CAROLE GRIFFITH

S-C 4060
STORED PROGRAM RECORDING SYSTEM

SOFTWARE DESCRIPTION
AND
SPECIFICATIONS

9500236

Stromberg-Carlson A Subsidiary of General Dynamics

1895 HANCOCK ST. P.O. BOX 2449, SAN DIEGO, CALIF. 92112 TEL.(714) 298–8331

Revision A
9500236

S-C 4060

STORED PROGRAM RECORDING SYSTEM

SOFTWARE DESCRIPTION

and

SPECIFICATIONS

Prepared ______________________ Approved ______________________

R. C. Foster   J. J. Konen, Jr.

Checked ______________________ ______________________

Stromberg-Carlson A Subsidiary of General Dynamics

1895 HANCOCK ST. P.O. BOX 2449, SAN DIEGO, CALIF. 92112 TEL.(714) 298-8331

ABSTRACT

This document describes the software packages which are delivered with the Stromberg-Carlson 4060 Microfilm Plotter-Printer Data Recording System.

TABLE OF CONTENTS

FIGURES

APPENDICES

I. INTRODUCTION

A. Primary Software Packages

The two primary software packages which are delivered with the S-C 4060 Microfilm Recorder Plotter-Printer are the Integrated Graphic Software (IGS) package and the Stromberg-Carlson Recorder Input Processing (SCRIP) package.

IGS was developed by the RAND Corporation for the S-C 4060. It is a high level (Fortran language) graphic application oriented software package, which when run on a general purpose computer produces output in the form of a meta-language (see Appendix A.)

SCRIP, developed by Stromberg-Carlson, is a low level (machine language) package of routines which controls the operation of the S-C 4060; in particular, the SCRIP system contains the necessary routines to control the input of the meta-language (either off-line or on-line) output by IGS, and supervise the processing of the metalanguage into commands which cause the S-C 4060 to generate the desired graphical and/or printed output. SCRIP routines also monitor the film and hard copy control functions and provide additional input processing capabilities as specified in Section III.

B. Standard Operating Environment

The standard operating environment of the S-C 4060 is defined as one in which a remote (external to the S-C 4060) general purpose computer uses IGS to produce the input (off-line, or optionally on-line) to the S-C 4060. This environment (see Figure 1), also referred to as an External System Configuration, is fully supported by the standard SCRIP software package. This system has the following advantages:

1. Simplified Programming - any desired display may be produced by using IGS subroutines either singly or in combination.

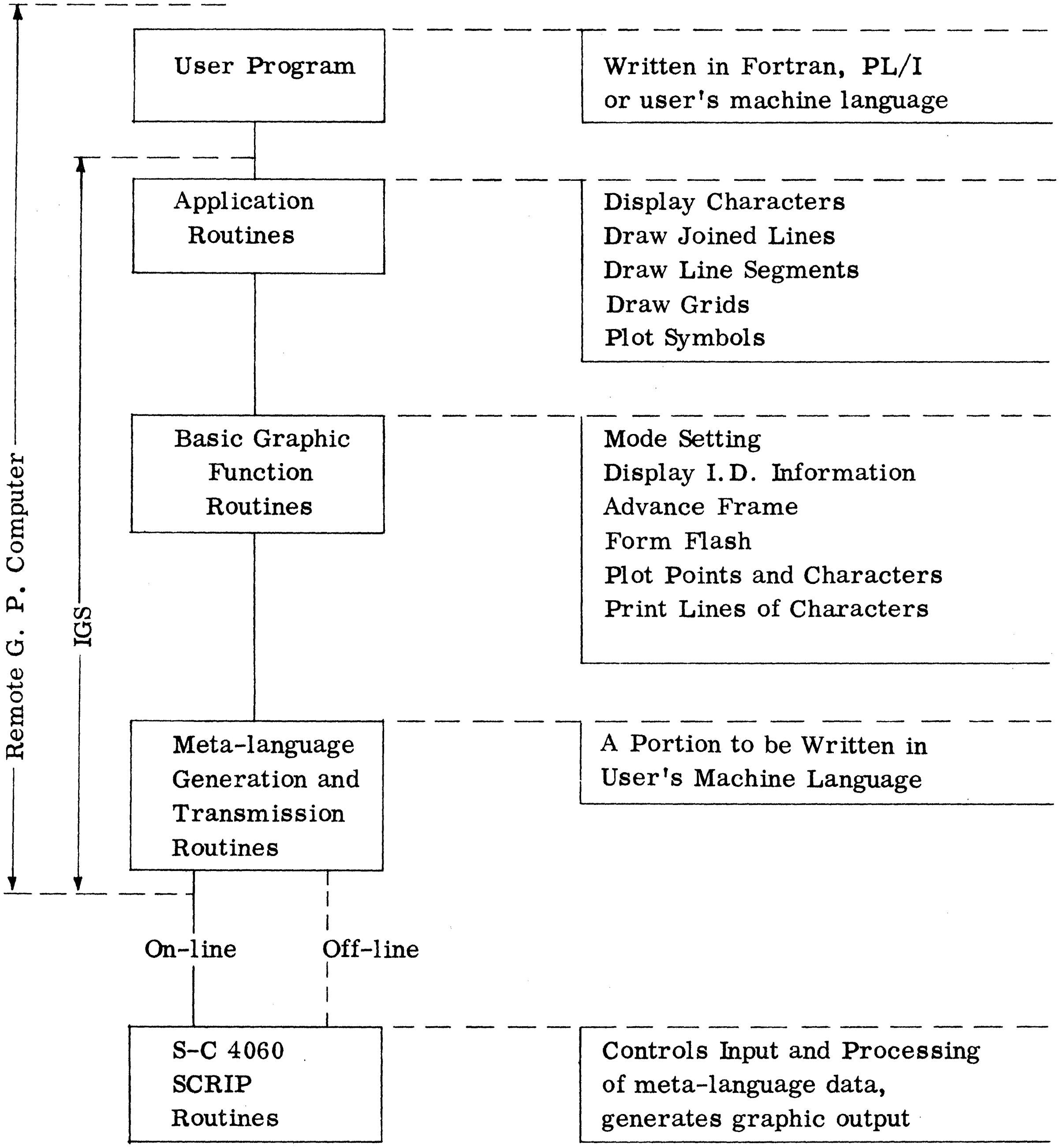

Remote G. P. Computer
IGS
User Program
Written in Fortran, PL/I or user's machine language
Application Routines
Display Characters
Draw Joined Lines
Draw Line Segments
Draw Grids
Plot Symbols
Basic Graphic Function Routines
Mode Setting
Display I.D. Information
Advance Frame
Form Flash
Plot Points and Characters
Print Lines of Characters
Meta-language Generation and Transmission Routines
A Portion to be Written in User's Machine Language
On-line
Off-line
S-C 4060 SCRIP Routines
Controls Input and Processing of meta-language data, generates graphic output

Figure 1. S-C 4060 Standard Operating Environment

2. S-C 4020/SCORS Compatibility - SCORS programs can be used in this environment.

3. Software Control Flexibility - software control of the S-C 4060 permits special user requirements to be incorporated into the system without the necessity of expensive equipment modification; only the SCRIP package need be extended. Thus, if the user wanted to implement any other generalized output graphic language from a remote general purpose computer he could add an appropriate non-standard SCRIP processor.

C. Supplementary Software Packages

In addition to the IGS package and the SCRIP package, the following supplementary software packages are delivered with the S-C 4060:

1. DAP-16 Assembler - a symbolic assembly language package which will enable the user to write routines to be added to the SCRIP software (which is written in DAP-16 - see Appendix F).

2. Fortran IV Compiler - will process the standard Fortran language specified by the American Standards Association (ASA - see Appendix G) and permit the use of the S-C 4060 as a "stand-alone" computer for small problems when it is not fulfilling its prime function - the recording and output of graphical and/or printed displays. A library of standard math and utility routines is included.

3. An EDITOR - to create or update the SCRIP system tape thus allowing the user to tailor the system to his own needs.

4. Utility Routines - Checkout and debugging aid; core dump routines to ASR-33, printhead, paper tape, and magnetic tape; card to tape program; punch program (for self-loading object tapes); one and two pass loaders; tape dump and copy program; cold start and library bootstraps.

II. EXTERNAL COMPUTER SOFTWARE - IGS

A. Background

In order to enable the programmer to utilize the S-C 4060 as a graphic output device effectively and efficiently, a graphic software package was developed to produce the input for the S-C 4060. This package, called Integrated Graphic Software (IGS), was designed after a careful study of the S-C 4020 SCORS package and other graphic output packages, in particular the SHARE committee report on Standard Graphic Output Subroutines.

S-C 4060 Users familiar with the SCORS package will note many differences between it and the IGS package. Since the SCORS package was originally designed for an S-C 4020/IBM 7090 combination, it reflects the hardware limitations of each. IGS was designed to provide for the powerful capabilities of the S-C 4060, and to eliminate as many of the previous limitations as possible, to be more efficient, and be easier to understand. Thus special features of the S-C 4060, such as automatic character rotation, line width, stroke characters, dashed lines, etc., and control features such as frame butt, on-line processing, expose hardcopy, on-line messages, void frames, etc., are programmable. IGS also was designed to take advantage of the new generation of computer hardware and software; it will operate in a time sharing environment, and is callable from PL/I. In addition to the above mentioned features, SCORS compatibility is maintained.

The IGS package will be given to the user in source form and is intended to be added to the system library of his computer. With the exception of the packing, transmission (output), and conversion routines, the system is written in Fortran and is thus "computer independent." The

packing, transmission, and conversion routines may require machine dependent subroutines which will normally be supplied by the user.

B. Description

The Integrated Graphic Software package is a computer-independent set of subroutines designed to remove the user from the hardware considerations of the S-C 4060. The subroutines are written in Fortran, but may be called from PL/I, Fortran or the computer's Assembly Language. When called by the user the individual subroutines supply the details necessary to perform the desired graphic functions (e.g., construct a grid, draw a line, plot a character, etc.). These details principally consist of structuring, formatting, and outputting (either on-line or onto a magnetic tape) a position keyed, character string meta-language for input to the S-C 4060 (see Appendix A). Thus, the user is allowed to think in terms of his displays rather than in terms of the hardware features which actually create the displays.

The heart of the IGS package consists of five subroutines which display characters, draw joined lines, draw line segments, plot symbols, and draw grids (linear, non-linear). The subroutines are easy to use, easy to understand, and place as few restrictions on the user as possible.

C. Special Features

1. Automatic Scaling - the user may set up his own coordinate system, and an IGS subroutine computes the scale factors needed and automatically scales the data.

2. Default Values - A powerful feature of IGS is the ability to assume appropriate default values for items the user chooses to ignore. For example, if the user doesn't specify the plotting area he desires on the tube face, the system will assume it to be the full 4096 x 3072 raster size of the S-C 4060 (Appendix B describes the S-C 4060 Standard Raster).

3. Mode Set Array - One of the most important concepts in IGS is the use of a Mode Set array. This array contains all of the information needed about both the user's and the installation's display environment. When the IGS system is initialized or reset, all the appropriate default values are stored in the array. They remain in effect until the user specifically changes them (see Appendix C-II).

   A major advantage of the Mode Set concept is that it allows the user to specify a minimum number of parameters in calling the graphic subroutines. Consider a subroutine to display characters: In its minimal form, the user only need specify an X, Y location of the first character, the number of characters, and the characters themselves. In its most complex form, the user might want to specify character size, character orientation, character spacing, right and left margins, line spacing, and line orientation.

   It would be unreasonable to expect a user to specify all of these parameters each time he wanted to display a line of characters. Therefore, the call to the character subroutine requires only the minimum information needed to display characters. All the other parameters are obtained automatically from the Mode Set array. The user may make changes in the array when he wants to change a value such as character orientation or left margins; the modified Mode Set will stay in effect until the user resets it, or until the IGS system is reinitialized.

4. Vector Characters - IGS contains the ability to draw vector characters of variable size and orientation.

5. Grid, Plot, Label - IGS contains a comprehensive set of subroutines to draw grids, plot symbols, and label graphs.

6. Addressable CHARACTRON Set - the full CHARACTRON character set will be addressable by normal 8-bit or 6-bit characters. Special control characters appearing in a 6-bit character string will select lower case characters and other special symbols. (See Figure 2)

7. S-C 4020 Software Compatibility - SCORS compatibility is maintained. SCORS subroutines have been modified to call on the IGS subroutines to produce the S-C 4060 input in meta-language form. None of the SCORS calling sequences need be changed. SCORS programs, running under the IGS system, will produce comparable output on the S-C 4060 with as good or better efficiency as they would have on the S-C 4020.

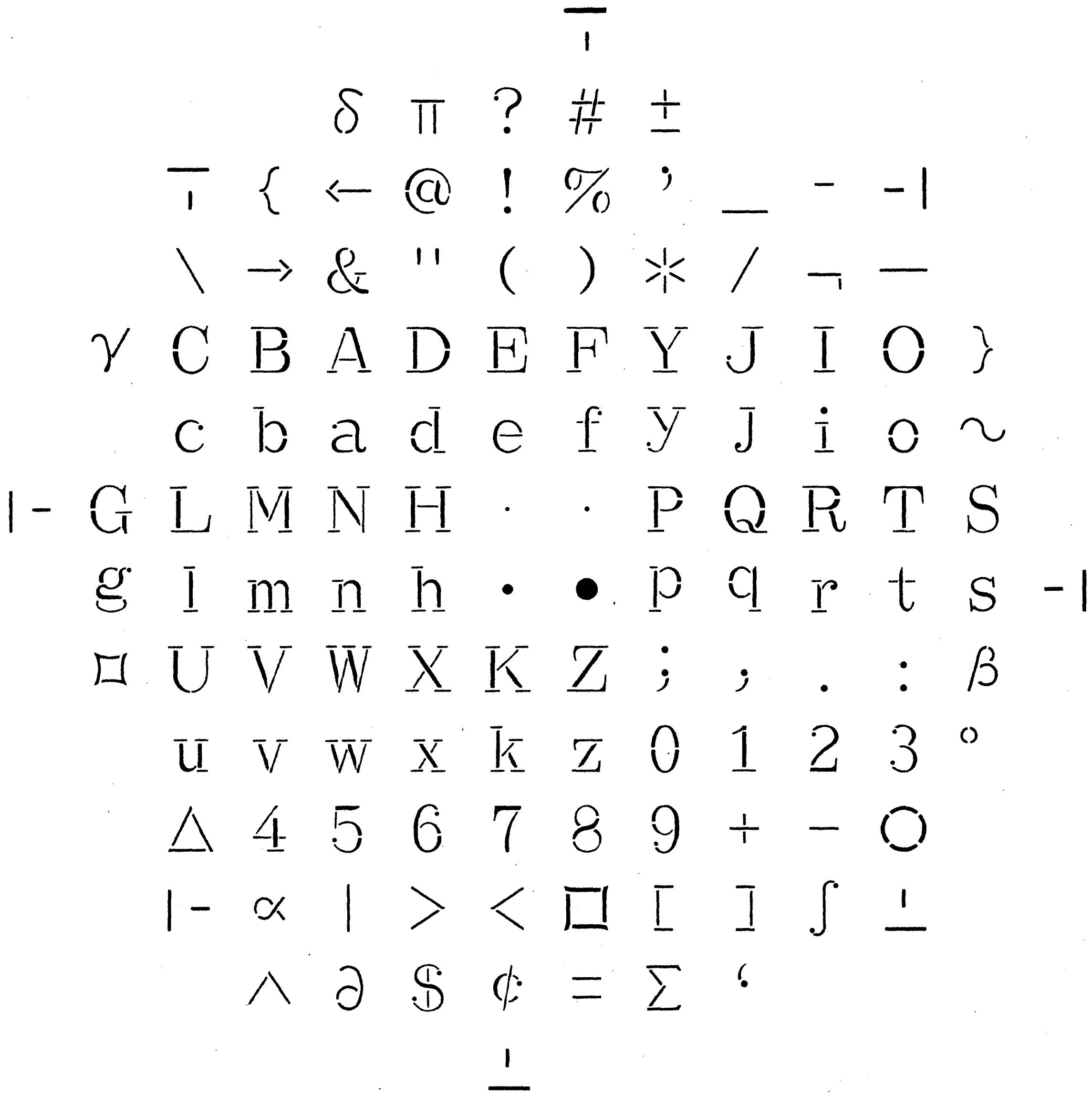


Figure 2. S-C 4060 CHARACTRON Set

D. Advantages of IGS

1. The basic S-C 4020 SCORS package is upwardly compatible with and is supported by the IGS system.

2. It is based on concepts which already have been analyzed and considered by major contributors and users of existing UAIDE (Society for Users of Automatic Information Display Equipment) software. Continued participation, review, and approval of these users will be invited in order to obtain the widest possible UAIDE consensus. Accordingly, IGS will be updated periodically to provide each user with the latest improvements to the package.

3. Much of the basic design was suggested by the SHARE specification for a standard graphic output language (GRAFPAC).

4. It is coded in ASA Standard Basic Fortran where possible; thus it may be compiled on less sophisticated compilers.

5. Emphasis is placed on machine independence. While the working package was developed and checked out on IBM 360 and 7044 computers, specific dependencies on word and character formats and machine language operations were avoided except for packing and transmission routines. Thus the package is substantially transferable to other computers such as IBM 7090/94, Univac 1107, 1108, RCA Spectra 70, GE 625, 635, CDC 3600, 6600, etc.

6. It is coded using re-entrant or serially re-usable techniques wherever possible, thus providing particular appeal for users of time-sharing computing systems and other multi-programming systems.

7. The package is usable by PL/I programs.

III. S-C 4060 INTERNAL SOFTWARE - SCRIP

A. Background

The S-C 4060 Microfilm Plotter-Printer was designed to have a software interface and to operate under software control, thus permitting a higher degree of flexibility than is normally associated with fully hard-wired equipment. The Stromberg-Carlson Recorder Input Processor (SCRIP) software package is a set of routines written in the DAP-16 language (see Appendix F) which efficiently implements the design criteria. The SCRIP package is modular in nature; the standard package supports the S-C 4060 hardware configuration (without optional hardware equipment) and the Standard Operating Environment (External System Configuration using the Integrated Graphic Software package). As the user adds optional hardware equipment to his configuration, the appropriate software module necessary to effectively use that option will be provided by Stromberg-Carlson.

The advantages afforded by software control of the S-C 4060 are as follows:

1. Minimum Operator Intervention - the basic software design provides for automatic production operation with operator activities restricted to tape mounting, bootstrap loading for cold starts, film and paper loading, and taking appropriate action in the case of an irrecoverable tape error.

2. Simplified Operational Control - An ASR-33 teleprinter is included as part of the operational system which provides the medium for operator communication with the S-C 4060. It provides for system access as well as system status printouts (see Appendices D and I).

3. No Equipment Modifications - special user requirements may be incorporated into the system without equipment modifications. Non-standard character and line spacing, special on-line data servicing, special plotting symbols and character fonts, non-standard input translation, and many other non-standard requirements may effectively be added to the system through modification of the SCRIP software, rather than by requiring the user to make hardware modifications.

B. SCRIP Executive Program - MCS

The executive program of SCRIP is called the Master Control System (MCS); during S-C 4060 operation it is the resident program in the Product Control Unit (PCU)* and is responsible for the following functions:

1. Total Operational Control of Execution

   a. MCS Manages ASR-33 Teleprinter Communications - instructions, status queries, and direct data may be entered into the system while execution and processing status, errors, direct messages, and other monitoring information are being returned. Operator communication with the S-C 4060 is in terms of phrases and meaningful mnemonics rather than through the method of interpreting panel light configurations and switch settings. (See Appendices D and I.)

   b. MCS provides a complete Input/Output Control System for all peripheral devices - I/O buffers are assigned and printhead interrupts are serviced automatically, as are all operator override instructions pertaining to magnetic tape transport functions.

      1) MCS thus performs all magnetic tape I/O activities**, such as input tape and library tape read, tape write, record and file backspacing and skipping, tape rewind, and character packing and unpacking. Tape format, parity, and execution errors are detected; the MCS recovery procedure informs the operator of the error, determines the level of the error, and either continues or halts S-C 4060 operation pending operator action.

*The PCU is an 8K (16 bit word) general purpose computer; its execution of MCS performs a function similar to that of a hard-wired interface.

**Input tape searches and special tape label handling are not included in the standard SCRIP system; the implementation of these functions will require a minimum programming effort either by the user or by Stromberg-Carlson by special request.

c. MCS monitors film processing - film exposure is coordinated with ıilm developing to prevent over-development of exposed frames during periods of interrupted execution.

d. MCS provides for continuous job processing - job initialization and termination procedures are designed for optimum operating efficiency. MCS outputs a complete log of each job in an S-C 4060 run. See Appendix E for a description of the principal operating features of MCS.

e. MCS furnishes for SCRIP an overall constants pool and a subprogram communications region. All constants and subprogram communications regions are kept in the resident MCS Nucleus.

2. Total Operational Control of Processing

a. MCS reads into the PCU from the library tape the programs that are necessary to process graphic instructions from the external device (tape unit if off-line, remote computer if on-line) into printhead circuit commands. Processing generally consists of:

1) Interpretation of the instruction;

2) Conversion of the instruction to the appropriate printhead command(s);

3) Output of the resulting command ("hardware logic") to the printhead.

b. The interpretations and conversions of the instructions are performed by subroutines in the library programs. These subroutines are the ultimate means by which the user's desired graphic output function is implemented. The standard SCRIP package allows the user to accomplish the following:

1) Plot any of the CHARACTRON matrix characters at any addressable point on the standard raster in any of four sizes and in either of two orientations (vertical or horizontal).

2) Type a string of matrix or stroke characters beginning at any addressable point on the standard raster and continuing horizontally or vertically to the limits of the

raster. Standard character spacing and line spacing (with carriage return) will be provided according to the selected character size. The character size may be any of the four allowable character sizes and in either of two orientations (vertical or horizontal).

3) Draw a solid or dashed line segment between any two addressable points on the standard raster. Line segments may be drawn in any of four widths and either of two densities.

4) Perform a frame advance.

5) Project a single form onto the film either on command or automatically with each frame advance.

6) "Fast Plot" - plot a string or cluster of characters such that each character is plotted within 80 raster units of the previously plotted character to enable a plotting rate greater than the normal plotting rate.

C. SCRIP Library Programs

The following list gives brief functional descriptions of the programs contained on the S-C 4060 system library tape.

1. META Processor - interprets the standard meta-language input tape functions generated by the external software (IGS), converts these functions to the required printhead commands, and outputs the resulting commands ("hardware logic") to the printhead.

2. S-C 4020 Simulator - reads an S-C 4020 binary input tape, converts the instructions to the necessary S-C 4060 printhead commands, and outputs the commands to the printhead to perform an S-C 4020 simulation. The following restrictions affect the simulation:

   a. A shortened film pulldown is substituted for the expand image command since image expansion is not included in the S-C 4060 circuitry.

b. Camera selection commands are treated as follows:

1) Select camera 1 turns off the hard copy mode.

2) Select camera 2 and select both cameras is interpreted as Expose Hardcopy.

3. BCD Tape Printing - interprets an input tape that has been formatted for a standard line printer and provides the corresponding printed output.

4. Test and Maintenance Programs - exercises the internal circuits of both the Printhead and the Product Control Unit. The routines test the Product Control Unit memory, basic machine instructions, logic and arithmetic units, input/output devices, and provide alignment and performance testing for the recording head (see Appendix H).

5. Optional Programs - are incorporated into the library when the user obtains the corresponding optional hardware.

a. Film and Hardcopy Processing Monitor.

b. On-line Control Monitor.

c. Stroke Generator Routine.

d. Card Reader Input Translator.

e. High Speed Paper Tape Monitor.

D. S-C 4060 Supplementary Software

The two software packages which supplement the primary software packages (IGS and SCRIP) of the S-C 4060 system are the DAP-16 Assembler and the Fortran IV Compiler. Both assembler and compiler will operate from magnetic tape and will treat the S-C 4060 print head as a line printer thus allowing listings to be generated on microfilm.

The DAP-16 Assembler will enable the user to write routines which may be incorporated into the SCRIP package, and thus take full advantage of the flexibility provided by the software control of the S-C 4060.

1. DAP-16 Assembler - generates a set of machine instructions which correspond to a program written in the DAP-16 symbolic language (see Appendix F); it may perform a one pass or two pass assembly. It interprets all symbols and mnemonics, allocates storage blocks, assigns buffers, provides subroutine linkage, allows Fortran compatibility, and provides an object program (either paper tape or magnetic tape) and an assembly listing on the ASR-33 or the S-C 4060 print head showing appropriate diagnostics.

   The following routines are included as part of the DAP-16 Assembler package:

   a. DAP/Fortran relocating loaders (ASR-33, High Speed Paper Tape, and Magnetic Tape).

   b. I/O Supervisor for DAP and Fortran.

   c. Memory dump on ASR-33 and Print Head.

   d. A debug program which enables dynamic tracing.

   e. Routine to punch a program from or onto paper tape preceeded by a bootstrap loader.

   Source input on punched paper tape may be prepared on the ASR-33 in the off-line mode and then read by the DAP-16 Assembly program from the ASR-33 (or the optional high speed reader); source input may also be prepared on punched cards and read in via the optional card reader or on magnetic tape. Object tapes are punched via the ASR-33 or the optional high speed punch.

2. Fortran IV Compiler - processes the standard Fortran language specified by the American Standards Association (ASA - see Appendix G). It is a one-pass compiler which requires a minimum of 8 K core storage* with one of each of the following peripheral equipment:

* The standard S-C 4060 PCU has 8 K core storage.

a. Source input device:

   1) ASR-33 key-in or ASR-33 punched paper tape reader.

   2) Magnetic tape.

   3) Card reader (optional).

   4) High speed punched paper tape reader (optional).

b. Listing device:

   1) ASR-33 printer.

   2) S-C 4060 film.

c. Object output:

   1) Magnetic tape.

   2) High speed paper tape punch (optional).

APPENDIX A

Meta-Language Description

The basic building unit for the meta-language is the 6-bit or 8-bit byte, depending upon the host computer. Each display function (i.e., plot, vector, fast plot, etc.) will be of the same logical structure. The first byte will be the delimiter, the second byte will be the function code, and the succeeding bytes will specify positional information, characters, or control codes.

The S-C 4060 internal matrix code for alphanumeric characters differs from any existing standard code while the ASR-33 requires ASCII. MCS is designed to perform the necessary conversion of the inherent alphanumeric code from the external computer by means of conversion tables.

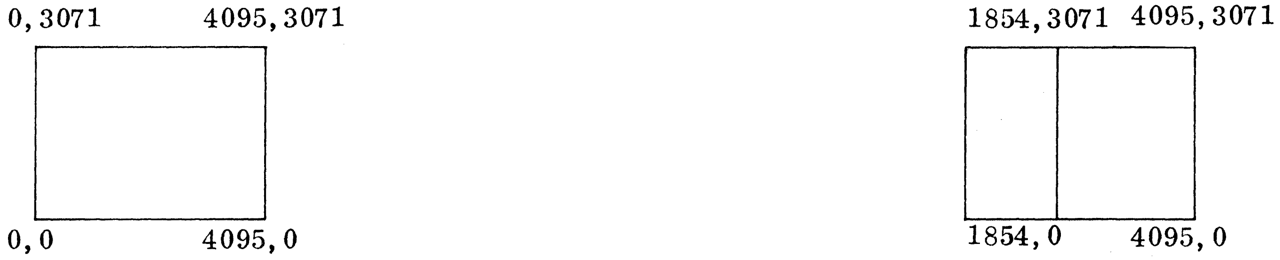
Any intermediate language which does not conform either logically or physically to the meta-language, will require the addition of a software input translator by the user as part of the S-C 4060 operating system.

APPENDIX B

S-C 4060 Standard Raster

The standard raster of the S-C 4060 is defined as a rectangular array of points 3072 vertically by 4096 horizontally, making a total of 12,582,912 points. Each point is addressable which means that it may be specifically referred to by common rectangular coordinate notation. The coordinates specifying the raster points must be positive integers and the standard raster is to be considered as first quadrant.

The standard raster with an accompanying six perforation pulldown (normal frame advance) will produce an image on film which will yield 11 x 14 hardcopy. For 8-1/2 x 11 hardcopy, the raster size will be partitioned to the raster area as shown in the following figure. Normal pulldown will also be six perforations.



Standard Raster and Partitioned Raster

(NOTE: The standard raster as it physically appears on the face of the CHARACTRON tube has for its origin the coordinates (0,512). The programmer, however, need not be concerned with these values because the SCRIP software automatically performs the required translations on all data to be plotted. For the partitioned standard raster (8-1/2 x 11), the physical origin is at (1854, 512). Again, automatic translation by the SCRIP software enables the programmer to use (0,0) for his origin.)

APPENDIX C

S-C 4060 Meta-Language

I. FUNCTION STRING PARTS

A. Delimiter

6 bit mode	1	0	1	1	1	0	1	1	1	1	1	0	Octal 5676
8 bit mode	0	0	1	0	1	0	1	0	Hex 2A Octal 052				

The delimiter is the start flag for a function string. It must be followed by either a null or a function code (See Appendix A).

B. Function Code (f)

6 bit mode								
8 bit mode								

List of Function Codes (decimal).

0	Set Mode Matrix
1	Plot Specified Point #1
2	Plot Specified Point #2
3	Plot Current Point
4	Plot Specified Point #3
5	Type Specified Point
6	Type Current Point
7	Pause
8	Draw Joined Vectors
9	Draw Line Segments
10	No operation (NOP)
11	Tab Set
12	Form Flash
13	Advance Frame
14	Stroke Table Input
15	Operator Message
16	Retrieval Codes
17	Start of Job
18	Frame ID
19	Repeat Frame
20	Reset Mode Matrix
21	End of Job
22	End of Run
23	Stroke Write Bit Pattern
24	Draw Vector Family

26 ID

The Function Code must be the first non-null character following a delimiter.

C. Raster Coordinates (X, Y)

6 bit mode	1 - 6	7 - 18	19 - 24	25 - 36
	Sign 00 = + 01 = -	0000 to 7777$_8$	Sign 00 = + 01 = -	0000 to 7777$_8$
	X		Y	
8 bit mode	1 - 4	5 - 16	17 - 20	21 - 32
	Sign	0 to 7777$_8$	Sign	0 to 7777$_8$
	X		Y	

The position within the function string is defined by the function code. Note leading zero bits. If S = 0, X (or Y) is taken as positive. If S = 1, X (or Y) is taken as two's complement (Module 409) negative value.

D. Plot or Type Characters (C)

6 bit mode □□□□□□

8 bit mode □□□□□□□□

The position within the function string is defined by the function code. The 6 bit character may be BCD, Fieldata, or Excess 3. The 8 bit code may be ASCII or EBCDIC.

E. Null Byte (NUL)

6 bit mode	1	0	1	1	1	0	1	1	1	1	1	1	Octal 5677
8 bit mode	1	1	1	1	1	1	1	1	Hex FF Octal 377				

The null byte may occur within a function string between any two function parts but must not break a function part. The occurrence of the null byte within a function string in no way changes the function of the string.

F. Control Bytes

1. Single Byte (8 bit Mode only)

Description	Octal	Hexadecimal
Delimiter	052	2A
Null Byte	377	FF
Null (typing control)	000	00
Tab	005	05
Carriage Return/Line Feed	025	15
Shift Case 1	066	36
Shift Case 2	006	06
Shift Case 3	051	29
Superscript Shift	004	04
Subscript Shift	060	30

2. Double Byte (Both 6 and 8 bit Mode)

Description	Control Byte (byte #1)		Control Code (byte #2)	
	6 bit	8 bit	6 bit	8 bit
Delimiter	56_8	52_8	76_8	----
Null	56_8	377_8	77_8	----

TYPING CONTROL

Function	Code	2nd Character		
		6 bit	8 bit	C
Null (typing)	$N	45	325	N
Tab	$T	63	343	T
Carriage Return/ Line Feed	$E	25	305	E
Shift Case 1	$U	64	344	U
Shift Case 2	$L	43	323	L
Shift Case 3	$S	62	342	S
Superscrip	$+	20	216	+
Subscript	$-	60	140	-
Capitalize Next	$C	23	203	C
Backspace	$B	22	302	B
Next Page	$P	47	327	P

NOTE: (Code for \$ is 53_8 in 6 bit or 133_8 in 8 bit)

II. Modeset Function String (M)

M	N			
1	0	Character size for plotting	-	Normal size (default)
	1		-	Small size
	2		-	Medium size
	3		-	Large size
2	0	Character orientation for plotting	-	vertical (default)
			-	horizontal
3	0	Line weight	-	Normal (default)
	1		-	Light
	2		-	Heavy
	3		-	Heaviest
4	0	Dash length	-	Solid
	1		-	32 raster units
	2		-	64 raster units
	3		-	128 raster units
	4		-	256 raster units
5	0	Character size for typing	-	Normal size (default)
	1		-	Small size
	2		-	Medium size
	3		-	Large size
6	0	Character and line orientation for typing	-	(default) vertical chars, horizontal line
	1		-	horizontal chars, vertical line
7	0	Vector speed	-	normal (default)
	1		-	fast
8	0	Plot Mode	-	normal (default)
			-	fast
12	0	Plotting character case	-	1 (upper) (default)
	1		-	2 (lower)
	2		-	3 (special)
13	0	Page overflow in typing	-	Same page (default)
	1		-	Frame advance
14	0	Set hardcopy for 4020	-	11 x 14 (default)
	1		-	8-1/2 x 11

M	N		
15	0	Void mark flag	- no void (default)
	1		- Void this frame
16	0	Multiple line spacing for typing	- single space (default)
	n		- skip n + 1 lines
17	0	Typing character case	- upper (default)
	1		- lower
	2		- Special
18	0	Character font	- CHARACTRON Characters (default)
	n		- font n required for stroke write
19	0	Programmer selection of	- On (default)
	1	Block Mode	- Off
20	0	Print list flag for PRNT	- Off, Print normal (default)
	1		- On, print list
21	0	Corner Marks	- No corner marks (default)
	1		- Corner marks required
22	0	Automatic Frame ID	- No automatic frame ID (default)
	1		- Automatic frame ID required
23	0	Automatic Form Flash	- No automatic form flash (default)
	1		- Automatic form flash required
24	0	Output Mode	- None (default)
	1		- Cut mode 11 x 14
	2		- Cut mode 8-1/2 x 11
	3		- Strip chart
26	0	Input Mode	- 8 bit mode (default)
	1		- 6 bit mode
27		Character spacing for typing follows - horizontal displacement default 31 (Normal Size Character)	
28		Character spacing for typing follows - vertical spacing default 0	
29		Line spacing follows default 52	
30		Next frame number follows initially = 1	
31		Working raster: X-min default = 0	

M	N					
32		Working raster:	Y-min		=	512
33		Working raster:	X-max		=	4095
34		Working raster:	Y-max		=	3583
35		Typing margins	XLFT	default	=	0
36		(relative	YBOT			0
37		to working	XRGT			4095
38		rasters)	YTOP			3071
39		Lines per page for PRNT		default	=	58

III. Function String Formats

Function Code	Name Format and Description
1	Plot Specified Point #1 ‡ $1CX_1Y_1X_2Y_2$ - - - - - - X_nY_n Plot a single character at different X and Y points.
2	Plot Specified Point #2 ‡ $2C_1X_1Y_1C_2X_2Y_2$ - - - - $C_nX_nY_n$ Plots different characters at different specified X and Y points.
3	Plot Current Point ‡ $3C_1C_2C_3$ - - - - - - - - -C_n Plot different characters all at current point.
4	Plot Specified Point #3 ‡ $4X_1Y_1C_1C_2C_3$ - - - - - - -C_n Plot different characters all at specified point.
5	Type Specified Point #1 ‡ $5X_1Y_1C_1C_2C_3$ - - - - - C_n Types a series of characters beginning at a specified point X, Y.
6	Type Current Point #1 ‡ $6C_1C_2C_3$ - - - - - - - - C_n Types a series of characters beginning at current point.
7	Pause ‡ 7 (no arguments) Causes the SCRIP operating program (MCS-Master Control System) to enter an idle state in the PCU for S-C 4060 operator action. This action should have been communicated to the operator either personally or by a preceding function code to type out an on-line message on the ASR (see function code 15).

Function Code	Name Format and Description
8	Draw Joined Vectors ‡ $8X_1Y_1X_2Y_2X_3Y_3$ ------ X_nY_n Draws series of joined (head to fail) vectors.
9	Draw Line Segments ‡ $9X_1Y_1X_2Y_2$ ---------- $X_{n-1}Y_{n-1}X_nY_n$ Draws series of separate line segments (n must be even)
10	No Operation ‡ 10
11	Communicate Table of Tabsets (f = 11) ‡, f, T_1, T_2, T_3, ----- T_n T_i represents the horizontal tab stop in raster units relative to XMIN (Mode 31_8) or YMIN (Mode 32_8) depending on value of ORTP (Mode 6_8).
12	Form Flash (f = 12) ‡, f
13	Frame Advance (f = 13) ‡, f, N N represents the number of frame advances to be performed (one byte)
14	Transmit Stroke Table (f = 14) ‡ f, n, h, s_1, s_2, ------- s_n, o, h, s_1, s_2, -- (n = Stroke Table Font Number in Octal) Each byte string from the height adjust designator n to the end of that stroke character terminated by a 0 (zero) repeats until a delimiter follows the 0.
15	Message to Operator (f = 15) ‡, f, c_1, c_2, c_3, c_4,---- c_n n —
16	Retrieval Code (f = 16) ‡, f, T, cd T represents the code type (Miracode or kodamatic) and cd is a fixed length bit string representing the code.

Function Code — Name Format and Description

17 Start of Job (f = 17)

‡, f, c_1, c_2, c_3, ------ c_n (71 character maximum)

1. The character string is typed on the ASR-33.
2. The character string is placed on the ID frame.
3. The function executes the user supplier accounting routine, if present.

18 Frame ID (f = 18)

‡, f, c_1, c_2, c_3, ----- c_n

First 20 characters are retained for use by the frame ID routine.

19 Repeat Frame (f = 19)

‡, f, n

n represents the number of times the previous frame is to be repeated.

20 Reset Mode Matrix (f = 20)

‡, f

Resets the mode matrix to the default state

21 End of Job (f = 21)

‡, f

Type comment to operator, proceed to next job.

22 End of Run (f = 22)

‡, f

Type comment to operator, halt for operator action.

23 Stroke Character (f = 23)

‡, f, x, y, sz, h, s_1, s_2, s_3, ------- s_n, 0)

SZ is size and H is height adjust.

SZ = 0 normal size, normal orientation
1 large size, normal orientation

S_i represents the stroke patterns.

24 Draw vector Family (f = 24)

‡, f, n, x_1, y_1, x_2, y_2

Draws n equidistant vectors between the two specified by x_1, y_1 and x_2 y_2

n = 2 bytes

APPENDIX D

Examples of ASR-33 Messages

A complete list of messages is contained in Document Number HMO-208, S-C 4060 Stored Program Recording System, Operator's Handbook.

**NO PGM	Requested program is not available for execution.
JOB	Part of STATUS reply, ERROR print out, and END OF JOB print out.
FRAME	Part of STATUS reply, ERROR print out, and END OF JOB print out.
**STOPPED	Signals operator that system has entered the STOP state.
**READY	Signals operator that system has entered the READY State.
**OK	Previously specified command has been successfully executed or issued.
**END OF JOB	End of Job sensed on input tape.
**WHAT?	System is unable to interpret ASR-33 input.
**FILM LOW	Operator Warning
**PAPER LOW	Operator Warning
**PAUSE	Programmer requested system halt. Operator action will be required.

APPENDIX E

MCS OPERATION

The following is a brief description of the principal operating features of MCS:

A. Initialization

Initialization is a process which automatically occurs after MCS is initially loaded, at the end of job, and after the RESTART, CNCL, or NEXT command is input from the ASR-33. This process sets all default conditions and interrogates the print head and input tape unit sense lines for the ready condition.

B. System Status

During MCS operation the system will be in either the RUN status or the IDLE status. The RUN status means that MCS is processing input data and outputting printer circuit commands to the print head. IDLE status signifies that the input data is not being processed and the system is waiting for an input from the ASR-33. Monitoring of film developing, however, is maintained during IDLE status.

Two separate conditions exist during IDLE status: Ready state and Stop state.

1. Ready State: The Ready state is that condition which immediately succeeds the initialization process and precedes processor execution.
2. Stop State: The Stop state is that condition which results from either the STOP command being input from the ASR-33, or when MCS detects an error which is of such a consequence as to require operator action, or when a PAUSE function code is encountered in processing meta-language.

C. Error Detection

MCS provides for continuous monitoring of all the major functions of the S-C 4060. If an error or a serious condition occurs, a comment will be printed on the ASR-33.

D. ASR-33 Communication

One of the principal features of the S-C 4060 operating system is that it enables direct communication with the S-C 4060 via the ASR-33. Because of this, the entire operation, with the exception of a cold start, is under control of the ASR-33.

Appendix I contains examples of input instructions and the system action resulting from them.

E. Library Director and Loader

MCS performs all loading from the library. The LOAD instruction when given by the operator (see Appendix I) causes MCS to locate the requested processor on the library tape and load it into a pre-designated location in core.

F. Input/Output Supervisor

IOS provides for general magnetic tape handling. It enables reading (and optionally, writing) in either even or odd parity; at all densities up to 800 bpi; and in either the 7 or 9 track configuration. It provides for backspacing and skipping of single records and files and performs a rewind.

All I/O operations are fully monitored for parity errors and when detected initializes corrective action. For example, nine attempts will be made to read when a read-error is encountered and, in the case of a 9 track tape, a CRC (Cylic Redundancy Check) test will be made to correct the error.

APPENDIX F

SUMMARY OF S-C 4060 SYMBOLIC INSTRUCTIONS (DAP-16)

Mnemonic	Octal Code	Instruction	Cycles
ACA	141216	Add C to A	1
ADD	06	Add	2
ALR	0416	Logical Left Rotate	1 + n/2
ALS	0415	Arithmetic Left Shift	1 + n/2
ANA	03	AND to A	2
AOA	141206	Add One to A	1
ARR	0406	Logical Right Rotate	1 + n/2
ARS	0405	Arithmetic Right Shift	1 + n/2
CAL	141050	Clear A, Left Half	1
CAR	141044	Clear A, Right Half	1
CAS	11	Compare	3
CHS	140024	Complement A Sign	1
CMA	140401	Complement A	1
CRA	140040	Clear A	1
CSA	140320	Copy Sign and Set Sign Plus	1
ENB	000401	Enable Program Interrupt	1
ERA	05	Exclusive OR to A	2
HLT	000000	Halt	
IAB	000201	Interchange A and B	1
ICA	141340	Interchange Characters in A	1
ICL	141140	Interchange and Clear Left Half of A	1
ICR	141240	Interchange and Clear Right Half of A	1
IMA	13	Interchange Memory and A	3
INA	54	Input to A	2
INH	001001	Inhibit Program Interrupt	1
INK	000043	Input Keys	1
IRS	12	Increment, Replace and Skip	3
JMP	01	Unconditional Jump	1
JST	10	Jump and Store Location	3
LDA	02	Load A	2
LDX	15	Load X	3
LGL	0414	Logical Left Shift	1 + n/2

APPENDIX F (Cont)

SUMMARY OF S-C 4060 SYMBOLIC INSTRUCTIONS (DAP-16)

Mnemonic	Octal Code	Instruction	Cycles
LGR	0404	Logical Right Shift	1 + n/2
LLL	0410	Long Left Logical Shift	1 + n/2
LLR	0412	Long Left Rotate	1 + n/2
LLS	0411	Long Arithmetic Left Shift	1 + n/2
LRL	0400	Long Right Logical Shift	1 + n/2
LRR	0402	Long Right Rotate	1 + n/2
LRS	0401	Long Arithmetic Right Shift	1 + n/2
NOP	101000	No Operation	1
OCP	14	Output Control Pulse	2
OTA	74	Output From A	2
OTK	171020	Output Keys	2
RCB	140200	Reset C Bit	1
SCB	140600	Set C Bit	1
SKP	100000	Unconditional Skip	1
SKS	34	Skip if Ready Line Set	2
SLN	101100	Skip if (A_{16}) is ONE	1
SLZ	100100	Skip if (A_{16}) is ZERO	1
SMI	101400	Skip if A Minus	1
SMK	74	Set Mask	2
SNZ	101040	Skip if A Not ZERO	1
SPL	100400	Skip if A Plus	1
SRC	100001	Skip if C Reset	1
SR1	100020	Skip if Sense Switch 1 is Reset	1
SR2	100010	Skip if Sense Switch 2 is Reset	1
SR3	100004	Skip if Sense Switch 3 is Reset	1
SR4	100002	Skip if Sense Switch 4 is Reset	1
SSC	101001	Skip if C Set	1
SSM	140500	Set Sign Minus	1
SSP	140100	Set Sign Plus	1
SSR	100036	Skip if no Sense Switch Set	1
SSS	101036	Skip if any Sense Switch is Set	1
SS1	101020	Skip if Sense Switch 1 is Set	1
SS2	101010	Skip if Sense Switch 2 is Set	1
SS3	101004	Skip if Sense Switch 3 is Set	1
SS4	101002	Skip if Sense Switch 4 is Set	1
STA	04	Store A	2
STX	15	Store X	2

APPENDIX F (Cont)

SUMMARY OF S-C 4060 SYMBOLIC INSTRUCTIONS (DAP-16)

Mnemonic	Octal Code	Instruction	Cycles
SUB	07	Subtract	2
SZE	100040	Skip if A ZERO	1
TCA	140407	Two's Complement A	1.5

APPENDIX G
PROPOSED AMERICAN STANDARD FORTRAN

The following Proposed American Standard of the FORTRAN language was developed by X3.4.3–FORTRAN Group under the American Standards Association Sectional Committee X3, Computers and Information Processing. The committee was established under the sponsorship of the Business Equipment Manufacturers Association. Here is presented the most recent issue of the proposed standard available at this printing. Any further issues are not expected to alter the technical content.

Inquiries regarding copies of the Proposed Standard should be addressed to the X3 Secretary, BEMA, 235 E. 42nd Street, New York, N.Y.

TABLE OF CONTENTS

SECTION

PROPOSED AMERICAN STANDARD FORTRAN

1. INTRODUCTION

1.1 PURPOSE. This standard establishes the form for and the interpretation of programs expressed in the FORTRAN language for the purpose of promoting a high degree of interchangeability of such programs for use on a variety of automatic data processing systems. A processor shall conform to this standard provided it accepts, and interprets as specified, at least those forms and relationships described herein.

Insofar as the interpretation of the form and relationships described are not affected, any statement of requirement could be replaced by a statement expressing that the standard does not provide an interpretation unless the requirement is met. Further, any statement of prohibition could be replaced by a statement expressing that the standard does not provide an interpretation when the prohibition is violated.

1.2 SCOPE. This standard establishes:

(1) The form of a program written in the FORTRAN language.

(2) The form of writing input data to be processed by such a program operating on automatic data processing systems.

(3) Rules for interpreting the meaning of such a program.

(4) The form of the output data resulting from the use of such a program on automatic data processing systems, provided that the rules of interpretation establish an interpretation.

This standard does not prescribe:

(1) The mechanism by which programs are transformed for use on a data processing system (the combination of this mechanism and data processing system is called a processor).

(2) The method of transcription of such programs or their input or output data to or from a data processing medium.

(3) The manual operations required for set-up and control of the use of such programs on data processing equipment.

(4) The results when the rules for interpretation fail to establish an interpretation of such a program.

(5) The size or complexity of a program that will exceed the capacity of any specific data processing system or the capability of a particular processor.

(6) The range or precision of numerical quantities.

2. BASIC TERMINOLOGY

This section introduces some basic terminology and some concepts. A rigorous treatment of these is given in later sections. Certain assumptions concerning the meaning of grammatical forms and particular words are presented.

A program that can be used as a self-contained computing procedure is called an *executable program* (9.1.6).

An executable program consists of precisely one main program and possibly one or more subprograms (9.1.6).

A *main program* is a set of statements and comments not containing a FUNCTION, SUBROUTINE, or BLOCK DATA statement (9.1.5).

A *subprogram* is similar to a main program but is headed by a BLOCK DATA, FUNCTION, or SUBROUTINE statement. A subprogram headed by a BLOCK DATA statement is called a specification subprogram. A subprogram headed by a FUNCTION or SUBROUTINE statement is called a procedure subprogram (9.1.3, 9.1.4).

The term *program unit* will refer to either a main program or subprogram (9.1.7).

Any program unit except a specification subprogram may reference an *external procedure* (Section 9).

An external procedure that is defined by FORTRAN statements is called a *procedure subprogram*. External procedures also may be defined by other means. An external procedure may be an external function or an external subroutine. An external function defined by FORTRAN statements headed by a FUNCTION statement is called a *function subprogram*. An external subroutine defined by FORTRAN statements headed by a SUBROUTINE statement is called a *subroutine subprogram* (Sections 8 and 9).

Any program unit consists of *statements* and *comments*. A statement is divided into physical sections called *lines*, the first of which is called an *initial line* and the rest of which are called *continuation lines* (3.2).

There is a type of line called a comment that is not a statement and merely provides information for documentary purposes (3.2).

The statements in FORTRAN fall into two broad classes—executable and nonexecutable. The executable statements specify the action of the program while the nonexecutable statements describe the use of the program, the characteristics of the operands, editing information, statement functions, or data arrangement (7.1, 7.2).

The syntactic elements of a statement are *names* and *operators*. Names are used to reference objects such as data or procedures. Operators, including the imperative verbs, specify action upon named objects.

One class of name, the *array name*, deserves special mention. An array name must have the size of the identified array defined in an array declarator (7.2.1.1). An array name qualified only by a subscript is used to identify a particular element of the array (5.1.3).

Data names and the arithmetic (or logical) operations may be connected into expressions. Evaluation of such an expression develops a value. This value is derived by performing the specified operations on the named data.

The identifiers used in FORTRAN are names and numbers. Data are named. Procedures are named. Statements are labeled with numbers. Input output units are numbered (Sections 3, 6, 7).

At various places in this document there are statements with associated lists of entries. In all cases the list is assumed to contain at lease one entry unless an explicit exception is stated. As an example, in the statement

SUBROUTINE s (a_1, a_2, · · · a_n)

it is assumed that at least one symbolic name is included in the list within parentheses. A *list* is a set of identifiable elements each of which is separated from its successor by a comma. Further, in a sentence a plural form of a noun will be assumed to also specify the singular form of that noun as a special case when the context of the sentence does not prohibit this interpretation.

The term *reference* is used as a verb with special meaning as defined in Section 5.

3. PROGRAM FORM

Every program unit is constructed of characters grouped into lines and statements.

3.1 THE FORTRAN CHARACTER SET. A program unit is written using the following characters: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, and:

Character	*Name of Character*
	Blank
=	Equals
+	Plus
−	Minus
*	Asterisk
/	Slash
(	Left Parenthesis
)	Right Parenthesis
,	Comma
.	Decimal Point
$	Currency Symbol

The order in which the characters are listed does not imply a collating sequence.

3.1.1 *Digits.* A digit is one of the ten characters: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Unless specified otherwise, a string of digits will be interpreted in the decimal base number system when a number system base interpretation is appropriate.

An octal digit is one of the eight characters: 0, 1, 2, 3, 4, 5, 6, 7. These are only used in the STOP (7.1.2.7.1) and PAUSE (7.1.2.7.2) statements.

3.1.2 *Letters.* A letter is one of the twenty-six characters: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z.

3.1.3 *Alphanumeric Characters.* An alphanumeric character is a letter or a digit.

3.1.4 *Special Characters.* A special character is one of the eleven characters blank, equals, plus, minus, asterisk, slash, left parenthesis, right parenthesis, comma, decimal point, and currency symbol.

3.1.4.1 *Blank Character.* With the exception of the uses specified (3.2.2, 3.2.3, 3.2.4, 4.2.6, 5.1.1.6, 7.2.3.6, and 7.2.3.8), a blank character has no meaning and may be used freely to improve the appearance of the program subject to the restriction on continuation lines in 3.3.

3.2 LINES. A line is a string of 72 characters. All characters must be from the FORTRAN character set except as described in 5.1.1.6 and 7.2.3.8.

The character positions in a line are called columns and are consecutively numbered 1, 2, 3, · · ·, 72. The number indicates the sequential position of a character in the line starting at the left and proceeding to the right.

3.2.1 *Comment Line.* The letter C in column 1 of a line designates that line as a comment line. A comment line must be immediately followed by an initial line, another comment line, or an end line.

A comment line does not affect the program in any way and is available as a convenience for the programmer.

3.2.2 *End Line.* An end line is a line with the character blank in columns 1 through 6, the characters E, N, and D, once each and in that order, in columns 7 through 72, preceded by, interspersed with, or followed by the character blank. The end line indicates to the processor, the end of the written description of a program unit (9.1.7). Every program unit must physically terminate with an end line.

3.2.3 *Initial Line.* An initial line is a line that is neither a comment line nor an end line and that contains the digit 0 or the character blank in column 6. Columns 1 through 5 contain the statement label or each contains the character blank.

3.2.4 *Continuation Line.* A continuation line is a line that contains any character other than the digit 0 or the character blank in column 6, and does not contain the character C in column 1.

A continuation line may only follow an initial line or another continuation line.

3.3 STATEMENTS. A statement consists of an initial line optionally followed by up to nineteen ordered continuation lines. The statement is written in columns 7 through 72 of the lines. The order of the characters in the statement is columns 7 through 72 of the initial line followed, as applicable, by columns 7 through 72 of the first continuation line, columns 7 through 72 of the next continuation line, etc.

3.4 STATEMENT LABEL. Optionally, a statement may be labeled so that it may be referred to in other statements. A statement label consists of from one to five digits. The value of the integer represented is not significant but must be greater than zero. The statement label may be placed anywhere in columns 1 through 5 of the initial line of the statement. The same statement label may not be given to more than one statement in a program unit. Leading zeros are not significant in differentiating statement labels.

3.5 SYMBOLIC NAMES. A symbolic name consists of from one to six alphanumeric characters, the first of which must be alphabetic. See 10.1 through 10.1.10 for a discussion of classification of symbolic names and restrictions on their use.

3.6 ORDERING OF CHARACTERS. An ordering of characters is assumed within a program unit. Thus, any meaningful collection of characters that constitutes names, lines, and statements exists as a totally ordered set. This ordering is imposed by the character position rule of 3.2 (which orders characters within lines) and the order in which lines are presented for processing.

4. DATA TYPES

Six different types of data are defined. These are integer, real, double precision, complex, logical, and Hollerith. Each type has a different mathematical significance and may have different internal representation. Thus the data type has a significance in the interpretation of the associated operations with which a datum is involved. The data type of a function defines the type of the datum it supplies to the expression in which it appears.

4.1 DATA TYPE ASSOCIATION. The name employed to identify a datum or function carries the data type association. The form of the string representing a constant defines both the value and the data type.

A symbolic name representing a function, variable, or array must have only a single data type association for each program unit. Once associated with a particular data type, a specific name implies that type for any differing usage of that symbolic name that requires a data type association throughout the program unit in which it is defined.

Data type may be established for a symbolic name by declaration in a type-statement (7.2.1.6) for the integer, real, double precision, complex, and logical types. This specific declaration overrides the implied association available for integer and real (5.3).

There exists no mechanism to associate a symbolic name with the Hollerith data type. Thus data of this type, other than constants, are identified under the guise of a name of one of the other types.

4.2 DATA TYPE PROPERTIES. The mathematical and the representation properties for each of the data types are defined in the following sections. For real, double precision, and integer data, the value zero is considered neither positive nor negative.

4.2.1 *Integer Type.* An integer datum is always an exact representation of an integer value. It may assume positive, negative, and zero values. It may only assume integral values.

4.2.2 *Real Type.* A real datum is a processor approximation to the value of a real number. It may assume positive, negative, and zero values.

4.2.3 *Double Precision Type.* A double precision datum is a processor approximation to the value of a real number. It may assume positive, negative, and zero values. The degree of approximation, though undefined, must be greater than that of type real.

4.2.4 *Complex Type.* A complex datum is a processor approximation to the value of a complex number. The representation of the approximation is in the form of an ordered pair of real data. The first of the pair represents the real part and the second, the imaginary part. Each part has, accordingly, the same degree of approximation as for a real datum.

4.2.5 *Logical Type.* A logical datum may assume only the truth values of true or false.

4.2.6 *Hollerith Type.* A Hollerith datum is a string of characters. This string may consist of any characters capable of representation in the processor. The blank character is a valid and significant character in a Hollerith datum.

5. DATA AND PROCEDURE IDENTIFICATION

Names are employed to reference or otherwise identify data and procedures.

The term *reference* is used to indicate an identification of a datum implying that the current value of the datum will be made available during the execution of the statement containing the reference. If the datum is identified but not necessarily made available, the datum is said to be *named.* One case of special interest in which the datum is named is that of assigning a value to a datum, thus defining or redefining the datum.

The term, reference, is used to indicate an identification of a procedure implying that the actions specified by the procedure will be made available.

A complete and rigorous discussion of reference and definition, including redefinition, is contained in Section 10.

5.1 DATA AND PROCEDURE NAMES. A data name identifies a constant, a variable, an array or array element, or a block (7.2.1.3). A procedure name identifies a function or a subroutine.

5.1.1 *Constants.* A constant is a datum that is always defined during execution and may not be redefined. Rules for writing constants are given for each data type.

An integer, real, or double precision constant is said to be signed when it is written immediately following a plus or minus. Also, for these types, an optionally signed constant is either a constant or a signed constant.

5.1.1.1 *Integer constant.* An integer constant is written as a nonempty string of digits. The constant is the digit string interpreted as a decimal numeral.

5.1.1.2 *Real Constant.* A basic real constant is written as an integer part, a decimal point, and a decimal fraction part in that order. Both the integer part and the decimal part are strings of digits; either one of these strings may be empty but not both. The constant is an approximation to the digit string interpreted as a decimal numeral.

A decimal exponent is written as the letter, E, followed by an optionally signed integer constant. A decimal exponent is a multiplier (applied to the constant written immediately preceding it) that is an approximation to the exponential form ten raised to the power indicated by the integer written following the E.

A real constant is indicated by writing a basic real constant, a basic real constant followed by a decimal exponent, or an integer constant followed by a decimal exponent.

5.1.1.3 *Double Precision Constant.* A double precision exponent is written and interpreted identically to a decimal exponent except that the letter, D, is used instead of the letter, E.

A double precision constant is indicated by writing a basic real constant followed by a double precision exponent or an integer constant followed by a double precision exponent.

5.1.1.4 *Complex Constant.* A complex constant is written as an ordered pair of optionally signed real constants, separated by a comma, and enclosed within parentheses. The datum is an approximation to the complex number represented by the pair.

5.1.1.5 *Logical Constant.* The logical constants, true and false, are written .TRUE. and .FALSE. respectively.

5.1.1.6 *Hollerith Constant.* A Hollerith constant is written as an integer constant (whose value *n* is greater than zero) followed by the letter H, followed by exactly *n* characters which comprise the Hollerith datum proper. Any *n* characters capable of representation by the processor may follow the H. The character blank is significant in the Hollerith datum string. This type of constant may be written only in the argument list of a CALL statement and in the data initialization statement.

5.1.2 *Variable.* A variable is a datum that is identified by a symbolic name (3.5). Such a datum may be referenced and defined.

5.1.3 *Array.* An array is an ordered set of data of one, two, or three dimensions. An array is identified by a symbolic name. Identification of the entire ordered set is achieved via use of the array name.

5.1.3.1 *Array Element.* An array element is one of the members of the set of data of an array. An array element is identified by immediately following the array name with a qualifier, called a subscript, which points to the particular element of the array.

An array element may be referenced and defined.

5.1.3.2 *Subscript.* A subscript is written as a parenthesized list of subscript expressions. Each subscript expression is separated by a comma from its successor, if there is a successor. The number of subscript expressions must correspond to the declared dimensionality (7.2.1.1), except in an EQUIVALENCE statement (7.2.1.4). Following evaluation of all of the subscript expressions, the array element successor function (7.2.1.1) determines the identified array element.

5.1.3.3 *Subscript Expressions.* A subscript expression is written as one of the following constructs:

$$\begin{array}{l} c*v + k \\ c*v - k \\ c*v \\ v + k \\ v - k \\ v \\ k \end{array}$$

where c and k are integer constants and v is an integer variable reference. See Section 6 for a discussion of evaluation of expressions and 10.2.8 and 10.3 for requirements that apply to the use of a variable in a subscript.

5.1.4 *Procedures.* A procedure (Section 8) is identified by a symbolic name. A procedure is a statement function, an intrinsic function, a basic external function, an external function, or an external subroutine. Statement functions, intrinsic functions, basic external functions, and external functions are referred to as functions or function procedures; external subroutines as subroutines or subroutine procedures.

A function supplies a result to be used at the point of reference; a subroutine does not. Functions are referenced in a manner different from subroutines.

5.2 Function Reference. A function reference consists of the function name followed by an actual argument list enclosed in parentheses. If the list contains more than one argument, the arguments are separated by commas. The allowable forms of function arguments are given in Section 8.

See 10.2.1 for a discussion of requirements that apply to function references.

5.3 Type Rules for Data and Procedure Identifiers. The type of a constant is implicit in its name.

There is no type associated with a symbolic name that identifies a subroutine or a block.

A symbolic name that identifies a variable, an array, or a statement function may have its type specified in a type-statement. In the absence of an explicit declaration, the type is implied by the first character of the name: I, J, K, L, M, and N imply type integer; any other letter implies type real.

A symbolic name that identifies an intrinsic function or a basic external function when it is used to identify this designated procedure, has a type associated with it as specified in Tables 3 and 4.

In the program unit in which an external function is referenced, its type definition is defined in the same manner as for a variable and an array. For a function subprogram, type is specified either implicitly by its name or explicitly in the FUNCTION statement.

The same type is associated with an array element as is associated with the array name.

5.4 Dummy Arguments. A dummy argument of an external procedure identifies a variable, array, subroutine, or external function.

When the use of an external function name is specified, the use of a dummy argument is permissible if an external function name will be associated with that dummy argument. (Section 8.)

When the use of an external subroutine name is specified, the use of a dummy argument is permissible if an external subroutine name will be associated with that dummy argument.

When the use of a variable or array element reference is specified, the use of a dummy argument is permissible if a value of the same type will be made available through argument association.

Unless specified otherwise, when the use of a variable, array, or array element name is specified, the use of a dummy argument is permissible provided that a proper association with an actual argument is made.

The process of argument association is discussed in Sections 8 and 10.

6. EXPRESSIONS

This section gives the formation and evaluation rules for arithmetic, relational, and logical expressions. A relational expression appears only within the context of logical expressions. An expression is formed from elements and operators. See 10.3 for a discussion of requirements that apply to the use of certain entities in expressions.

6.1 Arithmetic Expressions. An arithmetic expression is formed with arithmetic operators and arithmetic elements. Both the expression and its constituent elements identify values of one of the types integer, real, double precision, or complex. The arithmetic operators are:

Operator	*Representing*
+	Addition, positive value (zero + element)
−	Subtraction, negative value (zero − element)
*	Multiplication
/	Division
**	Exponentiation

The arithmetic elements are primary, factor, term, signed term, simple arithmetic expression, and arithmetic expression.

A primary is an arithmetic expression enclosed in parentheses, a constant, a variable reference, an array element reference, or a function reference.

A factor is a primary or a construct of the form

*primary**primary*

A term is a factor or a construct of one of the forms

term/factor

or

*term*term*

A signed term is a term immediately preceded by + or −.

A simple arithmetic expression is a term or two simple arithmetic expressions separated by a + or −.

An arithmetic expression is a simple arithmetic expression or a signed term or either of the preceding forms immediately followed by a + or − immediately followed by a simple arithmetic expression.

A primary of any type may be exponentiated by an integer primary, and the resultant factor is of the same type as that of the element being exponentiated. A real or double precision primary may be exponentiated by a real or double precision primary, and the resultant factor is of type real if both primaries are of type real and otherwise of type double precision. These are the only cases for which use of the exponentiation operator is defined.

By use of the arithmetic operators other than exponentiation, any admissible element may be combined with another admissible element of the same type, and the resultant element is of the same type. Further, an admissible real element may be combined with an admissible double precision or complex element; the resultant element is of type double precision or complex, respectively.

6.2 Relational Expressions. A relational expression consists of two arithmetic expressions separated by a relational operator and will have the value true or false as the relation is true or false, respectively. One arithmetic expression may be of type real or double precision and the other of type real or double precision, or both arithmetic expressions may be of type integer. If a real expression and a double precision expression appear in a relational expression, the effect is the same as a similar relational expression. This similar expression contains a double precision zero as the right hand arithmetic expression and the difference of the two original expressions (in their original order) as the left. The relational operator is unchanged. The relational operators are:

Operator	*Representing*
.LT.	Less than
.LE.	Less than or equal to
.EQ.	Equal to
.NE.	Not equal to
.GT.	Greater than
.GE.	Greater than or equal to

6.3 Logical Expressions. A logical expression is formed with logical operators and logical elements and has the value true or false. The logical operators are:

Operator	*Representing*
.OR.	Logical disjunction
.AND.	Logical conjunction
.NOT.	Logical negation

The logical elements are logical primary, logical factor, logical term, and logical expression.

A logical primary is a logical expression enclosed in parentheses, a relational expression, a logical constant, a logical variable reference, a logical array element reference, or a logical function reference.

A logical factor is a logical primary or .NOT. followed by a logical primary.

A logical term is a logical factor or a construct of the form:

logical term .AND. logical term

A logical expression is a logical term or a construct of the form:

logical expression .OR. logical expression

6.4 Evaluation of Expressions. A part of an expression need be evaluated only if such action is necessary to establish the value of the expression. The rules for formation of expressions imply the binding strength of operators. It should be noted that the range of the subtraction operator is the term that immediately succeeds it. The evaluation may proceed according to any valid formation sequence (except as modified in the following paragraph).

When two elements are combined by an operator, the order of evaluation of the elements is optional. If mathematical use of operators is associative, commutative, or both, full use of these facts may be made to revise orders of combination, provided only that integrity of parenthesized expressions is not violated. The results of different permissible orders of combination even though mathematically identical need not be computationally identical. The value of an integer factor or term is the nearest integer whose magnitude does not exceed the magnitude of the mathematical value represented by that factor or term. The associative and commutative laws do not apply in the evaluation of integer terms containing division, hence the evaluation of such terms must effectively proceed from left to right.

Any use of an array element name requires the evaluation of its subscript. The evaluation of functions appearing in an expression may not validly alter the value of any other element within the expressions, assignment statement, or CALL statement in which the function reference appears. The type of the expression in which a function reference or subscript appears does not affect, nor is it affected by, the evaluation of the actual arguments or subscript.

No factor may be evaluated that requires a negative valued primary to be raised to a real or double precision exponent. No factor may be evaluated that requires raising a zero valued primary to a zero valued exponent.

No element may be evaluated whose value is not mathematically defined.

7. STATEMENTS

A statement may be classified as executable or nonexecutable. Executable statements specify actions; nonexecutable statements describe the characteristics and arrangement of data, editing information, statement functions, and classification of program units.

7.1 Executable Statements. There are three types of executable statements:

(1) Assignment statements.
(2) Control statements.
(3) Input output statements.

7.1.1 *Assignment Statements.* There are three types of assignment statements:

(1) Arithmetic assignment statement.
(2) Logical assignment statement.
(3) GO TO assignment statement.

7.1.1.1 *Arithmetic Assignment Statement.* An arithmetic assignment statement is of the form:

$$v = e$$

where v is a variable name or array element name of type other than logical and e is an arithmetic expression. Execution of this statement causes the evaluation of the expression e and the altering of v according to Table 1.

7.1.1.2 *Logical Assignment Statement.* A logical assignment statement is of the form

$$v = e$$

where v is a logical variable name or a logical array element name and e is a logical expression. Execution of this statement causes the logical expression to be evaluated and its value to be assigned to the logical entity.

7.1.1.3 GO TO *Assignment Statement.* A GO TO assignment statement is of the form:

ASSIGN k TO i

where k is a statement label and i is an integer variable name. After execution of such a statement, subsequent execution of any assigned GO TO statement (Section 7.1.2.1.2) using that integer variable will cause the statement identified by the assigned statement label to be executed next, provided there has been no intervening redefinition (9.2) of the variable. The statement label must refer to an executable statement in the same program unit in which the ASSIGN statement appears.

Once having been mentioned in an ASSIGN statement, an integer variable may not be referenced in any statement other than an assigned GO TO statement until it has been redefined (Section 10.2.3).

TABLE 1. RULES FOR ASSIGNMENT OF *e* TO *v*

If v Type Is	*And e Type Is*	*The Assignment Rule Is**
Integer	Integer	Assign
Integer	Real	Fix & Assign
Integer	Double Precision	Fix & Assign
Integer	Complex	P
Real	Integer	Float & Assign
Real	Real	Assign
Real	Double Precision	DP Evaluate & Real Assign
Real	Complex	P
Double Precision	Integer	DP Float & Assign
Double Precision	Real	DP Evaluate & Assign
Double Precision	Double Precision	Assign
Double Precision	Complex	P
Complex	Integer	P
Complex	Real	P
Complex	Double Precision	P
Complex	Complex	Assign

*NOTES.

(1) P means prohibited combination.

(2) Assign means transmit the resulting value, without change, to the entity.

(3) Real Assign means transmit to the entity as much precision of the most significant part of the resulting value as a real datum can contain.

(4) DP Evaluate means evaluate the expression according to the rules of 6.1 (or any more precise rules) then DP Float.

(5) Fix means truncate any fractional part of the result and transform that value to the form of an integer datum.

(6) Float means transform the value to the form of a real datum.

(7) DP Float means transform the value to the form of a double precision datum, retaining in the process as much of the precision of the value as a double precision datum can contain.

7.1.2 *Control Statements.* There are eight types of control statements:

(1) GO TO statements.
(2) arithmetic IF statement.
(3) logical IF statement.
(4) CALL statement.
(5) RETURN statement.
(6) CONTINUE statement.
(7) program control statements.
(8) DO statement.

The statement labels used in a control statement must be associated with executable statements within the same program unit in which the control statement appears.

7.1.2.1 GO TO *Statements.* There are three types of GO TO statements:

(1) Unconditional GO TO statement.
(2) Assigned GO TO statement.
(3) Computed GO TO statement.

7.1.2.1.1 *Unconditional* GO TO *Statement.* An unconditional GO TO statement is of the form:

GO TO k

where k is a statement label.

Execution of this statement causes the statement identified by the statement label to be executed next.

7.1.2.1.2 *Assigned* GO TO *Statement.* An assigned GO TO statement is of the form:

GO TO i, $(k_1, k_2, \cdots, k_n)$

where i is an integer variable reference, and the k's are statement labels.

At the time of execution of an assigned GO TO statement, the current value of i must have been assigned by the previous execution of an ASSIGN statement to be one of the statement labels in the parenthesized list, and such an execution causes the statement identified by that statement label to be executed next.

7.1.2.1.3 *Computed* GO TO *Statement.* A computed GO TO statement is of the form:

GO TO $(k_1, k_2, \cdots, k_n), i$

where the k's are statement labels and i is an integer variable reference. See 10.2.8 and 10.3 for a discussion of requirements that apply to the use of a variable in a computed GO TO statement.

Execution of this statement causes the statement identified by the statement label k_j to be executed next, where j is the value of i at the time of the execution. This statement is defined only for values such that $1 \leqq j \leqq n$.

7.1.2.2 *Arithmetic* IF *Statement.* An arithmetic IF statement is of the form:

IF (e) k_1, k_2, k_3

where e is any arithmetic expression of type integer, real, or double precision, and the k's are statement labels.

The arithmetic IF is a three-way branch. Execution of this statement causes evaluation of the expression e following which the statement identified by the statement label k_1, k_2, or k_3 is executed next as the value of e is less than zero, zero, or greater than zero, respectively.

7.1.2.3 *Logical* IF *Statement.* A logical IF statement is of the form:

IF (e) S

where e is a logical expression and S is any executable statement except a DO statement or another logical IF statement. Upon execution of this statement, the logical expression e is evaluated. If the value of e is false, statement S is executed as though it were a CONTINUE statement. If the value of e is true, statement S is executed.

7.1.2.4 CALL *Statement.* A CALL statement is of one of the forms:

CALL s $(a_1, a_2, \cdots, a_n)$
or
CALL s

where s is the name of a subroutine and the a's are actual arguments (8.4.2).

The inception of execution of a CALL statement references the designated subroutine. Return of control from the designated subroutine completes execution of the CALL statement.

7.1.2.5 RETURN *Statement.* A RETURN statement is of the form:

RETURN

A RETURN statement marks the logical end of a procedure subprogram and, thus, may only appear in a procedure subprogram.

Execution of this statement when it appears in a subroutine subprogram causes return of control to the current calling program unit.

Execution of this statement when it appears in a function subprogram causes return of control to the current calling program unit. At this time the value of the function (8.3.1) is made available.

7.1.2.6 CONTINUE *Statement.* A CONTINUE statement is of the form:

CONTINUE

Execution of this statement causes continuation of normal execution sequence.

7.1.2.7 *Program Control Statements.* There are two types of program control statements:

(1) STOP statement.

(2) PAUSE statement.

7.1.2.7.1 STOP *Statement.* A STOP statement is of one of the forms:

STOP n
or
STOP

where n is an octal digit string of length from one to five.

Execution of this statement causes termination of execution of the executable program.

7.1.2.7.2 PAUSE *Statement.* A PAUSE statement is of one of the forms:

PAUSE n
or
PAUSE

where n is an octal digit string of length from one to five.

The inception of execution of this statement causes a cessation of execution of this executable program. Execution must be resumable. At the time of cessation of execution the octal digit string is accessible. The decision to resume execution is not under control of the program, but if execution is resumed without otherwise changing the state of the processor, the completion of the PAUSE statement causes continuation of normal execution sequence.

7.1.2.8 DO *Statement.* A DO statement is of one of the forms:

$$\text{DO } n\ i = m_1, m_2, m_3$$
or
$$\text{DO } n\ i = m_1, m_2$$

where:

(1) n is the statement label of an executable statement. This statement, called the terminal statement of the associated DO, must physically follow and be in the same program unit as that DO statement. The terminal statement may not be a GO TO of any form, arithmetic IF, RETURN, STOP, PAUSE, or DO statement, nor a logical IF containing any of these forms.

(2) i is an integer variable name; this variable is called the control variable.

(3) m_1, called the initial parameter; m_2, called the terminal parameter; and m_3, called the incrementation parameter, are each either an integer constant or integer variable reference. If the second form of the DO statement is used so that m_3 is not explicitly stated, a value of one is implied for the incrementation parameter. At time of execution of the DO statement, m_1, m_2, and m_3 must be greater than zero.

Associated with each DO statement is a range that is defined to be those executable statements from and including the first executable statement following the DO, to and including the terminal statement associated with the DO. A special situation occurs when the range of a DO contains another DO statement. In this case, the range of the contained DO must be a subset of the range of the containing DO.

A completely nested nest is a set of DO statements and their ranges, and any DO statements contained within their ranges, such that the first occurring terminal statement of any of those DO statements physically follows the last occurring DO statement and the first occurring DO statement of the set is not in the range of any DO statement.

A DO statement is used to define a loop. The action succeeding execution of a DO statement is described by the following five steps:

1. The control variable is assigned the value represented by the initial parameter. This value must be less than or equal to the value represented by the terminal parameter.

2. The range of the DO is executed.

3. If control reaches the terminal statement, and after execution of the terminal statement, the control variable of the most recently executed DO statement associated with the terminal statement is incremented by the value represented by the associated incrementation parameter.

4. If the value of the control variable after incrementation is less than or equal to the value represented by the associated terminal parameter, the action as described starting at step 2 is repeated with the understanding that the range in question is that of the DO, the control variable of which was most recently incremented. If the value of the control variable is greater than the value represented by its associated terminal parameter, the DO is said to have been satisfied and the control variable becomes undefined.

5. At this point, if there were one or more other DO statements referring to the terminal statement in question, the control variable of the next most recently executed DO statement is incremented by the value represented by its associated incrementation parameter and the action as described in step 4 is repeated until all DO statements referring to the particular termination statement are satisfied, at which time the first executable statement following the terminal statement is executed. In the remainder of this section (7.1.2.8) a logical IF statement containing a GO TO or arithmetic IF statement form is regarded as a GO TO or arithmetic IF statement respectively.

Upon exiting from the range of a DO by execution of a GO TO statement or an arithmetic IF statement, that is, other than by satisfying the DO, the control variable of the DO is defined and is equal to the most recent value attained as defined in the foregoing.

A DO is said to have an extended range if both of the following conditions apply:

(1) There exists a GO TO statement or arithmetic IF statement within the range of the innermost DO of a completely nested nest that can cause control to pass out of that nest.

(2) There exists a GO TO statement or arithmetic IF statement not within the nest that, in the collection of all possible sequences of execution in the particular program unit could be executed after a statement of the type described in (1), and the execution of which could cause control to return into the range of the innermost DO of the completely nested nest.

If both of these conditions apply, the extended range is defined to be the set of all executable statements that may be executed between all pairs of control statements, the first of which satisfies the condition of (1) and the second of (2). The first of the pair is not included in the extended range;, the second is. A GO TO statement or an arithmetic IF statement may not cause control to pass into the range of a DO unless it is being executed as part of the extended range of that particular DO. Further, the extended range of a DO may not contain a DO of the same program unit that has an extended range. When a procedure reference occurs in the range of a DO the actions of that procedure are considered to be temporarily within that range, i.e., during the execution of that reference.

The control variable, initial parameter, terminal parameter, and incrementation parameter of a DO may not be redefined during the execution of the range or extended range of that DO.

If a statement is the terminal statement of more than one DO statement, the statement label of that terminal statement may not be used in any GO TO or arithmetic IF statement that occurs anywhere but in the range of the most deeply contained DO with that terminal statement.

7.1.3 *Input/Output Statements.* There are two types of input/output statements:

(1) READ and WRITE statements.

(2) Auxiliary Input/Output statements.

The first type consists of the statements that cause transfer of records of sequential files to and from internal storage, respectively. The second type consists of the BACKSPACE and REWIND statements that provide for positioning of such an external file, and ENDFILE, which provides for demarcation of such an external file.

In the following descriptions, *u* and *f* identify input/output units and format specifications, respectively. An input/output unit is identified by an integer value and *u* may be either an integer constant or an integer variable reference whose value then identifies the unit. The format specification is described in Section 7.2.3. Either the statement label of a FORMAT statement or an array name may be represented by *f*. If a statement label, the identified statement must appear in the same program unit as the input/output statement. If an array name, it must conform to the specifications in 7.2.3.10.

A particular unit has a single sequential file associated with it. The most general case of such a unit has the following properties:

(1) If the unit contains one or more records, those records exist as a totally ordered set.

(2) There exists a unique position of the unit called its initial point. If a unit contains no records, that unit is positioned at its initial point. If the unit is at its initial point and contains records, the first record of the unit is defined as the next record.

(3) If a unit is not positioned at its initial point, there exists a unique preceding record associated with that position. The least of any records in the ordering described by (1) following this preceding record is defined as the next record of that position.

(4) Upon completion of execution of a WRITE or ENDFILE statement, there exist no records following the records created by that statement.

(5) When the next record is transmitted, the position of the unit is changed so that this next record becomes the preceding record.

If a unit does not provide for some of the properties given in the foregoing, certain statements that will be defined may not refer to that unit. The use of such a statement is not defined for that unit.

7.1.3.1 READ *and* WRITE *Statements.* The READ and WRITE statements specify transfer of information. Each such statement may include a list of the names of variables, arrays, and array elements. The named elements are assigned values on input and have their values transferred on output.

Records may be formatted or unformatted. A formatted record consists of a string of the characters that are permissible in Hollerith constants (5.1.1.6). The transfer of such a record requires that a format specification be referenced to supply the necessary positioning and conversion specifications (7.2.3). The number of records transferred by the execution of a formatted READ or WRITE is dependent upon the list and referenced format specification (7.2.3.4). An unformatted record consists of a string of values. When an unformatted or formatted READ statement is executed, the required records on the identified unit must be, respectively, unformatted or formatted records.

7.1.3.1.1 *Input Output Lists.* The input list specifies the names of the variables and array elements to which values are assigned on input. The output list specifies the references to variables and array elements whose values are transmitted. The input and output lists are of the same form.

Lists are formed in the following manner. A simple list is a variable name, an array element name, or an array name, or two simple lists separated by a comma.

A list is a simple list, a simple list enclosed in parentheses, a DO-implied list, or two lists separated by a comma.

A DO-implied list is a list followed by a comma and a DO-implied specification, all enclosed in parentheses.

A DO-implied specification is of one of the forms:

$$i = m_1, m_2, m_3$$

or

$$i = m_1, m_2$$

The elements i, m_1, m_2, and m_3 are as defined for the DO statement (7.1.2.8). The range of DO-implied specification is the list of the DO-implied list and, for input lists, i, m_1, m_2, and m_3 may appear, within that range, only in subscripts.

A variable name or array element name specifies itself. An array name specifies all of the array element names defined by the array declarator, and they are specified in the order given by the array element successor function (7.2.1.1.1).

The elements of a list are specified in the order of their occurrence from left to right. The elements of a list in a DO-implied list are specified for each cycle of the implied DO.

7.1.3.1.2 *Formatted* READ. A formatted READ statement is of one of the forms:

READ (*u*, *f*) *k*

or

READ (*u*, *f*)

where *k* is a list.

Execution of this statement causes the input of the next records from the unit identified by *u*. The information is scanned and converted as specified by the format specification identified by *f*. The resulting values are assigned to the elements specified by the list. See however 7.2.3.4.

7.1.3.1.3 *Formatted* WRITE. A formatted WRITE statement is of one of the forms:

WRITE (*u*, *f*) *k*

or

WRITE (*u*, *f*)

where *k* is a list.

Execution of this statement creates the next records on the unit identified by *u*. The list specifies a sequence of values. These are converted and positioned as specified by the format specification identified by *f*. See however 7.2.3.4.

7.1.3.1.4 *Unformatted* READ. An unformatted READ statement is of one of the forms:

READ (*u*) *k*

or

READ (*u*)

where *k* is a list.

Execution of this statement causes the input of the next record from the unit identified by u, and, if there is a list, these values are assigned to the sequence of elements specified by the list. The sequence of values required by the list may not exceed the sequence of values from the unformatted record.

7.1.3.1.5 *Unformatted* WRITE. An unformatted WRITE statement is of the form:

WRITE (u) k

where k is a list.

Execution of this statement creates the next record on the unit identified by u of the sequence of values specified by the list.

7.1.3.2 *Auxiliary Input Output Statements.* There are three types of auxiliary input output statements:

(1) REWIND statement.
(2) BACKSPACE statement.
(3) ENDFILE statement.

7.1.3.2.1 REWIND *Statement.* A REWIND statement is of the form:

REWIND u

Execution of this statement causes the unit identified by u to be positioned at its initial point.

7.1.3.2.2 BACKSPACE *Statement.* A BACKSPACE statement is of the form:

BACKSPACE u

If the unit identified by u is positioned at its initial point, execution of this statement has no effect. Otherwise, the execution of this statement results in the positioning of the unit identified by u so that what had been the preceding record prior to that execution becomes the next record.

7.1.3.2.3 ENDFILE *Statement.* An ENDFILE statement is of the form:

ENDFILE u

Execution of this statement causes the recording of an endfile record on the unit identified by u. The endfile record is an unique record signifying a demarcation of a sequential file. Action is undefined when an endfile record is encountered during execution of a READ statement.

7.1.3.3 *Printing of Formatted Record.* When formatted records are prepared for printing, the first character of the record is not printed.

The first character of such a record determines vertical spacing as follows:

Character	*Vertical Spacing Before Printing*
Blank	One line
0	Two lines
1	To first line of next page
+	No advance

7.2 Nonexecutable Statements. There are five types of nonexecutable statements:

(1) Specification statements.
(2) Data initialization statement.
(3) FORMAT statement.
(4) Function defining statements.
(5) Subprogram statements.

See 10.1.2 for a discussion of restrictions on appearances of symbolic names in such statements.

The function defining statements and subprogram statements are discussed in Section 8.

7.2.1 *Specification Statements.* There are five types of specification statements:

(1) DIMENSION statement.
(2) COMMON statement.
(3) EQUIVALENCE statement.
(4) EXTERNAL statement.
(5) Type-statements.

7.2.1.1 *Array-Declarator.* An array declarator specifies an array used in a program unit.

The array declarator indicates the symbolic name, the number of dimensions (one, two, or three), and the size of each of the dimensions. The array declarator statement may be a type-statement, DIMENSION, or COMMON statement.

An array declarator has the form:

v (i)

where:

(1) v, called the declarator name, is a symbolic name,

(2) (i), called the declarator subscript, is composed of 1, 2, or 3 expressions, each of which may be an integer constant or an integer variable name. Each expression is separated by a comma from its successor if there are more than one of them. In the case where i contains no integer variable, i is called the constant declarator subscript.

The appearance of a declarator subscript in a declarator statement serves to inform the processor that the declarator name is an array name. The number of subscript expressions specified for the array indicates its dimensionality. The magnitude of the values given for the subscript expressions indicates the maximum value that the subscript may attain in any array element name.

No array element name may contain a subscript that, during execution of the executable program, assumes a value less than one or larger than the maximum length specified in the array declarator.

7.2.1.1.1 *Array Element Successor Function and Value of a Subscript.* For a given dimensionality, subscript declarator, and subscript, the value of a subscript pointing to an array element and the maximum value a subscript may attain is indicated in Table 2. A subscript expression must be greater than zero.

The value of the array element successor function is obtained by adding one to the entry in the subscript value column. Any array element whose subscript has this value is the successor to the original element. The last element of the array is the one whose subscript value is the maximum subscript value and has no successor element.

TABLE 2. Value of a Subscript

Dimensionality	*Subscript Declarator*	*Subscript*	*Subscript Value*	*Maximum Subscript Value*
1	(A)	(a)	a	$A \cdot$
2	(A, B)	(a, b)	$a + A \cdot (b - 1)$	$A \cdot B$
3	(A, B, C)	(a, b, c)	$a + A \cdot (b - 1) +$ $A \cdot B \cdot (c - 1)$	$A \cdot B \cdot C$

Notes. (1) a, b, and c are subscript expressions.
(2) A, B, and C are dimensions.

7.2.1.1.2 *Adjustable Dimension.* If any of the entires in a declarator subscript is an integer variable name, the array is called an adjustable array, and the variable names are called adjustable dimensions. Such an array may only appear in a procedure subprogram. The dummy argument list of the subprograms must contain the array name and the integer variable names that represent the adjustable dimensions. The values of the actual arguments that represent array dimensions in the argument list of the reference must be defined (10.2) prior to calling the subprogram and may not be redefined or undefined during execution of the subprogram. The maximum size of the actual array may not be exceeded. For every array appearing in an executable program (9.1.6), there must be at least one constant array declarator associated through subprogram.

In a subprogram, a symbolic name that appears in a COMMON statement may not identify an adjustable array.

7.2.1.2 DIMENSION *Statement.* A DIMENSION statement is of the form:

$$\text{DIMENSION } v_1(i_1), v_2(i_2), \cdots, v_n(i_n)$$

where each $v(i)$ is an array declarator.

7.2.1.3 COMMON *Statement.* A COMMON statement is of the form:

$$\text{COMMON } / x_1 / a_1 / \cdots / x_n / a_n$$

where each a is a nonempty list of variable names, array names, or array declarators (no dummy arguments are permitted) and each x is a symbolic name or is empty. If x_1 is empty, the first two slashes are optional. Each x is a block name, a name that bears no relationship to any variable or array having the same name. This holds true for any such variable or array in the same or any other program unit. See 10.1.1 for a discussion of restrictions on uses of block names.

In any given COMMON statement, the entities occurring between block name x and the next block name (or the end of the statement if no block name follows) are declared to be in common block x. All entities from the beginning of the statement until the appearance of a block name, or all entities in the statement if no block name appears, are declared to be in blank or unlabeled common. Alternatively, the appearance of two slashes with no block name between them declares the entities that follow to be in blank common.

A given common block name may occur more than once in a COMMON statement or in a program unit. The processor will string together in a given common block all entities so assigned in the order of their appearance (10.1.2). The first element of an array will follow the immediately preceding entity, if one exists, and the last element of an array will immediately precede the next entity, if one exists.

The size of a common block in a program unit is the sum of the storage required for the elements introduced through COMMON and EQUIVALENCE statements. The sizes of labeled common blocks with the same label in the program units that comprise an executable program must be the same. The sizes of blank common in the various program units that are to be executed together need not be the same. Size is measured in terms of storage units (7.2.1.3.1).

7.2.1.3.1 *Correspondence of Common Blocks.* If all of the program units of an executable program that contain any definition of a common block of a particular name define that block such that:

(1) There is identity in type for all entities defined in the corresponding position from the beginning of that block,

(2) If the block is labeled and the same number of entities is defined for the block.

Then the values in the corresponding positions (counted by the number of preceding storage units) are the same quantity in the executable program.

A double precision or a complex entity is counted as two logically consecutive storage units; a logical, real, or integer entity, as one storage unit.

Then for common blocks with the same number of storage units or blank common:

(1) In all program units which have defined the identical type to a given position (counted by the number of preceding storage units) references to that position refer to the same quantity.

(2) A correct reference is made to a particular position assuming a given type if the most recent value assignment to that position was of the same type.

7.2.1.4 EQUIVALENCE *Statement.* An EQUIVALENCE statement is of the form:

$$\text{EQUIVALENCE } (k_1), (k_2), \cdots, (k_n)$$

in which each k is a list of the form:

$$a_1, a_2, \cdots, a_m.$$

Each a is either a variable name or an array element name (not a dummy argument), the subscript of which contains only constants, and m is greater than or equal to two. The number of subscript expressions of an array element name must correspond in number to the dimensionality of the array declarator or must be one (the array element successor function defines a relation by which an array can be made equivalent to a one dimensional array of the same length).

The EQUIVALENCE statement is used to permit the sharing of storage by two or more entities. Each element in a given list is assigned the same storage (or part of the same storage) by the processor. The EQUIVALENCE statement should not be used to equate mathematically two or more entities. If a two storage unit entity is equivalenced to a one storage unit entity, the latter will share space with the first storage unit of the former.

The assignment of storage to variables and arrays declared directly in a COMMON statement is determined solely by consideration of their type and the COMMON and array declarator statements. Entities so declared are always assigned unique storage, contiguous in the order declared in the COMMON statement.

The effect of an EQUIVALENCE statement upon common assignment may be the lengthening of a common block; the only such lengthening permitted is that which extends a common block beyond the last assignment for that block made directly by a COMMON statement.

When two variables or array elements share storage because of the effects of EQUIVALENCE statements, the symbolic names of the variables or arrays in question may not both appear in COMMON statements in the same program unit.

Information contained in 7.2.1.1.1, 7.2.1.3.1, and the present section suffices to describe the possibilities of additional cases of sharing of storage between array elements and entities of common blocks. It is incorrect to cause either directly or indirectly a single storage unit to contain more than one element of the same array.

7.2.1.5 EXTERNAL *Statement.* An EXTERNAL statement is of the form:

$$\text{EXTERNAL } v_1, v_2, \cdots, v_n$$

where each v is an external procedure name.

Appearance of a name in an EXTERNAL statement declares that name to be an external procedure name. If an external procedure name is used as an argument to another external procedure, it must appear in an EXTERNAL statement in the program unit in which it is so used.

7.2.1.6 *Type-statements.* A type-statement is of the form:

$$t\, v_1, v_2, \cdots, v_n$$

where t is INTEGER, REAL, DOUBLE PRECISION, COMPLEX, or LOGICAL, and each *v* is a variable name, an array name, a function name, or an array declarator.

A type-statement is used to override or confirm the implicit typing, to declare entities to be of type double precision, complex, or logical, and may supply dimension information.

The appearance of a symbolic name in a type-statement serves to inform the processor that it is of the specified data type for all appearances in the program unit. See, however, the restriction in 8.3.1 second paragraph.

7.2.2 *Data Initialization Statement.* A data initialization statement is of the form:

DATA $k_1 \, / \, d_1 \, / \, , \, k_2 \, / \, d_2 \, / \, , \cdots , \, k_n \, / \, d_n \, /$

where:

(1) Each *k* is a list containing names of variables and array elements,

(2) Each *d* is a list of constants and optionally signed constants, any of which may be preceded by *j**,

(3) *j* is an integer constant.

If a list contains more than one entry, the entries are separated by commas.

Dummy arguments may not appear in the list *k*. Any subscript expression must be an integer constant.

When the form *j** appears before a constant it indicates that the constant is to be specified *j* times. A Hollerith constant may appear in the list *d*.

A data initialization statement is used to define initial values of variables or array elements. There must be a one-to-one correspondence between the list-specified items and the constants. By this correspondence, the initial value is established.

An initially defined variable or array element may not be in blank common. A variable or array element in a labeled common block may be initially defined only in a block data subprogram.

7.2.3 FORMAT *Statement.* FORMAT statements are used in conjunction with the input/output of formatted records to provide conversion and editing information between the internal representation and the external character strings.

A FORMAT statement is of the form:

FORMAT $(q_1 t_1 z_1 t_2 z_2 \cdots z_{n-1} t_n q_2)$

where:

(1) $(q_1 t_1 z_1 t_2 z_2 \cdots z_{n-1} t_n q_2)$ is the format specification.

(2) Each *q* is a series of slashes or is empty.

(3) Each *t* is a field descriptor or group of field descriptors.

(4) Each *z* is a field separator.

(5) *n* may be zero.

A FORMAT statement must be labeled.

7.2.3.1 *Field Descriptors.* The format field descriptors are of the forms:

*sr*F*w*.*d*
*sr*E*w*.*d*
*sr*G*w*.*d*
*sr*D*w*.*d*
*r*I*w*
*r*L*w*
*r*A*w*
*n*H$h_1 h_2 \cdots h_n$
*n*X

where:

(1) The letters F, E, G, D, I, L, A, H, and X indicate the manner of conversion and editing between the internal and external representations and are called the conversion codes.

(2) *w* and *n* are nonzero integer constants representing the width of the field in the external character string.

(3) *d* is an integer constant representing the number of digits in the fractional part of the external character string (except for G conversion code).

(4) *r*, the repeat count, is an optional nonzero integer constant indicating the number of times to repeat the succeeding basic field descriptor.

(5) *s* is optional and represents a scale factor designator.

(6) Each *h* is one of the characters capable of representation by the processor.

For all descriptors, the field width must be specified. For descriptors of the form *w*.*d* , the *d* must be specified, even if it is zero. Further, *w* must be greater than or equal to *d*.

The phrase *basic field descriptor* will be used to signify the field descriptor unmodified by *s* or *r*.

The internal representation of external fields will correspond to the internal representation of the corresponding type constants (4.2 and 5.1.1).

7.2.3.2 *Field Separators.* The format field separators are the slash and the comma. A series of slashes is also a field separator. The field descriptors or groups of field descriptors are separated by a field separator.

The slash is used not only to separate field descriptors, but to specify demarcation of formatted records. A formatted record is a string of characters. The lengths of the strings for a given external medium are dependent upon both the processor and the external medium.

The processing of the number of characters that can be contained in a record by an external medium does not of itself cause the introduction or inception of processing of the next record.

7.2.3.3 *Repeat Specifications.* Repetition of the field descriptors (except *n*H and *n*X) is accomplished by using the repeat count. If the input/output list warrants, the specified conversion will be interpreted repetitively up to the specified number of times.

Repetition of a group of field descriptors or field separators is accomplished by enclosing them within parentheses and optionally preceding the left parenthesis with an integer constant called the group repeat count indicating the number of times to interpret the enclosed grouping. If no group repeat count is specified, a group repeat count of one is assumed. This form of grouping is called a basic group.

A further grouping may be formed by enclosing field descriptors, field separators, or basic groups within parentheses. Again, a group repeat count may be specified. The parentheses enclosing the format specification are not considered as group delineating parentheses.

7.2.3.4 *Format Control Interaction with an Input/Output List.* The inception of execution of a formatted READ or formatted WRITE statement initiates format control. Each action of format control depends on information jointly provided respectively by the next element of the input/output list, if one exists, and the next field descriptor obtained from the format specification. If there is an input/output list, at least one field descriptor other than *n*H or *n*X must exist.

When a READ statement is executed under format control, one record is read when the format control is initiated, and thereafter additional records are read only as the format specification demands. Such action may not require more characters of a record than it contains.

When a WRITE statement is executed under format control, writing of a record occurs each time the format specification demands that a new record be started. Termination of format control causes writing of the current record.

Except for the effects of repeat counts, the format specification is interpreted from left to right.

To each I, F, E, G, D, A, or L basic descriptor interpreted in a format specification, there corresponds one element specified by the input/output list, except that a complex element requires the interpretation of two F, E, or G basic descriptors. To each H or X basic descriptor there is no corresponding element specified by the input/output list, and the format control communicates information directly with the record. Whenever a slash is encountered, the format specification demands that a new record start or the preceding record terminate. During a READ operation, any unprocessed characters of the current record will be skipped at the time of termination of format control or when a slash is encountered.

Whenever the format control encounters an I, F, E, G, D, A, or L basic descriptor in a format specification, it determines if there is a corresponding element specified by the input/output list. If there is such an element, it transmits appropriately converted information between the element and the record and proceeds. If there is no corresponding element, the format control terminates.

If, however, the format control proceeds to the last outer right parenthesis of the format specification, a test is made to determine if another list element is specified. If not, control terminates. However, if another list element is specified, the format control demands a new record start and control reverts to that group repeat specification terminated by the last preceding right parenthesis, or if none exists, then to the first left parenthesis of the format specification. Note, this action of itself has no effect on the scale factor.

7.2.3.5 *Scale Factor.* A scale factor designator is defined for use with the F, E, G, and D conversions and is of the form:

$$n\text{P}$$

where n, the scale factor, is an integer constant or minus followed by an integer constant.

When the format control is initiated, a scale factor of zero is established. Once a scale factor has been established, it applies to all subsequently interpreted F, E, G, and D field descriptors, until another scale factor is encountered, and then that scale factor is established.

7.2.3.5.1 *Scale Factor Effects.* The scale factor n affects the appropriate conversions in the following manner:

(1) For F, E, G, and D input conversions (provided no exponent exists in the external field) and F output conversions, the scale factor effect is as follows:

> externally represented number equals internally represented number times the quantity ten raised to the nth power.

(2) For F, E, G, and D input, the scale factor has no effect if there is an exponent in the external field.

(3) For E and D output, the basic real constant part of the output quantity is multiplied by 10^n and the exponent is reduced by n.

(4) For G output, the effect of the scale factor is suspended unless the magnitude of the datum to be converted is outside the range that permits the effective use of F conversion. If the effective use of E conversion is required, the scale factor has the same effect as with E output.

7.2.3.6 *Numeric Conversions.* The numeric field descriptors I, F, E, G, and D are used to specify input/output of integer real, double precision, and complex data.

(1) With all numeric input conversions, leading blanks are not significant and other blanks are zero. Plus signs may be omitted. A field of all blanks is considered to be zero.

(2) With the F, E, G, and D input conversions, a decimal point appearing in the input field overrides the decimal point specification supplied by the field descriptor.

(3) With all output conversions, the output field is right justified. If the number of characters produced by the conversion is smaller than the field width, leading blanks will be inserted in the output field.

(4) With all output conversions, the external representation of a negative value must be signed; a positive value may be signed.

(5) The number of characters produced by an output conversion must not exceed the field width.

7.2.3.6.1 *Integer Conversion.* The numeric field descriptor Iw indicates that the external field occupies w positions as an integer. The value of the list item appears, or is to appear, internally as an integer datum.

In the external input field, the character string must be in the form of an integer constant or signed integer constant (5.1.1.1), except for the interpretation of blanks (7.2.3.6).

The external output field consists of blanks, if necessary, followed by a minus if the value of the internal datum is negative, or an optional plus otherwise, followed by the magnitude of the internal value converted to an integer constant.

7.2.3.6.2 *Real Conversions.* There are three conversions available for use with real data: F, E, and G.

The numeric field descriptor F$w.d$ indicates that the external field occupies w positions, the fractional part of which consists of d digits. The value of the list item appears, or is to appear, internally as a real datum.

The basic form of the external input field consists of an optional sign, followed by a string of digits optionally containing a decimal point. The basic form may be followed by an exponent of one of the following forms:

(1) Signed integer constant.
(2) E followed by an integer constant.
(3) E followed by a signed integer constant.
(4) D followed by an integer constant.
(5) D followed by a signed integer constant.

An exponent containing D is equivalent to an exponent containing E.

The external output field consists of blanks, if necessary, followed by a minus if the internal value is negative, or an optional plus otherwise, followed by string of digits containing a decimal point representing the magnitude of the internal value, as modified by the established scale factor, rounded to d fractional digits.

The numeric field descriptor E$w.d$ indicates that the external field occupies w positions, the fractional part of which consists of d digits. The value of the list item appears, or is to appear, internally as a real datum.

The form of the external input field is the same as for the F conversion.

The standard form of the external output field for a scale factor of zero is[1]

$$\xi 0.x_1 \cdots x_d Y$$

[1] ξ signifies no character position or minus in that position.

where:

(1) $x_1 \cdots x_d$ are the d most significant rounded digits of the value of the data to be output.

(2) Y is of one of the forms:

$$E \pm y_1y_2 \quad \text{or} \quad \pm y_1y_2y_3$$

and has the significance of a decimal exponent (an alternative for the plus in the first of these forms is the character blank).

(3) The digit 0 in the aforementioned standard form may optionally be replaced by no character position.

(4) Each y is a digit.

The scale factor n controls the decimal normalization between the number part and the exponent part such that:

(1) If $n \leqq 0$, there will be exactly $-n$ leading zeros and $d + n$ significant digits after the decimal point.

(2) If $n > 0$, there will be exactly n significant digits to the left of the decimal point and $d - n + 1$ to the right of the decimal point.

The numeric field descriptor G*w.d* indicates that the external field occupies w positions with d significant digits. The value of the list item appears, or is to appear, internally as a real datum.

Input processing is the same as for the F conversion.

The method of representation in the external output string is a function of the magnitude of the real datum being converted. Let N be the magnitude of the internal datum. The following tabulation exhibits a correspondence between N and the equivalent method of conversion that will be effected:

Magnitude of Datum	*Equivalent Conversion Effected*
$0.1 \leqq N < 1$	F$(w-4).d$, 4X
$1 \leqq N < 10$	F$(w-4).(d-1)$, 4X
$\vdots$	$\vdots$
$10^{d-2} \leqq N < 10^{d-1}$	F$(w-4)$.1, 4X
$10^{d-1} \leqq N < 10^{d}$	F$(w-4)$.0, 4X
Otherwise	sE*w.d*

Note that the effect of the scale factor is suspended unless the magnitude of the datum to be converted is outside of the range that permits effective use of F conversion.

7.2.3.6.3 *Double Precision Conversion.* The numeric field descriptor D*w.d* indicates that the external field occupies w positions, the fractional part of which consists of d digits. The value of the list item appears, or is to appear, internally as a double precision datum.

The basic form of the external input field is the same as for real conversions.

The external output field is the same as for the E conversion, except that the character D may replace the character E in the exponent.

7.2.3.6.4 *Complex Conversion.* Since a complex datum consists of a pair of separate real data, the conversion is specified by two successively interpreted real field descriptors. The first of these supplies the real part. The second supplies the imaginary part.

7.2.3.7 *Logical Conversion.* The logical field descriptor L*w* indicates that the external field occupies w positions as a string of information as defined below. The list item appears, or is to appear, internally as a logical datum.

The external input field must consist of optional blanks followed by a T or F followed by optional characters, for true and false, respectively.

The external output field consists of $w - 1$ blanks followed by a T or F as the value of the internal datum is true or false, respectively.

7.2.3.8 *Hollerith Field Descriptor.* Hollerith information may be transmitted by means of two field descriptors, *n*H and A*w*:

(1) The *n*H descriptor causes Hollerith information to be read into, or written from, the n characters (including blanks) following the *n*H descriptor in the format specification itself.

(2) The A*w* descriptor causes w Hollerith characters to be read into, or written from, a specified list element.

Let g be the number of characters representable in a single storage unit (7.2.1.3.1). If the field width specified for A input is greater than or equal to g, the rightmost g characters will be taken from the external input field. If the field width is less than g, the w characters will appear left justified with $w - g$ trailing blanks in the internal representation.

If the field width specified for A output is greater than g, the external output field will consist of $w - g$ blanks, followed by the g characters from the internal representation. If the field width is less than or equal to g, the external output field will consist of the leftmost w characters from the internal representation.

7.2.3.9 *Blank Field Descriptor.* The field descriptor for blanks is *n*X . On input, n characters of the external input record are skipped. On output, n blanks are inserted in the external output record.

7.2.3.10 *Format Specification in Arrays.* Any of the formatted input/output statements may contain an array name in place of the reference to a FORMAT statement label. At the time an array is referenced in such a manner, the first part of the information contained in the array, taken in the natural order, must constitute a valid format specification. There is no requirement on the information contained in the array following the right parenthesis that ends the format specification.

The format specification which is to be inserted in the array has the same form as that defined for a FORMAT statement; that is, begins with a left parenthesis and ends with a right parenthesis. An *n*H field descriptor may not be part of a format specification within an array.

The format specification may be inserted in the array by use of a data initialization statement, or by use of a READ statement together with an A format.

8. PROCEDURES AND SUBPROGRAMS

There are four categories of procedures: statement functions, intrinsic functions, external functions, and external subroutines. The first three categories are referred to collectively as functions or function procedures; the last as subroutines or subroutine procedures. There are two categories of subprograms: procedure subprograms and specification subprograms. Function subprograms and subroutine subprograms are classified as procedure subprograms. Block data subprograms are classified as specification subprograms. Type rules for function procedures are given in 5.3.

8.1 Statement Functions. A statement function is defined internally to the program unit in which it is referenced. It is defined by a single statement similar in form to an arithmetic or logical assignment statement.

In a given program unit, all statement function definitions must precede the first executable statement of the program unit and must follow the specification statements, if any. The name of a statement function must not appear in an EXTERNAL statement, nor as a variable name or an array name in the same program unit.

8.1.1 *Defining Statement Functions.* A statement function is defined by a statement of the form:

$$f(a_1, a_2, \cdots, a_n) = e$$

where f is the function name, e is an expression, and the relationship between f and e must conform to the assignment rules in 7.1.1.1 and 7.1.1.2. The a's are distinct variable names, called the *dummy arguments* of the function. Since these are dummy arguments, their names, which serve only to indicate type, number, and order of arguments, may be the same as variable names of the same type appearing elsewhere in the program unit.

Aside from the dummy arguments, the expression e may only contain:

(1) Non-Hollerith constants.

(2) Variable references.

(3) Intrinsic function references.

(4) References to previously defined statement functions.

(5) External function references.

8.1.2 *Referencing Statement Functions.* A statement function is referenced by using its reference (5.2) as a primary in an arithmetic or logical expression. The actual arguments, which constitute the argument list, must agree in order, number, and type with the corresponding dummy arguments. An actual argument in a statement function reference may be any expression of the same type as the corresponding dummy argument.

Execution of a statement function reference results in an association (10.2.2) of actual argument values with the corresponding dummy arguments in the expression of the function definition, and an evaluation of the expression. Following this, the resultant value is made available to the expression that contained the function reference.

8.2 Intrinsic Functions and Their Reference. The symbolic names of the intrinsic functions (see Table 3) are predefined to the processor and have a special meaning and type if the name satisfies the conditions of 10.1.7.

An intrinsic function is referenced by using its reference as a primary in an arithmetic or logical expression. The actual arguments, which constitute the argument list, must agree in type, number, and order with the specification in Table 3 and may be any expression of the specified type. The intrinsic functions AMOD, MOD, SIGN, ISIGN, and DSIGN are not defined when the value of the second argument is zero.

Execution of an intrinsic function reference results in the actions specified in Table 3 based on the values of the actual arguments. Following this, the resultant value is made available to the expression that contained the function reference.

8.3 External Functions. An external function is defined externally to the program unit that references it. An external function defined by Fortran statements headed by a FUNCTION statement is called a function subprogram.

8.3.1 *Defining Function Subprograms.* A FUNCTION statement is of the form:

$$t \text{ FUNCTION } f \ (a_1, a_2, \cdots, a_n)$$

where:

(1) t is either INTEGER, REAL, DOUBLE PRECISION, COMPLEX, or LOGICAL, or is empty.

(2) f is the symbolic name of the function to be defined.

(3) The a's, called the dummy arguments, are each either a variable name, an array name, or an external procedure name.

TABLE 3. Intrinsic Functions

Intrinsic Function	*Definition*	*Number of Arguments*	*Symbolic Name*	*Type of:*	
				Argument	*Function*
Absolute Value	$\lvert a \rvert$	1	ABS IABS DABS	Real Integer Double	Real Integer Double
Truncation	Sign of a times largest integer $\leq \lvert a \rvert$	1	AINT INT IDINT	Real Real Double	Real Integer Integer
Remaindering* (see note below)	$a_1 \text{ (mod } a_2)$	2	AMOD MOD	Real Integer	Real Integer
Choosing Largest Value	Max $(a_1, a_2, \cdots)$	≧2	AMAX0 AMAX1 MAX0 MAX1 DMAX1	Integer Real Integer Real Double	Real Real Integer Integer Double
Choosing Smallest Value	Min $(a_1, a_2, \cdots)$	≧2	AMIN0 AMIN1 MIN0 MIN1 DMIN1	Integer Real Integer Real Double	Real Real Integer Integer Double
Float	Conversion from integer to real	1	FLOAT	Integer	Real
Fix	Conversion from real to integer	1	IFIX	Real	Integer
Transfer of Sign	Sign of a_2 times $\lvert a_1 \rvert$	2	SIGN ISIGN DSIGN	Real Integer Double	Real Integer Double
Positive Difference	$a_1 -$ Min (a_1, a_2)	2	DIM IDIM	Real Integer	Real Integer
Obtain Most Significant Part of Double Precision Argument		1	SNGL	Double	Real
Obtain Real Part of Complex Argument		1	REAL	Complex	Real
Obtain Imaginary Part of Complex Argument		1	AIMAG	Complex	Real
Express Single Precision Argument in Double Precision Form		1	DBLE	Real	Double
Express Two Real Arguments in Complex Form	$a_1 + a_2\sqrt{-1}$	2	CMPLX	Real	Complex
Obtain Conjugate of a Complex Argument		1	CONJG	Complex	Complex

*The function MOD or AMOD (a_1, a_2) is defined as $a_1 - [a_1/a_2]a_2$, where $[x$ is the integer whose magnitude does not exceed the magnitude of x and whose sign is the same as x.

Function subprograms are constructed as specified in 9.1.3 with the following restrictions:

(1) The symbolic name of the function must also appear as a variable name in the defining subprogram. During every execution of the subprogram, this variable must be defined and, once defined, may be referenced or redefined. The value of the variable at the time of execution of any RETURN statement in this subprogram is called the value of the function.

(2) The symbolic name of the function must not appear in any nonexecutable statement in this program unit, except as the symbolic name of the function in the FUNCTION statement.

(3) The symbolic names of the dummy arguments may not appear in an EQUIVALENCE, COMMON, or DATA statement in the function subprogram.

(4) The function subprogram may define or redefine one or more of its arguments so as to effectively return results in addition to the value of the function.

(5) The function subprogram may contain any statements except BLOCK DATA, SUBROUTINE, another FUNCTION statement, or any statement that directly or indirectly references the function being defined.

(6) The function subprogram must contain at least one RETURN statement.

8.3.2 *Referencing External Functions.* An external function is referenced by using its reference (5.2) as a primary in an arithmetic or logical expression. The actual arguments, which constitute the argument list, must agree in order, number, and type with the corresponding dummy arguments in the defining program unit. An actual argument in an external function reference may be one of the following:

(1) A variable name.
(2) An array element name.
(3) An array name.
(4) Any other expression.
(5) The name of an external procedure.

If an actual argument is an external function name or a subroutine name, then the corresponding dummy argument must be used as an external function name or a subroutine name, respectively.

If an actual argument corresponds to a dummy argument that is defined or redefined in the referenced subprogram, the actual argument must be a variable name, an array element name, or an array name. Execution of an external function reference as described in the foregoing, results in an association (10.2.2) of actual arguments with all appearances of dummy arguments in executable statements, function definition statements, and as adjustable dimensions in the defining subprogram. If the actual argument is as specified in item (4) in the foregoing, this association is by value rather than by name. Following these associations, execution of the first executable statement of the defining subprogram is undertaken. An actual argument which is an array element name containing variables in the subscript could in every case be replaced by the same argument with a constant subscript containing the same values as would be derived by computing the variable subscript just before the association of arguments takes place.

If a dummy argument of an external function is an array name, the corresponding actual argument must be an array name or array element name (10.1.3).

If a function reference causes a dummy argument in the referenced function to become associated with another dummy argument in the same function or with an entity in common, a definition of either within the function is prohibited.

Unless it is a dummy argument, an external function is also referenced (in that it must be defined) by the appearance of its symbolic name in an EXTERNAL statement.

8.3.3 *Basic External Functions.* FORTRAN processors must supply the external functions listed in Table 4. Referencing of these functions is accomplished as described in (8.3.2). Arguments for which the result of these functions is not mathematically defined or is of type other than that specified are improper.

8.4 SUBROUTINE. An external subroutine is defined externally to the program unit that references it. An external subroutine defined by FORTRAN statements headed by a SUBROUTINE statement is called a subroutine subprogram.

TABLE 4. BASIC EXTERNAL FUNCTIONS

Basic External Function	*Definition*	*Number of Arguments*	*Symbolic Name*	*Type of:*	
				Argument	*Function*
Exponential	e^a	1	EXP	Real	Real
		1	DEXP	Double	Double
		1	CEXP	Complex	Complex
Natural Logarithm	$\log_e (a)$	1	ALOG	Real	Real
		1	DLOG	Double	Double
		1	CLOG	Complex	Complex
Common Logarithm	$\log_{10} (a)$	1	ALOG10	Real	Real
			DLOG10	Double	Double
Trigonometric Sine	$\sin (a)$	1	SIN	Real	Real
		1	DSIN	Double	Double
		1	CSIN	Complex	Complex
Trigonometric Cosine	$\cos (a)$	1	COS	Real	Real
		1	DCOS	Double	Double
		1	CCOS	Complex	Complex
Hyperbolic Tangent	$\tanh (a)$	1	TANH	Real	Real
Square Root	$(a)^{1/2}$	1	SQRT	Real	Real
		1	DSQRT	Double	Double
		1	CSQRT	Complex	Complex
Arctangent	$\arctan (a)$	1	ATAN	Real	Real
		1	DATAN	Double	Double
	$\arctan (a_1/a_2)$	2	ATAN2	Real	Real
		2	DATAN2	Double	Double
Remaindering*	$a_1 \pmod{a_2}$	2	DMOD	Double	Double
Modulus		1	CABS	Complex	Real

*The function DMOD (a_1, a_2) is defined as $a_1 - [a_1/a_2]a_2$, where $[x]$ is the integer whose magnitude does not exceed the magnitude of x and whose sign is the same as the sign of x.

8.4.1 *Defining Subroutine Subprograms.* A SUBROUTINE statement is of one of the forms:

SUBROUTINE s $(a_1, a_2, \cdots, a_n)$
or
SUBROUTINE s

where:

(1) s is the symbolic name of the subroutine to be defined.

(2) The a's, called the dummy arguments, are each either a variable name, an array name, or an external procedure name.

Subroutine subprograms are constructed as specified in 9.1.3 with the following restrictions:

(1) The symbolic name of the subroutine must not appear in any statement in this subprogram except as the symbolic name of the subroutine in the SUBROUTINE statement itself.

(2) The symbolic names of the dummy arguments may not appear in an EQUIVALENCE, COMMON, or DATA statement in the subprogram.

(3) The subroutine subprogram may define or redefine one or more of its arguments so as to effectively return results.

(4) The subroutine subprogram may contain any statements except BLOCK DATA, FUNCTION, another SUB-

ROUTINE statement, or any statement that directly or indirectly references the subroutine being defined.

(5) The subroutine subprogram must contain at least one RETURN statement.

8.4.2 *Referencing Subroutines.* A subroutine is referenced by a CALL statement (7.1.2.4). The actual arguments, which constitute the argument list, must agree in order, number, and type with the corresponding dummy arguments in the defining program. The use of a Hollerith constant as an actual argument is an exception to the rule requiring agreement of type. An actual argument in a subroutine reference may be one of the following:

(1) A Hollerith constant.
(2) A variable name.
(3) An array element name.
(4) An array name.
(5) Any other expression.
(6) The name of an external procedure.

If an actual argument is an external function name or a subroutine name, the corresponding dummy argument must be used as an external function name or a subroutine name, respectively.

If an actual argument corresponds to a dummy argument that is defined or redefined in the referenced subprogram, the actual argument must be a variable name, an array element name, or an array name.

Execution of a subroutine reference as described in the foregoing results in an association of actual arguments with all appearances of dummy arguments in executable statements, function definition statements, and as adjustable dimensions in the defining subprogram. If the actual argument is as specified in item (5) in the foregoing, this association is by value rather than by name. Following these associations, execution of the first executable statement of the defining subprogram is undertaken.

An actual argument which is an array element name containing variables in the subscript could in every case be replaced by the same argument with a constant subscript containing the same values as would be derived by computing the variable subscript just before the association of arguments takes place.

If a dummy argument of an external function is an array name, the corresponding actual argument must be an array name or array element name (10.1.3).

If a subroutine reference causes a dummy argument in the referenced subroutine to become associated with another dummy argument in the same subroutine or with an entity in common, a definition of either entity within the subroutine is prohibited.

Unless it is a dummy argument, a subroutine is also referenced (in that it must be defined) by the appearance of its symbolic name in an EXTERNAL statement.

8.5 Block Data Subprogram. A BLOCK DATA statement is of the form:

BLOCK DATA

This statement may only appear as the first statement of specification subprograms that are called block data subprograms, and that are used to enter initial values into elements of labeled common blocks. This special subprogram contains only type-statements, EQUIVALENCE, DATA, DIMENSION, and COMMON statements.

If any entity of a given common block is being given an initial value in such a subprogram, a complete set of specification statements for the entire block must be included, even though some of the elements of the block do not appear in DATA statements. Initial values may be entered into more than one block in a single subprogram.

9. PROGRAMS

An executable program is a collection of statements, comment lines, and end lines that completely (except for input data values and their effects) describe a computing procedure.

9.1 Program Components. Programs consist of program parts, program bodies, and subprogram statements.

9.1.1 *Program Part.* A program part must contain at least one executable statement and may contain FORMAT statements, and data initialization statements. It need not contain any statements from either of the latter two classes of statement. This collection of statements may optionally be preceded by statement function definitions, data initialization statements, and FORMAT statements. As before only some or none of these need be present.

9.1.2 *Program Body.* A program body is a collection of specification statements, FORMAT statements or both, or neither, followed by a program part, followed by an end line.

9.1.3 *Subprogram.* A subprogram consists of a SUBROUTINE or FUNCTION statement followed by a program body, or is a block data subprogram.

9.1.4 *Block Data Subprogram.* A block data subprogram consists of a BLOCK DATA statement, followed by the appropriate (8.5) specification statements, followed by data initialization statements, followed by an end line.

9.1.5 *Main Program.* A main program consists of a program body.

9.1.6 *Executable Program.* An executable program consists of a main program plus any number of subprograms, external procedures, or both.

9.1.7 *Program Unit.* A program unit is a main program or a subprogram.

9.2 Normal Execution Sequence. When an executable program begins operation, execution commences with the execution of the first executable statement of the main program. A subprogram, when referenced, starts execution with execution of the first executable statement of that subprogram. Unless a statement is a GO TO, arithmetic IF, RETURN, or STOP statement or the terminal statement of a DO, completion of execution of that statement causes execution of the next following executable statement. The sequence of execution following execution of any of these statements is described in Section 7. A program part may not (in the sense of 1.1) contain an executable statement that can never be executed.

A program part must contain a first executable statement.

10. INTRA- AND INTERPROGRAM RELATIONSHIPS

10.1 Symbolic Names. A symbolic name has been defined to consist of from one to six alphanumeric characters, the first of which must be alphabetic. Sequences of characters that are format field descriptors or uniquely identify certain statement types, e.g., GO TO, READ, FORMAT, etc. are not symbolic names in such occurrences nor do they form the first characters of symbolic names in these cases. In a program unit, a symbolic name (perhaps qualified by a subscript) must identify an element of one (and usually only one) of the following classes:

Class I	An array and the elements of that array.
Class II	A variable.
Class III	A statement function.
Class IV	An intrinsic function.
Class V	An external function.
Class VI	A subroutine.

Class VII An external procedure which cannot be classified as either a subroutine or an external function in the program unit in question.

Class VIII A block name.

10.1.1 *Restrictions on Class.* A symbolic name in Class VIII in a program unit may also be in any one of the Classes I, II, or III in that program unit.

In the program unit in which a symbolic name in Class V appears immediately following the word FUNCTION in a FUNCTION statement, that name must also be in Class II.

Once a symbolic name is used in Class V, VI, VII, or VIII in any unit of an executable program, no other program unit of that executable program may use that name to identify an entity of these classes other than the one originally identified. In the totality of the program units that make up an executable program, a Class VII name must be associated with a Class V or VI name. Class VII can only exist locally in program units.

In a program unit, no symbolic name can be in more than one class except as noted in the foregoing. There are no restrictions on uses of symbolic names in different program units of an executable program other than those noted in the foregoing.

10.1.2 *Implications of Mentions in Specification and* DATA *Statements.* A symbolic name is in Class I if and only if it appears as a declarator name. Only one such appearance for a symbolic name in a program unit is permitted.

A symbolic name that appears in a COMMON statement (other than as a block name) is either in Class I, or in Class II but not Class V. (8.3.1) Only one such appearance for a symbolic name in a program unit is permitted.

A symbolic name that appears in an EQUIVALENCE statement is either in Class I, or in Class II but not Class V. (8.3.1).

A symbolic name that appears in a type-statement cannot be in Class VI or Class VII. Only one such appearance for a symbolic name in a program unit is permitted.

A symbolic name that appears in an EXTERNAL statement is in either Class V, Class VI, or Class VII. Only one such appearance for a symbolic name in a program unit is permitted.

A symbolic name that appears in a DATA statement is in either Class I, or in Class II but not Class V. (8.3.1) In an executable program, a storage unit (7.2.1.3.1) may have its value initialized one time at the most.

10.1.3 *Array and Array Element.* In a program unit, any appearance of a symbolic name that identifies an array must be immediately followed by a subscript, except for the following cases:

(1) In the list of an input/output statement.

(2) In a list of dummy arguments.

(3) In the list of actual arguments in a reference to an external procedure.

(4) In a COMMON statement.

(5) In a type-statement.

Only when an actual argument of an external procedure reference is an array name or an array element name may the corresponding dummy argument be an array name. If the actual argument is an array name, the length of the dummy argument array must be no greater than the length of the actual argument array. If the actual argument is an array element name, the length of the dummy argument array must be less than or equal to the length of the actual argument array plus one minus the value of the subscript of the array element.

10.1.4 *External Procedures.* The only case when a symbolic name is in Class VII occurs when that name appears only in an EXTERNAL statement and as an actual argument to an external procedure in a program unit.

Only when an actual argument of an external procedure reference is an external procedure name may the corresponding dummy argument be an external procedure name.

In the execution of an executable program, a procedure subprogram may not be referenced twice without the execution of a RETURN statement in that procedure having intervened.

10.1.5 *Subroutine.* A symbolic name is in Class VI if it appears:

(1) Immediately following the word SUBROUTINE in a SUBROUTINE statement.

(2) Immediately following the word CALL in a CALL statement.

10.1.6 *Statement Function.* A symbolic name is in Class III in a program unit if and only if it meets all three of the following conditions:

(1) It does not appear in an EXTERNAL statement nor is it in Class I.

(2) Every appearance of the name, except in a type-statement, is immediately followed by a left parenthesis.

(3) A function defining statement (8.1.1) is present for that symbolic name.

10.1.7 *Intrinsic Function.* A symbolic name is in Class IV in a program unit if and only if it meets all four of the following conditions:

(1) It does not appear in an EXTERNAL statement nor is it in Class I or Class III.

(2) The symbolic name appears in the name column of the table in Section 8.2.

(3) The symbolic name does not appear in a type-statement of type different from the intrinsic type specified in the table.

(4) Every appearance of the symbolic name (except in a type-statement as described in the foregoing) is immediately followed by an actual argument list enclosed in parentheses.

The use of an intrinsic function in a program unit of an executable program does not preclude the use of the same symbolic name to identify some other entity in a different program unit of that executable program.

10.1.8 *External Function.* A symbolic name is in Class V if it:

(1) Appears immediately following the word FUNCTION in a FUNCTION statement

(2) Is not in Class I, Class III, Class IV, or Class VI and appears immediately followed by a left parenthesis on every occurrence except in a type-statement, in an EXTERNAL statement, or as an actual argument. There must be at least one such appearance in the program unit in which it is so used.

10.1.9 *Variable.* In a program unit, a symbolic name is in Class II if it meets all three of the following conditions:

(1) It is not in Class VI or Class VII.

(2) It is never immediately followed by a left parenthesis unless it is immediately preceded by the word FUNCTION in a FUNCTION statement.

(3) It occurs other than in a Class VIII appearance.

10.1.10 *Block Name.* A symbolic name is in Class VIII if and only if it is used as a block name in a COMMON statement.

10.2 DEFINITION. There are two levels of definition of numeric values, first level definition and second level definition. The concept of definition on the first level applies to array elements and variables; that of second level definition to integer variables only. These concepts are defined in terms of progression of execution; and thus, an executable program, complete and in execution, is assumed in what follows.

There are two other varieties of definition that should be noted. The first, effected by GO TO assignment and referring to an integer variable being defined with other than an integer value, is discussed in 7.1.1.3 and 7.1.2.1.2; the second, which refers to when an external procedure may be referenced, will be discussed in the next section.

In what follows, otherwise unqualified use of the terms definition and undefinition (or their alternate forms) as applied to variables and array elements will imply modification by the phrase on the first level.

10.2.1 *Definition of Procedures.* If an executable program contains information describing an external procedure, such an external procedure with the applicable symbolic name is defined for use in that executable program. An external function reference or subroutine reference (as the case may be) to that symbolic name may then appear in the executable program, provided that number of arguments agrees between definition and reference. In addition, for an external function, the type of function must agree between definition and reference. Other restrictions on agreements are contained in 8.3.1., 8.3.2, 8.4.1., 8.4.2., 10.1.3, and 10.1.4.

The basic external functions listed in (8.3.3) are always defined and may be referenced subject to the restrictions alluded to in the foregoing.

A symbolic name in Class III or Class IV is defined for such use.

10.2.2 *Associations That Effect Definition.* Entities may become associated by:

(1) COMMON association.

(2) EQUIVALENCE association.

(3) Argument substitution.

Multiple association to one or more entities can be the result of combinations of the foregoing. Any definition or undefinition of one of a set of associated entities effects the definition or undefinition of each entity of the entire set.

For purposes of definition, in a program unit there is no association between any two entities both of which appear in COMMON statements. Further, there is no other association for common and equivalenced entities other than those stated in 7.2.1.3.1 and 7.2.1.4.

If an actual argument of an external procedure reference is an array name, an array element name, or a variable name, then the discussions in 10.1.3 and 10.2.1 allow an association of dummy arguments with the actual arguments only between the time of execution of the first executable statement of the procedure and the inception of execution of the next encountered RETURN statement of that procedure. Note specifically that this association can be carried through more than one level of external procedure reference.

In what follows, variables or array elements associated by the information in 7.2.1.3.1 and 7.2.1.4 will be equivalent if and only if they are of the same type.

If an entity of a given type becomes defined, then all associated entities of different type become undefined at the same time, while all associated entities of the same type become defined unless otherwise noted.

Association by argument substitution is only valid in the case of identity of type, so the rule in this case is that an entity created by argument substitution is defined at time of entry if and only if the actual argument was defined. If an entity created by argument substitution becomes defined or undefined (while the association exists) during execution of a subprogram, then the corresponding actual entities in all calling program units becomes defined or undefined accordingly.

10.2.3 *Events That Effect Definition.* Variables and array elements become initially defined if and only if their names are associated in a data initialization statement with a constant of the same type as the variable or array in question. Any entity not initially defined is undefined at the time of the first execution of the first executable statement of the main program. Redefinition of a defined entity is always permissible except for certain integer variables (7.1.2.8, 7.1.3.1.1, and 7.2.1.1.2) or certain entities in subprograms (6.4, 8.3.2, and 8.4.2).

Variables and array elements become defined or redefined as follows:

(1) Completion of execution of an arithmetic or logical assignment statement causes definition of the entity that precedes the equals.

(2) As execution of an input statement proceeds, each entity, which is assigned a value of its corresponding type from the input medium, is defined at the time of such association. Only at the completion of execution of the statement do associated entities of the same type become defined.

(3) Completion of execution of a DO statement causes definition of the control variable.

(4) Inception of execution of action specified by a DO-implied list causes definition of the control variable.

Variables and array elements become undefined as follows:

(1) At the time a DO is satisfied, the control variable becomes undefined.

(2) Completion of execution of an ASSIGN statement causes undefinition of the integer variable in the statement.

(3) Certain entities in function subprograms (10.2.9) become undefined.

(4) Completion of execution of action specified by a DO-implied list causes undefinition of the control variable.

(5) When an associated entity of different type becomes defined.

(6) When an associated entity of the same type becomes undefined.

10.2.4 *Entities in Blank Common.* Entities in blank common and those entities associated with them may not be initially defined.

Such entities, once defined by any of the rules previously mentioned, remain defined until they become undefined.

10.2.5 *Entities in Labeled Common.* Entities in labeled common or any associates of those entities may be initially defined.

A program unit contains a labeled common block namε if the name appears as a block name in the program unit. If a main program or referenced subprogram contains a labeled common block name, any entity in the block (and its associates) once defined remain defined until they become undefined.

It should be noted that redefinition of an initially defined entity will allow later undefinition of that entity.

Specifically, if a subprogram contains a labeled common block name that is not contained in any program unit currently referencing the subprogram directly or indirectly, the execution of a RETURN statement in the subprogram causes undefinition of all entities in the block (and their associates) except for initially defined entities that have maintained their initial definitions.

10.2.6 *Entities Not in Common.* An entity not in common except for a dummy argument or the value of a function may be initially defined.

Such entities once defined by any of the rules previously mentioned, remain defined until they become undefined.

If such an entity is in a subprogram, the completion of execution of a RETURN statement in that subprogram causes all such entities and their associates at that time (except for initially defined entities that have not been redefined or become undefined) to become undefined. In this respect, it should be noted that the association between dummy arguments and actual arguments is terminated at the inception of execution of the RETURN statement.

Again, it should be emphasized, the redefinition of an initially defined entity can result in a subsequent undefinition of that entity.

10.2.7 *Basic Block.* In a program unit, a basic block is a group of one or more executable statements defined as follows.

The following statements are block terminal statements:

(1) DO statement.

(2) CALL statement.

(3) GO TO statement of all types.

(4) Arithmetic IF statement.

(5) STOP statement.

(6) RETURN statement.

(7) The first executable statement, if it exists, preceding a statement whose label is mentioned in a GO TO or arithmetic IF statement.

(8) An arithmetic statement in which an integer variable precedes the equals.

(9) A READ statement with an integer variable in the list.

(10) A logical IF containing any of the admissible forms given in the foregoing.

The following statements are block initial statements:

(1) The first executable statement of a program unit.

(2) The first executable statement, if it exists, following a block terminal statement.

Every block initial statement defines a basic block. If that initial statement is also a block terminal statement, the basic block consists of that one statement. Otherwise, the basic block consists of the initial statement and all executable statements that follow until a block terminal statement is encountered. The terminal statement is included in the basic block.

10.2.7.1 *Last Executable Statement.* In a program unit the last executable statement (which cannot be part of a logical IF) must be one of the following statements: GO TO statement, arithmetic IF statement, STOP statement, or RETURN statement.

10.2.8 *Second Level Definition.* Integer variables must be defined on the second level when used in subscripts and computed GO TO statements.

Redefinition of an integer entity causes all associated variables to be undefined for use on the second level during this execution of this program unit until the associated integer variable is explicitly redefined.

Except as just noted, an integer variable is defined on the second level upon execution of the initial statement of a basic block only if both of the following conditions apply:

(1) The variable is used in a subscript or in a computed GO TO in the basic block in question.

(2) The variable is defined on the first level at the time of execution of the initial statement in question.

This definition persists until one of the following happens:

(1) Completion of execution of the terminal statement of the basic block in question.

(2) The variable in question becomes undefined or receives a new definition on the first level.

At this time, the variable becomes undefined on the second level.

In addition, the occurrence of an integer variable in the list of an input statement in which that integer variable appears following in a subscript causes that variable to be defined on the second level. This definition persists until one of the following happens:

(1) Completion of execution of the terminal statement of the basic block containing the input statement.

(2) The variable becomes undefined or receives a new definition on the first level.

An integer variable defined as the control variable of a DO-implied list is defined on the second level over the range of that DO-implied list and only over that range.

10.2.9 *Certain Entities in Function Subprograms.* If a function subprogram is referenced more than once with an identical argument list in a single statement, the execution of that subprogram must yield identical results for those cases mentioned, no matter what the order of evaluation of the statement.

If a statement contains a factor that may not be evaluated (6.4), and if this factor contains a function reference, then all entities that might be defined in that reference become undefined at the completion of evaluation of the expression containing the factor.

10.3 Definition Requirements for Use of Entities. Any variable referenced in a subscript or a computed GO TO must be defined on the second level at the time of this use.

Any variable, array element, or function referenced as a primary in an expression and any subroutine referenced by a CALL statement must be defined at the time of this use. In the case where an actual argument in the argument list of an external procedure reference is a variable name or an array element name, this in itself is not a requirement that the entity be defined at the time of the procedure reference; however, when such an argument is an external procedure name, it must be defined.

Any variable used as an initial value, terminal value, or incrementation value of a DO statement or a DO-implied list must be defined at the time of this use.

Any variable used to identify an input output unit must be defined at the time of this use.

At the time of execution of a RETURN statement in a function subprogram, the value (8.3.1) of that function must be defined.

At the time of execution of an output statement, every entity whose value is to be transferred to the output medium must be defined unless the output is under control of a format specification and the corresponding conversion code is A. If the output is under control of a format specification, a correct association of conversion code with type of entity is required unless the conversion code is A. The following are the correct associations: I with integer; D with double precision; E, F, and G with real and complex; and L with logical.

APPENDIX H

S-C 4060 TEST PROGRAMS

1. Card reader test (for card reader option)
2. Central processor test
3. Core memory test #1 (tests contiguous memory limits)
4. Core memory test #2 (to insure that the contents of core memory are not disturbed when AC power is disconnected)
5. Power failure interrupt test
6. Teleprinter test for ASR-33 keyboard, reader, and punch
7. Alignment and performance test for print head section
8. DAP assembler test
9. Magnetic tape read/write test

APPENDIX I

Examples of ASR-33 Operator Instructions

A complete list of instructions is contained in Document Number HMO-208, S-C 4060 Stored Program Recording System, Operator's Handbook.

Input	Action	Normal Reply
LOAD p	The specified processor will be loaded from the library. p may be 4020, META, P704, P709, P360	**OK or **P NOT FOUND
S	Execution will halt	**STOPPED
REWTn	Tape n will be rewound	**OK
BKSRn	Tape n will backspace one record	**OK
BKSFn	Tape n will backspace one file	**OK
SKPRn	Tape n will skip forward one file	**OK
SKPFn	Tape n will skip forward one file	**OK
RESTART	Input tape will backspace to beginning of the current job and system will be initialized.	**READY
NEXT	Input tape will skip forward to beginning of next job and system will be initialized.	**READY
START	Execution of the process will begin	**START JOB n
GO	Execution will be resumed	**OK
STATUS	System status will be printed on ASR-33. Execution will be resumed upon completion of printout.	JOB NO., FRAME NO., and the name of the processor
(OTHER)	Not recognized by MCS	**WHAT ?
INP= n	Input tape will be set to logical unit n	**OK
LIB= n	Library tape will be set to logical unit n	**OK

REVISION SHEET FOR 9500236

Date	Description of Change	Revision	Approval
4/4/68	PRODUCTION RELEASE 68/248	"A"	A Fa