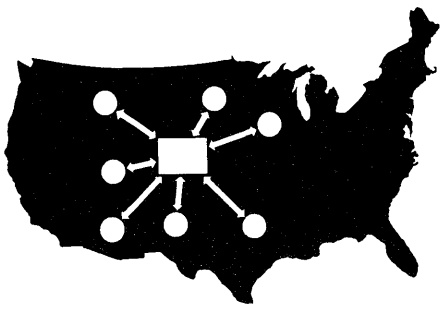
DATANET-30 Assembly Program Reference Manual



ADVANCE INFORMATION

CPB-1074

DATANET-30
ASSEMBLY PROGRAM
REFERENCE MANUAL

September 1964

GENERAL ELECTRIC

COMPUTER DEPARTMENT

CONTENTS

APPENDIXES

1. INTRODUCTION

It is assumed that the reader is familiar with the basic principles of assemblers. Readers who are not familiar with assemblers should read Appendix A to this manual, "Introduction to Symbolic Programming," before attempting the rest of the manual.

The DATANET-30 Assembler will accept either of two source-card formats, the one described below, or the "old" format used by the previous DATANET-30 General Assembly Program. Persons wishing to use the old format should read Appendix B, "Compatibility with Previous Card Format."

MACHINE REQUIREMENTS FOR ASSEMBLING

There are always two questions which must be answered before a piece of software can be written: 1) what memory size will be required, and 2) what peripherals will be required? The manufacturer is always torn between, on the one hand, writing software that will run on a minimum machine configuration so it will be available to all customers, and on the other hand, writing more powerful software which may require larger configurations. For the DATANET-30 Assembler, General Electric chose to go the "unlimited" route, requiring at least an 8k memory. However, General Electric will also provide a modified assembler for the 4k DATANET-30. The 4k assembler will lack some of the "nice to have but not essential" features of the 8 and 16k assemblers, and will have smaller Symbol Tables. The following features will not be available in the 4k assembler:

> old card format, subroutine call, automatic update, title card, symbol table dump, symbol table load, prefix, sequence check, "percent variation" of remarks card, field definition and multifield constants, micro programming, control card, resequencing of source program on master file.

The choice of required peripherals was much more difficult to make. For example, a DATANET-30 may be free-standing with no conventional computer peripherals. It may tie directly to a 200 Series, 400 Series, or 600 Series General Electric computer, or even to a competitor's computer. It may share peripherals with any other General Electric computer. With the wide variety of configurations being used, it was impossible to arrive at any standard peripheral configuration for assembling. Therefore, the problem was solved by stepping around it.

The DATANET-30 Assembler works with "files" instead of peripherals. The assembler never addresses any peripheral devices. Instead, it branches to predetermined memory locations, which are assumed to link to subroutines in an Input/Output Package, or I/O PAC. The I/O PAC is a "plug-in" subprogram which works with the assembler. One can change peripheral configurations by changing I/O PAC's.

For example, when the assembler needs a new source card, it branches to a particular linkage point in the I/O PAC. When the I/O PAC returns to the assembler, the assembler expects an 80-column BCD record at a given memory location. The assembler doesn't care whether the record came from an on-line card reader, or from a magnetic tape written by a satellite computer, or from another computer through a computer interface unit, or from another computer over a high-speed transmission line, or from a dual-access disc storage unit (DSU) where it was placed by another computer, or from Teletype or from a paper tape reader. The assembler doesn't care whether the record was always in BCD code or whether it originated in some other code and was converted by the I/O PAC. The assembler doesn't care whether the record was originally 80 columns or whether it was a condensed record which was spread and blank-filled by the I/O PAC.

General Electric will provide I/O PAC's for a variety of peripheral configurations. General Electric will also provide the specifications, as part of this manual, for writing I/O PAC's. Thus, customers may modify I/O PAC's or write their own I/O PAC's to better tie in with their own operating systems. Not all I/O PAC's will make use of the full power of the assembler. For example, automatic update of source programs or subroutine call from library are not really practical unless one has access to magnetic tapes. The most basic assembly requires a minimum of one input device and one output device (capable of handling both BCD and binary data). If a DSU is on-line, it could serve for more than one file. The DATANET-30 can assemble with nothing but a DSU, provided there is some way to get the source program onto the DSU and the object listing off the DSU. (This might be accomplished by a conventional computer which shares the DSU with the DATANET-30.)

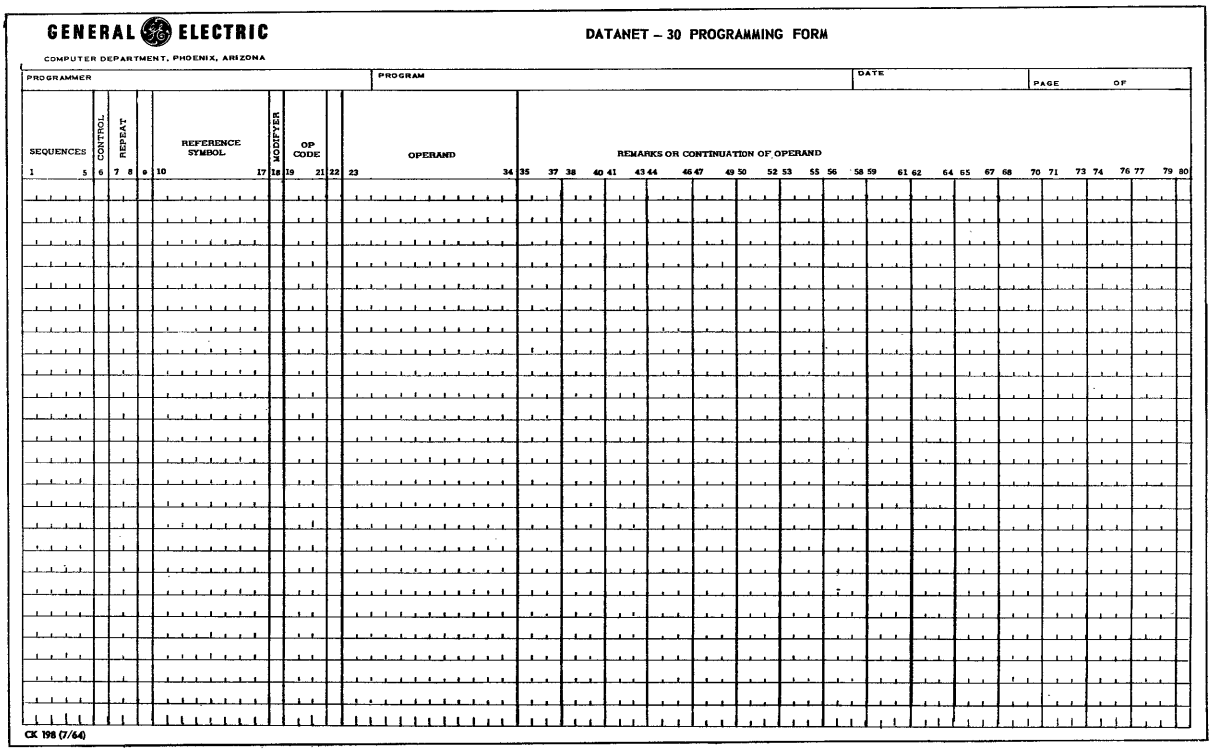
Appendix C of this manual discusses some of the principles behind the I/O PAC, and the pros and cons of buffering and blocking for each file. It also gives specifications for each file, such as recording mode, record size, working area, subroutine linkage, etc.

2. SOURCE-CARD FORMAT

The DATANET-30 is a single-address machine and therefore, only three fields are essential for assembly. Those three are the Symbol field for identifying an instruction, the Op Code to designate what action is to be taken, and the Operand field to designate which is to be acted upon. However, several other fields are provided for convenience. The fields will be discussed in the order in which they appear on the source card, left-to-right.

A. SEQUENCE FIELD, columns 1-5

The Sequence field is provided for the convenience of the programmer, and is normally ignored by the assembler. However, the programmer may, by pseudo-operation, instruct the assembler to check the source cards for ascending sequence. The programmer may restart sequencing as often as necessary. For example, canned subroutines frequently contain their own sequence numbers and it is necessary to restart the sequence check for each new subroutine.


GENERAL ELECTRIC

COMPUTER DEPARTMENT, PHOENIX, ARIZONA

DATANET – 30 PROGRAMMING FORM

PROGRAMMER | PROGRAM | DATE | PAGE OF

SEQUENCES 1 5 | CONTROL 6 | REPEAT 7 8 9 | REFERENCE SYMBOL 10 17 | MODIFYER 18 | OP CODE 19 21 22 | OPERAND 23 34 | REMARKS OR CONTINUATION OF OPERAND 35 37 38 40 41 43 44 46 47 49 50 52 53 55 56 58 59 61 62 64 65 67 68 70 71 73 74 76 77 79 80

CK 198 (7/64)

DATANET-30 Programming Form

If sequence checking is requested, the assembler will flag, on the object listing, any source card whose sequence number is not greater than that of the preceding card. All nonnumeric characters in the Sequence field will be treated as zeros.

B. CONTROL FIELD, column 6

This field is normally blank. It has several uses, which are discussed at various places elsewhere in this manual.

C. REPEAT FIELD, columns 7-8

This field is normally blank. It can serve any of several purposes. With most operation codes, the Repeat field tells the assembler to repeat the instruction contained on the card the indicated number of times. For example, if the programmer wants a 67-word block of memory preset to zero, he may write a DEC 0 instruction with the number 67 in the Repeat field.

On those operation codes used to enter messages in BCD or Teletype codes, the Repeat field tells how many computer words are required for the message. For example, to enter the message OUT OF STOCK on one card, the programmer would write the number 4 in the Repeat field because the message (12 characters) uses four words. If the Repeat field is blank on a message card, the assembler assumes a 1-word message.

D. SYMBOL FIELD, columns 10-17

This field has the same rules that apply to most assembly programs. If no symbol is desired, the field is left blank. If a symbol is desired, the field may contain up to 8 characters, at least one of which must be nonnumeric. A plus or minus sign and an asterisk are not allowed. Leading or imbedded blanks are deleted by the assembler as it analyzes the Symbol field.

On most cards, the use of the Symbol field is optional. On a few pseudo-operations, such as EQU, a symbol is mandatory. On some pseudo-operations, such as END or ORG, a symbol is either prohibited or ignored. On some pseudo-operations, the Symbol field is used, not for a symbol but for special controls, which are described with the pseudo-operation.

E. MODIFIER FIELD, column 18

This field is normally blank. It has many miscellaneous uses, which are described briefly here and in detail later.

D. This instruction must be placed at an even location.

M. This card contains multiple operands. See discussion of Nonrepetitive Tables. The assembler will generate one instruction for each operand on the card. The operands must be separated from each other by slashes.

I. If on an SBR card, means call subroutines immediately. On all other cards, means this card contains more than one instruction. Only the first instruction may contain a Reference Symbol. The instructions are separated from each other by slashes.

Example

```
MOVE  IPIC  O/LDD  $IN/STD  $OUT/AIC  2/XCZ  28/BNZ  *-4
```

S. The Reference Symbol should not be prefixed.

O. Symbols in the Operand should not be prefixed.

A. Use absolute address in the MIC instruction.

/ Ignore the Reference Symbol on this card. May be used to label the rows in a table for documentation.

F. OPERATION CODE (OP CODE) FIELD, columns 19-21

All Op Codes, both machine and pseudo, are three characters. A blank Op Code or an illegal Op Code will be flagged and replaced with a HALT.

G. OPERAND FIELD, columns 23-30

The Operand field is free-floating format. It may start anywhere between columns 23 and 30, inclusive, and may extend to the end of the card. The operand is terminated either by a blank or, if indirect addressing is used, by a comma. For this reason, imbedded blanks are not permitted within the operand, even between elements in an expression.

The operand may take any of several forms, depending partly upon the Op Code used.

1. Machine Memory Addresses.

   a. Numeric. Memory addresses may be written as numbers. In most cases, the number is assumed to be decimal. On some pseudo-operations the number is assumed to be octal.

   b. Symbolic. Since any machine instruction may be given a symbol, other instructions may refer to labeled instructions symbolically. Of course, a symbol does not have to refer to a memory address. By use of the EQU pseudo-operation, one may equate a symbol to some number or other symbol. The rules pertaining to symbols are defined in Chapter 3.

   c. Self-addressing: the asterisk convention. An asterisk in the operand of an instruction always represents the address of the instruction itself. The asterisk is used most often in "expressions" (see below) to refer forward or backward a few locations from the instruction itself.

   d. Expressions. Memory addresses may be defined by expressions which are the sum and difference of numbers, symbols, and/or asterisks. An operand may contain any number of elements (i.e., numbers, symbols, asterisks) but all elements must fit on one card. (A few pseudo-operations are restricted to one or two elements.) Thus, the instruction

      ```
      LDB *+TAX-SYMBOLZ+81+START
      ```

      is perfectly valid.

2. Register-Transfer and Shift Instructions

   These instructions do not refer to memory addresses, but to a limited number of registers or data lines. Operands for these instructions can take only one form. They must always consist of a FROM GROUP and a TO GROUP separated from each other by commas. Each register must be specified only by the character assigned to it in the DATANET-30 programming manual. Thus, to transfer the B-register to the A-register, the following is used:

   ```
   TRA B,A
   ```

   Several registers in either or both groups may be specified. Thus to transfer the LOGICAL OR of the receive data lines and the A-register to both the B- and C-register, the following is used:

   ```
   TRA AR,BC
   ```

3. Status Lines and Function Drivers.

Some instructions (e.g., DEF, NIS) refer neither to memory locations nor registers but to individual status lines or function drivers. Each line or driver is represented by one of ten bits in the Operand field. Each bit is identified by a digit, 0 through 9. To drive external function number 6, the following is used:

```
DEF 6
```

It is possible to call for several status lines or function drivers at the same time. For example, to drive external functions one, seven, and three all with one command, the following is used:

```
DEF 173
```

One may also call for status lines or function drivers symbolically by defining the symbol to represent the desired bit configuration. For example, interrogating the parity flip-flop requires ANDing Internal Status Line #0. This can be done as follows:

```
PARITY CDN 1000
       NIS PARITY
```

4. Literals. It is frequently necessary to include constants, parameters, messages and the like in object programs. In many cases, such data must be entered, not as binary numbers, but in some code such as BCD or Teletype. To facilitate this, the assembler contains several pseudo-operations which will insert the constants or messages in any of several codes.

In several pseudo-operations, it is permissible to enter several words of a message on one card. Because a message may contain leading, embedded, or trailing blanks, the assembler cannot determine from the message itself where the message begins and where it ends. Therefore, messages must always start in column 23 (the first column of the Operand field) and the programmer must tell the assembler, in the Repeat field, how long the message is.

In summary, then, there are seven types of operands:

a. Decimal
b. Octal
c. Symbolic
d. Asterisk
e. Register-transfer and shift
f. Status-line/function driver
g. Literal

Certain types of operands apply only to certain operation codes.

H. REMARKS FIELD

Remarks may start anywhere after the Operand. The only requirement is that there be at least one blank between the Operand and Remarks field. Normally remarks will always start in the same column for neatness of documentation, moving the remarks over only for cards with extra long operands. Columns 9 and 22 are not used for coding purposes. They may, however, be used on a remarks (*) card.

3. SYMBOLS (REFERENCE LABELS)

A. ORDINARY SYMBOLS

Symbols may be up to eight characters long. Ordinary symbols must contain at least one non-numeric character and at least one character other than a pound sign. They must not contain commas or plus or minus signs. Ordinary symbols of seven or fewer characters may be prefixed by the assembler (see PFX pseudo-operations).

B. REDEFINABLE SYMBOLS

A set of eight symbols, consisting exclusively of pound signs, is reserved for special usage. These symbols are never entered into the assembler symbol table, but are kept in a separate table. Any of the symbols in this set may be reused as often as desired. Each time the symbol is reused, it takes a new value. Any time a redefinable symbol is named in an Operand field, it equals the last value assigned to that symbol. Redefinable symbols can be referred to only in a backward direction.

Redefinable symbols are much more restricted than are ordinary symbols. For example, they cannot be equated to other values. They cannot be forced to odd or even locations. They are illegal in some pseudo-operations.

Redefinable symbols can be used in lieu of the asterisk in tight loops as illustrated below.

```
      PIC     0
#     LDD     $INPUT          MOVE INPUT TO OUTPUT
      STD     $OUTPUT
      AIC     2
      XCZ     38
      BNZ     #
#     NIS     7               WAIT FOR SELECT TO FINISH
      BNZ     #
#     CSR     6               WAIT FOR PRINTER READY
      BEV     #
```

The main reasons for using redefinable symbols, however, is to reduce the demand on the symbol table, and to reduce the probability of symbol conflict between subroutines and program segments.

C. MACHINE INSTRUCTIONS

The machine instructions are described in the DATANET-30 Programmers Reference Manual, and need not be discussed here.

D. ADDRESSING MODES

If an operand address is to be generated as a channel table, it must be so indicated. Normally, a channel-table operand will contain only one element, although it can be an expression if the resulting address is modulo 16. An expression will be assigned channel-table addressing if and only if the first element in the expression is a channel table. Thus, $INPUT+16 will be assigned channel-table addressing, but 16+$INPUT will not. Any symbol which starts with a dollar sign will be considered a channel table.

If channel-table addressing is not requested by the program the assembler will automatically select program-bank or common-data-bank addressing, as appropriate.

Because of the free format, no column is assigned for indirect addressing. Therefore indirect addressing is indicated by a comma immediately following the operand. For improved documentation, follow the comma by another character, as in the second example below:

```
LDB INPUT,
STB OUTPUT,X
```

4. DATA GENERATING PSEUDO-OPERATIONS

In this manual, the pseudo-operations are divided into two major classifications:

1. Data generating, those which generate constants or other data which become part of the object program.

2. Directive, those which do not become part of the object program, but which tell the assembler how to assign memory in the object program, how to use peripherals during the assembly process, how to document the listing, etc.

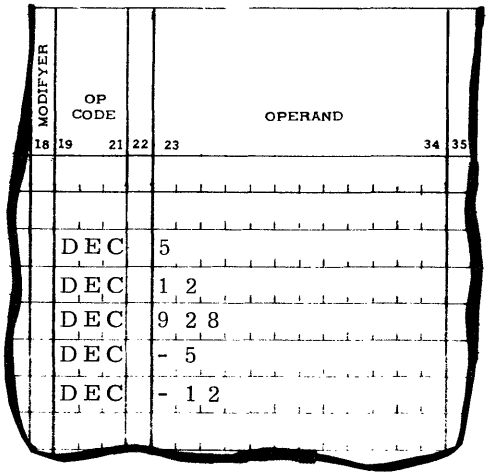
For convenience, data generating pseudo-operations are further divided into groups for generating "constants," for generating indirect addressing pointers, for generating tables, and for generating messages.

A. CONSTANTS

Decimal Number

DEC

This pseudo-operation places the binary equivalent of the operand into the object program. The operand will most often be numeric. When a numeric operand is used, the number will be evaluated as a decimal number. The operand may also be symbolic. A symbolic operand will represent either a memory address or some numeric value defined by an EQU, EQO, or CDN pseudo-operation. Operands may also be sum-and-difference expressions of numbers and/or symbols. Numbers must be not greater than 131,071 nor less than -131,071. Leading zeros are ignored and the number right-justified.



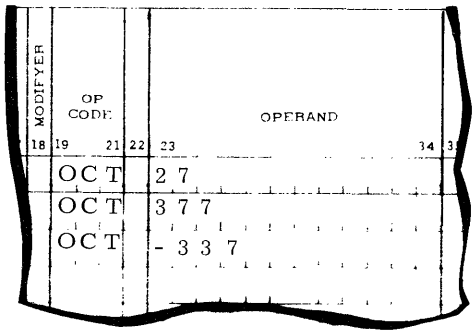
APPEARS IN MEMORY

000005
000014
001640
777773
777764

Octal Number

OCT

Generates a one-word constant in the object program. The operand may be either numeric or symbolic, or a sum-and-difference expression of numerics and/or symbolics. When a numeric operand is used, the number is interpreted as octal, and must not contain any digits greater than seven. A DATANET-30 word (18 bits) contains space for a maximum of six octal digits. All octal numbers will be right-justified by the assembler. Leading zeros will be ignored and therefore need not be supplied. For example, OCT 77 is the same as OCT 000077.



APPEARS IN MEMORY

000027

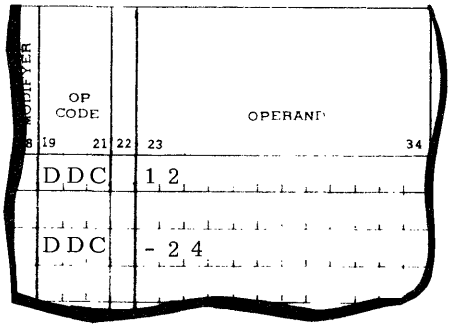
000377

400337

Double-Decimal Constant

DDC

This is the same as DEC except that it generates two data words instead of one. Unless told to do otherwise, the assembler will always place a DDC so that the most-significant word is at an even memory address. The allowable number range on the DDC is -34,359,738,367 to +34,359,738,367.



APPEARS IN MEMORY

000000

000014

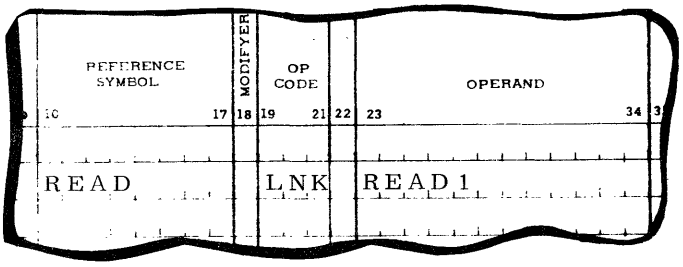
777777

777750

Subroutine Linkage

LNK

This pseudo-operation generates two words; the first will be a zero, and the second will be the address specified in the operand. Unless told to do otherwise, the assembler will force the LNK pseudo-operation to an even location. For example, a subroutine, whose calling name will be READ, starts at a location called READ1. One could set up the linkage as follows:



APPEARS IN MEMORY

000000

ADDRESS OF READ1

The first generated word, which would be at an even location, would contain a zero. The second would contain the address of READ1. The DDC pseudo-operation would accomplish the same thing.

Micro Programming

MIC

Occasionally, it is necessary to set up a word containg certain known bit combinations with addresses whose values are not known before assembly. It is for this purpose that the MIC instruction is provided.

The MIC instruction requires two operands. The first operand forms a "base" with which the second operand will be combined. The first operand may be an octal number or a numeric/symbolic sum-and-difference expression. If it is an expression, the elements will be ORed together. The first operand must be followed by a space, or by a comma, (or by a comma and a space). A comma in this position will not cause indirect addressing.

The second operand comprises an address which will be calculated as a unit and then ORed together with the "base" described by the first operand. The second operand may be a decimal number, a symbol or a sum-and-difference expression. If the second operand is an expression, the elements will be added. If the second operand is terminated by a comma, the indirect-addressing bit will be turned on.

After the second operand has been calculated, it will be converted to machine form, i.e., program-bank, common-data-bank, or channel table, and will then be ORed with the base provided. If, however, the Modifier field contains the letter A, the address will not be converted to machine form, but will be in absolute binary.

Examples:

Construct a word containg bits 17 and 18 and the address, in machine form, of READ+18.

```
MIC   600000,READ+18
```

Construct the same thing, but with the address of READ+18 in absolute binary form

```
AMIC  600000,READ+18
```

B. INDIRECT ADDRESSING POINTERS

In the DATANET-30, indirect addressing may be done with no indexing, or with indexing by one of the three registers, A, B, or C. The indication of indexing, plus the indirect address as an absolute binary number, are contained in one word, which may be set up by one of the following pseudo-operations.

Indirect Address with no Indexing

IND

Used to generate a constant, where the constant is a memory address. The operand may be a decimal number, a symbol, or a sum-and-difference expression. If numeric, it is assumed to be a decimal number and is converted to binary. If symbolic, the address of the symbol is used.

Examples:

```
IND     1280
IND     *
IND     *+INPUT+2
```

Indirect Address Indexed by the A-Register

INA

Similar to IND except that a bit is set in this word so that when it is used as an indirect address, the contents of the A-register will be added to the memory address portion of this word.

Indirect Address Indexed by the B-Register

INB

Same as INA except that the B-register is used instead of the A-register.

Indirect Address Indexed by the C-Register

INC

Same as INA except that the C-register is used instead of the A-register.

C. TABLES

Communications processing usually requires extensive use of tables. Therefore, the DATANET-30 Assembler has several features for minimizing the work of constructing tables.

Tables may take any of several forms, depending upon the job to be done. Most tables are two-dimensional, that is, with rows and columns. In a computer's memory, the table may be either column-oriented or row-oriented. In a column-oriented table, all elements of a column are adjacent to each other (different columns may or may not be adjacent to each other). In a row-oriented table, all elements of a row are adjacent to each other in memory. Most files are row-oriented tables that is, each record is a "row" containing all items for one unit, such as all data for one employee in a payroll record. For large tables kept in memory, on the other hand, there is frequently an advantage to column-oriented tables. Or tables may really be a mixture of column-oriented and row-oriented. For example, in DATANET-30 Communication processing,

there are frequently advantages in having all elements in a column adjacent to each other. At the same time, one will likely have to pack several items, that is, several columns, of information for a unit into one word. Thus, the table becomes a compromise between column-oriented and row-oriented.

Repetitive Tables

Tables may also be classified as either repetitive, with every row in a column containing the same value as every other row, or nonrepetitive. One normally thinks of nonrepetitive tables, for if every item were the same, there would hardly be need for a table. But in communication processing, many tables are really status indicators, whose values are changed by the program many times during a working day, or even several times per second. When the programmer writes the program, he sets each row to its initial condition, which typically will be the same for all rows.

With the DATANET-30 Assembler, one may construct a repetitive table of any size with just one card. To set up a repetitive table, the programmer fills out the card with the appropriate operation code or pseudo-operation, and operand. Then, in the Repeat field, the programmer indicates the total number of words to be generated in the table. The Repeat field may be either numeric (up to 99) or symbolic. Symbolic repeat symbols are limited, of course, to two characters because that is the size of the Repeat field. If the Repeat field is symbolic, the symbol must have been previously defined by an EQU or EQO card equating the symbol to a numeric value. Failure to properly predefine a symbol used in the Repeat field will usually result in an "unpatchable" error condition, for the assembler will not repeat the card if the symbol was not previously defined, and the next card will assemble at the next memory location, leaving no space to patch in the unassembled table. Nevertheless, use of a symbolic Repeat field is encouraged because frequently the size of several repetitive tables will be determined by the number of Teletype lines in the system, and if one uses symbols, he need only change the EQU card defining the symbol, and reassemble, in order to change the number of lines. To generate a table greater than ninety-nine words long, with one card, one must use a symbolic repeat.

If a Reference Symbol is used on a card with a Repeat field, the symbol will assume the address of the first word in the table. Below is an example of how a channel table for a five-line system, on addresses 1-5, would assemble:

LOC	INST.	SEQUENCE NO.	REPEAT	SYMBOL	OP CODE	OPERAND	REMARKS
					ORG	512	
01000	000777		6	$SW1	OCT	777	
01001	000777						
01002	000777						
01003	000777						
01004	000777						
01005	000777						

Nonrepetitive Tables

For our purposes here, nonrepetitive tables are further divided into unpacked and packed tables.

Unpacked

Unpacked tables (containing only one item per word) may be entered one item per card, using the appropriate operation code or pseudo-operation and an operand. However, if all items in a table (or section of a table) are to be constructed by the same Op Code (as they usually are), one may enter several rows of the table on one card. To do this, the programmer should place an M in the Modifier field and then list the operand values, one after another but separated by slashes, in the Operand field. Obviously, one cannot use symbols containing slashes on one of these cards.

As an example, to construct a code-conversion table (called CONVERT) using the OCT pseudo-operation where the first ten values of the table are 17, 12, 33, 4, 67, 15, 23, 42, 6, and 35, enter the first ten rows of the table as follows:

```
CONVERT   MOCT   17/12/33/4/67/15/23/42/6/35
```

Or, construct a jump-table consisting of branches to several subroutines:

```
JUMP   MBRU   ERROR/TYPEA/NORMAL/ERROR6/NORMAL
```

Putting several operands on one card does complicate changing a source deck, but it has particular value when slow peripherals, such as Teletype, must be used for assembling, because it reduces the volume of input records.

Packed Tables

Nonrepetitive packed tables (containing more than one item per word) must be entered one word per card, unless the programmer wants to do his own packing. Most packed tables are so tedious to construct that the programmer will use one of the techniques described below, even though the techniques allow entering only one word per card.

Of the two techniques described below, only the first will have wide-spread application in communication processing. Each of the techniques requires using two pseudo-operations in relation to each other.

Field Definition

FDN

This pseudo-operation is used in conjunction with the MFC pseudo-operation. See MFC below.

Multifield Constant

MFC

Permits packing several fields into one word of memory. Used together with the FDN card.

The FDN is a DIRECTIVE pseudo-operation, but it is explained here because of its relationship to the MFC. In a source program, the MFC must must always be preceded by the related FDN cards, but the MFC is explained first because it shows the need for the FDN.

In message switching, twenty or thirty parameters are generally needed for every Teletype station, and there might be several hundred stations in a network. Because memory is limited, one cannot afford to use a separate memory cell for every parameter for every station. Many of the parameters require only a few bits each, so one usually packs several parameters for one station into one word. The work of constructing large tables with several parameters in each word is tedious enough even if all parameters are in the same kind of code (that is, all octal, or all five-level Teletype). Frequently, however, the various parameters are in different codes. One way to construct the tables is to manually convert the codes to their binary representations, manually "shift" each parameter into its place, manually OR the parameter into a word, and when done, convert the word to its octal representation so that it may be entered into the program by the OCT pseudo-operation.

The MFC pseudo-operation does the converting, shifting, and ORing. The programmer need only list the parameters and the assembler will generate the constant.

The assembler must know the following items about <u>each</u> parameter:

1. What bit positions does the parameter occupy?

2. In what code should the parameter be stored? For example, if a parameter is "12," the assembler must know whether this is a decimal number to be translated to its binary equivalent, or an octal number, or an alphanumeric field to be retained in its BCD representation, or an alphanumeric field to be stored in one of the Teletype codes.

Normally, every word in any one table will contain the same type of information in the same bit positions in every word. For example, a table of 120 words might contain a line number in bits 18 through 14 of every word, and every line number would be expressed as a decimal number on the MFC card. Obviously, it should not be necessary to define on each of the 120 cards what bit positions are used by line number, or in what code the line number will be expressed. Rather, these items could be defined just once at the front of the table, and the assembler would remember how to process line numbers. It is the function of the FDN card to tell the assembler how to interpret data on following MFC cards, and where to put the data.

One FDN card describes only one of the parameters to be placed into the table. Since the purpose of the MFC card is to permit packing several fields into one word, one would expect several FDN cards to precede a table. There must, then, be some way to specify which FDN card defines which field.

To facilitate this, the fields are numbered from 1-31 inclusive. Each FDN card contains its field number, which must be written in the Repeat field (hence, the FDN card cannot be repeated). Columns 10-12 must contain one of the literals (for instance, DEC, OCT, ALF, T5L) given at the end of the MFC description below, to tell how the field should be interpreted. The operand contains the number of the leftmost bit position to be occupied by the field, followed by a comma or a space (or a comma and a space), followed by the rightmost bit position.

For example, a table consisting of one word per station is to contain four parameters for each station, as follows:

1. Line number, bit positions 18-14, to be expressed as a decimal number.
2. Call-directing code, bit positions 12-7, to be stored as five-level Teletype, left-justified.
3. A BCD code, in bit positions 6-1, to be used for communicating with an on-line computer.
4. A code, in bit 13, to indicate whether station is on a full-duplex or half-duplex line. This will be indicated symbolically where the symbols have been previously defined by a CDN, EQU, or EQO.

The FDN cards would be:

```
4   SYM    FDN    13,13    LINE TYPE
1   DEC    FDN    18,14    LINE NUMBER
2   T5L    FDN    12,7     CDC
3   ALF    FDN    6,1      COMPUTER ID CODE
```

The table can now be constructed by means of the MFC pseudo-operation. It is only necessary to list the parameters in the Operand field of the MFC card. The parameters are separated from each other by commas (and up to eight blanks, if desired). The last parameter is followed by a blank rather than a comma. The parameters must follow a specific sequence, however. Line number, because it was defined as field #1, must be the first parameter on the MFC card; call-directing code must be next, etc. The table might appear as follows:

```
MFC    17,C,K,FULL        CHICAGO
MFC    3,Y,S,FULL         NEW YORK
MFC    21,Z,A,HALF        PHOENIX
MFC    9,F,X,FULL         SAN FRANCISCO
MFC    11,P,X,FULL        PITTSBURGH
MFC    12,P,H,HALF        PRESCOTT
etc.
```

When finished with a table, any or all field numbers may be redefined for use in another table. At times, however, it may be desirable to save one set of field definitions while another set is used. For example, in a row-oriented table, the programmer might want to use several sets of field definitions, one after another, on a repetitive basis. To extend the example used above, suppose that each station required another word, containing three parameters, e.g., transmitter start code; line number for alternate routing, and CDC for alternate routing. Instead of putting this control word into a separate column, the programmer may desire to put both words for each station together. Thus, two sets of field definitions are needed. The second set can be defined by simply using other field numbers, as shown below.

```
*   FIELD DEFINITIONS FOR FIRST WORD, EACH STATION
*
    1   DEC    FDN    18,14      LINE NUMBER
    2   T5L    FDN    12,7       CDC, PRIMARY ROUTING
    3   ALF    FDN    6,1        COMPUTER ID CODE
    4   SYM    FDN    13,13      LINE TYPE
*   FIELD DEFINITIONS FOR SECOND WORD, EACH STATION
*
*
    5   T5L    FDN    12,7       TRANSMITTER STOP CODE
    6   DEC    FDN    5,1        LINE NUMBER, ALTERNATE ROUTE
    7   T5L    FDN    18,13      CDC, ALTERNATE ROUTE
```

The programmer may now use either set of field definitions as desired. Obviously, some way is needed to specify on each MFC card which set of field definitions to use. If the set to be used starts with field 1, no indication is needed; the assembler will automatically start with field 1. If the set to be used starts with some other field number, the programmer must place the number of the first field in the set into the Repeat field (MFC cards cannot be repeated). Below is an extension of the example given above, in which every station has two words adjacent to each other in the table:

```
      MFC    17,C,K,FULL        CHICAGO
5     MFC    L,3,U
      MFC    3,Y,S,FULL         NEW YORK
5     MFC    B,13,T
```

Note that there was nothing in the FDN cards which told the assembler that fields 1-4 comprise one set and fields 5-7 another. It is the number of parameters on the MFC card which determines the size of this set.

The programmer may err and provide a field too big to fit into the number of bits specified on the FDN cards. In the example above, only five bits were provided for line number. If the programmer calls for a line number greater than 31 (largest possible in five bits) the assembler will chop off all high-order bits in the oversize number and will flag the word as an error.

Commas and spaces are used on the MFC card to separate fields. Therefore, these two characters cannot be used as literals in alphanumeric or Teletype fields. Plus or minus signs, on the other hand, may be used as literals in the alphanumeric or Teletype fields, but if they are used in numeric (DEC or OCT) or symbolic fields, they will be treated algebraically.

Below is the list of parameter types that are available in the initial release of the assembler. Provision for eight-level and other codes will be added to the assembler after more research has been done to define the best ways to handle such codes internally.

DEC The parameter must be written as a decimal number. It will be converted to its binary equivalent.

OCT The parameter must be written as an octal number.

SYM The parameter may be symbolic, decimal, or a sum-and-difference expression of several symbols and/or decimal numbers.

ALF The characters in the parameter will be stored "as is," that is, in their BCD representation.

T5L Each character is converted to its five-level Teletype equivalent and stored left-justified, that is, with a start bit for a total of six bits.

T5R Same as T5L, but right-justified, without start bit.

Condition

CDN

Sets up the conditions used by the CDC pseudo-operation. Also used as an 18-bit EQO.

Conditional Constant

CDC

Primary intent is to construct words from conditions on the CDN cards. The CDC operand can be an octal number and/or a symbol. The elements are ORed together.

CDN and CDC are used to construct a special kind of packed table: a table in which each item can take one of only a few conditions, perhaps only two conditions. In such cases, each condition may be given a name. The CDN (Condition) is used to describe what bit combination is represented by the name. The CDN is a directive pseudo-operation; it does not generate any data, but it tells the assembler what bits to turn on when the name is called for later in some other pseudo-operation. The name on the CDN card is entered into the assembler symbol-table. Opposite the name in the symbol table is the bit configuration described by the programmer. In function the CDN card is just an 18-bit EQO card; but while the EQO card can have symbolic operands, the CDN can have only octal numeric operands. The CDN must have a Reference Symbol.

CDC, mate to the CDN, is used to generate constants, or words in a table. The operand can be any symbol or octal number. The CDC is very similar to the OCT pseudo-operation, but in CDC, the values in the operand are ORed together. In OCT, the values in the operand are added together.

The use of the CDN and CDC is illustrated below by showing how they were used in writing the DATANET-30 Assembler. The assembler is basically an interpretive system. Each Op Code has several words which describe just how that Op Code should be processed. The construction of just one of those control tables is described below.

Bit 18 ON tells the assembler that for this Op Code the operand should either be ignored (for instance, the EJT card) or requires some special handling (such as shift commands). Final determination depends on bits in other words. We might call this condition NOT/STD for "not standard." Bit 17 ON says that octal numeric operands are permitted; bit 16 ON says decimal numeric operands are permitted; bit 13 ON says symbolic operands are permitted; bit 9 ON says sum-and-difference expressions are permitted as operands; bit 8 ON says the command must be placed at an even location (e.g., DDC); bit 7 ON says operand address must be even (LDD, BRS, etc.); bits 1 and 2 ON tell how many words the Op Code generates in the object program. This is only a partial list, for every bit in the word is used, but it suffices to illustrate the use of CDN and CDC.

We now give names to each of the conditions, and then generate the tables:

```
NOT/STD     CDN     400000      BIT 18, NOT STANDARD OPERAND
OCT         CDN     200000      BIT 17, OCTAL OPERAND OK
DEC         CDN     100000      BIT 16, DECIMAL OPERAND OK
SYMBOLIC    CDN     10000       BIT 13, SYMBOLIC OPERAND OK
SUMDIF      CDN     400         BIT 9, SUM-AND-DIFFERENCE
DL          CDN     200         BIT 8, PLACE AT EVEN LOCATION
DR          CDN     100         BIT 7, EVEN OPERAND REQUIRED

(LDB)       CDC     DEC+SYMBOLIC+SUMDIF+1
(DDC)       CDC     DEC+SYMBOLIC+SUMDIF+DL+2
(OCT)       CDC     OCT+SYMBOLIC+SUMDIF±1
(EQU)       CDC     DEC+SYMBOLIC+SUMDIF
```

It turned out that certain combinations turned up very often. For example, most machine commands that use memory addresses have the same conditions. Rather than repeat DEC+SYMBOLIC+SUMDIF+1 over and over, this bit combination was lumped together under still another name to reduce the repetition.

Note that the CDN is not restricted to use with YES/NO (one-bit) conditions. For example, assume that the right-hand three bits are to be used to indicate color. CDN could be used as follows:

```
RED         CDN     0
GREEN       CDN     1
BLUE        CDN     2
YELLOW      CDN     3
PURPLE      CDN     4
BLACK       CDN     5
```

It is not really necessary to define a condition which represents zeros in the field, but one may wish to do so for purposes of documentation. In the above example, if no color were specified, the result would be RED, so it is not necessary to define RED; but calling for RED in an operand gives more positive documentation.

D. MESSAGES

Messages may be entered into the program in either BCD or a Teletype code, depending upon the pseudo-operation. Whatever the code selected, the format of the message is the same. The message must start at the beginning of the Operand field (column 23), and may continue as far as necessary, through and including column 79. This provides for 57 columns, or 19 DATANET-30 words of message on one card. The Repeat field must tell the assembler how many DATANET-30 words are in the message. If the Repeat field is blank, the assembler will generate only one word (three characters) of message.

Alphanumeric

ALF

The message is generated in BCD code.

Negative Alphanumeric

NAL

Same as ALF, except that each word is two's complemented. This might be used for generating constants on tables where an ADD command will be used for a three-way compare.

Teletype Left-Justified

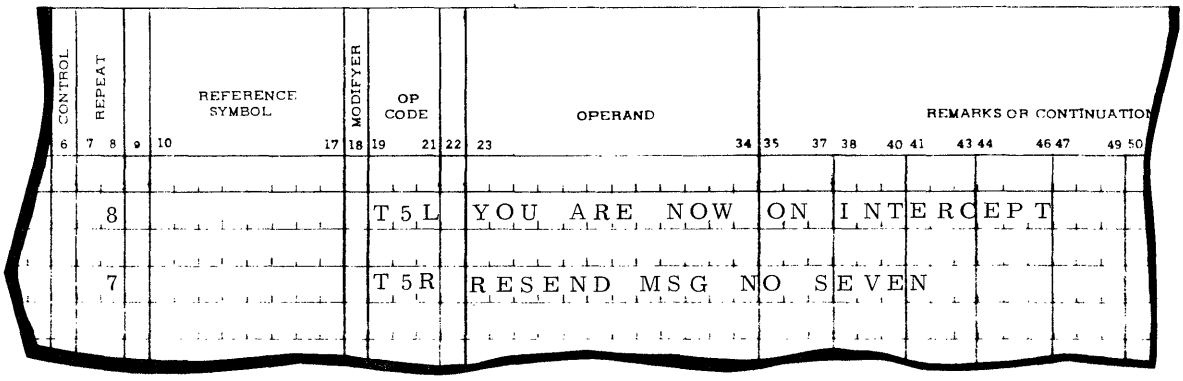
T5L

Teletype, five-level code, left-justified (with start bit). This operation code stores three characters per word.

Teletype Right-Justified

T5R

Teletype, five-level code, right-justified (no start bit). This operation code stores three characters per word, but only five bits per character. The three high-order bits of the word are zero.



5. DIRECTIVE PSEUDO-OPERATIONS

A. MEMORY ALLOCATION

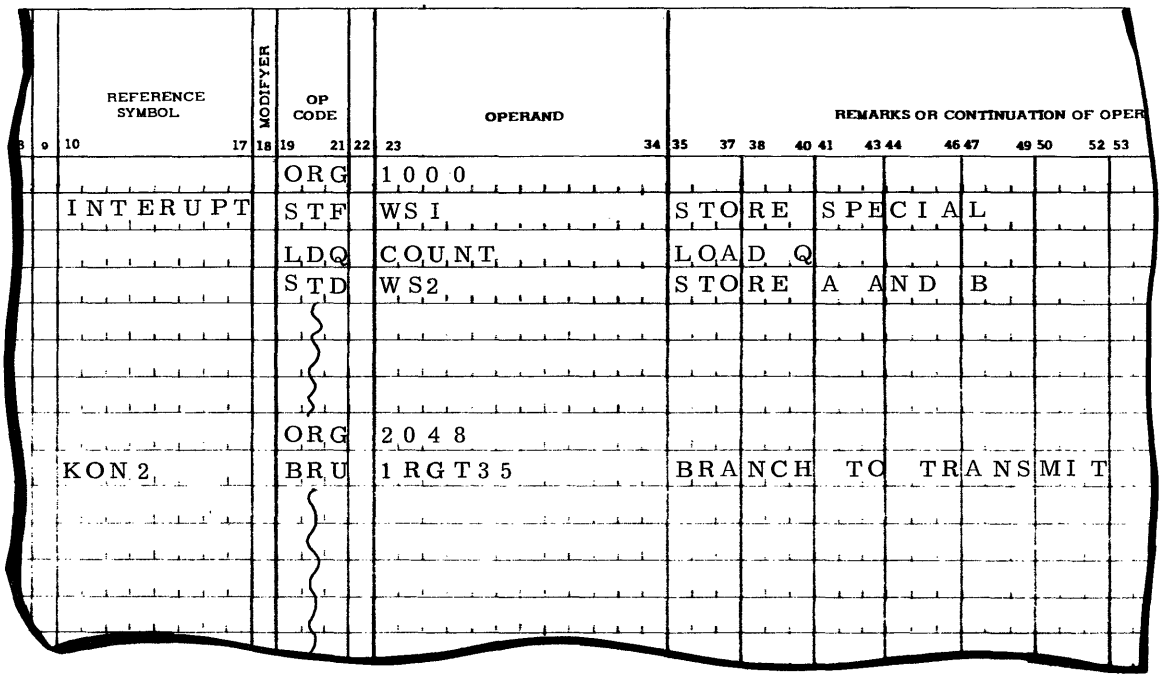
Origin

ORG

Establishes the starting location in memory of the program. The assembly program begins assembly of the object program as specified by ORG. One ORG card is required at the beginning of each assembly run. If no ORG card is included, the assembly of the program automatically begins at location 0000. Any number of ORG cards may be used in one assembly. The number following ORG must be in decimal.

This pseudo-instruction controls the memory assignments performed by the General Assembly Program. When an ORG instruction is encountered the assembly program uses the contents of the Operand field to reset an internal counter in the assembly program referred to as the memory allocation register (MAR). Normally, the MAR is increased by 1 for each instruction encountered.

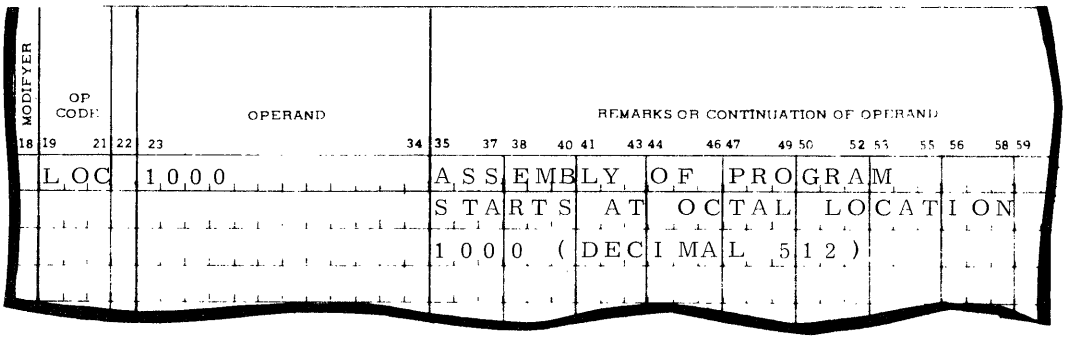
If the operand is a decimal, it is converted to binary by the program before being used. If the operand is symbolic, the symbol(s) must be predefined before being used. A symbol is defined by placing its name in the Symbol field (columns 10-17) once, and only once, in a given program. The General Assembly Program ignores all but the Operand field on an ORG instruction.



Location In Octal

LOC

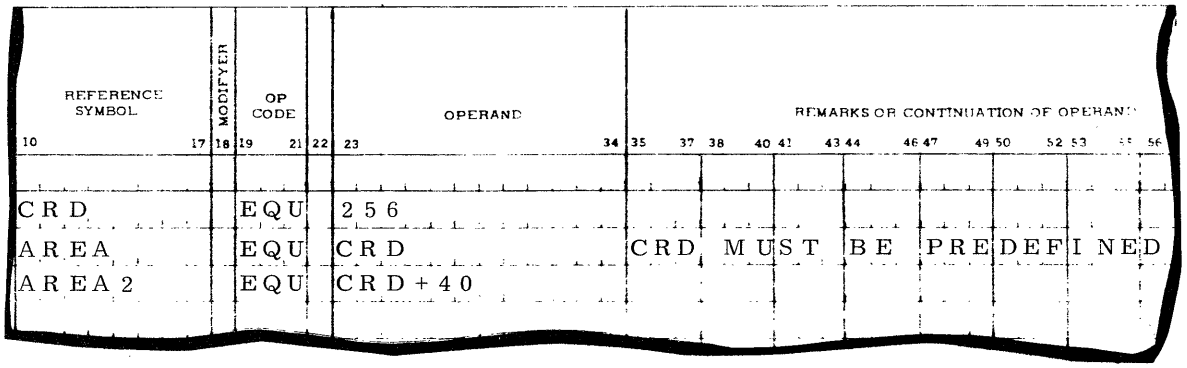
This operation performs the same functions as an ORG; however, the contents of the Operand field must be an octal number or symbolic. The assembly program will ignore leading zeros.



Equals

EQU

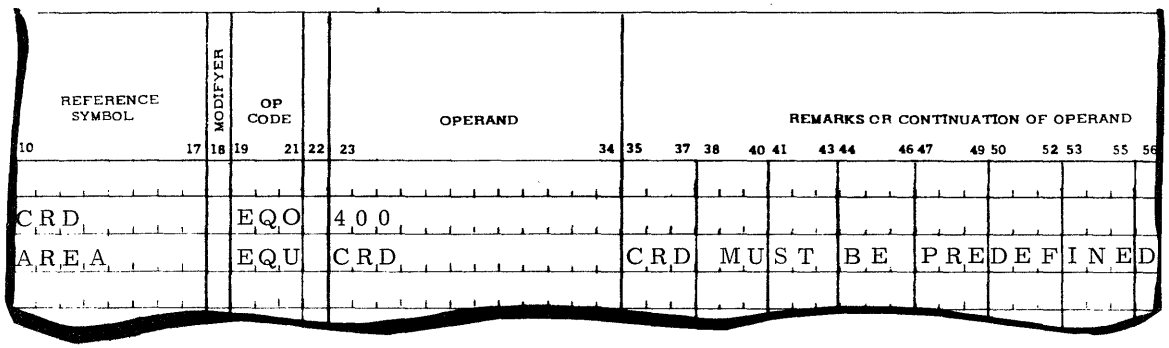
This instruction equates a new symbol to some memory location already known to the assembly program. The operand (decimal or symbolic) indicates the specific memory location to be used. This instruction does not affect the memory allocation register; thus, it may be used as often as necessary, and at any point within the source or symbolic program, without disturbing the memory assignment sequence. If the operand is symbolic, the symbol must be predefined. A decimal operand is converted to binary before being utilized.



Equals Octal

EQO

The EQO instruction is the same as the EQU except that the Operand field must be in octal form or symbolic. Leading zeros in the Operand field are ignored.

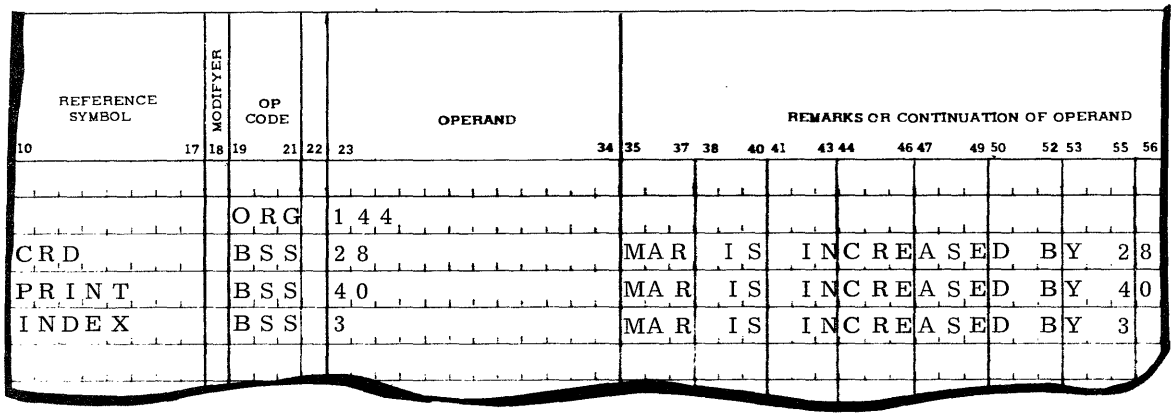


REFERENCE SYMBOL (10–17)	MODIFYER (18)	OP CODE (19–21)	(22)	OPERAND (23–34)	REMARKS OR CONTINUATION OF OPERAND (35–56)
CRD		EQO		400	
AREA		EQU		CRD	CRD MUST BE PREDEFINED

Block Started by Symbol

BSS

A BSS causes the assembly program to increase the memory allocation register (MAR) by the number in the Operand field. This instruction is used to reserve a block of memory locations in the object program. The Operand field may be decimal or symbolic. If symbolic, the symbol used must be predefined. If decimal, the operand is converted to binary by the assembly program before use. The BSS can be used as often as needed.



REFERENCE SYMBOL (10–17)	MODIFYER (18)	OP CODE (19–21)	(22)	OPERAND (23–34)	REMARKS OR CONTINUATION OF OPERAND (35–56)
		ORG		144	
CRD		BSS		28	MAR IS INCREASED BY 28
PRINT		BSS		40	MAR IS INCREASED BY 40
INDEX		BSS		3	MAR IS INCREASED BY 3

The BSS instruction of line 2 of the example will reserve 28 consecutive memory locations starting at location 144. The other BSS commands reserve additional blocks of memory.

A negative decimal operand can be used to reduce the MAR, in effect equating a block of memory to another block already defined.

Symbol Control

The pseudo-operations in this group must be used before the associated symbols are defined, or the pseudo-operation will be ignored.

Force Symbol to Even Location

EVN

The symbol named in the Operand field will be forced to an even location.

Force Symbol to Odd Location

ODD

The symbol named in the Operand field will be forced to an odd location.

Inhibit Assembler Movement of a Symbol

INH

Inhibit assembler's freedom to move a symbol. Usually, whenever the assembler finds a symbol in the operand of a double-length instruction (e.g., LDD, BRS) the assembler will force that symbol to the next available even location. The programmer does not always want that to happen; hence, this pseudo-operation.

B. CONTROL OF THE OBJECT PROGRAM DURING ASSEMBLY

DATANET-30 may work in any of several operating systems. Hence, the assembler makes no attempt to establish its own operating system, but rather attempts to provide the flexibility to enable the input/output packages to tie into any of several operating systems.

Object programs may be written in any one of three formats. "Transfer" records may be inserted at intermediate points in the program as well as at the end of the program. Although the assembler itself does not attach "identification" to object records, provisions are made to give input/output packages information they may need for identifying object programs.

Unless told to do otherwise, the assembler will build object programs in a Card Format, with an origin address, word-count, and hash total on each card. A similar format, but designed to fit a DSU record, is available when requested by pseudo-operation. Most production-type programs, however, will not be called in by loaders but will be read in large blocks directly from the DSU into the memory locations in which they will be executed. Therefore, the programmer may request by pseudo-operation a format which builds "program images" in DSU format.

Write Object Program on DSU

WOD

This causes the assembler to build the object program in 64-word records, with a record-origin, word-count, and hash total on each record. This format might also be used when programs are being stored on magnetic tape.

Write Object Program Absolute

WOA

The program is generated in 64-word records with no control words. Using this format requires the use of several other pseudo-operations, and can become quite complex. For detailed instructions, see Appendix D.

Punch Transfer Card

TCD

A TCD generates an instruction that will cause the loader to transfer control to the location specified by the Operand field. The operand may be decimal or symbolic. A TCD (transfer control data card) may be used as often as necessary in a source program, because this instruction does not affect the memory allocation register. A symbol in the operand must be predefined.

The transfer card is the last card of a segment of the object programs. When the program is loaded for execution, the transfer card directs the central processor to the location of the starting location of the program to be executed. The TCD cannot be used in place of an END instruction at the end of a source program. The assembler will look for additional source program cards following a TCD card.

End of Program

END

Causes the assembly program to generate an instruction that transfers control to the location specified in the Operand field when the object program is executed. The operand may be decimal or symbolic. If decimal, the operand is converted to binary. If symbolic, the symbol must be predefined. In addition, the END operation signifies end-of-program and terminates assembly. This operation may be used only once and must be the last instruction of the source program. If no END operation is used, an error comment will result but assembly will be terminated by the end-of-deck condition. The X-field of an END operation is not used by the assembly program.

The TCD instruction previously described should be used where a transfer control card is to be generated before the end of the assembly program.

The last card in a source program must be an END card or the assembler will halt.

REFERENCE SYMBOL	MODIFYER	OP CODE		OPERAND	REMARKS OR CONTINUATION OF O
10 17	18	19 21	22	23 34	35 37 38 40 41 43 44 46 47 49 50 52
START 1		LDA		FIRST	
		TCD		START 1	
START 2		LDB		SECOND	
		END		START 1	

Input/Output Control

IOC

This pseudo-operation does not cause any action in the assembler itself. The assembler turns control over to the Input/Output Package. The assembler makes the IOC card available to the I/O PAC so that the I/O PAC may extract any parameters it may need. The entire IOC card, except columns 1-6 and 19-21 are available for parameters, and the format of the IOC card is left to the I/O PAC author. The IOC card might be used, for example, to mark a "Program Index" on a disc-oriented system (see Appendix D). Or it might be used to give identification data to be attached to each record by an I/O PAC.

C. USE OF THE DSU DURING ASSEMBLY

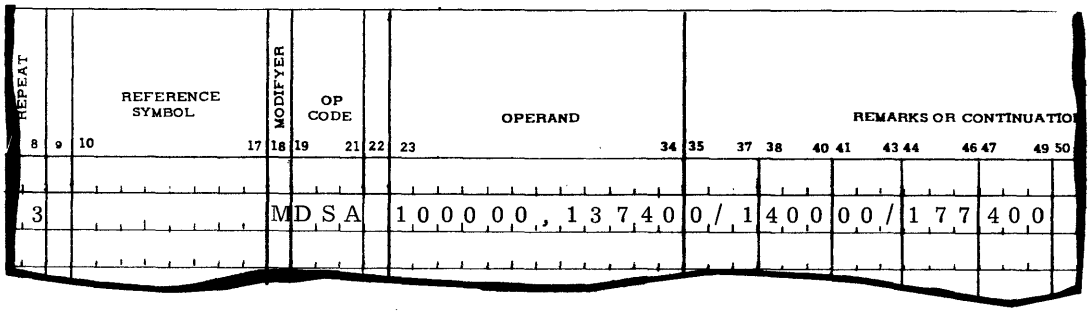
Disc Storage Addresses

DSA

In many installations, the disc storage unit (DSU) will be used for one or more of the files required by the assembler, and frequently the disc addresses to be used will vary from one assembly to the next (see Appendix C). The DSA pseudo-operation enables the programmer to specify what addresses should be used.

Each file used by the assembler, such as source program or object listing, is assigned a number (see Appendix C). On the DSA card, the programmer must specify, by number, for which file he is giving DSU addresses. The file number must be placed in the Repeat field (DSA cards cannot be repeated). In the Operand field, the programmer specifies a range of DSU addresses by giving a starting address and an ending address. The two addresses are separated from each other by a comma and/or one or more blanks. Addresses may be either octal or symbolic, but if symbolic, must be previously defined by a CDN command.

In some situations it will be necessary to define ranges for more than one file on the same card. This may be done if the files have adjacent numbers in the numbering system described in Appendix C. The Repeat field must contain the lowest file number in the group. The pairs of addresses must be separated from each other by slashes. For example, define DSU addresses for files 3 and 4:



Because of the M in the Modifier field, the assembler will continue analyzing pairs of DSU addresses until it finds a pair not terminated by a slash. Each time it finds a slash, it adds one to the file number.

D. SEGMENTATION

Provision is made for two types of Segmentation problems; one is that of assembling several segments together; the other is that of assembling segments, which must run together as one program, at different times.

Two problems which frequently arise with assembling segments together are those of overlapping memory and using the same symbol more than once. There is nothing the assembler can do to prevent overlapping memory, but it will flag any ORG or LOC instructions which set the Memory Allocation Register back, that is, reset it to a lower value.

Prefix

PFX

Used to make all symbols within a program segment unique from symbols used elsewhere. The PFX does not guarantee uniqueness, but it reduces the probability of duplicate symbols. Place some character in the first column of the Symbol field (column 10). The assembler will automatically attach this character to the front of all symbols, both in the Symbol field and in the Operand field, until another PFX card replaces this one, except under the following conditions:

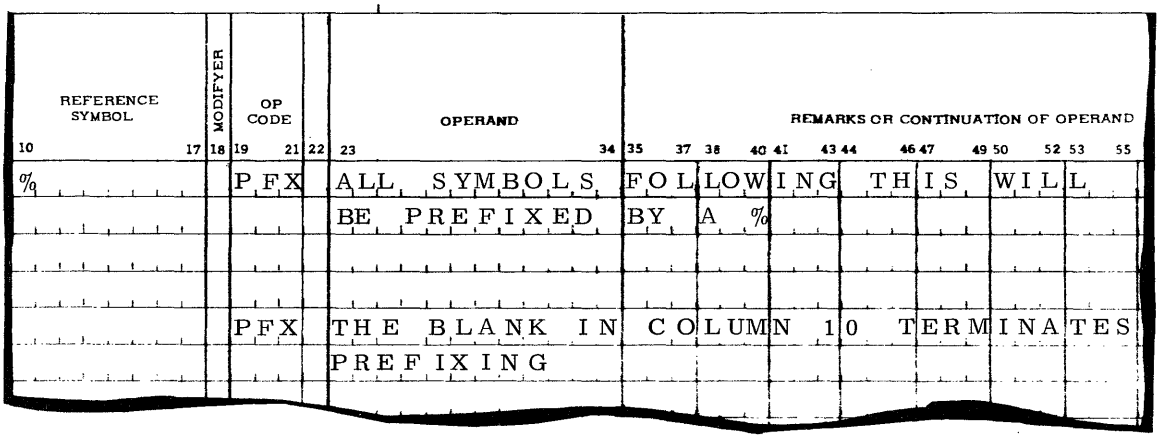
Symbols containing eight characters will not be prefixed

Redefinable symbols will not be prefixed

Symbols in the Symbol field will not be prefixed on any cards which contain an "S" in the Modifier field

Symbols in the Operand field will not be prefixed on any cards which contain an "O" in the Modifier field.

Prefixing may be started, stopped, or changed as often as desired. Prefixing will be stopped by any PFX card which contains either a zero or a blank in column 10.



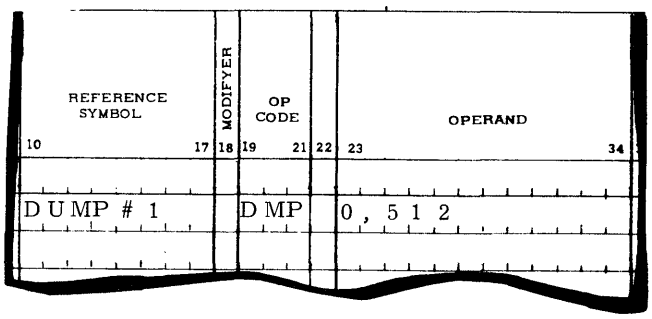
REFERENCE SYMBOL (10–17)	MODIFYER (18)	OP CODE (19–21)	OPERAND (22–34)	REMARKS OR CONTINUATION OF OPERAND (35–55)
%		PFX	ALL SYMBOLS	FOLLOWING THIS WILL
			BE PREFIXED	BY A %
		PFX	THE BLANK IN	COLUMN 10 TERMINATES
			PREFIXING	

Assembling segments separately has some of the same problems as assembling them together, but it has the additional problem of making symbols, which are referred to by different assemblies, fall in the same location in all assemblies. One can accomplish this by filling out EQO cards after the first assembly and adding the EQO cards to all subsequent assemblies. However, the assembler provides an easier way through two pseudo-operations, DMP, and LDS. The use of these pseudo-operations requires additional files.

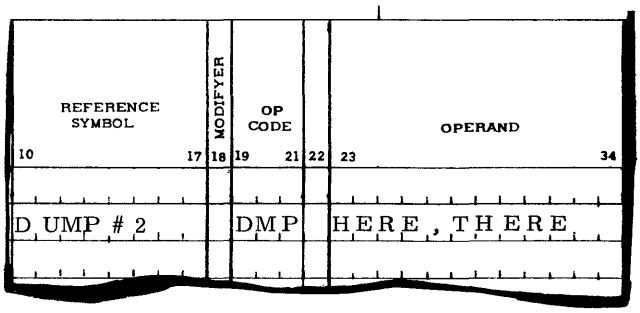
Dump Symbol Tables

DMP

This pseudo-operation requires two operands, which define a range of memory addresses. The assembler will dump every symbol, together with its address and control bits, between the two addresses indicated. For example, dump all symbols whose addresses are in the common data bank:



Or, dump all symbols whose addresses fall between the addresses assigned to the two symbols, HERE and THERE:



The addresses given are inclusive addresses. In the first example any symbol which has an address of either 0 or 512 or anything between will be dumped.

The first six columns of the Symbol field will be written onto each record of the symbol table file for identification.

Symbols defined by CDN cards will not be dumped. Prefixed symbols will be dumped with their prefixes.

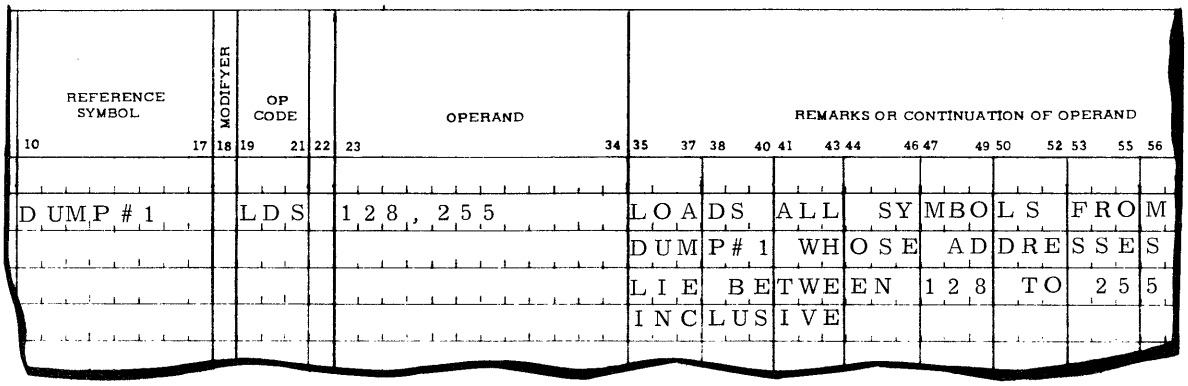
Load Symbol Tables

LDS

This pseudo-operation also requires two operands which define a range of addresses. The operands may be either numeric or symbolic, but if symbolic, they must have been previously defined. The first six columns should contain the identification which was written onto the symbol table file by the assembly which wrote the file (see DMP).

The LDS card causes the assembler to read the symbol table file. The assembler will accept only those records on the file whose identification matches that on the LDS card. The assembler will load into its symbol table all symbols whose addresses fall within the range specified.

If prefixing is in affect when the LDS card is executed, all symbols loaded from the symbol table file will be prefixed also. Normally, this is not desired, so the programmer should use caution.



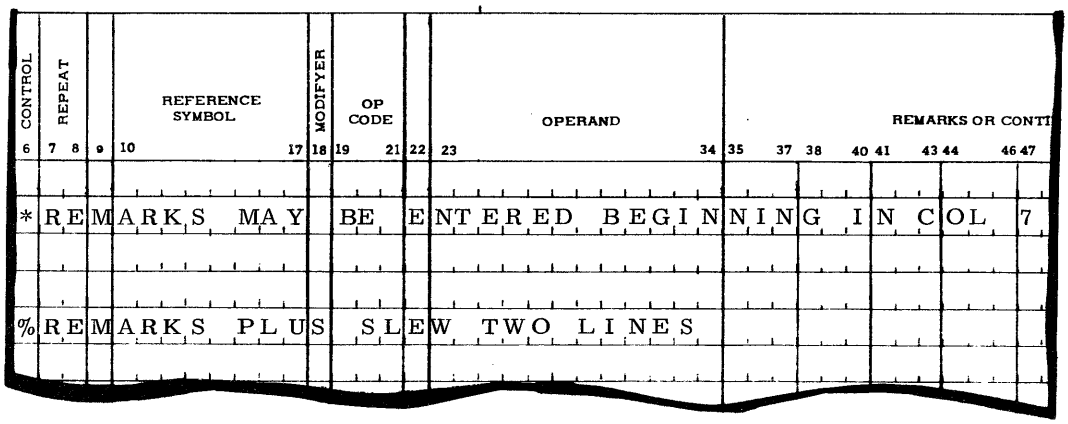
E. DOCUMENTATION

Asterisk (*)

An asterisk in the Control field (column 6) makes a Remark field out of the remainder of the card. Such cards will be printed on the object listing, but will cause no other assembler action.

Percent (%)

A percent (%) in the Control field yields a variation of the remark card. The assembler will slew two lines, print columns 7-80 (columns 1-6 will be blanked out by the assembler) and slew two more lines. Such cards will stand out on the listing.

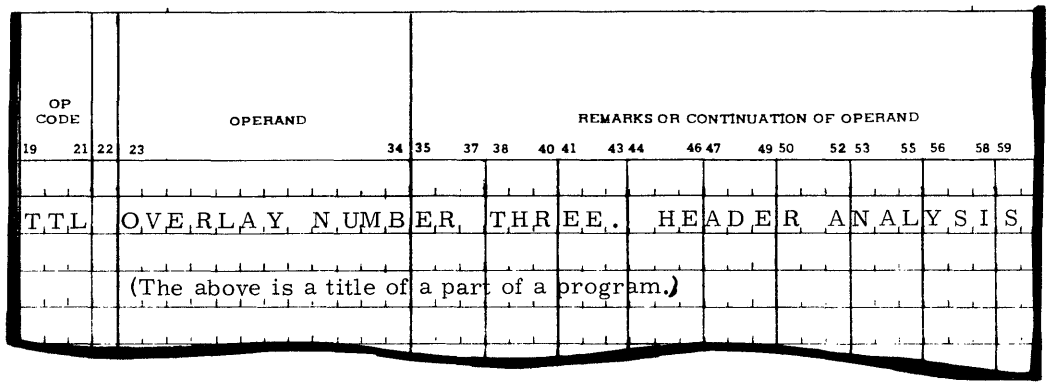


Title

TTL

Columns 23-80 of the title card will be printed at the top of each page hereafter until another title card appears. The first title card in the program will also be printed on all pages preceding the first card.

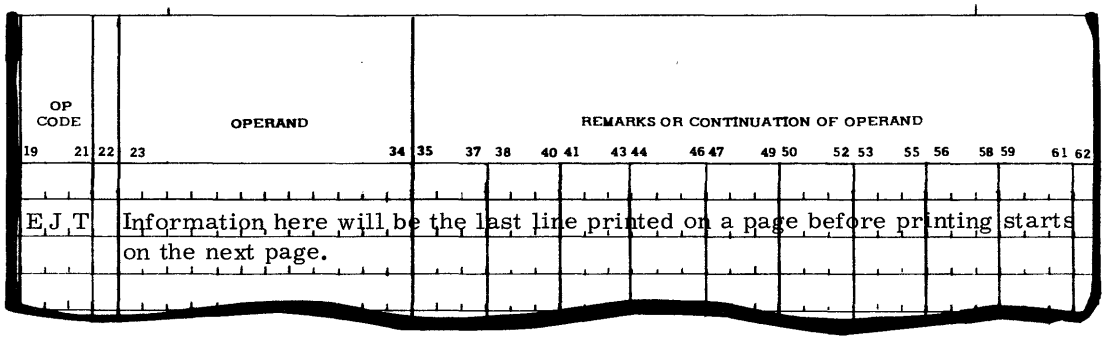
For example, a program contains two title cards, the first one appearing on page 5 and the second on page 12. The first title card will be printed on pages 1-12 and the second title card will be printed on pages 13 and following. Any number of title cards may be used.



Slew to Top of Page

EJT

Eject printer listing to top of page. The EJT card will be printed before the listing slews to top of page.



No List

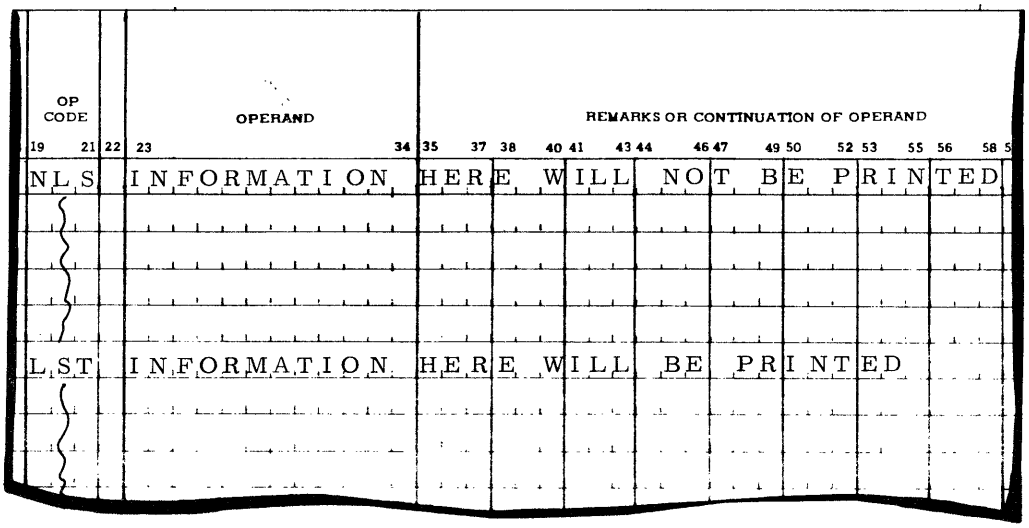
NLS

This card, and those following it, will not be printed on the object listing.

Resume Listing

LST

This card, and those following it, will be printed.

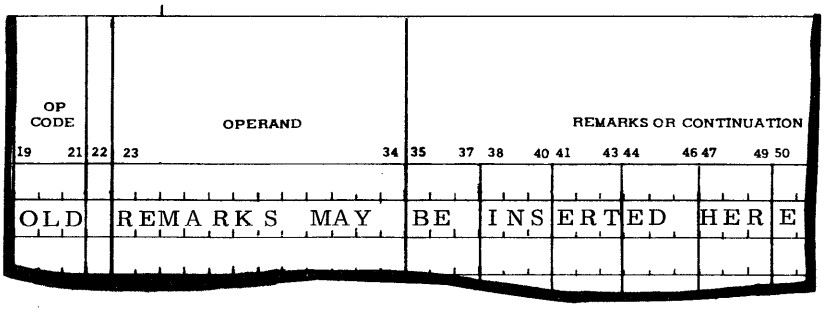


F. MISCELLANEOUS PSEUDO-OPERATIONS

Switch to Old Card Format

OLD

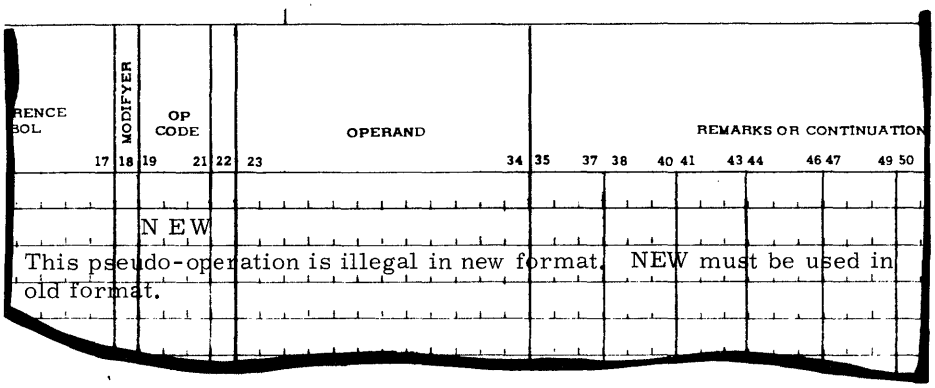
Must precede any cards written in the "old" format. (See Appendix B.)



Switch to New Card Format

NEW

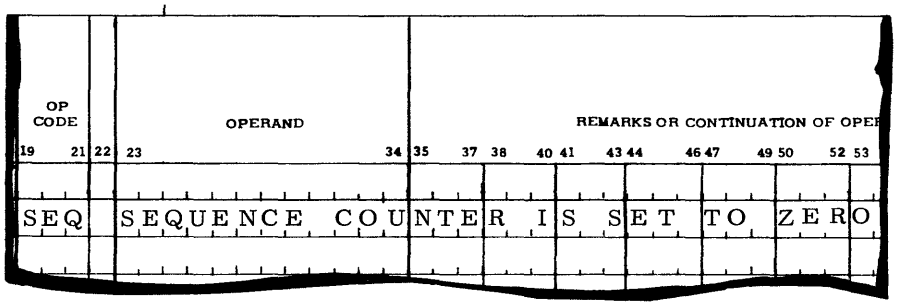
Must precede cards written in new format when some "old" format cards have been used ahead of the new. The assembly program assumes new format unless notified as "old." (See Appendix B.)



Reset Sequence Counter

SEQ

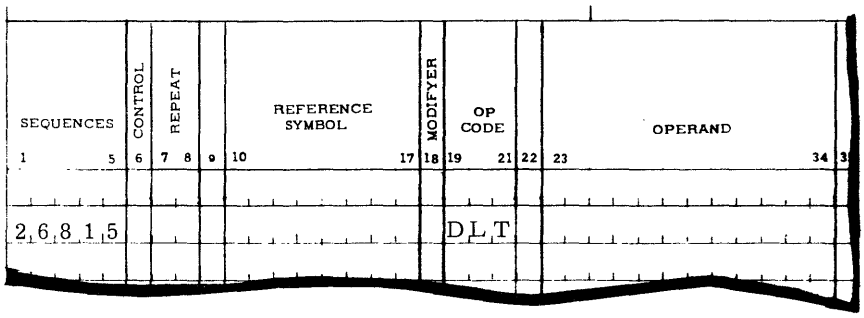
Reset the sequence counter to zero and check sequence of Source program. This pseudo-operation may be used as often as desired. Every time it is used, the sequence counter will be set back to zero. This is useful if each subroutine contains its own set of sequence numbers.



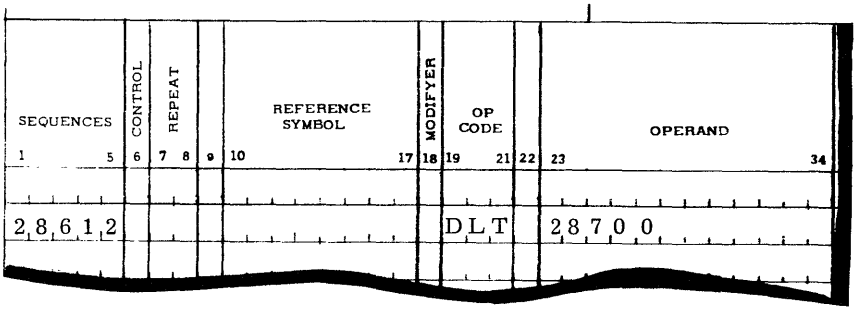
Delete

DLT

Delete cards from master file. The DLT pseudo-operation is used only on Update runs, and will be ignored if used in anything other than an Update. A DLT may delete either a single card or a block of cards from the master file. If only one card is to be deleted, the Operand field of the DLT card may be blank. If a block of cards is to be deleted, the Operand should contain the sequence number of the last card in the block. See "Updating Procedures."



Card Number 26815 will be deleted.



Cards 28612 to 28700, inclusive, will be deleted.

Subroutine Call

SBR

The assembler will call subroutines from a library file, provided, of course, that the computer used for assembling has the necessary peripheral device.

The programmer requests a subroutine by use of the SBR pseudo-operation. The Symbol field of the SBR card must contain the subroutine ID, left-justified. ID is limited to six characters. The ID on the SBR card must exactly match the ID on the library file, character-by-character, for the assembler does only a double-word compare, with no analysis of characters.

Normally, when the assembler encounters an SBR card, it adds the request to a list, but does not immediately call the subroutine. A maximum of fifteen subroutines may be requested at any one time. When the assembler reaches the END card of the source deck, it will call all requested subroutines which have not yet been called, and assemble them at the end of the program.

Occasionally, it may be desirable to call subroutines before the end of the program. To do this, the programmer places an I in the Modifier field (column 18) of the SBR card. The assembler will immediately call all subroutines in the Request table. This clears the Request table so the programmer may request up to ten more subroutines. One may request "immediate" call of subroutines as often as desired, but he should not do so indiscriminately, because the assembler must pass the library file from the beginning every time a request for immediate call is given.

When subroutines are called, the assembler counts how many requested subroutines were found. If, at end-of-file of the library file, not all subroutines have been found, the assembler will print a warning on the listing but will not indicate which subroutines were missing.

Subroutines will be called in the sequence in which they appear on the library file, regardless of the sequence they are requested. If it is necessary to assemble the subroutines in a different sequence, the Immediate-Call option must be used to force calling the later subroutines first.

Once a subroutine is called, it becomes part of the work file, which may serve as input to a later assembly if an update run is needed. One would not want the subroutines to be called again. To prevent this, the assembler converts subroutine request cards to REMARKS cards. For this reason, one cannot <u>delete</u> a subroutine by merely deleting the request card. He must delete every card in the subroutine. (See block-delete feature in Updating Procedures.)

Library call is not available in the 4k assembler.

REFERENCE SYMBOL (10–17)	MODIFYER (18)	OP CODE (19–21)	OPERAND (23–34)	REMARKS OR CONTINUATION OF OPERAND (35–59)
D30.16		SBR	REMARKS MAY START HERE. SUBROUTINE	
			D30.16 IS ADDED TO THE REQUEST LIST	
D30.37	I	SBR	SUBROUTINE D30.37 IS ADDED TO THE	
			REQUEST LIST AND ALL SUBROUTINES IN	
			THE LIST WILL BE INSERTED INTO THE	
			PROGRAM AT THIS POINT IN THE ASSEMBLY	

6. CONTROL CARD

The control card is optional, but if used, it must be the first card in the source deck. If it appears later in the deck, the assembler will halt. The control card is identified by a C in column 6.

However, the I/O PAC may require that some other card, such as a DSA or IOC card, be the first card. Therefore, all control information required on the control card is placed into columns not used by the DSA or IOC. Thus, the control card information may be placed on the same card with other pseudo-operations (such as DSA or IOC). If a control card is not used, the assembler will assume that the standard option for assembly is selected. (See column 11.)

If column 10 contains a U, the assembler will update an old master file containing a source program. The control card must be the first card of the change deck. If column 10 contains an S, the assembler will also update, but as it updates, the assembler will resequence the new work file so that it may be used as input to a later assembly.

When a new program is assembled the first time, it may contain several sets of sequence numbers. For example, each subroutine may start with sequence number 0. If these sequence numbers were copied onto the work file, it would be very difficult to update the work file on a later assembly. Therefore, the programmer may instruct the assembler to resequence the source deck while assembling. To do this, he must place an R in column 10. If column 10 does not contain a U, S, or R, the assembler will accomplish a normal assembly.

Column 11 determines whether the standard or the optional assembly will be used. The standard is when column 11 is blank. The option is when column 11 contains a Y.

If column 11 is blank, the assembler will not read the file the second time, but the programmer must then flag all symbols to be forced even with a D in the Modifier field, or must name the symbol in the operand of the EVN pseudo-operation before defining the symbol, or must refer to the symbol in a double-length command before defining the symbol.

If column 11 contains Y, the assembler will automatically force all symbols named in the operand of "double-length" instructions to an even location. This forces the assembler to read the work file an extra time, and increases the assembly time.

Columns 12-17 are available for a date, which will be printed on the title line of every page. If a date is used, the assembler will insert a slash between the second and third characters and between the fourth and fifth characters. Thus, 070464 will become 07/04/64.

Although the assembler itself has no use for them, it will save columns 1-5 of the control card in predetermined locations. The thought behind this is that a customer might stack several source programs on one magnetic tape. He could use columns 1-5 of the control card to identify the program being assembled. However, the ability to stack several source programs on one tape will not be built into the assembler. If this feature is to be incorporated it will be the responsibility of the I/O PAC to copy all programs preceding and following the subject program onto the new work file. The I/O PAC must also attach the ID to each record, etc.

7. UPDATING PROCEDURES

Message switching programs are usually long and it would be time-consuming to reassemble each time from cards or from Teletype. If suitable peripherals (for instance, magnetic tape) are available, the assembler will update a previous assembly. Updating implies at least two inputs: the previous assembly, called old master, and the changes to the old master. The old master is the work file from the previous assembly. The changes are what normally would be the source file.

If an update run is to be made, the first card of the changes must be a control card (see description of control card). If the first card is not a control card (requesting an update) the assembler will assemble only the changes.

Updating a file usually permits at least three basic transactions:

1. Changing existing records
2. Deleting existing records from the file
3. Inserting new records into the file

The key for the update is the sequence number. Zero is not a valid sequence number on change cards.

Changing of existing records is accomplished by simple replacement. Thus, to change a card in the old master, the programmer should fill out a new card containing the sequence number of the card to be changed and containing the instruction as it should be assembled.

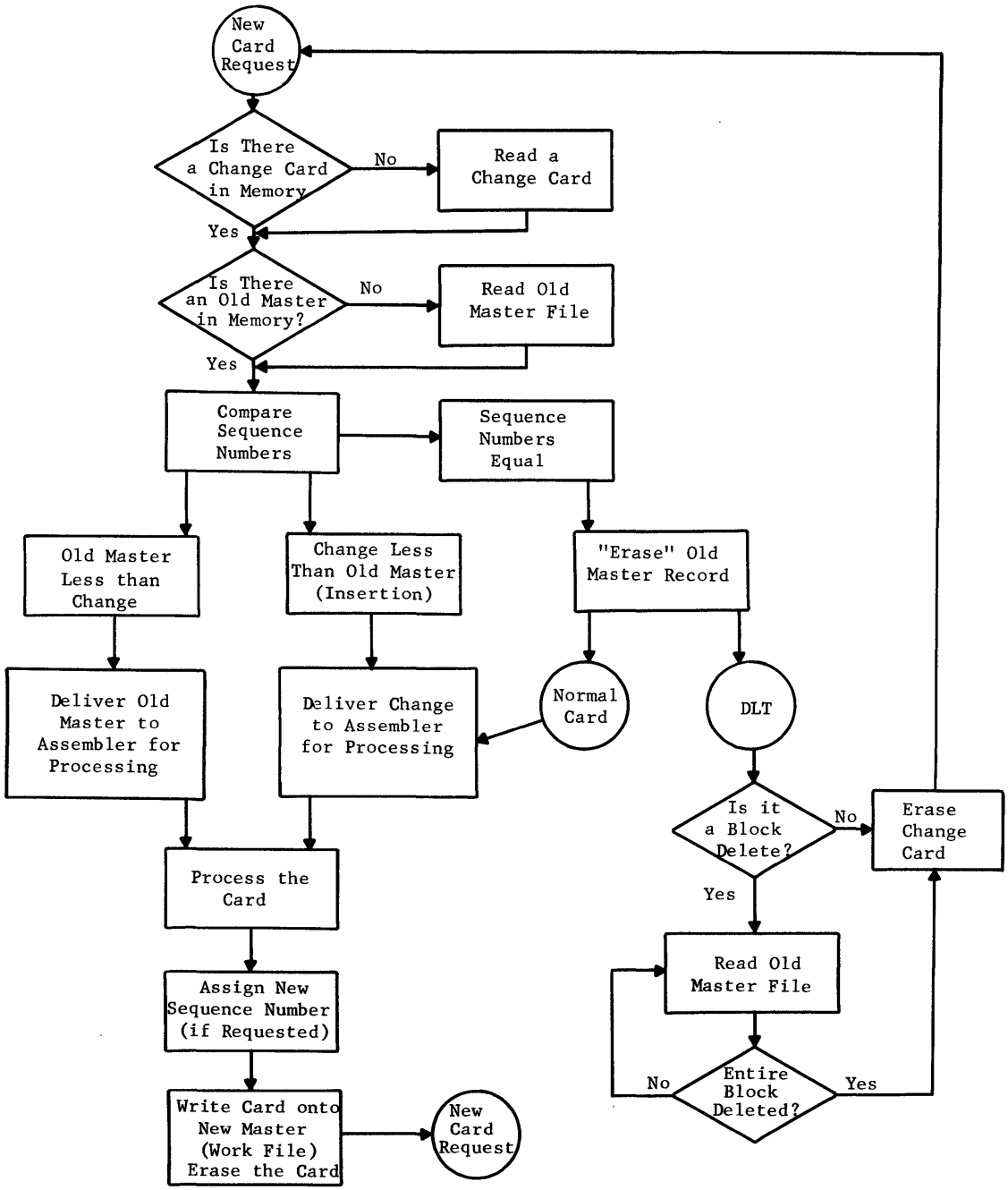
Deletion is handled by the pseudo-operation DLT. The DLT card should contain the sequence number of the card to be deleted. Sometimes it is necessary to delete entire blocks of cards. It is not necessary to fill out a DLT card for each card in the block to be deleted. Instead, the programmer should put the number of the first card in the block in the Sequence field, and the number of the last card in the block in the Operand field. (If only one card is to be deleted, the Operand field should be left blank.)

If new cards are to be inserted into a program, the insertion cards must contain sequence numbers greater than the last card before the insertion and less than the first card after the insertion. The assembler, when it assigns sequence numbers to the work file, assigns only numbers divisible by ten, so there is always opportunity to assign sequence numbers for insertion. This does not mean that one is limited to inserting nine cards between any two cards. One may have any number of change cards with the same sequence number. For example, if one wanted to insert fifty cards at one point, all fifty cards could have the same sequence number.

If one wishes to update an assembly containing both old and new card formats, he must observe certain conventions. See the appendix "Compatiblilty with Previous Format."

A simplified block-diagram of the update process is shown below. A study of this diagram may clear up questions on the procedures for updating.

The change deck must terminate with an END card. The old master will also have an END card. Of the two, the one with the higher sequence number will terminate the assembly. The one with the lower sequence number will be ignored, but must be present. It will not appear on the output listing.



8. USE OF THE CONSOLE FOR ASSEMBLY

CONSOLE SWITCHES

Console Switch 18. This is the general purpose toggle switch which will be used whenever an I/O PAC pauses to permit some kind of operator action. The I/O PAC will turn on the Error light and set up a code in the console lights (see below) to indicate why it paused. For example, the program might pause to permit the operator to change tape handlers, or to let him decide whether or not to ignore an input/output error. When the operator has taken the necessary action, he must turn switch 18 on and off. The program will turn off the Error light/buzzer and will continue.

Console Switch 17. If this switch is on, the assembler will delete all cards which contain a D (for delete) in the Control field (column 6). This option is provided to enable the programmer to delete debugging subroutines, and the calling sequences to them, during final assembly. It will be used mostly by installations which do not have the necessary peripherals for using the DATANET-30 Monitor and Debug system.

Console Switches 16-10 are used only when assembling with limited peripheral configurations. Many users will not be concerned with them at all. Console Switches 9-1 are available to I/O PAC's. For a discussion of the uses of switches 16-10, see Appendix C.

CONSOLE LIGHTS

If the assembler caused the stop, the C-register will contain 177_8. If the I/O PAC caused the stop, the C-register will contain the file numbers for which the stop was made.

If the stop was made because of an error condition, the A-register will contain 777000_8. If the stop is not for an error condition, the A-register will contain 000777_8. The B-register will contain a code indicating the specific reason for the stop. The B-register codes for peripheral conditions will depend upon the I/O PAC. The B-register codes used by the assembler are as follows:

Error Conditions (777000_8 in A-register)

000077. A control card appeared, but it was not the first card in the deck.

007700. A file assigned to the DSU has required more space on the DSA card, or no DSA card was provided for this file.

000777 in A-register "Normal" Conditions

777777. End-of-job.

Anything else: Assembler is pausing as requested by console switch (see Appendix C).

9. ERROR INDICATIONS ON OBJECT LISTING

ASSEMBLY ERRORS AND SUSPECTED ERRORS

The following codes listed are errors or suspected errors found during assembly by the General Assembly Program. The object is to convey as much error information as possible regarding errors in the object program.

Except for machine malfunctions, the computer will stop only under two circumstances during assembly:

1. The DSU addresses given by a DSA card have been exceeded.
2. A control card was inserted in the object program deck after the first card.

Error Codes

Following is a list of the error codes:

Code

O — Illegal Mnemonic Operation

This becomes a HLT (00).

U — Undefined Symbol

A symbol name appearing in the Operand field does not appear in the Symbol field of any instruction. Constant 0000 is inserted as an operand address.

M — Multiply Defined Symbol

Either the Symbol field or the Operand field contains a symbolic name which appears in the Symbol field of two different instruction lines. If the error detected was in the Symbol field, assembly will continue with the present setting of the memory allocation register. If the error detected was in the Operand field, the value assigned to the symbol the last time it appeared will be used as the operand address in the assembled instruction.

A — Error or Suspected Error in the Operand Address

Blank Operand field in a line normally requiring an address. An entry in the Operand field of a line which normally should be blank. The numeric value of the operand does not meet the requirement of the line in which it was used. The value of the operand address will be logically ORed into the instruction.

Code

T — Error or Suspected Error in X-Field

The X-field contains an entry in an instruction which does not access memory. The X-field contains any character other than X or is a numeric.

S — Size Error

An error occurred in the size of DEC, DDC, or OCT, or a number was too big to fit the indicated field size.

$ — Channel Table Usage

The $ character in the first position of a symbol indicates to DATANET-30 General Assembly Program that this is to be treated specially. This symbol must be assigned by the programmer to a memory location that is a multiple of 16_{10}. If this error tag appears, it means that either the specified address was not modulo 16 or less than 8192 or both.

E — Message Words Exceeded

Indicates that more words were specified on a message card than can fit on one card.

F — Symbol Table Full

Indicates that the symbol was omitted because the symbol table was full. This has the same effect as an undefined symbol.

C — Control Field Illegal

Indicates illegal use of the Control field, such as having a character not allowed in the Modifier field.

L — Illegal Symbol Field

Indicates a Symbol field error, such as the use of an illegal character or an all numeric symbol.

B — Backed Up Allocation Register

The assembler memory allocation register (MAR) was backed up (reset to a lower number) by an ORG or LOC instruction. This may not be an error but is flagged as a potential error.

Code	
Hyphen	Skipped Memory Location

The assembly program skipped a memory location because something was forced to an even (odd) location. This is not necessarily an error but is flagged as a potential error.

Slash	Assembler Error

Indicates an assembler error. If a slash is present on a large block of instructions on the listing, this probably means that the Input/Output Package lost a record on the work file. If it happens on isolated instructions, contact the DATANET-30 Programming Unit, Computer Department, Phoenix, Arizona.

Q	Source card out of sequence

APPENDIX A
INTRODUCTION TO SYMBOLIC PROGRAMMING

Before proceeding into the discussion of symbolic programming, it is necessary to define the following terms:

1. Machine instruction. This is the instruction actually used by the computer to perform an operation. It also means that the instruction is in a form the computer can actually use. For example, 400015 is the actual machine instruction for Load A with the contents of memory location 15.

2. Mnemonic instruction. This is the instruction notation used in writing programs to represent the actual machine instruction on the coding sheet. The mnemonic for an instruction is changed by the assembly program into an actual machine instruction form. For example, LDA (Load A) becomes 400000.

3. Source program. This is the program set down on the coding sheet. Each line of coding is then punched on a card.

4. Source program deck. The results obtained after punching all the instructions on a coding sheet onto punched cards and retained in the order as written on the coding sheet.

5. Object program. The result obtained after the source program has been processed by the assembly program to produce the instruction on the coding sheet into machine instruction form.

6. Pseudo-operations. A pseudo-instruction (operation) is an instruction used by the General Assembly Program when assembling the source program. The pseudo-operation has the same form as a computer instruction and is listed in the source program the same as a normal computer instruction. However, the pseudo-operations are never executed by the computer. Pseudo-operations are used to control the assembly process, generate constants and to annotate the object program listing.

7. Macro-Instructions. Macro-instructions are not actual machine instructions. The General Assembly Program will recognize the mnemonics for the macro-instructions and generate and insert in the object program the necessary series of instructions to do the specified operation.

8. Symbolic notation. The designation (NAME) given to a location in memory instead of the actual memory location.

A computer instruction must specify the location of data as well as the operation to be performed. In symbolic programming, the actual location of a word in memory is always referred to by a name or symbol. The symbol is usually chosen to have a meaning in relation to the purpose or use of the memory location. For example, if a number 4 is used as a constant, this could be referred to CONST4. When it is necessary to use this constant, the instruction

```
LDA     CONST4
```

would load the A-register with the contents of the memory location with the symbol, CONST4. The programmer does not need to know the actual memory location of this constant.

Thus, in the actual writing of a program, the use of symbols to designate memory locations and mnemonics to designate the operation, allows the programmer to list the desired operations in a language easily understood by the individual person.

APPENDIX B
COMPATIBILITY WITH OLD FORMAT

The old format is defined in the DATANET-30 Programming Manual, pages V-63 to V-65. The old format was used when programs were assembled on the GE-225 computer.

The DATANET-30 assembly program is designed to use the new format and will assume that the new format is being used. However, the old format may be used by preceding the old format deck with a card in the new format containing the pseudo-operation OLD. One can switch back to the new format by use of the pseudo-operation NEW, which would itself still be in the old format. One may switch back and forth between formats as often as desired, but he must always advise the assembler of the switch.

All pseudo-operations from the old assembler will also work in the new assembler, except as follows:

1. *ǂ* will print before it ejects rather than after.
2. Double decimal numbers which extend onto a second card will not assemble properly in the new assembler. Double decimal numbers exceeding 8 digits must be entered in the new format.
3. The assembler program will not do scaling of decimal or double numbers.
4. The Z-option has been replaced by the MIC instruction

Many pseudo-operations have been added to the new assembler. Although some of these will work in the old format, it should be assumed in general that they will not.

APPENDIX C
DIRECTIONS FOR CONSTRUCTING I/O PAC'S

A. INTRODUCTION

When one designs records without knowing the nature of the storage medium, he cannot hope to achieve full utilization of the storage capacity of the device to be used. For example, a binary card will hold forty DATANET-30 words. A 200-Series DSU record will hold 64 words. If the same data may be written onto either storage medium, at least one of the two media will be used inefficiently. Furthermore, placing the input/output functions onto a program which is not an integral part of the assembler will usually mean moving records more often than would otherwise be necessary, so the assembler is slightly slower than would be a more conventional assembler. Further limiting the speed of the assembler is the fact that there is not adequate memory for extensive blocking or buffering of records.

But the slight decreases in efficiency of speed and storage are a small price to pay for the flexibility and power available through the I/O PAC concept. Not only does the technique permit a wide variety of peripheral configurations, but it permits adapting to the operating systems of the various computers with which the DATANET-30 may be associated.

It was necessary to divide the work of record handling. Part of the work is assigned to the assembler and part of it to the I/O PAC. All work independent of the input/output device is given to the assembler. This includes formatting, packing, and unpacking. The I/O PAC should never have to modify or analyze a record except in those cases where code-conversions (such as from Teletype) are required or when space-suppression of other condensing are worth the effort.

The I/O PAC must perform all functions dependent on the peripheral device. These include not only the reading and writing of records, but all error checks and associated recovery procedures, end-of-tape and end-of-file checks, labeling, fencing, blocking and deblocking.

Persons who contemplate writing their own I/O PAC's are advised not to worry greatly about getting peak efficiency out of their peripherals, because the typical DATANET-30 user will find that he will not need to make many assemblies. This is because the DATANET-30 is most often used in situations where the same program (or a program set) runs day in and day out. Most customers will need several assemblies to get their original system debugged, and an occasional assembly over the years to make revisions. Thus, it is not worth a great programming effort to save five minutes per assembly. On the other hand, if one is using an extremely slow peripheral, such as Teletype, assembly time could run into hours. In that case, it would definitely pay to work with condensed records, where the I/O PAC blank-fills input records and blank suppresses output records.

B. FILE CONVENTIONS, GENERAL

1. Input Files.

The I/O PAC must return to the assembler with a code in the A-register to indicate the status of the "record" just read. The codes are:

Zero; normal record, no unrecoverable errors

Negative; End-of-file. (This should not be confused with end-of-reel on multireel tape file. If multireel files are permitted, the I/O PAC must determine for itself whether more reels are needed.) The EOF return does not deliver a record, but is given only if the assembler calls for a record after the last record has been delivered.

Positive, Nonzero; unrecoverable error. The assembler will continue, but if possible will flag the corresponding output record. BEFORE RETURNING to the assembler, the I/O PAC should, when practical, make appropriate attempts to reread the record. The I/O PAC, after determining that it cannot recover from the error, should place the file number (see below) in the C-register and add one to $ERRORS (see memory map at the end of this appendix). At the end of the run, the assembler will list the number of errors on each file. The I/O PAC may also inform the operator of errors, if the I/O PAC author so desires. If a standard Teletype configuration is available, the I/O PAC might log a message. Otherwise, it might load a register with an error code (and file code), turn on the buzzer, and wait for the operator to acknowledge the error (see PAUSE subroutine below).

2. Output Files.

The I/O PAC must return to the assembler with a code in the A-register to indicate the status of the record just written:

Zero; normal record, no unrecoverable errors

Nonzero; unrecoverable error. If possible, the assembler will flag the next output record to warn of the error. Before returning to the assembler, the I/O PAC should place the file number in the C-register and add one to $ERRORS. As was the case with input files, the I/O PAC may inform the operator of the error and let him take appropriate action. Whether or not assembly should continue after an output error is up to the I/O PAC author, and his decision may depend on the file. For example, an error in the object listing can probably be ignored. An error in the object program is more serious, and may call for rerun.

3. Nonexistent Files.

As stated earlier, some I/O PAC's will not provide for all allowable files because the required peripherals are not available. However, the assembler will not know that the file does not exist, so if a programmer uses a pseudo-operation which calls for a nonexistent file, the assembler will request a read (or write) of the I/O PAC. Therefore, the I/O PAC must provide linkage for every file. It is suggested that the I/O PAC place the file number in the C-register, add one to $ERRORS, and return to the assembler. If an input file, there should be an End-of-File return. If an output file, there should be a normal return.

4. Opening a File.

The assembler does not call for opening of files. When some opening procedures, such as rewinding, waiting for rewind, or initialization of buffering is required, the I/O PAC should take care of such things on the first call for the file, and should set a switch so that future calls on the file will bypass the opening procedure. If magnetic tapes are being used, it would be desirable to rewind all tapes on the first call for the Source file (even if the source file is not on tape) to minimize waiting time for rewinding tapes.

Particular care should be exercised on the opening procedures for those files which may be used several times, such as the Subroutine Library or the work file, if these are on magnetic tape. These tapes probably should not be rewound by the opening procedure after the first time they are opened, but should be rewound by the closing routines. The opening routines should, however, wait for rewinding tapes each time they are called upon.

5. Closing a File.

The assembler does call for the closing of every file, both input and output. The assembler may close an input file long before reaching end-of-file. For example, the assembler will close the Subroutine-Library as soon as it has found all requested subroutines.

When the assembler calls for the closing of nonexistent files, the I/O PAC should branch back immediately. There is no need to give any error indication. Even many existent files will not require any action for closing if they are not on magnetic tape. Magnetic tape files will normally require only rewinding if input files, or tape marking and rewinding if output files. If the files can be opened again, it may also be appropriate to set flags to indicate the "closed" status.

There is only one subroutine linkage for the closing of all files. The assembler enters the CLOSE routine with the file number in the C-register.

6. File Numbering System.

Every file is given a number, as described in the table at the end of this appendix. The number is used for several purposes. It serves as a key for the close routine, for logging errors in $ERRORS, and for keeping track of disc storage unit addresses assigned to the various files (see "Using the DSU for Input/Output" below).

C. SUBROUTINES AVAILABLE TO I/O PAC'S

The assembler contains several subroutines which are available to I/O PAC's. In general, they do not save registers. The MOVE and FILL routines are much less efficient than would be custom-programmed loops, but they do save space, which is important.

The subroutines are:

1. MOVE. Move a specified number of words from one area to another. Uses A-register as an index. Uses B-register for moving.

```
BRS     MOVE
INA     address from which data are to be moved
INA     address to which data are to be moved
DEC     number of words to be moved
```

2. FILL. Fills a specified number of consecutive words with a specified constant.

```
BRS     FILL
INA     address of area to be filled
                (Constant with which to fill the area)
DEC     number of words to be filled.
```

Example:

```
Fill 30 words at OUTPUT with zeros
BRS     FILL
INA     OUTPUT
ALF     000         (Constant for area)
DEC     30
```

3. PAUSE. Waits for Sign Switch (Console Switch 18) to be turned ON and then OFF. Turns off buzzer. (Does not turn on buzzer.)

4. DSUCYL. Calculate next disc address, cylindrically. (See "Using DSU as Peripheral.") Calculates the next available DSU address, in a cylindrical fashion, available for this file. File number must be indicated by C-register. The routine will exhaust one actuator position (96 records) and will then advance to next actuator position on the same disc. When the actuator reaches the edge of a cylinder, the DSU address advances to the next disc and moves to the inside edge of the cylinder (the first actuator position prescribed by the DSA card). When actuator reaches last disc allowed by "final address," will return to disc in "beginning address," next actuator position. Returns with DSU address in the A-register.

5. DSUDSK. Calculate next DSU address for this file in the Disc Mode. Will continue to advance along one disc until disc is exhausted. Will then go to disc with next-higher address. Returns with DSU address in the A-register.

D. USING THE DSU AS AN INPUT/OUTPUT DEVICE

Most DATANET-30's installed for message switching will have disc storage units. In many cases, the DSU will be the only high-speed peripheral available. Almost always there will be at least some open areas on the DSU which can be used as scratch areas, viz., the areas used during the day for queueing. Purely temporary disc areas may be used for the work file. Even semipermanent files, such as the object program and symbol-table dumps, may be placed onto the disc. In fact, in most installations, the user wants the object program on the DSU.

In some cases, the person who writes the I/O PAC will know what areas of the DSU will be available for the various files, or at least for some of the files. For example, if a given disc is always used for temporary storage, that area might also always be used for the work file, or for the source file if the source file is to be placed onto the DSU by an external computer. When fixed DSU areas are to be used by the assembler, the author of the I/O PAC can build that knowledge into the program. He may, if he desires, calculate his own DSU addresses.

In other cases, however, the DSU addresses to be used may vary from one assembly to the next. This is especially true of the object program and symbol-table dumps. To make this possible, the assembler includes a pseudo-operation, DSA, by which the programmer may specify a range of DSU addresses to be used for a given file. (For detailed instructions on filling out the DSA card, see Pseudo-Operations section of this manual.) The assembler will store the starting address in one table, and the ending address in another table. Each time the I/O PAC calls on one of the address-calculation subroutines described above (DSUCYL or DSUDSK) the subroutine will increment the starting address table to the next available disc address, and will check to see if the address is greater than the associated "ending address." If not, the subroutine will give an address to the I/O PAC. If the calculated address exceeds the ending address, the assembler will halt with the buzzer turned on. This is to protect other data on the DSU.

The DSA card has no provisions for assigning plug (channel) number or file-box number. It is assumed that the author of the I/O PAC knows the plug and box numbers. If he doesn't, he should select some file number which does not use the DSU, and use that to enter, on a DSA card, octal numbers which contain the necessary information.

The disc storage addresses must be specified before the file is called upon. The work file, and the old master if one is used, must be specified on the very first card. Since the control card must also be the first card, it was designed to dovetail with a DSA card so that they could both be the first card in the SOURCE deck.

Of course, if the source program is to be on the DSU, the I/O PAC must know the starting address of the source file even before the first card is read. The author of that I/O PAC has a problem. He may be able to assume a starting address (if source programs are always stored in the same place) or he may have to request the first address to be entered through the console switches.

If an I/O PAC is to use the address calculation routines in the assembler, even when using fixed addresses, it must make sure that the associated address ranges are stored in the proper tables discussed above.

Several files (for instance, work file, symbol-table dump) were deliberately designed as 32 words each so that they would block two per DSU record. The I/O PAC must take care of the blocking and deblocking. This is quite simple, since one can just add one to a flag and branch on ODD/EVEN. The blocking, of course, does require moving the records from or to an I/O area.

No provision was made for chaining. If chaining is necessary, the I/O PAC author should study the detailed instructions for each file to determine where he may place chaining addresses.

E. ASSEMBLING WITH TELETYPE

Assembling with Teletype is, to say the least, undesirable. The effective speed ratios between Teletype and DSU or magnetic tapes will be two or three orders of magnitude (i.e., 1/100 or 1/1000). Nevertheless, Teletype may occasionally be the only medium available, at least for the source program and for the object listing. (Usually, a DSU will be available for work file and object program.) When using Teletype, the objective is to reduce not only the number of records, but the number of characters to be processed. For this reason, the assembler contains a number of features, such as the Repeat field and the multiple-operand, to reduce the number

of characters required to define a problem. But the author of an I/O PAC could do considerably more to reduce character volume. For example, he might set up the source file as a free-format with field separators. Thus, the source file need not contain any redundant blanks. Sequence fields could be deleted. User programmers could be encouraged to use short symbols and concise remarks. The functions of the I/O PAC, then, would be not only to read/write records and convert them back and forth between Teletype code and BCD code (used by the assembler) but to "spread" an input record so that it looks like a card image, or to condense an output record to eliminate superfluous blanks.

If Teletype is being used also for the object listing, it might be desirable to print only the symbol tables, rather than an object listing. Or one might elect to print only selected records in the object listing. The idea of eliminating the object listing is not too farfetched, because one could have obtained a "free" copy of the source program either while it was being punched or while it was being transmitted.

If it is necessary to use Teletype for the work file and even the object program, the problem is more complicated because these are binary files, and one must store all eighteen bits of every word, and all bit combinations are legal. This may require going to some pseudo-code.

The interrupt linkage (memory cells 0 and 1) and the first 64 scan words were deliberately left open to permit assembling by Teletype. It is assumed that Teletype would be reading or writing concurrently with the assembly process. A limited amount of buffering should be provided in the I/O PAC so that the interrupt routine will have a place to assemble incoming data while the previous record is being processed. It should not be necessary to stop the Teletype and repoll it between records because the assembler should be able to keep up with 100 wpm Teletype. One case where the assembler would have trouble keeping up would be if a programmer used the multiple-operand option with many short symbolic operands (one or two characters each) whose symbols were toward the end of a long symbol table. This would mean that the assembler would have to search a long table in two or three character times, in addition to all its other processing.

File Descriptions

0. SOURCE FILE. This is a card image in one of the allowable formats. It really contains 81 columns. The 81st column should be blank.

   Occasionally a programmer forgets to follow his source program with an END card. In such cases, if the I/O PAC can detect an end-of-file condition, it should so indicate to the assembler, and the assembler will terminate the first pass and continue with subsequent passes. If the I/O PAC cannot detect an end-of-file condition, the program will just "hang" in a not-ready condition.

   The source file is a fairly high-volume file, so if buffering and/or blocking are feasible with the peripheral device, and if adequate memory is available, one may consider buffering or blocking. However, the source file is not the largest file. Furthermore, if an old master tape is used for updating source programs, the source file will consist only of changes on all but the first assembly. Therefore, unless there is memory to burn, one probably should not waste it on speeding up the source file.

1. WORK FILE. With the exception of a possible Subroutine-Library and the object listing, the work file is the largest file in the system. It contains the images of the source file, plus many control words generated by the first pass of the assembler. The work file will contain at least one record for every record in the source file, but it may also contain one or more overflow records. For example, any source card which generates several instructions or data words (such as messages or tables) will generate an overflow record.

   The work file is an output of the first phase. It serves as input to the last phase and to the optional phase if it is used (see control card). It also serves as the old master file for later updates. Because it is high volume, and because it may be passed several times, the work file should be put onto one of the highest-speed peripherals available for the job, such as DSU or magnetic tape. The work-file record was deliberately designed at 32 words so that it would block two per DSU record. Because two records completely fill a GE-200 Series DSU record, there is no space for a chaining address. However, if chaining is necessary for some reason, the author of the I/O PAC can cheat a bit on the use of the last word of each record because, as described below, only three bits of the last word are absolutely necessary.

   Work-file records are of two types: the main record, which contains the source-card image in the first 27 words and binary control information in the last five words; and overflow records, which consist entirely of binary control information. The last word of every record contains the following information:

   Bit 18 on identifies a main record; bit 18 off is an overflow record.

   Bit 17 on says this is the last record generated by the source card; bit 17 off says one or more overflow records follow this one.

   Bit 16 on says the source card is in old format.

   Bits 15-1 contain a binary sequence number which is assigned to each record by the assembler. The assembler checks the sequence number later when it reads the work file back. The sequence number is strictly for protection against lost records. If for some reason the I/O PAC needs to attach its own information (such as chaining address) to each record, the I/O PAC may use these bit positions, but when the file is read back, the I/O PAC should dummy in the sequence numbers to avoid extaneous error indications on the listing.

2. OLD MASTER FILE. This is simply the work file from a previous assembly. It is one of the inputs to an update run. It will necessarily have the same blocking factor as the work file. If one considers buffering the old master, he should remember that the old master will be in memory at the same time as the source file and the work file, as well as the Subroutine-Library and the symbol table load if they are used.

   The old master will contain both main and overflow records. It is not necessary for the I/O PAC to filter out the overflow records because the assembler will do this, but the I/O PAC may do this if it will have any particular speed advantage. In any case, the I/O PAC must deliver all 32 words of a record because the assembler will interrogate the sign bit of the last word in the record.

3. SUBROUTINE LIBRARY

This file may be recorded in any mode, and may be blocked and/or buffered according to the amount of memory available. The source file, work file, and old master (if used) will be in memory at the same time as the Subroutine Library. The Subroutine Library is a high-volume file and may be a high-frequency file, so speed is worth considering.

Records may be card images, or they may be "truncated" records to reduce file space. If they are truncated, the I/O PAC must blank fill the unused part of a card image before delivering the record to the assembler.

Each subroutine on the file must be preceded by an identification card. The ID card contains the literal "+-+" in columns 1-3 and the subroutine ID in columns 7-12. Columns 4-6 must contain blanks if the subroutine is in new card format. They must contain the literal "OLD" if the subroutine is in the old format (see Appendix B). The last subroutine must be followed by a card containing "+-+END" in columns 1-6. The I/O PAC need not watch for ID cards; the assembler will handle all of the housekeeping.

The Subroutine Library may be closed several times during an assembly. The library will usually be on mag tape. The I/O PAC should rewind the tape and set a flag when requested to close the library file. On later calls for subroutines, the I/O PAC should wait for rewinding tape if the closed flag is on.

4. SYMBOL TABLE DUMP

This is a low-volume file. There is little to be gained from either blocking or buffering the file.

The first two words of each record contain the Identification lifted from the DMP card. The third word is a binary sequence number attached by the assembler. The fourth word contains a word-count for this card. The remainder of the record consists of up to seven sets of four words each. Each set contains a symbol, its associated address, and its control bits.

5. LOAD SYMBOL TABLE

This reads the file which was written by symbol table dump. Again, there is little value in buffering or blocking.

6. OBJECT LISTING

This is one of the larger files in the system, so buffering will be desirable and, if mag tapes are used, blocking may be worthwhile for time-savings on the DATANET-30. (The media conversion routine will be printer-bound, so blocking won't do much for the tape-to-printer computer.)

The object listing is also the most complicated file because it contains slewing-control and variable-length records. The assembler will branch to the I/O PAC with two parameters in the registers. The B-register will contain the number of DATANET-30 records in the printer line. (39 maximum.) The I/O PAC should use this number to set whatever end-of-line indicator is needed by the machine that will do the printing.

The C-register will contain the slewing control. There are only three combinations.

Zero. Print, single-spaced.
One. Print, double-spaced.
Two. Slew to top of page, and then print, double-spaced.

7. OBJECT PROGRAM

This is a low-volume file and need not be either buffered or blocked. In fact, blocking would be undesirable in most cases because one wants the loader to use a minimum of memory for input buffers.

The record size will depend upon which output (card format or DSU format) was chosen by the programmer. The assembler will branch to the I/O PAC with the record-size in the B-register.

8. LOAD ASSEMBLER

This is a program-loader, to be included as part of the I/O PAC. The input area used by the loader must be in the areas assigned to the I/O PAC. (Most loaders use upper memory, but this is used by the assembler for the symbol table, which must be left intact between passes.) The loader may use any of the buffers used for other files. The loader must not clear memory.

9. READ WORK FILE

This routine reads back the work file which was written earlier (File #1). The assembler will read this file either once or twice, depending on whether or not the programmer elected to flag instructions which had to be forced to even locations.

The blocking factor for this file must be the same as for File #1.

It is not necessary to provide disc storage addresses for this file by DSA cards. The assembler will automatically use the addresses assigned to File #1.

If the magnetic tape is used for this file, it is recommended the close routine for File #1 mark and rewind the tape and that the close routine for File #9 rewind the tape and set a "closed" flag. Then, on the first call on File #9, the I/O PAC should wait for rewinding tape. This will eliminate the possibility of addressing the rewinding tape, even if the file is called upon more than once.

CONSOLE SWITCHES

Console Switches 16-10 are for use in installations which do not have sufficient peripheral devices to permit an "automatic start-to-finish" assembly. The switches may be used to give the operator a chance to change tape handlers, etc.

CS 16. Pause before executing source phase or before loading next phase of the assembly program. As soon as the assembler is loaded into memory, it will interrogate switch 16, and if CS 16 is on, the assembler will wait for it to be turned off before proceeding. When the first phase is finished, the assembler will once again interrogate CS 16 before calling for the loading of the next phase.

CS 15. Pause before executing optional phase. If the optional phase was not requested by the control card, the assembler will ignore CS 15.

CS 14. Pause before executing object phase, or before loading report phase.

CS 13. Repeat the object phase before loading the report phase, or repeat the report phase.

CS 12. Pause before reading or writing next record. This enables the operator to stop the assembler without losing data when the assembler is working in a read-time environment.

CS 11. Suppress printing during the object phase.

CS 10. Suppress "punching" of object program during object phase.

Example Of Use Of Switches 16-10.

Assume that the only peripherals, besides the standard-equipment paper tape reader, available for assembly are two tape handlers (one dual). This will not permit "automatic start-to-stop" operation. The operator will have to change tapes between phases.

Media conversions will be handled by an off-line computer. Subroutine-call, automatic update, symbol-table dump, and symbol table load will not be used because there are not enough peripherals available. Below are the procedures to use in running the assembly.

Mount a scratch tape on Handler B. This will be the work file. It may remain in place until the last phase of the assembly.

Mount the tape containing the assembly program on Handler A. Put CS 16 down. Mount the paper tape loop which will read the assembler from mag tape. Push Manual Load. The source phase of the assembler will be loaded into memory.

Because CS 16 is on, the assembler will not execute, but will wait for CS 16 to be turned off. Remove the program tape from Handler A and replace it with the source program. Turn CS 16 off for just an instant; then turn it back on again. The momentary off position will permit the source phase to execute.

When the source phase is finished, it will interrogate CS 16 again before loading the next phase. Because CS 16 is On, the program will pause. This gives the operator the chance to remove the source tape and to replace it with the program tape (which contains the assembler). When this is done, first turn CS 14 On and then turn CS 16 Off. This permits the assembler to load the optional and object phases (which are in memory together) into memory.

Assume that the optional phase is not being used (see control card). The assembler will proceed immediately to the Object phase, but because CS 14 is On, the assembler will not execute.

Replace the program tape on Handler A with a scratch tape. This tape will receive the object program. The work file is still needed during the object phase, so no peripheral is available to receive the object listing. Turn CS 11 and CS 13 on. Turn CS 14 Off for just an instant and then turn it back on again. This will permit the object phase to execute. Because CS 11 is on, the assembler will not call for the writing of the object listing.

At the end of the object phase, the assembler will interrogate CS 13. Because a "Repeat Run" was requested, the assembler will go back to the beginning of the object phase. But because CS 14 is on again, the assembler will not execute.

Replace the object program tape on Handler A with another scratch tape, which will receive the object listing. Turn CS 11 and CS 13 Off, and turn CS 10 on. Now turn CS 14 off for just an instant. This permits the object phase to execute once again. But because CS 10 is on, the assembler will not attempt to write the object program.

When the object phase is finished this time, the assembler will not re-execute the object phase because CS 10 is off. However, the assembler will pause before loading the report phase because CS 14 is on.

The reports are part of the object listing. Therefore, the tape on Handler A must not be disturbed. Remove the work file (no longer needed) from Handler B and mount the program tape. (Adjust handler addresses as necessary.) Turn CS 14 off. The reports phase will now load and the program will continue to end-of-job.

ASC II

DATA INTERCHANGE CODE

Character	Octal	Character	Octal	Non-Typing--Control	
				Function	Octal
@	600	Sp	500	NULL	400
A	602	!	502	SOM	402
B	604	"	504	EOA	404
C	606	#	506	EOM	406
D	610	$	510	EOT	410
E	612	%	512	WRU	412
F	614	&	514	RU	414
G	616	'	516	BELL	416
H	620	(	520	FEO	420
I	622	)	522	H. TAB	422
J	624	*	524	Line Feed	424
K	626	+	526	U. TAB	426
L	630	,	530	FORM	430
M	632	-	532	RETURN	432
N	634	.	534	SO	434
O	636	/	536	SI	436
P	640	0	540	DC_0	440
Q	642	1	542	X-ON	442
R	644	2	544	Tape Aux. On	444
S	646	3	546	X-OFF	446
T	650	4	550	Aux. Off	450
U	652	5	552	ERROR	452
V	654	6	554	SYNC	454
W	656	7	556	LEM	456
X	660	8	560	S_0	460
Y	662	9	562	S_1	462
Z	664	:	564	S_2	464
[	666	;	566	S_3	466
\	670	<	570	S_4	470
]	672	=	572	S_5	472
↑	674	>	574	S_6	474
←	676	?	576	S_7	476
				Ack.	770
				Alt. Mode	772
				Rub Out	776

Characters as they appear left-justified in DATANET-30.
_ Characters underlined are obtained in conjunction with shift key.

GENERAL ELECTRIC STANDARD CHARACTER SET

Standard Character Set	GE-Internal Machine Code	Octal Code	Hollerith Card Code	Standard Character Set	GE-Internal Machine Code	Octal Code	Hollerith Card Code
0	00 0000	00	0	↑	10 0000	40	11-0
1	00 0001	01	1	J	10 0001	41	11-1
2	00 0010	02	2	K	10 0010	42	11-2
3	00 0011	03	3	L	10 0011	43	11-3
4	00 0100	04	4	M	10 0100	44	11-4
5	00 0101	05	5	N	10 0101	45	11-5
6	00 0110	06	6	O	10 0110	46	11-6
7	00 0111	07	7	P	10 0111	47	11-7
8	00 1000	10	8	Q	10 1000	50	11-8
9	00 1001	11	9	R	10 1001	51	11-9
[	00 1010	12	2-8	-	10 1010	52	11
#	00 1011	13	3-8	$	10 1011	53	11-3-8
@	00 1100	14	4-8	*	10 1100	54	11-4-8
:	00 1101	15	5-8	)	10 1101	55	11-5-8
>	00 1110	16	6-8	;	10 1110	56	11-6-8
?	00 1111	17	7-8	'	10 1111	57	11-7-8
ↄ	01 0000	20	(blank)	+	11 0000	60	12-0
A	01 0001	21	12-1	/	11 0001	61	0-1
B	01 0010	22	12-2	S	11 0010	62	0-2
C	01 0011	23	12-3	T	11 0011	63	0-3
D	01 0100	24	12-4	U	11 0100	64	0-4
E	01 0101	25	12-5	V	11 0101	65	0-5
F	01 0110	26	12-6	W	11 0110	66	0-6
G	01 0111	27	12-7	X	11 0111	67	0-7
H	01 1000	30	12-8	Y	11 1000	70	0-8
I	01 1001	31	12-9	Z	11 1001	71	0-9
&	01 1010	32	12	←	11 1010	72	0-2-8
.	01 1011	33	12-3-8	,	11 1011	73	0-3-8
]	01 1100	34	12-4-8	%	11 1100	74	0-4-8
(	01 1101	35	12-5-8	=	11 1101	75	0-5-8
<	01 1110	36	12-6-8	"	11 1110	76	0-6-8
\	01 1111	37	12-7-8	!	11 1111	77	0-7-8

APPENDIX D

PROGRAM OVERLAYS

To be available at a later date.