IBM Customer Engineering

Manual of Instruction

7090 Advanced Programming

Issued to:_____

Department or Branch Office_____ Telephone Number_____

Address_____ City_____ State_____

Home Address_____ City_____ State_____

If this manual is mislaid, please notify the above address.

Printed in U.S.A.
Form 223-6860

CONTENTS

1.0.00 ADVANCED PROGRAMMING

1.1.00 INTRODUCTION

To make effective use of a system such as the 7090, there must be a means by which man can communicate with the machine rapidly and comprehensively. An easy way must be provided to instruct the machine as to the program steps necessary to solve the problem, a method of presenting the data to the machine and, finally, some means of compiling all the instructions and data in the proper sequence.

Two programming tools, symbolics and the assembly program, are provided to help the programmer accomplish the above steps.

1.2.00 SYMBOLIC PROGRAMMING

The most efficient use of the computer, from the standpoint of machine operation, lies in the use of machine language. This consists of writing a program using only elementary instructions, which would use machine vocabulary and machine addresses. The use of machine language represents most efficient usage since, under this condition, each instruction is doing work to perform the specified transformation of data.

Just as the system works most efficiently with machine language, so man works most efficiently in his own language. Machine language does not, in general, afford the programming ease that is provided with an abstract or symbolic coding system.

The programmer, when writing a lengthy program, soon has difficulty in keeping track of the instruction and data locations. A program written in machine language, when executed, depends on its location in core storage. If the program is to be relocated or insertions and deletions are to be made, the program itself would require alteration. It is desirable to remove from a program its dependence on its location in storage.

To overcome these difficulties, a method of writing programs has been evolved in which the programmer uses symbols to represent storage locations. Programs written in the symbolic form are readily changed and insertions and deletions do not affect other entries.

To write out in machine language the instruction to add the contents of location 3 to the accumulator would be very awkward. It would involve a binary representation as follows: 000100000000000000000000000000000011.

Mnemonics afford us a much easier way to write the instruction. Using mnemonics, it becomes CLA0003. It now has more meaning to a person reading it. However, to know the complete story, the person should know what is in location 3. To aid the programmer in this respect, the use of symbols may be extended to include the addresses as well as the operation codes. For example: CLA TAX. Now the programmer can not only see what operation is to be performed but also what quantity is involved. Programming in this manner becomes an instruction-by-instruction code in a familiar language.

By use of mnemonics and symbols when writing a program, the instructions may easily be punched in cards using decimal numbers and words or symbols.

Another process that would be long and involved in writing a program is keeping track of storage locations. Instead of trying to assign an absolute location to each instruction and piece of data, the programmer merely assigns symbols to a few locations and uses them as references. For example:

```
START    CLA    A
         ADD    B
         SUB    D
         STO    SAUIT
         LDQ    D
         MPY    SAUIT
         STO    SAUIT +1
         TRA    START +1
SAUIT    HLT
```

Using symbols, the programmer can easily delete, add, or correct instructions in the program without disrupting the program sequence. The program written in symbolics does not depend on absolute storage locations but merely on the symbolic references.

1.3.00 ASSEMBLY PROGRAM

Now that the use of symbols in program writing has been explained, there is a need to make the machine recognize these symbols and use them. The programmer submits his program to the key punch operator written in words or symbols and decimal numbers. The key punch operator then produces a deck, called a symbolic deck, punched in IBM code. The deck is then run into the 7090 system under control of an assembly program. The assembly program reads the coded information and translates it to binary machine language. The instructions are assembled in storage in the same order in which the cards are fed into the system. The final output is a detailed listing of the program and a deck punched in binary form. The final deck is called the object deck and the assembled program is called the object program.

The basic element of an assembly process is the assignment of absolute storage locations to the machine instructions. In the 7090, the assignment is made by simply giving consecutive storage locations to symbolic language instructions as they are read in. This function is under control of an element of the assembly program known as the location counter. This counter is given an initial value by the assembly program and, for each actual machine instruction encountered, it is advanced by one. Each instruction is assigned the value of the location counter at the time the instruction is first recognized by the program. Consider the following small program:

Card	Location Field	Operation	Variable Field
1	START	CLA	B
2		ADD	C
3		SUB	D
4		TRA	START +1
5	B	decimal number	
6	C	decimal number	
7	D	decimal number	

If the seven cards above were read into the system under control of an assembly program, the machine would, upon reading the first card, assign 0000 to START. The location 0000 would be the initial setting of the location counter. No absolute value would be assigned to B at this time. The location counter would then step to 0001 upon reading the next card, and 0001 would be assigned to ADD C. No location for C would be assigned at this time. This would continue until the fifth card is read. Upon reading the fifth card, the location counter is at 0004. Now B is assigned its absolute location (0004). Similarly, C and D are assigned an absolute location upon reading cards 6 and 7.

Notice that no symbol was given an absolute location until it appeared in the location field. Once a symbol appears in the location filed it is said to be a defined symbol.

Provided the storage capacity is not exceeded, every defined symbol in the deck is assigned a storage location. Symbols are limited to any six IBM characters, at least one of which is non-numerical. There are, however, a few exceptions. The symbols + - * / $, = , cannot be used in a symbolic term.

While the primary function of the assembly program is the assignment of absolute storage locations, assemblers usually provide additional information for the programmer. These include error print-outs, conversion routines, a means for calling in previously coded subroutines, and the ability to rename symbols in portions of a code. The latter allows the combining of independently written programs. The assembly also produces a symbolic and machine language listing of the programs.

1.4.00 SHARE

SHARE is a customer's organization whose primary function is to distribute among its members the programming developments of the individual members. SHARE is a completely voluntary organization. Through the realization that greater efficiency and more productivity could be attained through a cooperative effort, the SHARE membership has freely dispensed valuable programming techniques, talent, and a library of utility programs. In addition, SHARE has been able to assist IBM in its quest to provide better service to its customers. Its request for alterations or additions to existing equipment are well thought out and authoritative. The principal obligation of a SHARE member is to have a cooperative spirit.

For ease of communication, the SHARE body has established programming standards and conventions and procedural standards. Each program written is properly identified as to author, date and classification.

SHARE establishes a minimum system for use with its programs. If SHARE programs are written that utilize more than the minimum system, the additional features must be listed in the program description.

SHARE has distributed many valuable programs to its members. These programs are kept in a permanent library at each installation, generally on magnetic tape and cards. Programs fall into one of many general categories. To list a few: arithmetic, elementary and complex functions, differential and integral equations, input-output, sorting, and executive routines. With this wealth of reliable material, the organization of a master program is a matter of joining together the library subroutines, and those instructions and data that are unique to the particular program. Using the SHARE programs as basic building blocks and the ingenuity of the individual programmer as

mortar, the structure of the program takes shape. Certainly, as SHARE grows in scope and the distribution of more complex programs becomes more widespread, each programmer will have more time available for refining and originating his own programs.

1.5.00 SYMBOLIC LANGUAGE

The program written in symbolic language is a sequential series of instructions to the assembly program. The four principal parts of a symbolic instruction are recognized as the: location symbol, operation, variable field, and comment. The location field contains a symbol that represents either a storage location or other expression associated with the instruction. The operation determines the nature of the instruction and guides the interpretation of the various parts. The variable field adds the information necessary to complete the specifications of the instruction. The comment is not considered as far as the running of the program is concerned. Its sole function is to describe a remark intended to appear in the listing.

1.6.00 SYMBOLIC CARD FORMAT

The following format is the SHARE standard for punching symbolic cards:

Columns	
1-6	If an instruction has a location symbol, it appears in this field. For example, consider the following program:

```
START    CLA  A
         SUB  B
         ADD  C
FINISH   STO  D
```

For the CLA instruction, START would be punched in the location field. The next two instructions would have blank location fields. They would occupy the two absolute locations following START. If in the program it becomes necessary to transfer to the subtract instruction, the programmer can write TRA START +1 or TRA FINISH -2. The assembly program can perform + or - operations on the symbols.

Columns	
7	This column is always blank
8-14	This field is devoted to the operation code. It may be any three-to-seven-letter code.
11-72	This is the variable field of the card and contains the address, tag, decrement, and remarks. The starting column of the variable field is dependent on the operation code. One blank column must follow the operation code. The separate portions of the variable field are separated by commas. A blank column indicates the end of the variable field. The following instruction is written in proper form:

```
B   CLA   K, 1
```

This symbolic instruction states that CLA is located at symbolic location B. The contents of the effective address produced by subtracting the contents of index register A from K will be added to the accumulator. Both B and K will have absolute addresses assigned to them. B is punched in column 6, CLA is punched in columns 8, 9, and 10, K is punched in column 12, a comma is punched in column 13, and a one is punched in column 14. The lack of punching in column 15 indicates the end of the instruction.

Also, B TIX K, J+2 , 1 is a suitable form. This instruction states that TIX is located at address B. Transfer to location K if the contents of index register J+2 are greater than 1. B,K, and J are assigned absolute values by the assembly program. With J established, J+2 is also established.

If the variable field (12-72) is devoted to data, the punching may appear in either decimal or octal form. The data are assumed to be positive unless there is an 11 or 12 punch in the first column of the data field. This punch must appear alone in column 12 or it will be interpreted as a symbol. Several items may be punched in the same card as long as each item is separated by a comma. If, for example, a card contains START CLA 5,10026, 74, 35 the assembly assigns locations thus: 5 is located at location START, 10026 at START +1, 74 at START +2, and 35 at START +3.

2.0.00 SHARE SOS SYSTEM

2.1.00 INTRODUCTION

The SOS system is the automatic program designed to provide program control, programming aids for the programmer, and a means of debugging 709 and 7090 programs. The objective of the program is to provide a method for remaining in the symbolic language system efficiently throughout the preparation, verification, and execution of a program.

The SOS system may be broken down into a six-part program:

1. IB Monitor
2. SCAT
   a. Compiler
   b. Assembler-Translator
3. Debugging
4. I-O Translator
5. I-O Transmission
6. MockDonald Control System

2.2.00 IB MONITOR

The IB Monitor reads in the control cards preceding a symbolic deck and provides the control during the performance of the job. The IB monitor is loaded by the load tape procedure. Immediately after loading, a halt is executed. When the start key is depressed, the entry keys are entered and stored as the system data. The input and output tapes are rewound. The first job is read in and processed according to the control cards.

2.3.00 SCAT

SCAT's primary purpose is to aid the programmer in debugging his program. It reads in the symbolic deck, assembles the program by assigning storage locations to the instructions and data, and translates the symbolic information to machine language.

2.3.01 Compiler

The compiler portion of SCAT is the first program to act on the symbolic deck. It reads all of the symbolic information and stores it to be processed later by the assembler-translator.

The main task of the compiler is to set up the dictionary. The dictionary is the whole crux of the assembly process, and may be considered as a table of symbols. The dictionary contains all of the defined symbols in an ordered form. Each entry into the dictionary is coded with reference to the next location symbol in the program. This sets up the sequential flow of the program.

Symbols can all be considered to consist of six positions. Using standard BCD coding for characters would require 36 bits to express a symbol. By recognizing that there are only 49 characters plus a blank, it is possible to be somewhat more

conservative of bits than this. Any expression in the SCAT language may now be expressed as a number to the base 50. The numerical values of the characters are shown in Figure 2.3-1. The symbol is considered as two three-digit numbers to the base 50. The maximum number of bits required in this manner to express a six-character symbol is 34.

A symbol with the positions $S_1S_2S_3S_4S_5S_6$ is encoded as follows:

$$S_1S_2S_3S_4S_5S_6 = (S_1 \cdot 50^2 + S_2 \cdot 50 + S_3)\ 2^{17} + (S_4 \cdot 50^2 + S_2 \cdot 50 + S_6)$$

As an example, take the symbol ACFYDT and, by using the values in Figure 2.3-1, compute the value of this symbol:

$A = 11_{50}$ $\quad$ $Y = 35_{50}$
$C = 13_{50}$ $\quad$ $D = 14_{50}$
$F = 16_{50}$ $\quad$ $T = 30_{50}$

$$\begin{aligned}
ACFYDT = &(A \cdot 50^2 = C \cdot 50 + F)\ 2^{17} + (Y \cdot 50^2 + D \cdot 50 + T) \\
&(11 \cdot 50^2 + 13 \cdot 50 + 16)\ 2^{17} + (35 \cdot 50^2 + 14 \cdot 50 + 30) \\
&(27500 + 650 + 16)\ 2^{17} + (81 \cdot 500 + 700 + 30) \\
&(28166)\ 2^{17} + (88230) \\
&3691862182_{10} = 33403254246_8 \\
&0\ 011\ 011\ 100\ 000\ 011\ 010\ 101\ 100\ 010\ 100\ 110_2
\end{aligned}$$

	Character	Equivalent		Character	Equivalent
Digits	0	1		O	25
	1	2		P	26
	2	3		Q	27
	3	4		R	28
	4	5		S	29
	5	6		T	30
	6	7		U	31
	7	8		V	32
	8	9		W	33
	9	10		X	34
				Y	35
Alphabet	A	11		Z	36
	B	12			
	C	13	Special	-	37
	D	14		(	38
	E	15		)	39
	F	16		0(12-0)	40
	G	17		9(11-0)	41
	H	18		-(8-4)	42
	I	19			
	J	20	Punctuation	+	43
	K	21		-	44
	L	22		*	45
	M	23		/	46
	N	24		$	47
				,	48
				.	49

FIGURE 2.3-1. LEGITIMATE SCAT CHARACTERS AND THEIR BASE-50 EQUIVALENTS

The dictionary is put into order by sorting the symbols in their base 50 encoding. Because of the order in which characters are assigned the base 50 digits, this sorting technique puts the dictionary in alphabetic order.

The compiler also produces a squoze (squeezed) deck and an error listing. The squoze deck is a compacted, tightly encoded deck that is more conservative of bits than the SCAT language, and reduces the volume of cards to be handled. The squoze deck can then be easily used in further processing of the program. The two main reasons for obtaining and working with the squoze deck are:

1. Modifications can be made in the original symbolic language and simply added to the deck by use of a modify and load card.
2. All of the original symbolic information is retained during the program execution to permit the debugging program to give back printed output in the original symbolic language.

The squoze deck also results in a general reduction in read-in time and, consequently, the time to process a program input.

2.3.02 Squoze Deck

The squoze deck is composed of the following parts:

Part I	Preface
	Heading
	Macro name
	Blank card
	Macro skeleton
	Introduction
	Dictionary
	Footnotes
Part II	Text without commentary
Part III	Text with commentary

1. Either part II or III may be missing
2. Part III is necessary only when listing or when a new deck is punched.
3. If both parts II and III are present, the decoding will be done on part II sense switch settings.
4. Indicate to SCAT which text should be used.

Following is a description of the parts of the squoze deck:

1. The preface contains the name or other identification of the object program, the sizes of the other sections of the squoze deck, and the relative position of key items in the dictionary.
2. SCAT must know the names and skeletons of all programmer macro-instructions. This information is carried in the macro-table of the squoze deck, which is divided into two sections. The first section contains the name and size of the various programmer macros; the second, the skeletons, i.e., the actual instructions which constitute the macro-instruction.
3. The introduction contains a record of the generative pseudo-operations. These are operations that generate data words, instructions, or subroutines.
4. A heading table is included in the deck if any heading is used in the program.
5. The dictionary contains an entry for each symbol in a location field or variable field, and for each principal pseudo-operation. Each entry is of

fixed length and consists of two words. The entry gives the name of the symbol or pseudo-instruction; the entry number in the dictionary of the next symbol in the program; a count of the card containing the instruction; and, in the case of pseudo-instructions, the location of the footnote entry.

6. There is a footnote entry for each principal pseudo-operation. The footnote defines the variable field expressions of these pseudo-operations.
7. The text is the source program in a compact or squoze form. For example, the operation code of machine instructions is expressed in a 5- or 9-bit code; the variable field of machine instructions is expressed in terms of the dictionary entries of the symbols involved. The text is in symbolic language and, for many instructions, requires less space than would actual machine language. This squoze text along with the other information in the squoze deck enables modifications written in symbolic language to be incorporated into the program being executed.

   In order to execute the source program, the text is translated into actual machine language with all references to the dictionary being replaced by the absolute numerical values assigned to the symbols, and all variable field expressions being replaced by their numerical equivalences.

Although it is the option of the assembly to produce an absolute binary program deck, the basic structure of the SOS system assumes that normal operation is to obtain a squoze deck. A squoze deck cannot be run into the machine unprocessed. It is the result of the first pass of the assembly operation. To obtain an executable deck, it is necessary to perform the second pass. This is done by the loader. The loader allows the programmer to make changes and on request can produce a new squoze deck or an absolute binary deck, as well as a listing of the modified program.

As the squoze deck is compacted and contains more than one instruction per card, it is not possible to specify modifications in the same way as with symbolic cards. One cannot insert and delete instruction cards by simply shifting the individual cards. Thus, special modification specification cards are required. In general, these cards specify whether to insert or delete, where to do this, and, if deleting, how much to delete. Cards to be inserted, punched in the normal symbolic format, follow a modification specification card. For convenience, these modification specification cards are given a format identical with the instruction format of the symbolic language; thus, they may be considered as a special type of instruction: CHANGE, ALTER, ERASE, and ASSIGN. These instructions are discussed, in detail, in the SHARE 709 System (SOS) Manual, Part II, Modify and Load.

2.4.00 DEBUGGING

The debugging routine provides a set of instructions, which can be inserted in a program so that, at selected points in its execution, selected machine status information is saved for future use. The execution of the program then continues without interruption.

2.5.00 I-O TRANSLATION

I-O translation provides an easy and economical method to enter data for this program and to put out the results. The programmer can specify the format he wishes to use for his input and output.

2.6.00 I-O TRANSMISSION

I-O transmission provides a language in which a programmer can optimally arrange for parallel operations of input, output, and computing to achieve minimum problem time on the system.

2.7.00 MOCKDONALD SYSTEM

The MockDonald operating system accepts as input a group of jobs stored, each with its own input data, on a peripheral input tape. The MockDonald system has control of the 7090 between jobs and automatically loads the next job from the input tape. The MockDonald system provides certain functions for the program in the areas of input, output, and execution. Some of the functions are:

1. Routines to help the programmer utilize the several trapping features of the 7090
2. Tape handling routines
3. Debugging routines
4. Input and output conversion routines
5. Communication with the operator

Communication between the program and the MockDonald system is done by means of a communication region. Entrance to system routines, and symbolic assignment of input-output units, is made through the communication region.

The entire group of jobs on the input tape is processed in one or more successive phases. The number of phases needed to complete any particular job depends upon which method of input and output editing the job uses.

All peripheral input or output tapes, and binary mediary tapes which transfer jobs and data from phase to phase, are read or written through system subroutines. Final output accumulates on one or more peripheral output tapes.

The types of jobs that the MockDonald system can handle are:

Type E: These jobs are processed in one phase, the Execution phase. The customer's executing program uses conversion subroutines directly to process the data on the peripheral tape.

Type I-E: These jobs are processed in two phases, the Input and Execution phases. Data are converted automatically from BCD to binary and written on a binary tape during the input phase. During the execution phase, the customer's program obtains its data from the binary tape just as if it were reading the peripheral input tape. The only difference is that the conversion has already been accomplished.

Type E-O: These jobs are processed in two phases, Execution and Output. Input is accomplished the same as in type E jobs. During the execution phase it writes the information to be used as output on a binary tape using special system subroutines for this purpose. In the output phase a routine automatically converts the binary to BCD and writes the BCD data onto the peripheral output tape.

Type I-E-O: These jobs are processed in three phases, Input, Execution, and Output. BCD data are converted in the input phase and read from a binary tape during the execution phase in the same manner as in the type I-E jobs. The output data are

written on a binary tape during the execution phase and converted to BCD during the output phase in the same manner as in the type E-O jobs.

The MockDonald system can accept an input tape containing one or any combination of the above type jobs.

The various types of jobs are referred to by their configuration of input, execution, and output, using the initial letters, E, I-E, E-O, I-E-O. This reminded one of the legendary Scottish farmer, and the system was forthwith named the MockDonald system after him and one of the designers of the system, Owen Mock.

The operation of the system is continuous and automatic between jobs and requires a minimum of operator intervention. The supervisory control routine is the basic control routine which is always reverted to between jobs. It performs certain initialization functions to start each job and certain conclusion functions to conclude the job. It brings into core storage the system routines that are necessary to whatever phase the job is in, I, E, or O.

2.8.00 SCAT LANGUAGE

The SCAT language is the heart of the whole SOS system in that it contains all of the controls essential to assemble a program written in symbolic language. The language used in conjunction with SCAT is a basis for the language used throughout the SOS system. Therefore, a good understanding of SCAT leads to a more comprehensive understanding of the remaining sections of SOS.

2.8.01 Symbolic Language and Arithmetic Expressions

The basic units of the symbolic language are symbols, numbers, and operation codes. These units may be combined by punctuation marks, according to certain rules, to field expressions.

A symbol is a combination of from one to six IBM code characters, at least one of which is non-numerical and none of which is punctuation; i.e., a symbol may not contain a plus, dash, asterisk, slash, dollar sign, equal sign, comma, or an imbedded blank. A blank is not considered a character in this case. A symbol is defined if and only if it appears in the location field of some instruction; otherwise, it is undefined. It is desirable to label a symbolic instruction with a location symbol only if it is necessary to refer to this instruction in the program. An absolute location symbol (i. e., one containing no non-numerical characters) is flagged as an error and is ignored.

A number is a combination of digits which may be decimal or octal depending upon its context. An operation code may consist of from one to six characters, all alphabetic. An expression is a combination of symbols and integers separated by the following connectors or punctuation marks: + (addition), - (subtraction), * (multiplication),and/or /(division).

An expression can occur only in the variable field of an instruction and it can be one of three kinds: simple, relative, and complex. A simple expression is a single symbol or number without any punctuation. A relative expression is a symbol plus or minus a number. All other expressions are complex. An expression is terminated by a comma or a blank.

2.8.02 Special Characters

The asterisk character (*) has five different meanings in SCAT, depending upon its context. As a punch in column 1 of a card it defines the card as a remark. If it is found immediately after an operation code, it specifies indirect addressing. As a connector in a variable field expression it connotes multiplication. As a Boolean operator it specifies intersection, e.g., the logical AND process. Finally, if it occurs immediately after another connector or as the first character in a variable field, it is recognized as a term. In this context it is interpreted as having the current value of the location counter.

The character $ may be preceded by a numerical, alphabetic, or special character, or it may commence a term followed by five or fewer characters in an expression. These arrangements cause SCAT to head the symbol with the given character rather than the current heading character. Reference from a headed region to an unheaded symbol is made by preceding the $ with a 0 or no heading character. (The significance of the character $ is explained further under the pseudo-instruction HEAD in the SHARE System SOS Manual.)

2.8.03 Pseudo-Operations

Every operation in the SOS system belongs to one of two classes. It is either a 7090 machine operation (CLA, ADD, LXD, and so on) or else it is a non-machine operation. A non-machine operation is called a pseudo-operation. Instructions used to perform pseudo-operations are called pseudo-instructions.

A machine instruction always generates one 36-bit binary machine word in the object program. A pseudo-instruction, however, may generate more than one word in the object program, or it may not generate any words at all. Not all of the pseudo-operations are covered in this manual. A few are covered in order to give a general idea of the types of operations pseudo-operations can perform. A complete and detailed coverage of SCAT may be found in the SHARE System (SOS) Manual.

ORG (Origin)

As the basic function of an assembly process is to assign absolute storage locations to machine instructions, there must be an address at which this assignment begins. In SCAT, this value is furnished to the assembly program by the program being assembled via the ORG pseudo-instruction. ORG sets the location counter to the same integer value as that computed for its variable field. A location symbol associated with an ORG instruction is also assigned the computer value of the variable field.

Symbol Location	Operation	A,T,D,
A	ORG	100

In the above example, ORG assigns a value of 100 to both the symbol A and the location counter. The location counter determines the storage location to which subsequent instructions are assigned. The first instruction following the ORG card is assigned the location of the variable field value of the ORG card.

ORG instructions may occur anywhere in a program and the variable field of these instructions may be any combination of numbers and symbols acceptable to SCAT.

A symbol appearing in the variable field expression need not have been previously defined, i.e., need not have appeared in the location field, columns 1-6, of some previous instruction or pseudo-instruction. However, a symbol in the expression which is not eventually defined in the program renders the variable field of ORG non-computable. If the program being assembled does not have an ORG pseudo-instruction, load-and-go sets the location counter to the lowest location in memory not required by the SCAT system.

BSS (Block Started by Symbol)

This pseudo-instruction is used to reserve a block of storage whenever the program being assembled demands it. The block reserved is equal in length to the value of the variable field expression. The associated location symbol is given the value that the location counter has when it encounters the BSS, and corresponds, therefore, to the first word of the block reserved.

Location Counter	Symbol Location	Operation	A,T,D,
250	A	BSS	200
450	B	XXX	XXX

In the above example the BSS instruction reserves the 200 memory positions from locations 250 to 449, inclusive. The associated location symbol A is assigned the value 250 and the location counter is set to 450.

The rules for previous definition of symbols are the same as for the ORG pseudo-instruction. A BSS can occur anywhere in a program.

BES (Block Ended by Symbol)

This pseudo-instruction is also used to reserve a block of storage at the direction of the program being assembled. A BES is the same as a BSS in every respect except for its result upon the associated location symbol. This symbol is given the value of the location counter plus the variable field value and corresponds, therefore, to the first word following the block reserved.

Location Counter	Symbol Location	Operation	A,T,D,
250/450	A	BES	200

In the above example, the BES instruction reserves the 200 memory positions from locations 250 to 449 inclusive. The location counter, which is at location 250 when it encounters the BES, is reset to 450. The associated location symbol A is assigned the value 450.

The rules for previous definition of symbols are the same as for the ORG pseudo-instruction. A BES may occur anywhere in a program.

TCD (Transfer Card)

The purpose of this pseudo-instruction is to produce control information which directs the loading program to execute a transfer of control from the loading program itself to the program being loaded. The transfer is made to the storage location represented by

the value of the variable field expression of the TCD instruction. There can be more than one TCD instruction and they can appear anywhere in a program.

If a TCD has an associated location symbol, the symbol is assigned the value that the location counter has when it encounters the TCD instruction.

Location Counter	Symbol Location	Operation	A, T, D
200	A	TCD	2500

The above instruction sets A equal to 200 and transfer of control is made to location 2500.

END (End)

Since, as previously explained with the ORG pseudo-instruction, the computer must know where to start assigning absolute storage locations to machine instructions, it must also know when to stop this process. In SCAT, the termination of the assembly and loading operations is indicated by the END pseudo-instruction. It must appear in every program and it must be the last instruction read during the assembly process.

As in the case with a TCD, the END instruction causes a transfer of control to be made to the storage location represented by the value of the variable field expression. The rules governing the associated location symbol if there is one, are the same as for TCD.

Location Counter	Symbol Location	Operation	A, T, D,
800	A	END	1000

The above instruction sets A equal to 800 and transfer of control is made to location 1000.

2.8.04 Macro-Operations and Instructions

A special type of pseudo-operation, called a macro-operation is another feature of the SOS system. Macro, a combining term meaning long in extent, is a clue as to the type of operations these are. The most significant property of a macro-operation is that it generates N machine words, where N is greater than or equal to 1. Ordinarily, in a macro-operation, N is greater than 1.

A macro-operation is regarded as an abbreviation for a block of instructions. The block of instructions generated is determined by the particular macro-operation. Each macro-operation has its own definition, consisting of a skeletal pattern of instructions.

There are two classes of macro-operations. The first class consists of macro-instructions which the programmer can arbitrarily define. The second class is a large group of permanently defined macro-operations. The first class is called programmer macros, and the second, systems macros.

Suppose that a programmer writes a source program with the following type of structure:

```
CLA A
ADD B
STO C

CLA ALPHA
ADD YUMA
STO WAHOO

CLA Z
ADD M
STO P
```

Once the programmer realizes that the pattern of three instructions appears several times in his program, he can make a programmer's macro of them and insert them in the program with only one card containing the macro name. The only prerequisite is that he define the macro-operation in his program before he uses it. For example, call the three instructions used in the above program QSUM. They can be defined in the program in this way.

Location	Operation	Variable Field
QSUM	MACRO	V1,V2,V3
	CLA	V1
	ADD	V2
	STO	V3
	END	

QSUM is not a location in this instance, but a macro-operation name. V1, 2, and 3 define which piece of data in the variable field is to be associated with each following instruction. The first is associated with CLA, the second with ADD, and so on. CLA, ADD, and STO form the macro skeleton. This sequence generates no words in the object program, but merely defines the programmer macro called QSUM.

Now that the macro-operation is defined, the programmer may obtain this sequence in his program by merely giving the instruction:

Location	Operation	Variable Field
	QSUM	A, B, C

This will generate three words in his object program: CLA A, ADD B STO C.

An example of a system's macro is an operation named CORE. If given in a program, this operation expands into the instructions necessary to write information from storage onto tape. The variable field defines what portion of storage the programmer wishes to print. For example:

Location	Operation	Variable Field
	CORE	0,2000

The above macro-operation generates the following instructions:

Location	Operation	Variable Field
A	STO	X
	STL	LOC
	TXL	STPAN, 0, 7
	PZE	
	PZE	
B	CLA	Y

Upon reaching this point in the program, the registers are stored to retain their condition and the location in the program is saved. The machine prints out storage from locations 0-2000 and then returns to the program and continues execution.

2.9.00 INPUT DECKS

The programmer may introduce any one of three possible job decks to the SOS system. The names of the job decks, their make-up, and their functions are as follows:

Compilation Job Deck: This deck is the symbolic deck with three control cards. Compilation takes place on the first pass with SCAT and provides all relevant information to the machine to set up the program. The results of processing this deck are an error listing and a squoze deck. The deck consists of the following cards in sequential order:

1. Card punched JOB in columns 8-10 and any identification in columns 16-72
2. Card punched CPLRB in columns 8-12. If column binary is used, punch CPL in columns 8-10.
3. Symbolic deck
4. Card punched END in columns 8-10 and the starting point of the program punched in columns 12-72

Listing Job Deck: This deck is made up of the squoze deck and three control cards. When processed with SCAT, it produces a symbolic and machine language listing of the program. The deck consists of the following cards in order:

1. JOB card
2. Card punched LS in columns 8-9
3. Squoze deck
4. Blank card

Execution Job Deck: This deck consists of the squoze deck and four control cards. When read into the machine under control of SCAT, it causes execution of the program. The deck make up is as follows:

1. JOB card
2. Card punched LG in columns 8-9
3. Squoze deck
4. Blank card
5. Card punched GO in columns 8-9

There are several options that may be used with either of the three decks. The options are chosen by sense switch settings. The decks just covered are of the general format used. There are variations that produce an absolute binary deck, a new squoze deck, and so on. All of these variations are covered in the SOS System Operational Bulletins.

Figure 2.9-1 is an example of the processing of a symbolic deck from compilation to execution.

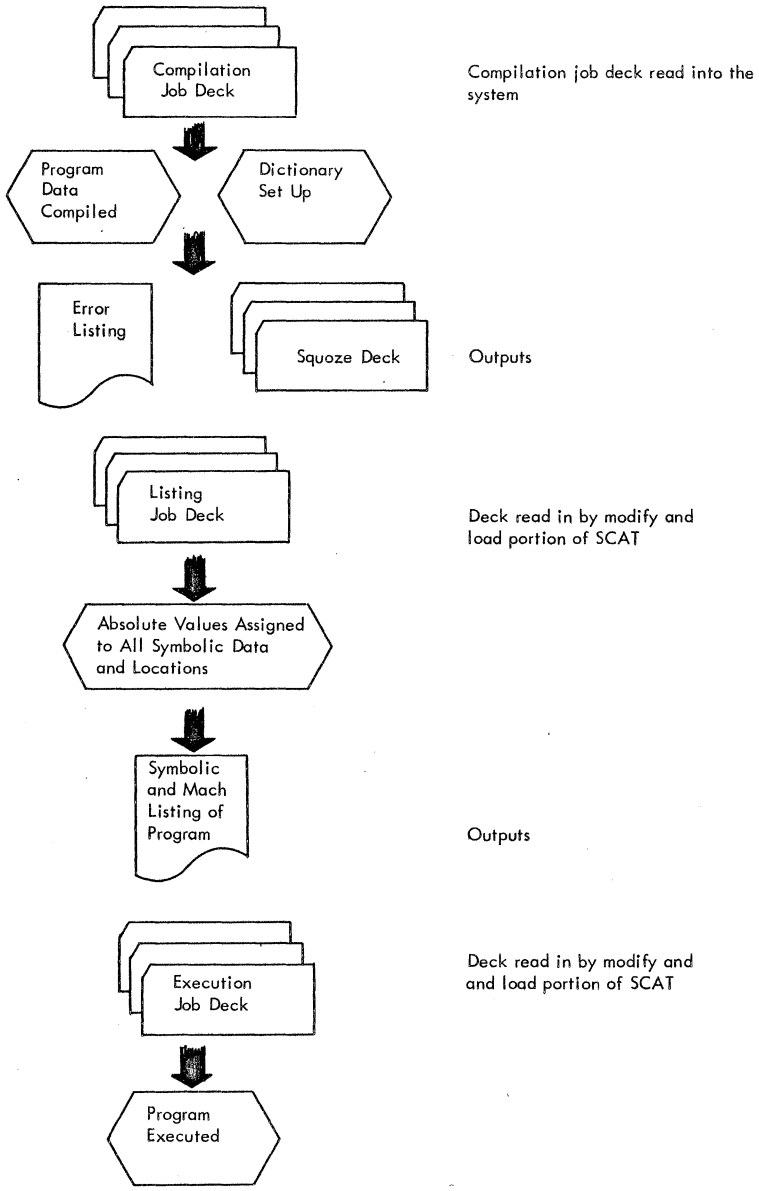

FIGURE 2.9-1. PROGRAM ASSEMBLY USING SOS

3.0.00 FORTRAN

3.1.00 INTRODUCTION

The Fortran system was developed to enable the programmer to write a program in a language similar to his own and obtain automatically an efficient machine program to carry out the procedure.

The 18 man years needed to develop this program have resulted in reducing the programmer's task to approximately one-fifth the task it had been.

Fortran is a two-part system. It consists of the Fortran language and the Fortran translator. The translator is commonly referred to as the executive routine.

The Fortran language is a concise language, mathematical in background, employing familiar symbols which are readily translated to machine language. The language consists of 32 statements, many of which resemble algebraic formulas. The executive routine translates these statements into binary machine language; thus the name FORTRAN--FORmula TRANslation.

3.2.00 TRANSLATOR

The complete Fortran translator program is written on the Fortran system tape. The program is written and executed in six sections. What each section accomplishes can be seen by tracing a program as it is assembled by the Fortran system.

The program steps are all listed in longhand, using the Fortran language. Each statement is then punched in an individual card. These cards form the Fortran source deck. The source deck is read into the machine under control of the Fortran system tape, section 1. The information from the cards is transmitted onto tape 2 in BCD. If tape input were used, the source program would originally be placed on tape 2 in BCD.

The translator now proceeds to code all of the statements in the source program. Every statement receives a code number called the internal formula number (IFN). These numbers are assigned sequentially starting with 1. All future references to the original statements are made using the IFN as identification. This scanning of information of the BCD file on tape 2 occurs only once. All information in this file is coded as it is read.

The input statements are then classified as arithmetic or non-arithmetic. The non-arithmetic statements are stored in a temporay buffer area in core storage. This buffer area is in lower storage and the buffer units are ten words long. When the buffers are full, the information is transposed to tape 4.

The arithmetic instructions are scanned, checked for mode, and translated into machine language. The result of this analysis is a compiled instruction table (CIT) stored temporarily on tape 3. Once the CIT file is complete, it is placed in a record called COMPAIL (complete arithmetic, input-output, and logical). The COMPAIL file is on tape 2.

Fortran now proceeds to section 2. This section arranges the non-arithmetic instructions associated with indexing in a COMPDO file. Because the program assumes the machine has many index registers, indexing loops are set up.

In section 3, the COMPDO and COMPAIL files are merged into a single file. At the same time, the rest of the non-arithmetic instructions are translated to machine language instructions. At this point, the object program is complete but assumes the machine has many index registers.

Section 4 performs an analysis of the program flow, arranging and re-arranging it to obtain optimum operation using only the three available index registers. The object program may be run several hundred times in this section.

The last section assembles the program and produces a machine language program on cards or tape ready for execution.

3.3.00 FORTRAN LANGUAGE

The Fortran language consists of 32 statements broken down into the following categories:

1. Arithmetic formulas which permit the object program to carry out a numerical operation
2. Control statements which govern the flow of the object program
3. I-O statements which provide for the necessary I-O functions
4. Three specification statements which provide various information required or desirable to make the program efficient

3.3.01 Arithmetic Operations

The Fortran arithmetic statements look exactly like a simple statement of equality (i.e., $A = 3^X$). The right side of all arithmetic statements is an expression which may involve parentheses, operation symbols, constants, variables, or functions, in accordance with a set of rules similar to those of ordinary algebra.

The symbols for the five basic arithmetic operations used in Fortran are:

(+) A + B means add B to A.
(-) A - B means subtract B from A.
(*) A * B means multiply A by B.
(/) A/B means divide A by B.
(**) A**B means exponentiation, A^B.

The equation A = (B*C) + D/E - 3 is an equation in Fortran language, or a Fortran arithmetic statement. With Fortran, the programmer has the ability to express variables (A), operations (A*B, D/E), and constants (3).

It is also possible to express functions. The number of functions possible is very large, and varies from computing center to computing center. Each center has its own list with information concerning the use of the functions. Some of the typical functions and their use are:

Fortran Symbol	Function
SQRTF (X)	X
SINF (X)	SIN X
ARCTANF (F)	ARCTAN X

One of the roots of the quadratic equation $3X^2 + 1 \cdot 7\ X - 31 \cdot 92 = 0$ can be found by the equation:

$$ROOT = \frac{-B + B^2 - 4AC}{2A}$$

Where $A = +3$,
$B = +1 \cdot 7$, and
$C = -31.92$

The four statements necessary to program this problem for solution using the Fortran system are:

```
A = 3
B = 1.7
C = 31.92

ROOT = (-B + SQTF(B**2. -4. *A*C))/(2. *A)
```

The statements are punched one to a card. The first statement assigns the value 3 to A; the next two statements have similar meanings. The fourth statement means to evaluate the expression on the right side and assign the result to ROOT.

The computer executes the statements in the order in which they are introduced to the machine. For example, if the fourth statement were placed first, the machine would evaluate ROOT first. Since the values of A, B, and C would not be known as yet, it would solve the equation using arbitrary, unknown values.

Notice that no symbolic locations are expressed in the Fortran program. Using Fortran, the programmer has no control over the symbolic addressing, an advantage in that the programmer need not concern himself with keeping track of the symbols. Fortran causes the machine to manufacture its own symbols for core storage addresses. These symbols appear in the symbolic listing of the assembled Fortran program.

3.3.02 Fortran Card Format

Each statement of the Fortran source program is punched in a separate card. If a single statement is too long to fit on a single card under the card layout system specified below, it may continue over as many as nine continuation cards.

3.3.03 Control Statements

The machine instruction TRA N may be literally interpreted as "go to instruction N and proceed from there." The Fortran language has a control statement that makes use of this literal translation. It is a "GO TO" statement. The statement "GO TO N" transfers control to statement N and execution proceeds from there.

Another type of transfer statement is the "IF" statement. IF statements are of the general form: IF (E) N_1, N_2, N_3. This statement may be considered as a 3-way GO TO statement. E may be any arithmetic expression (i.e., A-G, X+D, and so on). N is the statement the machine transfers to if E is negative; if E is zero it transfers to N_2; and if E is positive it transfers to N_3. To illustrate the use of these statements, consider the following problem:

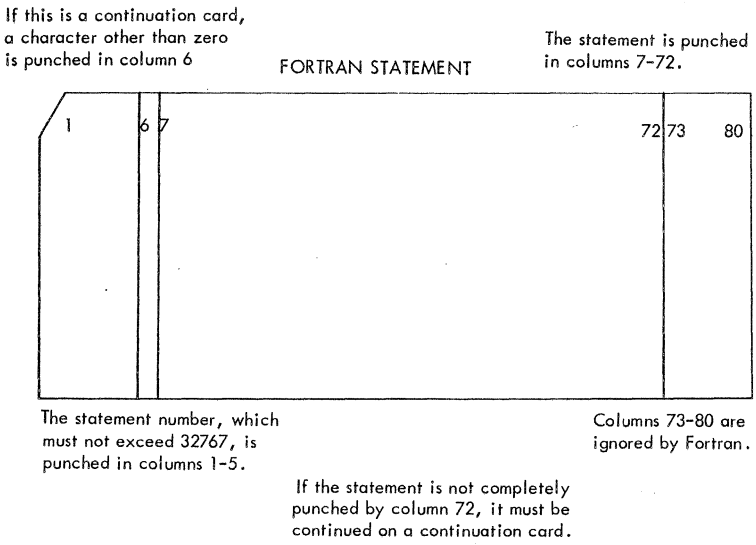

FIGURE 3.3-1 FORTRAN SOURCE CARD

Given: Values A, B, C and D punched in a card

Problem: Solve (A+B) C. If the result equals D, print the answer. If it is less than D, stop the machine.

Statement Number	Statement
10	READ 1, A, B, C, D
11	X = (A + B)*C
12	IF (X-D) 13, 14, 15
13	GO TO 16
14	PRINT 1, X
15	GO TO 16
16	STOP

The first statement reads in the values of A, B, C, and D. Statement 11 solves the equation for X. Statement 12 checks the answer against D and transfers accordingly. If X is less than or greater than D, the machine transfers to a GO TO statement that transfers to a machine STOP. If X equals zero, it transfers to statement 14, which prints the value of X. The statement following the PRINT causes the machine to transfer to a machine STOP.

3.3.04 I-O Statements

In a Fortran program, the I-O statements make possible the transmission of information between storage and I-O devices. These statements may be grouped as follows:

1. READ, PUNCH, PRINT, READ INPUT TAPE, and WRITE OUTPUT TAPE, all of which call for the transmission of a list of quantities.
2. FORMAT, a non-executed statement which defines the information format to be used (fixed or floating point, and so on).
3. READ TAPE and WRITE TAPE, both of which are used only for transmission of binary quantities.
4. END FILE, REWIND, and BACKSPACE are all used for the manipulation of tapes.

Each of the statements calling for the transmission of information takes the following form: Statement, N, List. The statement signals the computer as to which I-O device to read or write. N is the statement number of a FORMAT statement, which defines the type and layout of the information. The last portion lists the quantities to be transmitted.

READ 1, A, B, C is a typical I-O statement. Interpreted, it means "read three quantities from the card in the reader; assign A to the first, B to the second, and so on." The quantities are in a format in accordance with statement 1.

The FORMAT statements that define the input or output format use one or any combination of the following three forms: Iw, Ew, d, Fw,d. The first form indicates an integer decimal number having a field width of w columns. The second format, Ew.d, indicates a floating decimal point number E, having a field width of w columns, and d places to the right of the decimal point. The third format, Fw.d, indicates a fixed decimal point number, having a field width of w columns, and d places to the right of the decimal point.

For example:

```
1    FORMAT (E10. A, F8.3, I3)
     READ  1, A, B, C
```

could be used to read the following input data from cards:

Field 1	Field 2	Field 3
+.8765E06	+345.648	+81

Field 1 is the decimal number 10^6x. 8765; fields 2 and 3 are decimal numbers in their familiar form. The FORMAT statement is not executed and may appear anywhere in the program.