FP6000
computer system
AUTOCODER

Ferranti ELECTRONICS

A DIVISION OF FERRANTI-PACKARD ELECTRIC LIMITED

INDUSTRY STREET, TORONTO 15 - ONTARIO - CANADA

St. John's
Halifax
Saint John
Sherbrooke
Trois-Rivières
Chicoutimi
Montreal
St. Catharines
Fort William
Winnipeg
Regina
Calgary
Vancouver

MANUAL Q.153

FP6000 COMPUTER SYSTEM

AUTOCODER

Ferranti ELECTRONICS
A DIVISION OF FERRANTI-PACKARD ELECTRIC LIMITED
Industry Street, Toronto 15, Ontario, Canada

AUTOCODER

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

AUTOCODER constitutes a language for coding programs for the FP6000 computer, and a compiler for converting these programs to machine language.

1.1 FEATURES

As with other programming languages, AUTOCODER allows the use of symbolic names for instructions, data, etc., thereby making programs both easier to write and to read. AUTOCODER relieves the programmer of many details, such as specifying the actual memory locations to be used for each of a program's components.

Some types of AUTOCODER words produce several machine words, resulting in less writing for the programmer and therefore less opportunity for error. However, the correspondence of AUTOCODER to machine language is such that virtually all the hardware capabilities are still available to the programmer.

In developing AUTOCODER, every effort has been made to make it an integral part of the FP6000 programming system. For instance, compatibility with FP6000 FORTRAN is such that a section of program written in AUTOCODER can be combined, after compilation, with a section written in FORTRAN, to form a single executable program.

AUTOCODER is capable of generating calling sequences for the FP6000 Input/Output Package. This means that the programmer can avoid the FP6000 Input/Output instructions by stating his input-output requirements using AUTOCODER format and item descriptions. The necessary input-output and conversion instructions are then supplied in the form of the Input/Output Package.

1.2 CODING FORM

AUTOCODER programs are written on special coding forms (as shown overleaf) which are divided into horizontal lines and vertical columns. The columns are numbered 1 to 72 and are divided into groups representing "fields" of AUTOCODER words, as explained below. Each field except the OPERAND, is written left-justified. In the OPERAND field, with one exception (mentioned later), blank columns are ignored.

1.2.1 Instruction Statements

Each AUTOCODER instruction is written, as a statement, on a separate horizontal line and is divided into fields, as follows:-

LABEL (columns 1 - 5)

If used, this field contains the name by which other instructions may refer to this one.

OPERATION (columns 7 - 11)

This field contains the operation code, which indicates the type of instruction.

ACCUMULATOR or X-field (columns 13, 14)

This field usually indicates which accumulator(s) are to be used by this instruction.

OPERAND (columns 16 - 72)

This field is of variable length and usually indicates the data involved in the operation.

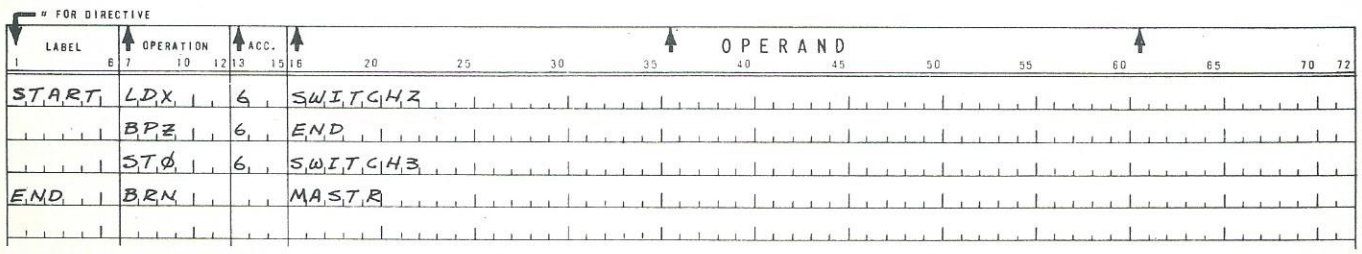


Fig. 1-1: INSTRUCTION STATEMENTS

Ferranti

FP6000 AUTOCODER CODING SHEET

Date .... Page ....

PROGRAM NAME .... Serial 73 ... 80

* FOR DIRECTIVE

LABEL	OPERATION	ACC.	OPERAND

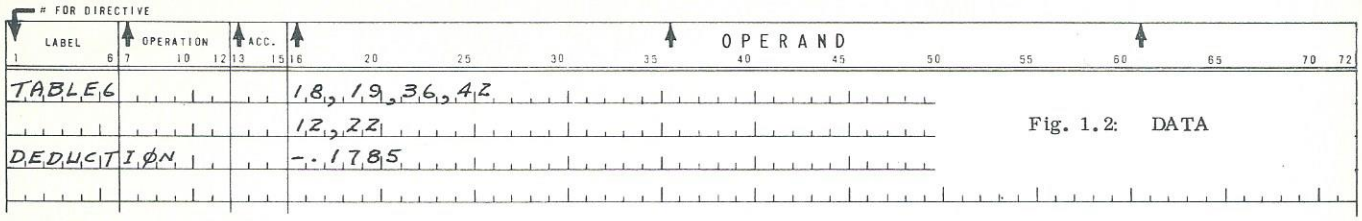


Fig. 1.2: DATA

1.2.2 Data

When data is written on the coding form, more than one item may be written on each line. If this is done, each item, except the last one on the line, is followed by a comma.

LABEL (columns 1 - 11)

If a name is written in this field, then it pertains to the first item of data on the line.

OPERAND (columns 16 - 72)

This field contains the items of data, written left to right. The number of items which may appear on one line is governed only by the length of the field (57 columns).

1.2.3 Directives

Although the rules for converting from AUTOCODER language to machine language are essentially fixed, the programmer is allowed some flexibility. Moreover, the compiler requires a minimum amount of information about the program to be compiled, such as peripherals used, program name, etc. Both requirements are satisfied through the use of directives, which are instructions to the compiler, ie. they are not included in the compiled program.

Each directive occupies one line on the coding form, although some types of directives are to be followed by other lines of information.

LABEL (columns 1 - 11)

This field always starts with the character # in column 1, followed by a name indicating the type of directive.

OPERAND (columns 16 - 72)

Some types of directives require details to be placed in this field.

1.3 THE COMPILER

A program in its symbolic or compiler language form is usually referred to as a source program. When the source program has been punched on cards (one card per line of coding), or on paper tape (each line preceded by a "New line" character), it is then ready for compilation into machine language. Thus, the compiler is a special program which accepts source programs (punched on cards or paper tape) as input and produces corresponding machine language programs as output. This output is usually referred to as an object program.

The FP6000 AUTOCODER compiler accomplishes this conversion from source to object program in one pass; in other words, it forms and writes out the object program while reading in the source program, as opposed to reading in the entire source program before writing out any of the object program. Whether produced on cards or paper tape, the object program is in a form suitable for loading into the computer and executing. This includes information about the program required by the FP6000 EXECUTIVE, such as the amount of storage used, the number of peripheral units used, etc.

As an additional output, a listing showing the correspondence between the source program and the object program, is provided. Whether this listing is produced simultaneously with the object program or in a separate pass, depends on the number of peripheral units available for the compiling job.

NOTE:

The remaining chapters in this manual assume that the reader is familiar with the FP6000 computer, as outlined in the FP6000 Programming Manual, ref. Q.144.

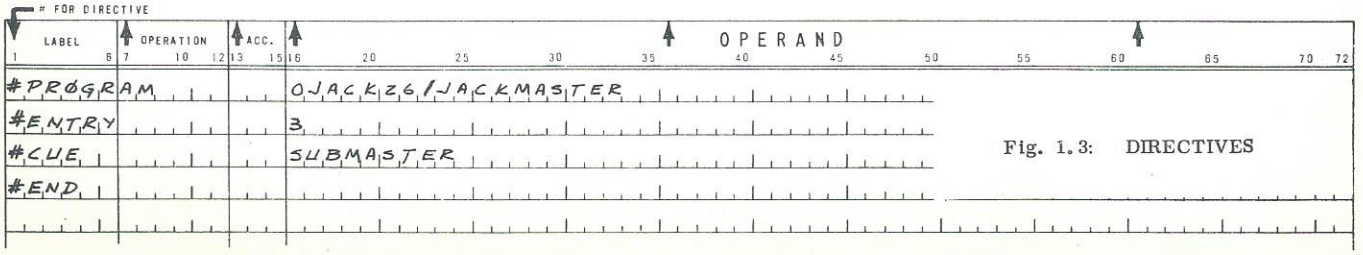


Fig. 1.3: DIRECTIVES

CHAPTER 2

SYMBOL DEFINITION

TABLE OF CONTENTS

CHAPTER 2

SYMBOL DEFINITION

2.1 USE OF SYMBOLS IN AUTOCODER

The basic element of any programming language is the symbol, which is some combination of characters representing a name. In AUTOCODER specifically, symbols are combinations of alphabetic and numeric characters, of which the first character must be alphabetic. Special characters such as comma, point, etc., are not allowed. Names thus formed usually represent one of the following four types:-

(1) Operation codes

(2) Directives

(3) Memory locations

(4) Peripheral units

The process of compilation consists, essentially, of interpreting these symbols and, where applicable, converting them to their machine language equivalents. Each symbolic operation code corresponds to a specific machine function code (or codes) and each directive represents a special action to be taken by the compiler. The actual names used for these two purposes have been assigned and are a permanent part of the AUTOCODER language. Location names, on the other hand, are assigned by the programmer. The actual location represented by a name depends on where and how the name is used in the source program.

2.2 SYMBOLIC LOCATIONS

The memory locations to be used by the object program are divided into three areas:-

(1) Lower data

(2) Program

(3) Upper data

The lower data area must lie completely within the first 4096 locations used by the program, whereas the other two areas may lie beyond this point. This means that, in machine language, lower data locations can be represented by 12 bits, while the others are represented by 15 bits.

As the compiler encounters symbolic locations in the source program, it must assign a numeric value to each one. This involves determining the area into which each one should be placed, and its relative position within that area. The rules by which this is done are as follows:-

2.2.1 Program Locations

As the compiler reads in source program instructions, it assigns locations in the program area to them, in consecutive order. Any symbol appearing in the LABEL field of an instruction (or of data if in the body of the program) will be given a value equal to its location (or first location, if it occupies more than one). When the symbol is encountered in an operand field it is replaced by this value. Sometimes, of course, the appearance of a symbol in an operand field precedes its "defining" appearance in a label field. In any case, a symbol is recognized as referring to a program location:-

(a) if it appears in a label field in the program section, or,

(b) if it appears in the operand field of a branch instruction, whichever occurs first.

Since references to program locations will be almost exclusively in the operands of branch instructions, and since the structure of FP6000 branch instructions allows for 15-bit addresses, the program area can be in upper memory. This means that the entire lower memory (if required) is available for data which can then be referred to directly in instructions having only a 12-bit address. In the few instances where it is necessary to refer to a program location in one of these instructions, it must be done indirectly (ie. using an index register). For this purpose there is a special AUTOCODER instruction for loading the index register, which is treated as a branch-type in-so-far as its operand is recognized as a program location, rather than data. (See Chapter 4-4).

2.2.2 Lower Data

Since all FP6000 instructions (except the branch type) have a 12-bit address, data locations may be referred to directly only if they are in lower memory (ie. below 4096). Consequently, the programmer will probably want all of his data, except for tables and arrays, etc., placed in the lower area. To facilitate this, the AUTOCODER compiler will treat as referring to a lower data location, any symbol:-

(a) which is encountered in the operand field of other than a branch instruction, and

(b) which has not previously appeared under some defining directive.

When each such symbol is first encountered by the compiler, space is allocated, in lower memory, as follows:-

(i) two consecutive locations if it is in the operand of a double-length macro-instruction (the symbol is given the value of the first location).

(ii) one location, if it is in any other type of operand.

This procedure eliminates the necessity for the programmer to state explicitly the data locations he wishes to have in lower memory, but does not give the programmer much control. If the programmer wishes to control the order in which any of his data lies in lower memory, he may do so by grouping the corresponding symbolic locations under the directive described next.

Directive I: LOWER

(1) This directive indicates that the symbols following in the lines below it, are to be allocated to lower memory.

```
#LOWER
```

(2) The symbols can be up to 11 characters each and are written in the order which their corresponding locations are to take. Only the operand field of each line is used, with successive symbols on the same line being separated by commas.

Example:

```
TOTAL,COUNT,ABCD123
T1,T2,T3,SUM
```

(3) Successive names correspond to adjacent locations in lower memory, except where a name is followed by a decimal integer enclosed in parentheses. In this case, n locations are reserved for the name, where n is the integer written. The value assigned to this name is that of the first of these locations and the value of any name following is n greater. This allows the programmer to allocate space for tables, where items are referred to, relative to the first one.

Example:

```
TOTAL,COUNTS(20),ABCD123
T1,T2,T3,SUM(125)
```

(4) Sometimes the programmer wishes to assign two or more names to a single location. This allows him to use a memory location for more than one item of data, or to refer to the same item of data by more than one name. To indicate this to the compiler, the names involved are written successively separated by equal signs instead of commas.

Example:

```
TOTAL=COUNT=SUM,ABCD123
T1,T2=T3
```

As with single names, a group of equivalent names may be followed by an integer enclosed in parentheses, to indicate the number of locations to be reserved. However, all of these names will refer to the first of these locations.

Example:

```
TOTAL=COUNT=SUM(20),ABCD123
T1,T2=T3(125)
```

2.2.2 continued

(5) In situations where the programmer cannot state the dimension (ie., the number of cells to be allocated) for a symbol as an absolute integer, he may use a symbolic expression. All of the symbols in the expression must have been previously defined, and it may take any of the forms of absolute expressions allowed with a DEFINE directive (see Section 2.3).

Examples:

```
COUNTS(B2-B1+60),T1,T2
T3=T4=T5(A+#227),SUM(20)
BLOCK(B2-B1/2)
```

(6) A comma is implied between the last item of one line and the first item of the next line. Therefore, each name or group of equivalent names, with or without a dimension, must be complete on one line. However, there is no restriction on the number of complete items which can be written on one line, except the length of the operand field (57 characters).

(7) LOWER may appear any number of times in a source program with the following restrictions:-

(a) The appearance of a name under a LOWER directive must precede its use in the program section.

(b) Each LOWER directive indicates a new section of lower memory which is not necessarily adjacent to that formed by the previous LOWER directive.

(8) Since the appearance of a symbol under this type of directive constitutes a definition of that symbol, a second appearance of the same symbol under this or any other "defining" directive would be ambiguous and would be treated as an error.

2.2.3 Upper Data

Since data stored in upper memory (above 4095) cannot be referred to directly, the programmer will probably restrict the use of this part of the memory to the storage of data which must, of necessity, be referred to through index registers. This includes tables, arrays, etc. whose individual locations are computed in index registers, as in program loops, etc.

Unlike the procedure for lower data, all data to be placed in upper memory must be explicitly stated as such, by appearing under the proper directive (described below).

Directive II: UPPER

(1) As with LOWER, the directive itself is written on a separate line, with a blank operand field.

```
#UPPER
```

(2) The rules for writing and interpreting the defining lines under the directive are the same as for LOWER, except, of course, that all names are allocated to upper memory.

2.3 SPECIAL SYMBOL DEFINING DIRECTIVES

Normally a programmer will define symbols through the use of a LOWER or UPPER directive or by writing the symbol, in the label field at some AUTOCODER line. However, symbols may be specially defined by the use of the directives to be described.

One use for specially defined symbols might be in the operands of AUTOCODER instructions which do not refer to storage locations. These operands will usually be written as simple integers without the use of symbols. This includes the operands of shift instructions (number of places to shift), operands of small-integer instructions (which are themselves data), etc. Often the same number will be repeated many times in such operands, and for this or some other reason, the programmer may wish to use a symbol instead, where the numerical value of the symbol is defined elsewhere. This means that to change this value before compilation he need only change the symbol definition and not each of the operands involved.

In addition the following directives may be used to define symbols whose definitions must be expressed as functions of other UPPER or LOWER symbols (such as differences, etc.)

2.3 continued

Directive IV (a): DEFINE

(1) The directive is written on one line in the following general form:

#DEFINE Symbol = Expression

(2) "Symbol" consists of 1 to 11 characters adhering to the AUTOCODER format (1st character is alphabetic, etc.).

The effect of this directive is to give this symbol a value equal to that of the "Expression" appearing to the right of the equal sign.

(3) "Expression" can take one of several forms:

An absolute expression consists of any number of decimal integers, octal integers, and symbols, each integer or symbol being preceded by a plus or minus sign.

Furthermore, each symbol in the expression:

(a) must be accompanied by another symbol of the same group, which is preceded by the opposite sign, or

(b) must have been itself set equal to an absolute expression in some previous defining directive.

In this context, two symbols can be considered of the same group only if:-

(a) they have both been previously defined under the same LOWER or UPPER directive, or

(b) they have both previously appeared in LABEL fields of the program area.

Examples:

```
#DEFINE   PLACES=8
#DEFINE   NUM=-3+PLACES
#DEFINE   BASE=#76
#DEFINE   L=TABLE2-TABLE1+BASE-5
#DEFINE   M=-L+#72
```

NOTES:

(i) If the first term of the expression is not preceded by a sign, a plus sign is assumed.

(ii) As will be explained in Chapter 5, the first character of an octal integer is "# ".

(4) Let us represent any absolute expression of the type defined in (3) above, by the character A. Then the "Expression" to the right of the equal sign may also take the forms:-

A * I
A / I

where I is any positive decimal or octal integer; * denotes that the value represented by A is to be multiplied by I; and / denotes that the value represented by A is to be divided by I, giving an unrounded integral value.

This form of expression is also absolute.

Examples:

```
#DEFINE   SIZE2=TABL2-TABL1*2
#DEFINE   SIZE3=SIZE2*4
#DEFINE   HALF=BLKB-BLKA/2
```

(5) A relative expression is of the same form as A, but preceded by any non-absolute symbol, S.

The symbol appearing to the left of the equal sign, will subsequently be considered of the same group as S.

Examples:

```
#DEFINE   ELEMENT=ARRAY+10
#DEFINE   VALUEA=TABLE-10
#DEFINE   AREA2=AREA1+BLKB-BLKA
```

(6) Let us represent any relative expression of the type defined in (5) above, by the character R. Then the "Expression" to the right of the equal sign may also take the forms:-

A * I + R

- A * I + R

A / I + R

- A / I + R

where A is as defined above (the A in -A should not start with a sign); and the value of R is added to the value of the leftmost-part which is evaluated as explained in (4) above.

2.3 continued

Examples:

```
#DEFINE    EL=SIZE2*4+ARRAY+10
#DEFINE    AR2=BLKA-BLKB/2+AR1
#DEFINE    ADDR=PLACES/3+NAME-#100
```

The symbol appearing to the left of the equal sign, will subsequently be considered of the same group as R.

(7) All symbols appearing in "Expression" (ie. to the right of the equal sign) must be previously defined.

(8) The final value of "Expression" must be in the range 0 to +32767 (during evaluation a full 24-bit word is used).

(9) The symbol appearing to the left of the equal sign, must not have previously appeared in the program, and cannot subsequently be re-defined by the use of other defining directives or in any other manner.

Directive IV (b): SET

(1) The directive is written on one line in the following general form: -

#SET	Symbol = Expression

(2) As with DEFINE, the effect is to give the "Symbol" a value equal to that of the "Expression".

(3) "Expression" can take all of the forms allowed with DEFINE.

(4) Any symbol defined by a SET directive may be re-defined by any number of other SET directives, provided that each new definition places the symbol in the same groups. It may not, however, be defined by another type of directive, or in any other manner. The value used for the symbol at a given point in the program is that set by its latest appearance in a SET directive.

CHAPTER 3

INSTRUCTION STATEMENTS

CHAPTER 3

TABLE OF CONTENTS

CHAPTER 3

INSTRUCTION STATEMENTS

This chapter deals with instruction statements that create only one machine instruction. The first part of this chapter describes instruction statements in general terms, while the latter part gives details of statements related to specific groups of instructions.

3.1 GENERAL FORMAT OF INSTRUCTION STATEMENTS

AUTOCODER instruction statements will result in the generation of machine instructions. Each statement is divided into four fields but, as explained below, all fields are not always present. The four fields are written in the following order.

LABEL OPERATION ACCUMULATOR OPERAND

3.2 LABEL FIELD

3.2.1 Form

A LABEL can be any symbol of 1 to 5 alphanumeric characters, the first character always being alphabetic.

Examples:

T116
START
TAG6

3.2.2 Use

When written, a label is given a value equal to the location of the instruction being generated. This label can therefore be used to refer to (the location of) this instruction by writing this symbol in the operand field of other instructions. The label field may be left blank.

3.3 OPERATION FIELD

3.3.1 Form

This is a symbol of 1 to 5 alphabetic characters, or a 3-digit octal number.

Examples:

000
LDX
MPY
SUSBY
040

3.3.2 Use

The operation field must be present in instruction statements and is used to define the machine function. Normally it would contain one of the mnemonic operation symbols indigenous to AUTOCODER, however, where applicable, the 3-octal digit code representing the machine function may be used. In the examples above MPY and 040 are synonymous, both specifying a specific type of multiplication function.

All basic mnemonic operation codes indigenous to AUTOCODER are listed later in this chapter, but it might be well to note now that a mnemonic for a double length operation will generally duplicate the mnemonic code of a corresponding single length instruction. This ambiguity is removed by specifying two accumulators in the accumulator field for the double length instruction, as opposed to one for the single length instruction.

3.4 ACCUMULATOR (or X) FIELD

3.4.1 Form

This may have 1 or 2 octal digits, left justified.

3.4.2 Use

Normally each octal digit refers to an accumulator to be used in the operation of the instruction. Some instructions always use the same accumulator or no accumulators, and in these cases, the X field is left blank. A few instructions use the X field for other purposes.

3.5 OPERAND FIELD

The operand results in the generation of the address portion (ie. N field) and, if applicable, the generation of the modifier portion (ie. M field) of the instruction word. Many types of operands are allowed but the type used will relate to the operation to be performed.

3.5.1 Decimal Operand

The operand of an instruction statement may be an unsigned decimal integer. Note that a blank operand is treated as a zero.

Examples:

```
      LDN   1   4
      ADN   1   10
      LDN   2
```

3.5.2 Octal Operand

An octal operand is an unsigned octal integer preceded by the character #.

Examples:

```
      ANDN  1   #77
      ERN   2   #6666
```

3.5.3 Symbolic Operand

As was described in Chapter 2, symbols may be used to represent memory locations or data. Thus an operand may be a symbol consisting of 1 to 11 alphanumeric characters, the first character always being alphabetic.

Examples:

```
      LDX   4   TIME
      LDX   5   V1
```

The symbolic operand may also be adjusted by following it with a signed decimal or octal integer, in which case the operand defines not the location (or value) represented by the symbol, but that location (or value) incremented or decremented by the adjusting number, according to sign.

The value of the adjusting integer must be in the range $-2^{+23} \leqslant$ integer $\leqslant 2^{23}-1$.

Examples:

```
      LDX   4   TIME+1
      ADX   5   V1-2
      SBX   6   V2-#77
```

3.5.4 Literals

Literals are detailed in Chapter 5. Literal operands always begin and end with a quotation mark.

Example:

```
      LDX   1   '+10000'
```

3.5.5 Operands for Non-branch Instructions

Non-branch instructions have a 12-bit address. Their operands can be decimal, octal, or symbolic, with the following restrictions:-

(1) A symbol used with a symbolic operand must be a symbol referring to lower data (see 2.2.2), or a symbol defined by a PERIPHERAL, DEFINE, or SET directive (see 2.4).

(2) The final "decimal value" of the operand must be in the range $0 \leqslant$ value $\leqslant 4095$, and a value greater than this will be reduced modulo 4096.

Non-branch instructions also have a 2-bit modifier portion. Thus, as required, their operands may be followed by (M), where M is one of the digits 1, 2, or 3 giving the address of the index register (or modifier) to be used when executing the instruction.

Examples:

```
      STO   4   1,(1)
      LDX   4   #77,(2)
      SLC   5   (3)
      ADS   6   ARRAY(2)
      LDX   7   ARRAY+1(2)
```

NOTE:

The context of ARRAY + 1 (2) is (ARRAY + 1) (2).

The operand of non-branch instructions may also be a literal, as described in Chapter 5. When using a literal the operand cannot indicate a modifier (ie. the literal cannot be followed by a digit enclosed within brackets).

3.5.6 Operands for Branch Instructions

Branch instructions have a 15-bit address. Their operands can be decimal, octal, or symbolic, with the following restrictions:-

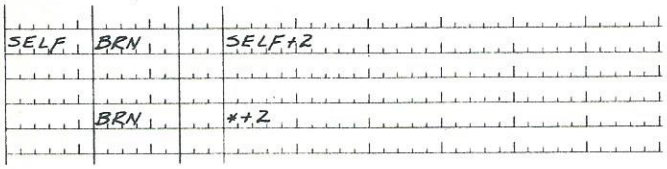
(1) A symbol used with a symbolic operand must be a symbol referring to a program location (see 2.2.1).

(2) The final "decimal value" of the operand must be in the range $0 \leqslant$ value $\leqslant 32767$, and a value outside these limits will be reduced modulo 32768.

Branch instructions do not have a modifier portion, thus a modifier address cannot be indicated (If the operand ends with a bracketted digit, then this digit will be ignored by AUTOCODER).

The symbol * is used only with branch instructions and always takes on the value of the location of the instruction in which it occurs.

For example, the two terms shown below both specify an unconditional branch (BRN) to the second word following the branch instruction.



The literal is not used with branch instructions.

3.6 INSTRUCTION STATEMENTS DETAILED BY GROUPS

The remainder of this chapter lists the various Instruction Statements and relates the mnemonic operation symbols to the corresponding FP6000 function code. The whole is divided into groups of related instructions and a brief description follows each group. Further details of each function can be obtained by referring to the FP6000 Programming Manual.

Instructions are listed, as shown below, with their actual mnemonic operation symbols adjacent to the octal function code and with pseudo symbols for the accumulator and operand fields.

Certain conventions apply to the pseudo symbols used for the accumulator and operand fields. Thus, for the accumulator field, X will indicate one and only one accumulator to be specified when writing these instruction statements. XX* will indicate a double length operation, and will require that two accumulators, X and (X + 1), be specified.

For the operand field, the use of N(M) indicates that the instruction has a 12-bit address plus a 2-bit modifier portion (see 3.5.5). Unless otherwise stated, N(M) may be taken to represent a data location; the cases where N(M) represents a number to be actually operated on will be stated explicitly.

When an instruction is shown with only N in the operand field, then this instruction has a 15-bit address (see 3.5.6).

Function	Mnemonic	Accumulator	Operand	Description
000	LDX	X	N(M)	Load into X

The accumulator symbol represents an octal digit and, by convention, the symbol X is generally used. Similarly, N(M) is generally used to represent some form of operand.

3.6.1 Arithmetic into Accumulators

000	LDX	X	N(M)	Load into X
001	ADX	X	N(M)	Add into X
002	NGX	X	N(M)	Negate into X
003	SBX	X	N(M)	Subtract from X

The instructions of this group can set V (overflow) but not C (carry). They are used with single length numbers or the most significant word of a multiple length number. They take the data word at location N(M), add to this the contents of C, and operate as indicated into accumulator X. The data word at N(M) is left undisturbed.

Without specifying a modifier, these can only refer to lower data, but with the use of the modifier they can refer to any data.

Examples:

```
        LDX   4   LØWERDATA1
        ADX   4   LØWERDATA1,2
        NGX   5   DATA(2)
        SBX   5   LØWERDATA2-4(2)
```

3.6.2 Arithmetic into Store

010	STO	X	N(M)	Store
011	ADS	X	N(M)	Add to store
012	NGS	X	N(M)	Negate into store
013	SBS	X	N(M)	Subtract from store

This group is essentially the converse of the above (3.6.1). They can set V but not C. They take the contents of accumulator X, add to this the contents of C, and operate as indicated into the data location N(M). The data in accumulator X is left undisturbed.

3.6.3 Arithmetic with Small Integers

100	LDN	X	N(M)	Load N into X
101	ADN	X	N(M)	Add N to X
102	NGN	X	N(M)	Negate N into X
103	SBN	X	N(M)	Subtract N from X

This group is the same as the group of 3.6.1, except that the data used is not that at the location N(M), but is the actual value of N(M). Thus, each take the value of N(M), add to this the contents of C, and operate as indicated into accumulator X.

3.6.4 Arithmetic with Carry

004	LDXC	X	N(M)	Load into X and set carry if appropriate
005	ADXC	X	N(M)	Add to X and set carry if appropriate
006	NGXC	X	N(M)	Negate into X and set carry if appropriate
007	SBXC	X	N(M)	Subtract from X and set carry if appropriate
014	STOC	X	N(M)	Store and set carry if appropriate
015	ADSC	X	N(M)	Add to store and set carry if appropriate
016	NGSC	X	N(M)	Negate into store and set carry if appropriate
017	SBSC	X	N(M)	Subtract from store and set carry if appropriate
104	LDNC	X	N(M)	Load N and set carry if appropriate
105	ADNC	X	N(M)	Add N and set carry if appropriate
106	NGNC	X	N(M)	Negate N and set carry if appropriate
107	SBNC	X	N(M)	Subtract N and set carry if appropriate

The above groupings correspond to the first three groups respectively, except they may set C but not V and are designed to be used with less significant parts of multiple length words. They always produce a result which has a zero in the sign bit.

3.6.5 Multiplication

040	MPY	X	N(M)	Multiply
041	MPR	X	N(M)	Multiply and round
042	MPA	X	N(M)	Multiply and accumulate

Multiplication instructions multiply the contents of location N(M) by the contents of accumulator X to produce a double word result in X and X + 1. The data at N(M) is left undisturbed. Rounded multiply adds 2^{-24} to the result. Cumulative multiply accumulates the product with the value already in accumulator X + 1.

3.6.6 Division

044	DVD	X	N(M)	Divide
045	DVR	X	N(M)	Divide and round
046	DVS	X	N(M)	Divide single length

The division group will set V if an attempt is made to divide by zero (in which case the division is not attempted) or if the quotient exceeds the capacity of a word. Each divide order will place the quotient in X + 1, and leave the remainder in X. The remainder always has the same sign as the divisor. Rounded divide adds 2^{-24} to the quotient.

For the first two divide instructions, the dividend is a double length number in X and X + 1. Single length divide uses a single length dividend in X + 1. The divisor is always the contents of location N(M), and is left undisturbed.

3.6.7 Comparison

026	TXU	X	N(M)	Test x unequal to n
027	TXL	X	N(M)	Test x less than n + c

TXU will set C if the contents of accumulator X and the contents of location N(M) are not equal, or it will leave C set if it is already set.

TXL will set C if the contents of X is less than the contents of location N(M) plus the contents of C. It is designed specifically for use with words of alphanumeric characters and should only be used with numbers in binary representation, if all the numbers are positive.

The contents of both accumulator X and of location N(M) are left undisturbed by this group.

3.6.8 Character Conversion

043	CDB	X	N(M)	Decimal to binary
047	CBD	X	N(M)	Binary to decimal

CDB will set C if it encounters a non-numeric character, and will set V if it results in a number that exceeds the capacity of a double length word. Both instructions deal with 6-bit binary coded decimal characters. The CDB instruction will take a character specified by N(M), check that it is numeric, then (if it is numeric) multiply the double length number in X and X + 1 by 10 and add the character to this. If the character is non-numeric neither the multiplication nor the addition takes place but instead C is set.

CBD will take a binary fraction stored in accumulators X and X + 1 and multiply this by 10. The integral part of the product is stored as a 6-bit character in the location N(M) while the fractional part is replaced in the accumulators.

If the instructions are not indexed then the character position n_3 applies; if they are indexed the character is specified by the most significant two bits of the index register.

3.6.9 Floating Point Conversion

130	FLOAT	X	Fixed to floating
131	FIX	X	Floating to fixed

FLOAT will convert the fixed point number, in accumulators X and X + 1, into a floating point number in accumulators X and X + 1, Overflow cannot be set.

FIX will convert a floating point number, in accumulators X and X + 1, into a fixed point number in X and X + 1 so that X contains the integral part and X + 1 contains the fractional part. If the number cannot be contained, overflow is set and the result in X and X + 1 will be meaningless.

3.6.10 Floating Point Arithmetic

132	FAD	X	N(M)	Add to F-P Accumulator
133	FSB	X	N(M)	Subtract from F-P Acc.
134	FMPY	X	N(M)	Multiply
135	FDVD	X	N(M)	Divide into F-P Acc.
136	LFP		N(M)	Load F-P Accumulator
137	SFP		N(M)	Store F-P Accumulator

Each of the instructions 132 - 135 performs a floating point operation between a quantity stored in the floating-point accumulator and the contents of N(M) and N + 1(M). The contents of the store locations are left undisturbed.

The X-field in instructions 132 - 135 is used to control the way in which the floating point operation is carried out.

X = 1	The result is unrounded
X = 2	The result is not normalized
X = 4	The quantities in the acc. and the store are reversed before the operation is carried.

Thus, in general X will be specified as zero, although the above effects may be produced by the appropriate combination of X values.

3.6.11 Floating Point Normalize

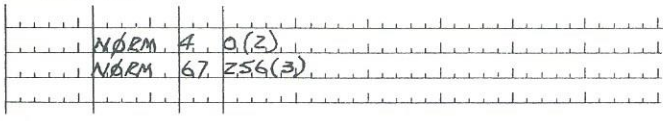
114	NORM	X	N(M)	Normalize
115	NORM	XX*	N(M)	Normalize (double)

This group will set V if during the normalize the exponent of a number becomes greater than 255. In order to preserve the greatest accuracy the argument of floating point numbers, which is stored as a fraction, must lie in one of the ranges $1 > \text{argument} \geq 1/2$ or $-1/2 > \text{argument} \geq -1$. The shifting of the argument into these ranges and the adjusting of the exponent is defined as normalizing. The first instruction statement will normalize a floating point number whose single length argument is given in accumulator X and whose exponent is the actual value of N(M). The resultant exponent is given in X + 1, which is initially cleared by the instruction.

The second instruction statement will normalize a floating point number whose argument is given in X and X + 1 and whose exponent is the actual value of N(M). The resultant floating point number has an argument of 38 + 1 bits (the sign bit of the second word is always zero) followed by an exponent of 9 bits.

In the normal use of this group, the original exponent is contained in the index register specified by (M), and N will be written as either 0 or 256, depending on the form the exponent takes. If it is in the same "e + 256" form, which is used in the exponent store of a standard floating point number, then N is written as 0. However, if the exponent in the index register is the sum or difference of two standard exponents, as happens in floating point multiplication or division, then N must be written as 256 to obtain the standard "e + 256" from in the normalized number.

Examples:



3.6.12 Logical Instructions

This group will not set V. Their grouping is related to the first three groups, 3.6.1, 3.6.2, and 3.6.3. The group is characterized by the fact that the numerical value of the words on which they operate are only of secondary significance and that these words are primarily thought of simply as strings of binary digits. Each operates on two words and produces one new word as a result. They are used for masking, collating, packing, unpacking, etc. of binary bit patterns.

020	ANDX	X	N(M)	AND to X
021	ORX	X	N(M)	OR to X (inclusive OR)
022	ERX	X	N(M)	Non-equivalence to X (exclusive OR)
030	ANDS	X	N(M)	AND to store
031	ORS	X	N(M)	OR to store (inclusive OR)
032	ERS	X	N(M)	Non-equivalence to store (exclusive OR) R)
120	ANDN	X	N(M)	AND with N
121	ORN	X	N(M)	OR with N (inclusive OR)
123	ERN	X	N(M)	Non-equivalence with N (exclusive OR)

ANDX to ERX leave their resultant word in accumulator X, and the word at location N(M) is left undisturbed. ANDS to ERS leave their resultant word at location N(M), and the contents of accumulator X is left undisturbed. ANDN to ERN are the same as ANDX and ERX except that the actual value of N(M) is used as data.

3.6.13 Part Word Manipulation

024	LDCH	X	N(M)	Load character into X
025	LDEX	X	N(M)	Load exponent into X
034	DCH	X	N(M)	Deposit character into store
035	DEX	X	N(M)	Deposit exponent into store
036	DSA	X	N(M)	Deposit short address
037	DLA	X	N(M)	Deposit long address

In addition to the logical operations of 3.6.12, which provide general facilities for the packing and unpacking of part words, the FP6000 function code provides separate functions to handle certain various standard sub-divisions of a word. LDCH and DCH handle 6-bit binary coded decimal characters. LDEX and DEX handle the least significant 9 bits of a word, ie. the floating point exponent. DSA handles the least significant 12 bits of a word, ie. a 12-bit address. DLA handles the least significant 15 bits of a word, ie. a 15-bit address or modifier.

LDCH, if not indexed, extracts the least significant 6-bit character of the word location N and places it in the least significant 6 bit positions of accumulator X, the remainder of X being cleared. When indexed, a character is placed in X in the same manner, but the character extracted from the location N(M) is specified by the most significant 2 bits of the index register (M).

3.6.13 continued

Chapter 5 will describe the convention used in AUTOCODER to define character index words. DCH is the converse of LDCH. Character indexing does not apply to the other instructions of this group.

LDEX extracts the exponent of the word in location N(M) and places it in the accumulator X, the remainder of X being cleared. DEX is the converse of LDEX except that the remainder of the word in store is not cleared.

DSA and DLA store into the word at location N(M) the least significant 12 bits or the least significant 15 bits, respectively, of the word in accumulator X.

Note that the instructions DCH to DLA will only replace part of a word; they leave the rest of the word undisturbed.

3.6.14 Shifting

The purpose of the instructions of this group is to take the bits of a binary word (or pair of binary words) and move them to the left or right. The shifts correspond to multiplication or division by powers of 2. The cyclic shift will rotate the bit pattern without losing any of the bits. The logical shifts will "drop off" bits from the end of the word towards which the shift is made, and fill the other end of the word with zero bits.

The arithmetic shifts are similar to the logical shifts, but account for the fact that they are dealing with signed numbers. They will set overflow if capacity is exceeded. The arithmetic right shifts propagate the sign bit each time the word is shifted as well as rounding to within $\pm 2^{-24}$ (in the fractional convention). The shift right arithmetic on overflow has application to the right shifting of a number whose production set overflow.

N_S is shown in the operand field to indicate that it is only the value of the least significant 10 bits of the N field of the instruction that are used in determining the amount of the shift (the value can be modified).

Note that the double length shifts must specify X and X + 1 in the accumulator field.

3.6.15 Branch on State of Accumulator

050	BZE	X	N	Branch if zero
052	BNZ	X	N	Branch if non-zero
054	BPZ	X	N	Branch if positive
056	BNG	X	N	Branch if negative

Each instruction of this group will carry out a test on contents of the accumulator X. If the test is satisfied then the instruction results in a branch to the program location N, otherwise the next instruction is obeyed in the usual way. Note that N represents a 15-bit address (see 3.5.6).

N_t				
110-0	SLC	X	N_S(M)	Shift left cyclic
-1	SLL	X	N_S(M)	Shift left logical
-2, 3	SLA	X	N_S(M)	Shift left arithmetic
111-0	SLC	XX*	N_S(M)	Shift left cyclic (double)
-1	SLL	XX*	N_S(M)	Shift left logical (double)
-2, 3	SLA	XX*	N_S(M)	Shift left arithmetic (double)
112-0	SRC	X	N_S(M)	Shift right cyclic
-1	SRL	X	N_S(M)	Shift right logical
-2	SRA	X	N_S(M)	Shift right arithmetic
-3	SRAV	X	N_S(M)	Shift right arithmetic on overflow
113-0	SRC	XX*	N_S(M)	Shift right cyclic (double)
-1	SRL	XX*	N_S(M)	Shift right logical (double)
-2	SRA	XX*	N_S(M)	Shift right arithmetic (double)
-3	SRAV	XX*	N_S(M)	Shift right arithmetic on overflow (double)

3.6.16 Unconditional Branch

	X		
074-0	BRN	N	Branch to N

This instruction always results in a branch to program location N. The mnemonic operation BRN also defines the X portion of the instruction, since the 074 function must be qualified by X.

3.6.17 Branch on State of Overflow

Each of these will test the state of the overflow register V, and if the condition tested is satisfied they result in a branch to the program location N, otherwise the next instruction is obeyed in the usual way.

Two of them, BVSR and BVCR, always clear V irrespective of whether or not a branch occurs. A third, BVCI, always inverts the state of V, ie. if V was set then it clears V, and if V was clear then it sets V.

As in 3.6.16 each mnemonic defines a qualifying value of X.

3.6.18 Branch on State of Carry

	X		
074-5	BCS	N	Branch if C set
074-6	BCC	N	Branch if C clear

Each of these tests the state of the carry register C, a branch to program location N results if the condition is satisfied. The performance of either clears C.

As in 3.6.16 each mnemonic defines a qualifying value of X.

3.6.19 Indexing and Counting

060	BUX	X	N	Branch on unit indexing
062	BDX	X	N	Branch on double indexing
064	BCHX	X	N	Branch on character indexing

Each of these operates on the word in accumulator X.

BUX operates on a counter and a modifier, the counter being the most significant 9 bits of the word in X and the modifier being the least significant 15 bits of the word in X. The action of the instruction is to increase the modifier by 1, decrease the counter by 1 and, if after this, the counter is non-zero, branch to program location N.

BDX is similar to BUX with the sole exception that the modifier is increased by 2. It is useful when dealing with double length data in arrays.

BCHX provides character indexing. The most significant 2 bits of the word in X are used as a character index, and the counter is reduced to 7 bits. The action of the instruction is to increase the character index by 1, add any carry from the character index to the modifier, decrease the counter by 1 and, if after this the counter is non-zero, branch to program location N. The effect of the instruction is to increment by 1/4 of a word (1/4 of a word is a 6-bit binary coded decimal character).

The maximum value of a counter for the BUX or BDX instructions, is 512, which is obtained by having the counter in accumulator X equal to zero. Similarly the maximum value of a counter for the BCHX instruction is 128.

3.6.20 Subroutine Linkage

070	CALL	X	N	Call subroutine at N
072	EXIT	X	N	Exit to N + x

This group is designed to provide linkage to and from subroutines.

	X		
074-1	BVS	N	Branch if V set
074-2	BVSR	N	Branch if V set and reset
074-3	BVC	N	Branch if V clear
074-4	BVCR	N	Branch if V clear and reset
074-7	BVCI	N	Branch if V clear and invert

3.6.20 continued

CALL provides the branch from the main program to a subroutine or from one subroutine to another at lower level. The effect of the instruction is to store in the most significant bit of accumulator X, the state of V and, in the least significant 15 bits, the location of the next instruction.

EXIT provides the connection back to the CALLing main program or higher level subroutine. The effect of the instruction is to branch to a point N locations beyond the address contained in the least significant 15 bits of accumulator X. If the most significant bit of X is 1, or if V is set when the instruction is obeyed, V is left set by this instruction.

3.6.21 Block Transfer and Block Check Sum

126	MOVE	X	N(M)	Move N words
127	SUM	X	N(M)	Sum N words

These instructions provide simple and automatic ways of, respectively, moving a block of N consecutive words, or forming a check sum of the values of a block of N consecutive words.

MOVE will copy the word whose storage location is specified in accumulator X into the location specified in accumulator X + 1. Unity is then added to the addresses of both these storage locations and the procedure repeated until a total of N(M) words has been transferred.

SUM will replace the contents of accumulator X with the sum of N(M) consecutive words starting with the word whose location is specified in accumulator X + 1. Bits will be lost at the most significant end of the sum when this exceeds single word capacity but in no circumstances will overflow be set by the instruction.

For both MOVE and SUM only the least significant 9 bits of the value of N(M) are used to give the number of words in the block. Hence the maximum number of words that can be operated on by each is 512, and this is obtained with the value of N(M) equal either to zero or a multiple of 512.

3.6.22 Miscellaneous

023	OBEY		N(M)	Obey the instruction in N
033	STOZ		N(M)	Store zero
123	NULL			Null operation
124	LDCT	X	N(M)	Load into counter
125	MODE		N(M)	Set mode to N

Of this group both OBEY and NULL do not themselves alter C.

OBEY causes the contents of the location specified by N(M) to be obeyed as an instruction, as though it were in the location occupied by the OBEY instruction. Since the OBEY instruction does not itself alter C, the instruction obeyed in N(M) may include carry which was set by the instruction preceding the OBEY.

STOZ will write a zero word into the location specified by N(M).

NULL results in no action in the object program. On the other hand, if the accumulator field of the instruction statement is filled with a 7, then the NULL instruction still has no effect in the object program but may transfer control to the DEBUG program. Refer to Chapter 8 of the FP6000 Programming Manual.

LDCT relates to the Indexing Instructions (see 3.6.19). The effect of the instruction is to set up in accumulator X a 9-bit counter equal to the value of N(M) and to clear the modifier (or index) part. If the value of N(M) exceeds 511, it will be reduced modulo 512.

MODE sets a 7-bit "transfer mode number" equal to the least significant 7 bits of the value of N(M). The transfer mode number, stored in part of the instruction number register, is used in connection with peripheral transfer instructions, and with suppression of non-significant zeros by the CBD instruction (see 3.6.8).

For this group, the programmer is not prevented from using the accumulator field of an instruction statement even though its use is not indicated in the group listing above. An octal number in the accumulator field will be assembled as part of the instruction as usual, but, except for the case noted in NULL above, will in no way alter the operation of the instruction. The same comment applies to the operand field of the NULL statement.

3.6.23 Peripheral Control

150	SUSBY	X	N(M)	Suspend if busy
151	REL	X	N(M)	Release
152	DIS	X	N(M)	Disengage
153	SPW	X	N(M)	Store peripheral control word
154	CONT	X	N(M)	Continue by reading more program
155	SUSDP	X	N(M)	Suspend and dump

3.6.23 continued

Each instruction may refer to a valid peripheral unit. As explained in 2.3, the programmer may use for N(M) a code indicating the type of unit and for X the unit number of this type; or he may use for N(M) the symbolic name assigned to the desired unit under a #PERIPHERAL directive (in which case the X field is ignored).

SUSBY will suspend the object program if the unit specified is busy.

REL will result in a message to the operator that the unit specified has been released. The program is prevented from making further use of this peripheral. The instruction is used at a point in a program where all required use has been made of a peripheral and it may now be released for use by some other program, yet to be loaded.

DIS will result in a message to the operator that the unit specified has been disengaged. The program is prevented from making further use of this peripheral until the unit is "re-engaged" by the operator. The instruction is used at a point in a program where some peripheral needs operator attention, eg. an on-line card punch requires cards in the feed hopper.

SPW will store in the program absolute location 9, the control register of the unit specified.

CONT will read more program from the unit specified. Obviously, only an input-type peripheral is valid for this instruction. Thus, for example, a paper tape punch is not valid.

SUSDP will suspend the program and dump the program as it then stands onto the unit specified.

3.6.24 Peripheral Use

170	PTR	X	N(M)	Paper tape reader
171	PTP	X	N(M)	Paper tape punch
172	LNP	X	N(M)	Line printer
173	CDR	X	N(M)	Card reader
174	CDP	X	N(M)	Card punch
175	MTU	X	N(M)	Magnetic tape unit
176	DRM	X	N(M)	Drum
177	TYP	X	N(M)	Typewriter

In this group the mnemonic of each instruction specifies the type of unit to be used. In the X field will be written the unit number, while N(M) will indicate the location of the control word to be used with the instruction. The control word will control the actual action of the instruction.

3.6.25 Interrupt and Abolish

156	ABOL		Abolish
157	SUSWT		Suspend and wait
160	SUSTY	N(M)	Suspend and type

This group is used to give end of job or error halts.

ABOL will result in a message to the operator requesting that the program be abolished. It is used as the final instruction to be obeyed in a program.

SUSWT will suspend the program awaiting an operator message to EXECUTIVE.

SUSTY will suspend the program awaiting an operator message to EXECUTIVE and will also print on the console typewriter any message from a control word specified by N(M).

3.6.26 Autoroutine Control

162	SUSMA	X		Suspend me - master
163	AUTO	X	N(M)	Activate autoroutine
164	SUSAR			Suspend me - autoroutine

This group is designed to control the use of autoroutines.

SUSMA will suspend the master program, if the autoroutine X is active. Since only two autoroutines may be used in a given program, X may only be written as a 1 or a 2.

AUTO results in the autoroutine X being activated with the autoroutine being entered at its program location specified by N(M). Again X may only be a 1 or a 2.

SUSAR will suspend the autoroutine in which it occurs awaiting activation by the master program, and, if necessary will release suspension on the master due to this autoroutine, ie. if the master has been suspended by a SUSMA instruction for this autoroutine.

CHAPTER 4

MACRO-INSTRUCTIONS

TABLE OF CONTENTS

CHAPTER 4

MACRO-INSTRUCTIONS

4.1 INTRODUCTION

In the compilation of source programs into object programs, most AUTOCODER instructions result in a single machine instruction. However, some, called macro-instructions (often abbreviated to "macros"), result in the generation of several machine instructions. Such instructions usually perform some frequently required task which cannot be done by any single machine instruction and eliminates the necessity for writing out the corresponding set of "one-for-one" instructions. However, since for each new appearance of a macro-instruction in the source program the corresponding machine instructions will be generated once, a subroutine is usually better if the task requires more than (say) 4 or 5 instructions.

Macro-instructions appearing in an AUTOCODER source program are of two different categories or types. First, there is a limited number of macro-instructions which are a permanent part of the AUTOCODER language and are recognized by the compiler as such. Second, a varying number of macro-instructions may be created by the programmer and defined for each new compilation. This chapter describes all the macro-instructions of the first category, grouped according to function, and will indicate how those of the second type are specified.

Macro-instructions of both types have a format similar to other AUTOCODER instructions, except for the following differences:-

(a) Each macro-instruction usually occupies several memory locations, and any symbol written in the LABEL field will be assigned the value of the first of these.

(b) An alphabetic code in the OPERATION field indicates the specific macro-instruction. A numeric code cannot be substituted for the alphabetic code, since it does not correspond to any one machine instruction.

(c) The OPERAND field may contain more than one operand, in which case, each one except the last, is followed by a comma.

During compilation each macro-instruction is replaced by the basic AUTOCODER instructions and these in turn are converted to machine instructions. Section 4.7 shows the correspondence between each macro-instruction and the basic instructions.

4.2 DOUBLE-LENGTH MACRO-INSTRUCTIONS

To facilitate working with double-length numbers, a set of macros which perform simple arithmetic operations on double-length numbers, are provided. Note that these macros have operation codes identical to their single-length counterparts, and differ from them only in the accumulator field where there are 2 digits, specifying adjacent accumulators (X and X*). The double-length numbers to be operated on, must be in standard form.

4.2.1 Double-length to the Accumulators

	LDX	XX*	N(M)
	ADX	XX*	N(M)
	NGX	XX*	N(M)
	SBX	XX*	N(M)

Description:

Each of the above macro-instructions involves two double-length registers, one made up of the accumulators X and X*, and the other made up of the memory locations N(M), and N + 1(M). In each case, the result of the operation will be stored in X and X* in standard double-length form. Overflow may be set.

Operand:

(1) Literals may not appear in the operand field.

(2) If the operand is a symbol not previously defined, two locations will be allocated in lower memory.

4.2.2 Double-length to Storage

```
STØ   XX*  N(M)
ADS   XX*  N(M)
NGS   XX*  N(M)
SBS   XX*  N(M)
```

Description:

These 4 instructions are identical to the first 4, except that the double length result will be stored in locations N(M) and N + 1(M).

Operand:

(1) Literals may not appear in the operand field.

(2) If the operand is a symbol not previously defined, two locations will be allocated in lower memory.

4.2 BRANCH-WITH-TEST MACRO-INSTRUCTIONS

Branch if X Unequal

```
BXU   X    N1(M),1N2
BXU   XX*  N1(M),1N2
```

This instruction compares the two single-length numbers in accumulator X and location $N_1(M)$, or the two double-length numbers in accumulators X and X* and locations $N_1(M)$ and $N_1+1(M)$, depending on whether there are one or two digits in the X-field.

(1) It branches to location N_2

(i) if the two numbers are not equal
(ii) or if the carry register (C) was previously set.

(2) Otherwise, it continues to the next sequential instruction.

(3) In either case, C is cleared.

Operands:

(1) $N_1(M)$ can be a literal only when representing a single-length number.

(2) If $N_1(M)$ contains a symbol not previously defined, one location is allocated in lower memory if it represents a single-length number, two locations are allocated if it represents a double-length number.

Branch if X Equal

```
BXE   X    N1(M),1N2
BXE   XX*  N1(M),1N2
```

This instruction is identical to BXU in all respects except that the branch to N_2 occurs if the two numbers are equal and if C was not previously set.

Branch if X Less

```
BXL   X    N1(M),1N2
BXL   XX*  N1(M),1N2
```

Description:

This instruction compares the two single-length or double-length numbers indicated to determine their relative size. The numbers involved must have the same algebraic sign, and the carry indicator (C) must be clear initially.

(1) It branches to location N_2 if the number in X (or X and X*) is algebraically less than the number in $N_1(M)$ (or $N_1(M)$ and $N_1+1(M)$).

(2) Otherwise, it continues to the next sequential instruction.

(3) In either case, C is cleared.

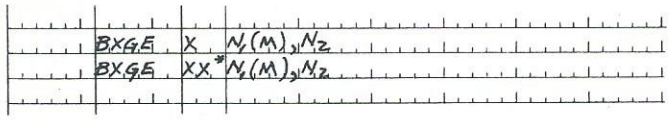
Operands:

(1) $N_1(M)$ can be a literal only when representing a single length number.

(2) If $N_1(M)$ contains a symbol not previously defined, one location is allocated in lower memory if it represents a single-length number, two locations are allocated if it represents a double length number.

4.2 continued

Branch if X Greater or Equal

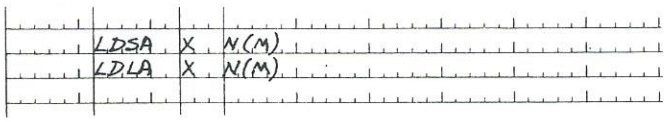


This instruction is identical to BXL in all respects except that the branch to N_2 occurs if the number in the accumulator(s) is algebraically greater than or equal to the number in the storage location(s).

4.3 PART-WORD MACRO-INSTRUCTIONS

These instructions add to the part-word operations provided by single machine instructions.

Load Short Address



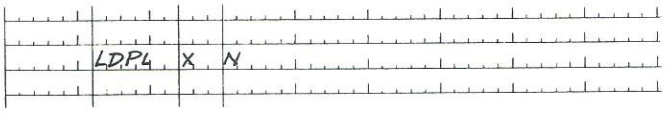
Description:

These instructions load the least significant 12 bits or the least significant 15 bits of location N(M) into the least significant 12 positions or 15 positions of accumulator X. The remainder of X is cleared.

Operand: A literal can be substituted for N(M).

4.4 SPECIAL LOAD INSTRUCTIONS

Load Program Location:



Description:

This instruction loads accumulator X with a word containing N.

Operand:

(1) As with the small-integer instructions (machine group 10) the operand is taken in a literal sense. It can have any of the forms of a literal, but with the quotation marks omitted.

(2) Any symbol appearing in the operand, which has not been previously defined, is taken as referring to a program location just as in a branch instruction. The main use of this instruction is in loading an index register to refer, indirectly, to a program location, in other than a branch instruction. It is the only non-branch instruction which can have a program location in the operand (see Section 2.2.1).

4.5 MAGNETIC TAPE CONTROL

Magnetic tape control operations may be specified by the following macro-instructions.

WEF	X	- Write End-of-File
REW	X	- Rewind
BSP	X	- Backspace
BEF	X	- Back to End-of-File
FEF	X	- Forward to End-of-File
CLOSE	X	- Close the File and release unit
SCR	X	- Open a scratch tape and retain as scratch

In the above instruction X specifies the unit number. The necessary mode constant is generated by AUTOCODER. The following instructions are used for labelling purposes.

OPEN	X	N(M)	- Open a file whose label matches the information stored in location N(M).
LABEL	X	N(M)	- Locate a scratch tape and label it according to the information stored in location N(M).

In both cases X specifies the unit number and the mode is set by the necessary instructions.

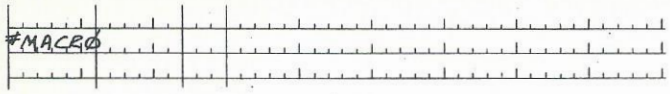
4.6 PROGRAMMER-DEFINED MACRO-INSTRUCTIONS

As mentioned earlier, the programmer may add, to the macro-instructions already described, others of his own specification. A definition for each such macro-instruction must appear in the source program, making it a part of the AUTOCODER language for use in that program. The form of the definition and the directive involved, are described below.

Directive V: MACRO

(1) Each macro-instruction definition must be preceded by this directive, written on a separate line, with no operand.

4.6 continued



(2) The definition of the macro-instruction consists of two main parts, the macro format and the basic format. The macro format indicates the make-up of the macro-instruction as it will be used in the program. It is written on a line immediately following the directive, using the fields, OPERATION, ACCUMULATOR and OPERAND.

OPERATION:

Operation consists of 1 to 5 alphabetic characters, to be used as the operation code. This code must not be used for any other AUTOCODER instructions or macro-instructions unless the length of the X-field differs (An operation code followed by a 1-digit X-field is interpreted as a different instruction than the same code followed by a 2-digit X-field).

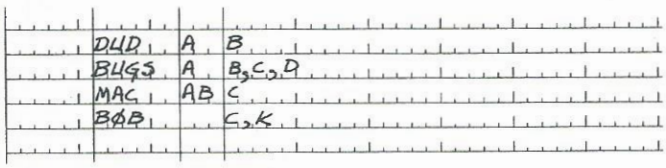
ACCUMULATOR:

The X-field can be blank or contain 1 or 2 alphabetic characters, left-justified. The number of characters indicates the number of digits required in this field.

OPERAND:

The Operand field can be blank or contain a variable number of alphabetic characters. Each character represents an operand of the macro-instruction and, except for the last one, is followed by a comma.

Examples:



Each of the alphabetic characters appearing in the X and Operand fields must be different, and since they are being used to represent a parameter of the macro-instruction they have no significance outside the definition. Symbols identical to these alphabetic parameters may be used elsewhere in the source program without confusion.

(3) The basic format indicates what the effect of the macro-instruction will be, in terms of basic AUTOCODER instructions. In fact, during compilation, the macro-instruction will be replaced by these basic instructions at each point it appears in the source program. These instructions will then be compiled into object language.

The basic format is written following the macro format; the basic instructions are written on separate lines, in the desired order, and using the 3 fields, OPERATION, ACC. and OPERAND.

OPERATION:

This can be any alphabetic operation code in the AUTOCODER language, except that of another macro-instruction. It can also be a 3-digit octal machine code.

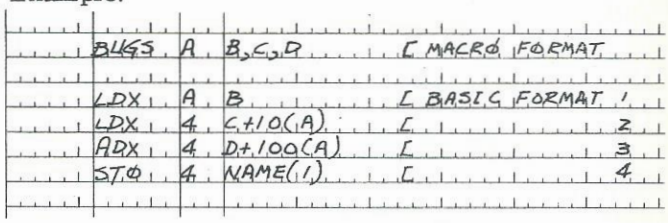
ACC or X-field:

This will be blank or have 1 or 2 characters, depending on what is allowed following the particular operation code. If it is not blank, the characters may include any of the alphabetic parameters mentioned in the macro format.

OPERAND:

This field may contain any of the forms of operand normally allowed with the operation code written, including literals and the symbol *. The alphabetic parameters used in the macro format may appear anywhere that symbolic operands are allowed. In addition, these parameters may be used as modifiers, enclosed in brackets.

Example:



The appearances of the symbolic parameters in the basic format indicate when the actual parameters are to be inserted during compilation. Other symbols in the basic format (such as NAME, in the example above) remain constant for each use of the macro-instruction and are eventually replaced by the numeric values assigned according to the rules in Chapter 2. Similarly any numeric X-fields or numeric portions of Operands retain their values for each use of the macro-instruction.

4.6.1 Use of Defined Macro-Instructions

Having defined a macro-instruction using the directive MACRO as explained above, the programmer may use it anywhere in this source program just as any other AUTOCODER instruction. With each use he must write the operation code indicated in the macro format, but he replaces the alphabetic parameters in the X and operand fields with actual parameters as follows:

4.6.1 continued

(1) Alphabetic parameters appearing in X-fields of the macro or basic format, or as modifiers in the operands of basic format, can be replaced by octal digits only (modifiers are restricted to 1,2 or 3).

(2) All other alphabetic parameters can be replaced by operands of any form allowed in the instructions (of the basic format) in which they appear. Where applicable, this includes:-

(a) Operands with modifiers, unless they appear in basic instructions already modified.

(b) Symbols adjusted by integers even if the parameters are again adjusted in the basic instructions.

(c) Literals, provided the alphabetic parameters appear unadjusted, unmodified, etc., in the basic operands.

(d) The symbol * (with or without adjustment), where * is assigned the value of the location of the first basic instruction each time.

Example:

An example of the use of the macro BUGS defined in a previous example, might be:-

```
        BUGS  2  ADDR(3),START+4,25
```

This would then be equivalent to the following basic instructions.

```
        LDX   2  ADDR(3)
        LDX   4  START+14(2)
        ADX   4  125(2)
        STØ   4  NAME(1)
```

4.7 LIST OF PERMANENT MACRO-INSTRUCTIONS

Following is the list of AUTOCODER permanent macro-instructions and the equivalent basic instructions produced.

MACRO-INSTRUCTION			EQUIVALENT		
LDX	XX *	N(M)	LDXC LDX	X * X	N + 1 (M) N(M)
ADX	XX *	N(M)	ADXC ADX	X * X	N + 1 (M) N(M)
NGX	XX *	N(M)	NGXC NGX	X * X	N + 1 (M) N(M)
SBX	XX *	N(M)	SBXC SBX	X * X	N + 1 (M) N(M)
STO	XX *	N(M)	STOC STO	X * X	N + 1 (M) N(M)
ADS	XX *	N(M)	ADSC ADS	X * X	N + 1 (M) N(M)
NGS	XX *	N(M)	NGSC NGS	X * X	N + 1 (M) N(M)
SBS	XX *	N(M)	SBSC SBS	X * X	N + 1 (M) N(M)

4.7 continued

MACRO-INSTRUCTION			EQUIVALENT			
BXU	X	N_1 (M), N_2	TXU BCS	X	N_1 (M) N_2	
BXU	XX *	N_1 (M), N_2	TXU TXU BCS	X * X	N_1 + 1 (M) N_1 (M) N_2	
BXE	X	N_1 (M), N_2	TXU BCC	X	N_1 (M) N_2	
BXE	XX *	N_1 (M), N_2	TXU TXU BCC	X * X	N_1 + 1 (M) N_1 (M) N_2	
BXL	X	N_1 (M), N_2	TXL BCS	X	N_1 (M) N_2	
BXL	XX *	N_1 (M), N_2	TXL TXL BCS	X * X	N_1 + 1 (M) N_1 (M) N_2	
BXGE	X	N_1 (M), N_2	TXL BCC	X	N_1 (M) N_2	
BXGE	XX *	N_1 (M), N_2	TXL TXL BCC	X * X	N_1 + 1 (M) N_1 (M) N_2	
LDSA	X	N (M)	LDX ANDN	X X	N (M) #7777	
LDLA	X	N (M)	LDX ANDX	X X	N (M) '#77777'	
LDPL	X	N	LDX	X	'N'	(Note 1)
WEF	X		MTU	X	'5'	(Note 2)
REW	X		MTU	X	'7'	(Note 2)
BSP	X		MTU	X	'2'	(Note 2)
BEF	X		MTU	X	'6'	(Note 2)
FEF	X		MTU	X	'4'	(Note 2)
CLOSE	X		MTU	X	'519'	(Note 2)
SCR	X		MTU	X	'256'	(Note 2)

4.7 continued

MACRO-INSTRUCTION			EQUIVALENT			
OPEN	X	N(M)	LDN	X	128	
			STO	X	N(M)	(Note 3)
			MTU	X	N(M)	
LABEL	X	N(M)	LDN	X	384	
			STO	X	N(M)	(Note 3)
			MTU	X	N(M)	

Note 1. An undefined symbol, N(M), in the operand field will be treated as a branch point.

Note 2. AUTOCODER will create a second literal which is zero, following the literal indicated.

Note 3. An undefined symbol in the operand field will be treated as an error.

CHAPTER 5

DATA GENERATION

CHAPTER 5

TABLE OF CONTENTS

CHAPTER 5

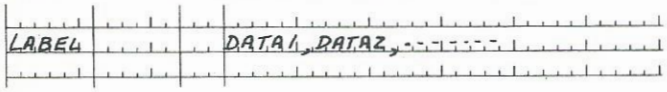
DATA GENERATION

5.1 DATA GENERATION

It is often desirable to assemble constant data as part of a program. This chapter will deal with the two methods of generating data available in AUTOCODER. The first part of the chapter will treat the method of using data generating statements, and the second part will treat the literal operand method.

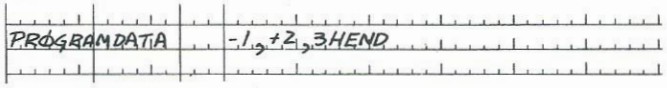
5.2 DATA STATEMENTS

Data statements result in the generation of preset constants by AUTOCODER. A constant may be numeric or binary coded decimal (BCD). Various types of numeric information are provided for, such as decimal, octal, or floating point. A data statement is divided into two fields, as follows:



The label field need not be present.

Example:



Data Statements may be used to enter data into work areas or intermingled with the program instructions, such as may be required for subroutine calling sequences. This is not to imply that the assembled program can differentiate between instructions and constants, it is still the programmer's responsibility to ensure that no attempt is made to obey stored words which are not instructions. When entering data into work areas the data statement should be associated with (ie. be one of the lines following) an appropriate directive (LOWER, UPPER).

Each data statement starts locating data items from the last location used by the previous line. Thus, if a data statement follows a line of storage allocating symbols coming under a LOWER or UPPER directive (see 2.2.2, 2.2.3), the first data word is located next to the last location allocated by the preceding line.

5.3 LABEL FIELD

5.3.1 Form

A LABEL can be a symbol of 1 to 11 alphanumeric characters, the first character always being alphabetic.

5.3.2 Use

When used, the LABEL is given a value equal to the location of the first data word generated by the data statement. Thus, if any particular data item is to be labelled then it must be the first data item in the operand field of the labelled statement.

5.4 DATA FORMS

Data items are written in the order they are to take in storage. They are entered in the operand field of the coding form, with successive items on the same line being separated by commas. A comma is implied between successive lines.

A data item may be any of the following:

Octal integer

Decimal integer

Decimal fraction

Mixed decimal number

Floating point decimal number

6-bit binary coded decimal characters

Index word

Character index word

These will be detailed in the sections following. For all numeric items non-significant zeros will be ignored.

5.4.1 Octal Integers

Signed or unsigned octal integers in the range #0 $\leqslant$ integer $\leqslant$ #77777777 (eight sevens), are allowed. The octal integer is preceded by the character "#".

Examples:

```
#77,#76,#7777
```

5.4.2 Decimal Integers

Signed or unsigned decimal integers in the range $-2^{46} \leqslant$ integer $\leqslant 2^{46}-1$ are allowed. Single precision integers, in the range $-2^{23} \leqslant$ integer $\leqslant 2^{23}-1$, are merely written out as data items, or can be followed by (1). Double precision integers must be followed by " (2) ". A data item which is blank will be treated as zero (ie. ",," will generate a zero).

Examples:

```
+2,-3,04,,,1234567890(2),+002
```

5.4.3 Decimal Fractions

Signed or unsigned decimal fractions in the range $-1.0 \leqslant$ fraction $\leqslant 1.0 - 2^{-46}$ are allowed. Fractions must contain a decimal point. Single precision fractions, in the range $-1.0 \leqslant$ fraction $\leqslant 1.0 - 2^{-23}$, are merely written out as data items having a maximum of 13 decimal digits following the decimal point, but AUTOCODER rounds off the value to just less than 7 digits so that the result is single length. Double precision fractions must be followed by " (2) " and may have up to 20 decimal digits following the decimal point, but AUTOCODER rounds off the value to just less than 14 digits so that the result is double length.

Examples:

```
-0.01,+.02,.03
-0.00000001234567 8(2)
```

5.4.4 Mixed Decimal Numbers

Mixed numbers are signed or unsigned numbers having both an integral and fractional part. Each of these numbers must be followed by an expression " (n.b) ", where n is the number of words to be occupied by the number, and b is the number of binary bits to be used for the fractional part.

For simplicity, an example of a single precision integer is given:

```
2(1.9)
```

And as further example, a single precision fraction:

```
0.125(1.23)
```

Other Examples:

```
198.75(1.8),283.6(2.23)
```

If either the integer or fraction is zero, then it may be omitted.

```
203(1.6),+.167(2.23)
```

5.4.5 Floating Point Decimal Numbers

These numbers are split into two parts; a fraction, which is the mantissa, and a signed or unsigned 1 or 2 digit integer, which is the decimal. The two parts are separated by the character E. The mantissa may be signed or unsigned and may have up to 13 decimal digits following the decimal point, but AUTOCODER rounds off the value to less than 12 digits. The exponent must be in the range $-77 \leqslant$ exponent $\leqslant 77$.

Examples:

```
.01356E-03,.7635E1
+.2095E+05
```

5.4.6 BCD Characters

BCD or Hollerith characters to be generated as data are preceded by "nH", where n is the number of characters and is limited by the number of characters that can be written in the operand field. The characters to be generated must follow the "H" close packed and any valid character, including blanks, commas, quotes, etc. is allowed. This is the one case where blanks are not ignored in the operand field.

```
1HH,1HX,8H*,.-'/()
```

Each BCD item will occupy an integral number of words with blanks being filled in on the right. ie., For the above example, the generated data is:

HbbbXbbb*,.-'/()
(small b represents a blank)

5.4.7 Index Words

Index items will generate words that are split into a 9-bit counter and a 15-bit modifier. The counter and the modifier parts of the item are separated by a "/" which must be present.

The counter may be:

(1) an unsigned decimal integer

(2) an unsigned octal integer, preceded by a "#"

(3) a symbol defined by a DEFINE or SET directive.

The value of the counter must be less than 512 decimal. For any value greater than 512 only the least significant 9 bits will be used.

The modifier part may be

(1) a signed or unsigned decimal integer

(2) a signed or unsigned octal integer

(3) any symbol, except the label of a program location (see 2.2.1)

(4) a symbol as in (3) followed by a signed decimal or octal integer.

The value of the modifier portion must be less than 32767 decimal. For any value greater than this the least significant 15 bits will be used.

Examples:

```
          6/NAME,1/DEDUCTION-17
          #77/TIME+#77
```

5.4.8 Character Index Words

Character index items will generate words that have a 7 bit counter, while the modifier includes 2 extra bits to specify a character position. Thus, they are the same as index words with the following two exceptions:

(1) The value of the counter must be less than 128.

(2) The modifier part terminates with a ".n", where n is one of the digits 0, 1, 2, or 3.

Examples:

```
          10/NAME.2,100/A-12.3
```

5.5 LITERAL OPERAND

The literal operand, used with instruction statements, is the second method of data generation and is recognized by a beginning and ending quote mark. An example may serve to illustrate the use of the literal. A data statement and an instruction statement, each properly placed within the program could be the following.

```
DATA                +9999
          LDX   4   DATA
```

but the following instruction statement, which has a literal operand, is equivalent to the above two statements:

```
          LDX   4   '+9999'
```

In general, the literal can be any of the forms of the data items of a data statement, except that all literals result in only a single word of data, and that mixed decimal numbers and floating point numbers are not permitted.

When AUTOCODER encounters an instruction with a literal operand, it will first "evaluate" the data within the quote marks to form a binary number. If this binary number is the same as that produced by some prior literal then the instruction address portion is given a value equal to the location already assigned to the binary number. If the binary number is unique then it is assigned a location in lower memory, and the instruction address portion is given a value equal to this assigned location.

5.5.1 Octal Integers

The same as for data statements (see 5.4.1)

```
          ANDX  1   '#777777'
```

5.5.2 Decimal Integers

The same as for data statements, limited to single precision (see 5.4.2).

```
          LDX   2   '+9467'
          LDXC  1   '-49863'
```

5.5.3 Decimal Fractions

The same as for data statements, limited to single precision (see 5.4.3).

```
          MPY   5   '+00.002'
          DVD   6   '-.0006'
```

5.5.4 BCD Characters

The same as for data statements, but limited to a maximum of four characters, ie. $1 \leqslant n \leqslant 4$ (see 5.4.6).

```
LDX  4  '2HXY'
LDX  5  '4H,'.X'
```

5.5.5 Index Words

The same as for data statements except that if the counter is not present the "/" may be omitted (see 5.4.7).

```
LDX  1  '10/NAME'
LDX  2  'NAME'
LDX  3  '#777/NAME+10'
```

5.5.6 Character Index Words

The same as for data statements but the "/" may be omitted if the counter is not present (see 5.4.8).

```
LDX  1  '10/NAME·2'
LDX  2  '100/A-12·3'
LDX  3  '#777/#1065·0'
LDX  4  'TABLE·1'
```

5.5.7 Restrictions in the Use of Literals

The context of the literal operand restricts their use. Thus, literals are not used

(1) with branch instructions

(2) with an instruction that leaves its result in the location specified by the operand, eg., STO, ADS, ANDS, etc.

(3) with floating point instructions

(4) with any double length operation, including macros.

CHAPTER 6

INPUT/OUTPUT OPERATIONS

CHAPTER 6

TABLE OF CONTENTS

CHAPTER 6

INPUT/OUTPUT OPERATIONS

6.1 PERIPHERAL DIRECTIVE

The #PERIPHERAL directive is used to specify to AUTOCODER the number and type of all peripheral units, except magnetic tapes, which are required by a program. This information appears in the resulting object program in the form of a requisition for the required units. Magnetic tapes are handled in a different manner as explained in Section 6.7.3.

The directive is written on a separate line with a blank operand field. The peripherals required are specified in the operand field of succeeding lines, using a standard 3-character code which consists of 2 letters specifying the type followed by a single digit unit number.

The 2-letter combinations assigned to peripherals are as follows:-

TR - Paper Tape Reader

TP - Paper Tape Punch

CR - Card Reader

CP - Card Punch

LP - Line Printer

TY - Typewriter

Example:

```
#PERIPHERAL
          CR0,CR1,,TR0,TP0,TP1,,TP2,LP0
```

The programmer may specify each unit to be used or the highest unit number of each type. The example below will cause requisition of the same number of units as the example shown above.

Example:

```
#PERIPHERAL
          CR1,,TR0,TP2,LP0
```

PERIPHERAL directives may appear any number of times in a source program.

6.2 PAPER TAPE READER

170 PTR X N(M)

This instruction specifies an input operation from unit X. The operation is controlled by the control word located in N(M). This word contains a character index in the most significant 2 bits, a character counter in the next 7 bits and a starting address in the least significant 15 bits.

The address field of the instruction may be a symbolic location or a literal.

The reading operation is performed according to a mode which may be set at any time prior to the read operation by the following instruction.

125 MODE N(M)

The mode specified by N(M) consists of the seven least significant bits of the address. Each bit has the following effect on the peripheral transfer.

Bit	State	Effect
0		Not used.
1	0	Read N computer characters
	1	Read N computer character or to Newline.
2	0	Read starting in ∝-shift.
	1	Read starting in previously established shift.
3	0	Read BCD mode, inserting control characters as required.
	1	Read binary mode.
4	0	Ignore erases and blank tape. TC4 switches reader off-line. TC3 causes skip to next TC1.
	1	All codes entered.
5	0	No effect.
	1	Skip to TC1 then start to read
6		Not used.

6.3 PAPER TAPE PUNCH

171 PTP X N(M)

This instruction specifies a paper tape punching operation on unit X and under control of N(M) which has the same form as the one described above (see 6.2).

Paper tape output may be in any mode established previously by:

125 MODE N(M)

The mode specified consists of the least significant seven bits of the N(M) address which have the following effect.

Bit	State	Effect
0,1		Not used.
2	0	Start punch in ∝-shift.
	1	Start punch in previously established shift.
3	0	Punch N computer characters.
	1	Punch N computer characters in binary mode.
4	0	Punch data.
	1	Punch N run-out characters.
5,6		Not used.

6.4 LINE PRINTER

172 LNP X N(M)

This instruction specifies an output operation on line printer unit X under control of word N(M) which has the same form as the one described above (see 6.2). The setting of the mode has no effect on this operation.

6.5 CARD READER

173 CDR X N(M)

This instruction causes reading of a punched card on unit (X) under control of N(M) which contains the starting internal memory location and number of characters as described above (see 6.2).

Card reading may be performed in one of two modes set previously by the MODE instruction.

These modes are as follows:-

MODE 0

The card is interpreted in BCD mode whereby each column is translated and stored as a character.

MODE 2

The card is interpreted in binary mode whereby each column is stored as is in successive word halves.

6.6 CARD PUNCH

174 CDP X N(M)

This instruction specifies punching of a card on unit X under control of N(M), the word which contains the character count and starting address. The setting of the mode has no effect on this operation. Punching is performed always in BCD mode.

6.7 MAGNETIC TAPE UNIT

175 MTU X N(M)

This instruction specifies a magnetic tape operation on unit X under control of information stored starting at location N(M). The mode which specifies the operation is stored in location N(M) while the next word, N + 1 (M), is reserved in order that EXECUTIVE may store in it the result of the transfer. While the operation is in progress this reply word is negative. The length of the control area is two words for control operations, four words for data transfers and nine words for labelling operations.

6.7.1 Magnetic Tape Control Operations

Following is a table of the various Magnetic Tape control operations which may be performed and the corresponding modes. The control area consists of two words, the mode and the reply word.

Operation	Mode
Write End of File	5
Rewind	7
Backspace	2
Forward to End of File	4
Back to End of File	6
Close a Magnetic Tape File (Release)	519
Open a Scratch Tape Reel and Retain it as a Scratch Reel	256

6.7.2 Magnetic Tape Input/Output Operations

The control area for read or write operations is four words arranged as follows:-

- Mode
- Reply word
- Number of words to be transferred
- Starting address, in memory

There are four possible modes which result in binary or BCD reading or writing.

Operation	Mode
Binary Read	0
Binary Write	1
BCD Read	8
BCD Write	9

At the end of the operation the reply word will be in one of the following conditions.

Zero	Transfer completed normally
Negative	Transfer still in progress
Positive	1 End of Tape 16 Long Block 64 End of File

6.7.3 Magnetic Tape Labelling Operations

Magnetic tape units need not be requisitioned at the start of a program. An object program may request from EXECUTIVE during executive time, a reel of tape by providing the label it requires and identifying this label with a specific unit number. EXECUTIVE will search all available tape units until it finds one with the appropriately labelled reel and will assign this unit to the requesting program.

Similarly, a new file may be labelled by providing the necessary label and asking EXECUTIVE to locate a scratch tape for the purpose.

The control areas required for these two operations are similar in that they consist of nine words as follows:

Word	Contents
1	Mode: 128 for opening an existing file 384 for labelling a new file
2	Reply
3 - 5	Label consisting of any 12 BCD characters
6	Reel number in binary form
7	File serial number in binary form
8	Retention cycle (number of days in binary form)
9	Date written (9 most significant bits containing year and 15 least significant bits containing day, both in binary form).

In opening an existing file EXECUTIVE will check the label and reel number and having found it will store the balance of the information in the area as shown above.

In labelling a new file, EXECUTIVE will insert the current date and will write the entire label.

Care must be taken to provide sufficient space for the labelling control areas, as is the case for all other types of magnetic tape operations.

6.7.4 Magnetic Tape Macro-Instructions

There are a number of macro-instructions provided to facilitate specification of magnetic tape control and labelling operations.

The following are operations for which AUTOCODER provides the basic MTU instruction as well as the required two word control area with the appropriate mode.

6.7.3 continued

WEF	X	- Write End-of-File on unit X
REW	X	- Rewind unit X
BSP	X	- Backspace unit X
FEF	X	- Forward unit X to next End-of-File
BEF	X	- Backspace unit X to previous End-of-File
CLOSE	X	- Rewind and release unit X
SCR	X	- Open a new File as unit X but do not change its label (Reel not to be kept after current run)

Two macro-instructions are provided for labelling operations. Both require that the labels be prespecified under some lower memory workspace directive at location N(M).

OPEN	X	N(M)	- Open existing file on some unit to be called X
LABEL	X	N(M)	- Label new file on some unit to be called X

These macro-instructions will set the mode in the control area accordingly and will then perform the required peripheral operation.

6.8 TYPEWRITER

177	TYP	X	N(M)

This instruction specifies an output operation on typewriter unit X under control of the contents of word N(M).

The characteristics of control word N(M) are similar to those described for paper tape reader (see 6.7.2).

6.9 INPUT/OUTPUT PACKAGE STATEMENTS

AUTOCODER provides for a special category of statements which are actually specifications to be used by the FP6000 Input/Output package at execution-time. This Input/Output package caters for conversion and editing of data in addition to controlling the peripheral operations in conjunction with the Executive program.

Entry into the input/output subroutine will require a calling sequence of the following generalized form:

OPN	UNIT
FORM	Format Description .....
$ITEM_1$	Item Description .....
$ITEM_2$	Item Description .....
.	.
.	.
.	.
$ITEM_n$	Item Description ..... END

For example, the following is a set of statements defining reading of 3 alphanumeric characters.

```
        READ    TR1
        FORM    'AAA'
        BCD     INPUTAREA,3,END
```

The following section will describe the elements of the Input/Output package statements in the order in which they occur in the generalized form.

6.10 TYPE OF INPUT/OUTPUT OPERATION

OPN gives the type of operation and can be one of the following:-

READ
- for input operations. Conversion and editing are from "external" form to "internal" form.

WRITE
- for output operations. Conversion and editing are from "internal" from to "external" form.

EDIT
- the same as WRITE, except that data is moved to a specified area of memory instead of being written to an output medium.

DEDIT
- the same as READ, except that data is taken from a specified area of memory instead of being read from an input medium.

6.11 PERIPHERAL UNIT

Unit specifies the source or destination of the data. With READ or WRITE, this field will indicate a peripheral unit in standard notation, which must have been previously defined under a PERIPHERAL directive (see 6.1) or in a magnetic tape labelling operation (see 6.7.3). With EDIT or DEDIT, this field may take any of the forms of the operand field of an instruction statement (see 3.5.5). This defines the starting location of data in memory. Editing or de-editing operations will proceed from that address under control of the Form and list items. The occurrence of an (R) in the Form statement will cause the editing or de-editing address to be reinitialized at the starting address specified.

NOTE:

In the remainder of this chapter, references will be made only to the use of input and/or output media but the reader may substitute 'core store' when EDIT or DEDIT operations are being considered.

6.12 FORMAT DESCRIPTIONS

The FORM statement describes the format of one or several records of information as they will exist in a peripheral unit. In the case of READ or DEDIT this will describe the incoming form of data while for WRITE or EDIT it will be the output form.

There are five basic elements used to make up a form statement.

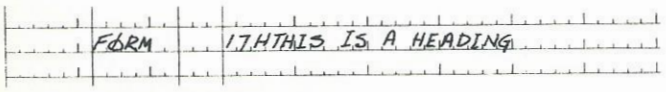
6.12.1 Spaces

Spaces between fields may be indicated by "nS" where "n" is an integer specifying the number of spaces. If the number is omitted then one space is implied.

6.12.2 Fixed Alphanumeric Headings

Fixed alphanumeric headings may also be specified. Each heading is preceded by "nH" where n, a decimal integer, is the number of characters in the heading.

Example:



6.12.3 Multiple Records

A form statement may describe more than one record. Each new record (except the first) must be preceded by the letter "R". If an integer (n) is placed before the "R" then (n-1) blank records will precede the record specified by the format following "R".

For example, the statement below describes two records (format 1 and format 2) separated by two blank records.

FORM format 1 3R format 2

6.12.4 Format Repetition

Any portion of a format may be repeated by enclosing it in parenthesis preceded by an integer indicating the number of repetitions.

Examples:

The statement

Format 1 2(format 2) Format 3

is the same as

Format 1 Format 2 Format 2 Format 3

Similarly,

Format 1 3(R Format 2)

will produce one record of Format 1 followed by three records of Format 2.

Format repetitions indicated by parentheses may be nested to any depth desired.

Example:

Format 1 3 (Format 2 2(Format 3))

6.12.5 Field Pictures

Pictures are used to indicate the form of alphanumeric and numeric fields. These pictures are enclosed in quotation marks. Repetition of a picture may be indicated by preceding the opening quotation mark with an integer.

Example: 2 'picture 1' 5 'picture 2'

Use of these integers is explained in section 6.14.

6.12.5 continued

As described below, specific characters can be used within each picture to describe the form of a field. An integer preceding any of these characters indicates the number of consecutive occurrences of this character.

6.12.5.1 Alphanumeric Fields

The following are the characters which may appear in a picture for an alphanumeric field, and their meaning:

A - this position will contain alphanumeric data

S - this position will contain a space

Any character other than A, S, quotation marks, or digits 0 to 9 - this position will contain the character indicated. For purposes of input these characters will be ignored, whereas they will be inserted in the output record.

Examples:

PICTURE	INTERNAL DATA	EXTERNAL
'AAAA'	JACK ST#1	JACK ST#1
'SS3A/2A'	APR63 128AB	sp sp APR/63 sp sp 128/AB

6.12.5.2 Fixed Point Decimal Fields

The following characters are allowed in a fixed point decimal picture:

X - this position will contain decimal data which is to the left of the assumed decimal point.

F - this position will contain decimal data which is to the right of the assumed decimal point.

S - this position will contain a space.

, - this position will contain a comma (,).

. - this position will contain a point (.) (This character may appear only once in the picture).

CR - these 2 positions will contain the characters CR, if the data is negative, spaces otherwise (These characters may appear only at the right-most positions).

DB - similar to CR, but using the characters DB.

$ - this position will contain a dollar sign ($) (See floating signs below).

\+ - this position will contain a plus sign (+) if the data is positive, a minus sign (-) otherwise (See floating signs below).

\- - similar to (+) except that positive data results in a space.

Z - similar to X, except that if the data is a leading zero, it is replaced by a space.

* - similar to X, except that if the data is a leading zero, it is replaced by a cheque protection symbol (*).

NOTES:

(i) Floating Signs

Multiple dollar, plus, or minus signs can be used to indicate data positions which may contain leading zeros (similar to Z and *). In this case, the leading zeros are replaced by spaces except for the right-most one which is replaced by the appropriate sign ($, +, space, or -). To allow for the case where there are no leading zeros, the number of signs in the picture should be one greater than the data positions.

(ii) Decimal Point Position

(1) With read-type operations, if the picture contains a point (.), the actual position of the point in the external data will override the position indicated in the picture.

(2) With either type of operation, scaling can be obtained by <u>not</u> aligning the point character with the assumed point position in the picture, ie. ,

'XXF. FF' scales by a factor of 10.

Examples are given overleaf.

6.4.5.2 continued

PICTURE		DECIMAL EQUIVALENT OF INTERNAL DATA	EXTERNAL
'3X'		19	019
		199	199
'3XFF'		19	01900
		217.65	21765
'XX.FFF'		1.358	01.358
'XXF.FF'		51.282	512.82
		319.67	3196.7
'S+XXX'		116	sp+116
		-23	sp-023
'SS---.F'		27.6	sp sp sp 27.6
		-1.9	sp sp sp -1.9
'$ZZ.FFDB'		2.16	$sp 2.16 sp sp
		-83.75	$83.75DB
		.32	$sp sp .32 sp sp
'$$,$$$.FF'	Note (1)	2.16	sp sp sp sp $2.16
'2$, 3$.FF'		1000.00	$1,000.00
'$***.**'		0	$***.**
'$ZZZ.ZZ'	Note (2)	0	sp sp sp sp sp sp sp

NOTES:

(1) When Z, * and floating dollar, plus and minus signs are used, leading commas are replaced, as well as leading zeros.

(2) If all data positions contain Z, $, +, or -, and the data is zero, the entire field will be replaced by spaces. (This is the only case where these suppression characters may appear to the right of the point).

6.12.5.3 Floating Point Decimal Fields

The following characters are allowed in a floating point decimal picture.

V - this position will contain a decimal digit of the mantissa.

S - this position will contain a space.

. - this position will contain the decimal point of the mantissa.

In addition, every picture of this type has the character (-) implied at the left end (but to the right of any S-characters) and the 4 characters (E-XX) at the right end. These characters have a meaning similar to that in the fixed point pictures. XX denotes a 2-digit decimal exponent (see below).

6.12.5.4 Variable Numeric Fields

The picture must contain the character (N) followed by any other single character except decimal point (.), plus (+), or minus (-).

This type of picture may be used for input only and specifies decimal numeric fields of variable length, separated by the character indicated. Signs and the location of the decimal point will be taken into consideration during the de-editing process.

Example:

FORM 15 'N,' specifies a record which contains fifteen decimal numeric fields separated by commas.

6.12.5.5 Variable Alphabetic Fields

This picture must contain the character (B) followed by any other single character. This indicates a variable, alphabetic field in the input record which will be terminated by the occurrence of the dividing character specified.

6.12.6 Rules for Writing Formats

(1) Spaces appearing in format descriptions are ignored by the AUTOCODER, except for those which are a part of a fixed heading. Thus, the programmer can use spaces to separate the fields in his description and must use the character S to specify the position of spaces in the actual Input/Output medium.

(2) If more than one line is required for a format description, then it may be continued on succeeding lines, with the operation code, FORM, repeated on each line.

(3) A field picture does not have to be complete on one line, it may continue on the next line.

(4) The programmer may use a format description written elsewhere in his program, by writing his Input/Output instructions in the following form:

```
OPN     UNIT     LABEL

Item    Item     Description .....
Item    Item     Description .....etc.
```

where LABEL is a 1 to 5 character symbol giving the starting location of the format description to be used. All characters of the symbol must be alphanumeric, and the first character is always alphabetic.

Example:

PICTURE	DECIMAL EQUIVALENT OF INTERNAL DATA	EXTERNAL
'V.3V'	198.6	sp 1.986E sp 02
	- 198.6	- 1.986E sp 02
	.02127	sp 2.127E - 02
'SS.3V'	.166	sp sp sp .166E sp 00
	-.0235	sp sp -.235E - 01

6.12.6 continued

Thus, any format descriptions to be used in this manner must have a label on the first line, and only the first line.

Example:

$LABEL_1$	FORM	Description
$LABEL_2$	FORM	Description
	FORM	Description

In the above example, the AUTOCODER would treat the first line as one format description, and the next two lines as another. These formats can only be written in the program area. In any other section the FORM pseudo instruction will be treated as an error.

6.13 ITEM DESCRIPTIONS

Each line of the item descriptions describes an item of data as it appears in the computer memory. Thus the combination of item and format descriptions specifies the data to be transmitted into or out of the computer memory, and the form it has or will take in the external medium, respectively. Each line of item description is written as follows:-

TYPE CT/DATA, N

6.13.1 Types of Internal Data

Internal data is split into two types indicated by the code used for TYPE:

BCD - this indicates that the data is in a binary-coded decimal form, in which 6 binary bits are used to represent each character. In format descriptions, alphanumeric pictures are used to describe its external form.

NUM - this indicates that the data is purely numeric and is in binary form. Fixed or floating point decimal pictures are used to describe its external form.

6.13.2 Item Counter

The operand CT specifies the number of consecutive items to which this description pertains. It is used in the same sense as an operand for non-branch instructions, that is, unless it is a literal, it represents the memory location of the counter. The least significant 15 bits of the word are used as the counter by the Input/Output package. The forms and restrictions of operands for non-branch instructions also apply (see 3.5.5).

CT may be a literal in one of the following forms:-

(a) a decimal or octal integer, enclosed in quotation marks.

(b) a symbol, plus or minus a decimal or octal integer, enclosed in quotation marks. This symbol must have a previously defined absolute value (see 2.3).

6.13.3 Data Location

The operand DATA specifies the lowest memory location occupied by the first item. Again, the forms and restrictions of operands for non-branch instructions apply (3.5.5) but literals are not allowed.

6.13.4 Item Length

The operand N specifies the amount of storage taken by each item and its form depends on the setting of TYPE:

If TYPE is BCD, N is a decimal integer giving the number of character positions (6 bits each) occupied by the item.

If TYPE is NUM, N can be:

(1) Two decimal integers separated by a decimal point, ie., $n_1.n_2$ where:

n_1 is the number of memory cells to be occupied by the item, ($n \leqslant 2$)

n_2 is the number of bits in the fractional part of the item, ($n_2 \leqslant 46$).

Thus, $n_1.n_2$ describes a fixed point binary item.

(2) The letter V describes a floating point binary item of standard double-length form, ie., a 38 bit mantissa and a 9 bit exponent at the right-most end.

6.13.5 Single-Item Descriptions

If an item description does not pertain to successive items, then it may be written:

TYPE DATA, N

where a unit CT is implied.

6.13.6 Repetition of Item Descriptions

To specify that a group of item descriptions be used a number of successive times, another type of line may be written:

RPT CT,n

where:

CT - gives the total number of items to be transmitted, using the group of descriptions. (CT can take the same forms as in the lines, BCD and NUM).

n - specifies the number of succeeding items which are to be repeated.

Example:

```
        BCD     CT1/DATA1,N1
        BCD     CT2/DATA2,N2
        RPT     CT
        NUM     CT3/DATA3,N3
        BCD     CT4/DATA4,N4
```

In the above example, the first two descriptions would be used once each, and the next two would be used repeatedly until the number of items specified by CT had been transmitted.

6.13.7 Ending an Input/Output Sequence

The last item description line in an Input/Output sequence should have the letters END as a final operand. Thus if the last line is of the BCD or NUM type, it takes the form:

TYPE CT/DATA, N, END

It is possible to have an Input/Output sequence with no item descriptions if the only data for transmission is a fixed heading in the format description. In this case, END appears on the top line.

OPN UNIT,, END

or

OPN UNIT, LABEL, END

6.14 CORRESPONDENCE OF FORM AND LIST ITEMS

In an output operation, the I/O package will generate information until the counters for all list items have been exhausted. For input purposes, record de-editing will proceed until all list items have been satisfied or until the end of the input record is reached. In both cases, the Format is used repetitively until the entire input or output operation is completed.

Example:

```
        FORM    '10A', 2S, '5X'
        RPT     '8',2
        BCD     NAME,10
        NUM     AMOUNT,1,END
```

This calling sequence will produce eight output records since the one record format will be used until all list items have been satisfied.

The I/O package will scan the FORM statement for the next picture when a new list item is selected.

Picture repetition is indicated by a preceding integer (n) where $0 \leq n \leq 127$ (omission of the integer is interpreted as 0). Similarly each list item has a counter (see 6.13.2).

The I/O package compares the list item counter with that preceding the corresponding picture and takes appropriate action as follows:

(i) If the list item counter is equal to or greater than the picture counter, editing or de-editing proceeds on the basis of the same format until the list item counter is exhausted.

(ii) If the list item counter is less than the picture counter, editing or de-editing is performed for the number of fields indicated by the list item counter while the remaining fields specified by the picture counter are bypassed.

6.14 continued

Example:

```
        WRITE   LPO
        FORM    '255X'
        NUM     COUNT/NAME,',END
```

The above calling sequence will print the number of fields indicated by COUNT. If COUNT is zero a blank line will be printed.

6.15 OPERATION OF PERIPHERALS

The Input/Output package provides different buffering capabilities depending upon the type of peripheral unit.

Three buffer areas will be provided for all peripheral units other than Magnetic tapes. One of these areas (32 words long) will be used as an input buffer for the card and paper tape readers while the other two (each 32 words long) will be used as output buffers for line printers, card punches, paper tape punches and typewriters.

When dealing with magnetic tapes the Input/Output package allows specification of any number and size of buffers. In addition the I/O package is capable of opening or labelling reels as required.

These specifications are established by means of a pseudo-instruction as follows:

Example:

```
        MTCON   X   NAME1(d1),NAME2(d2),N(M)
```

An MTCON pseudo-instruction must appear in the source program for each magnetic tape unit before reference is made to this unit in an I/O package calling sequence.

In the above instruction X specifies the unit number. NAME1 and NAME2 are names of buffer areas to be established to this unit. AUTOCODER will allocate upper common areas for these buffers which are dimensioned in accordance with the above specifications (d1) and (d2). The name indicated will become the name of the common area as well as the symbol of the first word in the buffer. Care must be taken that the names used to identify buffer areas are unique in any given compilation.

The Input/Output package requires that each magnetic tape unit addressed must be allocated at least one buffer. There are three possible buffer configurations which are used as follows:

(1) For Input Operations

(i) NAME1 (d1), 0

A read instruction is given and then the program is suspended until the operation is completed, to allow conversion of the input data.

(ii) NAME1 (d1), NAME1 (d1)

A check is initially made to determine if the next read operation has been completed. If not program suspension occurs. After the input data has been converted the next read is initiated before control is returned to the main program.

(iii) NAME1 (d1), NAME2 (d2)

In this configuration the Input/Output package always initializes the next read operation prior to conversion of the current record.

(2) For Output Operations

If one buffer is specified as in (i) or (ii), the I/O package will cause program suspension until the previous peripheral transfer has been completed. If two different buffers are indicated as in (iii), data assembly and conversion can proceed while the previous peripheral transfer is in progress.

6.15 continued

The same buffer areas may be allocated to more than one magnetic tape unit. Care must be taken that this does not result in a conflict. In the case of an input unit the buffers are declared busy until the data is converted and stored as specified by the list.

Thus, if a number of input units share the same two buffers the situation may arise whereby the program will become suspended indefinitely due to lack of available buffer areas.

In the case of output operations buffers are declared busy from the beginning of the data assembly and conversion operation until the peripheral transfer has been completed.

The I/O package will store in register L9 the address of the buffer released. After an input operation it will be the address of the buffer from which information was de-edited. For output operations it will be the address of the buffer from which writing was completed. Zero address indicates that no buffer has been released.

If requested, the Input/Output package will also open existing files or label new ones. In addition if end-of-reel is detected during a writing or reading operation the current tape reel will be automatically closed and a new file will be opened after the reel serial number has been incremented. The address of the label control area as described in 6.7.3, is the third item in the operand field of the MTCON pseudo-instruction shown as N(M). If this item is not included or is specified as zero, the Input/Output package will not open files and if during a writing or reading operation end-of-reel is detected this will be indicated in Register L18 as shown in 6.16.

6.16 REPLIES FROM INPUT/OUTPUT PACKAGE

Two registers are used by the Input/Output package to indicate results of a requested peripheral transfer.

Register L18

Sign bit will be on if end-of-file is detected on an input magnetic tape unit.

The least significant bit will be on if during a magnetic tape reading or writing operation end-of-reel is detected. Under these conditions the specified peripheral transfer will be completed. (Note that this condition will never be indicated if the Input/Output package is to perform magnetic tape labelling and opening functions).

Register L19

For paper tape and magnetic tape input operations L19 will have one of the following values.

Zero

This indicates that the record or records read were as long as the format specified.

Positive Integer

This indicates that a short record was read and the integer represents the length of the record in characters.

Negative Integer

This indicates that a long record was read. The value of the integer has no significance.

For all other peripheral transfers L19 will always be zero.

CHAPTER 7

PROGRAM SEGMENTATION

TABLE OF CONTENTS

CHAPTER 7

PROGRAM SEGMENTATION

Larger programs are usually divided into several pieces which we will call segments. Segments are often written by different programmers and/or in different languages. However, even when segments are written in the same language, they are compiled separately. This means that definitions of symbols must appear in every segment where they are used but it allows parts of the program to be put on the machine (for checkout) before others are ready, and allows changes to be made to one segment (say) without recompiling the whole program.

From the compiler's point of view, each segment is an independent program and there can be no communication between segments until their object versions are joined for execution. However, the programmer must be able to control the form of each object segment so that they will be joined correctly. This he specifies at the source level as explained in sections 7.1 and 7.2.

7.1 COMMON DATA

During execution of the complete object program, the data of each segment normally uses locations separate from those of other segments. However, to allow cross-referencing of data items, it is possible to allocate memory space to be shared by some or all segments.

Directives I and II: LOWER and UPPER for COMMON

(1) Blocks of lower or upper data storage to be shared by other segments, are defined under LOWER or UPPER directives respectively, with the symbol COMMON written in the operand field. This is followed by the slash character "/", and a block-name which is a symbol 1 to 11 characters in length, followed by a second slash.

Examples:

```
#LOWER   COMMON/INPUTS/
#UPPER   COMMON/TABLES/
#UPPER   COMMON/AREA15/
#LOWER   COMMON//
```

A block-name of all blanks is allowable.

(2) As with other LOWER and UPPER directives, the COMMON type can appear anywhere in the source program, provided that:

(a) The symbolic data-names appearing under each directive are preceding their use in the program.

(b) Each such directive (whether LOWER or UPPER) within the same segment, has a different block-name.

(3) When the resulting object program is loaded with other segments, each COMMON block will occupy the same storage locations as blocks of the same name in the other segments. A COMMON block in one segment will start at the same location as the same block in other segments, and the segment with the largest storage requirement for the block determines the total storage allocated to it.

Note that the data-names appearing under corresponding directives of segments may or may not be the same. These symbols do not appear in the object programs and it is only the block-name that indicates that two COMMON blocks are the same.

7.1 continued

(4) If a COMMON block-name appears in the operand field of a LOWER directive in one segment, then that block will be, during execution, in lower storage, even if it appears in UPPER directives of other segments.

(5) COMMON blocks can be used by any segmens of a program. However, any COMMON block which is used by an autoroutine segment and a master segment must be defined by an UPPER directive.

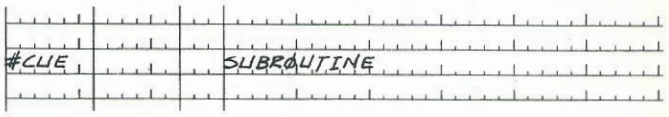
7.2 CUES FOR PROGRAM LOCATIONS

As with data-names, symbols referring to program locations normally are not retained in the object program but are replaced by numeric values depending on the order of the locations in the program area. However, if a program location is going to be referred to by another segment then its symbolic name must be saved until the corresponding object programs are loaded together for execution. The directive described below is used to indicate which program locations require this treatment.

Directive VI: CUE

(1) This directive is written with a symbol 1 to 11 characters in length in the operand field.

Example:



(2) The symbol in the operand field will be assigned a value equal to the location of the instruction or data statement which immediately follows the directive, and is the cue-name by which other segments may refer to this statement.

(3) If a symbol also appears in the label field of the statement, it is assigned the same value but may be used only within its own segment.

(4) A CUE directive must appear before every statement which is subject to these "outside" references. Each cue-name must be different and, since it may be used in any segment, it must not appear in any other CUE directive or be used as a COMMON block-name anywhere in the whole program.

(5) All symbols used in the operands of branch instructions, which are not defined by appearing in a label field of the same program segment, are assumed to be cue-names defined in other segments.

CHAPTER 8

SUMMARY OF DIRECTIVES

TABLE OF CONTENTS

CHAPTER 8

SUMMARY OF DIRECTIVES

8.1 MISCELLANEOUS DIRECTIVES

The following are directives which have not previously been defined.

Directive VII: PROGRAM

(1) This directive precedes each new section of program (see 8.2) and normally is written with a blank operand.

#PROGR AM

(2) This directive must also appear as the first statement, physically, of each AUTOCODER segment, with the following items in the operand field:-

(i) A single digit giving the program type

0	a master program
1	auto-routine 1
2	auto-routine 2

(ii) The 4-character name for the program, to be used when communicating with EXECUTIVE.

(iii) A 2-digit decimal number giving the priority to be assigned to this program (if this is not specified, a value of 01 will be assumed).

(iv) A slash character "/", followed by the 1 to 11 character name assigned to this segment of the program. Note that this segment name can be used to refer to the first program location of the segment, just as if it had appeared on the CUE directive immediately preceding the first non-directive statement in the program area, (see Section 7.2).

```
#PROGRAM   QJACK43/MAINSEG
#PROGRAM   ZFILE/SUBJECT6
#PROGRAM   /BILL09/BILLMASTER
```

If the segment is to be incorporated into the FP6000 library, then the program type, name, and priority should be omitted.

When segments are subsequently joined for execution, routines of the library type are included only if called by another segment.

Examples:

```
#PROGRAM   /SQRT
#PROGRAM   /MATRIXINVERT
```

Directive IX: ENTRY

(1) This directive must immediately precede the instruction at which EXECUTIVE is to start execution of the object program under normal conditions. It is written with a zero in the operand field.

```
#ENTRY     0
```

(2) The same directive can be written preceding other instructions which are to be used as alternative starting points. A maximum of nine such directives are allowed in each program (not each segment) with a different digit in the range of 1 to 9 written in each operand field.

Examples:

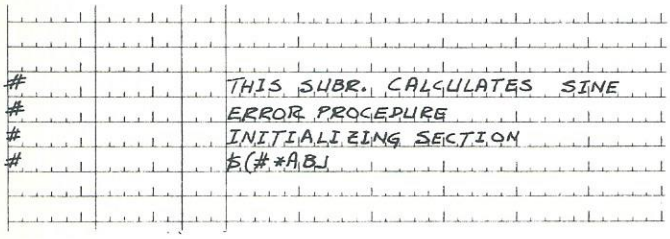
```
#ENTRY     8
#ENTRY     4
```

When the object program is in the computer, the operator can cause EXECUTIVE to start (or re-start) the program at any of these points by specifying an absolute location dependent on the digit used with the corresponding ENTRY directive.

(3) Any or all of the nine alternative entry points may be specified, but the normal entry (with zero operand) must be specified, if it is not required to start the program at the first compiled instruction.

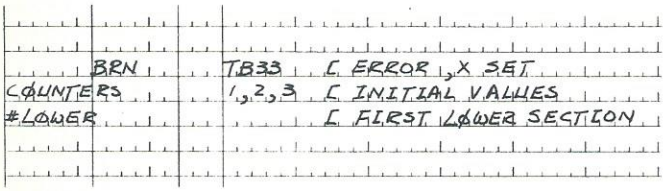
Directive X: for comments

(1) This directive is used to insert comments in the source program and is written with a blank name, and with the comments in the operand field. These comments will appear in the source listing, but will have no effect on the object program. They may be any combination of characters.



In addition, comments may be written on the same line with other statements, through the use of this left square bracket '['. The appearance of this character in the operand field, causes the compiler to treat everything to the right of the bracket as comments (unless the bracket is one of the n characters of an alphanumeric field preceded by nH).

Examples:



Directive XI: END

(1) This directive must appear as the last line in a program, or segment of a program. It causes the compilation to be completed.

#END

8.2 DIRECTIVES BY GROUPS

As can be seen from the previous chapters, a certain group of the directives control the interpretation of "non-directive" lines following them. That is, the way in which the compiler interprets any non-directive line depends on which directive of this group was last encountered. The appearance of any directive in this group ends the effect of any previous one in the same group. Specifically these directives are:-

(i) LOWER

(ii) UPPER

(iii) PERIPHERAL

(iv) MACRO

(v) PROGRAM

(vi) END - this is a special case which ends the compilation and therefore nothing follows it.

The programmer must be careful to insert these directives wherever necessary. For instance if a section of LOWER or a MACRO definition appears in the body of the program then a PROGRAM directive must follow before returning to statements for the program area.

Two other directives are used only in the program area and pertain to the first non- directive line following. These are:-

(i) CUE (ii) ENTRY

The remaining directives may appear anywhere in the program:-

(i) SET
(ii) Comments
(iii) DEFINE

SUMMARY OF OPERATIONS

Octal Code	Mnemonic Code	Time - usec 6 usec	Time - usec 2 usec	Page Ref.
	ARITHMETIC INTO ACCUMULATORS			
000	LDX			
001	ADX			
002	NGX	18	7	3-4
003	SBX			
	ARITHMETIC INTO STORE			
010	STO			
011	ADS			
012	NGS	18	7	3-4
013	SBS			
	ARITHMETIC WITH SMALL INTEGERS			
100	LDN			
101	ADN			
102	NGN	12	5	3-4
103	SBN			
	ARITHMETIC WITH CARRY			
004	LDXC			
005	ADXC			
006	NGXC			
007	SBXC	18	7	
014	STOC			
015	ADSC			3-4
016	NGSC			
017	SBSC			
104	LDNC			
105	ADNC			
106	NGNC	12	5	
107	SBNC			
	MULTIPLICATION			
040	MPY	67	40	
041	MPR	67	40	3-4
042	MPA	72	41	
	DIVISION			
044	DVD	76	45	
045	DVR	79	48	3-5
046	DVS	71	44	

Octal Code	Mnemonic Code	Time - usec 6 usec	Time - usec 2 usec	Page Ref.
	COMPARISON			
026	TXU	18	7	3-5
027	TXL			
	CHARACTER CONVERSION			
043	CDB	58	27	3-5
047	CBD			
	FLOATING POINT CONVERSION			
130	FLOAT			3-5
131	FIX			
	FLOATING POINT ARITHMETIC			
132	FAD	58+N (max)	46+N (max)	
133	FSB			
134	FMPY	39 max	27 max	3-5
135	FDVD	64 max	52 max	
136	LFP	18	6	
137	SFP	19	7	
	, FLOATING POINT NORMALIZE			
114	NORM	37+S	15+S	3-6
115	NORM	42+S	16+S	
	LOGICAL OPERATIONS			
020	ANDX			
021	ORX			
022	ERX			
030	ANDS	18	7	
031	ORS			3-6
032	ERS			
120	ANDN			
121	ORN	12	5	
122	ERN			
	PART WORD MANIPULATION			
024	LDCH			
025	LDEX			
034	DCH			
035	DEX	18	7	3-6
036	DSA			
037	DLA			

Summary of Operations (continued)

Octal Code	Mnemonic Code	Time - usec. 6 usec	Time - usec. 2 usec	Page Ref.
	SHIFTING			
110 N_t=0	SLC			
N_t=1	SLL	18 + N	6 + N	
N_t=2,3	SLA			
111 N_t=0	SLC			
N_t=1	SLL	42 + N	15 + N	
N_t=2,3	SLA			
112 N_t=0	SRC			3-7
N_t=1	SRL			
N_t=2	SRA	18+N	6+N	
N_t=3	SRAV			
113 N_t=0	SRC			
N_t=1	SRL			
N_t=2	SRA	42+ N	15+N	
N_t=3	SRAV			
	BRANCH ON STATE OF ACCUMULATOR			
050	BZE			
052	BNZ			
054	BPZ	13	5	3-7
056	BNG			
	CONDITIONAL BRANCHES			
074 X = 0	BRN			
X = 1	BVS			
X = 2	BVSR			
X = 3	BVC			
X = 4	BVSR	7	3	3-8
X = 5	BCS			
X = 6	BCC			
X = 7	BVCI			
	INDEXING AND COUNTING			
060	BUX			
062	BDX	13	5	3-8
064	BCHX			
	SUBROUTINE LINKAGE			
070	CALL	15	8	3-8
072	EXIT	12	4	

Octal Code	Mnemonic Code	Time - usec. 6 usec	Time - usec. 2 usec	Page Ref.
	BLOCK TRANSFER AND CHECK SUM			
126	MOVE	32+12N	13+4N	3-9
127	SUM	32+7N	14+3N	
	MISCELLANEOUS			
023	OBEY	7	3	
033	STOZ	18	7	
123	NULL	7	3	3-9
124	LDCT	13	5	
125	MODE	8	4	
	PERIPHERAL CONTROL			
150	SUSBY			
151	REL			
152	DIS			
153	SPW			3-9
154	CONT			
155	SUSDP			
	PERIPHERAL USE			
170	PTR			
171	PTP			
172	LNP			
173	CDR			3-10
174	CDP			
175	MTU			
176	DRM			
177	TYP			
	INTERRUPT AND ABOLISH			
156	ABOL			
157	SUSWT			3-10
160	SUSTY			
	AUTOROUTINE CONTROL			
162	SUSMA			
163	AUTO			3-10
164	SUSAR			

NOTE: Times shown above do not include address modification.
One cycle (ie. 2 or 6 usecs) should be added for these cases.