File No. 1620-21
Form J26-4201-2

IBM Systems Reference Library

IBM 1620 Symbolic Programming System Specifications

The 1620 Symbolic Programming System is designed to simplify program preparation for the IBM 1620 Data Processing System. This manual contains the specifications of the system (Library Nos. 1620-SP-008 and 1620-SP-009) and describes 1620 symbolic language programming techniques as well as general principles and concepts of symbolic programming. A knowledge of 1620 machine language is presupposed.

This is a revision of Form J26-4201-1. While the format has been changed to conform to that of the Systems Reference Library, the original publication (J26-4201-1) is not obsoleted.

This Manual is to be used with the following IBM Programming Systems:

1620-SP-008	SPS Two Pass for Paper Tape
1620-SP-009	SPS Two Pass for Cards

Copies of this and other IBM publications can be obtained through IBM Branch Offices. Address comments concerning the content of this publication to: IBM, Product Publications Department, San Jose, California

1620 SYMBOLIC PROGRAMMING SYSTEM: SPECIFICATIONS

In order to perform a given task, the IBM 1620 Data Processing System must be presented with a detailed set of instructions in its own intricate language. This language, which consists of numerical codes, is called the "machine language." A logical sequence of instructions (or steps) in this language is called the machine language program. Specifying the steps in the solution of a problem using machine language can at times be a difficult and time-consuming job. The intricacy of the machine language and the complexity of a program may frequently cause clerical and logical errors. Also, a hand coded machine language program is not relocatable; that is, it cannot be executed from a different section of the processing unit without being manually changed.

In order to simplify the preparation of a program for the IBM 1620, the 1620 Symbolic Programming System has been developed. This bulletin is devoted to a description of the specifications of the system. It also contains, as introductory material, a section discussing the basic principles and concepts of symbolic programming. A knowledge of programming in 1620 machine language is presupposed.

BASIC PRINCIPLES AND CONCEPTS OF SYMBOLIC PROGRAMMING

In the early stages of data processing system development, programs were almost always written directly in the numerical and alphabetic notation (machine language) used by the system. As data processing systems developed, larger and more sophisticated machines were designed. Concurrently, programs for these machines became greater in length and complexity. As a result, programming efforts were greatly increased. Not only did it become difficult to memorize the many numerical and alphabetic codes required to write a program, but the length and complexity of the programs were conducive to increased numbers of clerical and logical errors. In addition, the problem of correcting errors in a program was intensified by the difficulty in tracing a machine language program through its many steps and finding a convenient way of including corrections.

In order to relieve the programmer of writing programs in machine language, symbolic programming systems were developed. Symbolic programming may be defined as a method wherein names, characteristics of instructions, or closely related symbols are used in writing a program. Data to be processed is referred to by name or other meaningful designation. Operation codes are written in an easily remembered mnemonic form rather than in the numerical language of the machine. Such systems permit the programmer to use a simpler, more familiar language in writing programs and often require a less detailed knowledge of the machine. Associated with each symbolic programming system is a machine language program called a "processor". The function of the processor is to translate the symbolic language of the programming system into the language of the data processing system. Generally, this translation is a "one-for-one" translation. That is, for each instruction written in symbolic form, one machine language instruction is produced.

The specific benefits derived from using easily recognizable symbols consisting of letters, numbers and special characters to represent corresponding elements of machine language, will depend to a large extent on the characteristics of the machine for which the system is designed. Generally, however, this feature makes it easier for persons unfamiliar with a program to follow the program's logic. It frequently eliminates the necessity of detailed flow charts and coding comments by providing, in the body of the instruction itself, the information usually found in these items. Also, coding is simplified, thereby decreasing the time required for coding and increasing coding accuracy.

Another advantageous feature of symbolic programming is the relative relationship between symbolic entries. Coding is facilitated by this feature as it permits instructions and data to be referred to before they are assigned actual machine addresses and without regard to their machine addresses. The feature permits sections of other programs or subroutines (short programs or routines common to a number of programs) to be easily incorporated in a program. It also permits each routine in a program to be written independently of the others with little or no loss of efficiency in the final program. Since instructions are not assigned storage locations by the programmer, the addition, deletion, modification or correction of instructions entails no reassignment of addresses. Finally, the feature makes programs and subroutines readily relocatable — i.e., they can be placed in varying machine locations as desired.

Paralleling the development of symbolic coding was the development of the macro-instruction concept. A macro-instruction is a synthetic instruction which, during the translation to machine language, refers the processor to a "library" from which a sequence of machine language instructions is extracted and placed in the machine language program. These instructions have been previously written, tested and stored in the library. Macro-instructions are used to save the programmer from coding certain repetitious sequences of instructions, such as arithmetic functions.

After a symbolic program, which may consist of a combination of the "one-for-one" symbolic instructions and the "many-for-one" macros, is completed, it must be converted into the format required by the machine on which it will be used. As was mentioned earlier, this is the function of the processor. Processors (sometimes called assembly programs) are designed to accomplish this conversion quickly, accurately, and as automatically as possible. Usually, an assembly program utilizes the machine for which the symbolic program is written. The typical assembly program analyzes all symbolic entries and macro-instructions and converts them to actual machine operating instructions and data, and establishes the specified relationships between them. As an additional feature, assembly programs also indicate various types of coding errors.

GENERAL DESCRIPTION OF THE IBM 1620 SYMBOLIC PROGRAMMING SYSTEM

The 1620 Symbolic Programming System consists of two distinct areas of discussion: the symbolic language and the processor.

The symbolic language is the notation in which a programmer codes the program. This language is in the form of mnemonic operation codes provided within the framework of the system, and also includes definitions of work areas, record areas and other data supplied by the programmer. Each separate item of information is written on a coding sheet with one statement per line. Statements are normally written in the order in which they are to be executed unless some other sequence is specifically indicated by the programmer. A program written in this manner intended for translation into machine language is called a "source" program.

After the program has been written in the symbolic language it is punched into the input tape. Each statement (one line of the coding sheet) when punched on paper tape must be terminated by an end-of-line character. Blanks need not separate one statement from another on the input tape. That is, the next statement may be punched immediately following the end-of-line character of the previous statement.

The processor is the 1620 machine language program which performs the actual functions of translation and assembly. The processor takes the source program in symbolic language, translates the mnemonic codes into machine language codes, assigns core storage addresses to instructions and symbolic data references, and assembles a finished machine language program known as the "object" program.

The following pages are devoted to a detailed description of the 1620 symbolic programming language, and a brief outline of the processor. The description proceeds as follows:

1. The first of the two sections which immediately follow is concerned with a description of the coding sheet on which the source program will be written. The second section covers the organization of the processor. The listing and operating instructions for the processor are not included in this bulletin. These will be made available in a subsequent publication.

2. A section called "Programming the 1620 using the Symbolic Programming System" describes in detail the various steps to be followed in coding a source program. This section covers the rules and conventions required for using the symbolic language. It describes the manner in which input/output areas, storage areas and constants are defined. Examples of all pertinent information are provided.

1620 SYMBOLIC PROGRAMMING SYSTEM CODING SHEET

All information relevant to the coding and subsequent assembly of the object program is entered on the 1620 Symbolic Programming System Coding Sheet. This form is illustrated in Figure 1. The information required to produce an object program falls into three categories as follows:

IBM

1620 Symbolic Programming System
Coding Sheet

Form No. X26-1227

Program _____

Page No. [ ][ ] of _____

Programmed by _____ Date _____

LINE 3 5	LABEL 6 11	OPERATION 12 15	OPERANDS & REMARKS 16 20 25 30 35 40 45 50 55 60 65 70 75
0,1,0			
0,2,0			
0,3,0			
0,4,0			
0,5,0			
0,6,0			
0,7,0			
0,8,0			
0,9,0			
1,0,0			
1,1,0			
1,2,0			
1,3,0			
1,4,0			
1,5,0			
1,6,0			
1,7,0			
1,8,0			
1,9,0			
2,0,0			

Figure 1

1. Area Definition — These statements are used to assign core storage for input areas, output areas and working areas. Assigned areas will be utilized by the object program and may contain the data to be processed and/or the constants (fixed factors or combinations of characters) required in the object program when processing data. Area definition statements are never executed in the object program.

2. Instructions — Most of the statements on the coding sheet will be the instructions, in symbolic language, which specify the job to be done by the object program. These entries will be translated and assembled as the object program.

3. Processor Control Operations — Processor control operations are commands to the processor which provide the programmer with control over portions of the assembly process. Instructions of this type are never executed in the object program.

The following paragraphs explain the use of each field on the coding sheet. The term "field," as used in connection with the coding sheet applies collectively to the character positions under each heading. Space is provided at the top of the sheet to identify and date the program. These areas, i.e., Problem, Programmer and Date are not part of the source program and are not punched.

Page Number

A two-character page number entry sequences the program sheets. This number, which must be numerical, will be punched as the first two characters of each entry from a sheet.

Line Number (Columns 3-5)

A three-character line number sequences the statements on each coding sheet. The first twenty lines on the program sheet are prenumbered 010-200. The six non-numbered lines at the bottom of the sheet are provided for the entry of statements inadvertently omitted and/or for sheet extension. Insertions may be referenced by numbering the statement with the hundreds and tens digit of the statement it is to follow and completing the statement number by adding any one of the units digit 1-9. This allows for up to nine insertions between each statement. Insertions must be punched in their proper sequence when preparing the input tape. Each line number will be punched as the third, fourth, and fifth position of its associated statement.

Label (Columns 6-11)

A label is the symbolic name, chosen by the programmer, of an area being defined or an instruction referred to elsewhere in the program. All labels are assigned addresses in storage during assembly. A reference to a label in the program is a reference to the address of the area or instruction so labeled. Consequently, a programmer need not be concerned with actual memory locations. Only those items specifically referred to elsewhere in the program need have a label. Unnecessary labels delay the assembly process. Unlabeled instructions should contain blanks in the label field.

A label may consist of up to six alphameric characters, left-justified in the label field. The first character of the label must be alphabetic (A through Z). Special characters are not permitted in the label. It is best to choose labels which are descriptive of the area or instruction to which they are assigned. Labels which have an obvious meaning not only provide easily remembered references for the original programmer, but also assist others who may assume responsibility for the program.

Operation (Columns 12-15)

The four-digit operation field will contain the mnemonic representation of area definitions, 1620 machine language instructions, macro-instructions, and processor control operations. These mnemonics, which are abbreviations of the operation to be performed, must be left-justified. Actual 1620 machine codes are not permitted. A list of the mnemonic equivalents of machine codes is found on page 37.

The functions of area definitions, macro-instructions and processor control operations are described in the following portions of this bulletin. The functions of the 1620 machine language instructions are described in the IBM Reference Manual, "1620 Data Processing System," form A22-4500.

Operands and Remarks (Columns 16-75)

The operands and remarks field is used to specify the information which is to be operated upon, and may contain, if desired, any additional remarks concerning the statement. In the case of area definition type statements this field will contain constants if special constant areas are being defined.

For instructions, the operands and remarks field will contain, at most, four items, three of which are operands and the fourth the remarks. The first two operands may be the symbolic or actual addresses of data or instructions and will be assembled as the P and Q portions of the instruction. The third operand is used to set flags in the assembled instruction and is called the flag indicator operand. The final item is the remarks which may be associated with each statement. Items are separated from each other by commas. The entire statement is terminated by an end-of-line character. A description of the format and arrangement of the entries in the operands and remarks field is given on pages 20 and 21. A brief description of the types of addresses which may be used in the P and Q operands is given below.

The operands which will be assembled as the P and Q portions of instructions may be one of four types: actual, symbolic, immediate, and asterisk.

Actual

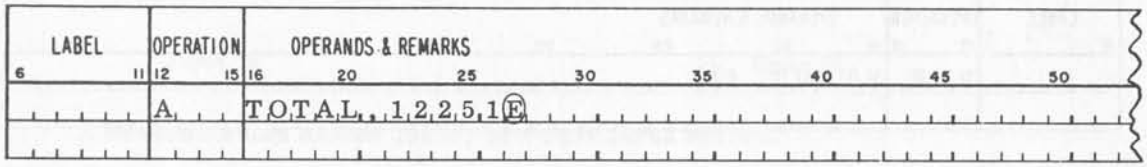
An actual address consists of five digits 00000-19999 and is, as the name implies, the actual 1620 core storage address of a piece of data or an instruction. High-order zeros of an actual address may be eliminated.

Symbolic

A symbolic address is the name assigned by the programmer to a piece of data or an instruction. A symbolic address is valid only if it is defined somewhere in the source program or if it is used as the label of an instruction. Symbolic addresses may contain from one to six letters or digits (no special characters) with the following restrictions:

a) The first, or left-most, position must be alphabetic.
b) Blanks may not appear within a symbol.

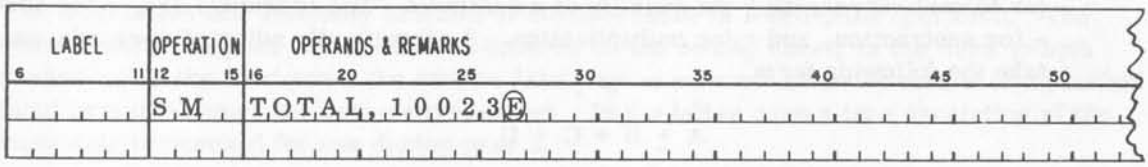
The example shown below contains both an actual address and a symbolic address.



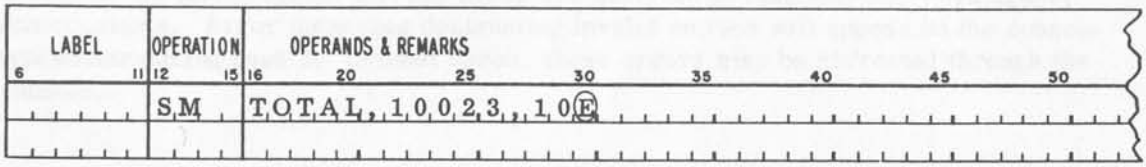
In this example, the data in the field whose actual address is 12251 is added to a field whose symbolic name (address) is TOTAL. The end-of-line character which terminates each statement is represented by the figure (E) .

Immediate

An immediate address is used with immediate type instructions and represents the actual data to be used by the instruction. It may be absolute or symbolic. During assembly the processor will automatically place a flag over position Q_7 of an immediate instruction unless otherwise indicated by a flag indicator operand. For example, the statement



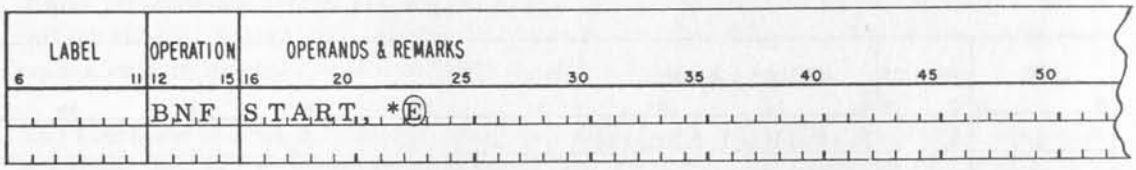
will cause the entire quantity 10023 to be subtracted from the field called TOTAL because the flag which terminates the field to be subtracted will be automatically placed over position Q_7. However the statement



will cause just the quantity 23 to be subtracted from the field called TOTAL because the flag indicator operand will place the field terminating flag over position Q_{10} rather than Q_7. There is one exception to this rule. That is, a transmit digit immediate instruction (TDM) does not require a flag; therefore, none will be set automatically by the processor.

Asterisk

The character, asterisk, when used as an operand will be interpreted by the processor as referencing the high-order (left-hand) position of the instruction itself. For example, the statement



indicates to the processor that Q portion of the instruction should contain the address of the instruction itself. If the assembled instruction is assigned address 1876 the assembled instruction would be 44 01234 01876 where START equals 1234. Thus, when executed in the object program, this instruction will examine its own left-hand position for a flag and either branch to instruction 01234 or continue on to the instruction located at 01888 on the basis of the examination.

Address Arithmetic

Address arithmetic may be used in conjunction with the P and Q operands. In this event an operand in the source program will contain more than one term. Thus, the address assembled in the P or Q portion of the instruction may be arithmetically adjusted if the first term is followed by an arithmetic operator and another term (or terms) representing the quantity of adjustment. The operators are, + for addition, - for subtraction, and * for multiplication. Arithmetically adjusted operands may take the following form

$$A \pm B * C \pm D$$

where A, B, C, and D may be either numerical quantities or symbols representing numerical quantities. In arithmetically adjusted operands, the multiplication operation is always performed first. A table containing examples of address arithmetic is shown in Figure 2b. Figure 2a contains the assigned address of the symbols used in 2b.

Symbols Used In Operands	Equivalent
ALPHA	1000
START	4000
L	12
ORIGIN	600
OUTPUT	15000

Figure 2a

P or Q Operands	Equivalent After Assembly
START + 40,	04040
ALPHA - 30,	00970
START + 2 * L,	04024
START * 3,	12000
ALPHA * 5 + 40,	05040
4 * 13 + OUTPUT,	15052
START + 4 * L - 1,	04047
ALPHA * L,	12000
500 + 20 * 3 - 11,	00549
OUTPUT - L * ALPHA + ORIGIN	03600

Figure 2b

Addresses which exceed 19999, are considered errors.

Address arithmetic could conceivably reduce the number of labels required by the source program by giving the programmer the ability to refer to a location which is a given number of locations away from a symbolic, actual, or asterisk operand. (Examples of address arithmetic on an asterisk operand is given in a later section of this bulletin, see page 21). Care must be exercised by the programmer when using address arithmetic since insertions or deletions could affect the adjusted operand. For example, if an operation is referred to an address plus 48 (e. g. , a transfer to skip four instructions,) the programmer must insure that no new instructions are introduced within the four instructions to make the "plus 48" incorrect.

ORGANIZATION OF THE PROCESSOR

The translation and assembly function of the processor is a two-pass operation. The source data, i. e. , the information as entered on the coding sheet, is the input to both passes. For the first pass, the source data may be entered either by means of a punched input tape or through the console typewriter. In the latter case a tape consisting of the input data is punched for use during pass 2.

Pass 1

During pass 1, the source data is read and a table of symbolic labels is prepared. Each symbolic label in the source program is placed in the table together with an equivalent address. By this process, storage areas are assigned to instructions, work areas, and constants. Error messages designating invalid entries will appear on the console typewriter during pass 1. In most cases, these errors may be corrected through the console.

Pass 2

The source data used in pass 1 is re-entered and processed. During this pass the instructions are assembled as follows:

a) **Operation codes are changed from mnemonic to actual machine language.**

b) **Operands are processed. Symbolic operands are looked up in the symbol table for their equivalent addresses. Address arithmetic is performed, if necessary, to complete the operands. In addition, flags are set in the assembled instruction as specified in the flag indicator operand.**

c) **The assembled instructions are punched out.**

The output tape produced by pass 2 contains, in addition to assembled instructions and addresses in which to store them, the constants and other data required by the object program. Loading instructions will appear at the beginning of the object tape; addition and multiplication tables will be punched at the end of the object tape. Thus the data necessary to load the object program, plus the object program, constants, etc., are on one tape ready to be entered into the 1620. In addition to the object tape, output from pass two will include a listing produced at the console, if desired. A sample listing is shown on pages 34 and 35. Error messages will be typed during pass 2 and corrections may be made from the console.

PROGRAMMING THE IBM 1620 USING THE SYMBOLIC PROGRAMMING SYSTEM

This section describes in detail the various steps to be followed in writing a program for the IBM 1620 using the Symbolic Programming System. The material contained in this section has been divided into the three categories of information required to write a symbolic program: Area Definitions, Instructions, and Processor Control Operations. In an effort to make this material more easily understandable, a numerical integration program (pages 28-33) is used as a theme from which examples are extracted to illustrate pertinent parts of the text.

AREA DEFINITIONS

In the course of performing its given function, a program will require the use of "input/output" areas, "work" areas, and "constants". An input and/or output area is, as might be expected, a portion of core storage assigned as an area into which a record, portion of a record, or piece of data will be read during input, or an area from which a record, portion of a record, or piece of data will be punched or typed during output. A work area is a portion of core storage assigned as an area into which a record, part of a record or piece of data will be transferred for processing. A constant is a fixed quantity or item of information which will remain the same throughout the course of the program or a phase of the program.

In programming the 1620, the input and output areas, work areas, and areas for storing constants must be assigned in storage for all records and any other data which are to be processed by the program. Such records and data normally consist of fields which are of known length and arrangement. Unless otherwise specified, areas being defined will be automatically assigned to core storage locations in the order in which they are defined in the program.

The use of symbolic programming enables the programmer to refer to input and output areas, work areas, and constants by their symbolic name without regard to their physical location within core storage. For example, the sample program used for reference utilizes a work area which is defined as a seven position field whose symbolic name (label) is DELTAX and a seven digit constant (0100000) whose symbol is X. To transfer this constant to the work area for processing merely requires an instruction

TF DELTAX, X (Transmit constant field X to DELTAX.)

To assign core storage space and to define the input and output areas, work areas, and constants requires the use of area definition codes. Area definition statements are never executed in an object program. Following are the area definition mnemonic operation codes.

Code	Description
DS & DAS	Define Symbol
DC & DAC	Define Constant
DSA	Define Symbolic Address
DSB	Define Symbolic Block.

Define Symbol (DS)

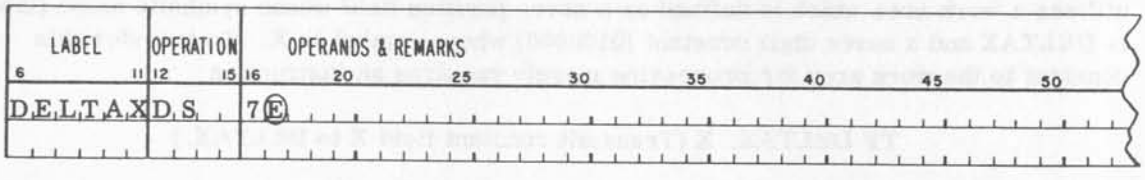
The mnemonic operation code, DS, may be used to define any contiguous portion of core storage as an area reserved for numerical data manipulation (i.e., input, output, or work area). The label (or symbolic name) by which this area may be referenced is placed in columns 6-11 of the coding sheet. The operation code DS must appear in the operation field (12-15).

The length of the field being defined must appear as the first operand. This operand may be an absolute value or a symbolic name. If a symbolic name is used the symbol must have been previously defined as an absolute value (see below). "Previously defined" means the definition of the symbol used must appear in a statement of the source program, which precedes the one in which it is used. Address arithmetic may be used with this operand.

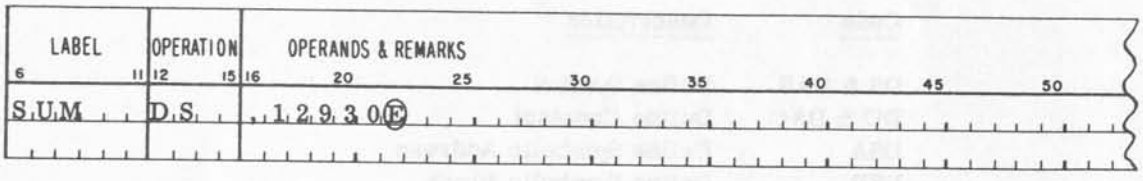
The address in core storage of the field being defined may be assigned by the programmer or the programmer may wish to let the processor assign the address. In the latter case, the statement may be terminated after the first operand. If the programmer wishes to assign the address, a second operand, which may be symbolic or actual, is used and the address of the field will be made equivalent to this operand. Since data fields are addressed by their rightmost (low-order) digit, the processor will assign this position as the address of the field. Address arithmetic may be used with the second operand. If the second operand is symbolic it also must have been previously defined. Addresses assigned by the programmer will not disrupt the sequence of addresses assigned by the processor.

To define a symbol without assigning any storage, i.e., to define it as an absolute value, a DS statement is used; however, in this case the first operand is omitted (or written as 0) and the second operand represents the equivalent value. The second operand may be an absolute value or a symbol; however, the symbol must again have been previously defined. To define storage which will not be referred to symbolically, the label of the DS statement may be omitted.

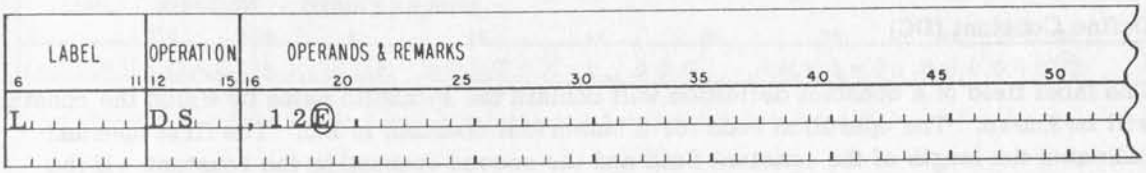
Several types of DS statements are explained below.



defines an area in core storage, seven positions in length into which an X value will be placed during calculation. The omission of the second operand indicates that the processor is to assign the address.



will assign to the area whose symbol is SUM the address 12930. This field may be referred to as SUM or 12930 since the latter will be its actual address.



will define the symbol L as being equivalent to the value 12. Subsequent uses of the symbol L in an operand are allowed since it has been defined.

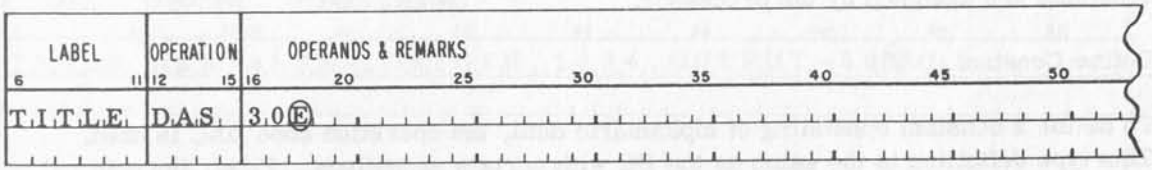
It should be noted that an area defined with a DS statement is always addressed at the rightmost position. Therefore, in order to use this area for input/output the leftmost digit must be addressed. This is accomplished by subtracting a number, which is one less than the length of the area, from the address of the area. The operand of an instruction which reads numerical data into the DELTAX field is written as DELTAX-6.

Define Symbol (DAS)

To define a field which will contain alphameric information (generally used for input/output areas) a define symbol statement with an operation code DAS is used. This type definition is much like the DS; however, certain differences do exist. First of all, the operand indicating length will be doubled by the processor to accommodate the alphameric coding of data.

Secondly, areas used for alphameric input and output are addressed by the "numerical" portion of the left-hand (high-order) position of the field. This address must be odd-numbered. For example, the word TEN when converted to alphameric coding requires six core positions and is coded as: 634555. The first, third and fifth positions represent the "zone" portion of the T, E, and N, respectively, while the second, fourth, and sixth the numerical portion. The address of this field must be odd-numbered and reference the numerical portion of the T (3). If the processor assigns the address, adjustments will be made, if necessary, to make the address odd.

The following example illustrates a DAS statement.



defines an area for input/output which will contain a title consisting of 30 characters. The processor will assign 60 positions in core storage to accommodate alphameric coding. The listing will indicate this by typing 30 X 2 when this statement is assembled and listed. The omission of the second operand will cause the processor to assign an address.

During an internal field transmission in which an input/output area defined with a DAS is utilized the area must be addressed at its right hand (low-order) position. This address may be achieved through address arithmetic, i.e., TITLE + 2 * 30 - 2.

Define Constant (DC)

The label field of a constant definition will contain the symbolic name by which the constant will be known. The operation code for a numerical constant is DC. The first operand indicates the length of the constant field and the second operand is the constant. If the programmer wishes to assign an address to this constant, a third operand may be used. Omission of the third operand causes the processor to assign the address. Constants are addressed by the rightmost (low-order) position of the field.

The first and third operands may be symbolic or actual and may be arithmetically adjusted. To be valid, a symbolic operand must have been previously defined. During assembly, the processor will place a flag over the left-hand position of the constant field. Should the length of the field be greater than the constant specified, the constant will be right-justified in the field with high order zeros inserted. A length of field which is smaller than the constant is invalid.

The constant must be in the form of an unsigned integer for a positive number and a signed integer for a negative number. A negative number produces a flag over the units position during assembly. A record mark may be used in the constant but must be in the units position and must be written as @. Negative constants containing a record mark (@) in the units position will have a flag placed over the digit preceding the record mark. Constants may not exceed 50 characters.

If the constants 0100000 and -0004337769 are required they may be defined as follows:

LABEL	OPERATION	OPERANDS & REMARKS
X	DC	7,,1,0,0,0,0,0Ⓔ
CONST	DC	1,0,,-4,3,3,7,7,6,9Ⓔ

In both cases the length of field is greater than the constant, and the addresses of these constants are assigned by the processor.

Define Constant (DAC)

To define a constant consisting of alphameric data, the operation code DAC is used. This type definition is the same as the DC with certain exceptions. First, the operand indicating length will be doubled to accommodate the alphameric coding of data. Second, the address assigned to an alphameric constant references the "numerical" portion of the left-hand (high order) character of the field. This address must be odd-numbered.

In the sample program the constant

LABEL (6–11)	OPERATION (12–15)	OPERANDS & REMARKS (16–50)
OUTPUT	DAC	31,FOR DELTAX=0.000,, AREA=0.00000@E

was assigned address 03101 in core storage. This address is the core location of the "numerical" portion of the F. (Second actual core position of the constant.) The entire constant occupies 62 core positions. A fifty character constant (maximum allowed) will occupy 100 positions of core. A flag will be set over the left hand position of the field. Addressing this constant for internal core transmission requires an address: OUTPUT+31*2-2.

It should be noted that in the sample program this constant was used as the output area. Quantities obtained in the calculations replaced the zeros of the constant.

Define Symbolic Address (DSA)

It may be desirable at some point in a program to store a series of addresses as constants. These addresses may be used for instruction initialization or modification, or for setting up a table of addresses through which the programmer may index to modify a routine.

The operation code for this statement is DSA. Each entry (symbolic or actual) in the operands field will cause its equivalent machine address to be stored as a five digit constant. The constants are stored adjacent to each other with a flag over the high-order position of each. The label field of this statement must contain the symbolic name by which the table of constants may be referenced. An address at which this table is stored in core storage may not be assigned by the programmer nor may any remarks be associated with the statement. The address assigned by the processor is the address at which the right-hand digit of the first constant will be located.

In the following example, the symbols used are equivalent to the addresses shown in Figure 2a (page 11).

LABEL (6–11)	OPERATION (12–15)	OPERANDS & REMARKS (16–50)
TABLE	DSA	ALPHA,ORIGIN,1234,OUTPUT-50@E

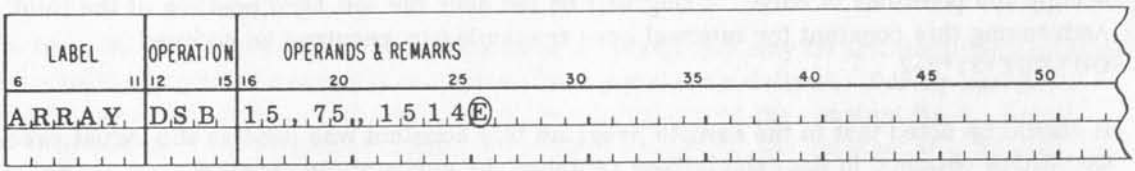
The constant will be stored as

$$\overline{0}1000\overline{0}0600\overline{0}123\overline{4}14950$$

If the first digit of the entire constant is located at 01200, then the address equivalent to TABLE is 01204.

Define Symbol Block (DSB)

To define an area of storage for the storing of a numerical array a DSB statement is used. The label of this statement will be converted to the address at which the first element of the array is stored (i.e., the right-hand position of the first element). The first operand indicates the size of each element; the second indicates the number of elements. Either or both operands may be symbolic or actual. (If symbolic, the symbol must have been previously defined). If the programmer wishes to assign the address, a third operand is required. For example, to store an array of 75 elements with each element containing 15 digits, the statement would be



In this example, the array would begin at location 1500 (left most position of the first 15 digit element). ARRAY is equivalent to 1514 (address of first element).

The area definition statements just described provide the object program with the input/output areas, work areas, and constants it requires to accomplish its assigned task. The statements discussed never produce instructions which are executed in the object program. The entries, DS, DAS, and DSB merely define storage. The entries, DC, DAC, and DSA define storage and produce, on the object tape, the machine address of the area being defined and the constant(s) which will be stored in this area. Constants are then loaded with the object tape.

Area definition statements may be entered at any point in the source program. However, it is wise to place these definitions off by themselves (not within the instruction area), preferably at the beginning or end of the program. Otherwise, the programmer must take special care to branch around the area defined so the program will not attempt to execute something in a data area as an instruction.

No statement in the source language may exceed 75 characters.

1620 SYMBOLIC PROGRAMMING INSTRUCTIONS

This section discusses the operations (instructions), written in symbolic language, which will be translated by the processor into 1620 machine language. The function of each machine language code is discussed in the IBM Reference Manual, "1620 Data Processing System", form A22-4500.

Each instruction in symbolic language may contain a label if so desired, must contain a mnemonic operation code, and may have none or as many as three operands. Remarks may be associated with each instruction. The entire instruction statement is terminated by an end-of-line character.

Label

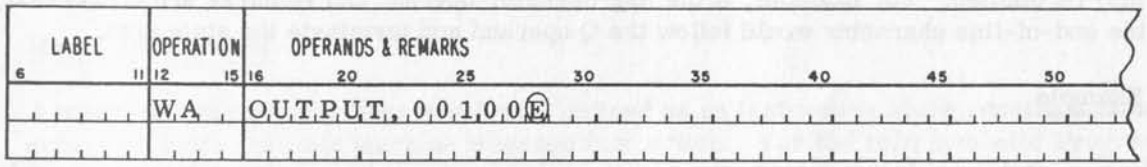
Any instruction in the source program may be labeled. Generally speaking, a labeled instruction is one which is referenced or used as a point of reference elsewhere in the program. Many times an instruction which is labeled is the first instruction of a subroutine. For example, the first instruction of the sample program is appropriately labeled START. A transfer or branch to the first instruction of the program can be effected by coding the instruction, B START. Any instruction between the one labeled START and the one labeled ASINE may be referenced by the operand START+24 (third instruction), START+60 (sixth instruction), or ASINE-12 (eighth instruction).

Operation Code

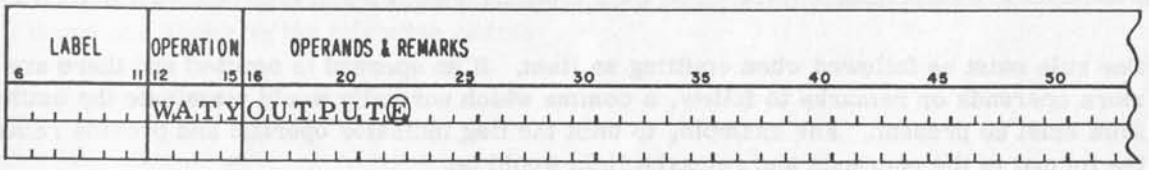
For each machine language operation code, there exists a mnemonic abbreviation. For example, the operation code 27, Branch and Transmit, is written as BT. A list of the machine language codes and their mnemonic equivalents is found on page 37. Note on this list the eight codes marked with an asterisk. These are the codes which require modifiers in the Q operand. When the direct mnemonic abbreviation is used for one of these codes, the programmer must supply the Q operand in absolute.

The 1620 Symbolic Programming System does, however, make available for these type codes a mnemonic representation for many, and in most cases all, of the several operations the code may perform. For example, the BI (Branch Indicator) instruction has twelve different mnemonics which represent nine of the possible fifteen configurations. The same is true for the BNI instruction. Input/Output instructions (i.e., Read, Write, Dump) have unique mnemonics for every possible configuration of these instructions (see page 38).

Thus when coding an instruction to write alphamerically on the console typewriter, the programmer may code



where 01 is the absolute modifier specifying the typewriter and is necessary to complete the instruction. Or the programmer may code

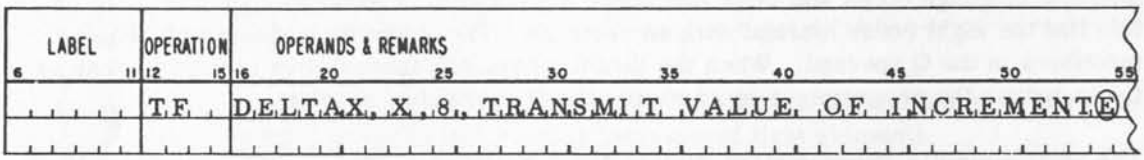


where the unique code WATY is all the information required as to the mode and unit desired. The processor will convert the mnemonic code to the proper machine language code (39) and insert the necessary modifier.

For the six operations (error indicator checks) of each of the codes BI and BNI for which no unique mnemonic is available, the programmer must use the mnemonic BI or BNI and must supply the modifier. A list of the unique mnemonics and their equivalents is found on page 38.

Operands and Remarks

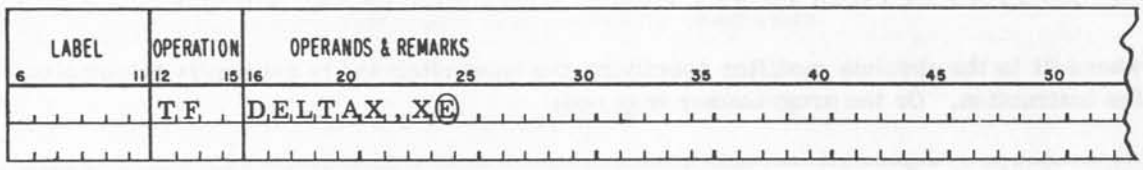
For each instruction, up to four items may be entered in this field; a maximum of three operands, and remarks if they are desired. These four items are coded and subsequently punched on tape in "free" form. That is to say, a fixed number of positions is not assigned to each item. Rather, each item desired in the instruction is separated from the following item by a comma, with an end-of-line character terminating the entire statement. At most, three commas will be used since a comma need not follow the last item entered. An instruction in which all four items are used is as follows:



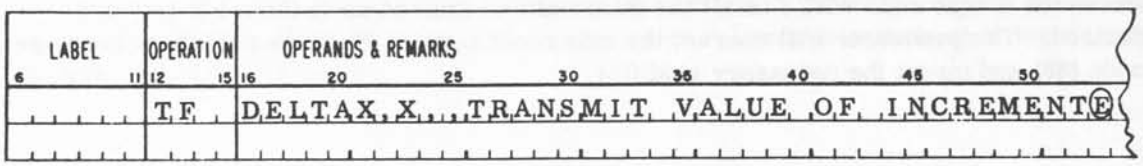
In this example, DELTAX and X are the symbolic operands which will be converted to the P and Q portions of the assembled instruction. The third operand will cause a flag to be set in position Q_8 of the assembled instruction. The remainder constitute the remarks. Remarks serve no function other than to aid the programmer when coding. This sequence of P and Q operands, flag indicator operands, and remarks must be followed in all instructions.

An instruction need not, however, contain all four items. Any one, or more than one may be omitted. For example, if the flag indicator operand and remarks are unnecessary, the end-of-line character would follow the Q operand and terminate the statement.

Example



One rule must be followed when omitting an item. If an operand is omitted and there are more operands or remarks to follow, a comma which normally would terminate the omitted item must be present. For example, to omit the flag indicator operand and provide remarks, the format of the operands and remarks field would be:



Commas indicating omission need not be present in statements in which the omitted item(s) follows the last item of data desired for the instruction. For example, in the statement in which both the flag indicator operand and the remarks are omitted, the end-of-line character follows the second operand with no intervening commas required.

An omitted P or Q operand will cause zeros to be placed in the P or Q portion of the assembled instruction.

The flag indicator operand specifies the positions of the assembled instruction, which are to contain flags. These positions are numbered, from left to right, 0-11 and must be listed sequentially. For example, if positions 2, 7 and 10 are to be flagged, the flag indicator operand should be 2710, not 2107. All positions may be flagged if desired. The operand would be 01234567891011 and must be written in that order.

If the flag indicator operand is omitted, normally no flags will be set. The exception to this has already been shown. All immediate instructions (except TDM) will have a flag set automatically in position Q_7 if the flag indicator operand is omitted. If the operand is present, only the position(s) indicated will be flagged.

Pages 8 to 11 describe various types of addresses which may be used in the P and Q operands. Examples of address arithmetic are also given. Address arithmetic is permitted on all types of operands including the asterisk. For example, an asterisk operand references the left hand (high-order) position of the instruction in which it is contained. Therefore to branch to an instruction five instructions ahead, the present instruction could be coded as B *+60. An asterisk operand may also be arithmetically adjusted using the multiplication operator (an asterisk). Thus, the instruction B *+5*L, where L = 12, is the same as the instruction B *+60. The second asterisk in this operand indicates multiplication.

The sample program on pages 29 to 33 contains many examples of address arithmetic used in conjunction with symbolic and asterisk operands.

Macro-instructions

A macro-instruction has been previously defined as an instruction which, during assembly, generates more than one machine language instruction. For the 1620 Symbolic Programming System, thirteen macro-instructions are available. Each of these macros, when used in a source program, will generate the instructions which provide linkage to one of thirteen subroutines. The subroutines, which are of two categories called arithmetic and functional, are described in the bulletin "IBM 1620 Subroutines: Preliminary Specifications," form J26-4203. The subroutines and their associated macro-instruction mnemonic operation codes are shown in the following chart:

ARITHMETIC				FUNCTIONAL	
Fixed Point		Floating Point		Floating Point	
Title	Macro	Title	Macro	Title	Macro
Divide	DIV	Floating Add	FA	Floating Square Root	FSQR
		Floating Subtract	FS	Floating Sine	FSIN
		Floating Multiply	FM	Floating Cosine	FCOS
		Floating Divide	FD	Floating Arctangent	FATN
				Floating Exponential (natural)	FEX
				Floating Exponential (base 10)	FEXT
				Floating Logarithm (natural)	FLN
				Floating Logarithm (base 10)	FLOG

NOTE: The bulletin (Form J26-4203) discussing the thirteen subroutines available through the use of macro-instructions, also contains general discussions of "linkage" instructions and "floating point arithmetic."

In addition to creating linkage to the subroutine desired, the use of a macro-instruction will cause the subroutine(s) required to be punched into the object program tape. The necessary subroutines will thus be loaded into core storage during the loading of the object program. Incorporating the subroutines into the object program requires, however, that all subroutines be available to the processor during assembly. Therefore, a separate tape (provided by IBM) containing all of the subroutines must be assembled in conjunction with the source program. This tape must be entered during both passes of assembly and must follow the source program tape. The subroutine(s) required will be selected by the processor from this tape, assigned core storage space and punched into the object program tape for subsequent loading as part of the object program.

During the processing of the source program, the first occurrence of a macro-instruction related to any one of the four floating point arithmetic subroutines will cause all four floating point arithmetic subroutines to be punched into the object program tape. Subsequent macros for these subroutines will, when encountered by the processor, merely cause the linkage to the desired subroutine to be generated.

If called for through the use of any one macro, the floating point arithmetic subroutines will, during the loading of the object program, always be loaded into core storage beginning at location 00402. That is to say, these subroutines are not relocatable and thus cannot be placed in varying machine locations as desired.

The floating point functional subroutines and the fixed point arithmetic subroutine (Divide) will appear on the subroutine tape in a form of symbolic language and will be relocatable. These subroutines, when called for by a macro, will be selected by the processor from the subroutine tape and will be assigned to an area in core storage which immediately follows the last location assigned to the source program by the processor. Care must be exercised by the programmer to provide, between the last location assigned by the processor and 19999, sufficient space to accommodate the subroutines called for.

Each floating point functional subroutine and the fixed divide subroutine on the tape will be complete with the constants and working areas it requires for execution. During assembly, however, those constants and working areas which may be common to several of the subroutines will be assembled into the object program only once. Sharing common work areas and constants eliminates redundancy and thereby minimizes storage requirements. In order to estimate the amount of storage required by the subroutines to be called for, it may be wise to total the number of instructions in each subroutine desired and the amount of working areas and constants associated with each subroutine. This will yield the most conservative figure.

It should be noted that the use of any macro-instruction related to a floating point functional subroutine will also automatically cause all four floating point arithmetic subroutines to be punched into the object program tape and subsequently loaded into the fixed area of core storage assigned to these arithmetic subroutines. Also, since the Arctangent subroutine and both Logarithm subroutines (natural and base 10) require the fixed point divide subroutine, the macros FATN, FLN, and FLOG will call in the Divide subroutine.

Macro-instructions for the Arithmetic Subroutines — The macro-instructions required for the arithmetic subroutines are written as follows:

LABEL (6–11)	OPERATION (12–15)	OPERANDS & REMARKS (16–)
	FA	A,,B(E)
	FS	A,,B(E)
	FM	A,,B(E)
	FD	A,,B(E)
	DIV	A,,B,,SHIFT,,LQ(E)

In each case the A and B operands represent the addresses of the quantities to be added, subtracted, etc. For the fixed divide routine two additional operands, SHIFT and LQ, are required. The SHIFT operand is described in the following paragraph. The fourth operand (LQ) indicates the length of the quotient desired by the programmer. All operands (A, B, SHIFT, and LQ) may be symbolic or actual, and may be used in conjunction with address arithmetic. Remarks are not permitted in macro-instructions.

The fixed divide subroutine (DIV) accomplishes division through successive subtraction. It is therefore necessary to specify the position of the divisor with relation to the dividend when the first subtraction takes place. This is specified in the SHIFT operand. The SHIFT operand represents the displacement of the right hand position of the divisor in relation to the right hand position of the dividend. If SHIFT is positive, the divisor is displaced to the left the number of positions specified by the operand; if negative, displacement is to the right. The SHIFT operand should be such that no more than nine subtractions will take place before the first overdraw is encountered; otherwise, a divide-overflow will be indicated.

When floating addition or subtraction is executed, the A operand is replaced by the sum or difference. In the case of floating multiplication or division, the product or quotient is stored with its low-order digit at 00099. The low-order digit of the quotient generated in the fixed point divide subroutine is stored at 00099 minus the length of the divisor.

The linkage instructions which the processor generates for the macros FA and FS are equivalent to the following symbolic instructions:

```
LABEL      OPERATION   OPERANDS & REMARKS
6          11 12  15  16  20        25        30        35        40        45        50
           TFM         SUBR+K,,*+35(E)
           TFM         SUBR+C,,A(E)
           B           SUBR,,B,,7(E)
```

where SUBR is the address of the first instruction to be executed in the desired subroutine; C and K are constants supplied by the processor; and A and B represent the addresses as specified in the macro.

For the macros FM and FD, the instructions generated are equivalent to the following symbolic instructions:

```
LABEL      OPERATION   OPERANDS & REMARKS
6          11 12  15  16  20        25        30        35        40        45        50
           TFM         SUBR+11,,*+35(E)
           TF          SUBR+C,,A(E)
           B           SUBR,,B,,7(E)
```

where the symbols are the same as described above.

The fixed-divide macro generates the following linkage instructions:

```
LABEL      OPERATION   OPERANDS & REMARKS
6          11 12  15  16  20        25        30        35        40        45        50
           TFM         SUBR+K,,*+35(E)
           TFM         SUBR+C,,A(E)
           B           SUBR,,B,,7(E)
           DSA         X,Y(E)
```

The quantities represented above by X and Y will be computed by the processor using the SHIFT and LQ operands. These quantities will be utilized in the fixed divide subroutine.

Macro-instructions for the Functional Subroutines — The macro-instructions for the floating point functional subroutines are written as follows:

```
LABEL   OPERATION   OPERANDS & REMARKS
6    11 12   15  16  20   25   30   35   40   45   50
        FSQRT   A,,B(E)
        FSIN    A,,B(E)
        FCOS    A,,B(E)
        FATN    A,,B(E)
        FEX     A,,B(E)
        FEXT    A,,B(E)
        FLN     A,,B(E)
        FLOG    A,,B(E)
```

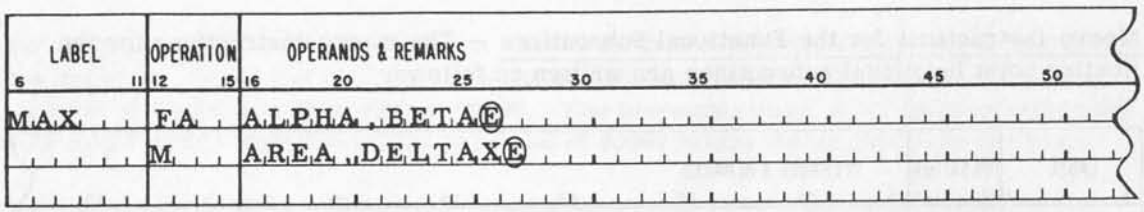
where B is the address of the argument to be evaluated and A is the address at which the programmer wishes the result to be placed. These addresses may be actual or symbolic and may be used in conjunction with address arithmetic. Asterisk may also be used as an address; it will reference the leftmost position of the first linkage instruction generated. The linkage instructions generated by any one of the above macros is as follows:

```
LABEL   OPERATION   OPERANDS & REMARKS
6    11 12   15  16  20   25   30   35   40   45   50
        TFM     SUBR+C,,*+3,5(E)
        TFM     SUBR+K,,A(E)
        B       SUBR,,B,,7(E)
```

where SUBR is the address of the first instruction to be executed in the desired subroutine, C and K are constants supplied by the processor, and A and B represent the addresses specified in the macro.

General Notes on 1620 Macro-instructions — Each of the subroutines for which a linkage is generated will compute the return address and transfer to this address at the completion of the subroutine when it is executed in the object program. Note that the information provided to the subroutine by the linkage instructions includes the necessary data to compute this return address.

A macro-instruction may be labeled in the source program. During assembly, a reference to this label will be a reference to the first instruction generated by this macro. Care must be exercised by the programmer when using address arithmetic with this label. For example, suppose in the source program the following entries are made:



Four instructions will result from this sequence, the first three of which are linkage to a floating add subroutine. MAX is the address of the first instruction of the linkage. Now suppose at a later point in the source program, the programmer decides to branch to the multiply instruction and codes B MAX+12. Since the processor has assigned the address equivalent to MAX to the first linkage instruction generated, a branch to MAX+12 is a branch to the second linkage instruction rather than to the desired multiply instruction. Similar precautions should be exercised if the operands of the macro-instruction are being modified by another instruction in the source program. A reference to MAX+5 is not a reference to the P operand of the macro but a reference to the P operand of the first linkage instruction generated.

PROCESSOR CONTROL OPERATIONS

For the 1620 Symbolic Programming System, two types of commands are provided to control the assembly process. These commands are Define Origin and Define End. They are never executed in the object program. In addition to a description of these two operation codes, this section includes a discussion of Comments.

Define Origin (DORG)

During program assembly, the 1620 SPS processor assigns core storage addresses to instructions, input/output areas, work areas, and constants, as they are encountered. If not otherwise specified, addresses will be automatically assigned beginning with an address which will be equivalent to 00402 plus the storage required for the arithmetic floating point subroutines. (The precise address will be made available at a later date.) Should the programmer wish to initiate address assignment at another location, he may do so through the use of a DORG statement. However, if any one of the macros which results in the incorporation of the four floating point arithmetic subroutines into the object program is used, the programmer should avoid specifying an origin which is less than the minimum address at which the processor automatically begins assignment. Failure to do so will result in the address assignment to begin where specified by the programmer and thereby cause the subroutines to be loaded over a portion of the object program.

The code DORG must appear in the operation field. The address at which assignment is to begin is specified as the first operand. A DORG statement may not be labeled. Origin statements may be used at any point in the source program. All entries subsequent to the DORG will be assigned addresses beginning at the particular location specified by the programmer. Addresses specified may be symbolic or actual. If symbolic, the symbol must have been previously defined.

Define End (DEND)

The statemend DEND is the last statement entered in the source program and is a command to the processor that all statements of the source program have been processed. A DEND statement may also be used to begin the execution of the object program immediately after loading. This is accomplished by the presence of an operand representing the starting address of the program. This operand may be symbolic or actual. Should the operand specifying the starting address be omitted the 1620 will halt at the completion of loading the object program and the operator must take steps to manually start the program.

Comments

Provisions have been made for the programmer to insert lines of descriptive information in the source program. This information, when encountered by the processor, will not be assembled nor will it affect the assembling procedure.

Comments are indicated by an asterisk in the first position of the label field, and the comments may begin immediately after the asterisk. See sample program on pages 29 to 33 for examples of comments.

FUNCTIONAL LIST OF 1620 SYMBOLIC PROGRAMMING MNEMONIC OPERATION CODES

AREA DEFINITION

Operation Code	Description
DS & DAS	Define Symbol
DC & DAC	Define Constant
DSA	Define Symbolic Address
DSB	Define Symbolic Block

INSTRUCTIONS

TYPE	OPERATION CODE		OPERATION
	Mnemonic	Machine	
Arithmetic	A	21	Add
	AM	11	Add (Immediate)
	S	22	Subtract
	SM	12	Subtract (Immediate)
	C	24	Compare
	CM	14	Compare (Immediate)
	M	23	Multiply
	MM	13	Multiply (Immediate)
Internal Data Transmission	TD	25	Transmit Digit
	TDM	15	Transmit Digit (Immediate)
	TF	26	Transmit Field
	TFM	16	Transmit Field (Immediate)
	TR	31	Transmit Record
Branch	B	49	Branch
	BNF	44	Branch No Flag
	BNR	45	Branch No Record Mark
	BD	43	Branch on Digit
	*BI	46	Branch Indicator
	*BNI	47	Branch No Indicator
	BT	27	Branch and Transmit
	BTM	17	Branch and Transmit (Immediate)
	BB	42	Branch Back
Input Output	*RN	36	Read Numerically
	*WN	38	Write Numerically
	*DN	35	Dump Numerically
	*RA	37	Read Alphamerically
	*WA	39	Write Alphamerically
Miscellaneous	*K	34	Control
	SF	32	Set Flag
	CF	33	Clear Flag
	NOP	41	No Operation
	H	48	Halt

* Mnemonic codes which require modifiers in the Q operand.

PROCESSOR OPERATION CODES

Operation Code	Description
DORG	Define Origin
DEND	Define End

UNIQUE MNEMONICS TO REPLACE MNEMONICS REQUIRING MODIFIERS

The following lists of unique mnemonics may be used in place of the mnemonics requiring a modifier. Modifiers for these mnemonics are supplied by the processor.

MNEMONICS			
Equivalent	Unique	OPERATION	UNIT REFERENCED
	BH	Branch High	High-positive indicator
	BP	Branch Positive	High-positive indicator
	BE	Branch Equal	Equal-zero indicator
	BZ	Branch Zero	Equal-zero indicator
BI	BV	Branch Overflow	Overflow indicator
	BA	Branch Any	ANY latch
	BNL	Branch Not Low	High-positive/equal-zero indicator
	BNN	Branch Not Negative	High-positive/equal-zero indicator
	BC1	Branch Console Switch 1 ON	Program Switch 1
	BC2	Branch Console Switch 2 ON	Program Switch 2
	BC3	Branch Console Switch 3 ON	Program Switch 3
	BC4	Branch Console Switch 4 ON	Program Switch 4
	BNH	Branch Not High	High-positive indicator
	BNP	Branch Not Positive	High-positive indicator
	BNE	Branch Not Equal	Equal-zero indicator
	BNZ	Branch Not Zero	Equal-zero indicator
	BNV	Branch No Overflow	Overflow indicator
	BNA	Branch Not Any	ANY latch
	BL	Branch Low	High-positive/equal-zero indicator
BNI	BN	Branch Negative	High-positive/equal-zero indicator
	BNC1	Branch Console Switch 1 OFF	Program Switch 1
	BNC2	Branch Console Switch 2 OFF	Program Switch 2
	BNC3	Branch Console Switch 3 OFF	Program Switch 3
	BNC4	Branch Console Switch 4 OFF	Program Switch 4
RN	RNTY	Read Numerically	Typewriter
	RNPT	Read Numerically	Paper Tape
WN	WNTY	Write Numerically	Typewriter
	WNPT	Write Numerically	Paper Tape
DN	DNTY	Dump Numerically	Typewriter
	DNPT	Dump Numerically	Paper Tape
RA	RATY	Read Alphamerically	Typewriter
	RAPT	Read Alphamerically	Paper Tape
WA	WATY	Write Alphamerically	Typewriter
	WAPT	Write Alphamerically	Paper Tape
K	TBTY	Tabulate	Typewriter
	RCTY	Return Carriage	Typewriter
	SPTY	Space	Typewriter

IBM 1620 Printed in U.S.A. J26-4201-2

IBM
International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, New York