions to two: the computation of Q, and the generation of improved values for the internal temperatures. However, this was not done because it would have changed the originality of the NELIAC program produced by a new programmer on the third try, which speaks well for the language.