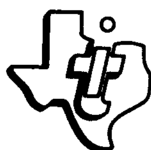
PROGRAMMING GUIDE

LEVEL ONE SOFTWARE FOR THE MODEL 980 COMPUTER

VOLUME I
LEVEL ONE SOFTWARE
MANUAL NO. 214851-9701
REVISED 30 JUNE 1970



TEXAS INSTRUMENTS
INCORPORATED
DIGITAL SYSTEMS DIVISION
P.O. BOX 66027 HOUSTON, TEXAS 77006

Copyright 1970

By

Texas Instruments Incorporated

All Rights Reserved

PRINTED IN U.S.A.

The information and/or drawings set forth in this document and all rights in and to inventions disclosed herein and patents which might be granted thereon disclosing or employing the materials, methods, techniques or apparatus described herein are the exclusive property of Texas Instruments Incorporated.

No disclosure of the information or drawings shall be made to any other person or organization without the prior consent of Texas Instruments Incorporated.

TABLE OF CONTENTS

Section		Page
I	GENERAL INFORMATION	
	1-1 Scope of Programming Guide	1-1
	1-2 Computer Characteristics	1-1
	1-3 Computer Control Panel	1-2
	1-4 Software	1-3
	1-4.1 Monitor	1-3
	1-4.2 Assemblers	1-4
	1-4.3 FORTRAN	1-4
	1-4.4 Link Edit	1-4
	1-4.5 Utilities	1-4
	1-4.6 Debug Package	1-4
	1-5 Standard Device Addresses	1-4
	1-5.1 Monitor Controlled Software	1-4
	1-5.2 Stand Alone Software	1-5
II	SYMBOLIC CODING IN ASSEMBLY LANGUAGE	
	2-1 Introduction	2-1
	2-2 Purpose of Assemblies	2-1
	2-3 The Assembly Process	2-1
	2-4 Symbolic Coding	2-3
	2-4.1 Location Counter	2-5
	2-4.2 Symbol Table	2-5
	2-4.3 Coding Summary	2-6
	2-5 Symbolic Line Format	2-6
	2-5.1 Comment Lines	2-6
	2-5.2 Name Field	2-6
	2-5.3 Operation Field	2-7
	2-5.4 Operand Field	2-7
	2-5.5 Comment Field	2-7
	2-6 Machine Instruction Formats	2-7
	2-6.1 Register-to-Memory (RM)	2-7
	2-6.2 Register-to-Register (RR)	2-10
	2-6.3 Shift (S)	2-10
	2-6.4 Register Skip (RS)	2-10
	2-6.5 Status Indicator Skip (SS)	2-11
	2-6.6 Sense Switch Skip (SS)	2-11
	2-6.7 Direct Memory Access Channel (DM)	2-11
	2-6.8 Data Bus Input/Output (DB)	2-11
	2-6.9 Fixed (F)	2-12
	2-7 Data Generation	2-12
	2-8 Assembler Directives	2-12
	2-8.1 BRS	2-12
	2-8.2 BRR	2-13
	2-8.3 BSS	2-13
	2-8.4 BES	2-14
	2-8.5 ORG	2-14
	2-8.6 DEF	2-14
	2-8.7 REF	2-14
	2-8.8 IDT	2-15
	2-8.9 HED	2-15
	2-8.10 PEJ	2-15
	2-8.11 LIS	2-15
	2-8.12 UNL	2-15
	2-8.13 EQU	2-15
	2-8.14 FRM	2-15
	2-8.15 IF	2-15
	2-8.16 END	2-15
	2-8.17 OPD	2-16
	2-8.18 COMM	2-16
	2-9 Object Formats	2-16
	2-9.1 Object Paper Tape Format	2-16
	2-9.2 Object Card Deck Format	2-17
	2-10 Listing Diagnostics for SAP-I	2-17
	2-11 SAP-I Operating Procedures	2-20
	2-12 Model 980 SAP-I On IBM System/360	2-20
	2-12.1 Error Diagnostics	2-21
	2-12.2 Table Sizes	2-21
	2-12.3 Loading and Operating	2-21
III	FORTRAN LANGUAGE PROGRAMMING	
	3-1 Introduction	3-1
	3-2 Program Preparation	3-1
	3-3 Arithmetic and Data	3-1
	3-3.1 Numerical Arithmetic	3-1
	3-3.2 Arithmetic Expressions	3-4
	3-3.3 Logical Arithmetic	3-6
	3-3.4 Summary of Operator Precedence	3-7
	3-3.5 Arithmetic Statements	3-7
	3-3.6 Use of Hollerith Data	3-8
	3-4 Control	3-8
	3-4.1 Introduction	3-8
	3-4.2 Unconditional GO TO Statement	3-8
	3-4.3 Computed GO TO Statement	3-8
	3-4.4 Assign and Assigned GO TO Statement	3-9
	3-4.5 Arithmetic IF Statement	3-9
	3-4.6 Logical IF Statement	3-10
	3-4.7 DO Statement	3-10
	3-4.8 CONTINUE Statement	3-13
	3-4.9 CALL Statement	3-13
	3-4.10 RETURN Statement	3-14

TABLE OF CONTENTS (Continued)

Section		Page
	3-4.11 PAUSE Statement	3-14
	3-4.12 STOP Statement	3-14
	3-4.13 END Statement	3-14
	3-5 Input/Output	3-14
	3-5.1 Transmission Statements	3-14
	3-5.2 Format Statement	3-17
	3-5.3 Auxiliary Input/Output Statements	3-22
	3-5.4 Logical Units	3-22
	3-6 Declarations	3-23
	3-6.1 Dimension Statements	3-23
	3-6.2 Type Declarations	3-23
	3-6.3 External Statement	3-24
	3-6.4 Common Statement	3-24
	3-6.5 Equivalent Statement	3-26
	3-6.6 Equivalence and Common	3-26
	3-6.7 Subprogram Definitions	3-27
	3-6.8 Subroutine Subprograms	3-28
	3-6.9 Data Specifications	3-29
	3-7 FORTRAN Output Listing	3-29
	3-7.1 Listing Elements	3-29
	3-7.2 Statement Error Diagnostics	3-30
	3-7.3 Program Error Diagnostics	3-30
	3-7.4 Pass Two Error Diagnostics	3-31
	3-8 Object Paper Tape Formats	3-31
	3-8.1 Group 1 Output Format	3-31
	3-8.2 Group 2 Output Format	3-31
	3-9 Library Subroutines	3-32
	3-9.1 Arithmetic Conventions	3-32
	3-9.2 Power Routines	3-32
	3-9.3 Mode Conversion Routines	3-32
	3-9.4 Complex Arithmetic	3-33
	3-9.5 Floating Point Arithmetic	3-33
	3-9.6 Double Precision Arithmetic	3-33
	3-9.7 FORTRAN Basic External Functions	3-34
	3-9.8 FORTRAN Intrinsic Functions	3-34
	3-9.9 FORTRAN Format Editor	3-34
	3-10 FORTRAN Operating Procedures	3-37
IV	REAL TIME MONITOR-I	
	4-1 Introduction	4-1
	4-1.1 Supervisor	4-1
	4-1.2 Service Subprograms	4-1
	4-1.3 Background Job Control	4-1
	4-1.4 The Disc Initialization Programs	4-1
	4-2 The Supervisor	4-1
	4-2.1 X-Level	4-1
	4-2.2 F-Level	4-4
	4-2.3 B-Level	6-6
	4-3 Service Subprograms	4-6
	4-3.1 I/O Service	4-6
	4-3.2 I/O Service Device Tables	4-6
	4-3.3 I/O Service Requests	4-8
	4-3.4 I/O Service General Flow and Linkage	4-8
	4-3.5 Control Function Service	4-8
	4-3.6 Non-System Illegal Instruction Traps	4-11
	4-4 Foreground Tasks	4-12
	4-5 Standard Service Requests	4-12
	4-5.1 Basic Control Functions	4-12
	4-5.2 Teletypewriter I/O Requests	4-13
	4-5.3 Disc I/O Requests	4-13
	4-5.4 High Speed Paper Tape Reader I/O Requests	4-14
	4-5.5 High Speed Paper Tape Punch I/O Requests	4-14
	4-5.6 Card Reader I/O Requests	4-15
	4-5.7 Line Printer I/O Requests	4-15
	4-5.8 Magnetic Tape I/O Requests	4-16
	4-6 SPEX	4-16
	4-7 BATCH	4-17
	4-7.1 Control Cards	4-17
	4-7.2 System Disc	4-17
	4-7.3 Batch Initialization	4-17
	4-8 System Generation	4-18
	4-9 Restore Disc	4-18
V	SPEX OPERATING PROCEDURES	
	5-1 General Procedures	5-1
	5-2 Symbolic Assembly Program	5-1
	5-2.1 Use of the Card Reader to Specify Options	5-1
	5-2.2 Use of the Teletypewriter to Specify Options	5-2
	5-2.3 Operating Procedure	5-2
	5-3 FORTRAN	5-2
	5-4 Link Edit	5-2
	5-4.1 Use of the Card Reader to Specify Options	5-2
	5-4.2 Use of Teletypewriter to Specify Options	5-3
	5-4.3 Operating Information	5-3

TABLE OF CONTENTS (Continued)

Section		Page
	5-4.4 Operating Procedure	5-3
	5-5 Correct Source	5-4
	5-5.1 Program Options	5-4
	5-5.2 Operating Information	5-4
	5-5.3 Operating Procedure	5-4
	5-6 List Source	5-5
	5-6.1 Use of the Card Reader to Specify Options	5-5
	5-6.2 Use of the Teletypewriter to Specify Options	5-5
	5-6.3 Operating Procedure	5-5
	5-7 User Program Execution	5-5
VI	BATCH OPERATING PROCEDURES	
	6-1 General Procedures	6-1
	6-2 SAP-I	6-1
	6-2.1 Control Card	6-1
	6-2.2 Operating Information	6-2
	6-2.3 Operating Procedure	6-2
	6-3 FORTRAN	6-2
	6-4 Link Edit	6-2
	6-4.1 Control Card	6-2
	6-4.2 Operating Information	6-3
	6-4.3 Operating Procedure	6-4
	6-5 Correct Source	6-4
	6-5.1 Control Card	6-4
	6-5.2 Operating Information	6-4
	6-5.3 Operating Procedure	6-5
	6-6 List Source	6-5
	6-6.1 Control Card	6-5
	6-6.2 Operating Information	6-6
	6-6.3 Operating Procedure	6-6
	6-7 Copy Source	6-6
	6-7.1 Control Card	6-6
	6-7.2 Operating Information	6-7
	6-7.3 Operating Procedure	6-7
	6-8 Copy Object	6-7
	6-8.1 Control Card	6-7
	6-8.2 Operating Information	6-7
	6-9 Install	6-7
	6-9.1 Control Card	6-7
	6-9.2 Operating Information	6-8
	6-9.3 Operating Procedure	6-8
	6-10 Delete	6-8
	6-10.1 Control Card	6-8
	6-10.2 Operating Information	6-8
	6-11 Compress	6-8
	6-11.1 Control Card	6-8
	6-11.2 Operating Information	6-8
	6-12 List Catalog	6-8
	6-12.1 Control Card	6-8
	6-12.2 Operating Information	6-8
	6-13 Save Disc	6-8
	6-14 System Control Cards	6-9
	6-14.1 JOB Card	6-9
	6-14.2 EOB Card	6-9
	6-14.3 EXEC Card	6-9
	6-15 Interface between BATCH and the Task Processor	6-9
	6-16 Sequence of Tasks	6-10
VII	BOOTSTRAPS AND LOADERS	
	7-1 Introduction	7-1
	7-2 Loading Procedures	7-1
	7-3 Bootstraps	7-2
	7-4 Reloading from Disc	7-2
	7-4.1 Loading from Disc	7-2
	7-4.2 Manually Loading and Executing Disc Bootstrap	7-2
	7-4.3 Restoring the Disc	7-3
	7-5 Disc Bootstrap	7-3
VIII	DEBUG PACKAGE	
	8-1 Introduction	8-1
	8-2 Program Options	8-1
	8-3 General Operating Procedures	8-1
	8-3.1 Loading a User Program	8-1
	8-3.2 Inspect and Change Core	8-2
	8-3.3 Store Masked Constant in Memory	8-2
	8-3.4 Search for Masked Constant	8-3
	8-3.5 Hexadecimal Dump	8-3
	8-3.6 Punch Object Tape	8-3
	8-3.7 Correction Load	8-4
	8-3.8 Move Debug Package	8-5
	8-3.9 Running a User Program	8-5

LIST OF ILLUSTRATIONS

Figure		Page
1-1	Model 980 Computer Block Diagram	1-2
1-2	Control Panel d 1-3	
2-1	Example of Symbolic Coding	2-2
2-2	The Assembly Process	2-3
2-3	Assembly Listing Produced by Assembly of the Program	2-4
2-4	Line Terminating Codes	2-6
4-1	RTM Basic Structure	4-2
4-2	Supervisor Priority Levels	4-3
4-3	I/O Service Request General Flow	4-9
4-4	Supervisor Flow Relative to I/O Request	4-10
4-5	Device Table Set Up	4-19
4-6	Work List Set Up	4-19
7-1	Bootstrap and Loader	7-1

LIST OF TABLES

Table		Page
2-1	IXB Usage	2-8
2-2	RM Format Symbolic Interpretation	2-9
2-3	Example for Data	2-13
2-4	Assembler Directives	2-13
2-5	Object Format	2-18
2-6	Object Card Deck Format	2-19
2-7	To Go from Object Deck	2-21
2-8	To Catalog the Object Deck Into the IBM System 1360	2-21
2-9	To Go from a Catalogued Version	2-21
3-1	Group 2 Output Format	3-31
3-2	FORTRAN Basic External Functions	3-35
3-3	FORTRAN Intrinsic Functions	3-36
3-4	FORTRAN Format Editor	3-37
4-1	X-Level Return	4-4
4-2	Worker Task List	4-5

INDEX

Subject	Page
Addressing Modes	1-1
Alphanumeric Fields	3-20
Arithmetic and Data	3-1
Arithmetic, Complex	3-35
Arithmetic Function Definition Statement	3-27
Arithmetic Statements	3-7
Arrays and Subscripts	3-4
Assemblers	1-10
Assembly Process	2-1
Assign and Assigned GO TO Statements	3-9
Auxiliary Input/Output Statements	3-22
B-Level	4-6
Background Job Control	4-1
BACKSPACE Statement	3-22
Basic Control Functions	4-12
BATCH	4-17
BATCH Operating Procedures	6-1
BES Directive	2-14
Bootstraps	7-1
BRR Directive	2-13
BRS Directive	2-12
BSS Directive	2-13
Card Reader Characteristics	1-5
Coding in Assembly Language, Symbolic	2-1
Coding Summary	2-6
COMM Directive	2-16
Comment Field	2-7
Comment Lines	2-6
Complex Arithmetic	3-35
Complex Constants	3-3
Compress	6-8
Computed GO TO Statement	3-8
Computer Characteristics	1-1
Common Statement	3-24
Constants	3-2
CONTINUE Statement	3-13
Control Cards	4-17
Control Features	1-2
Control (FORTRAN)	3-8
Control Panel	1-2
Correction Load	8-4
Data Bus Input/Output (DB) Instruction Format	2-11
Data Generation	2-12
Data Specifications	3-29
D-Conversion	3-20
Debug Package	1-10,8-1
Debug Program Caution	8-4
Declarations	3-23
DEF Directive	2-14
Delete	6-8
Device Tables, I/O Service	4-6
Direct Memory Access Channel (DM) Instruction Format	2-11

Subject	Page
Disc Initialization	4-1
Disc I/O Requests	4-13
Dimension Statement	3-23
Double Precision Arithmetic	3-35
Double Precision Constants	3-2
Double Precision Statement	3-24
Dummy Identifiers	3-27
E-Conversion	3-19
END Directive	2-15
END FILE Statement	3-22
EQU Directive	2-15
Equivalence Statement	3-26
Error Diagnostics (TI980SIM)	2-20
Execution Times for Direct Operands	1-1
Explicit Specification	3-4
Evaluation of Arithmetic Expressions	3-5
F-Conversion	3-18
Fixed Head Disc Drive Characteristics	1-9
Fixed (F) Instruction Format	2-12
Foreground Tasks	4-12
Format Fields, Alphanumeric	3-20
Formation of Logical Expressions	3-6
Formats Stored as Data	3-21
Formatted Read Statement	3-15
Formatted Write Statement	3-16
FORTRAN	1-10
FORTRAN Basic External Functions	3-36
FORTRAN Format Editor	3-36
FORTRAN Intrinsic Functions	3-36
FORTRAN Output Listing	3-29
FRM Directive	2-15
Function Reference	3-4
Function Statement	3-28
Function Subprograms	3-28
G-Conversion	3-19
General Information	1-1
General Flow and Linkage, I/O Service	4-8
GO TO Statements	3-8
Group 1 Output Format	3-31
Group 2 Output Format	3-34
HED Directive	2-15
Hexadecimal Dump	8-3
High Speed Punch I/O Requests	4-14
High Speed Punch Tape Reader Characteristics	1-4
High Speed Reader I/O Requests	4-14
High Speed Tape Punch Characteristics	1-5
Hollerith Data, Use of	3-8
I-Conversions	3-18
IDT Directive	2-15
IF Directive	2-15
IF Statement	3-9
Implicit Specification	3-3

Subject	Page
Input/Output	3-14
Input/Output Characteristics	1-1
Input/Output Lists	3-15
Interface Between BATCH and the Task Processor	6-9
Integer Constants	3-2
Integer Statement	3-24
I/O Service	4-6
IXB Usage	2-8
Library Subroutines	3-33
Line Printer Characteristics	1-6
Line Printer I/O Requests	4-15
Link Edit	1-10
LIS Directive	2-15
List Catalog	6-8
Listing Diagnostics for SAP-I	2-17
Listing Elements	3-29
Loaders	7-1
Loading and Operation (TI980SIM)	2-21
Loading a User Program	8-1
Loading Procedures	7-1
Location Counter	2-5
Logical Arithmetic	3-6
Logical Fields	3-20
Logical IF Statement	3-9
Machine Instruction Formats	2-7
Magnetic Tape I/O Requests	4-16
Magnetic Tape Transport Characteristics	1-9
Mixed Fields	3-20
Move Debug Package	8-5
Mode Conversion Routines	3-33
Model 980 SAP-I on IBM System/360	2-20
Model 980 Software	1-9
Monitor	1-9
Monitor Controlled Software	1-10
Monitor, Real Time	4-1
Multiple Record Formats	3-21
Name Field	2-6
Non-System Illegal Instruction Traps	4-11
Numerical Arithmetic	3-1
Numerical Fields	3-17
Object Card Deck Output	2-17
Object Formats	2-16
Object Paper Tape Format	2-16,3-31
OPD Directive	2-16
Operand Field	2-7
Operating Procedures (SAP-I)	2-20
Operation Field	2-7
Options for Batch	6-1
Options for SPEX	5-1
ORG Directive	2-14

Subject	Page
Output Format, Group 1	3-31
Output Format, Group 2	3-34
PEJ Directive	2-15
Power Routines	3-33
Program Error Diagnostics	3-30
Program Options	8-1
Program Preparation	3-1
Punch Object Tape	8-3
Program Caution (Debug)	8-4
Purpose of Assemblers	2-1
Real Constants	3-2
Real Statement	3-24
Real Time Monitor	4-1
REF Directive	2-14
Register Skip (RS) Instruction Format	2-10
Register-to-Memory (RM) Instruction Format	2-7
Register-to-Register (RR) Instruction Format	2-10
Reloading from Disc	7-2
Repetition of Field Specifications	3-20
Request, I/O Service	4-8
Restore Disc	4-18
RETURN Statement	3-14
REWIND Statement	3-22
RM Format Symbolic Interpretation	2-9
RTM-I Basic Structure	4-2
Running a User Program	8-5
Save Disc	6-8
Scale Factors	3-18
Scope of User's Guide	1-1
Search for Masked Constant	8-2
Sense Switch Skip (SX) Instruction Format	2-11
Sequence of Tasks	6-10
Service Subprograms	4-1,4-6
Shift (S) Instruction Format	2-10
SPEX	4-16,5-1
Stand Alone Software	1-11
Standard Device Addresses	1-10
Standard Service Requests	4-12
Statement Error Diagnostics	3-30
Statements, Format	3-17
Status Indicator Skip (SS) Instruction Format	2-11
Store Masked Constant in Memory	8-2
Subroutine Statement	3-28
Subroutine Subprograms	3-28
Subscripted Variables	3-4
Summary of Operator Precedence	3-7
Supervisor	4-1
Supervisor Flow Relative to I/O Request	4-10
Supervisor Priority Levels	4-3
Symbol Table	2-5

INDEX (Continued)

Subject	Page
Symbolic Coding	2-3
Symbolic Coding Summary	2-6
Symbolic Line Format	2-6
System Disc	4-17
System Generation	4-18
Table Sizes (TI980SIM)	2-21
Teletypewriter Characteristics	1-3
Teletypewriter I/O Requests	4-13
Unconditional GO TO Statement	3-8
UNL Directive	2-15
Use of Hollerith Data	3-8
User Program, Loading	8-1
Utilities	1-10
Variables, Logical	3-6
Variables (FORTRAN)	3-3
Worker Task List	4-5
X-Level	4-1

SECTION I

GENERAL INFORMATION

SECTION I

GENERAL INFORMATION

1-1 SCOPE OF PROGRAMMING GUIDE.

The Texas Instruments Model 980 Computer User's Guide is divided into three volumes:

a. Volume I contains the level one Real Time Monitor and related software description.

b. Volume II contains the performance assurance test descriptions.

c. Volume III contains the library routine descriptions.

Revisions and additions will be published whenever they are justified. Users who are on the authorized distribution list will receive published revisions and additions automatically.

The Programming Guide is broken into several sections according to subject. Refer to the Table of Contents to quickly locate a desired subject.

The user is given a general orientation to the Model 980 Computer in this section. Details of programming may be located also in the Model 980 Computer Programmer's Reference Manual. Detail information about the computer itself may be found in the Model 980 Computer Maintenance Manual.

1-2 COMPUTER CHARACTERISTICS.

The Texas Instruments Model 980 General Purpose Computer is used for data processing and system control applications. The computer is functionally organized into a central processing unit (CPU), an input/output (I/O) unit, and a power supply (Figure 1-1)

a. General characteristics

Parallel operation

Single address logic

Two's complement arithmetic

Eight 16-bit addressable registers

16-bit data word plus parity

16-bit or 32-bit instruction word

85 basic instructions

b. Execution times for direct operands

ADD	2.00 microseconds
SUBTRACT	2.00 microseconds
MULTIPLY	6.50 microseconds
DIVIDE	8.00 microseconds

c. Memory

1-microsecond cycle time

4096 words minimum capacity

65,536 words maximum capacity

400 nanoseconds access time (internal data access)

1.0-microsecond direct memory access time (I/O data access)

All of memory can be directly addressed.

Power failure protection

d. Input/Output

One Direct Memory Access Channel (DMAC), expandable to eight

17-bit parallel transfer including parity comparison

1-million words per second burst rate

One processor-controlled I/O bus, 16-bit parallel transfer

One (or more) teletypewriter or electronic printer I/O channels

Three priority interrupts

e. Addressing Modes

Direct addressing

Program Counter relative addressing

Base Register relative addressing

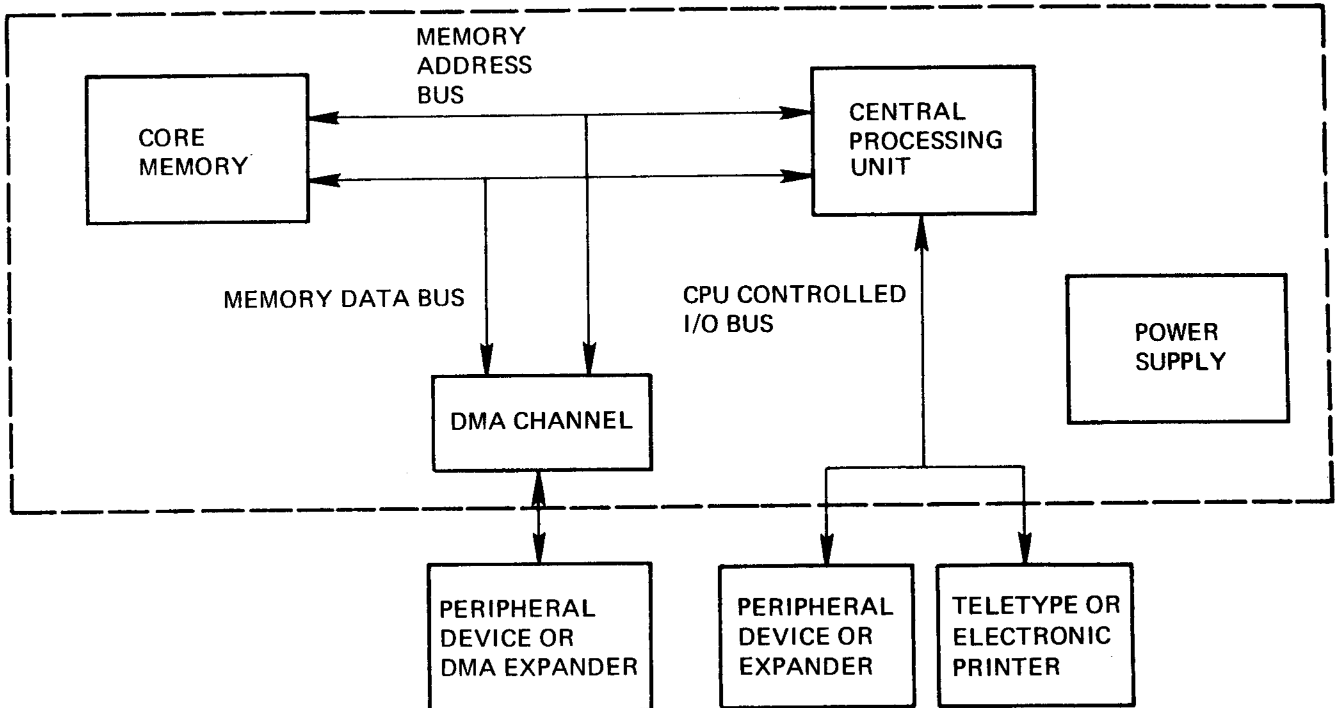


Figure 1-1 Model 980 Computer Block Diagram

Single-level indirect addressing

Direct indexing

Pre-indexing and post-indexing with indirect addressing

Immediate operands

f. Peripherals

Nine track magnetic tape

Silent 700 electronic data terminal

ASR-33 teletypewriter

Fixed head disc

High speed paper tape reader

High speed paper tape punch

Line printer

Card reader

Card punch

1-3 COMPUTER CONTROL PANEL.

The control panel is shown in Figure 1-2. Indicators and pushbuttons on the control panel allow the operator and the computer to establish basic communications. It facilitates the loading of programs, system checkout, maintenance, and software debugging. The address and contents of the memory instruction registers are constantly displayed. Any of the following data words may be observed on the DISPLAY lights by pressing the corresponding DISPLAY SELECT pushbuttons.

PC – Program Counter
MR – Memory Data
A – Primary Arithmetic Register
B – Base Register
E – Secondary (Extension) Arithmetic Register
L – Link Register
M – Maintenance Register
S – Storage Register
X – Index Register
SR – Status Register

The 16 DATA pushbuttons may be used to enter data into the register indicated by the DISPLAY SELECT using the LOAD pushbutton. Each data bit may be reset by pressing it a second time. All data bits may be reset by pressing the CLEAR pushbutton.

When loading or reading memory data, the memory word referenced will be the word at the location contained in the program counter. The data is entered into memory by loading into the MR.

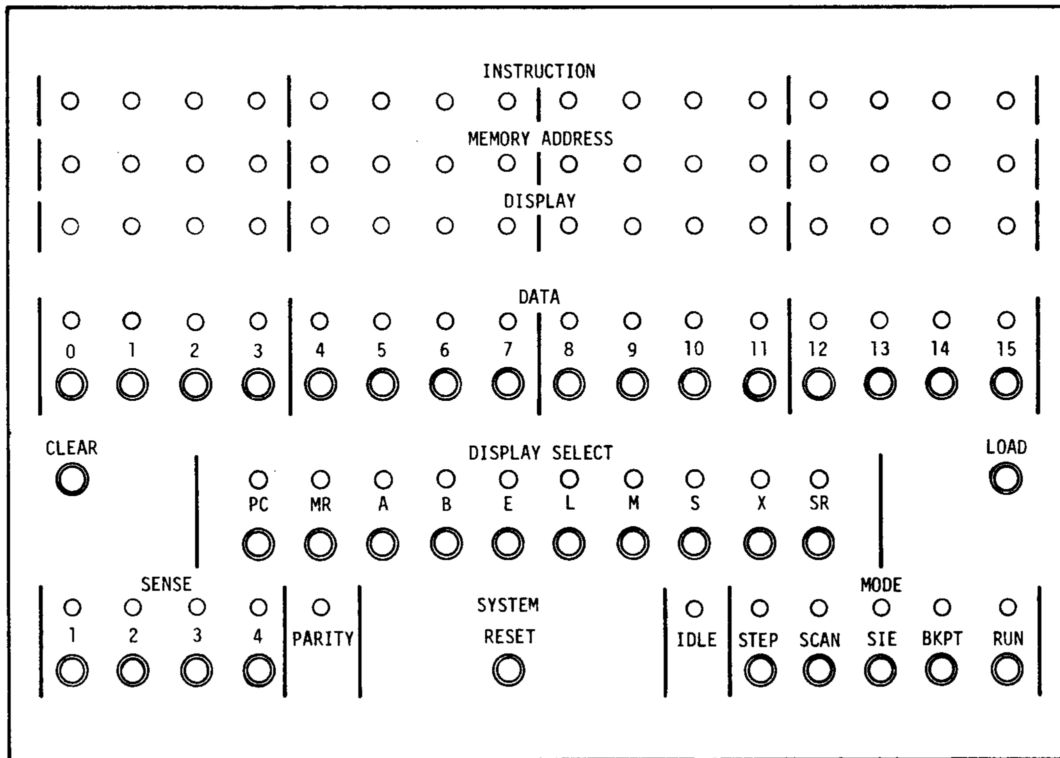


Figure 1-2. Control Panel

Four SENSE pushbuttons are provided and displayed at all times. These may be sensed by the program.

A PARITY ERROR indicator is illuminated when the parity error stop bit has been enabled in the status register and a memory parity error is detected.

The SYSTEM RESET pushbutton clears the status register, sense switches, and program counter.

The IDLE light is illuminated when an IDLE instruction is encountered in the program. It indicates that the CPU is in a waiting state.

Electronically interlocked MODE pushbuttons are provided and displayed which allow the operator to perform the following functions:

STEP – The first time the STEP pushbutton is pushed it inhibits the system clock. Each time thereafter it provides a single clock pulse to the system.

SIE – The first push halts the CPU. Each subsequent push frees the CPU until the next instruction has been executed.

SCAN – The first push halts the CPU. Each subsequent push increments the Program Counter and reads the memory at that address. Memory at that address may be altered by using the DATA and LOAD pushbuttons.

BKPT – Pressing the BKPT pushbutton causes the CPU to retain the address given in the console data switches. In and RUN mode the CPU will halt when that memory address is referenced as an instruction or data.

RUN – Frees the CPU to operate.

1-4 SOFTWARE.

1-4.1 MONITOR. The Real Time Monitor (RTM-I) is a multi-programming operating system utilizing an executive/worker method for program control and a multilevel priority scheme for program execution. It handles all software input and output, and schedules worker programs based upon real time input. RTM-I options include either sequential background job execution (BATCH) or background execution upon request (Single Program EXecution – SPEX).

1-4.2 **ASSEMBLERS.** Two distinct assemblers are provided for the Model 980. The SAP-I (Symbolic Assembly Program-I) assembler executes on the Model 980 Computer. The second assembler (TI980SIM) is an IBM System/360 program.

Several versions of SAP-I are available. These differ primarily in the particular configuration of computer peripherals which they assume, the mode of operation (BATCH, SPEX, or stand alone), and the number of physical passes through the input source statements which are required.

TI980SIM accepts the same source input as SAP-I. Likewise, the object output is fully compatible with the Model 980 link editor and relocating loader.

1-4.3 **FORTRAN.** A FORTRAN compiler is available with the Model 980 Computer. This compiler accepts ANSI standard FORTRAN X-3.9-1966 (commonly called FORTRAN IV). It additionally allows certain extensions, for example, general integer expressions for indexes.

The library supplied includes those routines required to support ANSI standard FORTRAN.

1-4.4 **LINK EDIT.** Several separately assembled or compiled programs can be merged into one program. All external references from one routine to symbolic locations in another routine and COMMON references are resolved through use of the Link Editor. An object program is produced in the format described in Section II. This object may be punched or left on some other media such as disc.

1-4.5 **UTILITIES.** Various utility programs are included in the Model 980 software. These are useful in manipulating source and object programs. The utility package includes several programs intended only for BATCH processor execution. These BATCH programs facilitate editing of disc storage.

The utility functions performed are:

Correct – insert corrections in a source program

Copy – copy a program from one media to another

List – produce a listing of a source program

Install – catalog a program on the disc for BATCH processing

Delete – flag a catalogued program as deleted

Compress – remove all deleted programs from the BATCH disc file

List Catalog – produce a listing of the contents of the BATCH disc file

Save Disc – copy the BATCH disc files on another media

1-4.6 **DEBUG PACKAGE.** The debug package is a compact stand-alone program intended to aid programmers in debugging their programs. It performs the following functions:

a. Inspect consecutive words of memory and change them if desired

b. Store a masked constant in memory

c. Search memory for a masked constant

d. Execute a hexadecimal dump of a specified area of memory

e. Punch an object tape of a specified area of memory

f. Load changes in memory as specified on a punched tape or deck of cards

g. Move the debug package to another area of memory.

All necessary parameters are manually entered into registers through the computer console.

1-5 STANDARD DEVICE ADDRESSES.

1-5.1 **MONITOR CONTROLLED SOFTWARE.** Software running under control of RTM-I and using the RTM-I supplied I/O programs will address their I/O requests to logical rather than physical devices. The actual correspondence between logical devices and physical device addresses is established as part of system generation. The standard logical device numbers are as follows:

1 – Teletypewriter output (punch or type)
2 – Teletypewriter input (reader or keyboard)
3 – High speed paper tape punch
4 – High speed paper tape reader
5 – Card reader
6 – Line printer
7 – Magnetic tapes
8 – Disc

Any deviations from these assumed logical device numbers requires re-assembly of the programs involved.

1-5.2 **STAND ALONE SOFTWARE.** Stand alone programs performing their own I/O functions directly address the physical devices. Software in this category includes:

- Stand Alone Assemblers
- Stand Alone FORTRAN Compilers
- Debug Package
- Bootstraps and Loaders
- Restore Disc Program
- Monitor Initialization
- Performance Assurance Tests (described in Volume II)

These programs assume certain standard device addresses. If the computer system uses non-standard addresses the programs involved must be reassembled to reflect the new device addresses. The assumed data bus addresses (hexadecimal) are:

Device	Control Register	Data Register
teletypewriter	0A	02
card reader	1F	1F
high speed paper tape reader	10	18
high speed paper tape punch	10	18

The assumed direct memory access channel device numbers are:

Device	Number
disc	0
magnetic tapes	1
line printer	5

SECTION II

SYMBOLIC CODING IN ASSEMBLY LANGUAGE

SECTION II

SYMBOLIC CODING IN ASSEMBLY LANGUAGE

2-1 INTRODUCTION.

The first portion of this section discusses symbolic coding in detail. If the user is already familiar with symbolic coding techniques, he may proceed directly to paragraph 2-4.3 for a summary of the first portion. Explanation of the Model 980 Computer assembly language begins with paragraph 2-5.

The programmers job is to:

a. Arrange input and output of data.

b. Establish "work areas" in storage.

c. Create constants or text values used in calculations and printed output.

d. Choose and write the instructions that move data, perform appropriate tests and calculations, handle exceptional conditions, and arrange data in a format specified for output.

Assembly language with symbolic notation is one method by which this work is done.

2-2 PURPOSE OF ASSEMBLERS.

Programming in a symbolic language offers important advantages over programming in the actual language of the computer.

a. Mnemonic operation codes are more meaningful than machine language. For instance, the actual machine language for the instruction Store Register A (in hexadecimal) is 80. The mnemonic operation code in the assembly language is STA.

b. Addresses of data and instructions can be written in symbolic form. The programmer is thereby relieved of problems in the effective allocation of storage, and the resulting program is far easier to modify. Furthermore, the use of symbolic addresses reduces the clerical aspects of programming and eliminates many programming errors. If the symbols chosen are meaningful, the program is also much easier to read and understand than if written with numerical addresses.

c. Symbolic assembly directives and data generation statements permit the introduction of constants, reservation of space for results, definition of instructions, control of the assembly process, introduction of base address values, and other items.

The sum effect of these advantages is so great that it is virtually out of the question to program in actual machine language; that is, to write actual operation codes and numerical address displacements.

2-3 THE ASSEMBLY PROCESS.

An assembly language program cannot be executed directly by the computer. The mnemonic operation codes and symbolic addresses must be translated into machine language. This is the function of the Symbolic Assembly Program (SAP-I).

The assembly process begins with a source program which is written by the programmer. Ordinarily, a special coding form is used (Figure 2-1). Cards or paper tape are punched to correspond with the data on this form. This source program becomes the primary input to the assembly process. The Symbolic Assembly Program (SAP-I) controls the assembly process in the Model 980 Computer (Figure 2-2).

SAP-I generates two outputs. The first is an object program. Actual machine instructions in the object program correspond to the source program statements written by the programmer. The object program can be output on punched cards, paper tape, or disc. The second output is an assembly listing. This important document shows the original source program statements side by side with the object program instructions created from them. Refer to Figure 2-3.

Note the following in the example:

a. The items listed under A should be exactly the same as the handwritten entries on the coding sheet. This provides a good check on the accuracy of the keypunching.

b. The items under B are a hexadecimal representation of the corresponding instructions and constants.

c. The items under C show the hexadecimal addresses of the instructions, constants, and areas of storage specified by the programmer.

PROBLEM ______________________

____

PROGRAMMER ______________________

SYMBOLIC CODING FORM

73 Identification 80

NAME	OPERATION	OPERAND	COMMENTS
	HED	MØDEL 980 ILLUSTRATIVE PRØGRAM	
	ØRG	8	
	BRS	8	
START	LDA	DATA	PUT DATA IN THE A
	ADD	NUMBER	ADD NUMBER TØ A
	STA	RESULT	SAVE RESULT
..THIS	LINE	IS A CØMMENT	
	LDA	LØGIC	GET LØGIC PATTERN
	IØR	RESULT	ØR IN RESULT
	BRL	DØ	PERFØRM SUBRØUTINE
	IDL		PAUSE
DØ	SEV	A	SKIP IF LSB=0
	LLA	2	LEFT
	RMØ	L,P	EXIT FROM DØ
DATA	DATA	506	DECIMAL VALUE
NUMBER	DATA	1523	
RESULT	BSS	1	RESERVE 1 LØCATIØN
LOGIC	DATA	>FA62	HEX CONSTANT
A	EQU	0	
L	EQU	5	
P	EQU	7	
	END	START	

Figure 2-1. Example of Symbolic Coding

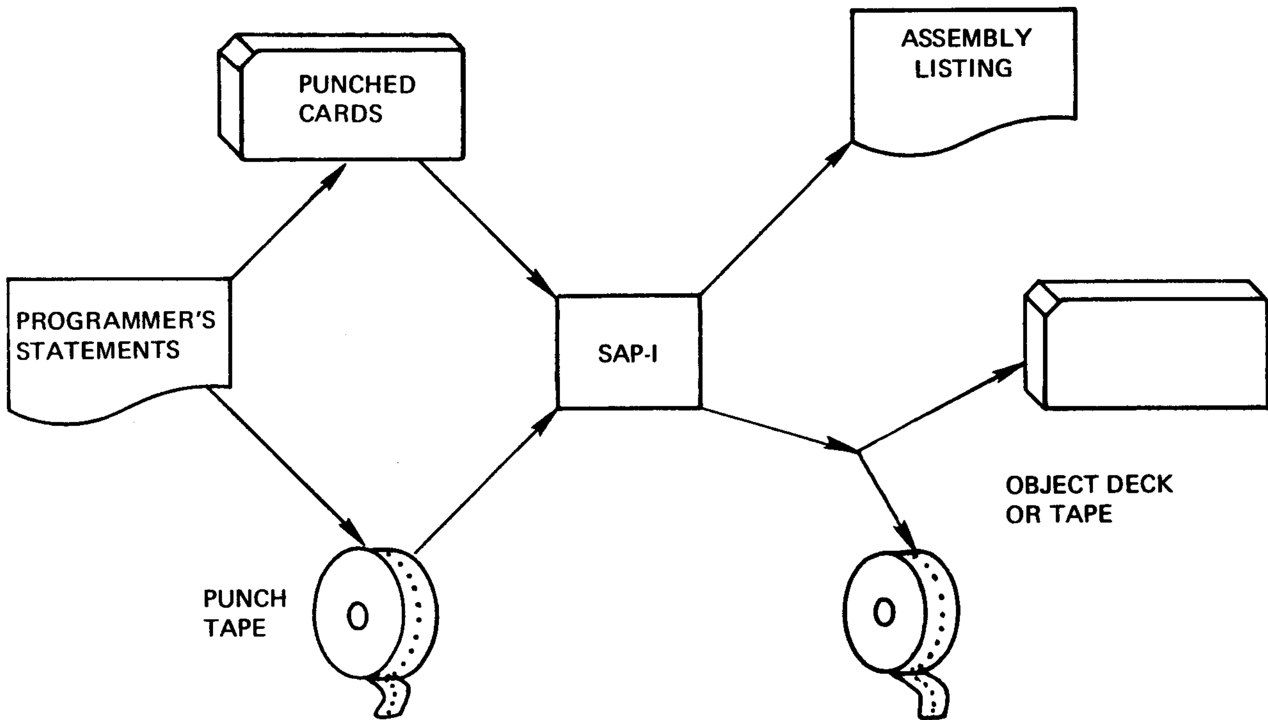


Figure 2-2. The Assembly Process

d. The items under D show the decimal line or sequence number of the source statements. It is printed to assist the programmer when correcting his source program. It has no effect on the object program.

2-4 SYMBOLIC CODING.

Each line of the coding sheet represents one symbolic statement. Each symbolic statement is used to tell SAP-I to assemble a machine language instruction, a data constant, or to do something during assembly time. Since most programs contain hundreds of instructions, several coding sheets are used to write a complete program.

All instructions have a location in memory which they will occupy when the object program is being executed. Instructions also have an operation code and usually one or more operands. For instance, take the case of an instruction which adds the content of one register to that of another. This instruction would have a hexadecimal operation code of C08, and contain the addresses of two registers. The address of an instruction, its operation code, and the data addresses correspond respectively to the following fields on the coding sheet: Name, Operation, Operand. The entries on the coding sheet are made symbolically, rather than in machine language.

The first six columns of the coding sheet are called the name field. This field gives symbolic names to the locations referred to by the program. For instance, if the program contains a routine to handle division overflows, it would be simpler if the machine address of the routine did not have to be remembered. After assigning a name to the first instruction in this routine, a symbolic branch instruction referring to that name can be written. Then SAP-I, in converting the program to machine language, will remember the machine address of the symbolic name and will use it in the object program whenever the programmer refers to it.

Symbolic names are also called symbols or labels. Some of the characteristics of symbolic names are:

a. Symbolic names are usually given to instructions or data fields referred to in programs.

b. A symbol cannot be used in the operand field unless it also appears in the name field of one of the symbolic statements. That is, a symbol in the program cannot be referenced unless it is used as the name of one of the instructions, directives, or data fields.

```
MODEL 980 ILLUSTRATIVE PROGRAM                                    Ø1/Ø9/7Ø  ØØØ1

  C         B         D                          A

                    ØØØ1             HED     MODEL 98Ø ILLUSTRATIVE PROGRAM
ØØØ8                ØØØ2             ORG     8
          ØØØ8      ØØØ3             BRS     8
ØØØ8      ØØØ9      ØØØ4    START    LDA     DATA       PUT DATA IN THE A
ØØØ9      2ØØ9      ØØØ5             ADD     NUMBER     ADD NUMBER TO A
ØØØA      8ØØ9      ØØØ6             STA     RESULT     SAVE RESULT
                    ØØØ7    ...THIS LINE IS A COMMENT
ØØØB      ØØØ9      ØØØ8             LDA     LOGIC      GET LOGIC PATTERN
ØØØC      3ØØ7      ØØØ9             IOR     RESULT     OR IN RESULT
ØØØD      7ØØ1      ØØ1Ø             BRL     DO         PERFORM SUBROUTINE
ØØØE      CEØØ      ØØ11             IDL                PAUSE
ØØØF      CCCØ      ØØ12    DO       SEV     A          SKIP IF LSB=Ø
ØØ1Ø      C8C2      ØØ13             LLA     2          LEFT SHIFT 2
ØØ11      C557      ØØ14             RMO     L,P        EXIT FROM DO
ØØ12      Ø1FA      ØØ15    DATA     DATA    5Ø6        DECIMAL VALUE
ØØ13      Ø5F3      ØØ16    NUMBER   DATA    1523
ØØ14                ØØ17    RESULT   BSS     1          RESERVE 1 LOCATION
ØØ15      FA62      ØØ18    LOGIC    DATA    >FA62      HEX CONSTANT
          ØØØØ      ØØ19    A        EQU     Ø
          ØØØ5      ØØ2Ø    L        EQU     5
          ØØØ7      ØØ21    P        EQU     7
          ØØØØ      ØØ22             END     START
```

Figure 2-3. Assembly Listing Produced By Assembly Of The Program

c. Symbols are restricted in length to six characters or less and may not contain blanks or special characters. They must begin with a letter.

d. Within the preceding limitations, any symbol may be used.

Each line on a coding sheet is one symbolic statement. A symbolic statement can be a machine instruction, a data definition, or an assembly directive which gives some information to SAP-I for use during the assembly process. The operation field on the coding sheet tells the processor the type of the statement. Although the name field of a symbolic statement may be left blank, the operation field must contain a symbol that can be recognized by SAP-I. If the symbolic statement is an instruction, the symbol represents one of the computer's operation codes.

For example, by writing LDX, the processor is caused to assemble a Load Register X instruction. The mnemonic for the instruction is placed in the operation field of the coding sheet. Each Model 980 Computer instruction has its own unique mnemonic which is given in the Programmer's Reference Manual.

The operand field on the coding sheet contains the remainder of the instruction. The data required in the operand field is dependent upon the operation specified. For each SAP-I operation the required operands are described in later paragraphs of this section.

The operand may be followed by a comment. This has no effect on the assembly.

The program coded using the assembly language symbology is called the source program.

The sole function of the source program is to provide input data for the assembly program. None of the instructions in the source program is executed during the assembly (translation) process. One output of the assembler will be the object deck or tape. The object deck or tape is the program converted into machine language. It can be loaded, either now or later, into the computer for execution. There is no need to reassemble the program each time it is executed. The object deck or tape can be used over and over again until changes are made in the program.

To obtain a machine language object program from the symbolic source program, SAP-I must first be loaded into the computer's main storage. As the assembler is being executed, it will read the source program and convert it to the machine language that will be the object program.

2-4.1 LOCATION COUNTER. The computer, while executing the assembly program, acts as a clerk. One of the clerical tasks of the assembler is to assign machine addresses to symbolic names, and to remember these addresses and use them in the object program whenever the symbol is used in the operand of the source statement.

For instance, when the assembler encounters the following source statement, it must assign a machine address to the symbol BEGIN.

NAME	OPERATION	OPERAND
BEGIN	STE	THERE

The assembler must remember the address of BEGIN so that it can insert that address when it encounters the following branch instruction.

NAME	OPERATION	OPERAND
	@BRU	BEGIN

To be able to assign a machine address to a symbol, SAP-I contains an internal counter. This counter is called the location counter, and keeps track of the addresses in the source program, as it is being assembled. The location counter is incremented as each symbolic statement is processed. The length (in words) of main storage area required by each statement determines how much the location counter is incremented. For instance, assume that the location counter is set to decimal 1000 when the following symbolic statement is read by the assembler.

NAME	OPERATION	OPERAND
BEGIN	STE	THERE

When the assembler encounters this statement it will assign the address of decimal 1000 to the symbol BEGIN, and step the location counter to decimal 1001.

Whenever the assembler finds an entry in the name field, it assigns the setting of the location counter to that name. It then increments the counter by the number of words required by the statement. The STE instruction in the example requires only one word. Hence, the location counter is stepped from 1000 to 1001.

2-4.2 SYMBOL TABLE. The assembler uses the location counter to assign addresses to symbols. However, the assembler needs to retain the address it assigns to each symbol. These are stored in another data area within the assembly program. This area is referred to as the symbol table. When a symbol is encountered in the name field of a symbolic statement, that symbol as well as the location counter setting is placed in the symbol table. The area of storage used for the symbol table is limited. It is for this reason that SAP-I puts a limit on the length of symbols and how many symbols may be used in a program.

Whenever the assembler finds a symbol in the operand field, it locates the symbol in the symbol table. When it locates the symbol, it obtains its machine address and uses it in computing the assembled instruction. Of course, the symbol must appear somewhere in the source program, in the name field.

SAP-I is a two-pass assembler. The first pass of a two-pass assembler does not produce an object program. Its purpose is to build up a complete symbol table.

If bulk storage is available, some versions of SAP-I have an intermediate output. The intermediate output from the first pass is used as input data for the second pass. This eliminates the requirement to manually enter the source data twice.

During the second pass, the assembler program uses the symbol table to complete the assembly of the statements. The output of the second pass is the object tape (or card deck) and assembly listing.

2-4.3 CODING SUMMARY.

a. During assembly time, the assembly program is executed using a source program as input data.

b. The output data from the assembler consists of an object tape (or card deck) and its assembly listing.

c. A location counter in the assembler is used to keep track of the storage locations that will be used by the object program.

d. When a source statement contains a name, the current setting of the location counter is given to the label.

e. Each label and the address assigned to it is placed in the assembler's symbol table.

f. SAP-I is a two-pass assembler.

g. During the first pass, the source program is read and the symbol table is generated.

h. During the second pass, the symbol table is used to complete the assembly, and produce the object tape (or card deck) with its assembly listing.

2-5 SYMBOLIC LINE FORMAT.

The symbolic input line accepted by the assembler may contain a name field, operation field, operand field, and a comment field – or the entire line may be a comment. An input line is the first 72 characters read from a card or in the case of paper tape, an input line is a string of characters the last of which are carriage return, reader off, rub out and line feed (Figure 2-4). The input line must not exceed 72 characters not including the terminal characters. Input lines are free form within the limits described below.

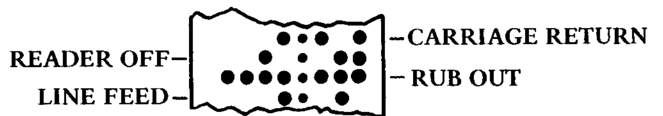


Figure 2-4. Line Terminating Codes

2-5.1 COMMENT LINES. Comment lines provide the user with the ability to annotate program listings. They are indicated by an initial character which is either a period (.) or an asterisk (*). The remaining characters are arbitrary. The comment line in no way affects the assembly process. The line is merely reproduced in the printed output.

2-5.2 NAME FIELD. Names (also called symbols or labels) are provided for symbolic references to instructions, values, and data. A label is composed of from one to six characters, none of which may be any of the special characters listed below. The first character of a label should be alphabetic (A-Z).

Plus	+
Minus	-
Asterisk	*
Slash	/
Greater than	>
Right Parenthesis	)
Left Parenthesis	(
Comma	,

If a label is used, the first character must begin the input line. The label is terminated by the first space.

2-5.3 **OPERATION FIELD.** The operation field describes the required action. It may be a mnemonic operation code, assembler directive, or data generation directive. The field consists of from one to four characters followed by a space or carriage return. The first character of the operation field must be preceded by at least one space.

2-5.4 **OPERAND FIELD.** The operand field consists of a sequence of expressions separated by commas and is terminated by a space or carriage return.

EXPRS1,EXPRS2,EXPRS3

If two commas appear successively, the value of the missing expression will be taken to be zero. If fewer than the required number of expressions appear in a source line, the missing expressions will be assumed to be zero. If the currency symbol ($) appears as an element in an expression, the current value of the assembler's location counter will be used as its numeric equivalent.

Expressions are strings of items separated by arithmetic operators and terminated by a space, comma, or carriage return. The arithmetic operators are:

Addition	+
Subtraction	-
Multiplication	*
Division	/

If two operators appear in succession, a zero item is assumed.

An item consists of a symbolic address, currency symbol ($), or a numeric value. If the first character of an item is not numeric, $, or >, it is assumed to be symbolic. Numeric items may be octal, decimal, or hexadecimal. An octal item is a string of octal characters (0-7), the first of which is zero. A decimal item is a string of numeric characters (0-9), the first of which is non-zero. A hexadecimal item is a greater than symbol (>) followed by a string of hexadecimal digits (0-9, A-F). When using paper tape input, the back slash (\) may be used in place of > to indicate hexadecimal.

Expressions are evaluated left to right using normal arithmetic precedence; i.e., all multiplications and divisions are performed in order of occurrence. The additions and subtractions are then performed in order of occurrence.

All quantities are treated as integers. In division only the quotient is retained and any remainder is discarded. Division by zero is performed as division by one and is not considered as an error.

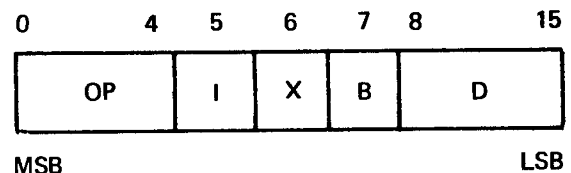
Sample expressions are:

JOE+TOM*3/BOB
$+5
LEA-6
5034
XYZ+>F4

2-5.5 **COMMENT FIELD.** Comments may optionally be written on any line. Any characters which appear between the space which terminates the operand field and the carriage return or card column 73 are treated as commentary. The comment field has no effect on the assembly process.

2-6 MACHINE INSTRUCTION FORMATS.

2-6.1 **REGISTER-TO-MEMORY (RM) INSTRUCTION FORMAT.** The Register-to-Memory (RM) instructions modify register contents based upon the effective operand and the function called for by the operation code. The instruction format is:



Field	Destination
OP	Operation Code
I	Indirect addressing
X	Index usage
B	Base Register usage
D	Displacement

Addressing can be either program counter relative or base register relative. The normal mode is program counter relative. However, the programmer can specify base relative addressing by setting bit 7 in the instruction. The assembler can also perform this function if it has been informed of the value residing in the base register by a Base Register Set assembler directive (paragraph 2-8.1).

The addressing mode is determined by the I, X, B, and D fields as shown in Table 2-1. SD indicates that the displacement field is used as a signed two's complement number. D indicates an unsigned positive displacement quantity. Thus, $-128 \leq SD \leq 127, 0 \leq D \leq 255$. P is the address of the next sequential instruction. The symbol X denotes the number contained in the Index Register, while B denotes the number contained in the Base Register. If immediate addressing is specified on a load, add, subtract, or algebraic compare instruction, D is treated as an eight-bit signed quantity and bit eight will be extended through bits

TABLE 2-1

IXB USAGE

IXB	Effective Operand	Description
000	(P+SD)	Program Counter Relative
001	(B+D)	Base Register Relative
010	(P+SD+X)	Program Counter Relative, Indexed
011	(B+X+D)	Base Register Relative, Indexed
100	((P+SD))	Indirect, Program Counter Relative
101	((B+D))	Indirect, Base Register Relative
110	((P+SD)+X)	Indirect, Program Counter Relative, Post-indexed
110	((P+SD+X))	Indirect, Program Counter Relative, Pre-indexed
111	D or SD	Immediate Addressing. D or SD is the operand

0 through 7 to give a 16-bit operand. If immediate addressing is specified on a store instruction, D is treated as the effective operand address. The notation (m) means the contents of core location m.

The index control bit in the Status Register permits optional Pre-indexing or Post-indexing. This controls the relation of indexing to indirect addressing. If the index control bit is one, indexing precedes indirect addressing. If the index control bit is zero, indexing follows indirect addressing. If indirect addressing is not involved, the two modes are equivalent.

If the B bit in the IXB field is zero and the displacement field contains zeros, the program counter will be incremented again during instruction execution. In this event, the next location in memory is referenced as either data or as an indirect address, but is skipped over in the instruction sequence. Thus, data or indirect addresses and instructions may be interspersed. When this feature is used, the instruction is referred to as an extended format instruction.

The general symbolic format of a RM instruction is:

NAME	OPERATION	OPERAND
LABEL	@OP	=*DISP,MODE

The name field, the asterisk which denotes indirect addressing, the @ which denotes extended format, and the = which denotes immediate operand, are optional. The = and * may not both be present. The MODE item indicates the addressing mode and may have a value of 0 through 7. It consists of the IXB bits of the object instruction. DISP and MODE are symbolic expressions. Typical symbolic statements are:

NAME	OPERATION	OPERAND
PT1	LDA	BUFFER+2
	STA	*ADDR
PT2	LDX	=-5
	@STA	ANS,X
	BIX	$-2

Table 2-2 defines the transliteration process performed by the assembler for RM instructions. Table 2-2 uses symbolic indicators for the mode expressions. The user must actually write equivalence statements himself for the assembler to interpret any such symbolism.

The assembler will translate the operand address expression in RM format instructions by first evaluating the expression as a 16-bit number. One of the following operations will be executed:

TABLE 2-2

RM FORMAT SYMBOLIC INTERPRETATION

@	=	*	Mode Expression	In Range P-Relative	In Range B-Relative	Base Register Assumed Value Given by Pseudo-Op	Assembler Action
NO	NO	NO	NONE	YES	N/A	N/A	IXB=0, D=P relative
NO	NO	NO	X	YES	N/A	N/A	IXB=2, D=P relative
NO	NO	NO	I	YES	N/A	N/A	IXB=4, D=P relative
NO	NO	YES	NONE or I	YES	N/A	N/A	IXB=4, D=P relative
NO	NO	NO	NONE	NO	YES	YES	IXB=1, D=B relative
NO	NO	NO	X	NO	YES	YES	IXB=3, D=B relative
NO	NO	NO	I	NO	YES	YES	IXB=5, D=B relative
NO	NO	YES	NONE or I	NO	YES	YES	IXB=5, D=B relative
NO	NO	NO	B	N/A	N/A	NO	IXB=1, D = absolute
NO	NO	NO	XB	N/A	N/A	NO	IXB=3, D = absolute
NO	NO	NO	IB	N/A	N/A	NO	IXB=5, D = absolute
NO	NO	YES	B or IB	N/A	N/A	NO	IXB=5, D = absolute
NO	NO	NO	B	N/A	YES	YES	IXB=1, D=B relative
NO	NO	NO	XB	N/A	YES	YES	IXB=3, D=B relative
NO	NO	NO	IB	N/A	YES	YES	IXB=5, D=B relative
NO	NO	YES	B or IB	N/A	YES	YES	IXB=5, D=B relative
NO	NO	NO	IX	YES	N/A	N/A	IXB=6, D=P relative
NO	NO	YES	X or IX	YES	N/A	N/A	IXB=6, D=P relative
NO	NO	NO	M	N/A	N/A	N/A	IXB=7, D = absolute
NO	NO	YES	XB or M	N/A	N/A	N/A	IXB=7, D = absolute
NO	YES	N/A	N/A	N/A	N/A	N/A	IXB=7, D = absolute
YES	YES	N/A	NONE	N/A	N/A	N/A	IXB=0, D=0, 2nd word = immediate value
YES	NO	N/A	NONE	N/A	N/A	N/A	IXB=4, D=0, 2nd word = indirect address
YES	NO	N/A	X	N/A	N/A	N/A	IXB=6, D=0, 2nd word = indirect address

Values given symbolically in the MODE expression column have the following values:

NONE = 0	X = 2	I = 4	IX = 6
B = 1	XB = 3	IB = 5	M = 7

a. Subtract a number one greater than the location counter. This will make the address Program Counter Relative.

b. Subtract a number given by a BRS directive. This will make the address Base Relative.

c. Use the number as is. This is done in extended format.

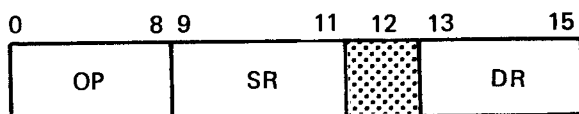
d. Truncate the number to an 8-bit value. This is done for single length immediate value references, and for base relative addressing under a base register reset (BRR) condition (paragraph 2-8.2).

e. Indicate a field size error if the resulting address is unattainable under the defined conditions.

RM instructions use the following mnemonics:

ADD DIV LDM
AND DLD LDE
BIX DMT LDX
BRL DSB MPY
BRU DST STA
CPA IMO STE
CPL IOR STX
DAD LDA SUB

2-6.2 REGISTER-TO-REGISTER (RR) INSTRUCTION FORMAT. The format of the Register-to-Register instruction is:



OP – Operation Code
SR – Source Register
DR – Destination Register

Register-to-Register instructions modify the contents of the destination register according to the operation code and using the source register.

The symbolic format of the RR instruction is:

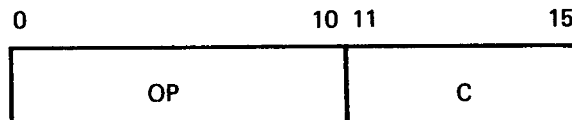
NAME	OPERATION	OPERAND
LABEL	OP	S,D

The label field is optional. S and D are expressions which, when evaluated, denote the source and destination registers, respectively.

RR instruction mnemonics are:

RAD RCO RIN
RAN RDE RIV
RCA REO RMO
RCL REX

2-6.3 SHIFT (S) INSTRUCTION FORMAT. Shift instructions have the format:



OP – Operation Code
C – Shift Count $0 \leq C \leq 31$

Shift instructions move data laterally within the registers.

Shift instructions have the symbolic format:

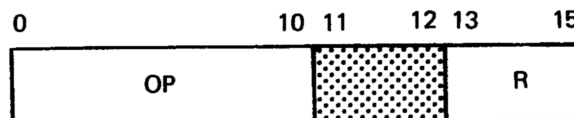
NAME	OPERATION	OPERAND
LABEL	OP	COUNT

The label field is optional. COUNT is an expression which is evaluated and used for the shift count.

S instructions use the following mnemonics:

ALA CRD LLD
ALD CRE LRA
ARA CRL LRD
ARD CRM LTO
CLD CRS LTZ
CRA CRX RTO
CRB LLA RTZ

2-6.4 REGISTER SKIP (RS) INSTRUCTION FORMAT. The format of register skip instructions is:



OP – Operation Code
R – Register Number

Register skip instructions cause the next instruction in sequence to be omitted if a specific condition exists in the referenced register.

The symbolic format of RS instructions is:

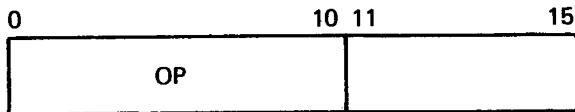
NAME	OPERATION	OPERAND
LABEL	OP	REG

The label field is optional. REG is an expression for the number of the register involved.

RS instruction mnemonics are:

SEV	SOD
SMI	SOO
SND	SPL
SNZ	SZE

2-6.5 STATUS INDICATOR SKIP (SS) INSTRUCTION FORMAT. These instructions do not require an operand. Their format is:



OP – Operation Code

Indicator skips omit the next instruction in sequence if a specific condition exists in the status register.

The SS instruction symbolic format is:

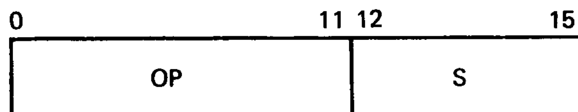
NAME	OPERATION	OPERAND
LABEL	OP	

where the label field is optional.

SS instruction mnemonics are:

SEQ	SLT	SNV
SGE	SNC	SOC
SGT	SNE	SOV
SLE		

2-6.6 SENSE SWITCH SKIP (SX) INSTRUCTION FORMAT.



S = Switch Indicators

These instructions omit the next instruction in sequence depending upon the setting of the indicated sense switches.

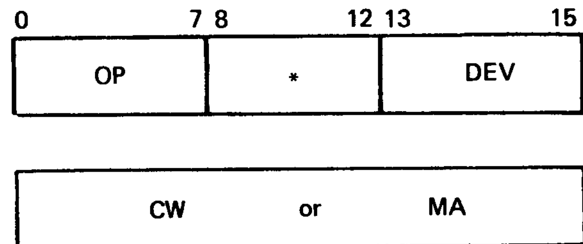
The symbolic format of the SX instructions is:

NAME	OPERATION	OPERAND
LABEL	OP	S

where the label field is optional.

The SX instructions are SSE and SSN.

2-6.7 DIRECT MEMORY ACCESS CHANNEL (DM) INSTRUCTION FORMAT. These instructions have the format:



OP – Operation Code
* – Device Dependent
DEV – Device Address
MA – Memory Address
CW – Control Word

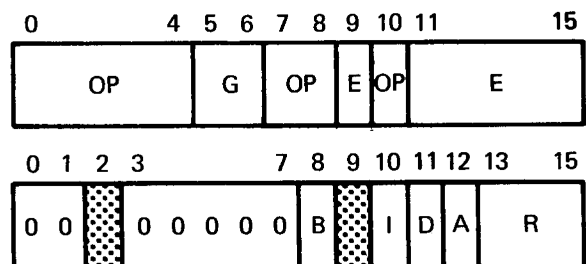
These two word instructions initiate direct memory access channel input and output.

The mnemonic code provided for the DMAC input/output has the symbolic format:

NAME	OPERATION	OPERAND
LABEL	ATI	DEV/CA
LABEL	DATA	CW or MA

The labels are optional.

2-6.8 DATA BUS INPUT/OUTPUT (DB) INSTRUCTION FORMAT. The format of a data bus I/O instruction is:



B – Sense Busy Designator
I – Increment Designator
D – Decrement Designator
A – Address Mode Designator
R – Register Number
G – External Device Group
E – External Register

These instructions transfer data between registers and external devices.

The symbolic format of DB instructions is:

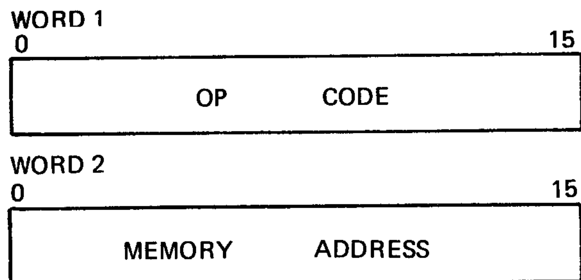
NAME	OPERATION	OPERAND
LABEL	OP	DEV
LABEL	DATA	BIDAR

The label fields are optional. DEV is an expression the value of which specifies register and group. The value is treated as a 16 bit number. It is tested for ones in bits 0-4 , 7, 8 and 10. If there are none it is logically ored with the OP code.

BIDAR is an expression the value of which will define the B, I, D, A, and R bits.

The DB mnemonics are RDS and WDS.

2-6.9 FIXED (F) INSTRUCTION FORMAT. Certain single-word or double-word instructions have a full 16-bit (fixed) operation code.



The second word is not applicable to single word instructions.

The symbolic format for F instruction is:

NAME	OPERATION	OPERAND
LABEL	@OP	DISP

All fields are optional except the OP field. The @ preceding the OP and DISP field are always used together and cause a double word F format instruction to be generated.

The mnemonics for F format instructions are NRM, IDL, LSB, and SSB.

2-7 DATA GENERATION.

A single operation is provided for data generation. Its format is:

NAME	OPERATION	OPERAND
LABEL	DATA	D1,D2,--,Dn

where LABEL is optional. D1,D2,--,Dn are expressions or strings. Expressions are evaluated and assigned to successive memory locations.

The DATA statement is used to define alphanumeric strings using following format:

NAME	OPERATION	OPERAND
LABEL	DATA	STRING

where STRING is a string of characters enclosed in single quotes. The string will be produced (ASCII code) two characters per word packed left to right. If there is an odd number of characters in the string, the last word will contain a space code in the last character position.

If a label is used, it will be assigned to the first core location involved.

2-8 ASSEMBLER DIRECTIVES.

Assembler directives have formats similar to symbolic instructions but do not directly cause code generation. Instead they are directives to the assembler itself and are provided for storage allocation, program identification, format control, and other similar functions. If labels are used with assembler directives, they will be assigned the current location counter value unless otherwise specified. Assembler directives are listed in Table 2-4.

2-8.1 BRS. The BRS directive informs the assembler of the current value in the Base Register. The operand field of the BRS directive itself is assumed to define a 16-bit value supposedly placed in the B Register by the programmer. When used, subsequent RM format instructions which would produce field size errors if Program Counter relative are generated Base Register relative, if possible. In this case, if D is an unsigned 16-bit evaluation of the displacement expression and B is the value assumed in the Base Register, then $0 \leq D-B \leq 255$ or else a field size error will occur.

An example of BRS usage follows:

```
        BRS    CAT     DEFINE BASE VALUE
*                      TO SAP-I
        .
        .
        .
        @LDA   =CAT    PUT ADDRESS OF CAT
*                      IN BASE REGISTER
        RMO    A,B
        .
        .
        .
        STA    CAT+3   ASSEMBLER USES BASE
*                      R E L A T I V E
*                      ADDRESSING
        .
        .
        .
CAT     BES    350     CAT IS DEFINED
        BSS    10
```

TABLE 2-3

EXAMPLE FOR DATA

```
ØAØ3    ØØØ1    Ø997   CAT     DATA 1,12,Ø12,>12,'12',CAT,,$ COMMENT
ØAØ4    ØØØC    Ø998
ØAØ5    ØØØA    Ø999
ØAØ6    ØØ12    1ØØØ
ØAØ7    B1B2    1ØØ1
ØAØ8    ØAØ3 P  1ØØ2
ØAØ9    ØØØØ    1ØØ3
ØAØA    ØAØA P  1ØØ4
```

TABLE 2-4

ASSEMBLER DIRECTIVES

BRS	Base Register Set
BRR	Base Register Reset
BSS	Block Starting With Symbol
BES	Block Ending With Symbol
ORG	Origin
DEF	Define Entry Point
REF	External Reference
IDT	Identification
HED	Heading
PEJ	Page Eject
LIS	List
UNL	Unlist
EQU	Equate
FRM	Format
IF	If (conditional assembly)
END	End of source input
OPD	Operation Define
COMM	Common Storage Allocation

3-8.2 **BRR.** BRR informs the assembler that the base register is not available to the assembler for addressing purposes. The programmer can still specify base register addressing in the mode field. The BRR directive informs the assembler to use the base register for addressing purposes only in the event the the mode field specifies that type of addressing. (This is the initial condition of assembly.) Under BRR directive control, if D is the unsigned displacement in RM instructions, then $0 \leqslant D \leqslant 255$ when the mode field contains B = 1 or else a field size error will occur.

2-8.3 **BSS.** BSS stands for BLOCK STARTED WITH SYMBOL. It reserves an area of memory. The first location in the reserved area is associated with the label given on the name field of the BSS directive. The location of the area reserved is that defined by the location counter, which is then advanced past the reserved area. Note that no object code is generated by the BSS directive. If the programmer desires some value(s) to be assembled in the reserved area he must do so by other means.

An example of the BSS directive is given below:

```
        BRU   TOM    BRANCH AROUND AREA
AREA    BSS   40     RESERVE AREA
TOM     LDA   AREA   REFERENCE AREA
```

It is possible to back up (or subtract from) the location counter by placing a negative operand in a BSS directive. For example the programmer may wish to let two or more symbols refer to the same core location. The following sequence will cause the names TOM, DICK, and HARRY to all reference the same location.

```
TOM     BSS   0
DICK    LDA   $+5
HARRY   BSS   0
```

Note that any symbol used as the operand of a BSS must be previously defined.

A common usage of symbols in a BSS operand is an expression which defines the length of a reserved area. In the following example, if the length of TABA is likely to change but TABB must be the same length, it may be symbolically stated.

```
TABA  BSS  50         MIGHT CHANGE
TABB  BSS  TABB-TABA  ALWAYS SAME AS TABA
      BSS  1
```

2-8.4 BES. This evaluates the operand field and advances the location counter by that amount. If a label is present, it is assigned to the new value of the location counter. BES stands for BLOCK END SYMBOL and is similar to BSS, except the label is applied to the first location past the reserved area.

2-8.5 ORG. This sets the value of the location counter to the value of the expression in the operand field.

Any symbol in the expression must be previously defined. If the operand field is invalid, the ORG directive will not be used. The ORG directive is commonly used to force loading of a program in specified core locations but may be used to perform other operations. For example

```
      ORG        $+500
```

increases the Location Counter by 500.

Therefore, the ORG directive provides an alternate way to reserve storage areas. The preferred way to reserve a storage area is with the BSS and BES assembler directives. However, where a storage area cannot be conveniently defined with the BSS directive, the ORG directive can be used.

NOTE

Use of the ORG directive causes the assembler to produce an absolute object program if the operand field is not a relocatable expression. Hence, the ORG directive should be used with care.

2-8.6 DEF. The program linking assembler directives DEF and REF allow the programmer to symbolically link independently assembled programs which are to be loaded and executed together. Symbolic linkages between programs are created by means of symbols defined in one program and used as operands in another program. Such symbols are termed linkage symbols. A linkage symbol is called a *defined entry point symbol* in the program in which it is defined; it is a *referenced external symbol* in the program in which it is used as an operand.

Every linkage symbol must be properly identified as such in the source program. A linkage symbol used as an external symbol is identified in each using program by the REF directive. A linkage symbol used as an entry point must be identified in the defining program by the DEF directive.

NAME	OPERATION	OPERAND
	DEF	A list of symbols

The relocatable symbols (separated by commas) in the operand field are defined elsewhere in the program and may be used as an entry point by other programs. A symbol that is not defined in the program is flagged in the listing as an error.

In the following sequence, SQRT is identified as an entry-point symbol.

```
SUBRO    BSS      10
         DEF      SQRT
          .
          .
          .
SQRT     STA      SAVE
```

2-8.7 REF. The REF directive identifies a linkage symbol as an external symbol that is referenced in this program. Each such external symbol must be identified in a REF directive. The format of the REF statement is:

NAME	OPERATION	OPERAND
	REF	A list of external symbols

The external symbols (separated by commas) in the operand field must be defined in another program and identified in that program as an entry-point symbol by the DEF directive.

As an example, if MTPLY is an entry point symbol in another program, the using program identifies it as an external symbol, thus:

```
REF      MTPLY
```

The only way an external symbol may be referenced is as a full 16-bit address. The external symbol may not be used in

an arithmetic calculation. In particular use of MTPLY+2 is an example of illegal usage. To link to a program named SINE, the following coding might be used:

```
PROGA     BSS       2
          REF       SINE
           .
           .
           .
ADSINE    @BRL      SINE
```

Use of SINE-2 would be illegal.

2-8.8 **IDT.** The symbol appearing in the operand field is reproduced in the object program as the program name section (Refer to paragraph 2-9).

2-8.9 **HED.** The remaining characters in the line are printed as page headings on the output listing. The first HED is used as the heading of the page on which it appears and subsequent pages until another HED is encountered. Subsequent HED's appear as page headings on the first page following the one on which the HED appears and subsequent pages until another HED is encountered.

2-8.10 **PEJ.** The remainder of the current listing page is ejected. The assembler begins a new page headed by the current HED. The PEJ itself will be printed as the first line on the new page.

2-8.11 **LIS.** Print the assembly listing. If it is desired to produce a complete listing, no LIS directive is required.

2-8.12 **UNL.** Suppress assembly listing except for error conditions and symbol table. Listing can be resumed by a LIS directive.

2-8.13 **EQU.** The EQU directive is used to define a symbol by assigning to it the value of an expression in the operand field. The format of the EQU instruction statement is:

NAME	OPERATION	OPERAND
SYMBOL	EQU	EXPRESSION

The symbol in the name field is given the same value as the expression. The expression in the operand field can be relocatable or absolute, and the symbol will be similarly defined. Any symbols in the expression must be previously defined.

If the expression in the operand field or the symbol in the name field, or both, are invalid, or are not present, the EQU statement will be flagged as in error in the listing and will not be used.

The EQU directive is the usual way of equating symbols to register numbers, input/output unit numbers, immediate data, actual addresses, and other arbitrary values. The examples below illustrate how this might be done:

```
REGX      EQU    2      REGISTER X
IO125     EQU    125    INPUT/OUTPUT DATA
TEST      EQU    >3F    IMMEDIATE DATA
TIMER     EQU    80     ACTUAL ADDRESS
```

To reduce programming time, the programmer can equate symbols to frequently used compound expressions and then use the symbols as operands in place of the expressions. Thus, in the statement

```
FIELD     EQU       ALPHA-BETA+GAMMA
```

FIELD will be defined as ALPHA-BETA+GAMMA and may be used in place of it. Note, however, that ALPHA, BETA, and GAMMA must all be previously defined.

2-8.14 **FRM.** Assign the label as an operation code. The operand field is of the form E1,E2,...,En, where the E expression values are positive and the sum is 16. When the label is used as an op code, n fields are evaluated, truncated to the length specified by the corresponding En and placed in the output word.

```
Example:            1006 PDQ FRM 3,4,1,8
          0A0B 32A7 1007        PDQ 1,9,, >A7
```

2-8.15 **IF.** The assembly process is altered in accordance with the results of a conditional test. The operand field consists of two expressions and an optional symbol. The two expressions are evaluated and compared. If they are not equal, the assembly process continues with the next line. If the values are equal, the assembly process is suspended. If only two expressions are contained in the operand field, assembly is suspended for one line. If three expressions occur, the third is a label following the IF statement. The assembly process is suspended until the input line is detected with this symbol in its name field.

All lines suspended from the assembly process are treated as comments; i.e., they are printed but no code is generated. Two or more IF statements may have overlapping ranges.

This directive allows assembly-time modification of a program.

2-8.16 **END.** The END directive terminates the assembly of a program. It also supplies a point in the program to which control is transferred after the program is loaded.

The END directive must always be the last statement in the source program.

The format of the END directive statement is:

NAME	OPERATION	OPERAND
	END	An expression or blank.

The expression in the operand field specifies the point to which control is transferred when loading is complete. If the operand field is invalid, the statement will be flagged as a possible error. If the operand field is blank, the first word of the program is taken as the point for transfer of control.

The point to which control usually is transferred is the first instruction in the program, as shown in this sequence:

```
NAME     OPERATION     OPERAND

         ORG           2000
AREA     BSS           50
BEGIN    LDA           =3
          .
          .
          .
         END           BEGIN
```

When several object programs are joined by link editing one will be specified as the main program. Its transfer point is taken as the transfer point for the link edited program.

2-8.17 OPD. The label is assigned as an operation code mnemonic. The first item in the operand field is evaluated as a 16-bit number and stored as the corresponding operation code. The second item in the operand field indicates the format type for the object instruction. When this label appears as an operation code mnemonic, it is assembled in the appropriate format with the assigned operation code and output in the object program. Any defined operation takes precedence over the standard symbolic operation codes. Format type codes are as follows:

SPACE - Register to Memory (RM)
0 - Register to Memory (RM)
1 - Register to Memory (RM)
2 - Register to Register (RR)
3 - Shift (S)
4 - Register Skip (RS)
5 - Status Indicator Skip (SS)
6 - Data Bus Input/Output (DB)
7 - Sense Switch Skip (SX)
8 - Direct Memory Access Channel (DM)

The final merging of the operation code and the operand fields is performed using a logical *or*. Thus the operation code may be used to force setting of any bit to one.

```
EXAMPLE          1009 XYZ OPD >9800,1
   OAOC  9AFF    1010 JOE  XYZ JOE,2   COMMENT
```

2-8.18 COMM. COMM directs allocation to common storage of the number of words indicated by the value of the expression in the operand field. If a label appears, it is assigned a value corresponding to the first word of the block.

Common storage is an area of memory the location of which is not known until load time. The area is *common* because separately assembled programs may share the same common storage. Unlike the BSS directive, COMM does not reserve area at the current location counter position.

COMM is used in a manner similar to FORTRAN COMMON. If a FORTRAN program and assembly language program are merged via link edit, any references in the FORTRAN program to *blank* COMMON and references in the assembly language program to COMM defined storage will be references to the same area of memory. In many applications this simplifies communications between the two programs.

2-9 OBJECT FORMATS.

The object program may be produced on either paper tape or cards or output in the paper tape format on disc.

2-9.1 OBJECT PAPER TAPE FORMATS. The object paper tape format is shown in Table 2-5.

All data on the object tape are punched as 16-bit words: four frames per word. The four low order channels of each frame contain one hexadecimal digit. Channel 5 is punched to produce an odd parity tape. Channels 6, 7, and 8 are always blank. The following table contains the hexadecimal digits and the hexadecimal codes corresponding to the eight channels on the tape.

Program Hex Digit	Object Hex Code	Program Hex Digit	Object Hex Code
0	10	8	08
1	01	9	19
2	02	A	1A
3	13	B	0B
4	04	C	1C
5	15	D	0D
6	16	E	0E
7	07	F	1F

For an assembler generated object tape, the program name is the name specified in the IDT assembly directive. Names less than six characters will have trailing blanks. If the IDT directive is not present, the name will consist of six blanks.

The format code specifies an absolute or relocatable assembly. Any assembly which contains an ORG statement with an absolute value in the operand field will be assembled as absolute. Otherwise, the assembly is relocatable. The format codes are 3 for absolute object and 5 for relocatable object. For absolute programs the program length is defined to be the highest numbered core location to be filled plus one.

Assembler generated object tapes will have no common reference or one common reference, depending upon whether the COMM assembler directive is used in the program. If a common reference exists, it is identified by

the symbol Δ BLANK, where Δ represents an ASCII blank.

Check sums are the negative of all data preceding the check sum and following the last check sum (or start of the tape). Thus, the sum of all data after one check sum up to and including the next check sum is always zero.

The stop code is the hexadecimal word 0030. When reading the tape with the teletypewriter reader, the reader will stop after reading this code.

A relocation block consists of a relocation word followed by up to sixteen text words. Relocation blocks are repeated, as required, to give the number of program words in the load block word count.

Successive load blocks appear until the end of the program. This is indicated by a dummy load block starting with a zero word count. The following word (four frames) is taken as the transfer location, or execution entry point.

The relocation properties of each word are specified by two bits of the relocation word which precedes the program text words in a relocation block. The high order two bits of the relocation word correspond to the first one or two of the following program words, etc. The two bits specify relocation as follows.

Relocation Bits	Meaning	Listing Identifier
00	Absolute – no relocation	Blank
01	Common Relocatable	C
10	Program Relocatable	P
11	External Reference	X

At load time, relocatable items will be modified by addition of the program base as required. External and common references must be satisfied by the Link Editor. External references which have not been resolved via link edit are treated as absolute data by the loader.

Each relocation block may contain up to 16 words of data determined as follows: for each external and common data item, two words are contained in the relocation block. For each absolute and program relocatable data item one word is contained in the relocation block.

Each external and common data item determines only one word of the object program even though two words are required on the object tape to describe the data item. The two words are as follows:

WORD 1
Common – reference number of common area
External – reference number of external variable

WORD 2
Common – displacement within that area
External – displacement from the external variable

2-9.2 OBJECT CARD DECK FORMAT. The object card deck format is shown in table 2-6. For object card output the format codes used are 0 – absolute, 1 – relocatable. The group 2 and group 3 cards (if any) are required when using object cards as input to the link editor. They must not be present when object cards are read by the card reader loader.

Successive load blocks will appear until the end of the program. A maximum of 16 program words per card exists. Fewer words are possible and will be designated by blank columns.

The last card contains four special punches (12-punch) in the load address field. The next four characters of that card (cc.5-8) will be the transfer location in coded hex.

In the case of common and external data in the relocation blocks only one word appears. That word is the displacement into blank common (for common data) or reference number of the external variable (for external data).

2-10 LISTING DIAGNOSTICS FOR SAP-I.

The various versions of the assembler may detect certain syntax errors in the source program. When this occurs, a diagnostic comment will appear in the listing produced by SAP-I. Listed below are the phrases which may appear adjacent to the line, in the listing, which contains the error indicated. These phrases apply only to those versions of SAP-I which operate in the Model 980 Computer. The IBM System/360 SAP-I produces error phrases which are slightly different, and are given in paragraph 2-12.

Comment	Meaning
UNDF OP	Undefined Operation Code
LONG SYM	Symbol > 6 Characters
MDF O/F	OPD or FRM Multiply defined
FRM > 16	FRM Fields contain more than 16 Bits
C AD > 10	Address Expression has >10 Elements
UNDF SYM	Symbol Not Defined
M DF SYM	Symbol Multiply Defined
RELOC	A relocation error
SYM OVF	Too Many Symbols Have Been Defined
BAD NUM	Numeric Element Not Valid
NO END	No END Assembler Directive Found
LN >72	Paper Tape Source Line is too Long
IMP R/D	A REF or DEF Symbol Has Been Used Improperly
X RF USE	A REF Symbol Has Appeared Invalidly In An Unrelocatable Expression
FIELD SZ	Address Beyond Reach
IXB ERR	Address Made Error
OPD ERR	No Such Format Number

TABLE 2-5

OBJECT FORMAT

Section	Field	Size
HEADER	PROGRAM NAME	12 Frames
	PROGRAM – LENGTH IN WORDS	4 Frames
	FORMAT CODE	4 Frames
	COMMON COUNT	4 Frames
	ENTRY POINT COUNT	4 Frames
	EXTERNAL REFERENCE COUNT	4 Frames
	STOP CODE	4 Frames
COMMON REFS	COMMON SYMBOL 1	12 Frames
	COMMON LENGTH 1	4 Frame
	COMMON SYMBOL 2	12 Frames
	COMMON LENGTH 2	4 Frames
	⋮	
	STOP CODE	4 Frames
ENTRY POINTS	ENTRY POINT SYMBOL 1	12 Frames
	ENTRY POINT ADDRESS 1	4 Frames
	ENTRY POINT SYMBOL 2	12 Frames
	ENTRY POINT ADDRESS 2	4 Frames
	⋮	
	STOP CODE	4 Frames
EXTERNAL REFS	EXTERNAL REFERENCE SYMBOL 1	12 Frames
	ZERO	4 Frames
	EXTERNAL REFERENCE SYMBOL 2	12 Frames
	ZERO	4 Frames
	⋮	
	CHECK SUM	4 Frames
	STOP CODE	4 Frames
LOAD BLOCK 1	WORD COUNT	4 Frames
	LOAD ADDRESS	4 Frames
	RELOCATION WORD	4 Frames
	UP TO 16 TEXT WORDS	4 Frames Each
	CHECK SUM	4 Frames
	STOP CODE	4 Frames
	RELOCATION WORD	4 Frames
	UP TO 16 TEXT WORDS	4 Frames Each
	CHECK SUM	4 Frames
	STOP CODE	4 Frames
	⋮	
LOAD BLOCK	WORD COUNT	4 Frames
	LOAD ADDRESS	4 Frames
	⋮	
END BLOCK	ZERO	4 Frames
	ENTRY ADDRESS	4 Frames
	CHECK SUM	4 Frames
	STOP CODE	4 Frames

TABLE 2-6

OBJECT CARD DECK FORMAT

	Card Column		
Group 1		IDENTIFICATION & CONTROL	
	1-6	Program Name	
	7	Format Code	
	8-11	Program Length (coded Hex)	
	12-15	Common Length (coded Hex)	
	16-18	Entry Point Count (Decimal)	
	19-21	External Reference Count (Decimal)	
	77-80	Card Number (Decimal)	
Group 2		ENTRY POINT	
	1-6	Entry Point Symbol	(repeated as per Entry Point Count). Maximum 7 per card. Columns 1-70.
	7-10	Entry Point Value	
	77-80	Card Number	
Group 3		EXTERNAL REFERENCE	
	1-6	External Reference Symbol (repeated 12 to a card (Column 1-72) as per count).	
	77-80	Card Number	
Group 4		LOAD BLOCK	
	1-4	Load Address (coded Hex)	
	5-8	Relocation Word (coded Hex)	
	9-40	Eight Program Words @four columns each (coded Hex)	
	41-44	Relocation Word (coded Hex)	
	45-76	Eight Program Words @four columns each (coded Hex)	
	77-80	Card Number	

2-11 SAP-I OPERATING PROCEDURES.

There are several versions of SAP-I, depending on equipment configuration and the operating system. The version which operates on an IBM System/360 is documented separately in paragraph 2-12. When SAP-I is run under control of RTM-I and either SPEX or BATCH, the operating procedures are as given in Sections V and VI.

The operating sequences given here apply to the versions of SAP-I which operate on a Model 980 Computer and are not supported by RTM-I. The following versions are available:

Source Device	Listing Device	Object Device
TTY or TI780	TTY or TI720	TTY or TI780
H.S.Reader	TTY or TI720	H.S.Punch
Card Reader	TTY or TI720	H.S.Punch
H.S.Reader	Line Printer	H.S.Punch
Card Reader	Line Printer	H.S.Punch

OPERATION:

a. Load program

   Results: Program is ready for execution.

b. Push RESET button and then push the RUN button.

   Results: A message is printed on the teletypewriter as follows: READY ASSEMBLY SOURCE AND HIT RUN.

c. Ready the source and push the RUN button.

   Results: The first pass is made. First pass errors, if any, are printed followed by a repeat of the above teletypewriter message.

d. Repeat step c until all source has been input to the first pass.

   Results: The assembler is ready for the second pass. The same message is printed on the teletypewriter, and the object media will be turned on.

e. Repeat steps c and d for second pass of assembly.

   Results: During this pass an assembled listing is printed and the object is punched. Assembly is completed. The same message is printed on the teletypewriter and the assembler is ready for the next assembly.

NOTE

Sense Switch 2, if on, will suppress the listing. Sense Switch 3, if on, will suppress object generation.

NOTE

All assembly source paper tapes should either have an equal (=) sign as the first character of a terminating line meaning there are more physical source reels of tape, or an END statement as the last line. If the above is not the case, the assembler will treat blank tape as an END statement, and cause the pass to come to an end, and print a NO END error message.

2-12 MODEL 980 SAP-I ON IBM SYSTEM/360.

TI980SIM is an IBM System/360 program to perform the functions of the Symbolic Assembly Program (SAP-I) of the Model 980 Computer. This assembler is written COBOL and operates on an IBM System/360 with a minimum configuration of:

65K bytes core memory for background
Card Reader
Card Punch
One Magnetic Tape
Printer

Input to TI980SIM is SAP-I source language statements on punched cards, one statement per card. Several such source programs may be batched in one assembly run with no IBM System/360 operator intervention or requirement for JCL between assemblies.

Output of TI980SIM consists of an assembly listing of the program, and a punched card object deck. Although the input is free form, the symbolic output will appear in fixed form with the label beginning in position 1, operation code in position 8, and operand field in position 13.

2-12.1 ERROR DIAGNOSTICS.

Error diagnostics produced by TI 980 SIM consist of short descriptive phrases printed on the assembly listing immediately preceding the line containing the error. The messages, shown below, are self-explanatory.

UNDEFINED REFERENCE
ADDRESS ERROR
DUPLICATE TAG
FORMAT ERROR
ILLEGAL OP CODE
EXPRESSION ERROR
SYNTAX ERROR

MODE ERROR
NO ENTRY POINT VALUE
SHIFT COUNT ERROR
REGISTER NO. ERROR
ILLEGAL I/O DEVICE
ILLEGAL SENSE SWITCH

In addition to the above, there are two errors that, if encountered, will terminate the assembly process for the current program. The messages SYMBOL TBL OVFLO and WORD TBL OVFLO will be typed upon the system console. In the one case the symbol table has exceeded the maximum, and in the other case either the external reference table or the entry point table has been filled. In both cases the assembly process is terminated for the current source program and will continue with the next assembly in the batch stream (if any).

2-12.2 TABLE SIZES.

a. Symbol Table
Holds all symbols encountered in the label field of the program, the address at which they were found, and a code as to their attribute (such as a 'C' for a label in a Common statement). Size is 512 symbols.

b. External Reference Table
Holds all external labels and their ordinal value, as defined by REF directives. Size is 32 entries.

c. Entry Point Table
Holds all labels described as being entry points into the program by DEF directives. Size is 32 entries.

d. FORMAT Structure Table (Format-Statement)
Holds the label (maximum of four characters) described as a future op code by an OPD or FRM directive. Size is 32 entries.

2-12.3 LOADING AND OPERATION. To load and operate TI980SIM, specific job control cards must be prepared. Tables 2-7, 2-8 and 2-9 demonstrate ways in which the user might wish to perform the operation on a DOS system. Regardless of the method used, the following must be taken into consideration. The I/O devices have been given definite assignments. The Card Reader has been assigned to 'SYS004', the Tape has been assigned to 'SYS010', the Card Punch has been assigned to 'SYS006', and the Printer has been assigned to 'SYS007'. If these assignments are not standard on the system on which TI980SIM is being run, //ASSGN cards must be prepared for the job control stream.

TABLE 2-7

TO GO FROM OBJECT DECK

```
// JOB Jobname
// ASSGN SYS010,X'280'
// ASSGN SYS006,X'006'
// ASSGN SYS007,X'004'
// ASSGN SYS004,X'002'
// OPTION LINK
   INCLUDE
   (IBM System/360 COBOL Object
    Deck of TI980SIM)
/*
// EXEC LNKEDT
// EXEC
   (980 Source Deck(s))
/*
/&
```

TABLE 2-8

TO CATALOG THE OBJECT DECK
INTO THE IBM SYSTEM/360

```
// JOB Jobname
// OPTION CATAL
   PHASE TI980SIM,*
   INCLUDE
   (IBM System/360 Object Deck
    of TI980SIM)
/*
// EXEC LNKEDT
/&
```

TABLE 2-9

TO GO FROM A CATALOGUED VERSION

```
// Job Jobname
// ASSGN SYS010,X'280'
// ASSGN SYS006,X'006'
// ASSGN SYS007,X'004'
// ASSGN SYS004,X'002'
// EXEC TI980SIM
   (980 Source Deck(s))
/*
/&
```

SECTION III

FORTRAN LANGUAGE PROGRAMMING

SECTION III

FORTRAN LANGUAGE PROGRAMMING

3-1 INTRODUCTION.

FORTRAN (FORmula TRANslation) is a language that closely resembles the language of mathematics; it is designed primarily for scientific and engineering computations. Since the language is problem-oriented rather than machine-oriented, it provides scientists and engineers with a way to communicate with a computer. FORTRAN is more familiar, easier to learn, and easier to use than assembly language for those problems for which it is suited.

A FORTRAN program is written as a sequence of statements.

ARITHMETIC statements which specify numerical or logical calculations.

CONTROL statements which specify the flow of control in the program.

INPUT-OUTPUT statements which govern the transmission of data between computer memory and various input-output units.

DECLARATION statements which supply descriptive information about the program.

3-2 PROGRAM PREPARATION.

The statements of a FORTRAN program are written on a coding form. Each statement is written on a separate line. If more than one line is required for a statement, as many as nineteen continuation lines may be used. Each line of the coding form is divided into 72 columns, each of which may contain one character. Each line is punched in paper tape followed by the line terminating characters (carriage return, reader off, rub-out and line feed).

Columns 1 through 5 of the first line of a statement may be used for the statement number. Statement numbers serve as statement identifiers for cross references.

The statement number consists of 1-5 digits of any value. Leading zeros are ignored, however, zero may not be a statement number. Statement numbers may appear anywhere in the statement number field but must not contain any non-numeric characters. The statement numbers may be assigned in any order; the sequence of operations is always dependent upon the order of the statements in the program, not on the value of the statement numbers.

Superfluous statement numbers may decrease efficiency during compilation and should, therefore, be avoided.

A non-zero, non-blank character in column 6 indicates that the line is a continuation line.

The statement proper is written in columns 7 through 72 of the initial and continuation lines. Excepting certain alphanumeric fields, blanks are ignored in these columns and may be used freely to aid readability.

Columns 73-80 are not used by the FORTRAN Compiler and may, therefore, be used for program identification, sequencing, or any other purpose.

Comments to explain the program may be written in columns 2-72 of a line if the character C is placed in column 1. Comments may appear anywhere except before a continuation line or after an END statement. The comments are not processed by the FORTRAN Compiler.

Every program must physically terminate with an END line.

A FORTRAN program 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

3-3 ARITHMETIC AND DATA.

In Model 980 FORTRAN both numerical and logical arithmetic are available to the programmer. The meaning and use of six distinct types of data (integer, real, double precision, complex, logical, and Hollerith) are described in the following paragraphs.

3-3.1 NUMERICAL ARITHMETIC.

BASIC ELEMENTS. There are three basic elements used in Model 980 FORTRAN numerical arithmetic: constants,

variables, and function references. All of these basic elements represent numerical quantities. There are four distinct types of numerical quantities: integer, real, double precision, and complex.

Integer quantities are used to represent the integers. They are represented in the machine in binary integer form and occupy one word.

Real quantities are used to represent the real numbers. They are represented in the machine in binary floating point form and occupy two words.

Double precision quantities represent the real numbers to greater precision than real quantities. Double precision quantities occupy three words.

Complex quantities represent the complex numbers. They consist of an ordered pair of real numbers representing respectively the real and imaginary parts of the complex number. Complex quantities occupy four words ordered thus:

Real part (two words)
Imaginary part (two words)

INTEGER CONSTANTS. An integer constant is a number written without a decimal point. An integer constant may have any value in the range -32767 ($-2^{15}+1$) to 32767 ($2^{15}-1$), including zero.

Commas are not permitted within integer constants. A preceding plus sign is optional for positive numbers. Any unsigned constant is assumed to be positive.

The following examples are valid integer constants:

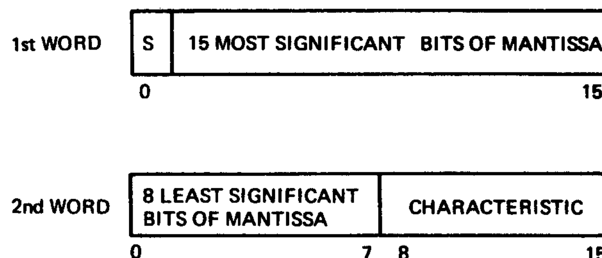
0
91
-173
+327

The following are *not* valid integer constants:

3.2	(contains a decimal point)
27.	(contains a decimal point)
31459036	(exceeds the magnitude permitted by the compiler)
5,496	(contains a comma)

REAL CONSTANTS. A real constant is a number written with a decimal point and consisting of 1-6 significant decimal digits.

Single precision (real) constants provide up to 23 significant bits of precision (six significant digits) stored in memory as shown below:



The magnitude of a real constant must not be greater than 2^{127} nor less than 2^{-128} (approximately 10^{38} and 10^{-39}) except that it may be zero.

A real constant may be followed by a decimal exponent written as the letter E followed by an integer constant (signed or unsigned) indicating the power of 10.

The following examples are valid real constants:

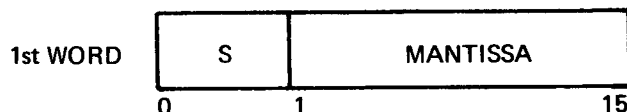
105.	
3.14159	
5.E3	(5.0×10^3)
5.0E3	(5.0×10^3)
-5.0E03	(-5.0×10^3)
5.0E-3	(5.0×10^{-3})
5.0E1	(5.0×10)
5E3	($5. \times 10^3$)

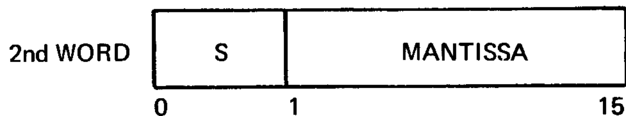
The decimal point may be omitted when using E notation.

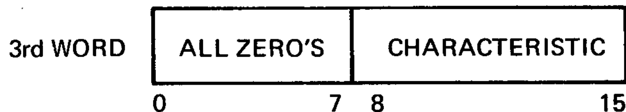
The following are *not* valid real constants:

325	(no decimal point; however, this is a valid integer constant)
5.0E	(no exponent)

DOUBLE PRECISION CONSTANTS. Double precision constants provide up to 30 significant bits of precision (9 significant decimal digits) stored in core storage as shown below:







Double precision constants are written the same as real constants except that the scale factor is specified by using a D instead of an E. The scale factor must be present.

Examples:

1D-15
2.718281828 DO

COMPLEX CONSTANTS. Complex constants are written as an ordered pair of real constants separated by a comma and enclosed in parentheses.

Examples:

(.70712,-.70712)
(-2E3,3E-5)

The first constant of the pair represents the real part of the complex number, and the second constant represents the imaginary part. The real and imaginary parts may each be signed. The enclosing parentheses are part of the constant and always appear, regardless of context.

VARIABLES. A FORTRAN variable is a symbolic representation of a quantity that may assume different values. The value of a variable may change either for different executions of a program or at different stages within the program. For example, in the statement:

A = 5.0 + B

both A and B are variables. The value of B is determined by some previous statement and may change from time to time. The value of A varies whenever this computation is performed with a new value for B.

VARIABLE NAMES. A variable name consists of one or more alphanumeric characters, excluding special characters, the first of which must be alphabetic. Blanks in a variable name will be ignored by the compiler. The string may be of any length but only the first 6 characters are used.

Examples:

X15
PERMUTATION
STRAIN

The rules for naming variables allow for extensive selectivity. In general, it is easier to follow the flow of a program if meaningful symbols are used wherever possible. For example, to compute distance it would be possible to use the statement:

X = Y*Z (Asterisk denotes multiplication)

but it would be more meaningful to write:

D = R*T

or:

DIST = RATE*TIME

Similarly, if the computation were to be performed using integers, it would be possible to write:

I = J*K

but it would be more meaningful to write:

ID = IR*IT

or

IDIST = IRATE*ITIME

In other words, variables can often be written in a meaningful manner by using an initial character to indicate whether the variable represents an integer or real value and by using succeeding characters as an aid to the user's memory.

VARIABLE TYPES. The type of a variable corresponds to the type of data the variable represents (i.e., integer, real double precision, or complex). Variables can be specified in two ways: implicitly or explicitly.

IMPLICIT SPECIFICATION. Implicit specification of a variable is made as follows:

a. If the first character of the variable name is I, J, K, L, M, or N, the variable is an integer variable.

b. If the first character of the variable name is *not* I, J, K, L, M, or N, the variable is a real variable.

EXPLICIT SPECIFICATION. Explicit specification of a variable type is made by using the Type statement (see Paragraph 3-6.2). The explicit specification overrides the implicit specification. For example, if a variable name is ITEM and a Type specification statement indicates that this variable is real, the variable is handled as a real variable, even though its initial letter is I.

SUBSCRIPTED VARIABLES. A subscripted variable represents a single element of an array of quantities. The variable is denoted by the array name followed by a subscript list enclosed in parentheses. The subscript list is a sequence of integer expressions separated by commas. Each expression corresponds to a subscript and the values of the expressions determine which element of the array is referenced. The value of a subscript must lie within the limits specified for the array. The number of subscripts must equal the dimension specified for the array. Arrays may have any number of dimensions.

Examples:

```
Y(1)
STATION (K)
Q(LINE(N,X)+RH,N)
```

ARRAYS AND SUBSCRIPTS. An array is an ordered set of data that is referred to by a single name. Each individual element in the set is referred to in terms of its position in the set. For example, assume that the following is an array named NEXT:

15
12
18
42
19

To refer to the second element in the group in ordinary mathematical notation, the form $NEXT_2$ would be used. In FORTRAN the form would be NEXT(2). The quantity 2 is called a subscript. Thus, NEXT(2) has the value 12 and NEXT(4) has the value 42.

Similarly, an ordinary mathematical notation might use $NEXT_i$ to represent any element of the array NEXT. In FORTRAN, this is written as NEXT(I) where I equals 1, 2, 3, 4, or 5.

The array could be two dimensional. For example, assume the following is the array LIST:

	COLUMN1	COLUMN2	COLUMN3
ROW1	82	4	7
ROW2	12	13	14
ROW3	91	1	31
ROW4	24	16	10
ROW5	2	8	2

To refer to the number in row 2, column 3, $LIST_{2,3}$ would be used in ordinary mathematical notation. In FORTRAN, the form LIST(2,3) would be used where 2 and 3 are the subscripts. Thus, LIST(2,3) has the value 14 and LIST(4,1) has the value 24.

Ordinary mathematical notation uses $LIST_{i,j}$ to represent any element of the two-dimensional array LIST. In FORTRAN, this is written as LIST(I,J) where I equals 1, 2, 3, 4, or 5, and J equals 1, 2, or 3. This indexing convention extends in a similar manner for any number of dimensions.

FUNCTION REFERENCE. A numerical function is a subprogram which acts upon one or more quantities called arguments, and produces a single numerical quantity called the function value. Function references are denoted by the identifier which names the function followed by an argument list enclosed in parentheses:

identifier (argument,argument,...argument)

At least one argument must be present. An argument may be an expression, an array identifier, or a subprogram identifier.

A function reference represents a quantity, namely the function value, and acts as a basic element. The type of the function value is given by the type of the identifier which names the function.

The type of the function is independent of the types of its arguments.

Examples:

```
COS (THETA)
ZETA (S+SQRT(S))
```

3-3.2 ARITHMETIC EXPRESSIONS. An arithmetic expression is a sequence of basic elements separated by numerical operators and parentheses in accordance with mathematical convention and the rules given below. An arithmetic expression has a single numerical value, namely,

the result of the calculations specified by the quantities and operators comprising the expression.

The numerical operators are +, -, *, /, **, denoting, respectively, addition or unary plus, subtraction or unary minus, multiplication, division, and exponentiation.

An expression may consist of a single element (constant, variable, or function reference):

```
2.71828
Z(N)
TAN (THETA)
```

Compound expressions may be formed by using operators to combine basic elements:

```
X+3
TOTAL/POINTS
TAN (PI*M)
```

Any expression may be enclosed in parentheses and considered to be a basic element:

```
(X+Y)/2
(ZETA)
COS (SIN(PI*M)+X)
```

Expressions may be preceded by a + or - (unary plus or unary minus):

```
+X
-(ALPHA*BETA)
-SQRT (-GAMMA)
```

However, no two operators may appear in sequence. For instance:

```
X*-Y
```

is improper. Use of parentheses yields the correct form:

```
X*(-Y)
```

If the precedence of operations is not given explicitly by parentheses, it is understood to be the following (in order of decreasing precedence):

Operator	Operation
–	Unary minus
**	Numerical exponentiation
* and /	Numerical multiplication and division
+ and -	Numerical addition and subtraction

For example the expression

```
A*B+C/D**E
```

is taken to be

```
(A*B)+(C/(D**E))
```

Sequences of operations of equal precedence can result in ambiguous expressions such as:

```
A**B**C
X/Y/Z
```

In Model 980 FORTRAN such ambiguous sequences are understood to be grouped from the left. For instance, the two sequences above are interpreted:

```
(A**B)**C
(X/Y)/Z
```

NOTE

Parentheses may not be used to imply multiplication. The asterisk arithmetic operator must always be used for this purpose. Therefore, the algebraic expression:

$$(AxB)\ (-C^D)$$

must be written as:

```
(A*B) * (-C**D)
```

EVALUATION OF ARITHMETIC EXPRESSIONS. The value of an arithmetic expression may be of integer, real, double precision, or complex type. The type of the expression is determined by the types of its elements according to the rules which follow.

The numerical types are ranked as follows:

Rank	Type
1	Integer
2	Real
3	Double Precision
4	Complex

The type of an expression is the type of the highest ranking element in the expression. The type of a subscripted expression is determined solely by the type of the variable.

An expression is evaluated by converting all elements to the expression type and performing all arithmetic according to this type. For example, an expression containing integer and complex elements is evaluated by converting all integer elements to complex elements and performing complex arithmetic throughout.

An integer expression is evaluated using binary integer arithmetic throughout, giving an integer value as the result. In integer arithmetic fractional parts arising in division are truncated, not rounded. For example:

7/3 yields 2; 6/7 yields 0.

In exponentiation (**), the types of the base and exponent are restricted as follows:

a. Complex exponents are not allowed.

b. Only integer exponents may be used with complex bases.

3-3.3 LOGICAL ARITHMETIC.

BASIC ELEMENTS. There are four basic elements used in Model 980 FORTRAN logical arithmetic: constants, variables, functional references, and relations. All of these basic elements represent logical quantities, the fifth type of arithmetic quantity.

A logical quantity may have either of two values; true or false. Logical quantities occupy only one word.

LOGICAL CONSTANTS. There are two logical constants written thus:

```
.TRUE.
.FALSE.
```

representing the values true and false, respectively. The enclosing periods are part of the constant and always appear, regardless of context.

LOGICAL IDENTIFIERS. Logical identifiers are written in the same way as numerical identifiers. They must be explicitly declared to be of logical type.

LOGICAL VARIABLES. Logical variables are written in the same way as numerical variables. The identifiers used to name the variables must be logical identifiers.

LOGICAL FUNCTION REFERENCES. References to logical functions are written in the same way as references to numerical functions. The identifier used to name the function must be a logical identifier. The function arguments may be expressions (logical or numerical), array identifiers, or subprogram identifiers.

RELATIONS. Relations are constructed from numerical expressions of integer, real, or double precision type through use of relational operators. The relational operators are:

Operator	Relation
.GT.	greater than
.GE.	greater than or equal to
.LT.	less than
.LE.	less than or equal to
.EQ.	equal to
.NE.	not equal to

The enclosing periods are part of the operator and must be present.

Two expressions of integer, real, or double precision type separated by a relational operator form a relation. For example:

```
X+2.LE.3*Y
```

is a relation. The entire relation constitutes a basic logical element.

The value of such an element is true if the relation expressed is true; otherwise, the value is false. In the example above, the element has the value true if X is 2 and Y is 2, and the value false if X is 2 and Y is 1.

Relational operators have lower precedence than arithmetic operators.

FORMATION OF LOGICAL EXPRESSIONS. A logical expression is a sequence of logical elements separated by logical operators and parentheses in accordance with the rules given below. A logical expression has a single value, true or false. This value is the result of the calculations specified by the quantities and operators comprising the expression.

The logical operators are .NOT., .AND., and .OR. denoting respectively logical negation, logical multiplication, and logical addition. The enclosing periods are part of the operators and must be present. The logical operators are defined as follows (where P and Q are logical expressions):

.NOT.P	true if P is false false if P is true
P.AND.Q	true if P and Q are both true, otherwise false
P.OR.Q	false if P and Q are both false, otherwise true

A logical expression may consist of a single logical element. For example:

```
.TRUE.
BOOL (N)
X.GE.3.14159
```

Single elements may be combined through use of the logical operators .AND. and .OR. to form compound expressions, such as:

```
TVAL.AND.INDEX
BOOL (M).OR.K.EQ.LIMIT
```

Any logical expression may be enclosed in parentheses and regarded as an element:

```
(T.OR.S).AND.(R.OR.Q)
(BOOL(M))
PARITY((2.GT.Y.OR.X.GE.Y).AND.NEVER)
```

Any logical expression may be preceded by the operator .NOT. as in:

```
.NOT.T
.NOT.X+7.GT.Y+Z
BOOL(K).AND..NOT.(TVAL.OR.R)
```

When the precedence of operations is not given by parentheses, it is understood to be the following (in decreasing order of precedence):

Operator	Operation
.NOT.	logical negation
.AND.	logical multiplication
.OR.	logical addition

Thus the expression

```
T.AND..NOT.S.OR..NOT.P.AND.R
```

is interpreted

```
(T.AND.(.NOT.S)).OR.((.NOT.P).AND.R)
```

3-3.4 SUMMARY OF OPERATOR PRECEDENCE. When the precedence of operators is not given explicitly by parentheses, it is understood to be as follows (in order of decreasing precedence):

Operator	Operation
–	negation
**	exponential
*,/	multiply, divide
+,-	add, subtract
.GT.,.GE., .LT.,.LE., .EQ.,.NE.	relational
.NOT.	logical negation
.AND.	logical multiply
.OR.	logical add

any remaining ambiguities are resolved by evaluating left-to-right.

For example, the logical expression

```
.NOT.ZETA**2+Y*MASS.GT.K-2.OR
.PARITY.AND.X.EQ.Y
```

is interpreted

```
(.NOT.(((ZETA**2)+(Y*MASS)).GT.(K-2))).OR
.(PARITY.AND(X.EQ.Y))
```

3-3.5 ARITHMETIC STATEMENTS. The Arithmetic statement is similar to a mathematical equation.

General Form:
A = B

where:

A is any variable (subscripted or nonsubscripted), and
B is an arithmetic expression.

In an Arithmetic statement the equal sign means *is to be replaced by* rather than *is equal to.* This distinction is important; for example, suppose the integer variable I has the value 3. Then, the statement

```
I = I + 1
```

would give I the value 4. This technique enables the programmer to keep counts and perform other required operations in the solution of a problem.

Examples:

```
Y = 2*Y
P = .TRUE.
X(N) = N*ZETA(ALPHA*M/PI)+(1.0,-1.0)
```

In evaluating arithmetic statements, the five arithmetic types are separated into the following sets:

SET	TYPES
Set 1	integer real double precision
Set 2	complex
Set 3	logical

Combinations of statement variable type and expression value type are subject to the following rules:

a. Any assignment between types of the same set is permitted.

b. No assignment from one set to another set is allowed.

Thus, for example, the combinations

integer = integer
integer = double precision
double precision = integer

are allowed. But neither

complex = real

nor

real = complex

are permitted.

For permissible combinations the expression value is made to agree in type with the statement variable before replacement takes place. For instance, in the statement

THETA = W*/ZETA+E)

if THETA is integer and the expression is real, the expression value is truncated to an integer before replacing THETA.

3-3.6 USE OF HOLLERITH DATA. A Hollerith datum is a string of characters. This string may contain any ASCII characters and need not be restricted to the FORTRAN character set defined in 3-2. The blank character is significant in Hollerith data.

A Hollerith constant is written as an integer n followed by the letter H followed by n ASCII characters.

A Hollerith constant may be written only in the argument list of a CALL statement or in a DATA statement. By use of the DATA statement a variable may be made to carry a Hollerith datum. Note that this implies at most 2 characters may be stored in an integer datum, 4 in a real datum, etc. If less than the maximum are specified, the characters are left justified.

3-4 CONTROL.

3-4.1 INTRODUCTION. The second class of FORTRAN statements is composed of control statements that enable the programmer to control the course of the program. Normally, statements are executed sequentially. That is, after one statement has been executed, the statement immediately following it is executed. However, it is often undesirable to proceed in this manner.

3-4.2 UNCONDITIONAL GO TO STATEMENT. This statement interrupts the sequential execution of statements, and specifies the number of the next statement to be performed.

General Form:
GO TO n

where

n is a statement number.

This statement transfers control to the statement numbered n.

Examples:

GO TO 25
GO TO 63468

The first example causes control to be transferred to the statement numbered 25; the second example causes control to be transferred to the statement numbered 63468.

3-4.3 COMPUTED GO TO STATEMENT. This statement also indicates the statement that is to be executed next. However, the statement number that the program is transferred to can be altered during execution of the program.

General Form:
GO TO $(n_1, n_2, ..., n_m)$, i

where

n_1, n_2, ..., n_m are statement numbers and i is a non-subscripted integer variable whose value is greater than or equal to 1 and less than or equal to the number of statement numbers within the parentheses.

This statement causes control to be transferred to statement $n_1, n_2, ..., n_m$, respectively.

Example:

```
GO TO (10, 20, 30, 40), ITEM
```

In this example, if the value of ITEM is 3 at the time of execution, a transfer occurs to the statement whose number is third in the series (30). If the value of ITEM is 4, a transfer occurs to the statement whose number is fourth in the series (40), etc.

3-4.4 ASSIGN AND ASSIGNED GO TO STATEMENTS.

General Form:
ASSIGN *i* to *variable*

.
.
.

GO TO *variable*
GO TO variable, (n_1, n_2, n_3, ..., n_k)

where *i* is an executable statement number. *Variable* is a non-subscripted integer variable.

The second form of the assigned GO TO is allowed for compatibility with other FORTRAN's only; the labels n_1, ..., n_k are not used.

This statement transfers control to the statement whose number was assigned to the variable. The assignment must take place in a previously executed ASSIGN statement.

The variable is a control variable, having a label as a value, not a numerical quantity. At the time of execution of an assigned GO TO statement, the current value of variable must have been defined by the previous execution of an ASSIGN statement. The value of the integer variable is *not* the integer statement number; ASSIGN 10 TO I is *not* the same as I = 10.

Example 1:

```
ASSIGN 40 TO ERROR
      .
      .
      .
GO TO ERROR
```

In example 1, control is transferred to the statement numbered 40.

Example 2:

```
GO TO N, (10,25,8)
```

If the current assignment of the integer variable N is statement number 8, then the statement numbered 8 is executed. If the current assignment of N is statement number 10, then the statement numbered 10 is executed next. If N is assigned statement number 25, statement 25 is executed next.

Example 3:

```
              .
              .
              .
      ASSIGN 10 TO ITEM
              .
              .
              .
13    GO TO ITEM, (8,12,25,50,10)
              .
              .
              .
8     A = B + C
              .
              .
              .
10    B = C + D
      ASSIGN 25 TO ITEM
      GO TO 13
              .
              .
              .
25    C = E**2
```

In example 3, the first time statement 13 is executed, control is transferred to statement 10. On the second execution of statement 13, control is transferred to statement 25.

3-4.5 ARITHMETIC IF STATEMENT. The simple GO TO statement causes an unconditional transfer of control to the statement having the statement number written after the GO TO. That is, execution of the transfer does not depend on any condition of the data, status of the machine, or anything else. The unconditional GO TO is important and heavily used, but by itself it would permit little work to be done. It is also necessary to be able to transfer if some condition is met during program execution. This is the function of the arithmetic IF statement.

The arithmetic IF statement has the form

IF (e)n_1, n_2, n_3

in which e stands for any expression and n_1, n_2, and n_3 are statement numbers. The operation of the statement is as follows: if the value of the expression within parentheses is

negative, the statement having statement number n_1 is executed next; if the value of the expression is zero, statement n_2 is executed next; if the expression is positive n_3 is executed next.

Example:

```
      IF (A(J,K)**3-B) 10,4,30
            .
            .
            .
4     D = B + C
            .
            .
            .
30    C = (D + C)**2
            .
            .
            .
10    E = (F*B)/(D+1)
```

In this example, if the value of the expression (A(J,K)**3-B) is negative, statement number 10 is executed next. If the value of the expression is zero, statement number 4 is executed next. If the value of the expression is positive, statement number 30 is executed next.

3-4.6 LOGICAL IF STATEMENT. Another tool for transfer of control is the logical IF statement, which has the general form

```
      IF (e) S
```

where e is a logical expression and S is any other statement except another logical IF or a DO (discussed later). The commonest form of logical expression in this context is one that asks a question about two arithmetic expressions. We write such relational expressions by using the six relational operators:

Relational Operator	Meaning
.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

The action of the logical IF is as follows: if the logical expression is true, statement S is executed; if the logical expression is false, statement S is not executed. Either way, the next statement executed is the one following the logical IF, unless S was a GO TO and the expression was true.

Example 1:

```
            .
            .
            .
      IF (A.LE.0) GO TO 25
      C = D + E
            .
            .
            .
25    W = X**Z
```

If the value of the first expression is .TRUE. (i.e., A is less than or equal to 0), the statement GO TO 25 is executed next and control is passed to statement number 25. If the value of the expression is .FALSE. (i.e., A is greater than 0), the statement GO TO 25 is ignored and control is passed to the next sequential instruction.

Example 2:

Assume that P and Q are logical variables.

```
            .
            .
            .
      IF (P.OR..NOT.Q) A = B
      C = B**2
```

In the first statement, if the value of the expression is .TRUE., the value of A is replaced by the value of B and the second statement is executed next. If the value of the expression is .FALSE., the statement A = B is skipped and the second statement is executed.

3-4.7 DO STATEMENT. The ability of a computer to repeat the same operations using different data is a powerful tool that greatly reduces programming effort. There are several ways to accomplish this when using the FORTRAN language. For example, assume that a manufacturer carries 1,000 different parts in inventory. Periodically, it is necessary to compute the stock on hand of each item (STOCK) by subtracting stock withdrawals of that item (OUT) from the previous stock on hand. These results could be achieved by the following statements:

```
            .
            .
            .
5     I=0
10    I=I + 1
25    STOCK (I) = STOCK (I) - OUT (I)
15    IF (I-1000) 10,30,30
```

The three statements (5, 10, 15) required to control this loop could be replaced by a single DO statement.

```
        .
        .
        .
  DO 25 I = 1, 1000, 1
  STOCK(I) = (STOCK(I) - OUT(I))
```

General Form:

```
  DO ni = m1, m2
  or
  DO ni = m1, m2, m3
```

where:

n is a statement number.

i is a non-subscripted integer variable.

m_1, m_2, m_3 are unsigned integer constants or non-subscripted integer variables. If m_3 is not stated (it is optional), its value is assumed to be 1. In this case, the preceding comma must also be omitted.

The DO statement is a command to repeatedly execute the statements that follow, up to and including the statement n. The first time the statements are executed, i has the value m_1, and each succeeding time, i is increased by the value of m_3. After the statements have been executed with i equal to the highest value that does not exceed m_2, control passes to the statement following statement number n. This is called a normal exit from the DO statement.

The *range limit* (n) defines the range of the DO. The range is the series of statements to be executed repeatedly. It consists of all statements following the DO, up to and including statement n. The range can consist of any number of statements.

The *index* (i) is an integer variable that is incremented for each execution of the range of statements. Throughout the range of the DO, the index is available for use either as a subscript or as an ordinary integer variable. However, the index may not be changed by a statement within the range of the DO. When transferring out of the range of a DO, the index is available for use and is equal to the last value it attained.

The *initial value* (m_1) is the value of the index for the first execution of the range. The initial value cannot be zero or negative.

The *test value* (m_2) is the value that the index must not exceed. After the range has been executed with the highest value of the index that does not exceed the test value, the DO is completed and the program continues with the first statement following the range limit. The test value is compared with the index value at the end of the range; therefore, a DO loop will always be executed at least once.

The value m_2 must be greater than or equal to m_1.

The *increment* (m_3) is the amount by which the value of the index will be increased after each execution of the range. The increment may be omitted, in which case it is assumed to be 1. If used, m_3 must be greater than 0.

Example 1:

```
      DO 25 I = 1,10
5          .
10         .
15         .
20         .
25    A=B+C
26         .
```

This example shows a DO statement that will execute statements 5, 10, 15, 20, and 25 ten times. Upon each execution, the value of I will be incremented by 1. After completion of the DO, statement 26 is executed.

Example 2:

```
           .
           .
           .
      DO 25 I = 1,1000
25    STOCK(I) = STOCK(I)-OUT(I)
      A = B+C
           .
           .
           .
```

In example 2, the DO variable, I, is set to the initial value of 1. Before the second execution of statement 25, I is increased by the increment, 1, and statement 25 is again executed. After 1000 executions of the DO loop, I equals 1000. Since I is now equal to the highest value that does not exceed the test value, 1000, control passes out of the DO loop and the third statement is executed next. Note that the DO variable I is now undefined; its value is not necessarily 1000 or 1001.

Example 3:

```
           .
           .
           .
      DO 25 I=1, 10, 2
      J = I+K
25    ARRAY(J) = BRAY(J)
      A = B + C
           .
           .
           .
```

In example 3, statement 25 is the end of the range of the DO loop. The DO variable, I, is set to the initial value of 1. Before the second execution of the DO loop, I is increased by the increment, 2, and the second and third statements are executed a second time. After the fifth execution of the DO loop, I equals 9. Since I is now equal to the highest value that does not exceed the test value, 10, control passes out of the DO loop and the fourth statement is executed next. Note that the DO variable I is now undefined; its value is not necessarily 9 or 11.

PROGRAMMING CONSIDERATIONS IN USING A DO LOOP.

a. The indexing parameters of a DO statement (i, m_1, m_2, m_3) should not be changed by a statement *within* the range of the DO loop.

b. There may be other DO statements within the range of a DO statement. All statements in the range of the inner DO must be in the range of the outer DO. A set of DO statements satisfying this rule is called a nest of DO's.

Example 1:

```
       DO 50 I = 1, 4                          }
       A(I) = B(I)**2                          } Range of
       DO 50 J=1, 5        } Range of          } Outer DO
50     C(J+1) = A(I)       } Inner DO          }
```

Example 2:

```
       DO 10 INDEX = L, M                      }
       N = INDEX + K                           } Range of
       DO 15 J = 1, 100, 2    } Range of       } Outer DO
15     TABLE(J) = SUM(J,N)-1  } Inner DO       }
10     (B(N) = A(N)                            }
```

c. A transfer out of the range of any DO loop is permissible at any time.

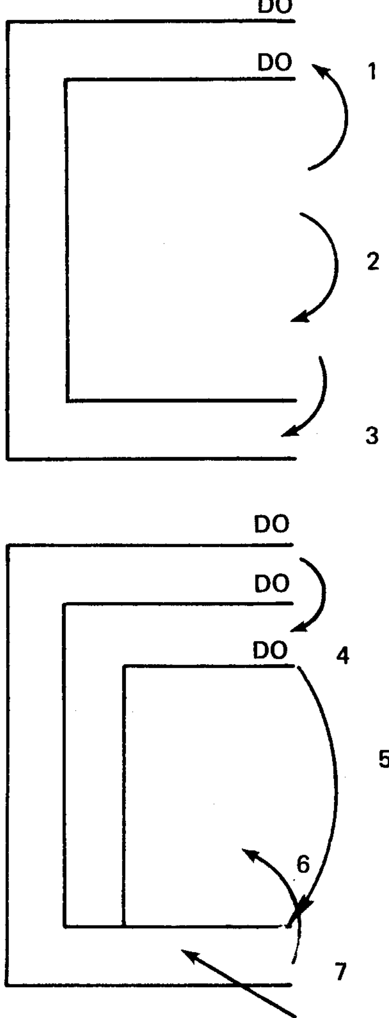
d. The extended range of a DO is defined as those statements in the program unit containing the DO statement that are executed between the transfer out of the *innermost* DO of a nest of DO's and the transfer back into the range of this innermost DO. The following restrictions apply:

Transfer into the range of a DO is permitted only if such a transfer is from the extended range of the DO.

The extended range of a DO statement must not contain another DO statement that has an extended range if the second DO is within the same program as the first.

The indexing parameters (i, m_1, m_2, m_3) cannot be changed in the extended range of the DO.

Note that a statement that is the end of the range of more than one DO statement is within the innermost DO. The statement label of such a 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.



In the preceding example, the transfers specified by the numbers 1, 2, and 3 are permissible, whereas those specified by 4, 5, 6, and 7 are not.

e. The indexing parameters (i, m_1, m_2, m_3) may be changed by statements outside the range of the DO statement only if no transfer is made back into the range of the DO statement that uses those parameters.

f. The last statement in the range of a DO loop must be an executable statement. It cannot be a GO TO statement of any form, or a PAUSE, STOP, RETURN, arithmetic IF statement, another DO statement, or a logical IF statement containing any of these forms.

g. The use of, and return from, a subprogram from within any DO loop in a nest of DO's is permitted.

3-4.8 CONTINUE STATEMENT. The CONTINUE statement is a dummy statement that does not produce any executable instructions. It can be inserted anywhere into a program. It simply indicates that the normal execution sequence continues with the statement following.

General Form:

```
      CONTINUE
```

The CONTINUE statement is principally used as the range limit of DO loops in which the last statement would otherwise be a GO TO, IF, PAUSE, STOP, or RETURN statement. It also serves as a transfer point for IF and GO TO statements within the DO loop that are intended to begin another repetition of the loop.

Example 1:

```
        .
        .
        .
      DO 30 I = 1, 20
7     IF (A(I)-B(I)) 5,30,30
5     A(I) = A(I) +1.0
      B(I) = B(I) -2.0
        .
        .
        .
      GO TO 7
30    CONTINUE
      C = A(3) + B(7)
        .
        .
        .
```

In example 1, the CONTINUE statement is used as the last statement in the range of the DO in order to avoid ending the DO loop with the statement GO TO 7.

Example 2:

```
        .
        .
        .
      DO 30 I=1,20
      IF (A(I)-B(I))5,40,40
5     A(I) = C(I)
      GO TO 30
40    A(I) = 0.0
30    CONTINUE
```

In example 2, the CONTINUE statement provides a branch point enabling the programmer to bypass the execution of statement 40.

3-4.9 CALL STATEMENT. The CALL statement is used only to call a SUBROUTINE subprogram.

General Form:

CALL name (a_1,a_2,a_3,...a_n)

where

name is the symbolic name of a SUBROUTINE subprogram.

a_1, a_2, a_3,...a_n are the actual arguments that are being supplied to the SUBROUTINE subprogram.

Example 1:

```
      CALL MATMP (X,5,40,Y,7,2)
      CALL QDRTI (X,Y,Z,ROOT1,ROOT2)
```

The call statement transfers control to the SUBROUTINE subprogram and replaces the dummy variables with the values of the actual arguments that appear in the CALL statement. The arguments in a CALL statement may be any of the following: any type of constant, any type of subscripted, or non-subscripted variable, any other kind of arithmetic expression, or a subprogram name (except that they may not be statement function names).

Note that the constants should not be used as parameters in a CALL statement if the subroutine is returning a value through that parameter.

Example 1:

```
            Calling              SUBROUTINE
            Program              Subprogram

               .                 SUBROUTINE JOE (A,B)
               .                 A = B + 10
        CALL JOE (5,6)           RETURN
               .                 END
               .
               .
  100   C = 5
               .
               .
               .
```

In this case the constant 5 in the calling program is replaced by the value of A as computed in the subroutine (A = B + 10). Subsequent execution of statement 100 in the calling program results in the variable C being assigned a value other than 5.

3-4.10 RETURN STATEMENT.

Form:

RETURN

This statement returns control from a subprogram to the calling program. Normally, the last statement executed in a subprogram is a RETURN statement. It need not be physically the last statement of the program. Any number of RETURN statements may be used. RETURN may appear only in subprograms.

3-4.11 PAUSE STATEMENT.

Forms:

PAUSE
PAUSE n

where n is a string of five or less octal digits.

This statement is included for compatability with other FORTRAN's. The octal digits or zero (first form) are typed on the teletypewriter and execution continues. This statement is sometimes useful in program debugging.

3-4.12 STOP STATEMENT.

Forms:

STOP
STOP n

where n is a string of five or less octal digits.

This statement terminates the program. It does not stop the machine and may be used to transfer control to RTM-I. The octal digits inform RTM-I of the status of the program execution. If the octal digits are not listed, a status word of all zeros is returned to the monitor. The use of this status word is explained in paragraph 6-15.

3-4.13 END STATEMENT.

Form:

END

The END statement is a non-executable statement that defines the end of a source program or source subprogram for the compiler. Physically, it must be the last statement of each program or subprogram. It may not have a statement number, and it may not be continued. If program execution reaches the END statement, the effect is exactly as if a STOP statement had been executed.

3-5 INPUT-OUTPUT.

The input/output (I/O) statements control the transmission of information between the computer and the I/O units.

I/O statements are classified as follows:

TRANSMISSION STATEMENTS. READ and WRITE are statements which specify transmission of information between computer memory and various input-output devices.

FORMAT STATEMENT. The FORMAT statement is a non-executable statement which specifies the conversions required between internal and external data forms.

AUXILIARY STATEMENTS. REWIND, BACKSPACE and END FILE provide positioning and file termination for magnetic tape and disc I/O.

3-5.1 TRANSMISSION STATEMENTS. In general, a transmission input-output statement provides:

a. Specification of the operation required; whether input or output and the particular device involved by using the verbs READ or WRITE.

b. Reference to a data format which specifies the sort of conversions required between internal and external data forms. The reference is either the number of a FORMAT statement or the identifier of an array which contains a data format.

c. A list of the variables the values of which are to be transmitted in the order in which the

information exists on the input medium or will exist on the output medium.

For example, the statement

```
WRITE(1,5)A,X,K
```

specifies that the values of A, X, and K, in that order, are to be written on logical I/O unit 1 according to the format given in the FORMAT statement numbered 5.

Transmission statements designate the logical input/output unit to be used by an expression. The correspondence between the logical unit and the actual I/O device is determined by the data supplied at system generation.

All FORTRAN input from or output to a given unit is with respect to a single sequential file. A single sequential file on a given unit has the following characteristics:

a. If the file contains one or more records, those records exist as a totally ordered set.

b. There exists a unique position of the file called its initial point. If a file contains no records, the unit is positioned (physically or logically) to write starting at the initial point. If the unit is positioned at the initial point and the file contains records, the first record of the file is defined as the next record. No record precedes the initial point.

c. If a unit is not positioned at the initial point of the file, there exists a unique preceding record and next record associated with that position.

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

e. When the next record is transmitted, the position of the unit is changed so the record just transmitted becomes the preceding record.

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 alphanumeric constants. The transfer of such a record requires that a FORMAT statement be referenced to supply the necessary positioning and conversion specifications. The number of records transferred by the execution of a formatted READ or WRITE is dependent upon the list and referenced format specification. 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.

INPUT/OUTPUT LISTS. The list of a transmission statement specifies the order of transmission of the variable values. During input, the new values of listed variables may be used in subscript or control expressions for variables appearing later in the list. For example

```
READ(2,3)L,A(L),B(L+1)
```

reads a new value of L and uses this value in the subscripts of A and B.

The transmission of array variables may be controlled by indexing similar to that used in the DO statement. The list of controlled variables, followed by the index control, is enclosed in parentheses and the whole acts as a single element of the list. For example

```
READ(7,23)(X(K),K=1,4)
```

is equivalent to

```
READ(7,23)X(1),X(2),X(3),X(4)
```

As in the DO statement the initial limit, and increment values may be given expressions, as in:

```
READ(4,2)N,(GAIN(K),K=1,M+N+1,M)
```

The indexing may be compounded as in the following

```
READ(2,13)((MASS(K,L),K=1,5),L=1,4)
```

This statement reads in the elements of array MASS in the order

```
MASS(1,1),MASS(2,1),...,MASS(5,1),MASS(1,2),...,MASS(5,4)
```

If an entire array is to be transmitted, the indexing may be omitted and only the array identifier written. The array is transmitted in order of increasing subscripts with the first subscript varying most rapidly. Thus, the example above can be written

```
READ(2,13)MASS
```

FORMATTED READ STATEMENT. The READ statement is used to transfer information from any input unit to the computer. Two forms of the formatted READ statement may be used, as follows:

```
READ (a,b) List
      or
READ (a,b)
```

where:

a is an unsigned integer constant or integer variable that specifies the logical unit number to be used for input data.

b is the statement number of the FORMAT statement describing the type of data conversion or the name of an array containing FORMAT data.

List is a list of variable names, separated by commas, for the input data.

The READ (a,b) List form is used to read a number of items (corresponding to the variable names in the list) from the file on unit a, using FORMAT statement b to specify the external representation of the data (see FORMAT Statement).

The List specifies the number of items to be read and the locations into which the items are to be placed.

If the formatted data to be read is punched onto paper tape, the following conventions apply. Each record terminates with a carriage return, transmitter off, rub out and line feed.

If the formatted data is punched on cards each card is one record.

For example, assume that an input card is punched as follows:

Columns	Contents
1-2	25
5-7	102
61-64	-101
70-71	10
80	5

If the following statements appear in the source program:

```
     READ (5,25) I,J,K,L,M,
25   FORMAT(I2,2x,I3,53x,I4,5x,I2,8x,I1)
```

the record is read and the program operates as though the following statements had been written:

```
I = 25
J = 102
K = -101
L = 10
M = 5
```

If the READ statement is executed again, I,J,K,L, and M will have new values, depending upon what is punched in the next card read.

Any number of quantities may appear in a single list.

If there are more quantities in an input record than there are items in the list, only the number of quantities equal to the number of items in the list are transmitted; remaining quantities are ignored. Thus, if a record contains three quantities and a list contains two, the third quantity is lost. Conversely, if a list contains more quantities than the number of quantities in an input record, succeeding input records are read until all the items specified in the list have been transmitted.

The READ (a,b) form may be used in conjunction with a FORMAT statement to read H-type alphanumeric data into an existing H-type field in core storage. The size of the data field determines the amount of data to be read. For example, the statements:

```
10   FORMAT(25HTHIS IS ALPHANUMERIC DATA)
          .
          .
          .
     READ (INPUT, 10)
```

cause the next 25 characters to be read from the file on the unit named INPUT and placed into the H-type alphanumeric field whose contents were:

THIS IS ALPHANUMERIC DATA

FORMATTED WRITE STATEMENT. The WRITE statement is used to transfer information from the computer to any of the output units. Two forms of the formatted WRITE statement may be used, as follows:

```
WRITE (a,b) List
      or
WRITE (a,b)
```

where:

a is an unsigned integer constant or integer variable that specifies the logical unit number to be used for output data.

b is the statement number of the FORMAT statement describing the data conversion or the name of an array containing FORMAT data. For unformatted transmission the format reference is omitted.

List is a list of variable names separated by commas for the output data.

The WRITE (a,b) List form of the WRITE statement is used to write the data specified in the list onto the file on unit a, using FORMAT statement b to specify the data format (see FORMAT Statement). The same conventions as described in the previous paragraph define an output record.

The WRITE (a,b) form is used to write alphanumeric data. The actual data to be written is specified within the FORMAT statement; therefore, an I/O list is not required. The following statements illustrate the use of this form:

```
25    FORMAT (24HWRITE ANY DATA IN H TYPE)
        .
        .
        .
      WRITE (1,25)
```

UNFORMATTED READ AND WRITE STATEMENTS. The READ and WRITE statements for unformatted I/O, i.e., I/O without data conversion, appear as:

```
READ (a) List
READ (a)
WRITE (a) List
```

where:

a is an unsigned integer constant or integer variable that specifies the logical unit number to be used.

List is a list of variable names, separated by commas.

The READ (a) List form is used to read a binary core-image record, without data conversion, into core storage from unit a. No FORMAT statement is required; the amount of data that is read corresponds to the number of list items. The total length of the list of variable names must not be longer than the logical record length. If the length of the list is equal to the logical record length, the entire record is read. If the length of the list is shorter than the logical record length, the unread items in the record are skipped.

The READ (a) form is used to skip a record on unit a.

The WRITE (a) List form is used to write a binary core-image record, without data conversion, on unit a.

3-5.2 FORMAT STATEMENT. All formatted input or output requires the use of a data format specifying the external format of the data and the type of conversion to be used. The data format is given in a FORMAT statement or as an alphanumeric string in a data array.

Form:

FORMAT(S_1,S_2,...,S_k)

where S is a data field specification.

NUMERICAL FIELDS. Conversion of numerical data may be one of five types:

a. type-D

   internal form - binary floating point (double precision)
   external form - decimal floating point (double precision)

b. type-E

   internal form - binary floating point
   external form - decimal floating point

c. type-F

   internal form - binary floating point
   external form - decimal fixed point

d. type-G

   internal form - binary floating point
   external form - decimal fixed point or floating point

e. type-I

   internal form - binary integer
   external form - decimal integer

These types of conversion are specified by the forms:

a. Dw.d
b. Ew.d
c. Fw.d
d. Gw.d
e. Iw

The letter D,E,F,G, or I designates the conversion type; w is an integer specifying the field width, which includes any preceding blanks, minus sign, decimal point, and exponent; d is an integer specifying the number of decimal places to the right of the decimal point. For example, the statement

FORMAT (15,F10.2,D25.8)

could be used to output the line

32   -17.60   .59625476D+03

on the output listing.

The type of conversion used must correspond to the type of the variable in the input/output list. I conversion is used for integer variables; E, F, or G conversion is used for real variables; and D conversion is used for double precision variables.

SCALE FACTORS. Scale factors may be specified for D, E, and F type conversions. A scale factor is written n P where P is the identifying character and n is a signed or unsigned integer specifying the scale factor.

For F type conversions the scale factor specifies a power of ten such that

external number = (internal number)*(power of ten)

For D and E type conversions the scale factor multiplies the number by a power of ten but the exponent is changed accordingly, leaving the number unchanged except in form. For example if the statement

FORMAT (F8.3,E16.5)

corresponds to the line

26.451   -0.41321E-01

then the statement

FORMAT (-1PF8.3,2PE16.5)

corresponds to the line

2.645   -41.32100E-03

When no scale factor is given, it is understood to be 0. However, once a scale factor is given, it holds for all following D, E, and F type conversions within the same format. The scale factor is reset to zero by giving a scale factor of zero.

Scale factors have no effect on I conversions.

I-CONVERSION (Iw). The specification Iw may be used to output a number in integer form; w positions are reserved for the number. It is output in this w-position field right-justified (that is, the units position is at the extreme right). If the number to be converted is greater than w positions, an error condition will exist. If the number is negative, an error condition exists if the converted number requires more than w-1 positions. A position must be reserved for the sign if negative values are output, but positive values do not require a position for the sign. If the number has less than w digits, the leftmost positions are filled with blanks. If the quantity is negative, the position preceding the leftmost digit contains a minus sign.

The following examples show how each of the quantities on the left is printed, according to the specification I3:

Internal Value	Printed
721	721
-721	***
-12	-12
8114	***
0	0
-5	-5
9	9

All error fields are filled in with asterisks.

F-CONVERSION (Fw.d). For F-type conversion, w is the total field length reserved and d is the number of places to the right of the decimal point (the fractional portion). For output, the total field length reserved must include sufficient positions for a sign, if any, a digit to the left of the decimal point, and a decimal point. The sign, if negative, is printed. In general w should be at least equal to d + 3 for output.

If insufficient positions are reserved by d, the fractional portion is truncated from the right. If excessive positions are reserved by d, zeros are filled in from the right to the extent of the specified precision. The integer portion of the number is handled in the same fashion as numbers converted by I-type conversion on input and output.

The following examples show how each of the quantities on the left is printed according to the specification F5.2:

Internal Value	Printed
12.17	12.17
-41.16	*****
-.2	-0.20
7.3542	7.35†
-1.	-1.00
9.03	9.03
187.64	*****

† Last two digits of accuracy lost due to insufficient specification.

NOTES

a. All error fields are filled with asterisks.

b. Numbers for F-conversion input need not have their decimal points appearing in the input field. If no decimal point appears, space need not be allocated for it. The decimal point will be supplied when the number is converted to an internal equivalent; the position of the decimal point will be determined by the format specification. However, if the decimal point does appear within the field and it is different from the format specification, this position overrides the position indicated in the format specification.

c. Fractional numbers for which F-type output conversion is specified are normally printed with a leading zero. If F-conversion is used and zero decimal width is specified (for example, F5.0), a fractional value is printed as a sign, a zero, and a decimal point. A zero value is printed with a zero preceding the decimal point.

d. F-conversion will accept input data in E-type format.

E-CONVERSION (Ew.d). For E-conversion, the fractional portion is again indicated by d. For output, the w includes the field d, a space for a sign, space for a digit preceding the decimal point, a decimal point, and four spaces for the exponent. Space must be reserved for each of these on output. An output error condition will result if $w \leqslant d+5$. For input, it is not necessary to reserve all of these positions. In general, w should be at least equal to d+7.

The exponent is a signed or unsigned one- or two-digit integer constant not greater than 38 and preceded by the letter E. Ten (10) raised to the power of the exponent is multiplied by the number to obtain its true internal value.

The following examples show how each of the quantities on the left is printed, according to the specification E9.3:

Internal Value	Printed
238.	0.238E 03
-.002	*********
.00000000004	0.400E-10
-21.0057	*********

If the last example above has been printed with a specification of E10.3, it would appear as:

-0.21 E 02

NOTES

a. All error fields are filled with asterisks.

b. For input, the start of the exponent field must be marked by an E, or, if that is omitted, by a + or - sign (not blank). Thus, E2, E+2, +2, +02, E02, and E+02 are all permissible exponent fields for input.

c. For input, the exponent field may be omitted entirely (i.e., E-conversion will accept input data in F-type format).

d. Numbers for E-conversion input need not have their decimal points appearing in the input field. If no decimal point appears, space need not be allocated for it. The decimal point will be supplied when the number is converted to an internal equivalent; the position of the decimal point will be determined by the format specification. However, if the decimal point does appear within the field and it is different from the format specification, this position overrides the position indicated in the format specification.

e. A leading zero is always printed to the left of the decimal point.

G-CONVERSION (Gw.d). The numeric field descriptor Gw.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
$0.1 \leqslant N < 1$	F(w-4).d,4X
$1 \leqslant N < 10$	F(w-4).(d-1),4X
.	.
.	.
.	.
$10^{d-2} \leqslant N < 10^{d-1}$	F(w-4).1,4X
$10^{d-1} \leqslant N < 10^{d}$	F(w-4).0,4X
Otherwise	sEw.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.

D-CONVERSION (Dw.d). The numeric field descriptor Dw.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 will replace the character E in the exponent.

LOGICAL FIELDS. Logical data can be transmitted in a manner similar to numeric data by use of the form:

```
Lw
```

Where L is the control character and w is an integer specifying the field width.

Data is transmitted as the value of a logical variable in the input/output list.

On input, the first non-blank character in the data field must be T or F, and the value of the logical variable will be stored as true or false, respectively. The remainder of the field is ignored. If the entire data field is blank, a value of false will be stored.

On output, W-1 blanks followed by T or F will be output if the value of the logical variable is true or false, respectively.

ALPHANUMERIC FIELDS. Alphanumeric data can be transmitted in a manner similar to numeric data by use of the form Aw; A is the control character and w is the number of characters in the field. The alphanumeric characters are transmitted as the value of a variable in an input/output list. The variable may be of any type. For example, the sequence

```
      READ (2,5)V
5     FORMAT (A2)
```

causes two characters to be read and placed in memory as the value of the variable V.

The number of characters transmitted is limited by the maximum number of characters which can be stored in the space allotted for the variable. If w exceeds the available space, leading characters are lost on input and replaced with blanks on output. When w is less than the available space, blanks are filled in after the given characters until the maximum is reached.

ALPHANUMERIC FORMAT FIELDS. An alphanumeric field may be specified within a format by preceding the alphanumeric string by the form n H. H is the control character and n is the number of characters in the string, counting blanks. For example, the statement

```
FORMAT (17H PROGRAM COMPLETE)
```

can be used to output

```
PROGRAM COMPLETE
```

on the output listing.

On input the external characters are stored in the format itself.

MIXED FIELDS. An alphanumeric format field may be placed among other fields of the format. For example, the statement

```
FORMAT (I5,8H FORCE = F10.5)
```

can be used to output the line

```
22 FORCE = 17.68901
```

The separating comma may be omitted after an alphanumeric format field.

BLANK OR SKIP FIELDS. Blanks may be introduced into an output record or characters skipped in an input record by use of the specification n X. The control character is X, n is the number of blanks or characters skipped. n must be greater than zero. For example, the statement

```
FORMAT (5H STEPI5,10X,3HY = F7.3)
```

may be used to output the line

```
STEP   28          Y = -3.872
```

where ten blanks separate the two quantities.

REPETITION OF FIELD SPECIFICATIONS. Repetition of a field specification may be performed by preceding the control character D, E, F, G, I, or A by an unsigned integer giving the number of repetitions desired. For example,

```
FORMAT (2E12.4,3I5)
```

is equivalent to

```
FORMAT (E12.4,E12.4,I5,I5,I5)
```

REPETITION OF GROUPS. A group of field specifications may be repeated by enclosing the group in parentheses and preceding the whole with the repetition number. For example,

```
FORMAT (2I8,2(E15.5,2F8.3))
```

is equivalent to

```
FORMAT (2I8,E15.5,2F8.3,E15.5,2F8.3)
```

Up to two levels of parentheses are allowed in group repetition, in addition to those enclosing the entire format. For example, the statement

```
FORMAT (I5,2(E15.5,2(F8.3)))
```

exhibits the maximum level of nesting. The outermost parenthesis pair is termed level zero, the next level one, and the innermost level two.

COMPLEX FIELDS. Complex quantities are transmitted as two independent real quantities. The format specification is given as two successive real specifications or one repeated real specification. For instance the statement

```
FORMAT (2E15.4,2(F8.3,F8.5))
```

could be used in the transmission of three complex quantities.

MULTIPLE RECORD FORMATS. To handle a group of input/output records where different records have different field specifications, a slash "/" is used to indicate a new record. For example, the statement

```
FORMAT (3I8/I5,2F8.4)
```

is equivalent to

```
FORMAT (3I8)
```

for the first record and

```
FORMAT (I5,2F8.4)
```

for the second record.

The separating comma may be omitted when a slash is used.

Blank records may be written on output or records skipped on input by using consecutive slashes.

Both the slash and the closing parentheses at the end of the format indicate the termination of a record. If the list of an input/output statement dictates that transmission of data is to continue after the closing parentheses of the format is reached, the format is repeated from the last open parentheses of level one or zero. Thus, the statement

```
FORMAT (F7.2,2(E15.5,E15.4),I7)
```

causes the format

```
F7.2,2(E15.5,E15.4),I7
```

to be used on succeeding records.

As a further example, consider

```
FORMAT (F7.2/2(E15.5,E15.4),I7)
```

The first record has the format

```
F7.2
```

successive records have the format

```
2(E15.5,E15.4),I7
```

FORMATS STORED AS DATA. The alphanumeric string comprising a format specification may be stored as the values of an array. Input/output statements may reference the format by giving the array name rather than the statement number of a FORMAT statement. The stored format has the same form as a FORMAT statement excluding the word "FORMAT". The enclosing parentheses are included.

As an example, consider the sequence

```
      DIMENSION SKELETON(6)
      READ (5,1),(SKELETON(I),I=1,6)
1     FORMAT (6A2)
      READ (5,SKELETON),K,X
```

The first READ statement enters the character string into the array SKELETON. In the second READ statement, SKELETON is referenced as the format governing conversion of K and X.

CARRIAGE CONTROL. If a record is to be printed, the first character in that record is used for carriage control. Normally, the character is specified at the beginning of the

format specification for the unit record as 1Hx, where x is a blank, 0, 1, or +. This character is not printed; it only controls character spacing as follows:

blank	A single space before the record is printed.
0	A double space before the record is printed.
1	A skip to form top before the record is printed.
+	All spacing or skipping to be suppressed before the record is printed.

3-5.3 AUXILIARY INPUT/OUTPUT STATEMENTS. There are three types of auxiliary input/output statements. Use of these statements is limited to magnetic tape and disc applications. They are REWIND, BACKSPACE, and ENDFILE.

REWIND STATEMENT.

Form:

```
REWIND u
```

where u is an I/O unit designation.

This statement directs the I/O unit designated to reposition to the first record of the first file. The unit designation is given as an integer expression.

Examples:

```
REWIND 7
REWIND K-MIO
```

BACKSPACE STATEMENT.

Form:

```
BACKSPACE u
```

where u is an I/O unit designation.

This statement directs the I/O unit designated to backspace one record. The unit designation is given as an integer expression. This is applicable to magnetic tape records only.

Examples:

```
BACKSPACE 7
BACKSPACE N(2)
```

END FILE STATEMENT.

Form:

```
END FILE u
```

where u is an I/O unit designation.

The statement directs the I/O unit designated to terminate the file being written.

The unit designation is given as an integer expression.

Examples:

```
END FILE 4
END FILE T(K)
```

3-5.4 LOGICAL UNITS. The logical unit assignments assumed by FORTRAN are

1	teletypewriter output
2	teletypewriter input
3	high speed paper tape punch output
4	high speed paper tape reader input
5	card reader input
6	line printer output
7	magnetic tape drive one input/output
8	disc foreground work area input/output (BATCH only) or entire disc (SPEX only)
9	magnetic tape drive two input/output
10	magnetic tape drive three input/output
11	disc background work area input/output (BATCH only)

The first six assignments are identical to those assumed by the system software. The last five are included to allow the FORTRAN programmer to specify which magnetic tape drive or which disc work area is to be used. The FORTRAN

I/O programs will convert I/O requests to these logical units to I/O requests to the assumed system logical I/O devices.

Up to four files may be written in the disc background area. This is discussed further in Sections IV and VI. Thus the END FILE instruction may reference logical units 7, 9, 10, and 11. REWIND may be used with logical units 7, 8, 9, 10, and 11. When referencing units 8 and 11, it causes reading or writing to commence at the initial sector of that disc area. BACKSPACE may be used with the magnetic tape drives only, logical units 7, 9, and 10.

3-6 DECLARATIONS.

Declarations are used to supply descriptive information about the program rather than to specify computation or other action. This descriptive information primarily concerns the interpretation of source program identifiers and object program storage allocation.

The following declaration statements must all appear in the program prior to any non-declarative statements, arithmetic function definition statements, or DATA statements:

DIMENSION statement
EXTERNAL statement
COMMON statement
EQUIVALENCE statement
Type declaration statements.

DATA statements must appear before arithmetic function definition statements. Both must appear in the program prior to any non-declarative statements.

3-6.1 DIMENSION STATEMENT. The DIMENSION statement is used to declare identifiers to be array identifiers and to specify the number and bounds of the array subscripts. The information supplied in a DIMENSION statement is required for the allocation of memory for arrays. Any number of arrays may be declared in a single DIMENSION statement.

Form:

DIMENSION $S_1, S_2, \ldots, S_k$

where S is an array specification.

Each array variable appearing in the program must represent an element of an array declared in a DIMENSION statement, unless the dimension information is given in another statement. When the dimension information is provided in a COMMON or type declaration statement, it may not appear in a DIMENSION statement.

Each array specification gives the array identifier and the maximum values each of its subscripts may assume, thus:

identifier($max_1, max_2, \ldots, max_n$)

The maxima must be integers. An array may have any number of dimensions.

For example, the statement

```
DIMENSION EDGE (10,8)
```

specifies EDGE to be a two dimensional array the first subscript of which may vary from 1 to 10 inclusive, and the second from 1 to 8 inclusive.

Example:

```
DIMENSION PLACE (3,3,3),HI(2,4),K(256)
```

Arrays may also be declared in the COMMON or type declaration statements in the same way:

```
COMMON X(10,4),Y,Z
INTEGER A(7,32),B
DOUBLE PRECISION K(6,10)
LOGICAL TREE(2,2,2,2,2)
```

Note that each element of an integer or logical array requires one word of storage. Each element of a real array requires two, double precision three and complex four.

Within a subprogram, array specifications may use integer variables instead of constants, provided that the array name and variable dimensions are dummy arguments of the subprogram. The actual array name and values of the dummy variables are given by the calling program when the subprogram is called.

Example:

```
DIMENSION BETA(L,M,G),B(20)
```

The identifiers BETA, L, M, and G must all be dummy arguments.

3-6.2 TYPE DECLARATIONS. The type statements INTEGER, REAL, DOUBLE PRECISION, COMPLEX, and LOGICAL are used to explicitly specify the type of the identifiers appearing in the program. An identifier may appear in only one type statement. Type statements may be used to declare arrays that are not dimensioned in DIMENSION or COMMON statements.

Identifiers whose type is not explicitly declared are implicitly typed as follows:

a. Identifiers beginning with I, J, K, L, M, or N are assigned integer type.

b. Identifiers not assigned integer type are assigned real type.

INTEGER STATEMENT.

Form:

INTEGER identifier,identifier,...,identifier

This statement declares the listed identifiers to be of integer type.

Example:

INTEGER ALPHA,P,LAMBDA

Notice that LAMBDA need not appear, since it would be implicitly declared integer.

REAL STATEMENT.

Form:

REAL identifier,identifier,...,identifier

This statement declares the listed identifiers to be of real type.

Example:

REAL LOGX,KAPPA,MASS(10,4)

MASS is also declared to be an array in this example.

DOUBLE PRECISION STATEMENT.

Form:

DOUBLE PRECISION identifier,identifier,...,identifier

This statement declares the listed identifiers to be of double precision type.

Example:

DOUBLE PRECISION RATE,Y,FLOW

COMPLEX STATEMENT.

Form:

COMPLEX identifier,identifier,...,identifier

This statement declares the listed identifiers to be of complex type.

Example:

COMPLEX ZETA,W,ROOT

LOGICAL STATEMENT.

Form:

LOGICAL identifier,identifier,...,identifier

This statement declares the listed identifiers to be of logical type.

Example:

LOGICAL BOOL,P,Q,ANSWER

3-6.3 EXTERNAL STATEMENT.

Form:

EXTERNAL identifier,identifier,...,identifier

This statement declares the listed identifiers to be subprogram names. Any subprogram name given as an argument to another subprogram must appear in an EXTERNAL declaration in the calling program.

Example:

EXTERNAL SIN,COS

3-6.4 COMMON STATEMENT.

Form:

COMMON block-list

The COMMON statement specifies that certain variables or arrays are to be stored in an area also available to other programs. By means of COMMON statements, a program and its subprograms may share a common storage area.

The common area may be divided into separate blocks identified by block names. A block is specified thus:

/identifier/identifier,identifier,...,identifier

The identifier enclosed in slashes is the block name. The identifiers which follow are the names of the variables or arrays assigned to the block. These elements are placed in the block in the order in which they appear in the block specification.

The block list of the COMMON statement consists of a sequence of one or more block specifications. For example, the statement

```
COMMON /R/X,Y,T/C/U,V,W,Z
```

indicates that the elements X, Y and T, in that order, are to be placed in block R and that U, V, W and Z are to be placed in block C.

Block entries concatenate throughout the program, beginning with the first COMMON statement. For example, the statements

```
COMMON /D/ALPHA/R/A,B/C/S
COMMON /C/X,Y/R/U,V,W
```

have the same effect as the statement

```
COMMON /D/ALPHA/R/A,B,U,V,W/C/S,X,Y
```

One block of common storage may be left unlabeled; blank common. Blank common is indicated by two consecutive slashes. For instance,

```
COMMON /R/X,Y//B,C,D
```

indicates that B, C, and D are placed in blank common.

The slashes may be omitted when blank common is the first block of the statement.

```
COMMON B,C,D
```

Array names appearing in COMMON statements may have dimension information appended, as in a DIMENSION statement. For example,

```
COMMON ALPHA, T(15,10,5),GAMMA
```

specifies the dimensions of the array T while entering T in blank common.

NOTES

a. Dummy arguments for SUBROUTINE or FUNCTION statements cannot appear in COMMON statements.

b. A single COMMON statement may contain variable names, array names, and dimensioned array names. For example, the following are valid:

```
DIMENSION B(5,15)
COMMON A, B, C(9,9,9)
```

Variables or arrays that appear in the main program or a subprogram may be made to share the same storage locations with variables or arrays of the same type and size in other subprograms, by use of the COMMON statement. For example, if one program contains the statement:

```
COMMON TABLE,A,B,C
```

and a second program contains the statement:

```
COMMON LIST
```

the variable names TABLE and LIST refer to the same storage locations (assuming the data associated with the names TABLE and LIST are equal in length and type).

If the main program contains the statement:

```
COMMON A,B,C
```

and a subprogram contains the statement:

```
COMMON X,Y,Z,XX,YY,ZZ
```

and A, B, and C are equal in length to X, Y, and Z, respectively, then A and X refer to the same storage locations, as do B and Y, and C and Z.

Within a specific program or subprogram, variables and arrays are assigned storage locations in the sequence in which their names appear in a COMMON statement. Subsequent sequential storage assignments within the same program or subprogram are made with additional COMMON statements.

A dummy variable can be used in a COMMON statement to establish shared locations for variables that would otherwise

occupy different locations. For example, the variable Z of the previous example will share storage with S if the following statement is used:

```
COMMON Q,R,S
```

where Q and R are dummy names that are not used elsewhere in the program.

Redundant COMMON entries are not allowed. For example, the following is invalid:

```
COMMON A, B, C, A
```

The assembler directive COMM can be used to access blank common from an assembly language program. Any data stored in the region defined by COMM will share storage with FORTRAN variables in blank common.

3-6.5 EQUIVALENCE STATEMENT. The equivalence statement allows more than one identifier to represent the same quantity.

Form:

EQUIVALENCE (R_1,R_2,...),(R_N,R_{N+1},...),...

The references of an EQUIVALENCE statement may be variables or array identifiers or array element references. The subscripts of an array element must be integer constants. The number of subscripts must be equal to the array dimension or must be one.

Example:

```
EQUIVALENCE (A,B,C(3)),(T(4),S(1,1,2))
```

The inclusion of two or more references in a parenthesis pair indicates that the quantities referenced are to share the same memory locations. For example

```
EQUIVALENCE (RED,BLUE)
```

specifies that the quantities RED and BLUE are stored in the same locations.

When no array subscript is given, it is taken to be 1, thus

```
EQUIVALENCE (X,Y)
```

is the same as

```
EQUIVALENCE (X,Y(1))
```

Elements of multiply dimensional arrays may be referenced with a single subscript by use of the element successor function. For example in the three-dimensional array specified by

DIMENSION ALPHA (N_1,N_2,N_3)

the position of element ALPHA (K_1,K_2,K_3) is given by

$$\text{element position} = (K_3\text{-}1)*N_1*N_2+(K_2\text{-}1)*N_1+K_1$$

Thus the sequence

```
DIMENSION BETA(4),ALPHA(2,3,4)
EQUIVALENCE (BETA(2),ALPHA(8))
```

specifies that BETA(2) and ALPHA(2,1,2) are stored in the same place.

NOTE

Conversion to single subscripts is permitted only in EQUIVALENCE statements.

Since the entire arrays are shifted to satisfy the equivalence, only the relative positions of the references are important. In the example above

```
   EQUIVALENCE (BETA(1),ALPHA(7))
or EQUIVALENCE (BETA,ALPHA(7))
```

will do as well.

Note that the relation of equivalence is transitive, e.g., the two statements

```
EQUIVALENCE (A,B),(B,C)
EQUIVALENCE (A,B,C)
```

have the same effect.

3-6.6 EQUIVALENCE AND COMMON. Identifiers may appear in both COMMON and EQUIVALENCE statements provided the following rules are observed.

No two quantities in common may be set equivalent to one another.

Quantities placed in a common block by means of equivalences may cause the *end* of the common block to be extended. For example, the statements

```
COMMON /R/X,Y,Z
DIMENSION A(4)
EQUIVALENCE (A,Y)
```

cause the common block R to extend from X to A(4), arranged as follows:

```
X
Y A(1)
Z A(2)
  A(3)
  A(4)
```

Equivalence which cause extension of the *start* of a common block are not allowed. For example, the sequence

```
COMMON /R/X,Y,Z
DIMENSION A(4)
EQUIVALENCE (X,A(3))
```

is not permitted since it requires block R to be arranged

```
  A(1)
  A(2)
X A(3)
Y A(4)
Z
```

A(1) and A(2) extend the start of block R.

3-6.7 SUBPROGRAM DEFINITIONS. FORTRAN subprograms may be internal or external.

Internal subprograms are defined within the program which calls them. They are defined within a single statement; the arithmetic function definition statement. Internal subprograms are defined and may be used only within the program containing the definition.

External subprograms are defined separately from (external to) the program which calls them and are complete programs conforming to all the rules of FORTRAN programs. They are compiled independently.

Two types of external subprograms may be defined: FUNCTION subprograms and SUBROUTINE subprograms. The use of the declarations FUNCTION and SUBROUTINE in the definition of these subprograms is described below.

Intrinsic functions are external subprograms which are predefined as part of the FORTRAN language. The function identifiers, definitions, and types are given in paragraph 3-7. These definitions are overridden by using the function identifier in any context other than a function reference.

Any subprogram, internal or external may call other subprograms; however, recursion is not allowed.

DUMMY IDENTIFIERS.

Subprogram definition statements declare certain identifiers to be dummies representing the arguments of the subprogram. They are used as ordinary identifiers within the subprogram definition and indicate what sort of arguments may appear and how the arguments are used. The dummy identifiers are replaced by the actual arguments when the subprogram is executed.

Dummy identifiers may not appear in COMMON, EQUIVALENCE, or DATA statements.

ARITHMETIC FUNCTION DEFINITION STATEMENT.

Form:

```
identifier(identifier,identifier,...)=expression
```

This statement defines an internal subprogram. The entire definition is contained in the single statement. The first identifier is the name of the subprogram being defined.

Arithmetic function subprograms are functions; they are single-valued and must have at least one argument. The type of the function is determined by the type of the function identifier.

The identifiers enclosed in parentheses represent the arguments of the function. These are dummy identifiers and have meaning and must be unique only within the statement. They may be identical to identifiers of the same type appearing elsewhere in the program. These identifiers must agree in order, number, and type with the actual arguments given at execution time.

The use of an argument in the defining expression is specified by the use of its dummy identifier. Expressions are the only permissible arguments of internal functions; hence the dummy identifiers may appear only as scalar variables in the defining expression. They may not appear as array identifiers.

Identifiers appearing in the defining expression which do not represent arguments are treated as ordinary variables.

The defining expression may include references to external functions or other previously defined internal functions.

The defining expression may not contain array element references.

All arithmetic function statements must precede the first executable statement of the program.

Examples:

```
SSQR(K) = K*(K+1)*(2K+1)/6
NOR(T,S) = .NOT.(T.OR.S)
ACOSH(X) = (EXP(X/A)+EXP(-X/A))/2
```

In the last example above, X is a dummy identifier and A is an ordinary identifier. At execution the function is evaluated using the current value of the quantity represented by A.

FUNCTION SUBPROGRAMS.

A FUNCTION subprogram is a function; single-valued and referenced as a basic element in an expression. A FUNCTION subprogram begins with a FUNCTION declaration and returns control to the calling program by means of one or more RETURN statements.

FUNCTION STATEMENT.

Forms:

```
FUNCTION identifier(identifier,identifier,...)
t FUNCTION identifier(identifier,identifier,...)
```

This statement declares the program which follows to be a function subprogram. The first identifier is the name of the function being defined. This identifier must appear as a scalar variable and be assigned a value during execution of the subprogram. This value is the function value.

Example:

```
FUNCTION ROOT (A,B,C)
ROOT = (-B+SQRT(B**2-4.0*A*C))/(2.0*A)
RETURN
END
```

Identifiers appearing on the list enclosed in parentheses are dummy identifiers representing the function arguments. They must agree in number, order, and type with the actual arguments given at run time. FUNCTION subprogram arguments may be expressions, array names or subprogram names. Dummy identifiers may appear in the subprogram as scalar identifiers, array identifiers or subprogram identifiers.

Dummy identifiers representing array names must appear within the subprogram DIMENSION, or type statements giving dimension information. Dimensions given as constants must equal the dimensions of the actual arrays given at run time. In a DIMENSION or type statement, dummy identifiers may be used to specify variable dimensions for array name arguments. For example, in the statement sequence

```
FUNCTION TABLE (A,M,N,B,X,Y)
     .
     .
     .
DIMENSION A(M,N),B(10),C(50)
     .
     .
     .
```

the dimensions of array A are specified by the dummies M,N and the dimension of array B is given as a constant. The values given for M,N at run time must be those of the actual array given for A.

Dummy dimensions may be given only for dummy arrays. In the example above the array C must be given absolute dimensions, since C is not a dummy identifier.

The type of the function is the type of the identifier which names the function. The identifier may be typed implicitly integer or real by use of the first form. The identifier is explicitly typed by using the second form with t replaced with INTEGER, REAL, DOUBLE PRECISION, COMPLEX, or LOGICAL.

Examples:

```
FUNCTION MAY(RANGE,XP,YP,ZP)
REAL FUNCTION COT(ARG)
COMPLEX FUNCTION HPME(S,N)
```

3-6.8 SUBROUTINE SUBPROGRAMS. A SUBROUTINE subprogram is not a function; it may be multi-valued and can be referred to only by a CALL statement. A SUBROUTINE subprogram begins with a SUBROUTINE declaration and returns control to the calling program by means of one or more RETURN statements.

SUBROUTINE STATEMENT.

Forms:

```
SUBROUTINE identifier
SUBROUTINE identifier(identifier,identifier,...)
```

This statement declares the program which follows to be a SUBROUTINE subprogram. The first identifier is the subroutine name. The identifiers in the list enclosed in parentheses are dummy identifiers representing the arguments of the subprogram. These identifiers must agree in number, order, and type with the actual arguments given to the subprogram at run time.

SUBROUTINE subprograms may have expressions, alphanumeric strings, array names, and subprogram names

as arguments. The dummy identifiers may appear as scalar, array, or subprogram identifiers within the subprogram.

Dummy identifiers which represent array names must be dimensioned within the subprogram by a DIMENSION, or type statement. As in the case of a FUNCTION subprogram, either constants or dummy identifiers may be used to specify dimensions in a DIMENSION or type statement.

A SUBROUTINE subprogram may use one or more of its dummy identifiers to represent results. The subprogram name is not used for return of results.

A SUBROUTINE subprogram need not have any arguments at all.

Examples:

```
SUBROUTINE EXIT
SUBROUTINE FACTOR (COEF,N,ROOTS)
SUBROUTINE RESIDU (NUM,D,DEN,M,RES)
```

3-6.9 DATA SPECIFICATION. The data specification statements DATA and BLOCK DATA are used to specify initial values for variables. These values are compiled into the object program. They become the values assumed by the variables when execution begins.

DATA STATEMENT.

Form:

```
DATA v/d/,v/d/,...
```

where v is a variable list and d is a data list.

The variable lists in a DATA statement consist of scalar variables and array elements separated by commas.

Variables in common may appear in the lists only if the DATA statement occurs in a BLOCK DATA subprogram.

The data items of each data list must correspond in total storage units with the variables of each variable list. Each data item specifies the value given to its corresponding variable.

Data items may be numerical constants or alphanumeric strings. For example

```
DATA ALPHA,BETA/5,16.E-2/
```

specifies the value 5 for ALPHA and the value .16 for BETA.

Alphanumeric data is packed into words two per word as in the case of A conversion. Excess characters are not permitted.

Example:

```
DATA NOTE/2HNO/
```

Any data item may be preceded by an integer and an asterisk. The integer indicates the number of times the item is to be repeated. For example

```
DATA A(1),A(2),A(3),A(4),A(5)/61E2,4*32E1/
```

specifies 5 values for the array A; the value 6100 for A(1) and the value 320 for A(2) through A(5).

BLOCK DATA STATEMENT.

Form:

```
BLOCK DATA
```

This statement declares the program which follows to be a data specification subprogram. Data specification for variables in common blocks require the use of a BLOCK DATA subprogram.

The first statement of the subprogram must be the BLOCK DATA statement. The subprogram may contain only the declarative statements associated with the data being defined.

Example:

```
BLOCK DATA
COMMON /R/X,Y/C/Z,W,V
DIMENSION Y(3)
COMPLEX Z
DOUBLE PRECISION X
DATA Y(1),Y(2),Y(3) /1E-1,2*3E2/
DATA X,Z/11.877D0,(-1.41421,1.41421)/
END
```

Data may be entered into more than one block of common in one subprogram. However, any common block mentioned must be listed in full. In the example above, W and V are listed in block C although no data values are defined for them.

3-7 FORTRAN OUTPUT LISTING.

3-7.1 LISTING ELEMENTS. The output listing of the FORTRAN compiler consists of the following elements (if required).

a. Source Program with error messages (if any)

b. Common Allocation – Blank and/or Labeled

c. Required Subroutines

d. Equivalence Allocation

e. Program Allocation

Program allocation lists all variables not appearing in COMMON or EQUIVALENCE statements. The listing is produced during first pass execution. Except for error comments no printed output is produced in pass two.

3-7.2 STATEMENT ERROR DIAGNOSTICS. During pass one, statements which violate the syntactic or semantic rules of the language are discarded and an error indication is output on the listing. Compilation proceeds as if the statement was never encountered. The statement label, if any, remains defined with the error statement executing as a CONTINUE.

One character of the statement is marked with an arrow (↑) output directly beneath the erroneous character, for example:

```
ZETA = X + Y * - A
               ↑
```

The character "-" is marked as an error.

In the case of a syntax error, the marker character itself was unacceptable, as in the example above. In the case of a semantic error, an identifier or other construct is in error, the mark indicating the last character of the construct. For example, in the line:

```
COMMON ALPHA,BETA,ALPHA,GAMMA
                      ↑
```

The mark indicates that the identifier ALPHA is misused.

The compiler attempts all interpretations of statement type before discarding a statement. The marked position indicates the greatest amount of correct information found under the most logical assumption of statement type.

A comment specifying the reason for the failure is output directly after the marked line.

The possible comments are:

SYNTAX. Erroneous punctuation or illegally constructed arithmetic expression.

NUMBER. A constant or label is too large or is incorrectly constructed.

ID CONFLICT. The identifier marked is being used in a context which contradicts a previous explicit or implicit declaration.

TYPE CONFLICT. The identifier or expression marked is in conflict with another identifier or expression.

MODE. The identifier or expression marked has a type in conflict with the context.

SUBSCRIPTS. The number of subscript expressions used in an array variable does not equal the number declared for the array.

ALLOCATION. A non-dummy variable has been given as an adjustable dimension, or a variable has been placed in COMMON more than once, or a dummy variable appears in a COMMON or EQUIVALENCE statement.

ORDER. The statement appears in the program at a point in violation to the stated rules governing the order or appearance of statements in the program.

MISSING LABEL. The statement must have a label in order to be reached or referenced.

DATA COUNT. The number of items in the data list of a DATA statement is not equal to the number of items in the variable list.

BLOCK DATA. An executable statement appears in a BLOCK DATA subprogram.

OVERFLOW. The statement caused the compiler capacity to be exceeded. Compilation does *not* continue.

During pass two the following error comments may occur INCORRECT FORMAT – tape not identified as a Fortran pass one output tape.

OVERFLOW – the program exceeds the capacity of the second pass.

INVALID CODE – the tape contains a code not recognized by the second pass.

3-7.3 PROGRAM ERROR DIAGNOSTICS. The comment LABELING ERRORS is output at the end of compilation to indicate any labeling or DO loop structure errors. The comment is followed by a list of statement numbers which may fall into any one of the following categories:

The statement number has been used within the program but has never appeared as the number of a statement.

The statement number appeared as the number of more than one statement.

The statement number is the number of a statement which closes the range of a DO statement, and:

The closing statement was never reached; or

The range ended with a transfer statement; or
The loop was illegally nested.

The label list is followed by the title ALLOCATION ERRORS and a list of identifiers which cannot be allocated memory. This may be because of violation of the rules of common or equivalence, because of multiple inconsistent allocations, or because no classification was ever established for the identifier.

3-7.4 PASS TWO ERROR DIAGNOSTICS

3-8 OBJECT PAPER TAPE FORMATS.

Programs which are compiled by Model 980 FORTRAN may be placed in two groups. Function subprograms, subroutine subprograms, and main programs are part of group 1, and block data programs make up group 2.

A different object output format is used for the two groups.

3-8.1 GROUP 1 OUTPUT FORMAT. The object paper tape produced by the FORTRAN second pass, for group 1 programs has the same format as assembler object output (see 2-9.1). All FORTRAN compliations are relocatable.

The PROGRAM ID is taken from the name of the Function or Subroutine-subprogram. The compiler allows six-character names for these subprograms. When the name is less than six characters long, the rightmost vacant characters are replaced with spaces. Thus a FORTRAN program heading SUBROUTINE SORT (A,N) will yield a subprogram named SORTΔΔ where Δ signifies space or blank. When the program name exceeds six characters, only the first six are used to identify the program; those programs which are not identified as Functions or Subroutines are automatically named ΔMAINΔ. Since the programmer can not generate a name with a leading space, this name is unique.

Labeled common names are derived in the same way. That common area which the programmer does not name is named ΔBLANK by the compiler.

3-8.2 GROUP 2 OUTPUT FORMAT. The object paper tape produced by FORTRAN for group 2 programs has the format shown in table 3-1.

TABLE 3-1

GROUP 2 OUTPUT FORMAT

Section	Field	Size
HEADER	PROGRAM NAME	12 Frames
	ZERO	4 Frames
	FORMAT CODE	4 Frames
	COMMON COUNT	4 Frames
	STOP CODE	4 Frames
COMMON REFS	COMMON SYMBOL 1	12 Frames
	COMMON LENGTH 1	4 Frames
	COMMON SYMBOL 2	12 Frames
	COMMON LENGTH 2	4 Frames
	⋮	
	CHECK SUM	4 Frames
	STOP CODE	4 Frames
DATA BLOCK 1	COMMON NUMBER	4 Frames
	RELATIVE LOCATION	4 Frames
	DATA COUNT	4 Frames
	DATA WORD 1	4 Frames
	DATA WORD 2	4 Frames
	⋮	
	CHECK SUM	4 Frames
	STOP CODE	4 Frames
DATA BLOCK	COMMON NUMBER	4 Frames
	RELATIVE LOCATION	4 Frames
	⋮	
END BLOCK	MINUS ONE	4 Frames
	STOP CODE	4 Frames

The program name is always Δ BLOCK. The format code is always 5.

All data on the object tape is punched as 16-bit words; four frames per word. The four low order channels of each frame contain one hexadecimal digit. Channel 5 is punched to produce an odd parity tape. The following table contains the hexadecimal digits and the hexadecimal codes corresponding to the eight channels of the tape.

Program Hex Digit	Object Hex Digit
0	10
1	01
2	02
3	13
4	04
5	15
6	16
7	07
8	08
9	19
A	1A
B	0B
C	1C
D	0D
E	0E
F	1F

Check sums are the negative of all data preceding the check sum and following the last check sum (or start of the tape). Thus the sum of all data after one check sum up to and including the next check sum is always zero.

The stop code is the hexadecimal word 0030. When reading the tape with the teletypewriter reader, the reader will stop after reading this code.

The data in each data block will be loaded starting at the relative location noted in the specified common block. Since the variables to be initialized in each common block are not necessarily contiguous, several data blocks may be required to initialize one common block.

3-9 LIBRARY SUBROUTINES.

3-9.1 ARITHMETIC CONVENTIONS. A number of library subroutines used by the FORTRAN compiler are available for incorporation into assembly language programs. These subroutines include those for the performance of arithmetic.

At the completion of any arithmetic operation on real, double precision, or complex data, the results will be located in the *floating point accumulator* or *complex accumulator*. Likewise input to arithmetic routines is often assumed to be in one or the other of these pseudo-registers. They are actually memory locations within the floating point arithmetic package. They are symbolically identified by the names ACC$ and EXP$ for the mantissa and exponent of real and double precision numbers and CACC$ and CACC2$ for the real and imaginary portions of complex numbers. The location MODE$ is used to set the mode of the floating point arithmetic as described in paragraph 3-9.5.

3-9.2 POWER ROUTINES. A series of power routines are available for computing

```
BASE ** EXP
```

The mode of the result will be the same as the mode of the base. The result will be returned in the appropriate hardware or software register as below.

Base	Result in
Integer	A-Register
Real	ACC$, EXP$
Double Precision	ACC$, EXP$
Complex	CACC$, CACC2$

The calling sequence is

```
BRL     *$+1
DATA    Subroutine Name
DATA    BASE
DATA    EXP
```

The following routines are available.

Name	Base	Exponent
$F4IIP	Integer	Integer
$F4RIP	Real	Integer
$F4DIP	Double	Integer
$F4CIP	Complex	Integer
$F4IRP	Integer	Real
$F4RRP	Real	Real
$F4DRP	Double	Real
$F4IDP	Integer	Double
$F4RDP	Real	Double
$F4DDP	Double	Double

3-9.3 MODE CONVERSION ROUTINES. Mode conversion routines are available for conversions between various arithmetic modes. These routines assume that the argument is in the hardware or software register appropriate to the mode of the argument. The result will be returned in the hardware/software register appropriate to the mode of the function. The registers and associated modes are as follows.

Mode	Register
Integer	A-Register
Real	ACC$, EXP$
Double	ACC$, EXP$
Complex	CACC$, CACC2$

Since arguments are assumed to be in an appropriate register, the calling sequence consists only of a call to the appropriate routine. For example:

```
BRL      *$+1
DATA     $F4IRC
```

The routines available are listed below.

Name	Function
$F4IRC	Integer to Real Conversion
$F4IDC	Integer to Double Conversion
$F4ICC	Integer to Complex Conversion
$F4RIC	Real to Integer Conversion
$F4RDC	Real to Double Conversion
$F4RCC	Real to Complex Conversion
$F4DIC	Double to Integer Conversion
$F4DRC	Double to Real Conversion
$F4DCC	Double to Complex Conversion
$F4CIC	Complex to Integer Conversion
$F4CRC	Complex to Real Conversion
$F4CDC	Complex to Double Conversion

3-9.4 COMPLEX ARITHMETIC. Complex arithmetic operations are performed by the sequence of routines listed below. The results of complex operations are stored in locations denoted symbolically by CACC$ and CACC2$. The calling sequence is

```
BRL      *$+1
DATA     Subroutine Name
DATA     Argument
```

Name	Function
$F4CLD	Complex Load
$F4CST	Complex Store
$F4CAD	Complex Add
$F4CSB	Complex Subtract
$F4CMP	Complex Multiply
$F4CDV	Complex Divide
$F4CSC	Complex Change Sign (Negate CACC$, CACC2$)

3-9.5 FLOATING POINT ARITHMETIC. Floating point arithmetic is performed by the use of the illegal instruction trap. The mode of operation is determined by the value in the location denoted symbolically as MODE$.

MODE$	Operation Mode
0	Real - Program Counter Relative
1	Real - Base Register Relative
2	Double Precision - Program Counter Relative

MODE$ is initially set to 1; therefore the user must reset MODE$ for the required mode if this mode is not to be used.

The addressing mode is determined by the I and X fields as in the example below for MODE$ = 1.

IX	Addressing
00	B+D
10	(B+D)
01	B+D+X
11	Illegal

Note that base register addressing is not specified through use of the B bit. It can be selected only by use of the mode word MODE$.

In any of the above modes, the following op codes are used. They may be defined by an OPD by the user.

FLD	OPD	>D000	Floating Load
FLN	OPD	>D100	Floating Load Negative
FST	OPD	>E000	Floating Store
FAD	OPD	>E800	Floating Add
FSB	OPD	>E900	Floating Subtract
FMP	OPD	>F000	Floating Multiply
MDV	OPD	>F100	Floating Divide
FNG	OPD	>C100	Floating Negate
FCP	OPD	>E100	Floating Compare

The codes thus defined are used in the same fashion as normal operation codes.

3-9.6 DOUBLE PRECISION ARITHMETIC. For convenience in code generation in FORTRAN, double precision arithmetic can also be performed by subroutines in addition to the illegal instruction traps described in 3-9.5. The calling sequence is

```
BRL      *$+1
DATA     Subroutine Name
DATA     Argument
```

The routines available are listed below.

Name	Function
$F4DLD	Double Precision Load
$F4DST	Double Precision Store
$F4DAD	Double Precision Add

Name	Function
$F4DSB	Double Precision Subtract
$F4DMP	Double Precision Multiply
$F4DDV	Double Precision Divide
$F4DCP	Double Precision Compare
$F4DCS	Double Precision Change Sign of Accumulator

3-9.7 FORTRAN BASIC EXTERNAL FUNCTIONS. The basic FORTRAN external functions are available for incorporation into assembly language programs. They are called by the standard calling sequence. See Table 3-2.

```
BRL         *$+1
DATA        Function Name
DATA        Number of arguments
DATA        ARG1 – First Argument
  .
  .
  .
DATA        ARGN – Last Argument
```

The function value is returned in the hardware/software register corresponding to the function type.

3-9.8 FORTRAN INTRINSIC FUNCTIONS. The FORTRAN intrinsic functions are called by the standard calling sequence.

```
BRL         *$+1
DATA        Function Name
DATA        Number of arguments
DATA        ARG1 – First Argument
  .
  .
  .
DATA        ARGN – Last Argument
```

The value of the function is returned in the hardware/software register corresponding to the function type shown in Table 3-3.

3-9.9 FORTRAN FORMAT EDITOR. The FORTRAN format editor is available as a relocatable library subroutine for use in assembly language programs. Formats must be provided by the user in the form of data statements, viz.

```
FMT1    DATA    '(2HA = F10.5)'
```

The portion enclosed between single quotes must be in accordance with the usual syntax of FORTRAN format statements.

There are 18 entry points to the FORTRAN format editor as shown in Table 3-4.

Calling sequences for the non-I/O functions are as shown below.

```
BRL         *$+1        End File
DATA        $F4EFI
DATA        UNIT NO.

BRL         *$+1        Rewind
DATA        $F4RWD
DATA        UNIT NO.

BRL         *$+1        Stop
DATA        $F4STP
DATA        N

BRL         *$+1        Pause
DATA        $F4PAU
DATA        N

BRL         *$+1        Back Space
DATA        $F4BSP
DATA        UNIT NO.
```

All Input/Output operations are initiated by a call to one of $F4WBN, $F4WBD, $F4RBN, or $F4RBD. The calling sequence is illustrated below.

```
BRL         *$+1
DATA        $F4WBD
DATA        FMT1
DATA        UNIT NO.
```

For each element of the argument list, there must be a call to an input/output routine. The last call in an I/O operation must be to the stop I/O ($F4SIO) routine.

To illustrate.

```
      BRL     *$+1        Initiate Output
      DATA    $F4WBD      Operation on unit
      DATA    FMT21       3 using FMT21
      DATA    3
      BRL     *$+1        Output I
      DATA    $F4IIO
      DATA    I
      BRL     *$+1        Output A
      DATA    $F4RIO
      DATA    A
      BRL     *$+1        Stop I/O
      DATA    $F4SIO
        .
        .
        .
FMT21 DATA    '(I4,F10.5)'
```

TABLE 3-2

FORTRAN BASIC EXTERNAL FUNCTIONS

Function	Definition	No. Arguments	Name	Arg. Type	Function Type
Exponential	e^a	1	EXP	Real	Real
			DEXP	Double	Double
			CEXP	Complex	Complex
Natural Log	$\log_e (a)$	1	ALOG	Real	Real
			DLOG	Double	Double
			CLOG	Complex	Complex
Common Log	$\log_{10} (a)$	1	ALOG10	Real	Real
			DLOG10	Double	Double
Sine	sin(a)	1	SIN	Real	Real
			DSIN	Double	Double
			CSIN	Complex	Complex
Cosine	cos(a)	1	COS	Real	Real
			DCOS	Double	Double
			CCOS	Complex	Complex
Hyperbolic Tan	tanh(a)	1	TANH	Real	Real
Square Root	$(a)^{1/2}$	1	SQRT	Real	Real
			DSQRT	Double	Double
			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*	$a_1 (\bmod a_2)$	2	DMOD	Double	Double
Modulus	$\lvert a_1+a_2 \rvert$	1	CABS	Complex	Real

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

TABLE 3-3

FORTRAN INTRINSIC FUNCTIONS

Function	Definition	No. Args.	Name	Arg. Type	Function Type
Absolute Value	$\|a\|$	1	ABS	Real	Real
			IABS	Integer	Integer
			DABS	Double	Double
Truncation	Sign of a times largest integer $\leqslant\|a\|$	1	AINT	Real	Real
			INT	Real	Integer
			DINT	Double	Integer
Remaindering*	$a_1 (\text{mod } a_2)$	2	AMOD	Real	Real
			MOD	Integer	Integer
Choose Largest Value	$\text{Max}(a_1,a_2,...,a_n)$	$\geqslant 2$	AMAX0	Integer	Real
			AMAX1	Real	Real
			MAX0	Integer	Integer
			MAX1	Real	Integer
			DMAX1	Double	Double
Choose Smallest Value	$\text{MIN}(a_1,a_2,...,a_n)$	$\geqslant 2$	AMIN0	Integer	Real
			AMIN1	Real	Real
			MIN0	Integer	Integer
			MIN1	Real	Integer
			DMIN1	Double	Double
Float	Convert Integer to Real	1	FLOAT	Integer	Real
Fix	Convert Real to Integer	1	IFIX	Real	Integer
Transfer of Sign	Sign a_2 times $\|a_1\|$	2	SIGN	Real	Real
			ISIGN	Integer	Integer
			DSIGN	Double	Double
Positive Difference	$a_1\text{-MIN}(a_1,a_2)$	2	DIM	Real	Real
			IDIM	Integer	Integer
Double to Real		1	SNGL	Double	Real
Real Part of Complex		1	REAL	Complex	Real
Imaginary Part of Complex		1	AIMAG	Complex	Real
Real to Double		1	DBLE	Real	Double
Two Reals to Complex	$a_1+a_2\sqrt{-1}$	2	CMPLX	Real	Complex
Complex Conjugate	a-ib	1	CONJ	Complex	Complex

*See footnote at bottom of Table 3-2.

TABLE 3-4

FORTRAN FORMAT EDITOR

Entry	Function
$F4EFI	End File
$F4RWD	Rewind
$F4STP	Stop
$F4PAU	Pause
$F4SIO	Stop Input/Output
$F4RBN	Read - Binary
$F4RBD	Read - ASCII
$F4WBN	Write - Binary
$F4WBD	Write - ASCII
$F4IIO	Integer/Logical Input/Output
$F4RIO	Real Input/Output
$F4CIO	Complex Input/Output
$F4DIO	Double Precision Input/Output
$F4IUA	Integer/Logical Unsubscripted Array Input/Output
$F4RUA	Real Unsubscripted Array Input/Output
$F4CUA	Complex Unsubscripted Array Input/Output
$F4DUA	Double Precision Unsubscripted Array Input/Output
$F4BSP	Back Space

The following example illustrates the input of a 3X4 integer array.

```
        BRL         *$+1
        DATA        $F4RBD
        DATA        FORMAT
        DATA        1
        BRL         *$+1
        DATA        $F4IUA
        DATA        12
        DATA        I
        BRL         *$+1
        DATA        $F4SIO
          .
          .
          .
FORMAT  DATA  '(I4)'
```

3-10 FORTRAN OPERATING PROCEDURES

The only version of FORTRAN which will be available initially will execute stand alone. It will use the high speed paper tape reader for source input and the high speed paper tape punch for object output. The listing is printed on the teletypewriter. Other stand alone and monitor controlled versions of the FORTRAN compiler are planned for future release. All code generated by the compiler is compatible with the RTM-I monitor.

The procedure for executing the compiler is as follows:

OPERATION:

a. Load program FTNPS1.

   Results: Program is ready for execution.

b. Push RESET button and then push the RUN button.

   Results: A message is printed on the teletypewriter as follows: READY FORTRAN SOURCE AND HIT RUN.

c. Ready the source and push the RUN button.

   Result: Compilation begins. The source tape is read, a source listing is produced on the teletypewriter (with appropriate error messages as required), and a pass 1 object tape is produced on the high speed paper tape punch. Compilation continues until the line PROGRAM END is printed on the teletypewriter. A trailer is punched on the object tape and the computer idles, and the FORTRAN first pass is ready for the next compilation. To execute the next first-pass compilation, repeat steps b. and c. To complete the

second pass of the compilation, continue with step d.

d. Load program FTNPSZ.

Results: Program is ready for execution.

e. Push RESET button and then push the RUN button.

Results: A message is printed on the teletypewriter as follows: READY PASS ONE OUTPUT FOR INPUT TO PASS TWO AND HIT RUN.

f. Ready the input and push the RUN button.

Results: The object is punched. At the completion of pass two the line COMPILATION COMPLETE is printed on the teletypewriter. The computer idles and the FORTRAN second pass is ready for the next input. To execute another pass two repeat steps e. and f.

SECTION IV

REAL TIME MONITOR – I

SECTION IV

REAL TIME MONITOR – I

4-1 INTRODUCTION.

The Real Time Monitor System–Version I, abbreviated RTM-I:

a. Provides continuous operation of the Model 980 Computer and its peripheral equipments with a minimum loss of computation time.

b. Provides software control of all inputs and outputs.

c. Schedules multiprogrammed foreground worker programs based upon real time input stimuli.

d. Schedules background batch processing tasks on a demand basis.

e. Facilitates use of program development tools which are available as background batch task processors.

RTM-I has four distinct but interdependent segments which are described in more detail in paragraphs 4-1.1 through 4-1.4.

4-1.1 SUPERVISOR. The supervisor is the heart of the monitor. It provides interrupt processing and delegates computer time to other program segments on a priority basis.

4-1.2 SERVICE SUBPROGRAMS. Service subprograms perform the work requested by the executive calls within the user programs. These service subprograms generally fall into two categories:

a. Input/output handlers which police the transmission of data between user programs and peripheral equipment

b. Control function handlers which perform housekeeping functions, such as user task termination.

4-1.3 BACKGROUND JOB CONTROL. This segment optionally has one of two program elements. Both elements provide a program interface between background program jobs and the supervisor.

The Background job control program SPEX (Single Program Executive) is the simplest option. SPEX allows a single background program to be loaded into core and executed.

The more complex option is called BATCH. BATCH accepts and interprets a job control input, and selects appropriate disc resident programs for background execution.

4-1.4 THE DISC INITIALIZATION PROGRAMS. This last segment consists of two self-contained programs. These programs load the disc for use in a BATCH environment. One restores the disc from object media to a state which existed when it was preserved. The second performs the more complex buildup of the disc from scratch.

Figure 4-1 shows the basic RTM-I structure. Dashed lines enclose elements unique to RTM-I with BATCH. The illustration also shows how user-generated programs relate to the basic RTM-I structure. The terms *foreground* and *background* refer to relative levels of priority.

4-2 THE SUPERVISOR.

The Supervisor is the nucleus of RTM-I, a multiprogram operating system utilizing an executive/worker method for program control and a multilevel priority scheme for program execution. *Multiprogramming* as used here implies the continuous coexistence of a number of worker programs in memory in varying stages of execution. The term *multilevel priority* refers to the technique used by the Supervisor to schedule and execute worker tasks based upon their relative urgency.

Tasks are divided between three major program levels: X-level (interrupt programs), F-level (foreground control and foreground worker task programs), and B-level (background batch processing). The interrelationship of these levels is shown in Figure 4-2.

4-2.1 X-LEVEL. The highest priority programs are those which directly answer hardware generated interrupts. Programs at this level are activated by:

a. Internal interrupts including power failure and undefined instruction detection.

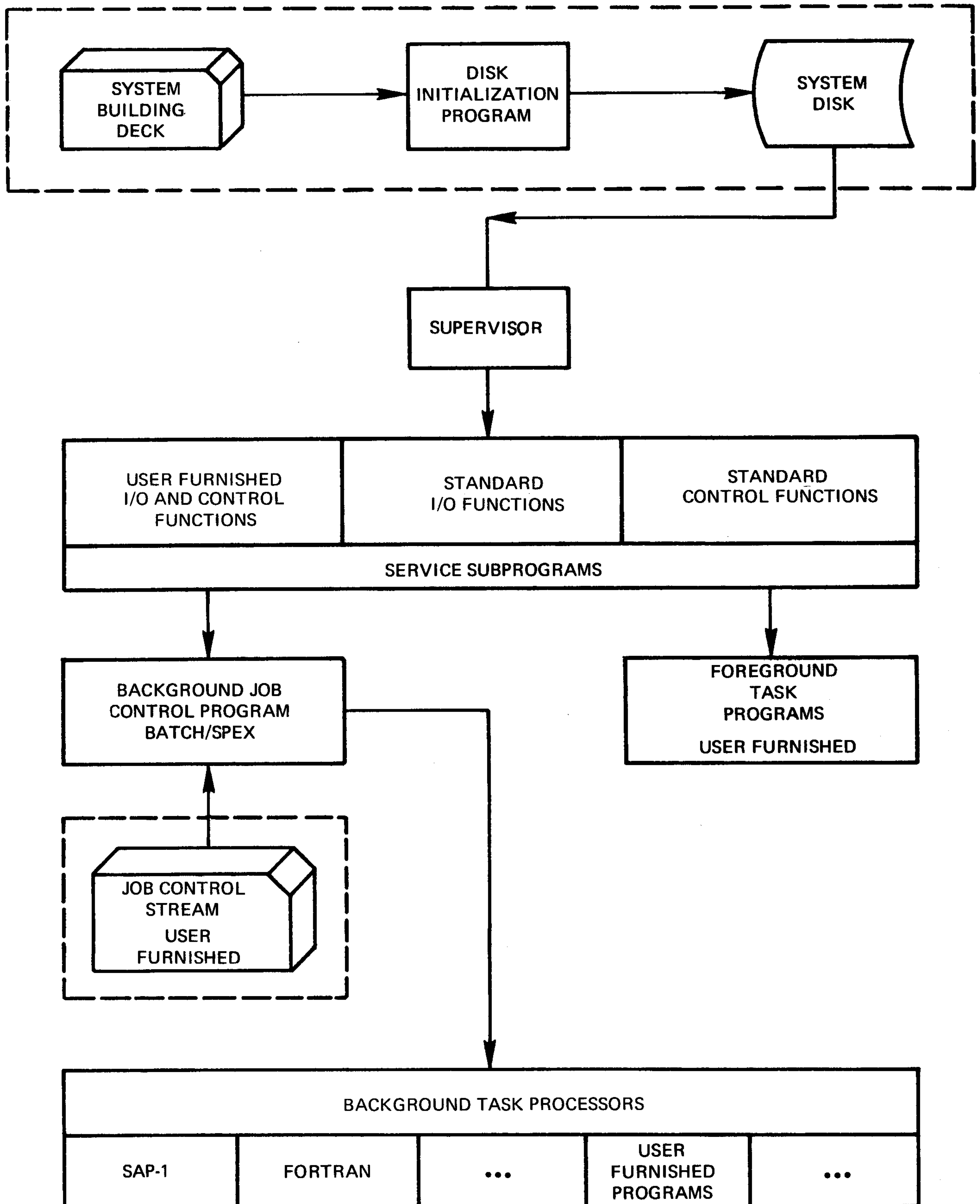


Figure 4-1. RTM Basic Structure

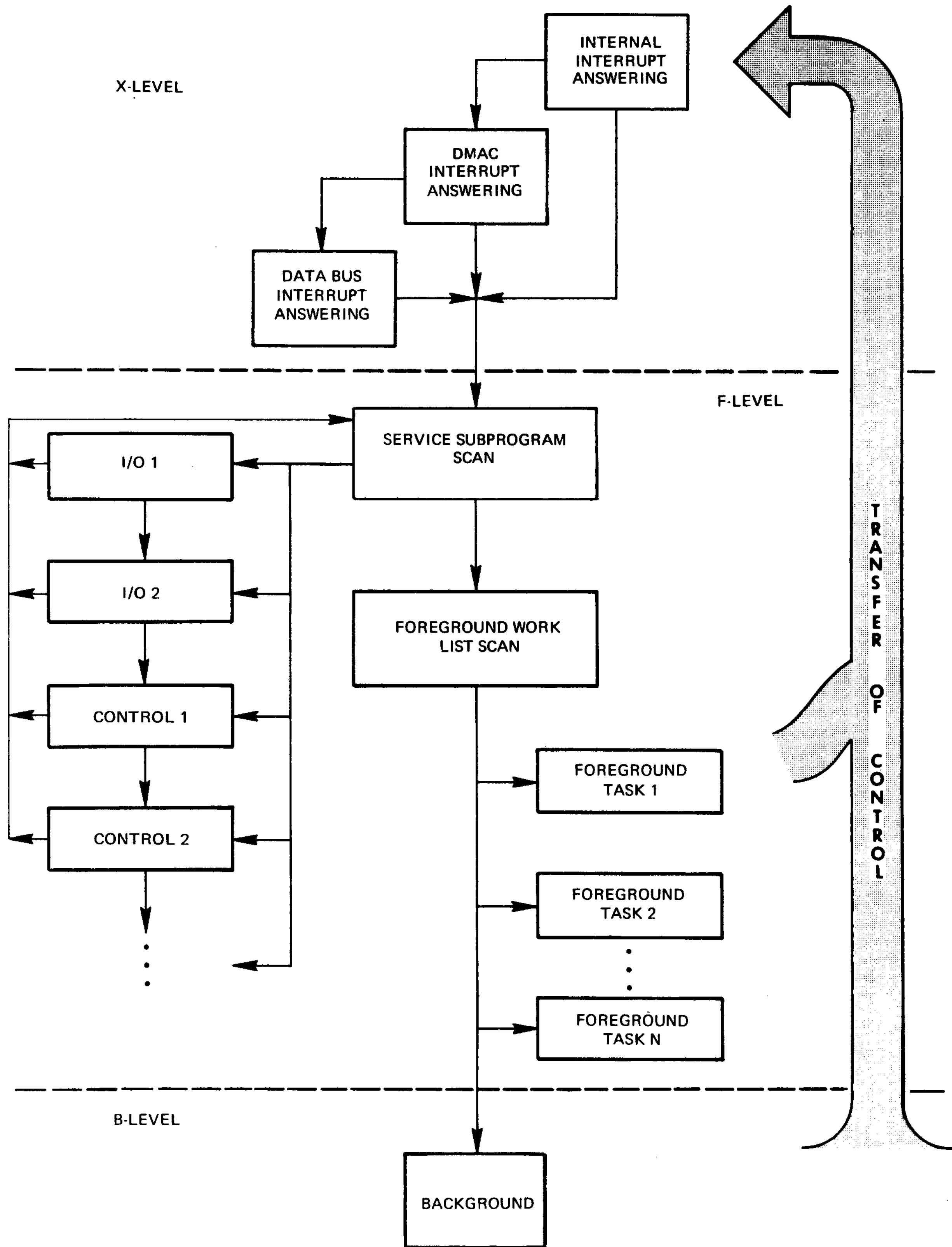


Figure 4-2. Supervisor Priority Levels

TABLE 4-1

X-LEVEL RETURN

Control Prior To Interrupt	Type of Interrupt	Control Return Point
Any	Power Fail	Power Fail Shut Down
X-Level Data Bus	DMAC	Point Interrupted
F-Level Control or Service Subprogram	Data Bus or DMAC	Point Interrupted
F-Level Foreground Task	Data Bus, DMAC, Undefined Op-Code	Restart F-level Control
B-Level	Data Bus, DMAC, Undefined Op-Code	Restart F-Level Control
Any Except Foreground or Background Task	Undefined Op-Code	Catastrophic Failure

b. DMAC interrupts, including disc, magnetic tape and line printer.

c. Data Bus interrupts including the teletypewriter, paper tape equipment, and card reader.

X-level programs operate completely within time-islands bounded by the inhibit of interrupts, which occur at interrupt answering, and the enabling of further interrupts at the end of each X-level sequence. Therefore, X-level subprograms must operate within the smallest practical time-frame.

Additional priority stratification is used within the X-level. Internal interrupts, being of highest priority, can actually interrupt the processing of DMAC and Data Bus interrupts. DMAC interrupts can occur during the processing of a Data Bus interrupt.

In general only minimum processing is done at the X-level. The primary purpose is to pass pertinent information to the F-level.

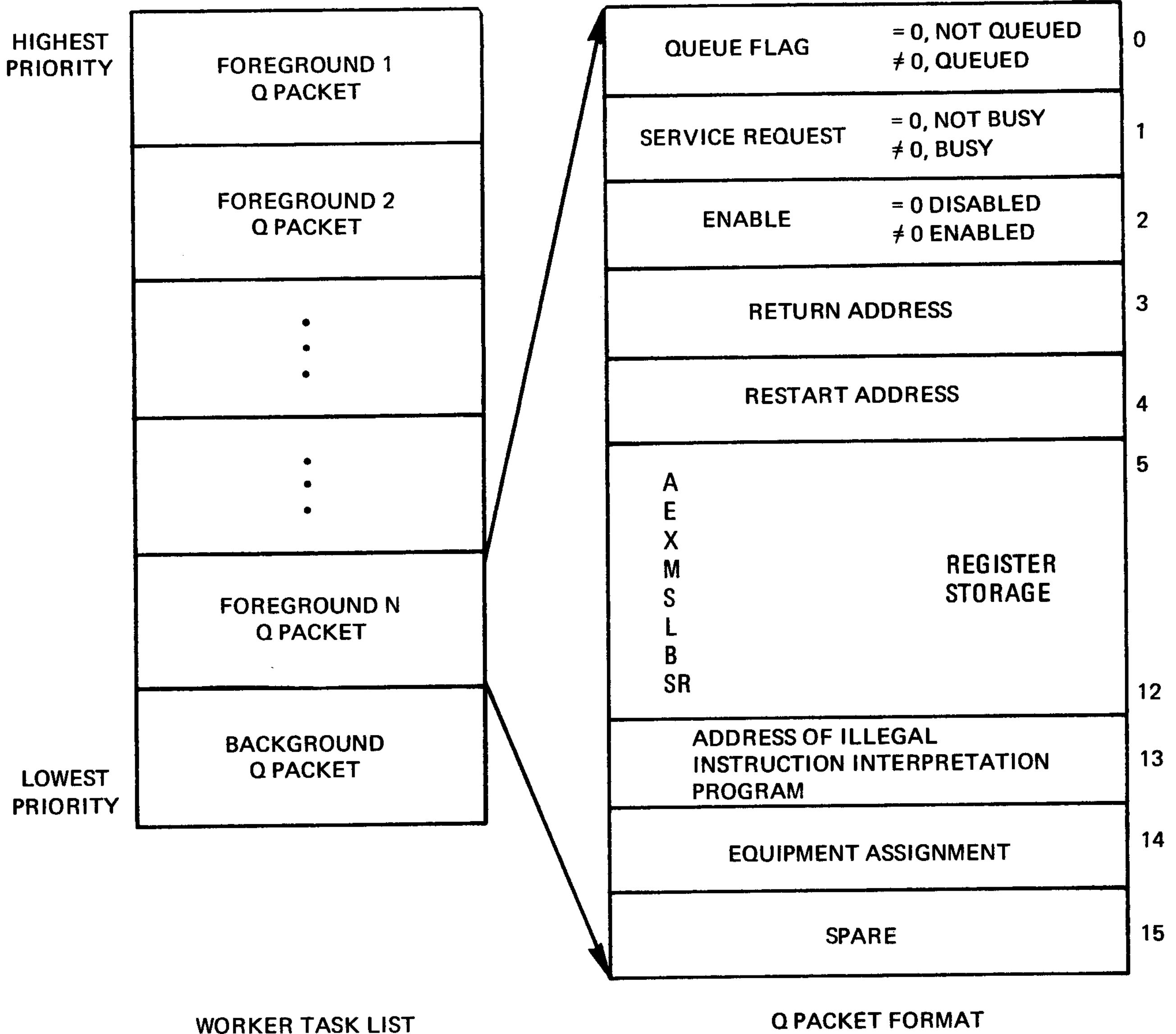
When an interrupt occurs, the X-level is executed. The previous operating level could have been the X-level, F-level, or B-level. Within the F-level, operation may have been within the control portion, or within a foreground task. When the X-level sequence is complete, control is yielded to varying points within the supervisor, depending upon which level was originally interrupted. This relationship is shown in Table 4-1.

4-2.2 **F-LEVEL.** The foreground level contains the resident worker programs and service subprograms.

The F-level is divided into two sections: the foreground control section (part of the supervisor) and the foreground worker tasks and service subprograms (external to the supervisor). The purpose of the control section is to schedule and execute the F-level worker tasks and service subprograms on a priority basis. The foreground worker tasks are supplied by the user and perform application functions. Service subprograms may also be user supplied.

The F-level gains control from the X-level whenever its tasks are complete. Any service request which may have occurred will have been placed in a queue (waiting list) at the X-level for subsequent activation of the appropriate service subprogram. The service requests (including I/O, control functions, and user service subprograms) and service subprogram status are checked to determine if an event has occurred that should cause them to operate. If any service subprogram execution is thus enabled it may either initiate processing of the next service request in its queue, or it may continue or terminate the processing of a previous request.

TABLE 4-2 WORKER TASK LIST



When all service subprograms have completed whatever processing can be accomplished without undue delays, the control portion of the F-level supervisor starts scanning the F-level worker task list to determine the next foreground worker to be executed. This is done by sequentially examining *Q Packets.* A typical Q Packet is shown in Table 4-2. The program executed will be the first one in the string of Q Packets which is enabled, queued (or requested to be made active), and is not awaiting completion of a service request.

The foreground program being executed retains control until a hardware interrupt occurs or until it returns control to the supervisor via a service request. The service requests available to the foreground task program can be categorized as follows:

a. Execute I/O Service. These cause an I/O request to be queued for one of the I/O service subprograms. In this case, execution of the worker is suspended until completion of the requested I/O action.

b. Perform Control Function. These are housekeeping function requests such as a request to assign a peripheral device, to activate (or queue) another foreground task, or to terminate the task.

If an illegal service request is encountered or an irrecoverable I/O condition occurs, the appropriate worker is removed from the foreground scan (disabled).

4-2.3 **B-LEVEL.** The B-level consists of a job control program (BATCH or SPEX) and various task processors which are considered external to RTM-I.

The function of BATCH is to schedule and load task processors as directed by job control statements input via the card reader. Once loaded the task processor is performed like any foreground program, i.e., control is retained until an interrupt occurs or a service request is encountered. SPEX will load and execute a program upon receipt of an operator request. It will use a teletypewriter, high speed paper tape reader, or card reader to read the object.

If an illegal service request or an irrecoverable I/O condition occurs in a background task processor, the task processor execution is terminated. BATCH or SPEX resume monitoring for background task requests.

4-3 SERVICE SUBPROGRAMS.

Service Subprograms generally fall into two categories:

a. those that respond to control function requests

b. those that respond to input/output service requests.

A library of System Service Subprograms is furnished with RTM-I. The user, however, may elect to add to this library or to replace a portion of the System Service Subprograms with his own. This portion of the User's Guide is devoted to acquainting the user with the interface which exists between Service Subprograms and the remainder of the system.

4-3.1 **I/O SERVICE.** RTM-I allows for a total of 80 Service Subprograms to be resident in core. Of these, 48 may respond to I/O service requests. A table of 48 address constants is resident in the supervisor in table X.DVTB. This table contains the following sequence of names, each defined (with respect to the supervisor) as an external reference:

```
X.DVTB DATA D.V01A
       DATA D.V01B
       DATA D.V02A
       DATA D.V02B
       DATA D.V03A
       DATA D.V03B
              .
              .
              .
       DATA D.V24A
       DATA D.V24B
```

These addresses act as pointers. Each address may point to a twelve-word *device table* which is supplied externally to the supervisor.

4-3.2 **I/O SERVICE DEVICE TABLES.** The names D.V01A, D.V01B, ..., D.V08B refer to device tables dealing with the eight potential DMAC ports. The digits 1 to 8 refer to ports 0 through 7 respectively. The A and B allow two tables per port. The two tables are used commonly by the input and output functions respectively if the device performs both input and output. However, there is no system requirement that the two tables be used this way.

The names D.V09A, D.V09B, . . ., D.V24B refer to device tables dealing with the 16 potential bit positions on the Data Bus interrupt expander. The A and B allow two tables per interrupt expander bit position. Again they commonly, but do not necessarily, refer to the input and output functions. The numbers 09 to 24 refer to bit positions 0 through 15 respectively.

Each device controller attached to the system *must* have a twelve-word device table bearing the appropriate name D.VxxA (where xx is a number from 01 to 24). It may optionally have a second device table named D.VxxB. The twelve words within the device table contain information which interfaces an I/O service subprogram with the supervisor.

WORD ONE. This word contains the address where the supervisor and the service subprogram may either find or put the status word received from the device controller. With the DMAC, the addresses begin with 0098_{16} and continue through $00A7_{16}$ at two words per port. Usually, the device controller will automatically deposit status in the first word of the pair for the port. It is possible to use the second word in conjunction with device table D.VxxB, provided the controller is built to operate that way.

With the Data Bus, the supervisor will read the status from the device at interrupt response time. The status is obtained from an external device address given in Word Twelve of the device table and is stored in the location given by Word One. If both D.VxxA and D.VxxB are defined, then both status words are stored in response to a single interrupt. If D.VxxA is not defined for a given interrupt expander bit position, then no status is stored.

WORD TWO. Word Two contains a static bit value which could never be attained as a legal status word. Usually, it is all zeros or all ones. When initiating an I/O activity, the service subprogram will be directed to an address by Word One. The value of the data at that address should be set equal to the value of Word Two by the service subprogram.

When determining which service subprograms to execute, the supervisor will go to the address which is given by Word One and compare the value of the content of that location

with the value of Word Two. If they are the same, the supervisor assumes the device has not completed the I/O activity or does not require service. If the values differ, then the supervisor will execute the service subprogram for device service.

WORD THREE. Word Three contains all zeros except for possibly a single one bit. The bit position of the one bit corresponds to the position for this device in an equipment assignment word. Bit position 0 is illegal for this purpose. Up to 15 devices can be assigned. That is, they may be requested for assignment by a F-level or B-level task prior to use. Once assigned, no other task may use the device until it is released. Word Three may be set to zero. If Word Three is set to zero, then the device may be used by any task on a *first come, first serve* basis. Standard equipment assignments follow:

Bit 9	Card reader
Bit 10	Line printer
Bit 11	Magnetic tape
Bit 12	Disc
Bit 13	High speed tape reader
Bit 14	High speed tape punch
Bit 15	Teletypewriter

WORD FOUR. Word Four contains the address of a block of memory which the service subprogram may use for data storage. This feature allows the same service subprogram to be used for more than one similar device. The amount of storage required is dependent upon the individual service subprogram.

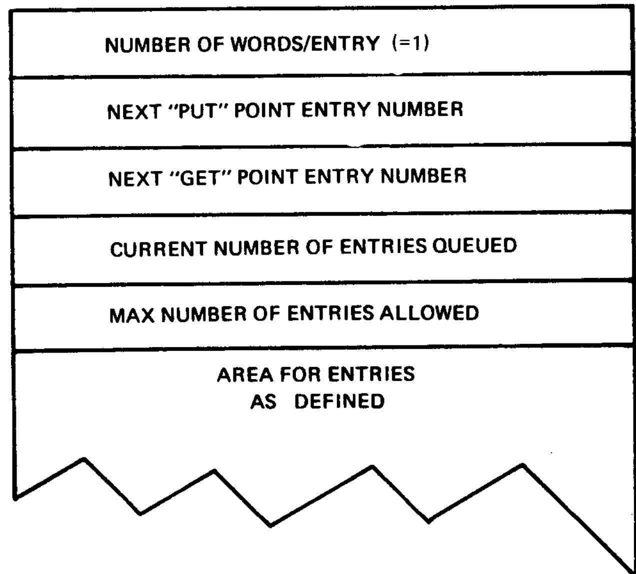
WORD FIVE. This word contains the address of the I/O service subprogram.

WORD SIX. Word Six is the address of a flag for communication between the service subprogram and the supervisor. If the flag is zero, the service subprogram is telling the supervisor there is no current activity with the device and the service subprogram is in a condition to process a new service request.

If the flag is set not zero by the service subprogram, then activity with the device is assumed, and the supervisor will return to the service subprogram only when the status word becomes different from the static value in Word 2.

WORD SEVEN. Word Seven is the queue starting address for the stacking of service requests for this device. Items are placed in the stack when a task performs a service request. The queue is composed of one-word entries.

The queue format follows:



WORD EIGHT. Word Eight is the service request call address. The service subprogram uses this address to locate the arguments for the request to be processed. The supervisor obtains this address from the queue defined in Word Seven. It places a new address in Word Eight whenever the flag addressed by Word Six is zero and the queue is not empty.

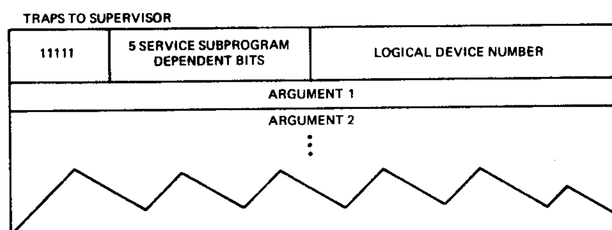
WORD NINE. This word is reserved for future expansion to the supervisor.

WORD TEN. This word serves as a flag to the supervisor and denotes the relative urgency of response to change in status value from the device. If zero, the supervisor will wait until the next Service Request Scan before entering the service subprogram. Normally this will be adequate. The service subprogram would then be entered in its turn as the supervisor scans the device tables in order. If Word Ten is not zero, the supervisor will enter the service subprogram directly from the X-Level. The Service Subprogram may, of course, vary its response urgency by altering the value of this flag between zero and not zero. Word Ten, however, only applies to Data Bus devices; that is D.V09A through D.V24B. This flag must always be zero for DMAC devices.

WORD ELEVEN. This word contains the logical device number for the peripheral to which this device table refers. Each device must have a unique logical device number.

WORD TWELVE. This word applies to Data Bus devices only and contains the group and external address number for obtaining device status and the group and external address number for obtaining data. The values correspond to the rightmost seven bits of an RDS or WDS instruction. The value for addressing the device status is contained in the least significant eight bits and the value for addressing data in the most significant bits.

4-3.3 I/O SERVICE REQUESTS. Service requests by a task program use the undefined operation interrupt feature to link to the supervisor. The op-code $(11111)_2$ is used for I/O Service Requests. A call takes the general form:



The request is queued for the device corresponding to the logical unit specified in the least significant six bits of the word containing the $(11111)_2$ op-code.

4-3.4 I/O SERVICE GENERAL FLOW AND LINKAGE. Figures 4-3 and 4-4 are general flow diagrams showing the structure of an I/O service subprogram and the corresponding supervisor segments.

The I/O Service Subprograms are entered at either of two sequential locations depending upon the purpose of entry. Entry at START implies the flag addressed by Word Six in the device table is zero, the device is not currently busy, and a new service request address has been placed in Word Eight of the device table. Entry at START +1 implies that a change in status word has occurred, and the device has been busy.

In any event the address of the device table is put in register M by the supervisor at entry. Entry is by Branch and Link. The service subprogram may use registers at will without the necessity to save and restore them. The service subprogram itself may not make any service request calls.

The service subprogram returns control to the supervisor at either of the two sequential locations following the branch and link instruction.

The first return is provided for unrecoverable error conditions, the second for normal returns. The error return causes the supervisor to *disable* the calling F-level or B-level task, and to unassign its equipment pending operator intervention.

The service subprogram, prior to return, must make appropriate adjustments to the device table. This includes updating Word Eight to now point immediately past the last argument of the call. The service subprogram must do this since it alone knows the length of the argument list.

4-3.5 CONTROL FUNCTION SERVICE. RTM-I interfaces with up to 32 service subprograms which respond to control function service requests. Certain of these belong to the basic system service subprogram library and are necessary for system integrity.

Each control function service subprogram bears the name "C.Fxx", where xx is a number from 01 to 32. Each such subprogram supplied by the user should be separately assembled and linked with the supervisor at system generation time. It should define its name, C.Fxx, as an entry point.

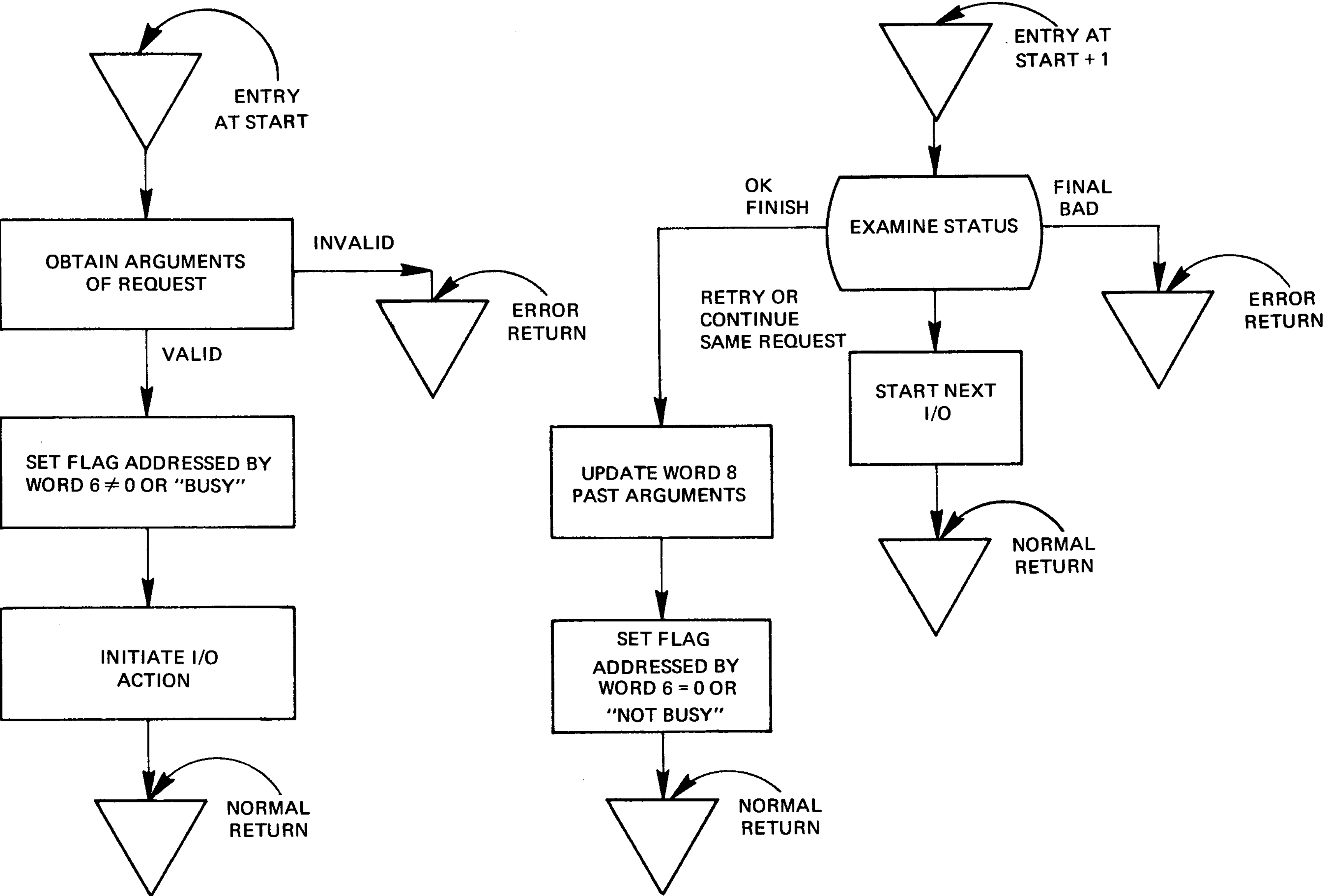


Figure 4-3. I/O Service Request General Flow

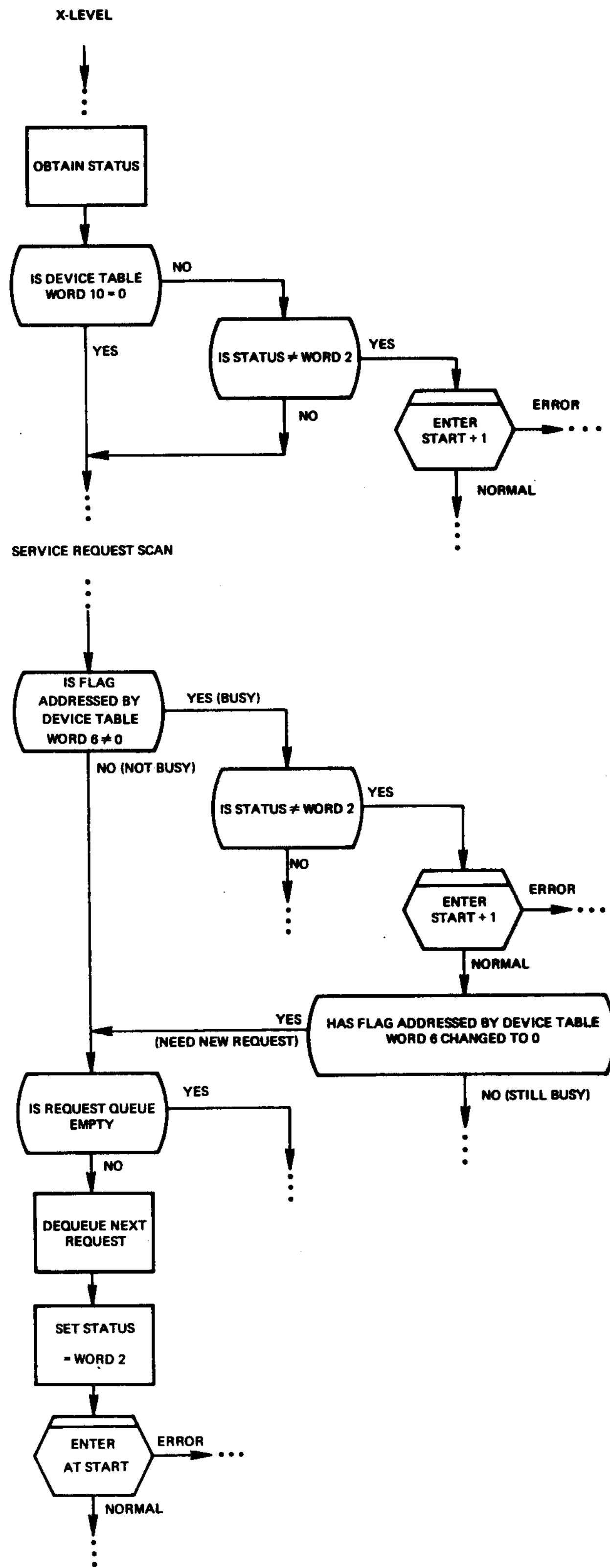


Figure 4-4. Supervisor Flow Relative To I/O Request

The routines C.V01 through C.F07 are predefined as part of the monitor. They accomplish the control functions outlined below:

C.F01 Terminates and dequeues calling task and returns control to supervisor. Releases its assigned equipment.

C.F02 Requests that another task, as specified, be made active.

C.F03 Releases control momentarily to supervisor for a scan of the work list deleting the calling task for one scan only (allows next lower priority program momentary access).

C.F04 Assigns indicated equipment.

Bits 1-8 - non standard equipment
Bit 9 - card reader
Bit 10 - line printer
Bit 11 - magnetic tape
Bit 12 - disc
Bit 13 - high speed tape reader
Bit 14 - high speed tape punch
Bit 15 - teletypewriter

C.F05 Terminates calling BATCH background task. Releases its assigned equipment. The next task in the input stream may now to processed.

C.F06 Requests a background task processor to be loaded and executed. This control function may be used by BATCH background task processors only. Through use of this function a large background program may be broken into several smaller programs and the smaller programs executed sequentially. Data is passed from one program to another via disc or magnetic tape or may be left in the high numbered core locations. If the last option is selected it is the programmer's responsibility to assure that the incoming program does not overlay its data. This technique is called *chaining*.

C.F07 Used to pass to the monitor the address of a program which will interpret any non-system illegal operation codes. This service request must be used before any non-system illegal operation codes are used. If not RTM-I will assume the illegal instructions are not valid and will disable the program using the illegal instruction. The floating point arithmetic package is the most commonly used program in this category.

User supplied control functions should be numbered C.F32, C.F31, . . ., as low numbered control functions will be used as future additions are made to the system service subroutine library.

Control function service subprograms work in much the same manner as I/O service subprograms. The supervisor will enter them with the address of a table (similar to a device table) in register M.

The following sequence represents a supervisor entry into a control function service subprogram:

Communication Table Address → M
BRL C.Fxx
error return
normal return.

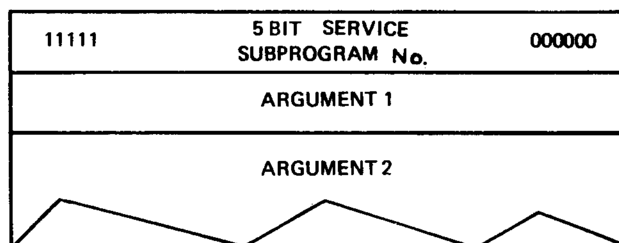
The error return will occur if the arguments of the service request call are invalid. In this event the supervisor will disable the offending task.

The communication table is an eight-word table within the supervisor. The address is in register M at entry to the service subprogram. The format is as follows:

Words 1-7 Reserved for supervisor use.

Word 8 Address of service request call. Must be advanced past arguments by the handler before exit.

Control function service requests in a task program use the undefined operation code $(11111)_2$ for linkage to the supervisor. A call appears as follows:



The five-bit number correspondence is: $(00000)_2$ to C.F01, $(00001)_2$ to C.F02, ..., $(11111)_2$ to C.F32.

4-3.6 NON-SYSTEM ILLEGAL INSTRUCTION TRAPS. By use of the control function C.F07 the user may interface a program with the monitor to do additional interpretation of illegal instruction traps. If such a service request has been made the monitor will test illegal

instructions for all ones in bits 0-4. If these bits are all ones it will proceed to process the trap as a service request. If not the illegal instruction will be passed to the user supplied program for further interpretation.

RTM-I calls the illegal instruction program as follows. In location START is stored the address of the illegal instruction + 1. In location START+1 is stored the contents of the status register at the time the illegal instruction was encountered. The L register contains an address to which control is returned if an error condition is detected. RTM-I will transfer control to location START+2. The illegal instruction interpretation program should return control directly to the instruction immediately after the illegal instruction for normal returns. This may be done by executing an @LSB START instruction. All interrupts are locked out during execution of the illegal instruction interpretation routine.

One such program is the Floating Point Arithmetic Package. This is discussed in more detail in Section III.

4-4 FOREGROUND TASKS.

Foreground task programs are supplied by the user. They perform the application oriented functions of the total system. Foreground task programs are assembled independently and linked to the supervisor at system generation time.

The load time linkage with the supervisor is through the worker task list (Table 4-2). During execution a foreground task calls upon the supervisor with service requests. These are outlined in paragraphs 4-3.4, 4-3.5, and 4-5.

Foreground tasks are priority oriented by their relative position in the worker task list. The user establishes this priority at system generation time.

Foreground tasks should not use any storage space not explicitly allocated to them through use of such statements as SAP-I BSS and COMM, or FORTRAN variable definition. In particular high numbered core locations are not necessarily unused. These locations are allocated to background task processors.

4-5 STANDARD SERVICE REQUESTS.

Each service request has three fields: the operation code in bits 0-4, the call code in bits 5-9 and the logical device number in bits 10-15. The service request may be followed by any number of arguments to be used by the service subprogram. The number of arguments is unique for each service subprogram.

The operation code used is the undefined operation code 11111_2.

The call codes are unique for each service subprogram. They generally correspond to the major functions which the user may wish the service subprogram to perform.

The logical device numbers are 0 for control functions and 01-48 for I/O service requests.

The service request may be assembled in the user program through use of a DATA statement or through use of an operation defined by a FRM directive. In the following it is assumed that a FRM has been used to define the operation SVR as follows:

```
SVR        FRM        5,5,6
```

The logical units indicated are the standard logical unit assignments listed in Section I.

Magnetic tape I/O Requests specify a unit number as part of the call code. This may be handled by use of an FRM assembler directive as follows:

```
SVT        FRM        5,2,3,6
```

In the following the tape unit number is indicated symbolically as UNIT.

4-5.1 BASIC CONTROL FUNCTIONS.

Terminate Task

```
SVR     >1F,0,0
```

Remove this task from foreground. Terminates program and unassigns its equipment.

Queue a Task

```
SVR     >1F,1,0
DATA     number
```

Activate (or queue) the foreground task associated with Q packet number N in the worker task list.

Momentarily Release Control

```
SVR     >1F,2,0
```

No operation. Releases control momentarily to supervisor for a single scan of the work list – deleting the calling task for one scan only. Allows the next lower priority program momentary access.

Assign Equipment

```
SVR     >1F,3,0
DATA    bits
```

Requests assignment of the equipments represented by ones in the least significant 15 bits of the argument word.

Return Control to BATCH

```
SVR     >1F,4,0
```

Terminates execution of a BATCH background task processor. Unassigns its equipment.

Chain Tasks

```
SVR     >1F,5,0
DATA    card address
```

Use the "control card" pointed to for requesting a new program to overlay the current background task. The "control card" will have the same format as a physical control card except that the card image need not necessarily be a full 80 columns. Only as many columns as contain useful information need be used. The card format uses 2-character per word ASCII code.

Initialize Illegal Instruction Interpretation

```
SVR     >1F,6,0
DATA    address
```

The program pointed to is to be used to perform additional interpretation of illegal instruction traps as described in paragraph 4-3.7.

4-5.2 TELETYPEWRITER I/O REQUESTS.

Print – One Character per Word

```
SVR     >1F,0,1
DATA    buffer address
DATA    number of characters
```

Print N characters, stored one per word right justified, on the teleprinter.

Print – Two Characters per Word

```
SVR     >1F,1,1
DATA    buffer address
```

Print characters from the buffer, stored two per word left to right, on the teleprinter until encountering a terminating character of all binary zeros.

Input – One Character Per Word

```
SVR     >1F,0,2
DATA    buffer address
DATA    terminating character
DATA    number of characters
```

Input a maximum of N characters or until the terminating character is found. Characters are stored right justified one per word in the buffer. At return the A-register contains the actual character count.

Input – Two Characters per Word

```
SVR     >1F,1,2
DATA    buffer address
DATA    terminating character
DATA    number of characters
```

Input a maximum of N characters or until the terminating character is found. Characters are stored left-to-right two per word in the buffer. At return the A-register contains the actual character count.

4-5.3 DISC I/O REQUESTS.

Read N Sectors

```
SVR     >1F,0,8
DATA    disc track-sector address
DATA    buffer address
DATA    number of sectors
```

Read N sectors from disc into the buffer.

Write N Sectors

```
SVR     >1F,1,8
DATA    disc track-sector address
DATA    buffer address
DATA    number of sectors
```

Write N sectors from buffer onto the disc.

Open Background Work Area File

SVR	>1F,2,8
DATA	file number F
DATA	relative track-sector address

Open a disc file number $F \leq 4$. The file is located in the BATCH Work Area at the address given. The address is relative to the start of the Work Area. This call is available only if BATCH is present. The file start remains defined until a new open is issued for that file number. The definition carries forward to the next job in the job stream. Thus files in the disc Work Area may be used to pass information from one job to the next.

Write on Background Work Area File

SVR	>1F,3,8
DATA	file number F
DATA	buffer address
DATA	number of sectors

Write N sectors onto disc file F starting at the next available sector in the file. The data is obtained from the buffer in core. This call is available only if BATCH is present. The disc file is treated as a sequential file. Each request adds more data to the file. At return the A-register will contain the next available track-sector address relative to the start of the Work Area. This may be used in building an XDOPEN for a subsequent file.

Read from Background Work Area File

SVR	>1F,4,8
DATA	file number F
DATA	buffer address
DATA	number of sectors

Read N sectors from disc file F starting with the next sequential sector in the file. The data is placed in the buffer in core. This call is available only if BATCH is present. The disc file is treated as a sequential file. Each request reads more data from the file. At return the A-register will contain the next sequential track-sector address relative to the start of the Work Area. This may be useful in opening a subsequent file.

Reset to Start of File

SVR	>1F,5,8
DATA	file number F

Reset file F to the start of the file. This call resets both the read and write pointers for the file.

4-5.4 HIGH SPEED PAPER TAPE READER I/O REQUESTS.

Read in Object Format

SVR	>1F,0,4
DATA	buffer address
DATA	number of words
DATA	terminating character

Read paper tape in object format. Store data beginning at the buffer address. Read up to N words or until a terminating character is read. At return the A-register contains the actual word count.

Read – One Character per Word

SVR	>1F,1,4
DATA	buffer address
DATA	number of characters
DATA	terminating character

Read up to N frames of paper tape. Store 2 characters per word left-to-right in the buffer. Stops after N characters or when the terminating character is read. Actual character count returned in the A-register.

Read – Two Characters per Word

SVR	>1F,2,4
DATA	buffer address
DATA	number of characters
DATA	terminating character

Read up to N frames of paper tape. Store 2 characters per word left-to-right in buffer. Stops after N characters or when the terminating character is read. Actual character count returned in the A-register.

4-5.5 HIGH SPEED PAPER TAPE PUNCH I/O REQUESTS. On return the A register contains punch status. The status values are zero-normal and non zero-low tape.

Turn Punch Motor On

SVR	>1F,0,3

Turn punch motor on.

Turn Punch Motor Off

```
SVR     >1F,1,3
```

Turn punch motor off.

Punch One Character

```
SVR     >1F,2,3
```

Punch character in 8 least significant bits of the A-register.

Punch in Object Format

```
SVR     >1F,3,3
DATA    buffer address
DATA    number of words
```

Punch N words from buffer in object format.

Punch – One Character per Word

```
SVR     >1F,4,3
DATA    buffer address
DATA    number of characters
```

Punch N Characters, stored 1 per word in the 8 least significant bits.

Punch – Two Characters per Word

```
SVR     >1F,5,3
DATA    buffer address
DATA    number of characters
```

Punch N characters, stored 2 per word left-to-right.

Punch Leader/Trailer

```
SVR     >1F,6,3
```

Punch 120 blank frames.

4-5.6 CARD READER I/O REQUESTS. The card reader service subprogram tests for $ in column 1 of any card it reads. If found it uses an alternate return. The user program should determine if a control card was actually read. If so the end of the input cards has been reached. Do not read further.

Read in Binary

```
SVR     >1F,0,5
DATA    buffer address
DATA    number of columns
        control card format return
        normal return
```

Read N columns from the card reader. Store the data as a binary card image (1 column per word), 9's row through 12's row in bits 4-15 respectively.

Read – One Character per Word

```
SVR     >1F,1,5
DATA    buffer address
DATA    number of columns
        control card format return
        normal return
```

Read N columns, convert to ASCII, store 1 per word right justified.

Read – Two Characters per Word

```
SVR     >1F,2,5
DATA    buffer address
DATA    number of columns
        control card format return
        normal return
```

Read N columns, convert to ASCII, store 2 per word, left-to-right.

4-5.7 LINE PRINTER I/O REQUESTS.

Print on Next Line

```
SVR     >1F,0,6
DATA    buffer address
DATA    number of characters
```

Advance form one line and print N characters stored 2 per word left-to-right.

Print at Top of Form

```
SVR     >1F,1,6
DATA    buffer address
DATA    number of characters
```

Advance to form top and print N characters stored 2 per word left-to-right.

Print on Current Line

SVR	>1F,2,6
DATA	buffer address
DATA	number of characters

Suppress form control and print N characters stored 2 per word left-to-right. Note this appends data to a partially completed line. Neither a carriage return nor a line feed are output unless present in the user's buffer.

4-5.8 MAGNETIC TAPE I/O REQUESTS. For all tape service requests, at return the A-register contains the following status word:

zeros	=	Normal
bit 0	=	End of Record (EOR)
bit 1	=	End of File (EOF)
bit 2	=	End of Tape (EOT)
bit 3	=	Beginning of Tape (BOT)
bits 4-15	=	Number of records not skipped or characters not read

All tape service requests contain an extra 2-bit parameter which specifies unit number. The call code is reduced to its rightmost 3 bits to make room for the unit number.

Unload Tape

SVT >1F,UNIT,0,7

Unload tape unit.

Rewind Tape

SVT >1F,UNIT,1,7

Rewind tape unit.

Write End-of-File

SVT >1F,UNIT,2,7

Write end of file mark.

Backspace

SVT	>1F,UNIT,3,7
DATA	number of records

Backspace N records or until detecting an EOF or BOT whichever occurs first. Can be used to backspace file by setting N to a large number.

Write

SVT	>1F,UNIT,4,7
DATA	buffer address
DATA	number of characters

Writes N characters, packed left-to-right 2 per word, as a tape record.

Read

SVT	>1F,UNIT,5,7
DATA	buffer address
DATA	number of characters

Read N characters or until EOR, EOF, or EOT whichever occurs first. Stores data packed 2 characters per word left-to-right.

Skip

SVT	>1F,UNIT,6,7
DATA	number of records

Skips forward N records or until detecting an EOF or EOT whichever occurs first. Can be used to forward space file by setting N to a large number.

4-6 SPEX.

SPEX is a background job control program for those users who do not require the disc capabilities of the BATCH processor. SPEX will load and execute a program upon request from the user. Upon completion of the program execution no further background processing will take place until the next request is received.

A SPEX load request is made by depressing the CONTROL button on the teletypewriter and typing N or O on the teletypewriter. Use of N or O determines the point at which loading commences.

If CONTROL N is typed no overlay will occur. The program will be loaded immediately following the last foreground task program. Foreground processing is suspended while the program is being loaded. SPEX restores foreground processing automatically after completion of loading.

If CONTROL O is typed, overlay will be used. SPEX will dequeue all foreground tasks, delay until any I/O in process is completed, and then load the program over the foreground Q packets and foreground task programs.

If overlay is used, the user will typically take whatever steps are necessary to terminate and disconnect any processes

being controlled by the foreground workers before initiating SPEX loading. To reinitiate foreground processing, the core resident monitor and foreground workers must be reloaded.

Overlay is useful to those whose computer memory is not large enough to allow background programs to be loaded after the foreground workers. Overlay may also be desirable when the user does not want a non-debugged program executing in the background to possibly interfere with foreground processing.

SPEX uses the loader program which loaded the monitor and workers. RTM-I and SPEX allow a background program to use all of the memory from the last location occupied by the background program to the highest numbered core location possible. Several of the background task programs discussed in Section V make use of this feature.

4-7 BATCH.

BATCH will process a job stream input as a background operation, using whatever CPU time the real time foreground programs do not require. BATCH will read each control card from the card reader and retrieve and execute the Task Processor named on the card. BATCH does not overlay any foreground programs. As with SPEX, BATCH task processors may use high numbered core locations for data storage.

Batch processing is initiated by depressing the CONTROL button on the teletypewriter and typing B on the teletypewriter. BATCH will then attach the card reader and disc and begin execution. BATCH execution is terminated by a card with $ EOB in columns 1-5 or by execution of the Terminate Task control function by some task processor. At this time, the card reader and disc are released.

NOTE

Most task processors will use the Return Control to BATCH control function to terminate their execution. The Terminate Task and Return Control to BATCH options are discussed further in Section VI.

4-7.1 CONTROL CARDS. Control cards have the following format:

Column 1	$
Column 2	Blank
Column 3-8	The name of the program on the disc to be retrieved and executed (left adjusted)
Column 13-71	The options, if any for the retrieved programs.

4-7.2 THE SYSTEM DISC. The disc memory is subdivided logically in RTM-I/BATCH operation into the following areas:

a. RTM-I Core Image

b. Work Area reserved for Foreground Tasks

c. Catalog of Background Task Processors

d. Task Processors for BATCH Background

e. Work Area reserved for Background Tasks.

The boundaries between areas a, b, c, and d are initially set at system generation time. The boundary between areas d and e changes dynamically as the INSTAL, DELET, and COMPRS task processors are used to add and remove user programs in area d.

Areas a, c, and d are protected from access by non-system routines by the disc I/O service subprogram. If more than one foreground task is using the area b, it is the users responsibility to resolve any conflicts in the use of the available storage area. Area b is accessed using call codes 0 and 1.

Area e is accessed by the use of any of the call codes of the disc I/O. Call codes 2 through 5 allow logical partitioning of area e into four files.

4-7.3 BATCH INITIALIZATION. The BATCH object includes three separate but interrelated programs:

a. The BATCH processor itself

b. The INSTAL task processor

c. The system disc initialization program.

Only the BATCH processor is core resident. The other two programs are overlaid by background task processors.

The disc initialization program will perform initializations of the areas a, c, and d as described in paragraph 4-7.2. It will then initiate normal monitor processing.

The first thing which should be done on any RTM-I/BATCH system after loading and initiating execution of the monitor is to install background task processors on the disc. This is done with the $ INSTAL control card as described in Section VI. After installing all desired background task processors, systems which include magnetic tape drives should save a copy of the disc by use of the $ SAVED control card. After completion of this operation normal processing may begin.

4-8 SYSTEM GENERATION.

The basic monitor will be supplied with external references so that the monitor can be linked to the users foreground task programs, the 'Q' packets, the I/O device tables and Service Request Subprograms.

The linkage sequence is defined below:

a. Supervisor Object; required.

b. Device Table Definition Objects for Each Device. These objects result from an assembly of the input given in Figure 4-5. Only devices actually used need be defined.

c. Service Request Subprogram Objects; as required.

d. SPEX Object; optional.

e. Work List Definition Object. This object results from an assembly of the input given in Figure 4-6. Each foreground task must be defined in priority order. Background is not listed.

f. Foreground Task Objects. Each task should indicate its name (entry point) in a DEF statement.

g. BATCH Object; optional. SPEX and BATCH cannot both be included although both can be left out.

h. Block Data for Foreground COMMON Areas; optional.

The link editor will require the name of the main program to be specified as a parameter. For system generation of a SPEX system, the name R.TMX must be used. For system generation of a BATCH system the name R.TMB must be used.

At the completion of the system generation link edit several symbols will be noted as unresolved external references by the link editor. Some of the symbols which may appear and their interpretation are noted below. If any other symbol appears the system generation is faulty. The elements linked to form the system should be checked for accuracy.

The symbols noted below may or may not be valid unresolved external references. The validity is a function of what is included in the system being generated. The appearance of any unexpected symbol in the list of unresolved external references is an indication of a faulty system generation.

The possibly valid undefined external references are:

D.VxxA
xx = 01-24
D.VxxB

Device xxA or xxB not present in system.

C.Fxx xx = 08-32

Control function xx not in system.

N.PQ

X.PQ1

No workers in system.

N.DS

Workers require no DISC area.

D.MAP

B.ATCH

BATCH option not in system.

S.PEX

SPEX option not in system.

4-9 RESTORE DISC.

The restore disc program is a stand-alone program for regenerating the BATCH disc from a previously generated copy of the disc contents.

The save disc utility should be used anytime a permanent disc content has been built. This utility will save the disc contents except for the background work area on magnetic tape.

If, at some future time, the disc contents are destroyed, they may be restored using the restore disc program without the necessity of performing a system generation. This step may be followed by the reload from disc operation. The restore disc program is coded to execute stand-alone on the assumption that, if the disc contents have been destroyed, the monitor has probably also been destroyed.

The following procedure is used:

a. Load and execute the restore disc program.

```
NAME        OPERATION     OPERAND          ITEM

            DEF           D.V□             -FILL IN XXA ØR XXB -XX=01 TØ 24 DEVICE NUMBER
D.V□        EQU           $                -SAME AS ABØVE, 01A-08B=DMAC,09A-24B=DATA BUS
            REF           [______]         -HANDLER NAME
            DATA          [______]         -ADDRESS FØR STATUS WØRD, DMAC > 98- > A7, BUS=ANY
            DATA          >[_____]         -HEX VALUE FØR IMPØSSIBLE STATUS, LIKELY FFFF
            DATA          >[_____]         -EQUIPMNT ASSGNMNT BIT- > 1, 2, 4, 8, 10, ..., 4000
.                                           8000 ILLEGAL, USE 0 FØR NØ ASSGNMNT
            DATA          DATAS            -ADDRESS FØR DATA STØRAGE BY HANDLER
            DATA          [______]         -NAME ØF USER HANDLER
            DATA          FLAG             -ADDRESS ØF DEVICE BUSY FLAG
            DATA          STACK            -ADDRESS ØF REQUEST QUEUE CØNTRØLS
            DATA          0                -SLØT FØR ADDRESS ØF TASK'S SERVICE REQUEST
            DATA          0                -RESERVED FØR FUTURE
            DATA          □                -0=CALL HANDLER FRØM SRVC RQST SCAN, 1=ALSØ X-LEVEL
            DATA          [__]             -LØGICAL DEVICE NUMBER FØR THIS DEVICE
            DATA          >[_____]         -USE 0 FØR DMAC DEVICE, DEVICE ADDR FØR DATA BUS
STACK       DATA          1,0,0,0           QUEUE CØNTRØLS
            DATA          □                -DECIMAL SIZE FØR REQST STACK IN ENTRIES
            BSS           □                -DECIMAL SIZE FØR REQST STACK IN ENTRIES
DATAS       BSS           □                -DECIMAL SIZE ØF DATA STØRAGE FØR HANDLER
FLAG        DATA          0                -DEVICE BUSY FLAG
            END
```

Figure 4-5. Device Table Set Up

```
NAME        OPERATION     OPERAND          ITEM

            DEF           N.PQ,X.PQ1,N.DS
N.PQ        DATA          □                -NUMBER ØF Q-PACKETS IN LIST
N.DS        DATA          □                -NUMBER ØF DISC SECTØRS RESERVED FØR FØREGRØUND
X.PQ1       EQU           $
.*REPEAT NEXT SECTIØN FØR EACH FØREGRØUND Q-PACKET***********************
.                                                                       *
            REF           [_____]          -FILL IN TASK PRØGRAM NAME   *
            DATA          0,0,1              IN EACH PØSITIØN           *
            DATA          [______]                                      *
            DATA          [_______]                                     *
            DATA          0,0,,,,,,,,,0                                 *
.                                                                       *
.************************************************************************
.
            END
```

Figure 4-6. Work List Set Up

b. The program will type a message requesting the disc copy to be placed on the magnetic tape drive.

c. The program will transfer the contents of the tape to the disc.

d. The program will idle. If it is desired to load the core resident programs from the disc, press the RUN pushbutton.

e. After loading core memory, control is transferred to the monitor.

SECTION V

SPEX OPERATING PROCEDURES

SECTION V

SPEX OPERATING PROCEDURES

5-1 GENERAL PROCEDURES.

SPEX (Single Program Executive) is an optional background control program supplied with RTM-I. It executes a single background task processor. SPEX is used on Model 980 systems which do not have a disc memory and a card reader.

All background task processors supplied by Texas Instruments and run under control of RTM-I/SPEX input program options similarly. Once the program is loaded, it will typewrite a request for appropriate option specifications. If a card reader is part of the computer system, the options are entered by placing a card in the format of a BATCH control card in the card reader (refer to Section VI). If no card reader is available, the teletypewriter is used.

Most options are specified by single alphanumeric characters. The characters used and their meanings are:

1	Magnetic tape drive No. 1
2	Magnetic tape drive No. 2
3	Magnetic tape drive No. 3
C	Cards
D	Disc
E	End of Task
G	Load and go
L	Line printer
N	Not used or None
P	High speed paper tape
T	Teletypewriter

When the input is read from cards, the options may be specified by a card using the BATCH control format. Columns 13, 15, 17, ..., 79 are used for single letter options. The columns between option letters are usually punched with a comma to aid readability. However, the task processors do not interpret these columns nor do they interpret columns 1 - 12. If an option specification requires more than one column, the specification will always start in a specific column (refer to paragraph 5-2).

When the input is read from the teletypewriter unit, the options are always separated by commas and terminated by a carriage return. Do not use any characters other than the option specifiers, commas, and the carriage return. Blanks are disallowed, except to specify a default option.

Options may be omitted. If omitted, the various task processors will select a default option automatically. If an invalid option is specified, the task processor will automatically use the default option as if no option had been specified.

The default options for any of the task processors are defined by a "default sequence." The default sequences are:

Input
- Cards
- Paper Tape – High Speed Reader
- Paper Tape – Teletypewriter

Punched Output
- Paper Tape – High Speed Punch
- Paper Tape – Teletypewriter

Printed Output
- Line Printer
- Teletypewriter

These sequences are used as follows: Input is read from cards if a card reader is available. If not the high speed paper tape reader is used. If a high speed paper tape reader is not available the teletypewriter is used. Punched output and printed output are treated in a similar manner.

Any input/output devices required by the task processor will be attached as they are needed. All attached input/output devices are released when execution is complete.

5-2 SYMBOLIC ASSEMBLY PROGRAM (SAP-I).

5-2.1 USE OF THE CARD READER TO SPECIFY OPTIONS.

SAP-I options are as follows:

Source (Card Column 13)

C	Cards
P	Paper Tape – High Speed Reader
T	Paper Tape – Teletypewriter
E	End of Assemblies

Object (Card Column 15)

P	Paper Tape – High Speed Punch
T	Paper Tape – Teletypewriter
N	No Object Output

N No Listing

Listing (Card Column 17)
- L Line Printer
- T Teletypewriter

Additional options will become available in the future. Card columns 18 - 80 are not interpreted by SAP-I.

Example of SAP-I Option Cards:

```
$  SAPI
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
```

Card I will produce the default options.

```
$  SAPI        P,P,N
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
```

Card II will produce the following:

Source	High Speed Paper Tape
Object	High Speed Paper Tape
List	None

```
$  SAPI        T,P,L
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
```

Card III will produce the following:

Source	Teletypewriter
Object	High Speed Paper Tape
List	Line Printer

5-2.2 USE OF THE TELETYPEWRITER TO SPECIFY OPTIONS. If a card reader is not available, then the options are entered on the teletypewriter. The same results as noted in paragraph 5-2.1 would be produced by the following:

Example I	C/R
Example II	P,M,NC/R
Example III	T,P,L,NC/R

5-2.3 OPERATING PROCEDURE. Take the following steps to perform a SPEX assembly.

a. Load SAP-I.

Result: ENTER OPTIONS is printed on the teletypewriter.

b. Enter the program control card if the card reader is available. If not, type options on the teletypewriter followed by a C/R.

Result: The following message is printed on the teletypewriter: READY ASSEMBLY SOURCE AND HIT C/R.

c. Ready the source and hit carriage return.

Result: Pass one is made. If there are pass one errors, they are printed. The above message is printed again.

d. Ready source and hit carriage return.

Result: ENTER OPTIONS is printed upon completion of the assembly.

e. If more assemblies are desired, enter options and continue with step b, otherwise, enter the end option.

Result: After completion of all assemblies, control is returned to RTM-I/SPEX.

5-3 FORTRAN.

A FORTRAN compiler running under RTM-I control is not available as part of the initial release of RTM-I but will be available in a later release.

5-4 LINK EDIT.

5-4.1 USE OF THE CARD READER TO SPECIFY OPTIONS. Link edit options are as follows:

Input Object Programs (Card Column 13)
- C Cards
- P Paper Tape – High Speed Reader
- T Teletypewriter
- E End of Link Editing

Output Object Programs (Card Column 15)
- P Paper Tape – High Speed Punch
- T Teletypewriter
- N No Object Output

Load Map Output (Card Column 17)
- T Teletypewriter

Load and Go (Card Column 19)
- N No Load and Go.
- G Load and Go: Execute the Link Edited Program

Additional options will become available in the future.

No load and go will be a default option for the link editor.

In addition to the options, three items are required. The number of programs being link edited must be listed in card columns 25-26. The name of the main program of the set of programs being link edited must be listed in card columns 31-36. The execution entry address of this program is used as the execution entry address of the link edited program. The name of the output object program must be listed in card columns 41-46. If the required items do not occupy all the columns allotted, they must be left justified in their field, i.e., start in columns 25, 31, or 41. Other columns of the card are not interpreted by the link editor. FORTRAN main programs have the name MAIN.

Examples of link editor cards:

```
$ LINKED                  5     MAIN1     MAIN2
000000000000000000000000000000000000000000000000
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48
111111111111111111111111111111111111111111111111
222222222222222222222222222222222222222222222222
```

Card I specifies all default options. The name of the main program is MAIN1. The name of the link edited program is MAIN2.

```
$ LINKED        P,P,T,G   5     MAIN1     MAIN2
000000000000000000000000000000000000000000000000
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48
111111111111111111111111111111111111111111111111
222222222222222222222222222222222222222222222222
```

Card II uses the following options:

Input	High Speed Paper Tape
Output	High Speed Paper Tape
Load Map	Teletypewriter
Load and Go	Used

```
$ LINKED        T,P,T     5     MAIN1     MAIN2
000000000000000000000000000000000000000000000000
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48
111111111111111111111111111111111111111111111111
222222222222222222222222222222222222222222222222
```

Card III uses the following options:

Input	Teletypewriter
Output	High Speed Paper Tape
Load Map	Teletypewriter
Load and Go	Default, i.e., No Load and Go

5-4.2 USE OF TELETYPEWRITER TO SPECIFY OPTIONS. If a card reader is not available, the options are typed on the teletypewriter. The same results as before would be produced by the following:

a. 5,MAIN1,MAIN2C/R
b. 5,MAIN1,MAIN2,P,P,T,GC/R
c. 5,MAIN1,MAIN2,T,P,TC/R

5-4.3 OPERATING INFORMATION. Object programs may be used in any order as input, subject only to the following restrictions.

Absolute object programs must be read into the link editor ahead of any relocatable object programs. The programmer is responsible to assure that absolute programs do not require the same core locations. If both absolute and relocatable programs are link edited together, the relocatable programs will be located after the absolute program using the highest numbered core locations. The output program will be absolute if one or more absolute programs are included in the link edit. If no input programs are absolute, the output will be relocatable.

A program with undefined external references may be loaded and executed if the undefined external references are not used.

During execution of the link editor first pass the load blocks are not read. The only required information is obtained from the tape or cards preceeding the load blocks. During pass two the complete object program input is read.

If the load and go option is used, the link editor builds a relocated program in core and transfers control to this program. This program is restricted as follows. The program, link editor, core resident programs, and the link editor generated tables must be loaded simultaneously. The link editor uses the area immediately following itself for symbol tables. The program for execution must be loadable into the area following the link editor which is not used for tables.

No more link edits may be performed after execution of a link edit with load and go. It is the responsibility of the program executed via load and go to return control to RTM-I by use of the Terminate Task control service request (code = 0).

Before transferring control to the program being executed, the link editor will release all peripherals it has assigned.

5-4.4 OPERATING PROCEDURE. The following steps are taken to perform a SPEX link edit.

a. Load the link editor.

Result: ENTER OPTIONS is printed with the teletypewriter.

b. Enter the program control card if the card reader is available. If not, type options on the teletypewriter followed by a carriage return.

Result: The following message is printed on the teletypewriter: READY LINK EDIT INPUT AND HIT C/R.

c. Ready input and hit C/R.

Result: The input is read and again the message READY INPUT AND HIT C/R is typed. This step is repeated until all input object programs have been read. At that time, step d is executed.

d. Ready input and hit C/R. All input to pass two must be in the same order as input to pass one.

Result: The input is read and again the message READY INPUT AND HIT C/R is typed. This step is repeated until all input object programs have been read. If mass storage is used, this step is omitted.

e. Ready the last input object program and hit carriage return.

Result: After reading the last input object program, ENTER OPTIONS is again printed on the teletypewriter if the load and go option was not selected. If it was, the message STARTING EXECUTION is typed and step f is omitted.

f. If more link edits are desired, enter options and continue with step b. Otherwise, enter the end option.

Result: After completion of all (non load and go) link edits control is returned to RTM-I/SPEX.

5-5 CORRECT SOURCE.

5-5.1 PROGRAM OPTIONS. No options are available currently for the Correct Source Task Processor. The only version available reads input source on the high speed paper tape reader and corrections on the same device. Output source is punched on the high speed paper tape punch. Additional options will become available in the future.

5-5.2 OPERATING INFORMATION. The corrections consist of a series of insert and delete commands plus lines for insertion. Insert commands have the format

+NUM1,NUM2

where NUM2>NUM1 are integer line numbers. An insert command causes all lines from the current line to NUM1, including NUM1, to be copied. All lines from NUM1 to NUM2, not including NUM2, are deleted. All lines following the insert command until the next command line will be inserted. After completion of processing the insert command, NUM2 becomes the current line number.

Delete commands have the format

-NUM1,NUM2

where NUM2>NUM1 are integer line numbers. The delete is the same as insert except no insertions are made.

Initially, the current line number is set to line 1. All correction commands must be in numerical order. After the last command, the remaining source is copied.

The correct source utility may be used to copy by inputing one insert command where NUM1=1 and NUM2=2.

5-5.3 OPERATING PROCEDURE. The following steps are taken in order to perform a tape correction under SPEX.

a. Load correct program.

Result: READY CORRECTIONS AND HIT C/R is typed on the teletypewriter.

b. Ready correction tape and hit carriage return.

Result: The correction tape is read and READY SOURCE FOR CORRECTION AND HIT C/R is typed on the teletypewriter.

c. Ready source and hit carriage return.

Result: The corrected tape is output. The message READY CORRECTIONS AND HIT C/R is again typed on the teletypewriter and another source tape may be corrected starting with step b. To terminate the correction process, type E followed by a carriage return on the teletypewriter.

5-6 LIST SOURCE.

5-6.1 USE OF THE CARD READER TO SPECIFY OPTIONS. Listing options are as follows:

Source (Card Column 13)
- C Cards
- P Paper Tape – High Speed Reader
- E End of Listing Task

Output Listing (Card Column 15)
- L Line Printer
- T Teletypewriter

Additional options will become available in the future. Card columns 16-80 are not interpreted by the listing program.

Examples of listing option cards.

```
$ LISTS
```

Card I will produce the default options.

```
$ LISTS     P,T
```

Card II will produce the following:

Source	Paper Tape
Listing	Teletypewriter

5-6.2 USE OF THE TELETYPEWRITER TO SPECIFY OPTIONS. If a card reader is not available, the options are typed on the teletypewriter. The same results as above would be produced by the following:

a. C/R
b. P,TC/R

5-6.3 OPERATING PROCEDURE. The following steps are taken in order to obtain a listing under SPEX:

a. Load the listing program.

Result: ENTER OPTIONS is printed on the teletypewriter.

b. Enter the program control card if the card reader is available. If not, type options on the teletypewriter followed by a carriage return.

Result: The following message is printed on the teletypewriter: READY LISTING SOURCE AND HIT C/R.

c. Ready source and hit carriage return.

Result: The listing is produced. At the completion, the ENTER OPTIONS message is again printed.

d. If more listings are desired, enter options and continue with step b. Otherwise, enter the end of listing option.

Result: After completion of all listings desired control is returned to RTM-I.

5-7 USER PROGRAM EXECUTION.

Any user program may be loaded and executed by SPEX. To load and execute any program using the SPEX background processor, place the object program in the appropriate reader, depress the CONTROL key on the teletypewriter, and type N or O on the teletypewriter. SPEX uses the loader program which was used to load the monitor. The options CONTROL N or CONTROL O specify no overlay or overlay (see Section IV).

The user program may use all of memory from the last instruction in itself to the highest numbered core location available. The size of memory may be obtained by reading core location $001A_{16}$.

Any input/output devices required by the program should be attached before use (refer to Section IV). The FORTRAN input/output programs perform this automatically if they are used.

The Terminate Task control service request (Code = 0) will return control to the monitor. From FORTRAN programs, control is returned by the STOP command.

SECTION VI

BATCH OPERATING PROCEDURES

SECTION VI

BATCH OPERATING PROCEDURES

6-1 GENERAL PROCEDURES.

BATCH is an optional background control program supplied with RTM-I. BATCH reads a job control language input from the card reader. It then calls and executes the appropriate processor programs from the system disc.

All Texas Instruments supplied background task processors which run under RTM-I/BATCH use the same general approach for specifying program options. Options are specified in columns 13-80 of the control card which requests execution of the background task processor.

Most options are specified by single alphanumeric characters. The characters used and their meanings are:

1 Background Work Area File 1 or Magnetic Tape Drive 1

2 Background Work Area File 2 or Magnetic Tape Drive 2

3 Background Work Area File 3 or Magnetic Tape Drive 3

4 Background Work Area File 4

C Cards

D Disc

L Line Printer

N Not Used or None

P High Speed Paper Tape

T Teletype

If single letter options are used they are in columns 13,15,17,...,79 of the card. Typically the columns between option letters are punched with a comma to aid readability, however the background task processors do not interpret these columns. If an option specification requires more than one column, the specification will always start in a given column.

Options may be omitted. If omitted, the various task processors will select a default option automatically. If an option is specified but found to be invalid, the task processor will automatically use the default option as if no option had been specified.

The default options for the various task procressors are:

Input
Cards

Punched Output
High Speed Paper Tape

Printed Outout
Line Printer if available, otherwise, Teletypewriter

Any I/O devices required by the task processor will be attached as they are needed. At the completion of execution all attached I/O devices are released.

The task processors for BATCH will complete one task and return control to the monitor. This allows several tasks to operate sequentially. For example a job might consist of the following tasks: correct source, assemble corrected source, link edit the assembly with previous assemblies, and execute the link edited program. This differs from the approach used by SPEX task processors which are designed to repeat the same task as many times as required before the next task processor is loaded.

The following subsections describe the usage of each background task processor in an RTM-I/BATCH environment.

6-2 SAP-I.

6-2.1 CONTROL CARD. The program control card input to BATCH contains the following:

Card Column		
	1	$
	2	blank
	3	S
	4	A
	5	P
	6	I
	7-12	blank
	13	source option
	15	object option
	17	list option

SAP-I options are as follows:

Source (Card Column 13)
- C Cards
- P Paper Tape – High Speed Reader
- D Disc

Object (Card Column 15)

P	Paper Tape – High Speed Punch (and Disc)
D	Disc (only)

List Option (Card Column 17)

L	Line Printer
T	Teletypewriter
N	No List

Additional options will become available in the future. Card columns 18-80 are not interpreted by SAP-I.

Examples of control cards:

```
$ SAPI
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
```

Card I will produce the default options.

```
$ SAPI          P,N
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
```

Card II will produce the following:

Source	Default
Object	High Speed Paper Tape
List	None

```
$ SAPI        C,P,L
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
```

Card III will produce the default options.

6-2.2 OPERATING INFORMATION. If the card reader or disc is specified as the source input device, SAP-I will proceed immediately to read from these devices. If the high speed paper tape reader is used, the program will delay to allow the operator to ready the tape.

If the disc is the source input device, SAP-I will expect to find the input in disc file 2 of the background task work area. This file is used as the output file by the source correct and source copy utility programs.

If the disc is not the source input device, SAP-I will copy the source onto disc file 2 as part of the assembler first pass. Pass two of the assembler therefore always gets its input from disc file 2 of the background task work area.

The assembler always outputs the object program to disc file 3. If the high speed paper tape punch is listed as the object option, this output is in addition to the disc output.

6-2.3 OPERATING PROCEDURE. When using disc as the source input device the control card is placed in the cards for BATCH processing in the appropriate position. Similarly when using the card reader as the source input device except that the source cards must immediately follow the control card.

When using the high speed paper tape reader as the source input device the following procedure is used:

a. Enter control card through the card reader.

   Result: Program is loaded from disc and executed. A message is printed on the teletypewriter READY ASSEMBLY SOURCE AND HIT C/R.

b. Ready source and type a carriage return.

   Result: The assembly is performed.

6-3 FORTRAN.

A FORTRAN compiler running under RTM-I/BATCH control is not available as part of the initial release of RTM-I. It is planned for a later release.

6-4 LINK EDIT.

6-4.1 CONTROL CARD. The program control card input to the Link Editor contains the following:

Card Column	
1	$
2	blank
3	L
4	K
5	E
6	D
7-12	blank
13	input option
15	output option
17	load map option
19	link with disc option
25-26	number of programs being link edited
31-36	name of main program
41-46	name of output object program.

Link editor options are as follows:

Input Object Programs (Card Column 13)
- C Cards
- P Paper Tape – High Speed Reader

Output Object Programs (Card Column 15)
- P Paper Tape – High Speed Punch (and Disc)
- D Disc (only)

Load Map Output (Card Column 17)
- T Teletype

Link with Disc (Card Column 19)
- D Include program in disc file 3 in the link edit
- N Do not include disc file 3 in the link edit

Additional options will become available in the future.

No link with disc is a default option when link editing.

In addition to the options, three items are required. Card columns 25-26 must contain the number of programs being link edited. In card columns 31-36 must be listed the name of the main program of the set of programs being link edited. The execution entry address of this program will be used as the execution entry address of the link edited program. In card columns 41-46 must be listed the name of the output object program. If the required items do not occupy all the columns allotted, they must be left justified in their field, i.e., start in columns 25, 31, or 41. Other columns of the card are not interpreted by the link editor. Note that FORTRAN main programs have the name MAIN.

Examples of link editor control cards.

```
$ LINKED                 5     MAIN1     MAIN2
```

Card I will use all default options. The name of the main program is MAIN1. The name of the link edited program is MAIN2. Five programs are to be link edited.

```
$ LINKED      P,T        5     MAIN1     MAIN2
```

Card II will use the following options:

Input	Default, i.e., Cards
Output	Disc and High Speed Paper Tape
Load Map	Teletypewriter
Link With Disc	Default, i.e., No Link With Disc

```
$ LINKED      C,D,T,D    5     MAIN1     MAIN2
```

Card III will use the following options:

Input	Cards
Output	Disc (only)
Load Map	Teletypewriter
Link With Disc	Used

6-4.2 OPERATING INFORMATION. Object programs may be used in any order as input subject only to the following restrictions:

If there are any absolute object programs input to the link editor, they must be input before any relocatable object programs. It is the programmer's responsibility to assure that absolute programs do not require the same core locations. If both absolute and relocatable programs are link edited together, the relocatable programs will be located after the program using the highest numbered core locations. The output program will be absolute if one or more absolute programs are included in the link edit. If no input programs are absolute, the output will be relocatable.

A program with undefined external references may be loaded and executed provided the undefined external references are not used.

If the card reader is specified as the input device, the link editor will proceed immediately to read from this device. If the high speed paper tape reader is used, the program will pause for each input tape to allow the operator to ready the input.

If the link with disc option is used, the link editor will expect to find one object program in disc file 3 of the background work area. In any case all input object programs are added to disc file 3 as part of the link editor first pass. Thus pass 2 of the link editor will use only the disc for input.

The link editor always outputs to disc file 4 of the background work area. If the high speed paper tape punch is listed as the output option, this output is in addition to the disc output.

6-4.3 OPERATING PROCEDURE. When using the card reader as the input device the input object program cards must immediately follow the control card. When using the high speed paper tape reader as the source input device the following procedure is used:

a. Enter control card through the card reader.

   Result: Program is loaded from disc and executed. A message is printed on the teletypewriter READY LINK EDIT INPUT AND HIT C/R.

b. Ready input and type a carriage return.

   Result: The input is read and saved on disc. Step b is repeated until all tapes have been input.

6-5 CORRECT SOURCE.

6-5.1 CONTROL CARD. The program control card input to BATCH contains the following:

Card Column	
1	$
2	blank
3	C
4	O
5	R
6	C
7	T
8-12	blank
13	input option
15	output option
17	corrections option

The correct source options are as follows:

Input (Card Column 13)

- C Cards
- P Paper Tape – High Speed Reader
- 1 Magnetic Tape Drive 1
- 2 Magnetic Tape Drive 2
- 3 Magnetic Tape Drive 3

Output (Card Column 15)

- P Paper Tape – High Speed Punch (and Disc)
- D Disc (only)
- 1 Magnetic Tape Drive 1 (and Disc)
- 2 Magnetic Tape Drive 2 (and Disc)
- 3 Magnetic Tape Drive 3 (and Disc)

Corrections (Card Column 17)

- C Cards
- P Paper Tape – High Speed Reader

Additional options will become available in the future. Card columns 18-80 are not interpreted by the correct source program.

Examples of control cards:

```
$ CORCT
```

Card I specifies all default options.

```
$ CORCT         P
```

Card II will produce the following:

Input	Default
Output	High Speed Paper Tape (and Disc)
Corrections	Default

```
$ CORCT         1,2,C
```

Card III will produce the following:

Input	Magnetic Tape Drive 1
Output	Magnetic Tape Drive 2 (and Disc)
Corrections	Cards

6-5.2 OPERATING INFORMATION. The corrections consist of a series of insert and delete commands plus source lines for insertion.

Insert commands have the format

+NUM1,NUM2

where NUM2>NUM1 are integer line numbers. An insert command causes all lines from the current line to NUM1, including NUM1, to be copied. All lines from NUM1 to NUM2, not including NUM2 are deleted. All lines following the insert command until the next command line will be inserted. After completion of processing the insert command, NUM2 becomes the current line number.

Delete commands have the format

-NUM1,NUM2

where NUM2>NUM1 are integer line numbers. The delete is the same as insert except no insertions are made.

Initially the current line number is set to line 1. All correction commands must be in numerical order. After the last command the remaining source is copied.

Corrections are read by the correct source program first. These are stored on disc file 1 of the background work area. The source program to be corrected is read next and the corrected source program stored on disc file 2 of the background work area.

If corrections are on cards the correct source program will proceed immediately to read these cards. If corrections are on paper tape the program will delay to allow the operator to ready the tape.

If the program to be corrected is on cards the correct source program will proceed to read these cards after finishing reading the corrections. If not it will delay to allow the operator to ready the tape.

The correct source program always leaves the corrected source program on disc file 2. If specified it will produce a second output on the high speed paper tape punch or magnetic tape.

If both corrections and source to be corrected are on cards it is necessary to place a separator card between the two input card decks. The separator card should have the currency symbol ($) punched in column 1. The remainder of the card is ignored by the correct program.

If the output device is the high speed paper tape punch, the program will monitor the tape status for low tape. This status indicates approximately 50 feet of tape remain on the reel. The program will punch an additional 40 feet of tape and then punch a line with an equals (=) in column one. This indicates the program is continued on another reel of tape. The program will then pause to allow the operator to mount a new tape reel.

6-5.3 OPERATING PROCEDURE. If corrections or the source program to be corrected are on cards these should immediately follow the control card. The corrections precede the source program if both are on cards. No special procedure is required in this case. If not the following procedure is used:

a. Enter control card through the card reader.

   Result: Program is loaded from disc and executed. If corrections are not on cards the message READY CORRECTIONS AND HIT C/R is typed. If corrections are on cards they are read and step c is the next step.

b. Ready corrections and type carriage return.

   Result: The corrections are read.

c. The program is now ready for the input source.

   Result: If the input source program is on cards they are read and the output generated. If not the message READY SOURCE FOR CORRECTION AND HIT C/R is typed.

d. Ready source for correction and hit carriage return.

   Result: The remainder of the processing is performed.

6-6 LIST SOURCE.

6-6.1 CONTROL CARD. The program control card input to BATCH contains the following:

Card Column	1	$
	2	blank
	3	L
	4	I
	5	S
	6	T
	7	S
	8-12	blank
	13	source option
	15	list option

The list source options are as follows:

Source (Card Column 13)

C	Cards
P	Paper Tape – High Speed Reader
D	Disc
1	Magnetic Tape Drive 1
2	Magnetic Tape Drive 2
3	Magnetic Tape Drive 3

List (Card Column 15)

L Line Printer
T Teletypewriter

Additional options will become available in the future. Card columns 16-80 are not interpreted by the listing program.

Examples of control cards:

```
$ LISTS
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
```

Card I will produce the default options.

```
$ LISTS       C,T
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
```

Card II will produce the following:

Source	Cards
Listing	Teletypewriter

6-6.2 **OPERATING INFORMATION.** If the card reader or disc is specified as the source input device, the listing program will proceed immediately to read from these devices. If not, the program will delay to allow the operator to ready the tape.

If the disc is specified as the source input device, the program will expect to find the source program on disc file 2 in the background work area.

6-6.3 **OPERATING PROCEDURE.** When using the disc as the source input device the control card is placed in the cards for BATCH input in the appropriate position. Similarly when using the card reader as the source input device except the source cards must immediately follow the control card.

When source input is not on cards or disc the following procedure is used:

a. Enter control card through the card reader.

Result: Program is loaded from disc and executed. A message is typed on the teletypewriter, READY LISTING INPUT AND HIT C/R.

b. Ready listing input and type carriage return.

Result: The listing is produced.

6-7 COPY SOURCE.

6-7.1 **CONTROL CARD.** The program control card input to BATCH contains the following:

Card Column		
	1	$
	2	blank
	3	C
	4	O
	5	P
	6	Y
	7	S
	8-12	blank
	13	input source option
	15	output source option

The copy source options are as follows:

Input Source (Card Column 13)

C Cards
P Paper Tape – High Speed Reader
D Disc
1 Magnetic Tape Drive 1
2 Magnetic Tape Drive 2
3 Magnetic Tape Drive 3

Output Source (Card Column 15)

P Paper Tape – High Speed Punch (and Disc)
D Disc (only)
1 Magnetic Tape Drive 1 (and Disc)
2 Magnetic Tape Drive 2 (and Disc)
3 Magnetic Tape Drive 3 (and Disc)

Additional options will become available in the future. Card columns 16-80 are not interpreted by the copy source program.

Examples of control cards:

```
$ COPYS
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
```

Card I will produce the default options.

```
$ COPYS       1,D
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
```

Card II will produce the following:

Input Source	Magnetic Tape Drive 1
Output Source	Disc

6-7.2 OPERATING INSTRUCTIONS. If the card reader or disc is specified as the source input device, the copy source program will proceed immediately to read from these devices. If not the program will delay to allow the operator to ready the tape.

The copy source program always leaves the source program on disc file 2 of the background work area. If specified it will produce a second output on the high speed paper tape punch or magnetic tape.

Note that specifying disc as both the source input and output device is illegal.

If the output device is the high speed paper tape punch, the program will monitor the tape status for low tape. This status indicates approximately 50 feet of tape remain on the reel. The program will punch an additional 40 feet of tape and then punch a line with an equals (=) in column one. This indicates the program is continued on another tape. The program will then pause to allow the operator to mount a new tape reel.

6-7.3 OPERATING PROCEDURE. When using the disc as the source input device the control card is placed in the cards for input to BATCH in the appropriate position. Similarly when using the card reader as the source input device except the source cards must immediately follow the control card.

When source input is not on cards or disc the following procedure is used:

a. Enter control card through the card reader.

   Result: Program is loaded from disc and executed. A message is typed on the teletypewriter, READY SOURCE FOR COPYING AND HIT C/R.

b. Ready source for copying and hit carriage return.

   Result: The copy is produced.

6-8 COPY OBJECT.

6-8.1 CONTROL CARD. The program control card input to BATCH contains the following:

Card Column	1	$
	2	blank
	3	C
	4	O
	5	P
	6	Y
	7	O
	8-12	blank
	13	input object option
	15	output object option

Only one option is provided in the initial release. The copy object program will copy from the high speed paper tape reader to the high speed paper tape punch. Thus columns 13 and 15 may contain either the character P or blank.

Additional options will become available in the future. Card columns 16-80 are not interpreted by the copy object program.

6-8.2 OPERATING PROCEDURE. The following steps are taken in order to copy object tape:

a. Enter control card through the card reader.

   Result: Program is loaded from disc and executed. A message is typed on the teletypewriter, READY OBJECT TAPE FOR COPYING AND HIT C/R.

b. Ready object tape for copying and type carriage return.

   Result: The copy is produced.

6-9 INSTALL.

6-9.1 CONTROL CARD. The program control card input to BATCH contains the following:

Card Column	1	$
	2	blank
	3	I
	4	N
	5	S
	6	T
	7	A
	8	L
	9-12	blank
	13	input option

Only one option is provided in the initial release. The install program will use the high speed paper tape reader as an input device. Thus column 13 may contain either the character P or blank.

6-9.2 OPERATING INFORMATION. The install program is used to catalog a program for use by the BATCH processor. When catalogued the program is stored on disc with the other task processors for background BATCH processing. Its name is recorded in the catalog of task processors. It may then be executed by placing a card in the BATCH card input containing $ in column 1, blank in column 2 and the program name in columns 3-8.

6-9.3 OPERATING PROCEDURE. The following steps are taken in order to install a program in the BATCH background task processor disc file:

a. Enter control card through the card reader.

   Result: Program is loaded from disc and executed. A message is typed on the teletypewriter, READY OBJECT TAPE FOR INSTALLATION AND HIT C/R.

b. Ready object tape for installation and type carriage return.

   Result: The program is installed and catalogued.

6-10 DELETE.

6-10.1 CONTROL CARD. The program control card input to BATCH contains the following:

Card Column	1	$
	2	blank
	3	D
	4	E
	5	L
	6	E
	7	T
	8	E
	9-12	blank
	13-18	name of program to be deleted.

6-10.2 OPERATING INSTRUCTIONS. Delete will cause a previously installed program to be flagged for removal from the background task processor file. Delete does not remove the program from the disc. In order to remove the program from the disc the compress program must be executed. Once flagged by the delete program a task processor may not be used. An exception to this occurs if the deleted program is the last one currently catalogued. In this case the compress program need not be used.

The name of the program to be deleted must start in column 13 of the control card.

6-11 COMPRESS.

6-11.1 CONTROL CARD. The program control card input to BATCH contains the following:

Card Column	1	$
	2	blank
	3	C
	4	O
	5	M
	6	P
	7	R
	8	S
	9-12	blank

6-11.2 OPERATING INSTRUCTIONS. Compress will physically remove from the disc any background task processors flagged by Delete. The remaining programs will be placed starting in adjacent disc sectors such that a minimum disc area is used for storage of these programs. The catalog and disc map are updated to reflect the new disc content.

6-12 LIST CATALOG.

6-12.1 CONTROL CARD. The program control card input to BATCH contains the following:

Card Column	1	$
	2	blank
	3	L
	4	I
	5	S
	6	T
	7	C
	8-12	blank
	13	list option

The list options are as follows:

L	Line Printer
T	Teletypewriter

Card columns 14-80 are not interpreted by the list catalog program.

6-12.2 OPERATING INSTRUCTIONS. The list catalog program produces a printout of the names of those programs included in the background task processor file. Deleted programs are not listed. The presence of deleted programs on the disc is however noted on the listing.

6-13 SAVE DISC.

The program control card input to BATCH contains the following:

Card Column		
	1	$
	2	blank
	3	S
	4	A
	5	V
	6	E
	7	D
	8-12	blank
	13	output option

Only one option is provided in the initial release. The save disc program will copy the contents of the disc (except for the background work area) onto magnetic tape drive 1. Thus column 13 may contain either the character 1 or blank.

Additional options will become available in the future. Card columns 14-80 are not interpreted by the copy object program.

6-14 SYSTEM CONTROL CARDS.

In addition to those control cards used to load and execute background task processors from the disc, certain control cards are interpreted and processed directly by BATCH.

6-14.1 **JOB CARD.** This card contains the following:

Card Column		
	1	$
	2	Blank
	3	J
	4	O
	5	B
	6-12	blank

The $ JOB Card is used to logically separate tasks into related groups. Columns 13-80 are not interpreted and may contain information identifying the group of tasks which follow.

6-14.2 **EOB CARD.** This card contains the following:

Card Column		
	1	$
	2	blank
	3	E
	4	O
	5	B
	6-12	blank

The $ EOB Card marks the end of the input on the card reader. Upon reading this card BATCH will release the disc and card reader and terminate.

6-14.3 **EXEC CARD.** This card contains the following:

Card Column		
	1	$
	2	blank
	3	E
	4	X
	5	E
	6	C
	7-12	blank
	13	file option
	15-80	user option

The possible file options are:

3 execute the program in disc file 3
4 execute the program in disc file 4

Upon detecting the $ EXEC card the BATCH processor will load and execute a program from the background work area. If the program to be executed was generated by an assembly, disc file 3 should be specified. If generated by a link edit, disc file 4.

The program executed in this manner interfaces with RTM-I/BATCH exactly like a program catalogued by the install program and loaded from the background task processor file. If the program executed by use of a $ EXEC card requires input read from cards, the cards should be placed immediately after the $ EXEC card.

6-15 INTERFACE BETWEEN BATCH AND THE TASK PROCESSOR.

Background task processors receive as input a 16 bit status word and the core image of that portion of the control card containing their options. Upon entry the L register contains the address of the following 3 words:

```
DATA 2
DATA address of status word
DATA address of control card options
```

When exiting by use of the Terminate Batch Processing Task control function the task processor must load a 16 bit status word in the M register.

If the background task processor is coded in FORTRAN it may communicate with the BATCH processor as follows:

a. The main program may be coded as a subroutine with two arguments, the status word and option string. For example

```
SUBROUTINE MAIN(ISTAT,IOP)
DIMENSION IOP(34)
```

If the FORTRAN program does not require these arguments it need not be coded as a subroutine.

b. Status may be returned to the monitor by use of the statement

```
STOP n
```

The number n is returned to the monitor as a status value. If n is omitted a status word of all zeroes is returned.

The status word is used to pass information from one task to the next. Its purpose is to direct the mode of processing employed by the next task. One common use is to inform a subsequent task of a situation in which processing by a subsequent task is to be aborted.

The BATCH processor tests for a status word with bit 15 set. This means unconditional abort. The BATCH processor will read cards until a card with $ JOB or $ EOB in columns 1-5 is encountered. Upon detection of the $ JOB card the BATCH processor returns to its normal mode of operation, execution of the task processor corresponding to each control card. Thus an unconditional abort status causes all following tasks up to the $ JOB card to be automatically aborted.

Bit 14 of the status word is set if a FORTRAN program terminates by reading through an end-of-file. Bits 8-13 are available for use by user programs. Bits 0-7 are reserved for system use.

6-16 SEQUENCES OF TASKS.

The background work area on the disc is logically divided into four files. This structure is useful in performing sequences of tasks. For example, consider the following common sequence.

a. Correct a source program. The correct program will read corrections into file one and place the corrected source program in file two.

b. Assemble the corrected source program. The SAP-I assembler will assemble the source program in file two and place the object program in file three.

c. Link edit the object program with other object programs. The link edit will add other object programs to file three and then place the link edited program in file four.

d. Execute the link edited program. The BATCH processor will load the program in file four into the computer memory and initiate execution.

e. Copy the corrected source onto magnetic tape. The copy program will copy the source in file two onto magnetic tape. This magnetic tape may then be used as input to repeat this sequence.

SECTION VII

BOOTSTRAPS AND LOADERS

SECTION VII

BOOTSTRAPS AND LOADERS

7-1 INTRODUCTION.

Bootstraps and loaders are used to load the monitor and initiate execution. These programs remain core resident and are used by the background job control program SPEX to load task processors. They may be used also to load stand alone programs.

A bootstrap is a short sequence of instructions which must be manually entered into the computer. The bootstrap, when executed, causes the computer to read a more sophisticated loader program from a specific peripheral device. There is a separate bootstrap and loader program set for each peripheral input device. The loader program loads an object program in the assembly format from the same input device from which the loader itself was loaded (Figure 7-1). Once loaded into memory, the bootstrap and loader programs need not be reloaded. They may be used repeatedly, provided they are not overwritten by some user task program.

The loaders do not resolve COMMON references. All programs containing a reference to COMMON must be link edited in order to allocate space for the COMMON variables. Likewise the loaders do not accept BLOCK DATA input. Unresolved external references are not considered errors.

The loaders detect the following error conditions:

a. COMMON references

b. Parity errors on tape

c. Blank cards

d. Check-sum errors

e. Programs larger than the available core.

Any of these conditions will cause loading to stop, and the M-register will be set to non-zero.

7-2 LOADING PROCEDURE.

a. Press the SCAN mode pushbutton. This halts computer execution.

b. Enter the hexadecimal value 0008 in the Program Counter.

c. Press the MR pushbutton.

d. Enter each word of the bootstrap (refer to paragraph 7-3) as follows:

   Press the CLEAR pushbutton.

   Set the DATA buttons to correspond with the word of the bootstrap being entered.

   Press the LOAD pushbutton.

   Press the SCAN pushbutton.

e. Press the SIE and SYSTEM RESET pushbuttons.

f. Make sure the BKPT light is not illuminated.

g. Enter the hexadecimal value 0008 in the Program Counter.

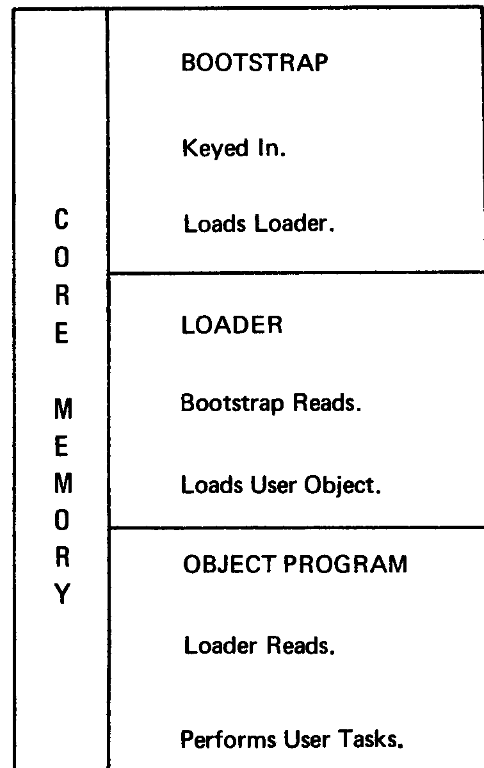


Figure 7-1. Bootstrap and Loader

h. Load the loader tape or card deck into the appropriate input reader and make sure the reader is ready.

i. Press the RUN mode pushbutton. The loader will now load.

j. Place the program to be loaded in the appropriate reader.

k. Enter the hexadecimal value 0018 in the Program Counter.

l. Press the RUN pushbutton. The program will idle. At this time, the load point and core size may be entered according to the following steps.

m. If the program is to be loaded immediately after the loader skip this step. If a load point is to be specified, place it in the A register at this time. If an absolute program is to be loaded, the load point used will be that obtained from the program itself and this step may be skipped.

n. If the assumed default core size is incorrect, enter the core size in the E register. This feature may be used to protect an area of upper memory from being loaded into or used by background programs

o. Press the RUN pushbutton. The object program will now load.

p. Press the RUN pushbutton to start execution of the object program.

If it is desired to manually perform the load again to load another program, press the SCAN pushbutton to halt the computer and perform steps j through p. If it is desired to reload the loader using a previously entered bootstrap, press the SCAN pushbutton and perform steps e through i.

NOTE

RTM-I/BATCH will load a disc bootstrap over the manually entered bootstrap as part of its initialization procedure (paragraph 7-4).

7-3 BOOTSTRAPS.

The following sequences of hexadecimal instructions must be manually loaded to load the indicated loaders.

Location	Teletypewriter	High Speed Paper Tape Reader	Card Reader
8	7802	7802	1781
9	OOA8	OOA8	1F01
A	FF6E	FF6E	D83F
B	0791	0701	2003
C	D822	D830	C574
D	2000	2000	D81F
E	10FB	10FB	2080
F	C574	C574	78FD
10	D802	D818	C868
11	2080	2080	C845
12	78FD	78FD	CCC0
13	C868	C968	78F5
14	C7C7	C7C7	C7C7
15	8EF3	8EF3	8A7F

7-4 RELOADING FROM DISC.

Systems using the BATCH background processor will save a copy of the core resident programs on disc as part of the system initialization procedure. Core resident programs which are accidentally destroyed may be restored from the disc copy.

7-4.1 LOADING FROM DISC. The following procedure is used to load from disc.

a. Press the SCAN pushbutton to halt the computer.

b. Enter the hexadecimal value 0008 in the Program Counter.

c. Press the RUN pushbutton. This will execute a core resident disc bootstrap and loader program included as part of the supervisor.

d. If the core resident programs fail to load and execute properly, the bootstrap program itself has been altered. The disc bootstrap should be manually loaded and executed.

7-4.2 MANUALLY LOADING AND EXECUTING DISC BOOTSTRAP. To manually load and execute the disc bootstrap, the following procedure is used:

a. Press the SCAN pushbutton to halt the computer.

b. Enter the hexadecimal value 0008 in the Program Counter.

c. Press the MR pushbutton.

d. Enter each word of the bootstrap as described in paragraph 7-5.

e. Press the SIE and SYSTEM RESET pushbuttons.

f. Make sure the BKPT light is not illuminated.

g. Enter the hexadecimal value 0008 in the Program Counter.

h. Press the RUN mode pushbutton.

i. If the core resident programs fail to load and execute properly, the copy of the monitor on disc has also been destroyed. The disc content must also be restored.

7-4.3 RESTORING THE DISC. To restore the disc, the following procedure is used:

a. Load the restore disc program using the procedure of paragraph 7-2.

b. Execute the restore disc program using a previously generated copy of the disc. This copy is made using the save disc program. This procedure is described in Section IV.

c. If a copy of the disc has not been made using the save disc program, the core resident programs must be reloaded and the disc contents regenerated as described in Section IV.

7-5 DISC BOOTSTRAP.

The following sequence of hexadecimal instructions may be manually loaded and executed to initiate reloading from the disc.

Location	Instruction
8	D900
9	000D
A	1700
B	40FF
C	7804
D	0000
E	0000
F	1007

SECTION VIII

DEBUG PACKAGE

SECTION VIII

DEBUG PACKAGE

8-1 INTRODUCTION.

The Debug Package consists of seven function programs:

a. Inspect and change core (IC)

b. Store masked constant (ST)

c. Search for masked constant (SR)

d. Hexadecimal dump (HD)

e. Punch object tape (PT)

f. Correction load (CL)

g. Move debug program (MV).

8-2 PROGRAM OPTIONS.

The debug package for any particular installation may use the high-speed reader, card reader, or teletypewriter tape reader for input; line printer or teletypewriter for printed output; and high-speed punch or teletypewriter for punched output. The user should know which options are present in the debug package before attempting to use the program.

Each function program will be assigned a unique program number. This program number is used to select the function program for execution. Hence, it will be necessary to know the program number assignments assumed to use a particular debug function. The program number assignments are as follows:

IC = 1
ST = 2
SR = 3
HD = 4
PT = 5
CL = 6
MV = 7

8-3 GENERAL OPERATING PROCEDURES.

Each program is addressed by entering the sum of the starting address of the debug package and the program number in the program counter. Data is entered in a register in the following manner:

a. Press the CLEAR pushbutton.

b. Enter data by pressing each DATA pushbutton which should correspond to a one-bit. If the light above a DATA pushbutton shows that a one-bit has been set, which should have been left as a zero-bit, press the pushbutton again to reset it to a zero (light out).

c. Select the correct register by pressing the corresponding pushbutton in the DISPLAY SELECT. Refer to Section I for an explanation of the DISPLAY SELECT panel.

d. Press the LOAD pushbutton.

If, after completing this instruction sequence, wrong data has been entered, repeat the sequence until the correct data has been entered. Data in a register is examined in the following manner:

a. Press the SIE mode pushbutton.

b. Select the correct register by pressing the corresponding pushbutton in the DISPLAY SELECT.

c. The contents of the register may be seen in the DISPLAY lights.

d. To examine another register before continuing, repeat instructions b and c.

Parameters for each function program should be entered *only* in the registers specified by its operating procedure and *only* at the points indicated.

After execution of the Move Debug Package operation (MV), the program counter must be reset to whatever program is to be run next and its parameters entered. After execution of any of the other function programs, a new set of parameters may be entered to re-execute the program. To enter any other program, the program counter must be reset.

8-3.1 LOADING A USER PROGRAM. When loading a user program after loading the debug program, the starting address must be greater than or equal to 0095, if DMAC is not attached to the computer, and greater than or equal to 00A8 if DMAC is attached to the computer. *Be sure that the program will not destroy any part of the debug program.* If the exact limits of the debug program are not definitely known, execute the MV instruction sequence through the examination of the registers to obtain the boundaries of the debug program.

If the tape is absolute rather than relocatable, the program will be loaded at the point of origin as indicated on the tape and the starting address does not have to be loaded by hand. However, the restrictions noted above on starting addresses and program limits are still applicable.

8-3.2 INSPECT AND CHANGE CORE. The Inspect and Change Core (IC) operation permits examination of consecutive memory words, starting wherever desired, and changes to these words, if desired. The inputs required for this operation are:

a. The sum of the starting address and program number (DEBUG+1) entered into the Program Counter

b. The hexadecimal address of the first word of memory to be inspected entered into the A register.

Once the program has been addressed, the computer will continuously execute the program until the Program Counter is reset. Each time through the loop, the program will idle twice. During the first idle, the contents of the address specified in the A register may be examined and changed in the E register. During the second idle, the next address may be examined and changed. The operating procedure is as follows:

a. Press the SIE and SYSTEM RESET pushbuttons. Enter the first address to be inspected in the A register. Enter the value DEBUG + 1 in the Program Counter. Press the RUN mode pushbutton.

b. Examine the contents (of the address specified in the A register) in the E register. Change the contents if desired by entering the new contents in the E register. Press the RUN pushbutton.

c. Examine the next address to be inspected in the A register. Change the address if desired by entering the new address in the A register. Press the RUN pushbutton.

d. Repeat steps b and c.

8-3.3 STORE MASKED CONSTANT IN MEMORY. The Store Masked Constant (ST) routine calculates and stores a value using the following Boolean equation:

$$\text{Value} = (\text{Y and } M') \text{ or } (\text{K and M}),$$

where:

Y = the contents of a memory word
M' = the one's complement of the mask
K = the constant
M = the mask.

The inputs required by this operation are:

a. A mask word in the M register

b. A constant in the S register

c. The address of the first location to be changed in the A register

d. The address of the last location to be changed in the E register

e. The starting address of the function program (DEBUG + 2) in the Program Counter.

When the program has calculated and stored the new values within the limits defined in the A register and the E register, new values for parameters a through d may be entered to repeat the program.

Do not include any part of the Debug Package within the limits entered into the A and E registers.

This routine may be used for such operations as setting an area to zero or altering a table of eight-channel ASCII codes to seven-channel ASCII codes.

The operating procedures are as follows:

a. Press the SIE and SYSTEM RESET pushbuttons.

b. Enter the mask value in the M register.

c. Enter the constant value in the S register.

d. Enter the first address in the A register.

e. Enter the last address in the E register.

f. Enter the value DEBUG + 2 in the Program Counter.

g. Press the RUN pushbutton.

h. To execute the program again with different parameters, press the SIE pushbutton and repeat steps b through g but *do not* reset the Program Counter.

8-3.4 SEARCH FOR MASKED CONSTANT. The Search for Masked Constant (SR) routine calculates and performs the following comparison:

$$(\text{Y and M}) = (\text{K and M}),$$

where:

Y = the contents of a memory word

M = the mask
K = the constant.

The inputs to this operation are:

a. A mask word in the M register

b. A constant in the S register

c. A starting address in the A register

d. The last consecutive address to be treated in the E register

e. The starting address of the program (DEBUG + 3) in the Program Counter.

An output to the print device occurs when a value (Y and M) matches the value (K and M). The output consists of the four-digit hexadecimal address and the four-digit hexadecimal representation of the contents of that location. For example: a printout of '1A00=FFFF' indicates that the value of location 1A00 is FFFF.

This routine may be used to find a particular instruction (such as a branch unconditional (BRU) instruction) or find a particular value (such as zero). Any word in memory may be tested by this operation.

The operating procedure is:

a. Depress the SIE and SYSTEM RESET buttons.

b. Enter the mask value in the M register.

c. Enter the constant in the S register.

d. Enter the first address in the A register.

e. Enter the last address in the E register.

f. Enter the value DEBUG + 3 in the Program Counter.

g. Make sure the printing device is on-line and ready to go.

h. Depress the RUN pushbutton.

i. To execute again with different parameters, wait until printing has ceased. Then depress the SIE pushbutton and repeat steps b through h but *do not* reset the Program Counter.

8-3.5 HEXADECIMAL DUMP. The Hexadecimal Dump (HD) routine dumps the contents of the specified area of memory as four hexadecimal digits per memory word and eight words per printed line, with the first word in each line on a multiple-of-eight boundary (ending in zero or eight).

The inputs to this operation are:

a. The address of first word to be dumped in the A register

b. The address of the last consecutive word to be dumped in the E register

c. The program address (DEBUG + 4) in the Program Counter.

The output to the print device consists of one or more lines in the following format:

AAAA = CCCC CCCC CCCC CCCC CCCC CCCC CCCC CCCC,

where AAAA is the address of the first CCCC in the line, and each CCCC is the hexadecimal representation of the contents of a memory word. The eight CCCC's then represent eight consecutive words of memory. Any word in memory may be dumped to the print device.

The operating procedure is:

a. Press the SIE and SYSTEM RESET pushbuttons.

b. Enter the first address in the A register.

c. Enter the last address in the E register.

d. Enter the value DEBUG + 4 in the Program Counter.

e. Make sure the printing device is on-line and ready to go. Press the RUN pushbutton.

f. To execute again with different addresses, wait until printing has ceased. Then press the SIE pushbutton and repeat steps b through e but *do not* reset the Program Counter.

8-3.6 PUNCH OBJECT TAPE. The Punch Object Tape (PT) routine outputs the specified area of memory as an absolute object tape. (Refer to Section II for the format).

The inputs to this operation include:

a. The first address to be output in the A register

b. The last address to be output in the E register

c. The entry point address in the X register

d. The address of the punch tape program (DEBUG + 5) in the Program Counter.

The operating procedure is as follows:

a. Press the SIE and SYSTEM RESET pushbuttons.

b. Enter the first address in the A register.

c. Enter the last address in the E register.

d. Enter the entry point address in the X register.

e. Enter the value DEBUG + 5 in the Program Counter.

f. Make sure there is sufficient tape loaded in the tape punch and that it is on-line and ready.

g. Press the RUN pushbutton.

h. To execute the program again with different limits, wait until the punch motor has stopped. Then press the SIE pushbutton, and then repeat steps b through g but *do not* reset Program Counter.

8-3.7 CORRECTION LOAD. The Correction Load (CL) routine loads a large number of memory corrections without having to enter each data word through the IC routine. The corrections are entered in a specific format on cards or punched tape.

The basic rules for the input format are as follows:

a. Both addresses and contents must be in the form of one to four hexadecimal digits, right-justified. If more than four digits are used to represent either address or contents, the last four digits of the number will be used by the program.

b. An equal sign (=) indicates that the number to the left of the equal sign is the next address to be changed (called the current address).

c. A comma (,) indicates that the number to the left of the comma shall be stored at the current address, after which the current address shall be incremented by one.

d. An exclamation point (!) indicates the end of the corrections and signals the end of program execution. It also results in the storage of the number to the left of the exclamation point in the current address.

e. If the corrections are on paper tape, a carriage return has *exactly* the same effect as a comma. All characters except the digits 0-9, the letters A-F, and four punctuation marks (comma, equals, exclamation point, and carriage return) are ignored by the program.

f. If the corrections are on cards, all characters except the digits 0-9, the letters A-F, and three punctuation marks (comma, equals, and exclamation point) are ignored. If the program discovers an end-of-card status when scanning for a punctuation mark, it will read and scan cards until it does find a punctuation mark.

PROGRAM CAUTION

Any of the following sequences of legal characters will result in the designation of the zero location as the current location, a condition which would result in destroying the bootstrap: comma-equals; carriage return-equals; equals-equals.

Any of the following sequences of characters will result in the storage of a zero in the current address: comma-carriage return; comma-exclamation point; comma-comma; carriage return-comma; carriage return-exclamation point; carriage return-carriage return.

If an exclamation point is *not* present in the corrections, the program will keep trying to read until the SIE pushbutton is pressed to halt the machine. To re-enter the program in such a case, press the SYSTEM RESET pushbutton and reset the Program Counter.

Do not alter any portion of the Debug Package.

The operating procedure is as follows:

a. Punch the card deck or tape off-line. *Do not* forget to put in the end character (exclamation point). If the corrections are on cards, do not forget to end cards with a comma before proceeding to the next card.

b. Press the SIE and SYSTEM RESET pushbuttons.

c. Load the corrections into the reading device.

d. Make sure the reading device is on-line and ready to read.

e. Enter the value DEBUG + 6 in the Program Counter.

f. Press the RUN pushbutton.

g. After the reading device has stopped, the program may be re-entered by pressing the SIE pushbutton and repeating steps c through f. *Do not* reset the Program Counter.

8-3.8 MOVE DEBUG PACKAGE. The Move Debug Package (MV) routine permits the Debug Programs as a whole (excluding the bootstrap and loader) to be moved to a different memory location.

The inputs to the program are:

a. The starting address of the program (DEBUG + 7) in the Program Counter

b. The new starting address (the new value of DEBUG) in the A register.

When the MV program is addressed, it will provide the present starting and ending addresses of the Debug Program and the bias length for a move to the left of the present location. It is the user's responsibility when deciding upon the new starting address to make sure of the following items:

a. that the new address conforms to the restrictions for the minimum value of the address, as explained in paragraph 8-3.1.

b. that the new DEBUG value is at least (DEBUG + total length of the Debug Programs) or less than (DEBUG - bias).

At the end of execution, control is transferred to the idle command at the new DEBUG location. To perform any of the functions, simply repeat the normal procedures, using the new value of DEBUG in calculating the Program Counter value.

The operating procedure is as follows:

a. Press the SIE and SYSTEM RESET pushbuttons.

b. Enter the value DEBUG + 7 in the Program Counter.

c. Press the RUN pushbutton.

d. Examine the starting address of the Debug Package (the value DEBUG) in the E register.

e. Examine the address of the last word of the Debug Package in the X register.

f. Examine the bias of the program for a move to the left in the A register.

g. Enter the new value for DEBUG in the A register, complying with the rules stated above.

h. Press the RUN pushbutton.

i. To enter this or any other function program of the DEBUG package, perform the appropriate operating procedure.

8-3.9 RUNNING A USER PROGRAM. After a user program has been loaded it may be executed as follows:

a. Make sure any input/output devices the program requires are on-line and ready.

b. Press the SIE and SYSTEM RESET pushbuttons.

c. Enter the absolute starting address of the program in the Program Counter.

d. Press the RUN pushbutton.