digital

For industrial detection, analysis and control

indac 8/2

handbook

In the Experimental Laboratory, the QC Station, the Process Control Room, wherever you work in modern industry, you will find the growing explosion of minicomputers. Digital Equipment Corporation, the company with the largest number of minicomputer installations in the world, presents an integrated hardware/software system to help industry solve the problems of detection, analysis and control.

for industrial detection, analysis and control

indac 8/2 handbook

For additional copies, order No. DEC-IN-GRZA-D from the Program Library, Digital Equipment Corporation, Maynard, Mass. 01754 Price: $6.00

digital equipment corporation

1st Edition December 1971

The material in this manual is for informational purposes and is subject to change without notice.

CONTENTS

		Page
CHAPTER 1	**AN OVERVIEW**	
	The Industrial Environment	1-1
	Data Collection	1-1
	Process Control	1-1
	INDAC Language Capabilities	1-2
	Facilities of the INDAC Software System	1-2
CHAPTER 2	**BUILDING THE SAMPLE SYSTEM**	
2.1	Introduction	2-1
2.2	Loading the HINDAC (Tape 1) Program	2-1
2.3	Loading the Monitor System Dump and INDAC File Tapes	2-2
2.4	Loading and Building the INDAC Compiler	2-4
2.5	Running the SGEN 8/2 Program	2-5
2.6	Loading the INDAC Executive 8/2	2-5
2.7	Loading GENDAC to Configure the Sample System	2-6
2.8	Loading Sample Program 2 Using the Editor	2-7
2.9	Compiling Sample Program 2	2-8
2.10	Executing Sample Program 2	2-9
CHAPTER 3	**PROGRAMMING THE INDAC 8/2 SYSTEM**	
3.1	Introduction	3-1
3.2	Programming an Industrial Algorithm	3-2
3.2.1	LET Statement (Assignment)	3-3
3.2.1.1	Arithmetic	3-3
3.2.1.2	Boolean	3-6
3.2.1.3	Statement Evaluation Rules	3-8
3.2.2	GOTO Statement (Branching)	3-9
3.2.3	IF Statement (Conditional Testing)	3-10
3.2.4	FOR/NEXT Statements (Looping)	3-12
3.2.5	DO/RETURN Statements (subroutines)	3-15
3.3	Message and Logging Capabilities	3-16
3.3.1	.HEADER Statement	3-22
3.3.2	.FORMAT (Picture) Statement	3-22
3.3.3	.FORMAT (Declarative) Statement	3-23
3.3.4	.STORAGE Statement	3-24
3.3.4.1	Development of Array Names	3-24
3.3.4.2	Specification of Array Size	3-25
3.3.4.3	Specification of Array Elements	3-25
3.3.4.4	Specification of an Array Window	3-25
3.3.4.5	Array Spanning Windows	3-26
3.3.4.6	Presetting Stored Values	3-26
3.4	Data Collection and Control	3-27
3.4.1	DAC (Digital-to-Analog Conversion Devices)	3-34

		Page
3.4.1.1	Hardware Device	3-34
3.4.1.2	Equipment Declaration Statement	3-34
3.4.1.3	Language Statement	3-34
3.4.2	ADC (Analog-to-Digital Conversion Devices)	3-34
3.4.2.1	Hardware Device	3-34
3.4.2.2	Equipment Declaration Statement	3-34
3.4.3	AF04 (Integrating Digital Voltmeter)	3-36
3.4.3.1	Hardware Device	3-36
3.4.3.2	Equipment Declaration Statement	3-36
3.4.3.3	Language Statement	3-37
3.4.4	UDC (Universal Digital Controller)	3-37
3.4.4.1	Hardware Device	3-37
3.4.4.2	Equipment Definition Statement	3-37
3.4.4.3	Language Statement	3-38
3.4.5	FILE (Pseudo-Device)	3-38
3.4.5.1	Hardware Device	3-38
3.4.5.2	Equipment Definition Statement	3-38
3.4.5.3	Language Statement	3-38
3.4.6	Equipment Statement Summary	3-39
3.5	Structure of an INDAC Job	3-40
3.5.1	Program Segmentation	3-40
3.5.1.1	Job Specification Segment	3-41
3.5.1.2	.PHASE Segment	3-42
3.5.1.2.1	Activity Statements	3-42
3.5.1.2.2	.ACTION Statement	3-42
3.5.1.3	.SNAP Segments	3-43
3.5.1.4	Subroutine	3-44
3.5.1.4.1	Internal Subroutine	3-44
3.5.1.4.2	External Subroutine	3-44
3.5.1.4.3	Implicit Subroutine	3-45
3.5.2	Scheduling Capabilities	3-46
3.5.2.1	Activity Statements	3-47
3.5.2.1.1	EVERY Activity Statements	3-47
3.5.2.1.2	DELAY Activity Statements	3-48
3.5.2.1.3	AT Activity Statements	3-49
3.5.2.2	Action Statements	3-49
3.5.2.2.1	Timer Action Statements	3-49
3.5.2.3	EXIT Program Control Statement	3-49
3.5.2.4	Resolving Timer Requests	3-50
3.6	The Run-Time System/Making the Pieces Work	3-50
3.6.1	Time and Priority Scheduling	3-51
3.6.1.1	Division of the Process Stack	3-51
3.6.1.1.1	Executive Command Decoder	3-51
3.6.1.1.2	Field Interrupt Processing	3-51

		Page
3.6.1.1.3	Foreground Processing	3-52
3.6.1.1.4	Background Processing	3-53
3.6.2	Dynamic Core Management and Swapping	3-53
3.6.2.1	Job Specification Segment	3-53
3.6.2.2	The PHASE Segment	3-53
3.6.2.3	The SNAP Segment	3-53
3.6.2.4	External (Disk-Resident) Subroutines and Functions	3-53
3.6.2.5	Switching Priority Levels	3-53
3.6.2.6	Executive to SNAP Communication	3-54
3.6.3	Dynamic I/O Buffers	3-54
3.6.4	Operator Communication	3-54
3.7	Servicing Field Interrupts	3-55
3.7.1	Equipment Declaration Statement	3-55
3.7.2	Language Statements	3-55
3.7.2.1	The Standard Call	3-55
3.7.2.2	INITIALIZE Request	3-56
3.7.2.3	IDENTIFY Request	3-56
3.7.2.4	TRANSFER Request	3-57
3.7.3	Sample Program	3-58
3.8	Handling the Consoles	3-59
3.8.1	Equipment Declaration Statement	3-60
3.8.2	Language Statements	3-60
3.8.3	Program Examples	3-61

CHAPTER 4 CONFIGURING A SYSTEM HAVING STANDARD DEC PROCESS I/O DEVICES

4.1	Introduction	4-1
4.2	Loading the HINDAC Program	4-2
4.3	Loading the Monitor System Dump and INDAC File Tapes	4-3
4.4	Loading and Building the INDAC Compiler	4-5
4.5	Running the SGEN 8/2 Program	4-6
4.6	Loading the INDAC Executive 8/2	4-6
4.7	Loading GENDAC to Configure a Specific System	4-7

CHAPTER 5 PREPARING THE PROGRAM

5.1	Introduction	5-1
5.2	Monitor	5-1
5.2.1	Monitor Residence	5-1
5.2.2	Starting the Monitor	5-1
5.2.3	Bootstrapping the Monitor	5-2
5.2.4	Monitor Error Messages	5-2
5.3	System Programs	5-2
5.3.1	Command String Format	5-3

		Page
5.3.1.1	Device Names	5-3
5.3.1.2	Filenames	5-3
5.3.1.3	Punctuation	5-4
5.3.1.4	Special Characters	5-4
5.3.2	Examples of Command Strings	5-4
5.3.3	Editor	5-5
5.3.3.1	Modes of Operation	5-7
5.3.3.2	Input Commands	5-7
5.3.3.3	Output Commands	5-8
5.3.3.4	Editing Commands	5-8
5.3.3.5	Special Characters and Functions	5-11
5.3.3.6	Editor Error Messages	5-13
5.3.4	INDAC Compiler	5-13
5.3.4.1	Compiler Output	5-14
5.3.4.2	Errors During Compilation	5-15
5.3.4.3	Correcting Compilation Errors	5-15
5.3.5	PIP	5-16
5.3.6	Loading Programs – Disk System Binary Loader	5-22
5.3.6.1	Binary Loader Operating Procedures	5-22
5.3.6.2	Binary Loader Error Messages	5-24
CHAPTER 6	**EXECUTING THE PROGRAM**	6-1
CHAPTER 7	**MODIFYING THE SYSTEM**	
7.1	Introduction	7-1
7.2	System Communication Tables	7-1
7.2.1	Intrinsic Functions ⟨IF⟩	7-1
7.2.2	External Subroutines ⟨XS⟩	7-1
7.2.3	System Devices ⟨SD⟩	7-2
7.2.4	Core Map ⟨CM⟩	7-2
7.2.5	Page Zero ⟨PZ⟩	7-2
7.2.6	Special GENDAC Table ⟨**⟩	7-3
7.3	Updating the Software	7-3
7.3.1	System Mode of Updating	7-3
7.3.2	Binary Mode of Updating	7-4
7.4	Library Structure	7-5
7.4.1	Basic Module	7-5
7.4.2	Extension Modules	7-6
7.4.3	Call-Up Modules	7-6
7.4.4	Additional Notes	7-6
7.5	Library Tape Format	7-9
7.5.1	Definitions	7-9
7.5.2	Library Tape Header	7-9
7.5.3	Module Header	7-10

CONTENTS (Cont)

		Page
7.5.3.1	Functional Group Declaration	7-10
7.5.3.2	Module Declaration	7-11
7.5.3.2.1	Physical Descriptions of Module	7-11
7.5.3.2.2	Logical Description of Module	7-13
7.5.3.3	Module Body	7-14
7.5.3.3.1	Module Title	7-15
7.5.3.3.2	Fixup Declarations	7-15
7.5.3.4	Group and Module Termination	7-16
7.6	Coding Details	7-16
7.6.1	Function (Core-Resident in Field 0)	7-17
7.6.2	Subroutine	7-18
7.6.2.1	Subroutine Requirements and Analysis	7-18
7.6.2.2	Sample of Subroutine Coding	7-18
7.6.3	I/O Handler	7-19
7.6.3.1	I/O Handler Operation	7-19
7.6.3.1.1	The Control Driver Table	7-20
7.6.3.1.2	The Format Driver Table	7-20
7.6.3.1.3	Data Information and Calls	7-20
7.6.3.1.4	Executive Page Zero Parameters	7-20
7.6.3.1.5	Interrupting Devices	7-21
7.6.3.2	A Typical I/O Handler	7-21
7.7	Vector Code	7-22
7.7.1	Numeric Formats	7-22
7.7.2	Vector Code Example	7-22
7.7.3	Load and Store Instructions	7-23
7.7.3.1	Load a Simple Variable	7-23
7.7.3.2	Store a Simple Variable	7-24
7.7.3.3	Load an Array Element (Indexed Variable)	7-24
7.7.3.4	Store an Array Element	7-24
7.7.3.5	Load an External Argument	7-24
7.7.3.6	Store into an External Argument	7-24
7.7.3.7	Load an External Argument, Indexed	7-24
7.7.3.8	Store into an External Argument, Indexed	7-25
7.7.4	GOTO Statements	7-25
7.7.4.1	Unconditional GOTO	7-25
7.7.4.2	Computed GOTO	7-25
7.7.5	IF Statement	7-25
7.7.6	DO Statements	7-26
7.7.6.1	Internal Subroutine Call	7-26
7.7.6.2	External Subroutine Call	7-26
7.7.7	Return Statement	7-26
7.7.8	Loop Statements	7-27
7.7.8.1	FOR Statement	7-27
7.7.8.2	NEXT Statement	7-27

		Page
7.7.9	Arithmetic and Logical Operators	7-27
7.7.9.1	Binary Vectors	7-27
7.7.9.2	Unary Vectors	7-28
7.7.10	Special Vectors	7-28
7.7.11	Special PAL-I Code	7-28
7.8	Running GENDAC	7-29

APPENDICES

APPENDIX A	**SUMMARY OF INDAC STATEMENTS**	
A.1	Executable Statements	A-1
A.2	Non-Executable Statements	A-2
APPENDIX B	**EXAMPLE PROGRAM**	B-1
APPENDIX C	**USING THE DISK MONITOR SYSTEM**	
C.1	Editor Command Summary	C-1
C.2	Editor Key Function Summary	C-2
C.3	Error Messages	C-2
APPENDIX D	**INDAC COMPILER ERROR MESSAGES**	D-1
APPENDIX E	**SPUT ERROR CONDITIONS**	E-1
APPENDIX F	**INDAC 8/2 EXECUTIVE ERROR MESSAGES**	F-1
APPENDIX G	**GENDAC ERROR MESSAGES**	G-1
APPENDIX H	**EXECUTIVE COMMAND DECODER OPERATION**	
H.1	Operator Commands	H-1
H.2	Command Decoder Error Messages	H-1
APPENDIX I	**THE GENDAC LIBRARY**	I-1
APPENDIX J	**SYSTEM COMMUNICATION TABLES**	J-1
APPENDIX K	**ESUP OPERATION**	
K.1	Introduction	K-1
K.2	Loading Procedure	K-1
K.3	Producing the Equate Tape	K-1
K.4	Producing the Core Map	K-2

APPENDICES (Cont)

		Page
K.5	Returning to Monitor	K-2
K.6	Restrictions	K-2
K.7	References	K-3
K.8	Examples	K-3
K.8.1	Loading and Usage	K-3
K.8.2	Sample Equate Printout	K-3

ILLUSTRATIONS

Figure No.	Title	Art No.	Page
2-1	Sample System Log Output		2-i
3-1	Sample Process Control Application	08-0727	3-29
3-2	Structure of an INDAC Job		3-41
4-1	GENDAC and I/O Handler Dialogue		4-11
4-2	GENDAC and Function Dialogue		4-17
7-1	System Communication Tables – Interaction	08-0657	7-2
7-2	Paper Tape Binary Image	08-0671	7-5
7-3	Internal Format of Numeric Data	08-0670	7-22

SKELETONS

Skeleton No.	Title	Page
	Basic Skeleton	3-1
1	The Algorithm	3-3
2	Sample Alarm Test Routine	3-16
2	Header/Format Details	3-19
2	Storage Details	3-21
3		3-28
3	Equipment Details	3-32

CHARTS

Chart No.	Title	Page
4-1	INDAC I/O Device Handler Library Summary	4-9
4-2	INDAC Function Library Summary	4-10

TABLES

Table No.	Title	Page
3-1	Basic Arithmetic Notation, Algebraic vs. INDAC	3-4
3-2	Relational Operators	3-10
3-3	Mask Generation	3-13
3-4	Declarative .FORMAT Specifications	3-24
3-5	Job Specification Segment, Organization and Contents	3-41
3-6	Organization of PHASE Segment	3-42
3-7	Organization of a SNAP Segment	3-43
5-1	System Error Messages	5-2
7-1	Allocation Codes	7-12
7-2	Allocation Parameters	7-12
7-3	Table Update Information	7-13
7-4	CALL Statements	7-20
7-5	Executive Parameters	7-21
7-6	Binary Vectors	7-27
7-7	Unary Vectors	7-28
7-8	ESUP Commands	7-29
G-1	Numbered GENDAC Error Messages	G-1
G-2	Unnumbered GENDAC Error Messages	G-3
J-1	System Communication Tables	J-1

CHAPTER 1
AN OVERVIEW

THE INDUSTRIAL ENVIRONMENT

Accurate gathering and reporting of data is indispensable to the efficient operation of a modern industrial facility. The data is needed to evaluate the overall operation, to determine its efficiency, and to calculate actual vs. theoretical yields. Data must also be displayed either permanently in printed copy or temporarily through CRT displays and indicator panels, to keep operators informed of the status, trends, and disturbances within the process or test. As a result of calculations performed on the data, the process can be controlled and modified by operator interaction or through feedback hardware.

Multichannel analog graphic recorders can monitor a number of variables and compare them to a known time base, but interpretation has limited accuracy and the reduction and analysis of the data requires considerable time and effort. Also, with graphic recorders, the possibility of automatic feedback control is lost.

The high speed, accuracy, and flexibility of a computer-based system provides data gathering techniques greatly beyond the ability of simple displays and recorders.

DATA COLLECTION

The basic function of any computer-based data acquisition system is to measure and record the operating characteristics of various equipment or process sensors and to compare these measured values with predetermined standards. The system controller, the computer, monitors input signals, controls the data sampling process, evaluates input data, detects process malfunctions, calculates data necessary to define system performance, formats data for display or storage, and controls process actuators.

A computer-based system will characteristically monitor many process sensors, both analog and digital. Analog inputs, such as those from thermocouples, potentiometers, strain gauges and flow meters, must be converted to digital signals acceptable to a data processor. In addition to converted analog inputs, the computer may accept digital information from sources such as contact closures, limit switches, and manual entry consoles. System outputs, both direct digital and digital converted to analog, may be used to control displays, recorders, and actuators. Data and information may also be transferred to mass storage units, alphanumeric printers and punched paper tape.

PROCESS CONTROL

A computer controller gives the data acquisition system total flexibility. The computer program is used to determine what to measure, when to measure, and how to interpret the measurement. The system can thereby translate voltage readings into meaningful engineering units and can subject the inputs from different transducers to the proper linearization functions. The computer program can calculate indirect measurements, such as flow, from the measurable quantities of velocity, cross-sectional area, and time. The data acquisition and control function may need to be established as a foreground priority task. If this function does not demand all available

computer time, a background task could be implemented to perform such functions as calculating overall process or plant efficiency or maintaining inventory control records.

In an application such as engine testing, a designer may build a computer-based data acquisition system that can format and pre-process incoming data before placing it on tape for later analysis at a computation center. Formatting and pre-processing save considerable computation time. The computer can also set limits for critical parameters, and provide an alarm when the limits are exceeded, thereby increasing the safety of facilities and personnel. With feedback hardware, the computer can control fuel flow rate, air mixture, load environmental conditions, etc.

Because of their high speed, accuracy, and flexibility, computers are being implemented in the industrial world. Successful implementation depends largely on the availability of process interface devices and easy-to-use software. INDAC 8/2, for example, offers an industrial language to simplify bringing the computer into the process.

INDAC LANGUAGE CAPABILITIES

The current high-level languages do not allow such functions as time and priority sequencing, handling of random external interrupts, analog and digital input and output, and file handling – all of which may be called for in data acquisition and control systems.

With these limitations in mind, Digital's engineers and programmers implemented statements, similar to those used in BASIC and FORTRAN for arithmetic and logical computations and program control with additional English statements to provide for:

- Control of the start of tasks according to a time lapse, or time-of-day.
- Segmentation of programs into several units, with each unit occupying computer storage only when it is actually in use.
- Definition of process interface equipment, associated channels, and modes of operation.
- Easy-to-use data collection and control statements.
- Transfer of data between the computer and mass storage for recordkeeping or analysis.

The result: the INDAC 8/2 language, the first compiler-level language designed for real-time data acquisition and control in a mini-computer. With a few hours training an engineer familiar with BASIC or FORTRAN can produce simple programs in the INDAC 8/2 language for analysis of process environments.

FACILITIES OF THE INDAC SOFTWARE SYSTEM

The INDAC Software System provides the user with facilities for:

- Rapidly loading the INDAC 8/2 Software System
- Configuring or modifying the INDAC 8/2 System to meet the users application needs
- Creating and Editing the application source program
- Compiling the application algorithms into an efficient run-time program.
- Running under control of a real-time Executive
- Utility Support Functions

All INDAC 8/2 programs are on paper tapes supplied with the system. The tapes can be loaded quickly using the detailed procedures given in Chapter 2 or 4.

Once the basic INDAC software system is loaded it can be configured to meet specific users application needs using GENDAC (the system configuration program).

The standard PDP-8 Disk Monitor System, which is unique in the field of small computers, controls all INDAC 8/2 program preparation. With this powerful tool the user can establish and maintain files for INDAC 8/2 source programs, edit, compile, and execute them using simple keyboard commands. The Disk Monitor System also includes utility programs for loading, deleting, and transferring INDAC files.

The compiler converts the source program into object code, which is executed at run time. Extensive diagnostics help the user debug his programs.

The operating system (or Executive) provides for automatic overlays of program units and provides for program or task scheduling based on time, priority, events, or program decisions. Tasks made of program overlays have distinct priority levels. The tasks may be executed in response to process interrupts, events, or timers; or whenever the system is idle. The system transfers tasks, program units, and data automatically from disk to core and vice versa. For slow peripherals, such as the Teletype®, the INDAC 8/2 System provides for buffered output — this allows the output operation to go on independently of the program processing.

®Teletype is a registered trademark of Teletype Corporation.

SAMPLE SYSTEM LOG

(Sheet 1 of 5)

Fold out and use this log for reference while you build the sample system (see Chapter 2)

```
LOAD, DUMP OR VERIFY-L
ENTIRE DISK OR FILES?-E

LOAD, DUMP OR VERIFY-L
ENTIRE DISK OR FILES?-F

LOAD, DUMP OR VERIFY-L
ENTIRE DISK OR FILES?-F

LOAD, DUMP OR VERIFY-
.LOAD
*IN-R:
*
ST=200
↑↑
.LOAD
*IN-R:
*
ST=2
↑↑
.LOAD
*IN-R:
*
ST=2
↑↑
.LOAD
*IN-R:
*
ST=2
↑↑
.LOAD
*IN-R:
*
ST=2
↑↑
.SGEN
.SAVE <SD>!1400;
.HELD
.LELD

.
GENDAC 427
TYPE '?' IF CONFUSED.
4K?-N

*OPT-B
*IN-R:
↑↑P
```

Figure 2-1 Sample System Log Output (Sheet 1 of 5)

SAMPLE SYSTEM LOG

(Sheet 2 of 5)

Fold out and use this log for reference while you build the sample system (see Chapter 2)

```
I/O LIBRARY INDAC 8/2 <U2IC>

SAVED <SD>!1400; FROM SGEN?-Y
CHKS...  STORED AT 10200

INITIAL RUN?-Y
[ZC]  STORED AT 0117
END    ADDED TO IO CHAIN.

DISABLE CTRL/C IN COMMAND MODE?-N

60 HZ PDP-8/E LINE FREQ CLOCK?-Y
CLK8  STORED AT 05102
CLK8  STORED AT 05045
CLK8  STORED AT 05316

50 HZ PDP-8/E LINE FREQ CLOCK?-N

PDP-8/E PROGRAMMABLE CLOCK?-N

60 HZ PDP-8/I OR PDP-8/L?-N

50 HZ PDP-8/I OR PDP-8/L?-N

60 HZ PDP-8 CLOCK?-N

50 HZ PDP-8 CLOCK?-N

UDCP?-N

GENERIC 2?-N

AF01?-N

AF02?-N

AF03?-N

ANY OF THE ABOVE A/D?-N

AFC-8?-N

AD01?-N

AF04?-N

UDCE?-N

AA01?-N

AA50?-N

PIP?-Y
PIP   ADDED TO <SD>
[PP]  STORED AT 01320
PIP    ADDED TO IO CHAIN.
```

Figure 2-1 Sample System Log Output (Sheet 2 of 5)

SAMPLE SYSTEM LOG

(Sheet 3 of 5)

Fold out and use this log for reference while you build the sample system (see Chapter 2)

```
TTY2?-Y
[T2]  ADDED TO <**>
TTY2  ADDED TO <$D>
[T2]  STORED AT Ø1344
TTY2   ADDED TO IO CHAIN.
[T2]  STORED AT Ø3632

TTY3?-N

TTY4?-N
END OF LIBRARY TAPE .

*OPT-
GENDAC COMPLETED.

.EDIT
*OUT-S:TEST
*
*IN-R:
*
*OPT-B

*R

*E

.COMP
*OUT-S:TEST
*
*IN-S:TEST
*
*OPT-
ICLK@   1  INPU@   4  IDEV@   9  ISWI@  10  ICTR@  11  IMOD@  12
IEXE@  13
2244 2244
2215 2215
2150 2150
2211 2211
2733 2733
Ø211 Ø211  2051
.SPUT
*IN-S:TEST
*FILE START Ø368
EXEC LOADED
↑A
*↑DR#1
DO YOU WISH TO BYPASS DESCRIPTIONS?
N
THIS PROGRAM CONTAINS 3 EXERCISERS. ALL THREE EXECISERS MAY BE CALLED
TO OPERATE AUTOMATICALLY BY TYPING 'AUTO' AND 'C/R' (CARRIAGE-
RETURN). ANY SINGLE EXERCISER MAY BE CALLED TO OPERATE BY TYPING
'GOTO' FOLLOWED BY THE EXERCISER NUMBER (ONE-DIGIT) AND A
'C/R'. ANY EXERCISE MAY BE STOPPED BY TYPING 'STOP' AND 'C/R'.
*
*
```

Figure 2-1 Sample System Log Output (Sheet 3 of 5)

SAMPLE SYSTEM LOG

(Sheet 4 of 5)

Fold out and use this log for reference while you build the sample system (see Chapter 2)

```
*
PLEASE ENTER THE CORRECT TIME.
TO SET THE TIME-OF-DAY, PRESS '↑A' (CTRL A), GET RESPONSE,
PRESS '↑V' (CTRL V), 'C' (NO CTRL), AND THEN 'C/R'. THE EXECUTIVE WILL
TYPE OUT THE CURRENT TIME-OF-DAY AND WAIT. WHEN THE TYPING STOPS,
ENTER EXACTLY 6 DIGITS AND TWO COLONS AS FOLLOWS:
'HH:MM:SS'  FOR HOURS, MINUTES  AND SECONDS. THEN TYPE 'C/R'.
AFTER YOU HAVE INSERTED THE CORRECT TIME, PRESS '↑P' (CTRL P)
TO PROCEED IN OPERATOR MODE, THEN PRESS 'C/R' (CARRIAGE RETURN)
TO CONTINUE WITH THE PROGRAM.
↑A
*↑VC
  CLOCK  0: 2: 7,08:00:00
↑P
#
THANK YOU!
DO YOU HAVE- TTY2 ?N
DO YOU HAVE- PTP  ?Y
TURN ON PTP-TYPE CARRIAGE RETURN WHEN READY

ENTER 'AUTO' AND 'C/R' FOR AUTOMATIC TESTING OR ENTER
'GOTON' WHERE N IS THE EXERCISE NUMBER REQUIRED.
AUTO


EXERCISER #1- 20 SECOND DELAY BEGINS NOW

1234567890:-QWERTYUIOPASDFGHJKL;ZXCVBNM,./
THE TIME IS NOW   8: 1:11

1234567890:-QWERTYUIOPASDFGHJKL;ZXCVBNM,./
THE TIME IS NOW   8: 1:31

1234567890:-QWERTYUIOPASDFGHJKL;ZXCVBNM,./
THE TIME IS NOW   8: 1:51

1234567890:-QWERTYUIOPASDFGHJKL;ZXCVBNM,./
THE TIME IS NOW   8: 2:11

1234567890:-QWERTYUIOPASDFGHJKL;ZXCVBNM,./
THE TIME IS NOW   8: 2:31

1234567890:-QWERTYUIOPASDFGHJKL;ZXCVBNM,./
THE TIME IS NOW   8: 2:51
```

Figure 2-1 Sample System Log Output (Sheet 4 of 5)

SAMPLE SYSTEM LOG

(Sheet 5 of 5)

Fold out and use this log for reference while you build the sample system (see Chapter 2)

```
EXERCISER #2
     THIS EXERCISE DISPLAYS THE VALUE REPRESENTED BY THE COMPUTER
SWITCH REGISTER IN OCTAL, DECIMAL, SIGNED DECIMAL, AND E-TYPE
NOTATION. THE EXERCISE REPEATS EVERY 20 SECONDS FOR 3 MINUTES IN
AUTOMATIC MODE.

0200   128  + 128  +0.128000E+0003

4000  2048  -2048  -0.204800E+0004

7777     1  -    1  -0.100000E+0001

0000     0  +    0  +0.000000E+0000

5252  1366  -1366  -0.136600E+0004

2525  1365  +1365  +0.136500E+0004

0001     1  +    1  +0.100000E+0001

3333  1755  +1755  +0.175500E+0004

6666   586  - 586  -0.586000E+0003


EXERCISER #3

THE TIME IS NOW   8: 6:39
WORKING!-INPUT ALLOWED AFTER NEXT TYPEOUT OF TIME
THE TIME IS NOW   8: 6:41
STOP
END OF TEST
↑A
*
.
```

NOTE

EXERCISER #3 was stopped by typing "STOP". This exerciser will repeat every 15 minutes if allowed to continue. Following the "STOP↵" command, the operator typed CTRL/C (control key and C key together) to return to the Disk Monitor System.

Figure 2-1 Sample System Log Output (Sheet 5 of 5)

CHAPTER 2
BUILDING THE SAMPLE SYSTEM

2.1 INTRODUCTION

This chapter contains step-by-step procedures for the new user to check out the basic software/hardware operation of his INDAC 8/2 System by building a sample system. In building the sample system, the new user becomes familiar with some of the operating mechanics of the system which will be of benefit when he builds his specific system.

NOTE
The PDP-8/E control panel differs slightly from other PDP-8 Computers. The procedures presented in this section detail the PDP-8/E controls. When INDAC is implemented with a PDP-8/I or L, use the START switch whenever CLEAR/CONT is specified in the procedures; use LOAD ADDR whenever EXTD ADDR LOAD or ADDR LOAD is specified in the procedures.

The log output from the loading and execution of the sample system is shown in Figure 2-1.

2.2 LOADING THE HINDAC (Tape 1) PROGRAM

To load the computer from a cold-core start proceed as follows.

NOTE
Verify that RUN light is off. If the light is on, press HALT and return HALT switch to up position.

Step	Procedure
1	Load the switch register with 0000.
2	Press EXTD ADDR LOAD.
3	Load the switch register with 0027.
4	Press CLEAR.
5	Press ADDR LOAD.

(continued on next page)

Step	Procedure
6	Successively deposit the following:

Location	Instruction
0027	6011
30	5027
31	6016
32	7450
33	5027
34	7012
35	7010
36	3007
37	2036
40	5027

7	Load the HINDAC (Tape 1) program in the high-speed reader – begin anywhere in the initial blank tape portion.
8	Load switch register with 0031.
9	Press ADDR LOAD.
10	Press Clear.
11	Press CONT.

NOTE

The tape should now read completely through the reader and stop on the trailer portion (code 0200) of the tape. The computer should also halt. At this point, both RIM and the Binary Loader are in core. If the RUN light does not go out, or if the tape does not read in properly, repeat the above procedure.

2.3 LOADING THE MONITOR SYSTEM DUMP AND INDAC FILE TAPES

After the HINDAC program is loaded successfully, load the Monitor Support program using the Binary Loader.

Step	Procedure
1	Place the MSUP tape (Tape 2) in the high-speed reader. Set the leader portion (code 200) of the tape under the read lamp.
2	Load switch register with 7777.
3	Press ADDR LOAD.
4	Set switch register to 3777.
5	Set the rotary console switch to AC.
6	Press CLEAR, then CONT. The tape should now read until the trailer portion (code 200) is under the read lamp. At this point, the computer will halt with the AC containing 0, and the link may be on.

(continued on next page)

Step	Procedure
7	Load switch register with 200.
8	Press ADDR LOAD.
9	Press CLEAR, then CONT. The MSUP program will begin a series of questions to determine the operation required. Each question may be answered by a single letter followed by a carriage return (designated ↵). After it is loaded, the first question is: LOAD, DUMP OR VERIFY-
10	Type L↵ The next question asked is: ENTIRE DISK OR FILES?-
11	Type E↵ The MSUP program will come to a halt after typing "E↵ ". The program is now waiting for the user to load the Monitor System Dump (Tape 3) in the high-speed reader.
12	At this point, before loading the tape, set the switch register to 0001. The LSB switch is used to control the loading of the System Dump tape. When the switch is in the "1" position, the program will idle after completion of loading the current block (one "block" of information). When the switch is set to "0", the program will resume loading. **NOTE** **Make certain that all checksums are torn from the end of the Monitor System Dump tape before loading.**
13	Place the Monitor System Dump (Tape 3) in the high-speed reader. Start tape at leader portion (code 200).
14	Press CONT. At this point, if you have correctly set the LSB switch, the program will be idling.
15	Set the LSB to "0" to start the loading process. At any time setting the LSB to "1" will stop the tape at the next leader/trailer. Loading will resume whenever the LSB is reset to "0". **NOTE** **During the loading process, if the reader malfunctions, MSUP will print "CHECKSUM OR VERIFY ERROR" and halt. Back the tape up one block (blocks are groups of punches separated by leader/trailer, code 200). Place the leader of the block that failed under the reader lamp. Set LSB for continued loading and press CONT. If the block will not load correctly, the tape has been damaged and must be replaced.**
16	When MSUP completes the loading of the Monitor System Dump (Tape 3), there will be a slight pause and the program will type out: LOAD, DUMP OR VERIFY-

(continued on next page)

Step	Procedure
17	Place the INDAC Support Programs (Tape 4) in the high-speed reader. Start tape at leader portion (code 200). **NOTE** **Tear all checksums from the end of the tape (information following the last leader/trailer punches).**
18	Type L ↵ The next question asked is: ENTIRE DISK OR FILES?-
19	Type F ↵ When MSUP completes the loading of the INDAC Support Programs (Tape 4) there will be a slight pause and the program will type out: LOAD, DUMP OR VERIFY-
20	Place the INDAC System Tables (Tape 5) in the high-speed reader. Start tape at leader portion (code 200). **NOTE** **Tear all checksums from the end of the tape (information following the last leader/trailer punches).**
21	Type L ↵ The next question asked is: ENTIRE DISK OR FILES?-
22	Type F ↵ When MSUP completes the loading of the INDAC System Tables (Tape 5) there will be a slight pause and the program will type out: LOAD, DUMP OR VERIFY-
23	When both file tapes have been loaded, type CTRL/C (hold both CTRL and C keys down); this will return control to the Disk Monitor System just loaded. The Monitor will respond with a period. At this point, the disk system is established for a single disk and the Disk Monitor is resident in core.

2.4 LOADING AND BUILDING THE INDAC COMPILER

Load the MAKE 8/2 (Tape 6) program using the Disk Monitor Loader. To load MAKE 8/2 from the high-speed reader, use the following command string:

```
.LOAD↵
*IN-R:↵
*
ST=200↵
↑↑ (User types CTRL/P after each ↑).
```

NOTE

Start the tape at leader portion (code 200).

MAKE 8/2 will execute and return to the monitor. Now load the Compiler tapes:

COMP1 (Tape 7)
COMP2 (Tape 8)
COMP3 (Tape 9)
COMP4 (Tape 10)

The loading sequence (using the high-speed reader) is the following for each tape:

.LOAD↵
*IN-R:↵
*
ST=2↵
↑↑ (User types CTRL/P after each ↑).

NOTE
Start tape at leader portion (code 200).

After the user's second CTRL/P, control returns to the Monitor after about one minute. The same sequence is repeated for each compiler tape.

2.5 RUNNING THE SGEN 8/2 PROGRAM

SGEN 8/2 takes a single-disk system and expands the file structure into as many disks as are connected to the system (Limit 4), and protects the Executive and data file areas that will be used later in the system. Any disk units on the system that are not to be used by the INDAC system should be switched to "off".

To run SGEN 8/2, type the following sequence:

.SGEN↵
.SAVE ⟨SD⟩! 1400;↵

SGEN 8/2 expands the single-disk system to include as many as the user has DS32 expander units selected and returns to the monitor after a short pause. The SAVE command initializes the System Devices Table in preparation for the system configurator program.

2.6 LOADING THE INDAC EXECUTIVE 8/2

Load the INDAC Executive as follows:

Step	Procedure
1	Place the Executive 8/2 (Tape 11) in the high-speed reader. Start the tape at leader portion (code 200).
2	Set the switch register to 3600.
3	Type HELD↵ The HELD program will load the Executive and halt after loading.

NOTE 1
Since paper-tape to disk operations are taking place, the loading may appear "jerky" or "erratic"; this is normal.

(continued on next page)

Step	Procedure
3 (cont)	**NOTE 2** **At this point the AC register should be "0", indicating that the checksum comparison is correct. If the AC is not "0", repeat the Executive loading procedure.**
4	Set the switch register to 7600.
5	Press EXTD ADDR LOAD.
6	Press ADDR LOAD.
7	Press CLEAR, then CONT.

NOTE
Control returns to the Monitor after pressing CONT.

This completes the procedures required to build the initial system. Continue with the following procedures to build the specific sample system for checkout and experimental usage.

2.7 LOADING GENDAC TO CONFIGURE THE SAMPLE SYSTEM

The function of GENDAC is to configure the INDAC System for the specific requirements of each installation. To configure the sample system for checkout and initial experimentation, follow the log as indicated.

NOTE
The sample system contains the specific real-time clock purchased for this system, the high-speed paper-tape punch, and the TTY2. This configuration is specified regardless of the exact hardware actually connected.

Step	Procedure
1	Load GENDAC (Tape 12) in the high-speed reader. Start tape at leader portion (code 200).
2	Set the switch register to 3600.
3	Type LELD ↵
4	The LELD program will load GENDAC and return to the monitor after loading.

NOTE
Since paper-tape to disk operations are taking place, the loading may appear "jerky" or "erratic"; this is normal. About a third of the way through loading the tape the Teletype® will echo a carriage return/line-feed.

5	When loading is completed, LELD will return to the monitor.
6	Press HALT, then return switch to normal position.
7	Set switch register to 0200.

(continued on next page)

®Teletype is a registered trademark of Teletype Corporation.

Step	Procedure
8	Press ADDR LOAD, CLEAR, then CONT.
9	GENDAC will begin execution.
10	The initial dialogue from GENDAC types the version number and a reminder to the operator that if he is unsure of the response to a question, type "?".
11	GENDAC will then ask if this is to be a 4K system. The response will be "N", because you do not have a 4K system.
12	GENDAC will request the mode of operation to be used via the question "*OPT-", the response will be "B" for binary mode.
13	The next request is for the input device via the question "*IN-" the response will be "R:↵" for the high-speed reader.
14	GENDAC then types "↑", waiting for the binary tape to be loaded. Load the I/O Handlers (Tape 13) into the high-speed reader (start tape at leader code 200) and reply by striking CTRL/P (hold CTRL key and strike P key). GENDAC echos ↑P.
15	Answer all questions as in the log, except for selection of clocks; only respond to the correct computer and line frequency for your configuration. In general, if you do not understand what the routine in question is type "I" (inspect). The requests for TTY2 and PTP must be answered "Y" for the sample system.
16	When GENDAC reaches the end of the I/O Handlers tape it will type "END OF LIBRARY TAPE and *OPT-". Then respond with carriage return to complete GENDAC.
17	GENDAC will return to the Monitor System.

2.8 LOADING SAMPLE PROGRAM 2 USING THE EDITOR

Load the Sample Program as follows:

Step	Procedure
1	Place Sample Program 2 (Tape 14) in the high-speed reader. Make certain that all punched checksums are torn from the end of the tape.
2	Type EDIT↵
3	Follow the dialogue listed below: *OUT-S:TEST↵ * *IN-R:↵ * *OPT-B (NOTE: Carriage return is not used) *R↵
4	At this point, the EDIT program will read one buffer of information from the tape, stop reading, and return an asterisk (*).
5	Type E↵
6	The EDIT program will process the entire tape, close the TEST file on the disk and return to the Monitor System.

2.9 COMPILING SAMPLE PROGRAM 2

Compile the Sample Program as follows:

Step	Procedure
1	Type COMP↵ This command will begin execution of the Compiler. Follow the dialogue listed below: *OUT-S:TEST↵ * *IN-S:TEST↵ * *OPT-↵ **NOTE** **Both the input and output files are allowed to have the same name — this is a feature of the monitor system to simplify bookkeeping for the user.**
2	The Compiler will now begin processing. The Compiler is disk-bound and may appear to be in a loop accessing the disk; this is normal.
3	The Compiler will type out the information as shown in the log, with pauses between typeouts as the information is processed.
4	If the Compiler generates any error conditions, there are two possible causes for the errors: *a.* Sample Program 2 has not been loaded correctly, or the checksum at the end of the tape has not been torn off. If this is the case, go back to Paragraph 2.8, Loading Sample Program 2 Using the Editor. *b.* The compiler does not recognize a device name. If this is the case, the system was not configured properly with GENDAC. Refer to compiler error messages, Appendix D. You have now gained the experience necessary to proceed correctly again from Paragraph 2.3 or as follows: *(1)* Press CTRL/C to return to Monitor. *(2)* Load the MSUP tape from the high-speed reader using the following command string: .LOAD↵ *IN-R:↵ * ST=200↵ ↑↑ (User types CTRL/P after each ↑.) **NOTE** **The MSUP program will begin a series of questions to determine the operation required. Each question may be answered by typing a single letter followed by a carriage return (designated ↵). After the MSUP program is loaded, the first question is:** **LOAD, DUMP or VERIFY-** *(3)* Proceed with Paragraph 2.3, Step 17.

2.10 EXECUTING SAMPLE PROGRAM 2

To run the Sample Program proceed as follows:

Step	Procedure
1	Type SPUT↵ This command calls the System-Put Together program. SPUT will transform the Test program to an absolute configuration used during run time. SPUT will also automatically link to the Executive for execution. **NOTE** **If any errors are generated in this portion of the system, refer to Appendix E.**
2	Follow the dialogue listed below: *IN-S:TEST↵ *FILE START 0368 EXEC LOADED
3	At this point, the Executive is loaded and operating. This is the "idle" loop portion of the Executive waiting for commands. To begin Sample Program 2, follow the dialogue listed below: ↑A (CTRL/A) *↑DR #1↵ (CTRL/D followed by an "R", an optional space, a "#", a "1", and a "↵".)
4	The Sample Program will begin typing instructions. If this is the first time you are running this system, answer "N↵" to the first question asked. The dialogue should follow the sample log. You may now answer correctly when asked about the equipment you have. If you have TTY2, this device should be "ON-LINE" and connected for the test. Refer to Figure 2-1 for Sample Program 2 dialogue.

CHAPTER 3
PROGRAMMING THE INDAC 8/2 SYSTEM

3.1 INTRODUCTION

The INDAC language is an application language for IDACS 8 Data Acquisition and Control Systems. Although the language utilizes BASIC/FORTRAN like program statements, it also contains facilities for scheduling, for specifying industrial interface equipment and for servicing field interrupts; making it a much more powerful industrial application tool. The facility for specifying industrial user interface equipment is also available for special devices the user may wish to integrate into his system.

A process, whether it is:

Calibration and Testing
Material Preparation
Parts Manufacturing Control
Process Control (Batching)
Materials Handling
Machine Control

is controlled by an algorithm that can effect data collection; operate on process variables and parameters to develop control values (set points, on-off signals, open-close signals, etc.) and send these values to process control equipment; activate schedules; and accept operator inputs or commands. The INDAC language permits the user to create the algorithm that he needs using simple, easy to understand English statements.

If the algorithm is too large for the available computer core, INDAC's segmented structure permits the user to develop individual segments for each function of the algorithm. Each segment will be automatically maintained on mass storage. The user can then schedule these segments using INDAC's scheduling capability so that each segment is executed as though the whole algorithm were core resident. Since INDAC has a segmented structure and certain key words must be used to identify executable code from scheduling parameters, program skeletons are used in this chapter to place the segments into perspective for the reader.

Basic Skeleton

```
#1      .PHASE
        .ACTION

#2      .SNAP
        .PROCESS
        .END
```

The above skeleton shows the two basic segments of the INDAC language: the PHASE and the SNAP. The PHASE contains the key word .ACTION; the SNAP contains the key word .PROCESS. The last statement of the job is .END.

3.2 PROGRAMMING AN INDUSTRIAL ALGORITHM

An algorithm contains routines and/or subroutines for handling process control functions such as:

a. Conversion of process variables to engineering units

b. Conversion of calculated control values for driving process equipment

c. Decision making based on process variables and parameters

d. Initializing

e. Sampling

f. Limit checking

g. Normalizing

h. Scaling

i. Optimizing Control Parameters

The INDAC language provides the user with the necessary facilities for coding these and many other algorithmic functions. Some of the basic facilities the INDAC language provides are:

a. Arithmetic including:

Add
Subtract
Multiply
Divide
Exponentiation
Square Root

b. Bit isolation facilities for:

Testing
Setting
Resetting

c. Relational Comparison for process decisions including:

Greater than
Greater than or equal to
Equal to
Not equal to
Less than or equal to
Less than

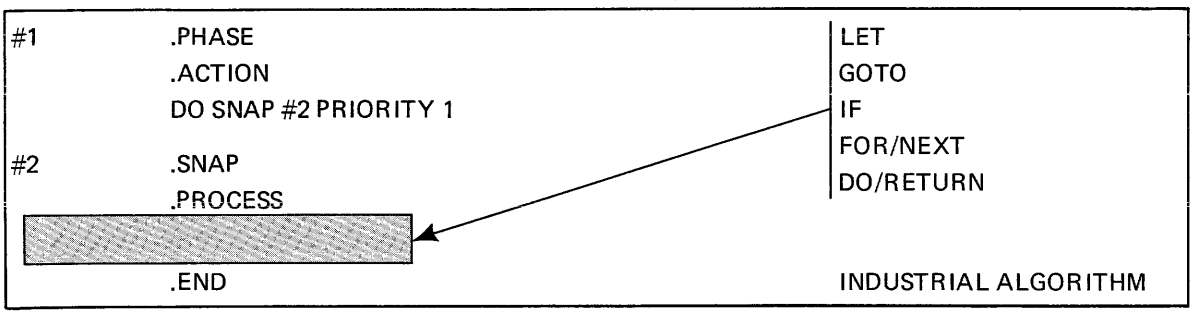
d. Multilateral branching

e. Looping

f. Subroutines

Skeleton No. 1 shows the basic structure of an INDAC program and where the algorithm must be placed. To compile an INDAC source program without error, the program must contain a .PHASE and a .SNAP. The PHASE must contain at least one scheduling statement following the key word .ACTION. The algorithm must appear in the .SNAP and must be bracketed by the key word .PROCESS. The last statement of the job is .END.

Skeleton No. 1 – The Algorithm



3.2.1 LET Statement (Assignment)

To resolve process parameters or process variables in developing control values or set points, "Arithmetic" will have to be performed in the algorithm. The LET statement allows the user to do arithmetic. The LET statement also allows the user to deal with the computer bits (Boolean type operations) as though he were opening or closing valves, turning solenoids on or off, setting indicators high or low, etc.

3.2.1.1 Arithmetic – Consider the following:

```
LET HILIMIT = 80.0
LET LOLIMIT = 60.0
```

Notice the use of the LET statement to assign a fixed parameter to a name. The assigned name can then be used in a limit checking algorithm. Assigning a value to a name rather than using the value itself in the program provides the option of changing the value at a later time. This option may be required in applications where dynamic limits prevail such as checking reject figures under varying production rates. To reduce HILIMIT from 80.0 to 70.0 one simply writes:

```
LET HILIMIT = HILIMIT - 10.0
```

The resulting value of an arithmetic expression can also be assigned to a variable name. For example:

```
LET RANGE = HILIMIT - LOLIMIT
```

The expression, HILIMIT - LOLIMIT, is evaluated during program execution and the result (+10) is transferred to the assigned name RANGE.

In arithmetic assignment statements, the assigned name and the expression or fixed parameter represents either integer or real quantities. Integer quantities require less core storage than real quantities because an integer quantity is represented by one computer word, while a real quantity is represented by three computer words.

The criteria for assigning a quantity a real or integer representation is its size. Integer quantities are limited to -2048 to +2047 inclusive. Real quantities are within the range $\pm 1 \times 10^{(\pm 99)}$.

To distinguish integer from real variables in the assigned name, the prefix I is used for all integer representation. Assigned names of real variables are prefixed by any alphabetic character except I.

The preceding examples of the LET statement define real variables and expressions. These statements can be rewritten to define integer variables and expressions as follows:

```
LET IHILIMIT = 80
LET ILOLIMIT = 60
LET IRANGE = IHILIMIT - ILOLIMIT
```

Notice that the integer constants, 80 and 60, do not use a period as do the real constants.

The LET statement allows the user not only to define and then operate on the data, but also to redefine and operate on process data collected from sensors. For example:

```
      LET TEMP = REFERENCE + SENSOR
      IF TEMP GE 90.0 THEN GOTO 21
      GOTO 50

21    LET IVALVE = 1
```

The LET statement in this example shows how two data items (REFERENCE and SENSOR) that have been collected from the process are added together and defined as TEMP. A decision, using the IF statement, is then made using the variable named TEMP.

Arithmetic expressions for the LET statement may be formed using the INDAC arithmetic operators detailed in Table 3-1. A comparison between the algebraic and the INDAC notations is also given in the Table 3-1. The primary differences are that the operator (symbol) used to denote arithmetic must appear, and all data must be on the same line (that is, $\frac{5}{2}$ notation not permitted).

Table 3-1
Basic Arithmetic Notation, Algebraic vs. INDAC

Notation		Arithmetic Operation	Differences in Notation
Algebraic	**INDAC**		
A + B	A + B	ADDITION	NONE
A - B	A - B	SUBTRACTION	NONE
A X B or A · B	A * B	MULTIPLICATION	An asterisk is used as the INDAC operator
$\frac{A}{B}$ or A ÷ B	A/B	DIVISION	A slash is used as the INDAC operator
A^2	A ↑ 2	EXPONENTIATION	An up arrow is used as the INDAC operator instead of a superscript

Two major factors that must be observed when writing arithmetic expressions in INDAC are:

a. Compatibility of *data type*

b. INDAC-established *precedence* in the execution of arithmetic operations.

Except for exponentiation, unlike *data types* (integer vs. real) cannot be used in the same expression. In exponentiation, the exponent is always an integer. The following examples illustrate both acceptable and unacceptable expressions based on data compatibility.

Acceptable	Comments	Unacceptable	Comments
HILIMIT + LOLIMIT	Both real variables	IHILIMIT + LOLIMIT	Mixed types, IHILIMIT is integer, LOLIMIT is a real variable
IHILIMIT + ILOLIMIT	Both integer variables	HILIMIT + ILOLIMIT	Mixed integer ILOLIMIT and real HILIMIT variable

(continued on next page)

Acceptable	Comments	Unacceptable	Comments
5 + IRANGE	Both integer types	5 + RANGE	Mixed integer number and real variable
5 + 10	Both integer numbers	5 + 10.2	Mixed integer and real numbers
12E5 + RANGE	Both real data types	12 E5 + IRANGE	Mixed real number and integer variable
1.2 ↑ 5	Mixed types permitted	1.2 ↑ 5.3	Power must be expressed as an integer number

Evaluation of arithmetic expressions is carried out according to the following *precedence* of execution:

Precedence	Operator	Operation
1	↑	Exponentiation
2	* /	Multiplication and division
3	+ -	Addition and subtraction

NOTE

Where operations have the same precedence of execution, they are evaluated from left to right, according to their occurrence within the expression.

The steps that are performed by INDAC in evaluating the expression 2 ↑ 2 * 3/2 + 1 according to the established hierarchy are:

Step 1	Raise 2 to the 2 power	$2^2 = 4$
Step 2	Multiply result by 3	$3 \times 4 = 12$
Step 3	Divide product by 2	$12/2 = 6$
Step 4	Add 1 to the quotient	$6 + 1 = 7$

Parentheses are used to group terms and factors of an arithmetic expression to specify a desired sequence (*hierarchy*) of execution.

Operations grouped within parentheses are always performed first. For example, in the expression:

5 * (1 + 3)

the grouped expression is evaluated first (that is, 1 + 3 = 4); then the remaining operation is carried out (that is, 5 * 4 = 20).

Multilevel grouping by parentheses is permitted in INDAC (for example, (((A+B) +C) *D)). In multilevel grouping, the evaluation of the expression is carried out starting with the innermost group and proceeding in sequence to the outermost group, as illustrated in the following example:

(...((((FIRST) SECOND) THIRD) FOURTH) ....)

The manner in which grouping affects INDAC's evaluation of an expression is illustrated by the following versions of the expression 2 ↑ 2 * 3/2 + 1.

Examples:

1. a. 2 ↑ (2 * 3)/2 + 1
   b. 2 ↑ (6) /2 + 1
   c. 64/2 + 1
   d. 32 + 1
   e. 33
2. a. ((2 ↑ 2) * 3) /2 + 1
   b. ((4) * 3) /2 + 1
   c. (12) /2 + 1
   d. 6 + 1
   e. 7
3. a. ((2 ↑ 2) * 3) / (2 + 1)
   b. ((4) *3) / (2 + 1)
   c. (12) / (3)
   d. 4

3.2.1.2 Boolean – Consider the following

```
LET IMASK = '1000
LET IOUT = '4000
```

Note the use of the LET statement to assign an octal constant to a name. In INDAC, octal constants are identified by an apostrophe preceding the number. Octal constants are used in Boolean operations to isolate or set bits in bit patterns. The assigned name must be an integer type because all INDAC bit patterns are limited to one computer word. The reason that octal constants rather than decimal constants are used in Boolean operations is that octal constants are easily identifiable with a specific bit pattern. A named variable may also be set to the result of a Boolean expression. For example, to test the state of a bit (0 or 1) one may write

```
LET IBIT = IMASK AND IPATTERN
```

The expression IMASK AND IPATTERN is evaluated on a bit-by-bit basis during program execution and the result is transferred to the assigned name. Remember that the IMASK isolates the bit of interest. If bit 2 of IPATTERN is set then the result will be 1000_8 because IMASK = '1000, but if bit 2 is not set, then the result will be 0.

NOTE

Engineering conventions prevail on the PDP-8 and bits are labeled left to right.

To set a bit to "1" one may write:

```
LET IPATTERN = IOUT OR IPATTERN
```

The expression IOUT OR IPATTERN is evaluated on a bit-by-bit basis during program execution and the result is transferred to the assigned name. Assuming that IOUT is set to 4000_8 (bit 0 is set to 1), the result (IPATTERN) will be that bit 0 will be set and the remaining bits will not be affected.

Boolean expressions for the LET statement may be formed using the following INDAC Boolean operators:

a. NOT
b. AND
c. OR

Some examples of Boolean expression using the above operators follow:

a. IAB OR ICD

b. IAB AND ICD

c. IAB OR '1777

d. IAB AND '1777

e. NOT IAB

f. NOT (IAB AND ICD)

In AND operations, the resultant of the corresponding bits of the logical data being combined is a "1" only if both corresponding bits are "1". To illustrate, given that:

a. ICHA = 010 100 111 000

b. ICHB = 101 000 101 000

the value of the dependent variable IRES in the expression:

LET IRES = ICHA AND ICHB

results in

IRES = 000 000 101 000

The AND operation is used to isolate one or more bits of a logical word for testing or to eliminate a given bit in the pattern so that their individual states (states of the devices and/or variables they represent) may be evaluated.

Isolating data using the AND operation is performed by ANDing the data word with a mask word designed to eliminate all unwanted data bits from the resultant. The mask word must contain a "0" corresponding to each unwanted data bit and a "1" for each desired bit. For example, to isolate the third bit of a word (that is, bit 2 in word format):

Bit No:	Bit Pattern												Octal Equivalent
	0	1	2	3	4	5	6	7	8	9	10	11	
Data Word	1	1	1	1	1	1	1	1	1	1	1	1	7777
AND													
Mask Word	0	0	1	0	0	0	0	0	0	0	0	0	1000
Resultant	0	0	1	0	0	0	0	0	0	0	0	0	1000

Mask words are expressed in octal forms as constants or as integer variables.

In OR operation, the corresponding bits of the words being combined result in a "1" if either or both of the combined bits are "1". To illustrate, given that:

a. ICHA = 010 100 111 000

b. ICHB = 101 000 101 000

the expression LET IRES = ICHA OR ICHB results in

IRES = 111 100 111 000

If either of the bits being ORed contains a "1", the result will contain a "1".

The NOT operator inverts or complements the value of any logical data or expression which it precedes. For example, if

ICHA = 010 100 111 000
NOT ICHA = 101 011 000 111

The order in which the logical operators are evaluated by INDAC is:

Precedence	Operation
1	NOT
2	AND
3	OR

As with arithmetic expressions, INDAC always evaluates quantities within parentheses first, regardless of the operation involved. This feature permits the user to dictate the order of evaluation by grouping within parentheses. In the following expression, the result is changed by the use of parentheses to alter the sequence in which the expression is evaluated. For example, the basic expression:

ISW1 AND ISW2 OR NOT IAB

may also be represented as

(ISW1 AND ISW2) OR (NOT IAB)

The meaning of the expression is not altered by adding the parentheses as shown above because the basic order of evaluation is not changed by the parentheses. The result of ISW1 AND ISW2 is still ORed with NOT IAB resulting in a "1" if a given bit is a "1" in both ISW1 and ISW2 or is a "0" in IAB.

Now consider the same expression with the parentheses placed as follows:

ISW1 AND (ISW2 OR (NOT IAB))

In this expression NOT IAB is still resolved first. Since the expression formed by the OR operator is in parentheses, it takes precedence over the expression formed by the AND operator, thereby changing the result of the whole expression. Now, the result of ISW2 OR NOT IAB is ANDed with ISW1 resulting in a "1" if a given bit is a "1" in ISW1 and the result of ISW2 OR NOT IAB is a "1". The result of ISW2 OR NOT IAB is a "1" if ISW2 is a "1" or if IAB is a "0". As shown above, the result of the expression was changed by adding parentheses to alter the sequence in which the expression was to be evaluated. Therefore, care must be taken in writing Boolean expressions to ensure that they are interpreted as intended.

3.2.1.3 Statement Evaluation Rules – Since assignment statements may cause the transfer of integer, real, or logical values into either integer or real variables or arrays, the evaluation of this type of statement is performed according to the following rules (where LET v = e):

Assigned Name (v) if	And Expression (e) is	INDAC does the following
Integer	Integer	Transfers the result of e, unchanged, to v
Integer	Real	Truncates any fractional part of e and transfers it as an integer to v
Integer	Logical	Transfers e, unchanged, to v

(continued on next page)

Assigned Name (v) if	And Expression (e) is	INDAC does the following
Real	Integer	Transforms value of e to a real data form and transfers it to v
Real	Real	Transfers the result of e, unchanged, to v
Real	Logical	Not permitted

3.2.2 GOTO Statement (Branching)

The GOTO Statement provides the user with a convenient way to transfer control from one part of his program to another. The statement can be used to link procedures in the algorithm. Control can be transferred unconditionally using the following statement:

```
          GOTO 21
            ⋮
21        LET ITEMP = IREFERENCE + ISENSOR
```

This statement, which references a statement label, can be used anywhere in the process algorithm to transfer control to another part of the algorithm.

NOTE

A label is an integer number from 1 to 999 that precedes an executable statement so that it can be referenced by a GOTO statement.

Control can also be transferred conditionally using the following statement:

```
GOTO (21,25,29), INDEX
```

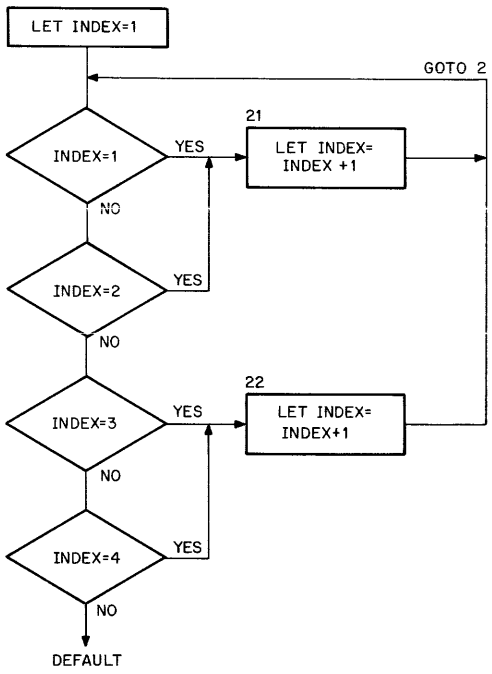
This statement is the computed GOTO statement that can be used to transfer control to one of many possible alternative procedures based on the value of an integer control parameter or index (INDEX). Statement labels identifying the starting point of alternative procedures are enclosed in parentheses. There is virtually no limit to the number of labels that can be referenced. If the index of the computed GOTO statement is 0, negative, or greater than the number of referenced labels a default to the statement following the GOTO will occur. The integer quantity INDEX referenced in the above example may be derived in a number of different ways depending on what the user wishes to do. One way to generate the index is shown below:

```
          LET INDEX = 0
2         LET INDEX = INDEX + 1
          GOTO (21,25,29), INDEX
            ⋮
          GOTO 2
```

These statements set INDEX to 0 and then increment INDEX by one. When used as one control parameter of the computed GOTO statement, INDEX acts as a switch causing a transfer to the statement labeled 21 if INDEX is 1, or to the statement labeled 25 if INDEX is 2, or to the statement labeled 29 if INDEX is 3. If INDEX is either negative, 0 or greater than 3 the computed GOTO statement defaults to the next statement in the program.

The following program and flow chart illustrate the operation and power of a computed GOTO statement.

```
1          LET INDEX = 1
2          GOTO (21,21,22,22),INDEX
           default
21         LET INDEX = INDEX + 1
           GOTO 2
22         LET INDEX = INDEX + 1
           GOTO 2
```



3.2.3 IF Statement (Conditional Testing)

In any process algorithm, decisions have to be made based on collected or calculated process data so that the desired controls can be activated.

The IF statement is a control statement that allows the user to compare process data (quantities or expressions). The statement uses the words IF and THEN GOTO to select another process procedure (program sequence) according to the outcome of the comparison. The statement employs a relational operator to make the comparison and a GOTO statement to branch to the alternative procedure.

```
IF TEMP GR 90.0 THEN GOTO 4
```

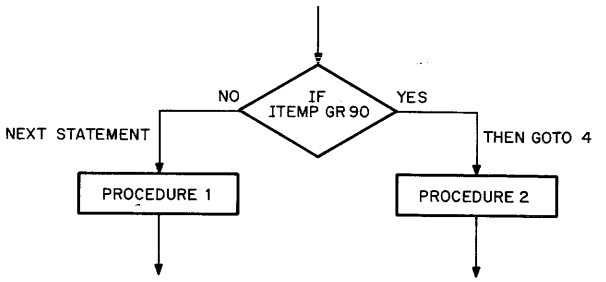
This version of the IF statement uses an unconditional GOTO statement. The real variable TEMP may be the product of an arithmetic operation using the LET statement or may represent data collected from the process. GR is a relational operator meaning greater than. Six different relational operators are at the disposal of the user. These are defined in Table 3-2.

Table 3-2
Relational Operators

Operator	Symbol	Meaning
GR	$>$	Greater than
GE	$\geqslant$	Greater than or equal to
EQ	$=$	Equal to
NE	$\neq$	Not equal to
LE	$\leqslant$	Less than or equal to
LS	$<$	Less than

Relational operators can only be used with the IF statement. The sample IF statement above compares real quantities. Real assignments require three computer words of storage and should be used only when the quantities or expressions can not be represented in one computer word (an integer quantity between -2048 and +2047 inclusive). The IF statement can be rewritten to compare integer quantities as follows:

```
IF ITEMP GR 90 THEN GOTO 4
```



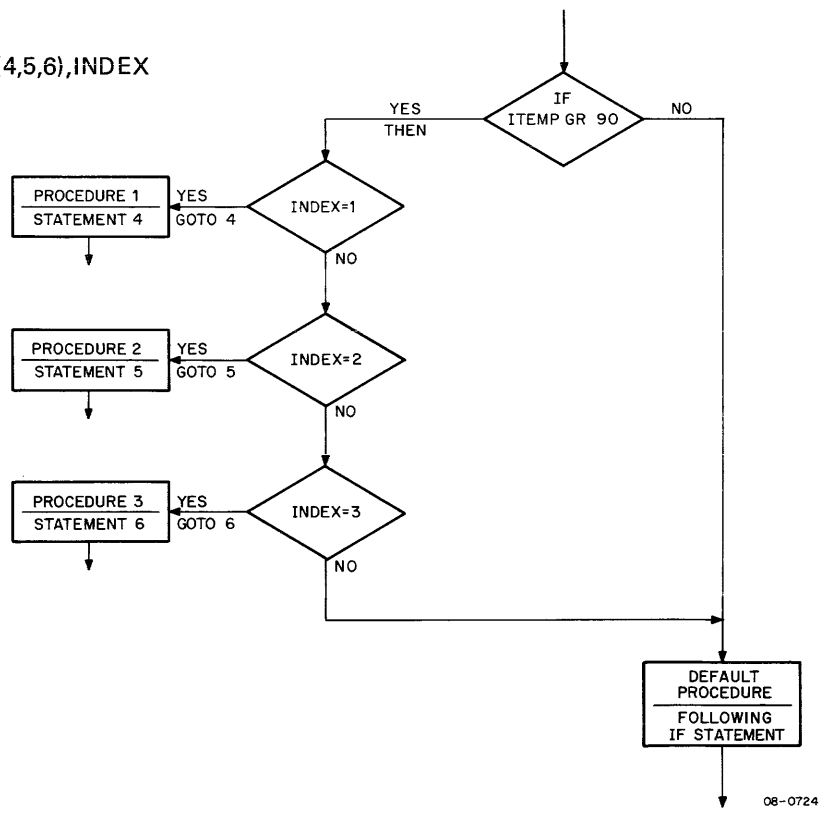
Notice that the integer constant (90) does not use a period but the real constant (90.0) does. Both sample IF statements above use an unconditional GOTO statement. If the quantity represented by TEMP (ITEMP) is greater than 90.0 (90), then program control is automatically given to the statement labeled 4; otherwise, the next statement in the program is executed.

NOTE

A label is an integer number from 1 to 999 that precedes an executable statement so that it can be referenced by a GOTO statement.

A computed GOTO can be used in the IF statement instead of one unconditional GOTO. This capability allows the user to resolve decisions where there are multiple alternatives in the algorithm.

```
IF ITEMP GR 90 THEN GOTO (4,5,6),INDEX
```



If the value of ITEMP is greater than 90, then the value contained in INDEX is used as a control parameter to branch to labeled statements. (Refer to GOTO description.)

If INDEX is neither 1, 2 nor 3, or if ITEMP was not greater than 90, then the statement following the IF statement is executed (called a default condition).

The quantities and/or expressions being compared in an IF statement may be integer or real, but not mixed; for example:

a. ISW1 GR ISW2 (acceptable, both terms are integer)

b. ABSA GR ABSB (acceptable, both terms are real)

c. ISW1 GR ABSA (not acceptable, terms are mixed types)

The terms of a relational expression may be simple or complex. For example:

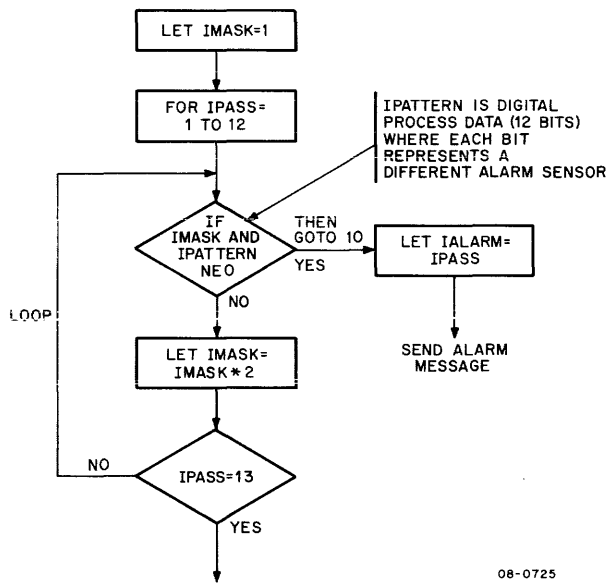
a. (ISW1-5) GR (ISW2-5)

b. (12E-05*ABSA) GE (128E05*ABSB)

3.2.4 FOR/NEXT Statements (Looping)

Repetitive operations can be implemented by using the FOR/NEXT statements to form a loop. For example:

```
1       LET IMASK = 1
        FOR IPASS = 1 TO 12 STEP 1
        IF IMASK AND IPATTERN NE 0 THEN GOTO 10
        LET IMASK = IMASK * 2
        NEXT IPASS
          :
10      LET IALARM = IPASS
```



Notice that the FOR statement precedes a sequence of instructions that are to be repeated. The FOR statement defines a control variable (IPASS) and specifies the number of times the instructions following are to be repeated. The number of times the loop is to be repeated is defined by an initial parameter, a terminal parameter, and an incremental parameter (1 TO 12 STEP 1). The NEXT statement is the last statement in the loop. This statement

signals the end of a pass through the instructions and initiates a test to determine if the number of passes specified have been executed. After the last pass through the loop, control is automatically released to the statement following the NEXT statement. The FOR/NEXT flow chart illustrates the operation of the FOR/NEXT loop example presented on the previous page. The example illustrates how a FOR/NEXT loop can be used to test a particular computer word to see which bit is set. In this example, the variable IPATTERN which was collected from the process is to be tested. This variable is a 12-bit computer word, where each bit is associated with a different process alarm sensor. If during the test, a bit is found to be set, a unique alarm message can be issued using IALARM as an index. Notice that the integer parameter IMASK is set to "1" before entering the loop. IMASK is then multiplied by 2 before repeating the next pass through the loop, resulting in an IMASK variable that has a progressive value of the powers of two. The IF statement in the loop is used to test each bit. IMASK is used to isolate bits in IPATTERN using the AND operation; the relational operator NE is used to test if the specific bit is set ("1"). Table 3-3 shows the powers of two progression with the octal equivalent associated with the alarm index.

Table 3-3
Mask Generation

IPASS/ALARM BIT	IMASK	
	Decimal	Octal
1	1	0001
2	2	0002
3	4	0004
4	8	0010
5	16	0020
6	32	0040
7	64	0100
8	178	0200
9	256	0400
10	512	1000
11	1024	2000
12	-2048	4000

Once a bit is found to be set, the control variable (IPASS) of the FOR/NEXT loop can be used to inform an operator of the indicator in alarm or to take programmatic action in the process.

NOTE

PDP-8 engineering conventions prevail. The bits of a PDP-8 computer word are numbered from left to right and not from right to left reflecting the programming convention of the binary weight of the bits. Therefore, when the algorithm is developed, the user must keep this characteristic in mind.

In the preceding example, the FOR statement specified an incrementing parameter of 1 (STEP 1) for the loop control variable IPASS. This resulted in 12 passes through the Loop because after each pass IPASS was incremented by 1.

NOTE

The incrementing parameter (STEP) may be omitted for increments of 1. However, if increments other than 1 are desired, STEP must be included to specify the increment.

If we change the incrementing parameter to 3 (STEP 3) then the FOR/NEXT loop would only make 4 passes and would test alarm sensors 1, 4, 7, and 10 because the control parameters will progress from 1 to 4 to 7 to 10 and then release control.

In addition, the preceding example used fixed parameters (integer quantities) in the FOR statement to define the number of times the loop was to be executed. Named integer variables can also be used to define the number of passes to be performed. For example:

```
1       LET IMASK = 1
        FOR IPASS = ISTART TO ISTOP STEP INCREMENT
        IF IMASK AND IPATTERN NE 0 THEN GOTO 10
        LET IMASK = IMASK * 2
        NEXT IPASS
          :
10      LET IALARM = IPASS
```

Using named integer variables to define the number of passes the loop is to execute provides the option to change the FOR statement under program control. This option may be helpful, as related to the example, to selectively test alarm messages based on process variable evaluations. If the following assignment statements preceded the FOR/NEXT loop, then the loop would make 6 passes, thereby permitting any of the 6 alarm sensors to be tested:

```
LET ISTART = 1
LET ISTOP = 6
LET INCREMENT = 1
```

The previous examples of the FOR/NEXT loop illustrated how 12 indicator alarms can be tested. Many industrial applications contain more than 12 such alarms. In these applications, more computer words would be required for computer storage. The simple FOR/NEXT loop developed in previous examples only tested the bits of a single word. However, this single loop can be incorporated (inserted) into a larger loop to test additional words. Inserting one loop into another loop is called *nesting*. In the following example, the parameter IWORD is used to select the word for the test and the parameter IPASS is used to test the individual bits of the word. The following example illustrates how 5 alarm indicator words can be tested:

```
        FOR IWORD = 1 TO 5
        LET IMASK = 1
        FOR IPASS = 1 TO 12
        IF IMASK AND IPATTERN (IWORD) NE 0 THEN GOTO 10
        LET IMASK = IMASK * 2
        NEXT IPASS
        NEXT IWORD
10      LET IOFFSET = (IWORD -1) * 12
        LET IALARM = IOFFSET + IPASS
```

In the earlier example, the alarm number was the same as the pass number. In this example, the alarm number, which could be from 1 to 60 (5 computer words) is calculated. An offset based on the IWORD parameter is established. This offset assumes a value that is based on the value of IWORD (1 through 5) when a set bit is found. IOFFSET can assume the following values:

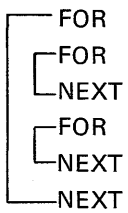
0 when IWORD = 1
12 when IWORD = 2
24 when IWORD = 3
36 when IWORD = 4
48 when IWORD = 5

With this offset defined, the alarm number can be determined by simply adding IOFFSET and IPASS.

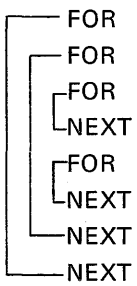
Nesting is allowed as long as the range of one loop does not cross the range of another loop. The following diagrams illustrate acceptable nesting techniques.

Acceptable Nesting Techniques

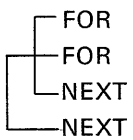
Two-Level Nesting



Three-Level Nesting



Unacceptable Nesting Techniques



An inner (nested) loop is cycled through its entire range for each cycle of the outer (primary) loop. If, for example, the primary loop has an iteration of 5 cycles and the inner loop an iteration of 10 cycles, the nested loop will have completed a total of 50 cycles when the outer loop reaches its limit of 5 cycles.

3.2.5 DO/RETURN Statements (subroutines)

When particular process variables or parameters are evaluated several times, or a sequence of instructions must be executed several times in an algorithm, the user may code these operations as a subroutine that can be called by the algorithm when needed. Two types of subroutines are permitted in INDAC:

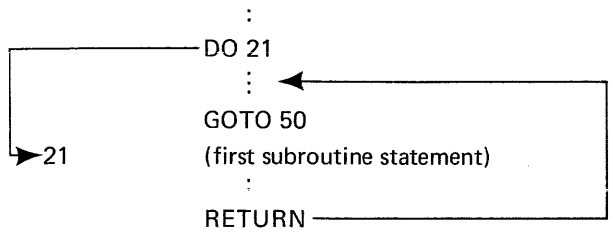
a. Internal

b. External

Internal subroutines are an integral part of the algorithm, while external subroutines are stored on the disk separately. External subroutines are discussed later in this chapter. An internal subroutine is called into operation by the DO statement referencing a label. For example:

```
DO 21
```

The statement labeled 21, which should be the first instruction in the internal subroutine, and all statements until a RETURN statement is encountered are then executed. Executing the RETURN statement terminates the subroutine and returns control to the statement directly following the call DO 21. The following example shows the techniques for installing a subroutine into the algorithm.



Other uses of the DO/RETURN statements are discussed later in this chapter.

3.3 MESSAGE AND LOGGING CAPABILITIES

Messages, Reports, and Logs may be required to inform the operator of alarm conditions, process states, process variables, and parameters or any other information that may be significant in implementing and keeping the process running smoothly. To illustrate the reporting and logging capabilities of the INDAC language, the examples for testing alarm indicators developed in programming an industrial algorithm (Paragraph 3.2) will be expanded.

NOTE

Skeleton No. 2 shows the basic structure of an INDAC program and shows where the algorithm and information pertaining to the Message, Report or Log is to be placed. A sample alarm test routine is included in the algorithm section.

Skeleton No. 2 – Sample Alarm Test Routine

```
#1          .PHASE
            .ACTION
            DO SNAP #2 PRIORITY 1

#2          .SNAP
            [          ]  <-- .STORAGE
                          <-- .FORMAT
                          <-- .HEADER
            .PROCESS
            [          ]  <-- GET/SEND
            LET IPATTERN = '0100
5           LET IMASK = 1
            FOR IPASS = 1 TO 12
            IF (IMASK AND IPATTERN) NE 0 THEN GOTO 21
            GOTO 10
21          LET IALARM = IPASS
            [          ]  <-- GET/SEND
10          LET IMASK = IMASK*2
            NEXT IPASS

            .END                                    MESSAGE AND LOGGING
```

Messages, Reports, or Logs can be sent to the Teletype, the paper-tape punch, or other ASCII type output devices using a simple SEND statement in the algorithm. The SEND statement must identify the device name, the message to be sent, and a data list if data is to be sent with the message.

```
SEND (TTY,#101)
```

This statement sends the message tagged #101 to the Teletype. Notice the device name and the tag are separated by a comma and are enclosed in parentheses. A message without accompanying data is identified by the .HEADER statement as follows:

```
#101        .HEADER
            ALARM INDICATOR REPORT
```

The .HEADER statement must always be tagged so that the message can be referenced in the SEND statement. The message declared in the .HEADER statement can occupy more than one line.

To send the same message to another ASCII type output device, it is only necessary to change the device name in the SEND statement. For example, to send to the paper tape punch:

```
SEND (PTP, #101)
```

Operators consoles may not be attended so when a message is sent, it may be desirable to report the time of day the message was sent. For this reason and for scheduling purposes, the INDAC System contains a real-time clock (CLOCK) that maintains the time of day. The time of day is maintained as integer quantities in three computer words of storage; one for hours, one for minutes, and one for seconds. Before the time of day can be reported it must first be obtained. The time of day can be obtained using a simple GET statement in the algorithm. The GET statement must identify the device name (pseudo-device) and a data list of three integer variables.

```
GET (CLOCK) ICLK1, ICLK2, ICLK3
```

This statement gets the time of day from the real-time clock and stores it in the specified data list. Notice each integer variable specified in the list is separated by a comma. Now the time of day may be sent with the alarm message using a simple SEND statement as before. This time, however, a data list must be included in the statement to send the time of day with the message

```
SEND (TTY,#102) ICLK1, ICLK2, ICLK3
```

This statement sends the message tagged #102 and the data list to the Teletype. Notice the use of parentheses and commas as described earlier.

A message with accompanying data is identified by a FORMAT statement. INDAC offers two types of .FORMAT statements: picture type and declarative type. A picture .FORMAT statement for the alarm message and accompanying data is written as follows:

```
#102        .FORMAT
            "ALARM INDICATOR REPORT AT: "XX" : "XX" : "XX
```

The .FORMAT statement must always be tagged so that the message can be referenced in the SEND statement. Notice the use of quotation marks and Xs in the message. The quotation marks are used to bracket characters, thereby identifying the characters of the message to be printed. Each digit of the data to be printed is represented by an X. Military time representation is used in INDAC.

0 – 23 for hours
0 – 59 for minutes
0 – 59 for seconds

If the time is 11:30 a.m., the above .FORMAT statement causes the time to be printed as follows:

```
ALARM INDICATOR REPORT AT:  11:30:00
```

Notice digits for hours, minutes, and seconds are printed where the Xs were placed in the .FORMAT statement. Since the colons (:) are declared as printable characters of the message, the colons are also printed. To send the same message to another ASCII type output device, it is only necessary to change the device name in the SEND statement. For example:

```
SEND (PTP,#102) ICLK1, ICLK2, ICLK3
```

The message declared in the picture .FORMAT statement cannot exceed one line of a printing device. If the line length of the input device (device program is prepared on) is shorter than the output device (device to which the message is sent), then a continuation line can be used to prepare a message that will extend the full line length of the output device. A continuation line is indicated by typing an up-arrow (↑). Another limitation of the picture type .FORMAT statement is that it can only be used to provide formats for integer or real decimal data having a maximum of 15 digits including sign.

The declarative .FORMAT statement is more powerful than the picture type .FORMAT statement. The declarative .FORMAT statement permits the use of a virtually unlimited number of continuation lines (indicated by an up-arrow, ↑) and permits the user to specify the data to be output as fixed point, floating point, signed, unsigned, octal, binary, exponential, or ASCII representation. The statement also permits the user to specify special characters for CR/LF Editing and for control of the operator console modes. A declarative .FORMAT statement for the Alarm message and accompanying data is written as follows:

```
#103        .FORMAT ("ALARM INDICATOR REPORT
↑           AT:  ",R3.0":",R3.0":",R3.0)
```

The ASCII string to be printed is bracketed by quotation marks as in the picture FORMAT statement. The output data format however, is represented differently. The commas indicate the start of a data format declaration. Also, notice that the entire message and data declaration is enclosed in parentheses. In the above example, R3.0 means the data is to be output as 2 unsigned digits with no decimal point. R3.1 means that the data is to be output as 2 unsigned digits with the decimal point between the two digits. The SEND statement is the same whether the message is defined by a picture or a declarative .FORMAT statement, that is:

```
SEND (TTY,#103) ICLK1, ICLK2, ICLK3
```

The sample algorithm in the skeleton (starting with LET IPATTERN = '0100) is similar to those developed in programming an industrial algorithm, Paragraph 3.2. Notice that the sample algorithm in the skeleton is structured to continue testing after an alarm is found.

When an alarm is found (a bit is found to be set) by the IF statement in the loop, the statement labeled 21 is then executed to assign the IPASS parameter to IALARM. The value of IALARM will then be the number of the sensor that caused the alarm. The following SEND statement can then be used to report the sensor number that caused the alarm.

```
SEND (TTY,#104) IALARM
```

As explained earlier, the tag should reference a .FORMAT statement since data is to accompany the report. The statement may be of the picture or declarative type. For example:

Picture Type

```
#104        .FORMAT
            "SENSOR NO. " XX
```

Declarative Type

```
#104        .FORMAT ("SENSOR NO.", R3.0)
```

NOTE

If no FORMAT is declared in the SEND statement, the data list will be output as real data in exponential type notation of the following form (16 data items per line).

±0.XXXXXXE±XXXX

Some of the statements developed in this paragraph have been placed in perspective for the reader in the following update to skeleton No. 2 for instructive purposes. Notice that all message declarations .HEADER and .FORMAT statements are located between the key words .SNAP and .PROCESS, and all executable statements are located in the algorithm section between .PROCESS and .END.

Skeleton No. 2 — Header/Format Details

```
#1          .PHASE
            .ACTION
            DO SNAP #2 PRIORITY 1
#2          .SNAP
#102        .FORMAT
"ALARM INDICATOR REPORT AT: "XX":"XX":"XX
#104        .FORMAT ("SENSOR NO. ", R3.0)
            .PROCESS
            GET (CLOCK) ICLK1, ICLK2, ICLK2
            SEND (TTY, #102) ICLK1, ICLK2, ICLK3
            LET IPATTERN = '0100
5           LET IMASK = 1
            FOR IPASS = 1 TO 12
            IF (IMASK AND IPATTERN) NE 0 THEN GOTO 21
            GOTO 10
21          LET IALARM = IPASS
            SEND (TTY, #104) IALARM
10          LET IMASK = IMASK*2
            NEXT IPASS
            .END
```

In the previous examples illustrating the message and logging techniques of the INDAC language, the parameters IMASK and IPATTERN, and the storage location for the time of day (ICLK1, ICLK2 and ICLK3), were defined in the algorithm. Parameters (or constants) and storage locations can also be defined with the .STORAGE statement.

The following .STORAGE statement defines the integer parameters IPATTERN, IMASK and integer storage locations for the time of day.

```
.STORAGE IPATTERN/1/,
↑IMASK (1,12)/'0001,'0002,'0004,'0010,'0020,'0040,
↑'0100,'0200,'0400,'1000,'2000,'4000/,ITIME (1,3)
```

Continuation lines, indicated by an up-arrow (↑), are permitted in the STORAGE statement.

Notice the use of names, numbers in parentheses and numbers between slashes. The name identifies the parameter or storage location so that it can be referenced in the algorithm. The two numbers separated by a comma in parentheses define the number of storage locations (commonly referred to as a dimensioned array) to be defined for the named variable. When the number of words are not defined, as in IPATTERN, only one storage location is set aside. A storage location is one computer word for integer variables and three computer words for real variables. The number enclosed by slashes are the values to which the storage locations of the array are to be set. For real arrays the preset values must contain a period. If no preset values are specified (as in ITIME) the locations will be automatically set to 0. Notice that the integer storage locations defined in the .STORAGE statement can be preset with decimal or octal quantities. Having the parameters and storage locations defined in the .STORAGE statement, a simpler code can be developed for the algorithm as follows:

```
          ⋮
          GET (CLOCK) ITIME
          SEND (TTY, #102) ITIME
          FOR IPASS = 1 TO 12
          IF IMASK (IPASS) AND IPATTERN NE 0 THEN GOTO 21
          GOTO 10
21        LET IALARM = IPASS
          ⋮
10        NEXT IPASS
          .END
```

The statement

```
GET (CLOCK) ITIME
```

obtains the time of day from the real-time clock as before but now stores the three words in the predefined array called ITIME. Therefore, to report the time of the alarm message the following SEND statement is used.

```
SEND (TTY, #103) ITIME
```

No change is required in the .FORMAT statement when storage locations are predefined. All three words of ITIME are automatically transferred since the whole array is declared in the data list of the SEND statement.

Notice that the statements that defined IPATTERN and IMASK and the statement that generated the powers of two progression in the earlier algorithm are no longer needed. Removing these statements makes the Algorithm faster to operate since the multiply operation is no longer used. Also notice that the IF statement has been changed slightly. The statement now reads:

```
                IF IMASK (IPASS) AND IPATTERN NE 0 THEN GOTO 21
```

IMASK is now subscripted with IPASS to step through the IMASK array defined in the .STORAGE statement. Subscripted variables provide the user with additional computing capabilities for dealing with lists, tables, matrices, or any set of related variables defined in the .STORAGE statement. In INDAC, variables are allowed one subscript. A named variable followed by a constant or a named integer variable in parentheses indicates the location of the variable in the list. This facility allows the user to reference any data item or storage location defined in the .STORAGE statement. In the previous example, IMASK is subscripted with IPASS and IPASS assumes the values 1 through 12 as the loop is executed. Therefore, for each pass through the loop, the corresponding IMASK parameter in the list defined in the .STORAGE statement is used in the IF statement.

The .STORAGE statement and Alarm test algorithm developed in the previous paragraph have been placed in perspective for the reader in the following update to Skeleton No. 2 for instructive purposes. Notice that the .STORAGE statement is located between the key words .SNAP and .PROCESS along with the .FORMAT and .HEADER statements; the executable statements are located in the algorithm section between the key words .PROCESS and .END.

Skeleton No. 2 — Storage Details

```
#1              .PHASE
                .ACTION
                DO SNAP #2 PRIORITY 1
#2              .SNAP
#102            .FORMAT
"ALARM INDICATOR REPORT AT: "XX": "XX":"XX
#104            .FORMAT ("SENSOR NO: ", R3.0)
                .STORAGE IPATTERN/1/,IMASK(1,12)/('0001,'0002,
↑'0004,'0010,'0020,'0040,'0100, '0200,'0400, '1000, '2000,
↑'4000/,ITIME (1,3)
                .PROCESS
                GET (CLOCK) ITIME
                SEND (TTY,#102)ITIME
5               FOR IPASS = 1 TO 12
                IF IMASK (IPASS) AND IPATTERN NE 0 THEN GOTO 21
                GOTO 10
21              LET IALARM = IPASS
                SEND (TTY, #104)IALARM
10              NEXT IPASS
                .END
```

The following paragraphs serve as a summary and provide additional details of the .HEADER, .FORMAT, and .STORAGE statements.

3.3.1 .HEADER Statement

The specification of messages or headings to be printed for tabular and other types of reported information is accomplished by .HEADER statements. This type of statement, referenced by a SEND statement, consists of two or more lines with the initial line having the form:

```
#t          .HEADER
```

where t presents a mandatory reference tag. The following lines contain alphabetic and alphanumeric information arranged exactly as they are to appear when printed.

The .HEADER statement is terminated by any line whose first nonblank character is a period.

The following example illustrates the use of .HEADER statements:

```
#2          .HEADER
                                        COMPUTER COOLING ANALYSIS
            SAMPLE      CLOCK       BANK    BANK    BANK    TEMP
            NR          TIME        A       B       C       DEG
```

3.3.2 .FORMAT (Picture) Statement

This .FORMAT statement is used to specify message and/or the format in which designated output data is to be printed. These statements are referenced by SEND statements for formatting purposes.

The picture .FORMAT statement consists of two lines, with the initial line having the form:

```
#t          .FORMAT
```

Where t is a mandatory reference tag. The second line of the .FORMAT statement contains a message, if any, and an actual representation of the data as it is to appear in its printed form. The representation will consist of:

a. Declaration of space characters (blanks). Spaces are regarded as meaningful elements in a picture .FORMAT statement; they specify the actual spacing to be established between nonblank format elements. Tabs are retained as TAB characters and are effective only on Teletypes with TAB capability.

b. Declaration of number formats. Both real and integer numbers are represented by:

1) A letter S for the sign (if used)

2) A letter X for each digit (15 digit maximum — INDAC maintains 6 digits of accuracy), and

3) A decimal point which cannot precede the sign (letter S).

For example, the format for a 6-digit signed integer would be represented by:

```
SXXXXXX
```

A 6-digit real number with a 3-digit integer portion is represented by:

```
SXXX.XXX
```

c. Declaration of Text. Titles, comments, etc., may be specified for printout of a .FORMAT statement by enclosing each text line segment within quote (") marks. These quote marks are not printed; they identify and delimit text units.

The following are examples of text declarations:

"FURNACE" "VALUE" "NO." "T104"

Thus the information contained in the second line of a declared .FORMAT statement consists of number and text declarations separated by spaces. For example:

#10 .FORMAT

"TEMP"␣SXXX.XX␣"DEGREES CENTIGRADE"

NOTE

In the examples given, the symbol ␣ represents unit spaces that would be entered into systems using the INDAC keyboard space bar.

The .FORMAT statement may contain continuation lines that extend the number of characters to be printed on a single line. For example:

#5 .FORMAT

"SENSOR NO. 1" "TEMP" SXXX.XX "SENSOR NO. 2 TEMP"

↑ SXXX.XX

A format statement that specifies a group repeated in a serial manner throughout the line (for example, SXXXX SXXXX SXXXX . . . .) may be written using a shorthand technique. This technique consists of placing the format group to be repeated (including interposing space units) within parentheses and preceding the enclosed group with an integer that specifies the number of times the group is to be repeated.

For example, the line:

SXXX.XX SXXX.XX SXXX.XX

may be expressed:

3 (SXXX.XX)

The shorthand technique may be used to reduce statements which specify mixed repetitive and nonrepetitive format groups. For example, the statement:

"TEMP" SXX SXX SXX "PRESSURE" XX.XX XX.XX XX.XX

may be written:

"TEMP" 3(SXX) "PRESSURE" 3(XX.XX)

3.3.3 .FORMAT (Declarative) Statement

A declarative form of .FORMAT statement is available with expanded capability. The statement has the form:

#t .FORMAT (S_1, S_2, ..., S_n)

Where #t is a mandatory tag, the left parenthesis is the first non-blank character following the ".FORMAT" and S is a FORMAT element containing either an ASCII string, a data field specification, or a special FORMAT character. Blanks may be used for readability but will be ignored by the Compiler (except in text strings). The declarative format string may contain continuation lines, and repeat groups as in the picture type format. Repetition of format groups is restricted as in picture type format to a single level of nesting. Declarative format statements are terminated by the first right parenthesis not terminating a repeat group. A synopsis of the declarative format is in the following table.

Table 3-4
Declarative .FORMAT Specifications

Control	Function	Sample	Sample Output
	Output Specifications		
Fw.d	Output a real or integer variable as a signed field, w characters wide, including sign and period, with d decimal places.	F7.3	+99.999
Rw.d	Same as "Fw.d" except that field is unsigned.	R7.3	999.000
E	Output real data in exponential notation. Requires 15 spaces on output.	E	+0.123456E-0003
O	Output integer data as octal number	O	7654
nX	Output n blanks, if n not specified one (1) assumed	2X	
nW	Transmit n words, each as a twelve bit word — no conversions	2W	
	Special Format Characters		
"	As in picture format used to delimit an ASCII string to be generated.		
,	Delimits FORMAT elements.		
/	Generate a CR/LF in format.		
#	Generate a CR in format. Does not generate LF.		
$	Suppress CR/LF at termination of format. Must be followed by terminating parenthesis of statement.		
?	Suppress CR/LF at termination of format but leaves console in "Operator" mode. Must be followed by terminating parenthesis of statement.		

NOTE
Integer data is limited to four digits (-2048 to +2047). Real data is accurate to six digits; data output greater than those limits should be viewed with these restrictions in mind. Data output greater than 15 digits will generate a compiler error.

3.3.4 .STORAGE Statement

In many applications, INDAC is required to accumulate and store large amounts of identifiable numeric data. Acquired and computed data as well as constants and other items entered by the user must be stored in lists or tables (arrays) that are defined by the user. In addition to defining arrays, the most important function of .STORAGE is to supply common areas for data communication.

To define required arrays, the user must write a .STORAGE statement and follow it with a listing of symbolic names unique to each required array. Once defined, an array name may be referenced by a statement (GET, SEND, IF, LET) each time the data is to be entered into or read from that array.

3.3.4.1 Development of Array Names — Observe the following rules in developing an array symbolic name:

a. It must be unique within the program.

b. It may consist of an unlimited number of alphabetic or alphanumeric characters.

(continued on next page)

c. The first character must be alphabetic and must identify the type of data or constant (real or integer) which the array is to store. The first character must be:

1. I for an integer array. (For example, IAMP, IH10, I256, etc.)
2. Any letter but I for a real array. (For example, AMPS, B526, ZZZZ, etc.)

3.3.4.2 Specification of Array Size – The use of a symbolic name in a .STORAGE statement reserves only one storage element (one word for integers, three words for real numbers). When multielement storage is required, the user must specify the size of the array by adding an enclosed range of elements having the form (1, N) to the array's symbolic name. The subscript is interpreted as:

1 = The first element in the array
N = An integer representing the last element in the array (N > 1)

For example, an integer array for the storage of 50 sampled voltage levels could be specified as:

```
ISVOLTAGE (1,50)
```

3.3.4.3 Specification of Array Elements.– Individual elements of an array can be specified by using the symbolic array name followed by an enclosed integer that represents the number of the element within the array. For example, given the designation of a 50-element array:

```
IABC (1,50)
```

The 30th word element is specified by the expression:

```
IABC (30)
```

The following examples illustrate the manner in which the .STORAGE statement is written:

Statement	Meaning
.STORAGE ISUM	Single-element array ISUM
.STORAGE ISUM (1,20)	20-element array ISUM
.STORAGE ISUM (1,20),I20,C,D	20-element integer array ISUM, single-element integer array I20, and single-element real arrays C and D

3.3.4.4 Specification of an Array Window – INDAC provides a unique technique that permits the user to specify a subset of an array or a superset spanning a number of arrays.

Dimensioned arrays consist of a sequential group (list) of word locations within the processor memory; to specify a particular subset of locations within an array, the programmer:

a. Lists the name of the array, (for example, IAMP)

b. Indicates the first element of the subset (for example, IAMP (5) or IAMP (IJ))

c. Specifies the number of elements in the subset. For example, the statement:

```
IAMP (5) * 4
```

establishes a window 4 elements long (4 word locations for integer variable elements and 12 word locations for real variable elements) beginning with the fifth element of array IAMP (that is, elements 5, 6, 7, and 8) of array IAMP. Note that the number of elements must be an integer constant.

3.3.4.5 Array Spanning Windows – When arrays are defined in the same .STORAGE statement, they are established in core as a continuous series of word locations in the order listed. For example:

```
.STORAGE IAMP (1,6), ISTAT (1,10), ILOG (1,5)
```

establishes the list:

IAMP (1)	1
⋮	⋮
IAMP (6)	6
ISTAT (1)	7
⋮	⋮
ISTAT (10)	16
ILOG (1)	17
⋮	⋮
ILOG (5)	21

In situations such as described in the preceding example, the user may specify a window that can include portions of two or more sequentially defined arrays by using the same technique described in Paragraph 3.3.4.4.

For example, the following statement:

```
IAMP (5) * 14
```

specifies a window that spans portions of arrays IAMP, ISTAT, and ILOG of the preceding example. The following array elements are included in the window:

IAMP (5)	1
IAMP (6)	2
ISTAT (1)	3
⋮	⋮
ISTAT (10)	12
ILOG (1)	13
ILOG (2)	14

The array window permits the user to specify a consecutive series of small arrays that simplifies gaining access to data; moreover, it considerably reduces the code developed by the Compiler by having only one window bringing in or sending out data to and from all the defined arrays.

3.3.4.6 Presetting Stored Values – In INDAC, the user can preset the value of stored data whenever the array to contain the data is defined. This feature of the .STORAGE statement is useful for establishing constants or initial data values. The required statement formats and procedures for presetting value in .STORAGE statements are as follows:

a. To preset a single defined item the format is:

```
.STORAGE NAME/value/
```

NOTE

a. The value must be preceded by a slash (/).

b. A value preset is terminated by a slash (/).

c. To specify an octal value, precede the number by an apostrophe ('), e.g.

d. Commas are used to delimit each item specified.

a. (cont)

Examples:

.STORAGE ISUM/50/
.STORAGE ISUM/'0050/

b. To write a statement which presets a series of values into an array, the required format is:

.STORAGE NAME (ARRAY SIZE)/n_1,n_2,...,n_n/

For example:

.STORAGE ISUM (1,4)/50,60,70,80/

NOTE

a. The number of values specified in the preset list may not exceed the defined size of an array.

b. Any elements of an array which are not preset are set to 0.

c. If the same value is to be preset into more than one array element, the statement has the following format:

.STORAGE NAME (array size)/No. of Elements (Value)/

Example:

.STORAGE ISUM (1,20)/5 (50)/

This statement establishes the 20-element array ISUM and presets the first five elements to the decimal value 50; the remaining 15 elements are set to 0.

3.4 DATA COLLECTION AND CONTROL

In automating any industrial process, process variables must be collected and control must be effected. Since there are usually many sensors and controls involved in a process, multiplexing devices are used to interface the computer with the process. It is through individual channels of these devices that process data is collected and control is effected. DEC provides a full line of such multiplex devices to service the sensors and control equipment of industrial plants.

DEC multiplex devices (IDACS) include:

Digital-to-Analog Converters
Analog-to-Digital Converters
Digital Output
Digital Input
Integrating Digital Voltmeter

The following names have been assigned to the INDAC System multiplex devices:

ADC (Analog-to-Digital Converter)
DAC (Digital-to-Analog Converter)
UDC (Digital I/O)
AF04 (IDVM)

To illustrate the data collection and control capability of the INDAC language, a simple example of a process control application is detailed in the following paragraphs.

NOTE

Refer to Skeleton No. 3 to place the program statements of the example into perspective.

The example demonstrates maintaining the bath temperature of a solution, including the following operations:

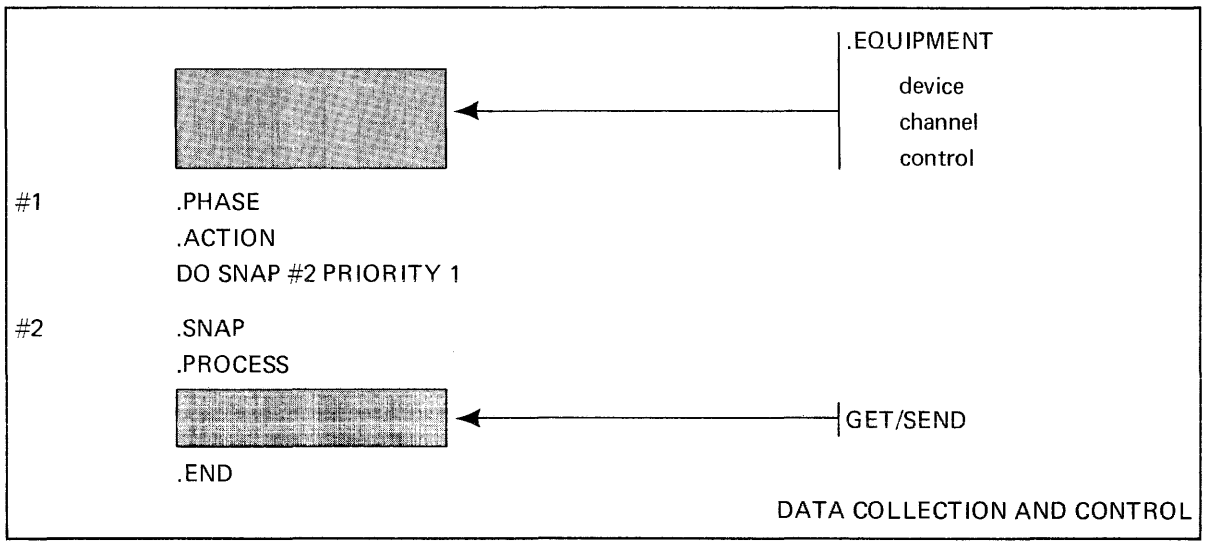
a. Open hot and cold water valves

b. Open drain and turn BATH OK lamp off.

c. Measure water temperature. If water temperature is 68 ±0.5 degress, turn BATH OK lamp on. If water temperature is not 68 ±0.5 degrees turn BATH OK lamp off and adjust hot water valve.

d. Repeat *c.*

Skeleton No. 3



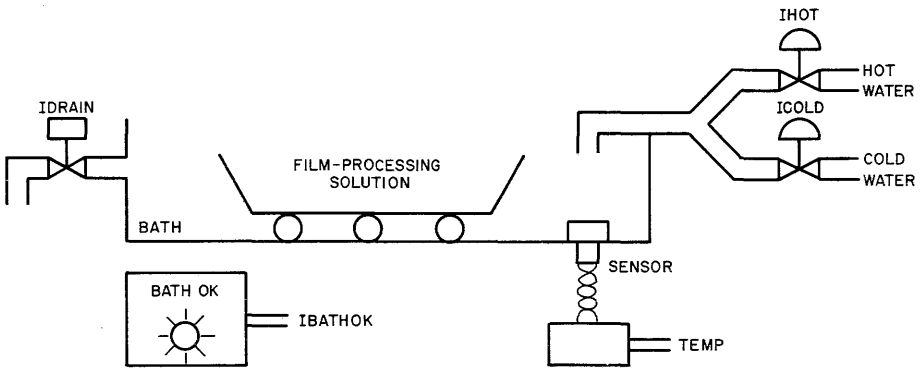
An illustration of the process and sensor device and channel assignments is shown in Figure 3-1. Since the process interface devices are multiplexed (that is, more than one channel can be accessed in each device) the channels of interest must be specified in the program to send data to and to get data from the process elements. The .EQUIPMENT statement permits the user to declare the channels of interest. The .EQUIPMENT statement must be the first non-comment line in the program. A series of continuation lines (indicated by an up-arrow, ↑) are used to declare the channels of interest for each multiplex device. Control options and any associated signal-conditioning subroutines can also be declared in the .EQUIPMENT statement. Details relating to control option and subroutine declaration are discussed later in this chapter. The following statement declares two DAC channels, IHOT and ICOLD:

```
        .EQUIPMENT
↑       *DAC
↑       CHAN (1) IHOT
↑       CHAN (2) ICOLD
```

Notice that devices are identified by a preceding asterisk and channel declarations follow the name of the associated device. Also notice that both channel declarations contain the key word CHAN, a number in parentheses, and a name. The name declares both channels as integer variables. The IHOT variable is assigned channel one and the ICOLD variable is assigned channel two. Notice that when a DAC channel of interest is declared it is just like declaring the process control element itself. Once the above .EQUIPMENT declaration is made, the HOT and COLD valves can be positioned by writing:

```
LET IHOT = '2000
LET ICOLD = '2000
SEND (DAC) IHOT, ICOLD
```

Note that the .EQUIPMENT statement does not declare the data to be sent to the device; but, only the channel associated with the variable name. The name can then be used for assignment of data using a LET statement. The octal number 2000 has been chosen to represent a value setting for a half-open valve.



PROCESS ELEMENT	MUX DEVICE	CHANNEL
HOT valve	DAC	1
COLD valve	DAC	2
Temperature sensor	ADC	37
Drain valve	UDC	15*
BATH OK indicator	UDC	16*

***NOTE**

Drain valve and BATH OK indicator are connected to the screw terminals for UDC channels 15 and 16 as follows:

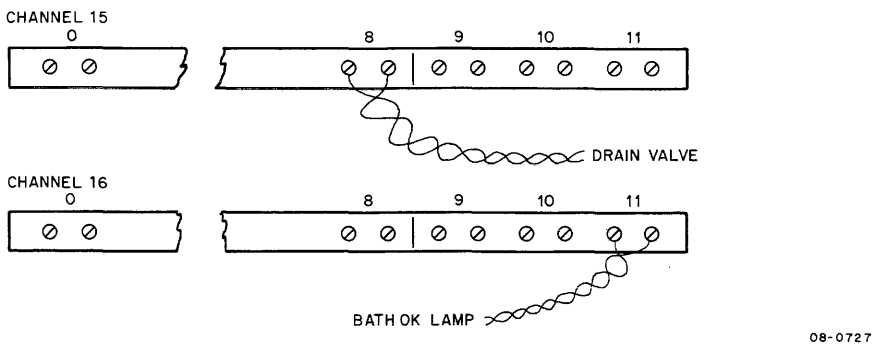


Figure 3-1 Sample Process Control Application

The next step in the sample program is to open the drain and turn off the BATH OK lamp. As defined earlier, the drain valve and the BATH OK lamp are connected to the UDC. There are two methods of declaring UDC channels: Explicit and Point methods. Only the explicit method is discussed at this time. The following .EQUIPMENT statement declares the channels for the drain valve and the BATH OK lamp.

```
        .EQUIPMENT
↑       * UDCE
↑       CHAN (15) IDRAIN
↑       CHAN (16) IBATHOK
```

The device name UDC is modified with an E to denote explicit channel declaration. Both channels are declared as integer variables. IDRAIN is assigned channel 15 and IBATHOK is assigned channel 16.

NOTE

Octal constants are generally used to set or reset bits of interest in a particular channel. The bits of interest and the associated constants for setting these bits in this example are as follows:

Process Element	CHAN	Bit	Constant
Drain Valve	15	8	'0010
BATH OK lamp	16	11	'0001

When bit 8 of channel 15 is set the drain valve is open; when bit 11 of channel 16 is set the BATH OK lamp lights.

The following program statements cause the drain to open and the lamp to go out.

```
LET IDRAIN = '0010
LET IBATHOK = 0
SEND (UDCE) IDRAIN, IBATHOK
```

The LET statements assign octal 10 (bit 8 is set) to IDRAIN and 0 (bit 11 and all other bits are reset) to IBATHOK. The SEND statement sends these two bit patterns to the UDC channels declared in the .EQUIPMENT statement.

The next step in controlling the sample process is to measure the bath temperature, adjust the HOT water to obtain a bath temperature of 68 ±0.5 degrees, and turn on the BATH OK lamp. As defined earlier, the temperature sensor is connected to channel 37 of the ADC. In addition to declaring the channel to which the sensor is connected, ADC's with variable gain capability also require a control option declaration (see Paragraph 3.4.2). The following .EQUIPMENT statement declares the temperature sensor and a control option

```
        .EQUIPMENT
↑       *ADC
↑       CHAN (37) TEMP
↑#901   X200 DO TCONV
```

The temperature sensor is declared on channel 37 as a real variable named TEMP. By declaring a real variable for the temperature channel, temperature may be maintained in 0.5 degree increments. An integer variable would restrict the temperatures to full degree increments. The control option declared is X200 with a tag of #901. Control options must always be tagged so that the option can be referenced in the GET statement. A control option declaration is required when the ADC of the system has scaling capabilities. Some DEC ADCs do not have scaling capabilities (fixed gain) and therefore do not require a control option declaration. To get and test the temperature and to control the water temperature and indicator lamp, the following statements are used.

```
11          GET (ADC, #901) TEMP
            IF TEMP GR 68.5 THEN GOTO 12
            IF TEMP LS 67.5 THEN GOTO 13
            LET IBATHOK = '0001
            GOTO 10
              :
12          LET IHOT = IHOT - 1
            GOTO 9
13          LET IHOT = IHOT +1
 9          LET IBATHOK = '0000
            SEND (DAC) IHOT
10          SEND (UDCE) IBATH OK
            GOTO 11
```

Notice the structure of the GET statement for the temperature (TEMP); the device name and the control option tag are separated by a comma and enclosed in parentheses. Since TEMP was declared to be associated with channel 37 in the .EQUIPMENT statement, the GET statement transfers the quantity in channel 37 to the storage location TEMP while the ADC is operating with a gain of 200 (X200). The two IF statements test the quantity that was stored in TEMP by the GET statement. If the sensor supplies a value that must be converted or linearized before a meaningful measurement can be made, the required conversion routine can be declared with the control option (see Paragraph 3.4.2). In the example, a conversion routine is declared (DO TCONV). The routine is performed during the GET statement and must be coded to perform the following: convert the 12-bit ADC value to a real variable, linearize the value and store it in the location named TEMP. If the temperature is not within 0.5 of 68 degrees, the statement labeled 12 or the statement labeled 13, depending on the temperature, will be executed. These statements increase or decrease the variable named IHOT. In either case, the statement labeled 9 is executed next to set IBATHOK to 0, thereby turning the light out. The two SEND statements that follow transmit the new position value to the HOT water valve and turn off the power to the IBATHOK lamp. Program control is then transferred to the statement labeled 11 to repeat the sequence. When the bath temperature reaches 68 ±0.5 degrees, then the statement following the IF statements is executed to set bit 11 of IBATHOK to 1. The statement labeled 10 is then executed to turn on the BATH OK lamp and the sequence is repeated.

All the statements related to data collection and control that were developed in the previous paragraphs have been placed in perspective in the following update to Skeleton No. 3. Slight changes have been made to the program to illustrate how algorithm code can be minimized. Notice that the channel and option declarations for each device appear on continuation lines following the .EQUIPMENT statement. All executable statements are located in the algorithm section between the key words .PROCESS and .END.

In the previous example, 2 UDC channels (words) were used; one for the drain valve and another for the BATH OK lamp. Since each UDC channel contains 12 bits, the valve and lamp can both be connected to one channel, in addition to other controls and indicators.

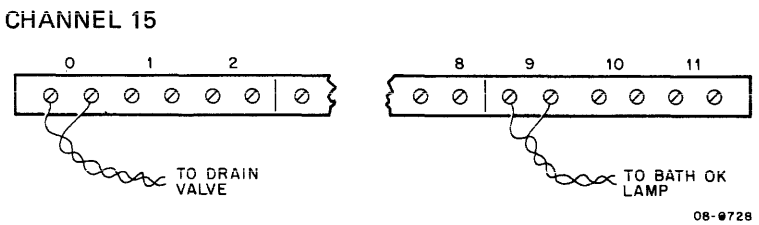
NOTE

A given UDC channel is limited to input or output applications, not both.

Skeleton No. 3 — Equipment Details

```
                .EQUIPMENT
↑               *DAC
↑               CHAN (1) IHOT
↑               CHAN (2) ICOLD
↑               *UDCE
↑               CHAN (15) IDRAIN
↑               CHAN (16) IBATHOK
↑               *ADC
↑               CHAN (37) TEMP
↑#901           X200 DO TCONV
#1              .PHASE
                .ACTION
                DO SNAP #2 PRIORITY 1
#2              .SNAP
                .PROCESS
                LET IHOT = '2000
                LET ICOLD = '2000
                LET IDRAIN = '0010
 9              LET IBATHOK = 0
                SEND (DAC) IHOT, ICOLD
10              SEND (UDCE) IDRAIN, IBATHOK
                GET (ADC,#901) TEMP
                IF TEMP GR 68.5 THEN GOTO 12
                IF TEMP LS 67.5 THEN GOTO 13
                LET IBATHOK = '0001
                GOTO 10
12              LET IHOT = IHOT - 1
                GOTO 9
13              LET IHOT = IHOT + 1
                GOTO 9
                .END
```

When more than one control, indicator, or other field element is connected to one UDC channel, then Boolean operations must be used to isolate the bit of interest to test, set, or reset the bit. The following example illustrates how Boolean operations can be used to operate on specific bits of a UDC word. Assume that the drain valve and the BATH OK lamp are connected to one UDC channel as follows:



With the valve and lamp connected to one channel as shown above the .EQUIPMENT statement should be changed to reflect that only one UDC channel is used. The following .EQUIPMENT statement defines UDC channel 15.

```
                .EQUIPMENT
↑               *UDCE
↑               CHAN (15) IOUT
```

Notice that a different integer variable name is used in declaring the channel. The algorithm for the sample process can now be rewritten as follows:

```
                LET IHOT = '2000
                LET ICOLD = '2000
                LET IOUT = '4000
9               LET IOUT = IOUT AND NOT '4
                SEND (DAC) IHOT, ICOLD
10              SEND (UDCE) IOUT
                GET (ADC, #901) TEMP
                IF TEMP GR 68.5 THEN GOTO 12
                IF TEMP LS 67.5 THEN GOTO 13
                LET IOUT = IOUT OR '4
                GOTO 10
12              LET IHOT = IHOT -1
                GOTO 9
13              LET IHOT = IHOT +1
                GOTO 9
```

Notice that octal constant 4000 is assigned to IOUT. The LET statement labeled 9, using Boolean operations, sets up the bit pattern required for opening the drain and turning off the lamp. The LET statement following the IF statements, using another Boolean operation, sets up the bit pattern required to turn on the lamp. Otherwise, the algorithm is the same as the one presented earlier.

The reader was introduced to the following three process I/O devices in the preceding example:

DAC
ADC
UDC

All three devices require channel declarations in that they are all multiplex devices. Only the ADCs require control option specifications. The following paragraphs provide additional details for each of the above devices and introduce the following two devices:

AF04
FILE

3.4.1 DAC (Digital-to-Analog Conversion Devices)

3.4.1.1 Hardware Device – DEC offers two digital-to-analog conversion devices: the AA01 and the AA50. Both devices are assigned the name DAC. GENDAC declares this name to the INDAC software system when the I/O handler of either device is selected from the GENDAC library. Therefore, only one of these devices can be declared in the INDAC System.

3.4.1.2 Equipment Declaration Statement – The DAC and associated information is declared in the INDAC program as follows:

```
↑       *DAC
↑       CHAN (m,n) v
```

Where m, n are, respectively, the first and last channels associated with the integer array. If v is a single integer variable, then only a single channel is declared. There may be as many repetitions of the channel multiplexer line as there are channels to declare. The DAC has no control options to be declared.

3.4.1.3 Language Statement – The INDAC language statement is:

```
SEND (DAC) ⟨data list⟩
```

Where ⟨data list⟩ is the list of the items to be output. Note that every variable specified in the list must also appear in the equipment definition to establish the correlation between the variable and the associated channel address.

3.4.2 ADC (Analog-to-Digital Conversion Devices)

3.4.2.1 Hardware Device – DEC offers five analog-to-digital converter devices: the AF01, AF02, AF03, AFC8 and AD01. All of these devices are assigned the name ADC. GENDAC declares this name to the INDAC software system when the I/O handler of any one of the above devices is selected from the GENDAC library. Therefore, only one of these devices can be declared in the INDAC System.

3.4.2.2 Equipment Declaration Statement – The ADC and associated information is declared in the INDAC program as follows:

```
↑       *ADC
↑       CHAN (m,n)v
↑#t     DO s
↑#t     Xn DO s
↑#t     DO AREF
```

Where m, n are, respectively, the first and last channels associated with the integer or real array v. If v is a single variable, then only a single channel is declared. There may be as many repetitions of the channel multiplexer line as there are channels to declare. For the AF03, AFC8 and AD01 the control line contains a tag reference #t and a gain factor (Xn) where n may be:

Xn	AF01 AF02	AF03	AFC8	AD01A
1		X	X	X
2			X	X
4				X
5				
8	(Fixed Gain)			X
10		X	X	
20		X	X	
50			X	
100		X	X	
200		X	X	
1000		X	X	

The control line may also contain a subroutine call DO s, where s is a linearizing or signal conditioning subroutine. There may be as many control lines of this type as there are variable gains or signal conditioning subroutines to declare. In addition to the optional control lines, one line must contain the statement DO AREF. This subroutine establishes the location of the reference junction or offset variable required by some of the conversion routines. The AF01 and AF02 are fixed gain devices and do not require a gain specification on the control line, but may require specification of the same subroutines described above.

Language Statement — The INDAC language statements are:

```
GET (ADC) ⟨data list⟩
GET (ADC, #t) ⟨data list⟩
```

Where #t is a tag reference to the control line and ⟨data list⟩ is the list of the items (v) to be converted. Note that every variable specified in the list must also appear in the equipment definition to establish the correlation between the variable and the associated channel.

If the signal conditioning subroutines created by the user are called in a control line (DO s) and used in a GET statement, then the first GET statement for the ADC executed in the job must define the variable used as the reference junction or offset variable by referencing the "AREF" subroutine call on the control line in the GET statement. Then the GET statement for the data channel of interest can be executed with the reference to the conversion routine call. For example:

```
                .EQUIPMENT
                    ⋮
↑               *ADC
                    ⋮
↑               CHAN (0) IREF
↑               CHAN (1,6) ILEV
↑#10            X200 DO AREF
↑#11            X100 DO CONV
                    ⋮
                GET (ADC, #10) IREF
                GET (ADC, #11) ILEV
                    ⋮
```

NOTE

The I/O handler for the ADC returns a 12-bit integer count for the conversion. Normally the variable in the data list is an integer variable. If a real variable input is required, it is the responsibility of the subroutine called through the control line to convert the integer variable to a real variable. If a channel declaration is made using a real variable name, then that channel may only be called by referencing a control line with a subroutine declaration.

3.4.3 AF04 (Integrating Digital Voltmeter)

3.4.3.1 Hardware Device – DEC offers one Integrating Digital Volt Meter (IDVM) designated and named AF04. GENDAC declares this name to the INDAC software system when the I/O handler of the IDVM is selected from the GENDAC library.

3.4.3.2 Equipment Declaration Statement – The AF04 and associated information is declared in the INDAC program as follows:

↑	*AF04
↑	CHAN (m,n) v
↑#t	a_1, a_2, a_3 DO s
↑#t	DO AREF

When m, n are, respectively, the first and last channels associated with the array v, the array must be a real array. If v is a single variable, then only a single channel is declared. There may be as many repetitions of the channel multiplexer line as there are channels to declare. The control line contains the item specifying the selectable function, range, and resolution required of the AF04. One each from the following three lists may be selected (a_1, a_2, a_3)

a_1 FUNCTION	a_2 RANGE	a_3 RESOLUTION
DC	10MV	.1
FREQ	100MV	.01
PERIOD	1000MV	.001
OHMS	10V	
AC	100V	
	AUTO	

Note that the selectable voltage ranges provide for increased conversion accuracy. The array or variable v is always supplied in volts (with the proper exponent) regardless of the selected range.

The control line may also contain a subroutine call DO s, where s is a linearizing or signal conditioning subroutine. There may be as many control lines of this type as there are variable gains or signal conditioning subroutines to declare. In addition to the optional control lines, there must exist one line containing the statement DO AREF. This subroutine establishes the location of the reference junction or offset variable required by some of the conversion routines.

3.4.3.3 Language Statement – The INDAC language statement is:

GET (AF04, #t) ⟨data list⟩

where #t is a tag reference to the control line and ⟨data list⟩ is the list (real variables) of the items to be converted. Note that every variable specified in the list must also appear in the equipment definition to establish the correlation between the variable and the associated channel.

If the signal conditioning subroutines created by the user are called in a control line (DO s) and used in a GET statement, then the first GET statement for the AF04 executed in the job must define the variable used as the reference junction or offset variable by referencing the "AREF" subroutine control line in the GET statement. For example:

```
        .EQUIPMENT
            ⋮
↑       *AF04
            ⋮
↑       CHAN (0) REF
↑       CHAN (1,6) LEV
↑       #10 DC, 100 MV, .01 DO AREF
↑       #11 DC, 1000M, .01 DO CONV
            ⋮
        GET (AF04, #10) REF
        GET (AF04, #11) LEV
            ⋮
```

NOTE

The I/O handler for the AF04 returns a real variable for the conversion. Normally the data list contains only real variables. If an integer output is required, it is the responsibility of the subroutine called through the control line to convert the real variable into an integer variable name, then that channel may only be called by referencing a control line with a subroutine declaration.

3.4.4 UDC (Universal Digital Controller)

3.4.4.1 Hardware Device – DEC offers one highly flexible digital I/O device designated UDC8. Two I/O handlers have been developed for this device: an explicit handler called UDCE and a point table handler called UDCP. GENDAC declares these names to the INDAC software system if the handlers are chosen from the GENDAC library. Either or both handlers may be selected. The following descriptions of the equipment declaration and language statements deal with the UDCE handler only. The UDCP handler is discussed later in this chapter.

3.4.4.2 Equipment Definition Statement – The UDCE and associated information is declared in the INDAC program as follows:

```
↑       *UDCE
↑       CHAN (m,n) v
```

where m, n are, respectively, the first and last channels associated with the variable v. If v is a single variable, then only a single channel is declared. There may be as many such multiplexer lines as are required to define the input and output channels.

3.4.4.3 Language Statement – The INDAC language statements are detailed below:

a. Each set of 12 input lines, connected to a digital input channel, is available to the system on demand through the statement

GET (UDCE) ⟨data list⟩

where ⟨data list⟩ is the list of items to be input. Note that every variable specified in the list must also appear in the equipment declaration to establish the correlation between the variable and the associated channel.

b. Each set of 12 output lines, connected to an output channel, may be exercised by the system on demand, through the statement

SEND (UDCE) ⟨data list⟩

where ⟨data list⟩ is the list of the items to be output. Note that every variable specified in the list must also appear in the equipment definition to establish the correlation between the variable and the associated channel.

3.4.5 FILE (Pseudo-Device)

3.4.5.1 Hardware Device – The FILE is maintained on the system storage device, Mass Storage Disk DF32.

3.4.5.2 Equipment Definition Statement – The FILE and associated information is declared in the INDAC program as follows:

```
↑          *FILE
↑#t        k
```

where #t is the tag reference of the control line and k may be one of the following three identifiers: 1, 2, or 3.

3.4.5.3 Language Statement – The INDAC language statements are:

GET (FILE, #t) Ia, ⟨data list⟩
SEND (FILE, #t) Ia, ⟨data list⟩

where #t is a tag reference to the control line specifying the FILE identification, Ia is an integer variable specifying the record (page) within the FILE, and ⟨data list⟩ is the list of the items contained within the record.

NOTE

The total word count of the data-list items may not exceed one page (128 decimal words). The total number of records available is dependent upon the size of the user job and the number of disks in the user's system.

The total file is organized as one sequential set of records from the base of FILE 1, for as many records as are used by the program. FILE 2 and 3 are simply convenient locations within the sequential set of records for reference purposes. There is no protection within the FILE, since the overlap feature has a useful function. The starting location of each of the FILEs may be altered via the Executive Command Decoder at run-time.

3.4.6 Equipment Statement Summary

The .EQUIPMENT statement must be the first non-comment line of any program that uses multiplex devices or devices that require specification of control options. This statement uses a series of continuation lines to specify each device, its channel declarations, control option, and any associated-conditioning subroutines.

The .EQUIPMENT statement has the following general form:

```
        .EQUIPMENT
↑       (device code line)
↑       (multiplexer line)
↑       (control line)
```

Each specified device (code line) must be followed immediately by applicable multiplexer lines, then applicable control lines.

The .EQUIPMENT statement is used to declare symbolic names of devices, device control options, and subroutines. Standard device declarations (TTY, KEYS, STATUS, PRIORITY and FILE) are a permanent part of the compiler tables. These tables can be modified by GENDAC (Generate INDAC) to add I/O handlers for additional devices.

The description and format of the line types which make up the .EQUIPMENT statement are as follows:

a. Device Code Line

This continuation line contains a device symbolic name and has the form:

```
↑*u
```

where u is the assigned symbolic name for the device to be specified. The asterisk (*) indicates that all information presented until the detection of the next asterisk or (.) statement refers to the specified device.

The symbolic name used must correspond exactly to that supplied to the system during initialization (refer to Appendix I).

b. Multiplexer Line

When the specified device has multiplexed input or output lines, the code line must be followed by a multiplexer line which has the form:

```
↑CHAN mux v
```

where

1. CHAN identifies the type of information presented.
2. mux represents the multiplexed channel or channels of the device to be associated with v.
3. v represents a name of a variable array to which the specified channel or channels are assigned and defines the data buffers for the I/O operations.

When more than one device channel is to be specified, the following form is used:

```
↑CHAN (m, n) v
```

where m is the first channel and n is the last channel in the desired range. For example, the list (0, 19) specifies a range of 20 channels identified as 0 through 19.

The array named is automatically dimensioned to the number of elements specified by m and n. The array name is the data buffer where the digitized values of input signals connected to multiplexer channels will be stored.

When the specified input or output channel or channels of a multiplexed device are to be assigned to more than one variable or array, one multiplexer line is required for each different assignment.

(continued on next page)

c. Control Line

One or more control lines may be used to list the symbolic names assigned to the device-control options available for the specified device. If a signal-conditioning subroutine is required for the device input or output operations, it is specified in the control line.

↑ #t $SCN_1, SCN_2, ..., SCN_n$

or

↑ #t $SCN_1, SCN_2, ..., SCN_n$ DO s

where

1. t is a mandatory reference tag.
2. SCN represents symbolic control names assigned to control options available for the specified device.
3. s represents the assigned symbolic name of a conversion subroutine.

Control statements for each device must be listed immediately after the last multiplexer line for that device.

Control lines must be tagged since they are referenced by input/output statements (GET, SEND) to specify the manner in which the manipulated data is to be handled.

Control lines are used only to specify a particular control option or set of options required to carry out the GET or SEND statement.

A separate, individually tagged control line is required for each specified set of options used.

3.5 STRUCTURE OF AN INDAC JOB

Usually in an on-line, real-time acquisition and control application, there are many tasks that must time-share core memory for economy of core storage. To attain such a goal in INDAC 8/2, all programs (or program segments) are disk resident; they are brought into core by the operating system in response to process interrupts, elapsed timers, or task requests, see Figure 3-2. To allow this kind of time-sharing, an INDAC program is not made of a string of statements such as a BASIC or FORTRAN program which is executed as a single program unit. Rather, it is a collection of different tasks or program units separated by appropriate segmentation statements. Process interrupt handling is covered later in this chapter.

3.5.1 Program Segmentation

a. Job Specification – This segment of the program includes the .EQUIPMENT, .INTERRUPT, .STORAGE, .HEADER, and .FORMAT statements. These statements define the Global parameters of the job. This information is transferred to core during the initial call for the job and remains in core throughout the life of the job. This area is never swapped.

b. The PHASE – This segment contains the timer parameters for calling SNAPS or other PHASEs. The PHASE segment exists in core only when the PHASE is operational. If a PHASE is released and then recalled, that PHASE will be completely initialized on recall.

c. The SNAP – This segment (task) contains the actual algorithms used to perform the job. SNAPs may also call other SNAPs or PHASEs as required to service the job.

d. The SUBROUTINE – This unit contains code for common operations to avoid duplication of effort. When included in a job, the SUBROUTINE becomes an integral part of the job.

The INDAC Executive and the INDAC language permit a variety of sequencing and scheduling functions. Some of the functions are programmed to occur at a specific interval of time or at the next available opportunity. Other functions are dependent upon the operation of a peripheral device. Due to the random operation of such devices, the Executive is prepared to resolve conflicts between the different requests to operate parts of the system.

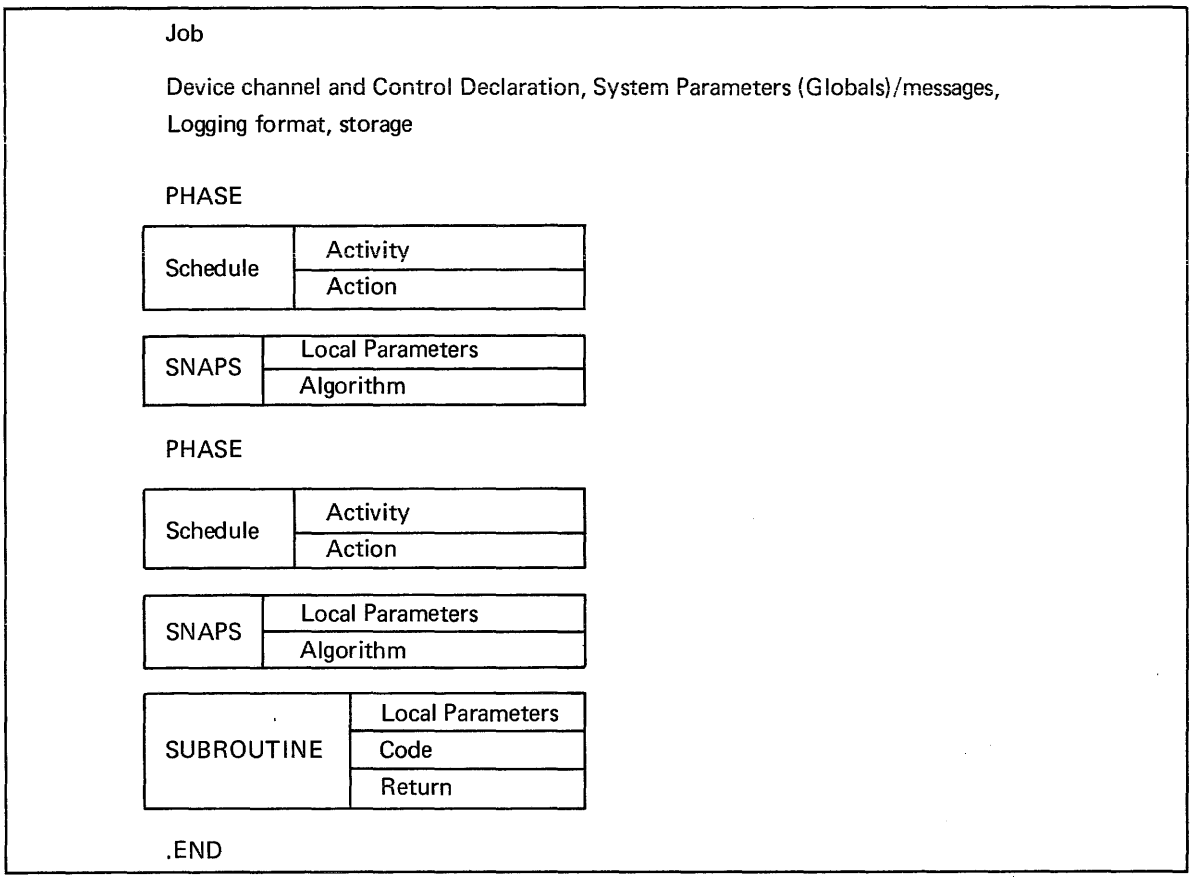


Figure 3-2 Structure of an INDAC Job

3.5.1.1 Job Specification Segment – This is the first segment in a program. It presents any parameters required for systems-level program activities (Global). A detailed breakdown of the organization and contents of the Job Specification segment is given in Table 3-5. The segment components introduced in the table are described in detail in the following paragraphs.

Table 3-5
Job Specification Segment, Organization and Contents

Hierarchy of Possible Job Statements	Statement Type and Use
.EQUIPMENT	Specifications of logical device units, channels, control modes, and linearization subroutines associated with the particular device
.INTERRUPT	Specification of field interrupt device
.STORAGE	Specification of system-level data storage areas
.HEADER	Specification of messages, titles, or other types of information which is to be output (printed or displayed) on demand from other segments within the program
.FORMAT	Specifications of exact format(s) for the output of data common to the system

3.5.1.2 .PHASE Segment – The phase segments consist of:

a. The key word .PHASE preceeded by a tag.

b. A group of statements that specify the scheduling of the data-handling operations to be performed in the PHASE.

c. One or more .SNAP program segments that contain the actual operational instructions for carrying out the task assigned the PHASE segment.

The PHASE identifier has the following form:

```
#t          .PHASE
```

where #t is a mandatory reference tag.

The basic organization and content of a .PHASE segment are shown in Table 3-6.

Table 3-6
Organization of PHASE Segment

Hierarchy of Possible PHASE Statements		Statement Type and Use
#10	.PHASE	PHASE segment identifier
#14	DO SNAP #50 EVERY 5 SEC PRIORITY 2	Example of an activity statement
	.ACTION	Compiler directive statement which introduces a list of PHASE action and timer statements
	TIMER (START, #14)	TIMER statement initializing previously defined activity statement
	DO SNAP #55 PRIORITY 1	Example of a SNAP call statement Identifies first SNAP to operate in PHASE

3.5.1.2.1 Activity Statements – The activity statements define the scheduling parameters of the actions listed under the .ACTION statement. The key words EVERY, DELAY and AT, in conjunction with the DO statement, are used in the activity statements. For example:

```
#10         .PHASE
#14         DO SNAP #500 EVERY 5 SEC PRIORITY 2
#15         DO SNAP #501 DELAY 10 MIN PRIORITY 3
#16         DO SNAP #502 AT 13:15 PRIORITY 1
```

3.5.1.2.2 .ACTION Statement – .ACTION is a statement which is used in PHASE segments to introduce a list of specified actions to take place in the PHASE. The .ACTION statement occurs only once in each PHASE.

The action list consists of executable DO and TIMER statements. The list is started by the .ACTION statement and is terminated by the occurrence of the first .SNAP segment. For example:

```
            .ACTION
            DO SNAP #500 PRIORITY 2
            TIMER (START, #14)
#500        .SNAP
```

3.5.1.3 .SNAP Segments — The snap segments consist of:

a. The key word .SNAP preceeded by a tag.

b. SNAP specification which may include .STORAGE, .HEADER, and .FORMAT statements. These statements define the local parameters of the task.

c. A list of executable program statements which are introduced by a SNAP level .PROCESS statement.

The SNAP identifier has the following form:

#t .SNAP

where t is a mandatory reference tag.

The SNAP specification statement is a list of statements written immediately after the SNAP identifier that specifies SNAP-level (local) data storage and output header and format requirements. The organization and contents of a .SNAP are shown in Table 3-7.

A .PROCESS compiler directive is employed in each SNAP to introduce the executable statements that form the operational portion of this type of program segment.

Table 3-7
Organization of a SNAP Segment

Hierarchy of Possible SNAP Statements		Statement Type and Use
#10	.SNAP	SNAP segment identifier
	.STORAGE	Statement for specification of SNAP-level storage
#12	.HEADER	Statement for specification of SNAP-level header outputs
#13	.FORMAT	Statement for specification of SNAP-level output data formats
	.PROCESS	Compiler directive, identifying executable portion of SNAP
	.	
	.	
	.	List of executable statements
	EXIT or	Program control statements
	EXIT THEN DO SNAP #_ PRIORITY _ or	
	EXIT THEN DO PHASE #_ PRIORITY _	

NOTE
Comment lines may appear anywhere in the INDAC program to simplify readability. A comment line is written as follows:

·␣———

Notice that a space delimits the period and the first word of the comment.

3.5.1.4 Subroutine – Often certain operations are used repeatedly throughout the procedures specified in the program segments. In order to avoid duplication of effort, such common operations may be specified as subroutines. Subroutine call statements are inserted into the program list at each point where the subroutine operation is required. During the execution of the program, each call causes the processor to execute the specified subroutine before proceeding to the next listed statement.

There are two types of INDAC subroutines.

a. Internal

b. External

The manner in which each is written and used is described in the following paragraphs.

3.5.1.4.1 Internal Subroutine – Subroutines of this type are written as part of a standard .SNAP segment. The repeatable group of statements that may be referenced (called) as a subroutine unit is defined by:

a. A labeled executable statement used as the first statement in the subroutine group.

b. The use of a program control RETURN statement.

The call statement for an internal subroutine has the form:

```
DO k
```

where k represents the label of the first subroutine statement.

For example, the internal subroutine:

```
401     LET N = N + 1
        RETURN
```

would be called by the statement:

```
DO 401
```

Internal subroutines are defined only for the .PROCESS segment in which they appear. More than one RETURN statement may be used within a subroutine to facilitate alternative operations.

3.5.1.4.2 External Subroutine – Subroutines of this type may be included as part of a complete job. When included in a job, external subroutines may be called only from the segments of that job. When entered into INDAC separately, an external subroutine may be called from within any INDAC system program. External subroutine identifiers have the form:

.SUBROUTINE s (v_1, v_2, ..., v_n)

where s is a unique symbolic name assigned to the subroutine, and the parenthetical list contains dummy variables for input and output data.

The identifier dummy variables represent those used throughout the subroutine only; their equivalent variables in the program segments that call the subroutine must be specified in the call statement. The list v_1,...v_n may contain variable or array names. If a variable is an array name, it represents the base of the array.

a. Calling the External Subroutine – Call statements for external subroutines have the following form:

DO s (v_1, v_2, ..., v_n)

(continued on next page)

a. *(cont)*

where s represents the name of the called subroutine, and the call list contains a one-for-one equivalent variable for each dummy variable in the subroutine identifier. For example, given the subroutine identifier:

```
.SUBROUTINE TCON (X, Y, Z)
```

one example of an acceptable call would be:

```
DO TCON (A, B, C)
```

An example of a subroutine call is given in a listing shown in Appendix B. Note the way in which arrays IK and A are made available to the subroutine.

b. Returning Control to the Caller – External subroutines return control when the following statement is encountered:

```
RETURN s
```

where the RETURN statement specifies that the processor return to and execute the statement immediately following the subroutine call. The external subroutine is organized in a manner similar to that of a SNAP segment, that is,

```
.SUBROUTINE s (v1, v2, ..., vn)
.STORAGE
.PROCESS
(List of Executable Statements)
RETURN s
```

As indicated above, the only nonexecutable statement which may be used in the body of the external subroutine are .STORAGE and .PROCESS. Executable I/O statements GET and SEND, plus TIMER and activity statements are not permitted.

NOTE

There may be more than one RETURN s statement in the external subroutine.

3.5.1.4.3 Implicit Subroutine – Implicit subroutine calls are defined in the device control of the .EQUIPMENT definition statement, such as:

```
          .EQUIPMENT
↑         *ADC
↑         CHAN (0, 10) VALUE
↑#10      X200 DO  IRONC
```

Whenever a GET (ADC, #10) VALUE statement accesses the device ADC, the converted value from ADC is given as input to the implicit subroutine IRONC; the output value of the subroutine is then deposited in VALUE. Three parameters are implicit in this subroutine call. The first is the raw value read from the device; the second is the output value of the routine, and the third is an argument to supply a reference or offset value for conversions or linearizations.

The following is an example of a temperature scanning program using iron/constantan thermocouples in channels 1 through 10. The RTD of the isothermal reference box is connected to channel 0.

```
                .EQUIPMENT
↑               *ADC
↑               CHAN (0) REF
↑               CHAN (1, 10) TEMPERATURE
↑ #10           X200 DO IRONC
↑ #30           DO AREF
                .STORAGE REF
#1              .PHASE
                .ACTION
#2              DO SNAP #200 EVERY 5 MIN PRIORITY 1
#3              DO SNAP #300 EVERY 40 SEC PRIORITY 1
                TIMER (START, #2)
                TIMER (START, #3)
                DO SNAP #100 PRIORITY 1
#100            .SNAP
                .PROCESS
                GET (ADC, #30) REF
                LET REF = 10.94 * (REF + 57.5)
                EXIT
#200            .SNAP
                .PROCESS
                GET (ADC) REF
                EXIT
#300            .SNAP
                .PROCESS
                GET (ADC, #10) TEMPERATURE
                EXIT
                .SUBROUTINE IRONC (IVAL, CONV, REFDUM)
                .PROCESS
                LET COUNT 1 = IVAL
                LET COUNT 2 = K1 * REFDUM + K2
                LET CONV = COUNT 1 + COUNT 2
                LET CONV = C0 + (C1 * CONV + (C2 * CONV + (C3 * CONV)))
                RETURN IRONC
                .END
```

The implicit subroutine AREF called at initialization establishes the location of the reference variable required. The variable "REF" contains the reference value, in this case, reference junction temperature.

3.5.2 Scheduling Capabilities

The activity and action sections of a .PHASE segment define the initial scheduling parameters of a job. One or more phase segments may be used depending on the size of the job or the scheduling requirements of the job. The scheduling statements that may be included in the phase segment are:

```
#1          .PHASE
#_          DO SNAP #_ EVERY ___ PRIORITY _
#_          DO SNAP #_ DELAY ___ PRIORITY _
#_          DO SNAP #_ AT __ PRIORITY _
#_          DO PHASE #_ DELAY ___ PRIORITY _
#_          DO PHASE #_ AT __ PRIORITY _
            .ACTION
            TIMER (START, #_)
            DO SNAP #_ PRIORITY _
```

Note that a DO PHASE #_ EVERY ___ PRIORITY _ is not permitted. This statement is not permitted because the scheduling parameters of the calling phase are overlayed at call time.

Scheduling requests may also be specified in the .SNAP segments intermingled with executable code. The scheduling requests that may be included in a .SNAP segment are:

```
#10         .SNAP
               ⋮
            DO SNAP #_ PRIORITY _
               ⋮
            TIMER (STOP, #_)
               ⋮
            DO PHASE #_ PRIORITY _
               ⋮
            TIMER (START, #_)
               ⋮
            EXIT THEN DO SNAP #_ PRIORITY _
            EXIT THEN DO PHASE #_ PRIORITY _
            EXIT
```

When a SNAP terminates (exits) with an EXIT THEN DO _ request, the request is stacked on the scheduling queue for execution at the priority specified. This type of request is placed in front of all other requests at that priority level. A SNAP terminated with a simple EXIT statement releases control to the INDAC Executive.

3.5.2.1 Activity Statements – Activity statements operate in conjunction with TIMER statements to establish incremental control timers that schedule the performance of specified SNAP and PHASE program segments. These statements can appear only in the activity section of a phase.

There are three activity statements:

a. EVERY

b. DELAY

c. AT

All activity statements must be listed in the PHASE before a .ACTION statement.

3.5.2.1.1 EVERY Activity Statements – EVERY activity statements are used in the formation of incremental timers to control the repetitive execution of program SNAP segments. EVERY statements must be assigned a tag reference (that is, #xxx) and can be referenced only by TIMER statements located in the ACTION or in the SNAP program segments.

The EVERY statement has the form:

$\#t_1$ DO SNAP $\#t_2$ EVERY c PRIORITY k

where

a. t_1 is a mandatory tag reference

b. t_2 is the tag reference assigned to the SNAP to be executed

c. c is a time unit declaration which has the form

1. i SEC

2. i MIN

3. i HR

where i in an integer variable or integer constant

d. k is the assigned priority level 1 through 11

For example:

#200 DO SNAP #300 EVERY 5 MIN PRIORITY 1

This execution of this statement causes the SNAP tagged #300 to be executed at five-minute intervals.

3.5.2.1.2 DELAY Activity Statements – DELAY activity statements are used in the formation of elapsed time-control statements that specify the time at which designated SNAPs or PHASEs are to be performed.

DELAY activity statements must be assigned a tag reference. This type of statement has the form:

$\#t_1$ DO SNAP $\#t_2$ DELAY c PRIORITY k

or

$\#t_1$ DO PHASE $\#t_2$ DELAY c PRIORITY k

where:

a. t_1 is a mandatory tag reference

b. t_2 is the tag reference assigned to the SNAP or PHASE to be executed

c. c is a time unit declaration which specifies a time interval of delay after which the designated program segment is to be executed. This declaration unit may have the form:

1. i SEC

2. i MIN

3. i HR

where i represents an integer variable or integer constant

d. k is the assigned priority level 1 through 11

For example, the execution of:

#101 DO SNAP #300 DELAY 10 MIN PRIORITY 1

causes the SNAP tagged #300 to be scheduled for operation 10 minutes after the TIMER (START, #101) statement is executed.

DELAY statements are single-shot in operation; they specify an action to be carried out after one specified interim of elapsed time.

3.5.2.1.3 AT Activity Statements – AT activity statements are used to specify the time-of-day program segment execution.

The AT statement must be assigned a tag reference. This statement has the form:

$\#t_1$ DO SNAP $\#t_2$ AT HH:MM PRIORITY k

where HH is the hour (0–23) and MM is minutes (0–59) of the day. For example:

#101 DO SNAP #1 AT 18:30 PRIORITY 1 or

#102 DO SNAP #2 AT 01:09 PRIORITY 1

will, when system clock has been set, execute SNAP #1 at 6:30 p.m. or SNAP #2 at 1:09 a.m. This statement is re-initialized every time it is executed and will take place at the same time every day unless the timer is stopped with a TIMER (STOP, #_) statement.

3.5.2.2 Action Statements – The action statements are executable DO and TIMER statements. These statements may appear in the .ACTION section of a PHASE or the PROCESS section of a SNAP.

DO action statements are used for executing program segments without regard to system incremental timers. DO action statements have the form:

DO SNAP $\#t_1$ PRIORITY k

where t_1 is a SNAP tag in this PHASE segment and k is the priority level 1 through 11.

DO PHASE $\#t_2$ PRIORITY k

where t_2 is a tag of the current or another PHASE.

3.5.2.2.1 Timer Action Statements – TIMER statements control the operation of the EVERY, DELAY, and AT activity statements. This statement has the forms:

a. TIMER (STOP, #t)
Where t is the reference tag of an EVERY, DELAY, or AT statement. Execution of this statement halts the timing actions associated with the designated activity statement.

b. TIMER (START, #t)
Where t is the reference tag of an EVERY, DELAY, or AT statement. Execution of this statement starts the timing action associated with the designated activity statement.

3.5.2.3 EXIT Program Control Statement – This program control statement is used to terminate the program .SNAP segments; its execution halts the operation of the SNAP being processed, and either returns control to the Executive or specifies a new program segment to be executed.

The EXIT statement has two forms:

a. As a stand-alone statement

EXIT

which, when executed, terminates the current SNAP, or

b. As a combined statement

EXIT THEN DO SNAP #t PRIORITY k

or

EXIT THEN DO PHASE #t PRIORITY k

which, when executed, terminates the current SNAP operation, then directs the processor to the next SNAP or PHASE to be executed.

3.5.2.4 Resolving Timer Requests – Timed requests are resolved using the single system incremental counter and the process queue. As each second is recorded by the hardware clock, the system clock is incremented by one. Each phase contains the individual timer parameters established by the user under the activity list of the phase. A timer parameter is established by one of the following five activity statements:

$\#t_1$ DO SNAP $\#t_2$ EVERY c PRIORITY k
$\#t_1$ DO SNAP $\#t_2$ DELAY c PRIORITY k
$\#t_1$ DO PHASE $\#t_2$ DELAY c PRIORITY k
$\#t_1$ DO SNAP $\#t_2$ AT HH:MM PRIORITY k
$\#t_1$ DO PHASE $\#t_2$ AT HH:MM PRIORITY k

These language statements generate timer parameters examined by the timer scan operation. The parameters are initially inactive and remain inactive until the statement: TIMER (START, #t) in the .ACTION section of the PHASE or the PROCESS section of the SNAP is executed. At this point, the timer is considered active in the scan. Correspondingly, an active timer may be disabled by executing the statement: TIMER (STOP, #t) in the .PROCESS section of a SNAP.

A phase may contain more than one active timer at any point. The timer scan resolves any conflict by selecting the timer with the highest priority and time-due. If there is no single, highest time-due and the priorities are the same, the scanner resolves the conflict by selecting the timers in the order in which they appear in the .ACTION list. A DO or an EXIT THEN DO statement from a SNAP is resolved by the process queue. The highest priority SNAP or PHASE is executed first. If a new PHASE is initialized all subsequent SNAPs in the calling PHASE are terminated.

An arbitrary division is made in the assignable priority levels: priority 0 (Interrupt) and priorities 1 through 7 are considered "foreground" levels that must be run to completion, once initiated. This guarantees completely dependable sequential operation. Each SNAP within each priority level must run to completion before the next priority may operate. Priority 0 is discussed later in this chapter. Priority levels 8 through 11 are considered interruptable levels that may be suspended. Each SNAP called to operate at these levels will be stacked in the process queue and called in priority sequence. When a background SNAP begins operation, no other background priority SNAPS, higher or lower, can intervene. The operating background SNAP can be suspended, however, if a foreground SNAP is called.

3.6 THE RUN-TIME SYSTEM/MAKING THE PIECES WORK

The INDAC 8/2 Executive executes the compiled program. Services provided to the user by the INDAC Executive include:

a. Time and Priority Scheduling of SNAPs

b. Dynamic core management and swapping

c. Dynamic I/O buffers

d. Operator communications (Command Decoder)

3.6.1 Time and Priority Scheduling

The INDAC Executive and the INDAC language permit a variety of sequencing and scheduling functions. Some of the functions are dependent upon the operation of a peripheral device; other functions are programmed to occur at specific times, intervals of time, or the next available opportunity. Due to the random operation of such devices as the operator's console, or the field interrupt device, the Executive must be prepared to resolve conflicts between the different requests to operate parts of the system.

The Executive resolves requests to operate by building a process stack. Following the successful operation of any member of the stack, the executive returns to the stack and executes the next scheduled process call. All process calls scheduled in the stack have an associated priority level ranging from 0 (highest) to 11 (lowest). If any priority level is inactive, the Executive proceeds to the next lower level. If no level is active, the Executive is considered idle and returns to the highest priority level to begin again.

3.6.1.1 Division of the Process Stack – The process stack is considered to have three primary scheduling divisions:

a. Interrupt Priority (0) – This level is available only to the Executive Command Decoder and the Interrupt Device Handlers.

b. Foreground Priority (1–7) – These levels are used for SNAPs that cannot be suspended in their operation.

c. Background Priority (8–11) – A SNAP running at this priority will be suspended in its operation if a Foreground or Interrupt call is scheduled for operation. Only one such background SNAP may be held in suspension at any moment of system operation.

3.6.1.1.1 Executive Command Decoder – The Executive Command Decoder may be called from the command Teletype. Once called, the following job requests can be made:

a. Display and/or modify system parameters

b. Stop or suspend operation of job

c. Activate a specific PHASE of the job

Operator communication through the Teletype is usually a time-consuming typing process compared with the speed of the computer. To avoid lengthy system tie-ups, the operator input is buffered and the Command Decoder is not activated until the entire request is completely entered. The Command Decoder is scheduled at priority level 0 when a completed process request is received.

3.6.1.1.2 Field Interrupt Processing – Field interrupt devices (the I/O handlers) are declared by the INDAC language statement:

```
.INTERRUPT (device names)
```

This statement must appear with the job specification statements following the .EQUIPMENT statement. The statement must declare all device handlers expected to call the INTERRUPT SNAP. For example:

```
.INTERRUPT (UDCP)
```

The name UDCP identifies the point-table I/O handler for the UDC. This name is declared to the INDAC software system by GENDAC if the point-table handler is selected. Programming details employing the UDCP handler are covered in Paragraph 3.7.

The actual INTERRUPT SNAP that will service the interrupt is declared by the INDAC language statement:

```
DO SNAP #t PRIORITY INTERRUPT
```

This statement must appear as part of the PHASE scheduling parameters immediately following the .ACTION statement of a phase. Only one PRIORITY INTERRUPT SNAP call is permitted within a phase. For example:

```
.ACTION
DO SNAP #10 PRIORITY INTERRUPT
```

This statement declares the INTERRUPT SNAP to the interrupt device handlers. The INTERRUPT SNAP request will be placed on the process stack at priority level 0 (first-in, first-out) whenever called. Since only one PRIORITY INTERRUPT SNAP call is permitted, the INTERRUPT SNAP algorithm must first determine which device interrupted if more than one interrupt device is declared by the .INTERRUPT statement. The pseudo-device STATUS will contain the index to the device specified in the INTERRUPT statement when a field interrupt occurs. Therefore, the following statements may be used to segment the SNAP for handling more than one interrupt device:

```
GET (STATUS) IDEX
GOTO (1,2,3), IDEX
1   Code for
    device 1
EXIT
2   Code for
    device 2
EXIT
3   Code for
    device 3
EXIT
```

The INTERRUPT SNAP is activated by a hardware interrupt-level I/O handler. Once activated, the SNAP will run until it releases control through an EXIT or EXIT THEN DO statement. The INTERRUPT SNAP itself cannot be interrupted.

3.6.1.1.3 Foreground Processing – SNAPs assigned priority levels 1–7 are considered foreground segments. Once a foreground priority SNAP is activated, system control is retained until that SNAP executes one of the following statements:

a. EXIT THEN DO SNAP #t PRIORITY k

b. EXIT THEN DO PHASE #t PRIORITY k

c. EXIT

Statements *a.* and *b.* above allow the SNAP to release control, but they establish a process request at the stated priority level. Statement *c.* releases control unconditionally. In all three cases, the Executive interprets the EXIT key word, terminates the SNAP, and returns to the top of the process queue for the next request. Statement *b.* schedules an action that will unconditionally release the operating phase, eliminate all associated timers of that phase, and establish a new phase for operation. A foreground SNAP may also execute the following statement calling a background priority SNAP.

```
DO SNAP #t PRIORITY k
```

In this case, the background priority SNAP is scheduled and the current SNAP proceeds with the next language statement.

3.6.1.1.4 Background Processing – SNAPs assigned priority levels 8–11 are considered background segments. Any background SNAP may be interrupted in its operation at the following points:

a. While the SNAP is processing a source language statement.

b. Any bid for the teletype core buffer that cannot be serviced (all bids are broken down to no more than 20 characters).

c. An execution of an EXIT statement.

A background priority SNAP may also execute the following statement calling a foreground priority SNAP:

DO SNAP #t PRIORITY k

In this case, the foreground SNAP is scheduled and the background SNAP is suspended. The Executive then returns to the process queue to activate the scheduled SNAP.

3.6.2 Dynamic Core Management and Swapping

3.6.2.1 Job Specification Segment – This segment of the program is the .EQUIPMENT, .INTERRUPT, .STORAGE, .HEADER, and .FORMAT information that the user defined at system level in his job. This information is transferred to core during the initial call for the job and remains in core throughout the life of the job. This area is never swapped.

3.6.2.2 The PHASE Segment – This segment contains the scheduling of the SNAPS that may be run under that PHASE. The PHASE segment exists in core only when that .PHASE is operational. If a PHASE is released and then recalled, that PHASE will be completely initialized on recall.

3.6.2.3 The SNAP Segment – The base core location of all of these units, and the address referencing within each unit to the other unit, are all precalculated during compile time. The relative disk addresses of each of the units are all converted to absolute addresses by SPUT – the System Put-Together Program – before the Executive is activated. Thus, all calls during the operation of the Executive are absolute; there is no time lost in table look-up, relocating, or link loading.

3.6.2.4 External (Disk-Resident) Subroutines and Functions – The INDAC 8/2 System allows the user to develop external subroutines and functions that may be called at run time. The system also allows the user to call mathematical functions supplied with the library. Calls for these routines direct the Executive to transfer the requested code from the disk to allocated call pages, in core. The Executive maintains two subroutine call areas: one reserved for interrupt or foreground priority SNAPs and the other reserved for background priority SNAPs. Because there is only one reserved area for each priority group, subroutine nesting is not permitted (that is, one external subroutine may not call another external subroutine).

3.6.2.5 Switching Priority Levels – The priority level at which a SNAP operates is determined by the process call scheduling the SNAP, rather than by some pre-assigned value associated with the actual SNAP. It is possible, therefore, to schedule a single SNAP within a PHASE to operate at different levels of priority. The Executive is structured to take advantage of this as follows:

a. Common routines are not re-entrant but, maintain linkage parameters in page zero. This reduces the overhead time and core of re-entrancy (significant on the PDP-8), yet allows the linkage parameters to be saved on demand.

b. A save area is reserved in core for the linkage parameters. This save area is used when a SNAP operating at a background priority level is suspended.

(continued on next page)

c. A swap area is reserved on the disk. This area is used when a suspended SNAP must be moved out of core.

The suspension of a SNAP, therefore, takes place in two stages. Initially, the Executive reacts to the suspend command by transferring the linkage parameters for the background priority to a reserved core area. If the next scheduled request is to operate that same SNAP at a foreground priority level, then the SNAP is directly executed and no disk transfer is required. If the next scheduled request is to operate a new SNAP, then the suspended SNAP is swapped to the disk to make room for the new request.

With this system, it is practical to develop a program of one SNAP, run that SNAP at different priority levels, and never access the disk again after the startup procedures.

3.6.2.6 Executive to SNAP Communication – Since SNAPs may operate at different priority levels, the Executive must provide some technique of communicating to the SNAP the reason why it was scheduled. Communication between the Executive and source-level code is provided generally by the STATUS item; a one word, integer item available through the pseudo-device command: GET (STATUS) I_a. When a SNAP is called into execution by the Executive, the STATUS item is set to one of three values, depending upon the original function that scheduled the SNAP to operate.

a. SNAP scheduled by a timer – STATUS contains the index value of the timer in the activity list (for example, the third timer statement would have index value 3).

b. SNAP scheduled by a device interrupt handler – STATUS contains the index value of the interrupting device in the .INTERRUPT (. . .) statement; if only one device is declared in the statement, then it is not necessary to test for the interrupting device.

c. SNAP scheduled by a DO SNAP . . . statement – STATUS contains a 0.

To differentiate which value of *a.* or *b.* STATUS contains (only required for SNAPS that run at both interrupt and foreground priority), the pseudo-call GET (PRIORITY) I_a will set I_a to the current priority level of the SNAP running.

3.6.3 Dynamic I/O Buffers

The Executive provides a number of I/O handlers to service the different devices supported by INDAC 8/2. Those handlers that require core buffers (FILE, TTY, etc.) request allocated core from the Executive. Such allocations are on a dynamic basis by job; thus, for each job only the core required to support the I/O handlers for that job is allocated. This is opposed to many systems where the user defines the total set of I/O devices supported by the system and must relinquish the core whether the specific job uses all those devices or not. The 8K Executive allocates buffers in one page segments. The total number of words allocated, subtracted from the highest location allowed (7000) will yield the top limit of User Area. This information, combined with the top limit of SNAPs supplied by the Compiler, is sufficient to calculate the core requirements of the job.

3.6.4 Operator Communication

INDAC 8/2 applications vary from almost completely automatic operations to highly interactive, almost manual control of systems and experiments. A wide variety of operator communication is available, ranging from the Teletype console (using the Command Decoder supplied with the Executive) to more sophisticated operator consoles supplied by the user and interfaced to the digital I/O subsystems.

The standard INDAC 8/2 System is supplied with a Model 33 (optionally 35 or 37) Teletype. This Teletype is the Command Console and is required to initiate the different jobs. The Executive's Command Decoder recognizes this console and provides the user with a means of inspecting and modifying parameters during the execution

of a job. The Executive can be configured to support additional Teletypes and may have any or all Teletypes in Command mode. The command instructions recognized by the Executive are covered in Chapter 6, Executing the Program. The essential information is that the Command Decoder can access only permanently core-resident parameters. Such parameters are defined as system-level storage items in each job. It is possible to access a greater amount of data, but, this involves code which the user must create within his own job.

3.7 SERVICING FIELD INTERRUPTS

The contact interrupt modules of the UDC provide the necessary hardware interface for field devices capable of issuing interrupts. The point-table I/O handler (UDCP) provides a convenient method for obtaining the interrupt information. In addition, the UDCP handler allows the user to access his UDC by channel selection rather than by scanning a predefined set of channels.

A consistent method of accessing the UDC is held throughout the handler by using a point-table to define the channels required. The UDCE handler requires that the user define to the Compiler the explicit channels that will be used during his running program. The Compiler then allows a data item to be associated with the defined channels. While this has the advantage of coding simplicity, there is no provision for altering the number of channels to be scanned, or the channels that an operator or program would like to select at any given moment. The UDCP handler allows the user to develop a "point-table" defining the channels (or points) to be selected for a given scan. Any point-table may be associated with any data item (or array). This allows the input or output of information without pre-defining channels and data items for the Compiler.

3.7.1 Equipment Declaration Statement

The UDCP and associated information is declared in the INDAC program as follows:

```
↑       *UDCP
↑#1     INITIALIZE
↑#2     IDENTIFY
↑#3     TRANSFER
```

where the equipment control lines define the requests that may be made on the point-table I/O handler to supply information.

3.7.2 Language Statements

The INDAC language statements are detailed in the following paragraphs.

3.7.2.1 The Standard Call – The standard calls for accessing digital input and output functional modules are:

GET (UDCP) $\langle v_1, v_2 \rangle$
SEND (UDCP) $\langle v_1, v_2 \rangle$

where v_1 and v_2 are the two components of the data list. The first component is the point-table. This item may be a simple variable, an indexed variable, or an array defining the channels to be accessed. The second component specifies the data items that either contain the information to be output, or will receive the information to be input. Generally, the point-table and the data item contain the same number of elements; however, this is not required. The UDCP handler will sequentially process through both components until one component is exhausted. The first component to be exhausted terminates the I/O operation.

NOTE

The control lines declared in the .EQUIPMENT statement are not referenced in the standard call. Reference to these control options are required only to service contact interrupt modules.

3.7.2.2 INITIALIZE Request – The INITIALIZE request is required only for those modules that require the Executive to buffer information, that is, information relating to the previous status of the device. The initialize statement has the following form:

SEND (UDCP, #t) ⟨data list⟩

where #t is a tag reference to the device control line specifying the INITIALIZE command, and the ⟨data list⟩ is constructed as follows:

a. The first item of the data list must be a system-level storage array containing two (2) words of storage for every channel of Contact Interrupt (Generic Code 2).

NOTE

This array must be provided for the exclusive use of the Executive.

b. The second item of the data list, and all succeeding items, are point-table declarations. Each item is a simple variable or an array containing the channel numbers of those interrupting modules that require special buffering by the Executive (like the Contact Interrupt Module). Each point-table must be a system-level storage item. The Executive will reference these tables whenever interrupts occur, and the tables must be core-resident. Each array declared must contain channel numbers of the same generic type modules. The first channel declared in each array will be sampled for generic type by the Executive; the remaining channels of that array will be assumed to have the same generic code.

3.7.2.3 IDENTIFY Request – The IDENTIFY request is required for all modules that create a "PRIORITY INTERRUPT" in the job (that is, call the INTERRUPT SNAP). The identify statement has the following form:

GET (UDCP, #t) I_v

where #t is a tag reference to the device control line specifying the IDENTIFY command, and I_v is an integer variable. The command will return the generic code of the module forcing the system interrupt in the item I_v. This statement is normally followed by a computed GOTO statement using the item I_v. By this technique, the user can branch to the correct access statement for the specific generic code of the module.

A typical coding scheme for servicing a UDC with several channels of interrupt modules is to create a loop using the IDENTIFY request and the computed GOTO to control the loop. The GET statement is executed repetitively until a 0 is returned in the integer item. The following GOTO statement is then executed with a 0 value, causing a default condition. For example:

```
↑          *UDCP
↑#101      IDENTIFY
              ⋮
900        GET (UDCP,#101) IGEN
           GOTO (901, 902, 903, xxx), IGEN
           (default condition — no more interrupts outstanding)
              ⋮
901        (service UDC device error)
              ⋮
902        (service Contact Interrupt)
              ⋮
903        (temp, reserved)
              ⋮
xxx        (etc.)
              ⋮
           GOTO 900
```

3.7.2.4 TRANSFER Request – The TRANSFER request is used in conjunction with the IDENTIFY request. TRANSFER and IDENTIFY are a matched set and must occur in pairs or the Executive will generate an error condition. Any GET or SEND statement separating an IDENTIFY and TRANSFER pair is prohibited. The transfer statement has the following form:

GET (UDCP, #t) ⟨data list⟩

where #t is a tag reference to the device control line specifying the TRANSFER command, and the ⟨data list⟩ is formatted according to the specific generic type module being accessed.

In all cases where the TRANSFER command is used, the Executive will allow only one channel of data to be accessed, regardless of whether that channel is directly scanned from the UDC or buffered in core by an interrupt-level I/O handler.

The following UDC access request is available:

a. For Contact Interrupt Module (Generic Code 2)

GET (UDCP,#t) I_{p1}, I_{v1}, I_{v2}, I_{v3}

I_{p1} is the name of the point-table containing all the channel numbers that hold a Contact Interrupt board already specified under INITIALIZE.

I_{v1} will be supplied by the Executive with an index into the point-table I_{p1}, of the interrupting channel. If it is required, the statement:

LET ICHAN = $I_{p1}(I_{v1})$

will place the interrupting channel number in ICHAN.

I_{v2} is a logical item (may be array element) that will be supplied with a 12-bit identification of the interrupting channel status. The 12-bit word will contain a "1" bit for every relay of the Contact Interrupt board that has changed state (COS), in the direction wired by the user, since the last time that the board was sampled.

I_{v3} is a logical item (may be an array element) that will be supplied with the last scanned value (LSV) of the Contact Interrupt channel. This value will contain a "1" bit for every relay that is "closed". Note that "closed" refers to the state of the isolation relay on the module.

3.7.3 Sample Program

The following "program" was created to illustrate the usage of the UDCP handler. The UDC in the example has the following configuration:

a. A BM804 Flip-Flop Relay board in channel 2.

b. A BW732 Contact Interrupt board in channel 4.

c. A BM804 Flip-Flop Relay board in channel 5.

The function of the program is to indicate that an operator has pressed a push-button and to display the total number of buttons or switches closed, as well as the last button(s) or switch(es) closed. The BW732 board has been wired as Close-Only (Pulse-Close). The information is displayed on both the Console Teletype and the UDC. Both BM804 boards are connected to lamp indicators and the BW732 board is connected to either 12 switches or 12 push-buttons. Channel 5 will display the last change and channel 2 will display the current status of the input.

There are some features of the program that require explanation:

a. The array IRAY(1,6)... has been set aside for exclusive use of the Executive according to the requirements of INITIALIZE.

b. The array ICONIS(1,3)/4,104,105/... provides for the current channel 4 of Contact Interrupt and two expansion channels (containing Contact Interrupt modules) that the user would like to implement in the future. Since the Executive will not "see" any interrupts on these channels, there is no loss in time by including these channels in the table. If the empty channels had been specified first in the table, there would be a time loss in lookup and match for the active channel.

c. The array IDIGS(1,3)/5,2,-1/... contains a negative channel designation and also demonstrates that channels do not have to be specified in ascending sequence. The negative channel is another method of reserving a channel slot; but, for a totally different purpose. The Executive will bypass all negative channel declarations on input and output; however, the Executive will also bypass the data item associated with that channel. The user, for example, may code an output transmission to six channels using six items of data and dynamically modify the actual list of channels to receive the information, as well as the number of channels to be output. Conversely, the user may code the system to allow an operator to add or delete sample or display points to the process scanner.

d. The item IDIG2 is reserved in system-level storage and used in #100 .SNAP(102+1). The item is initially associated with the negative channel and will be bypassed by the Executive; however, if the operator, or another .SNAP entered a legal channel value in place of the "-1", the Executive would output IDIG2 to that channel.

```
                .EQUIPMENT
↑               *UDCP
↑#10            INITIALIZE
↑#11            IDENTIFY
↑#12            TRANSFER
                .STORAGE    IRAY(1,6). ICONIS(1,3)/4,104,105/,
↑IDEX, ICOS, ILSV, IDIGS(1,3)/5,2,-1/, IDIG2
                .INTERRUPT (UDCP)
#1              .PHASE
                .ACTION
                DO SNAP #100 PRIORITY INTERRUPT
                DO SNAP #2 PRIORITY 2.
#100            .SNAP
#121            .HEADER
UDC FAULT
#122            .FORMAT
"CONI INPUT" XX XXXX XXXX
                .PROCESS
100             GET (UDCP,#11) IA
                GOTO (101,102), IA
                EXIT
101             SEND (TTY,#121)
                GOTO 100
102             GET (UDCP,#12) ICONIS, IDEX, ICOS, ILSV
                SEND (UDCP) IDIGS, ICOS, ILSV, IDIG2
                SEND (TTY,#122) ICONIS (IDEX), ICOS, ILSV
                GOTO 100
#2              .SNAP
                .PROCESS
                SEND (UDCP,#10) IRAY, ICONIS
                . . .
                . . .
                EXIT
        .END
```

3.8 HANDLING THE CONSOLES

INDAC 8/2 is capable of handling up to four ASCII compatible (TTY type) consoles such as the ASR33, KSR35, VT05, VT06, LP30, etc. One console, which is standard in the INDAC 8/2 System, is permanently defined in the compiler tables as TTY. Three additional consoles, TTY2, TTY3, and TTY4, can be configured through GENDAC. The TTY console has been discussed in terms of the system output (message and logging) device in previous paragraphs. The use of the console in this application is termed the system mode. The standard TTY console and any additional consoles can also be used in the "operator mode" for operator guidance of the job. Two options are available in the operator mode:

System input
Command functions

The ability to perform command functions can be allowed through program control. Thus, consoles primarily designated for operator interface can be restricted for system input functions, that is, operator responses to programmed questions. The console designated for the designer or supervisor can be opened for command functions as well, so that he can inspect and modify system parameters, stop a job, or start another job. After a job is started (by calling SPUT and commanding the Executive to start the job) all consoles are placed in the system mode unless specifically programmed otherwise. If any console is placed in the operator input mode, and the operator responds with an answer, the console is automatically reverted back to the system mode. If the operator does not respond, the console remains in the operator input mode unless the program cancels the request.

3.8.1 Equipment Declaration Statement

Consoles and associated information are declared in the INDAC program as follows:

```
↑          *TTY _
↑#1        COMMAND
↑#2        CANCEL
```

where the equipment control lines define the requests that may be made of the I/O handler to handle information.

3.8.2 Language Statements

The language statements for accessing the consoles vary in format depending on the mode in which the user wishes to access the console.

a. System Mode – The following statements are acceptable when the console is in the system mode:

1. SEND (TTY, #t)
This statement is used to send the .HEADER or a .FORMAT statement tagged #t to the console. Normally, a .FORMAT statement would be referenced when the user wishes to place the console in the operator mode without having a data list to be sent. The declarative .FORMAT statement terminated with a question mark (?) should be used for this purpose.

2. SEND (TTY) ⟨data list⟩
This statement is used to send unformatted data (in exponential notation) to the console where ⟨data list⟩ is the list of items to be sent.

3. SEND (TTY,#t) ⟨data list⟩
This statement is used to send formatted data to the console where ⟨data list⟩ is the list of items to be sent and #t is the tag reference to the .FORMAT statement. The .FORMAT statement may be of the picture or declarative type. The declarative .FORMAT statement should not be terminated with a question mark control character (?) unless the user wishes to place the console in the operator mode.

4. SEND (TTY,#t_1, #t_2) ⟨data list⟩
This statement is used to send formatted data to the console and place the console in the operator mode with the command option permitted. The ⟨data list⟩ is the list of items to be sent, #t_1 is the tag reference to the control line containing the key word COMMAND, and #t_2 is the tag reference to a Declarative .FORMAT statement that is terminated with a question mark control character (?).

To access the Command Decoder of the Executive the user simply types CTRL/A; to release the Command Decoder the user types CTRL/P.

NOTE

Once a console is placed in the operator mode, it will remain in this mode until released. This may be accomplished by programming a CANCEL command or by the operator completing an input message.

b. Operator Modes

1. System Input Option

The console may be placed in this mode to allow the user to supply the job with additional information. Up to one line of information terminated with a CR/LF may be typed. The following statement must be used in the program to retrieve the information.

GET (TTY) ⟨data list⟩

where ⟨data list⟩ are the locations where the information in ASCII representation (one character per word) is to be stored. After this statement is executed, the program may examine the operator message.

2. Command Option

The console may be placed in this mode to allow the user to call the Executive Command Decoder to inspect and modify system parameters, to stop the job, or to start a new phase.

3. Cancel

The operator mode can be cancelled to revert the console to the system mode by using the following statement

SEND (TTY,#t)

where #t is the tag reference to the control line containing the key word CANCEL.

3.8.3 Program Examples

The following program examples are included to illustrate some techniques in programming the consoles. Example 1 illustrates two methods of placing a console in the operator mode. The coder has the option to allow any console access to the command mode by simply referencing the control line containing the key word COMMAND.

Example 1 — Placing console in operator mode

```
                .EQUIPMENT
↑               *TTY
↑ #100          COMMAND
  #1            .PHASE
                .ACTION
                DO SNAP #2 PRIORITY 2
  #2            .SNAP
  #3            .FORMAT ("ENTER VALUE-",?)
  #4            .FORMAT ("COMMAND MODE AVAILABLE",?)
                .PROCESS
    1           SEND (TTY, #3)
                      or
    2           SEND (TTY, #100, #4)
                EXIT
```

The INDAC output command SEND (TTY . . .), assumes that the specified console is in the system mode, not the operator mode. If the console happened to be in the operator mode because it was not released by the user, the output command will default to the next statement. If this conflict is possible, the "STATUS" item may be tested to determine if the output was successful.

Example 2 presents an INDAC code that waits for the console to be released from the operator mode. The STATUS item will be set to "0" when the output is successful.

Example 2 — Waiting for a console

```
101         SEND (TTY,#t)
            GET (STATUS) I_x
            GOTO (101), I_x
```

Any console placed in the operator mode may be cancelled to release the console under program control. The system designer can then take appropriate action. Example 3 illustrates some programming techniques for cancelling a non-responding console.

Example 3 — Canceling a non-responding console

```
            .EQUIPMENT
↑           *TTY
↑ #101      CANCEL
              ⋮
  #1        .PHASE
  #10       DO SNAP #3 DELAY 20 SEC PRIORITY 11
  #11       DO SNAP #4 DELAY 40 SEC PRIORITY 2
            .ACTION
            DO SNAP #2 PRIORITY 2
  #2        .SNAP
  #21       .FORMAT ("ENTER VALUE-", ?)
              ⋮
            .PROCESS
            SEND (TTY, #21)
            TIMER (START, #10)
            TIMER (START, #11)
              ⋮
            EXIT
  #3        .SNAP
            .PROCESS
            GET (TTY) IVALUE
              ⋮
            EXIT
  #4        .SNAP
              ⋮
            .PROCESS
            SEND (TTY, #101)
            EXIT
```

The SNAP that typed the output message and placed the console in operator mode was at a "foreground" priority level. This is usual though not required; however, notice that the SNAP containing the ...GET(TTY... is operated at a "run-interruptable" level. This operation guarantees that if the SNAP must be suspended, the entire system is not suspended also. By this technique, the Executive suspends the input requests and allows the "cancel" to take place.

a. The user may:

1. Output information to a console in system mode

2. Change a console to operator mode

3. Allow a console to access command mode

4. Cancel the operator mode of a console, placing it in system mode

5. Input information from a console . . . GET (TTY . . .

6. Test if an output request was successful

b. The Executive will:

1. Initialize the primary console in operator mode (command mode allowed) to access the Command Decoder

2. Initialize all other consoles in system mode to prevent unauthorized access to the system

3. Allocate a single page of dynamic buffer for each console, to be used for input or output of data respectively

4. Collapse the input or output buffer if a CANCEL command is received

5. Bypass the output request if a ...SEND command is received for a console in operator mode

6. Set the STATUS word to a "0" if an output request is successful or to "1" if the request cannot be honored. GET (STATUS) I_a will transfer the status word into I_a

7. Revert any console that has initiated a "run" command (↑D R# . . .) back to system output mode

8. Revert any console that has completed a GET (TTY) . . . back to system output mode

c. System Considerations:

1. Once the primary console initiates the first phase run command, all consoles will be in system output mode. It is the prerogative of the user to place any console in operator mode and to permit any operator mode console access to the Command Decoder and system-level storage.

2. Any SNAP requesting a . . .GET (TTY . . . will be suspended until the operator has completed the input. If the suspended SNAP is a "foreground" priority then, by definition, the entire system is suspended. It is therefore, adviseable to interrogate consoles from run-interruptable SNAPs.

CHAPTER 4
CONFIGURING A SYSTEM HAVING STANDARD DEC PROCESS I/O DEVICES

4.1 INTRODUCTION

A specific system can be built and configured by the guidelines and procedures contained in this chapter. The cold-start procedure for building and configuring the system is as follows:

Step	Procedure
1	Load the following programs: Manually enter the bootstrap loader for HINDAC HINDAC MSUP Monitor System Dump INDAC Support Programs INDAC System Tables MAKE 8/2 COMP1 COMP2 COMP3 COMP4
2	Run SGEN.
3	Load following programs: Executive 8/2 GENDAC
4	Run GENDAC to configure system.

After all programs are loaded and SGEN is run, the specific system can be configured using GENDAC and its library tapes. Currently, the GENDAC library tapes contain I/O handlers for standard DEC I/O devices and trig functions. As new I/O devices are developed and new applications or requirements encountered, the GENDAC library tapes will be updated accordingly by DEC. GENDAC, in conjunction with the library tape, generates a dialogue to enable the operator to select the desired I/O handlers and functions. Handlers for the TTY, pseudo-devices: KEYS, STATUS, PRIORITY and FILE are not included in the library since they are a permanent part of the Executive.

Once a library routine is selected for inclusion in the system, the routine becomes a permanent part of the system and cannot be deleted by GENDAC. If a routine is inadvertently selected, or if a routine is no longer desired, the complete system must be rebuilt starting with loading the Monitor System Dump (Paragraph 4.3). Then, when GENDAC is run again, it should be run under the initial run option. However, if the user wishes to add a library

routine, he can simply run GENDAC under the noninitial run option and select only that routine he wishes to add. If the user answers "Y" (yes) to a question pertaining to a routine already in the system, GENDAC will abort and print "CANNOT CONTINUE". At this point, the system communications tables are incomplete and the system must be rebuilt starting with loading the Monitor System Dump (Paragraph 4.3).

The following paragraphs contain the procedures for building and configuring the software system to complement the user's hardware. If a sample system was built (see Chapter 2), the system communications tables are configured to reflect the sample system. Since these tables cannot be modified, the system must be rebuilt starting with loading the Monitor System Dump (Paragraph 4.3).

NOTE

The PDP-8/E control panel differs slightly from other PDP-8 Computers. The procedures presented in this chapter detail the PDP-8/E controls. When INDAC is implemented with a PDP-8/I or L, use the START switch whenever CLEAR/CONT is specified in the procedures; use LOAD ADDR whenever EXTD ADDR LOAD or ADDR LOAD is specified in the procedures.

Paragraphs 4.2 and 4.3 contain procedures that are designed for starting the building process from a cold start; that is, the monitor head is not in core or disk resident (refer to Chapter 5 for the procedure to bootstrap the Monitor into core). If the monitor head is in core (the monitor period is typed when CTRL/C is pressed), the MSUP program required for loading the Monitor System Dump (Paragraph 4.3) can be loaded from the high-speed reader with the following command string, instead of with the cold-start procedure described in Paragraph 4.3.

```
.LOAD↵
*IN-R:↵
*
ST=200
↑↑ (User types CTRL/P after each ↑)
```

Then continue with Step 10 of Paragraph 4.3.

4.2 LOADING THE HINDAC PROGRAM

To load the computer from a cold-core start proceed as follows:

NOTE

Verify that RUN light is off. If the light is on, press HALT and return HALT switch to up position.

Step	Procedure
1	Load the switch register with 0000.
2	Press EXTD ADDR LOAD.
3	Load the switch register with 0027.
4	Press CLEAR.
5	Press ADDR LOAD.

(continued on next page)

Step	Procedure
6	Successively deposit the following:

Location	Instruction
0027	6011
0030	5027
0031	6016
0032	7450
0033	5027
0034	7012
0035	7010
0036	3007
0037	2036
0040	5027

7	Load the HINDAC (Tape 1) program in the high-speed reader – begin anywhere in the initial blank tape portion.
8	Load switch register with 0031.
9	Press ADDR LOAD.
10	Press CLEAR.
11	Press CONT.

NOTE

The tape should now read completely through the reader and stop on the trailer portion (code 0200) of the tape. The computer should also halt. At this point, both RIM and the Binary Loader are in core. If the RUN light does not go out or if the tape does not read in properly, repeat this procedure.

4.3 LOADING THE MONITOR SYSTEM DUMP AND INDAC FILE TAPES

After the HINDAC program is successfully loaded, the Monitor Support program is loaded using the Binary Loader:

Step	Procedure
1	Place the MSUP tape (Tape 2) in the high-speed reader. Set the leader portion (code 200) of the tape under the read lamp.
2	Load switch register with 7777.
3	Press ADDR LOAD.
4	Set switch register to 3777.
5	Set the rotary console switch to AC.
6	Press CLEAR, then CONT. The tape should now read until the trailer portion (code 200) is under the read lamp. At this point, the computer will halt with the AC containing 0; the link may be on.
7	Load switch register with 200.

(continued on next page)

Step	Procedure
8	Press ADDR LOAD.
9	Press CLEAR, then CONT. The MSUP program will begin a series of questions to determine the operation required. Each question may be answered by a single letter followed by a carriage return (designated ↵). After it is loaded, the first question is: LOAD, DUMP or VERIFY-
10	Type L↵ The next question asked is: ENTIRE DISK OR FILES?-
11	Type E↵ The MSUP program will come to a halt after typing the "E↵". The program is now waiting for the user to load the Monitor System Dump (Tape 3) in the high-speed reader.
12	At this point, before loading the tape, set the switch register to 0001. The LSB switch is used to control the loading of the Monitor System Dump tape. When the switch is in the "1" position, the program will idle after completion of loading the current block (one "block" of information). When the switch is set to "0", the program will resume loading. **NOTE** **Make certain that all checksums are torn from the end of the Monitor System Dump tape before loading.**
13	Place the Monitor System Dump (Tape 3) in the high-speed reader. Start tape at leader portion (code 200).
14	Press CONT. At this point, if you have correctly set the LSB switch, the program will be idling.
15	Set the LSB to "0" to start the loading process. At any time, setting the LSB to "1" will stop the tape at the next leader/trailer. Loading will resume whenever the LSB is reset to "0". **NOTE** **During the loading process, if the reader malfunctions, MSUP will print "CHECKSUM OR VERIFY ERROR" and halt. Back the tape up one block (blocks are groups of punches separated by leader/trailer, code 200). Place the leader of the block that failed under the reader lamp. Set LSB for continued loading and press CONT. If the block will not load correctly, the tape has been damaged and must be replaced.**
16	When MSUP completes the loading of the Monitor System Dump (Tape 3), there will be a slight pause and the program will type out: LOAD, DUMP OR VERIFY-
17	Place the INDAC Support Programs (Tape 4) in the high-speed reader. Start the tape at leader portion (code 200).

(continued on next page)

Step	Procedure
	NOTE **Tear all checksums from the end of the tape (information following the last leader/trailer).**
18	Type L↵ The next question asked is: ENTIRE DISK OR FILES?-
19	Type F↵ When MSUP completes the loading of the INDAC Support Programs (Tape 4) there will be a slight pause and the program will type out: LOAD, DUMP OR VERIFY-
20	Place the INDAC System Tables (Tape 5) in the high-speed reader. Start tape at leader portion (code 200). **NOTE** **Tear all checksums from the end of the tape (information following the last leader/trailer).**
21	Type L↵ The next question asked is: ENTIRE DISK OR FILES?-
22	Type F↵ When MSUP completes the loading of the INDAC System Tables (Tape 5) there will be a slight pause and the program will type out: LOAD, DUMP OR VERIFY-
23	When both file tapes have been loaded type CTRL/C (hold both CTRL and C keys down), this will return control to the Disk Monitor System just loaded. The Monitor will respond with a period. At this point, the disk system is established for a single disk and the Disk Monitor is resident in core.

4.4 LOADING AND BUILDING THE INDAC COMPILER

Load the MAKE 8/2 (Tape 6) program using the Disk Monitor Loader. To load MAKE 8/2 from the high-speed reader, use the following command string:

```
.LOAD↵
*IN-R:↵
*
ST=200↵
↑↑ (User types CTRL/P after each ↑).
```

NOTE
Start the tape at leader portion (code 200).

MAKE 8/2 will execute and return to the Monitor. Now load the compiler tapes.

COMP1 (Tape 7)
COMP2 (Tape 8)
COMP3 (Tape 9)
COMP4 (Tape 10)

The loading sequence (using the high-speed reader) is the following for each tape:

```
.LOAD↵
*IN-R:↵
*
ST=2↵
↑↑ (User types CTRL/P after each ↑).
```

NOTE

Start tape at leader portion (code 200).

After the user's second CTRL/P control returns to the Monitor after about one minute. The same sequence is repeated for each compiler tape.

4.5 RUNNING THE SGEN 8/2 PROGRAM

SGEN 8/2 takes a single-disk system and expands the file structure into as many disks as are connected to the system (Limit 4), and protects the Executive and data file areas that will be used later in the system. Any disk units on the system that are not to be used by the INDAC System should be switched to "off".

To run SGEN 8/2, type the following sequence:

```
.SGEN↵
.SAVE ⟨SD⟩! 1400;↵
```

SGEN 8/2 expands the single-disk system to include as many as the user has DS32 expander units selected and returns to the monitor after a short pause. The SAVE command initializes the System Devices Table in preparation for the system configurator program.

4.6 LOADING THE INDAC EXECUTIVE 8/2

Load the INDAC Executive as follows:

Step	Procedure
1	Place the Executive 8/2 (Tape 11) in the high-speed reader. Start the tape at the leader portion (code 200).
2	Set the switch register to 3600.
3	Type HELD ↵ The HELD program will load the Executive and halt after loading. **NOTE 1** **Since paper-tape to disk operations are taking place, the loading may appear "jerky" or "erratic"; this is normal.** **NOTE 2** **At this point the AC register should be "0" indicating that the checksum comparison is correct. If the AC is not 0, repeat the Executive Loading procedure.**
4	Set the switch register to 7600
5	Press EXTD ADDR LOAD.
6	Press ADDR LOAD.
7	Press CLEAR, then CONT.

NOTE

Control returns to the Monitor after pressing CONT.

This completes the procedures required to build the initial system. Continue with the following procedures to build the specific sample system for checkout and experimental usage.

4.7 LOADING GENDAC TO CONFIGURE A SPECIFIC SYSTEM

The function of GENDAC is to configure the INDAC System for the specific requirements of each installation. To configure the system proceed as follows:

Step	Procedure
1	Load GENDAC (Tape 12) in the high-speed reader.
2	Set the switch register to 3600.
3	Type LELD ↵
4	The LELD program will load GENDAC and return to the Monitor after loading.

NOTE

Since paper-tape to disk operations are taking place, the loading may appear "jerky" or "erratic"; this is normal. About a third of the way through loading the tape the Teletype will echo a carriage return/ line-feed.

5	When loading is completed, LELD will return to the Monitor.
6	Press HALT, then return switch to normal position.
7	Set switch register to 0200.
8	Press ADDR LOAD, CLEAR, then CONT.
9	GENDAC will begin execution (see Figure 4-1).
10	The initial dialogue from GENDAC types the version number and a reminder to the operator that if he is unsure of the response to a question, type "?".
11	GENDAC will then ask if this is to be a 4K system. The response will be "N" because you do not have a 4K system.
12	GENDAC will request the mode of operation to be used via the question "*OPT-"; the response will be "B" for binary mode.

NOTE

In the binary mode GENDAC expects to find a formatted binary image produced by the assembly of a program such as the I/O Handler and Function paper tapes. The contents of these tapes is summarized in Charts 4-1 and 4-2. In the system mode GENDAC expects to find vector code produced by the INDAC compiler.

13	The next request is for the input device via the question "*IN-". The response will be "R: ↵ " for the high-speed reader.
14	GENDAC then types "↑", waiting for the binary tape to be loaded. Load the I/O Handler (tape 13) tape into the high-speed reader and reply by striking CTRL/P (hold CTRL key and strike P key). GENDAC echos ↑P.

(continued on next page)

Step	Procedure
15	Answer all questions. Answer "Y" (yes) if the I/O Handler is to be inserted into the system; answer "N" (no) if the I/O Handler is not to be inserted. In general, if you do not understand what the routine in question is, type "I" (inspect). Refer to Figure 4-1 for sample dialogue of current I/O Handler tape.

NOTE

GENDAC also asks whether CTRL/C is to be disabled when the console is in the Command mode. If the user answers yes to this question return to the Monitor is not possible via the Executive Command Decoder.

16 When GENDAC reaches the end of the I/O Handler tape it will type:

```
END OF LIBRARY TAPE .
*OPT-
```

17 If you wish to add one or more of the following functions:

```
SIN
COS
EXP
LOG
ATAN
```

the response to GENDACs question "*OPT-" should again be "B" for binary mode. If none of these functions are desired, respond with a carriage return to the question "*OPT-" to terminate GENDAC. GENDAC will then release control to the monitor system.

18 If GENDAC is terminated at this point in the system building phase, proceed to Chapter 5 for information on program preparation or to Chapter 7 for information on modifying the basic system. If one or more of the specified functions are to be added to the system, proceed as follows:

a. Respond with "B" to GENDACs question "*OPT-"

b. GENDAC will then type "*IN-" requesting the input device specification. Type "R:↵ " for the high-speed reader.

c. GENDAC then types "↑" and waits for the binary tape to be loaded.

d. Place the Function tape (Tape 15) into the high-speed reader and strike CTRL/P (hold CTRL key down and hit the P key). GENDAC echos "↑P."

e. Answer all questions. Refer to Figure 4-2 for sample dialogue of current Function tape.

f. When GENDAC reaches the end of the Function tape, it will type:

```
END OF LIBRARY TAPE .
*OPT-
```

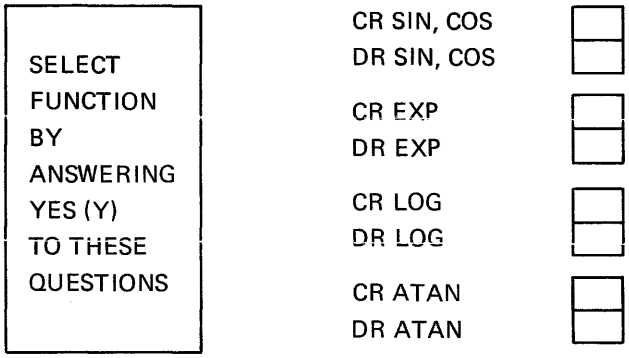
g. Respond with carriage return to terminate GENDAC and refer to Chapters 5 and 7 for operating and modifying the system.

Chart 4-1
INDAC I/O Device Handler Library Summary

SELECT ONE CLOCK I/O HANDLER (CLK8) BY ANSWERING YES (Y) TO ONE OF THESE QUESTIONS	60 Hz PDP-8/E LINE FREQ CLOCK	☐
	50 Hz PDP-8/E LINE FREQ CLOCK	☐
	PDP-8/E PROGRAMMABLE CLOCK	☐
	60 Hz PDP-8/I	☐
	50 Hz PDP-8/I	☐
	60 Hz PDP-8	☐
	50 Hz PDP-8	☐
DIGITAL I/O– POINT-TABLE I/O HANDLER	UDCP	☐
	GENERIC 2 INTERRUPT MODULE	☐
SELECT ONE ANALOG-TO-DIGITAL CONVERTER (ADC) I/O HANDLER	AF01	☐
	AF02	☐
	AF03	☐
	AFC8	☐
	AD01	☐
	AF04	☐
DIGITAL I/O– EXPLICIT I/O HANDLER	UDCE	☐
SELECT ONE DIGITAL-TO-ANALOG CONVERTER (DAC) I/O HANDLER	AA01	☐
	AA50	☐
HIGH-SPEED PAPER TAPE PUNCH	PTP	☐
ADDITIONAL CONSOLES	TTY2	☐
	TTY3	☐
	TTY4	☐

Chart 4-2

INDAC Function Library Summary



```
.LELD

.
GENDAC 427
TYPE '?' IF CONFUSED.
4K?-?
4K INDAC EXECUTIVE?-N

*OPT-B
*IN-R:
↑↑P

I/O LIBRARY INDAC 8/2 <U2IC>

SAVED <SD>!1400; FROM SGEN?-?
TYPE 'Y' TO INCLUDE,
     'N' TO OMIT,
     'I' FOR MORE INFORMATION.-I

HAVE YOU SAVED <SD> FOR INITIAL RUN
IF NOT TYPE CTRL/C TO RETURN TO MONITOR?-Y
CHKS...  STORED AT 10200

INITIAL RUN?-I

IS THIS THE INITIAL LOADING OF ANY I/O HANDLER?-Y
[ZC]  STORED AT 01176
END    ADDED TO IO CHAIN.

DISABLE CTRL/C IN COMMAND MODE?-I

DO YOU WISH TO DISABLE CTRL/C RETURNING TO THE
MONITOR FROM CONSOLE COMMAND MODE?-Y
SCAN  STORED AT 03235

60 HZ PDP-8/E LINE FREQ CLOCK?-I

60 HZ PDP-8/E WITH DK8-EA LINE FREQ CLOCK?-Y
CLK8  STORED AT 05102
CLK8  STORED AT 05045
CLK8  STORED AT 05316

50 HZ PDP-8/E LINE FREQ CLOCK?-I

50 HZ PDP-8/E WITH DK8-EA LINE FREQ CLOCK?-N

PDP-8/E PROGRAMMABLE CLOCK?-I

PDP-8/E WITH DK8-EP PROGRAMMABLE CLOCK?-N

60 HZ PDP-8/I OR PDP-8/L?-I

IS LINE FREQ 60 HZ FOR PDP-8/I OR PDP-8/L?-N

50 HZ PDP-8/I OR PDP-8/L?-I

IS LINE FREQ 50 HZ FOR PDP-8/I OR PDP-8/L?-N
```

Figure 4-1 GENDAC and I/O Handler Dialogue (Sheet 1 of 3)

```
60 HZ PDP-8 CLOCK?-I

IS PROCESSOR A PDP-8 WITH A 60 HZ CLOCK?-N

50 HZ PDP-8 CLOCK?-I

IS PROCESSOR A PDP-8 WITH A 50 HZ CLOCK?-N

UDCP?-I

UDC POINT-TABLE AND INTERRUPT MODULE HANDLER?-Y
UDC3  ADDED TO <**>
UDC1  ADDED TO <**>
UDCP  ADDED TO <SD>
UDCP  STORED AT 02200
UDCI   ADDED TO IO CHAIN.
UDC3  STORED AT 02000

GENERIC 2?-I

DO YOU HAVE UDC CONTACT-INTERRUPT MODULES?-Y
UDC6  ADDED TO <**>
UDC5  ADDED TO <**>
UDC5  STORED AT 07200
UDC6  STORED AT 07400
UDC0  STORED AT 02246

AF01?-I

A-D CONVERTER MODEL AF01?-N

AF02?-I

A-D CONVERTER MODEL AF02?-N

AF03?-I

A-D CONVERTER MODEL AF03?-N

ANY OF THE ABOVE A/D?-I

DO YOU HAVE AN AF01, AF02, AF03?-N

AFC-8?-I

DO YOU HAVE AN AFC-8?-Y
ADC   ADDED TO <SD>
[AF]  STORED AT 01504
AFC    ADDED TO IO CHAIN.

AD01?-I

A-D CONVERTER MODEL AD01?-N

AF04?-I

DO YOU HAVE AN IDVM AF04?-N
```

Figure 4-1 GENDAC and I/O Handler Dialogue (Sheet 2 of 3)

```
UDCE?-I

UDC EXPLICT CHANNEL I/O HANDLER?-Y
UDCE  ADDED TO <SD>
[DO]  STORED AT 01715

AA01?-I

AA01 DIGITAL TO ANALOG CONVETER?-N

AA50?-I

AA50 DIGITAL TO ANALOG CONVETER?-N

PTP?-I

HIGH-SPEED PAPER TAPE PUNCH?-Y
PTP   ADDED TO <SD>
[PP]  STORED AT 01320
PTP    ADDED TO IO CHAIN.

TTY2?-I

ADDED TELEPRINTER #2?-Y
[T2]  ADDED TO <**>
TTY2  ADDED TO <SD>
[T2]  STORED AT 01344
TTY2   ADDED TO IO CHAIN.
[T2]  STORED AT 03632

TTY3?-I

ADDED TELEPRINTER #3?-Y
[T3]  ADDED TO <**>
TTY3  ADDED TO <SD>
[T3]  STORED AT 01646
TTY3   ADDED TO IO CHAIN.
[T3]  STORED AT 03633

TTY4?-I

ADDED TELEPRINTER #4?-Y
[T4]  ADDED TO <**>
[T4]  STORED AT 06153
TTY4   ADDED TO IO CHAIN.
TTY4  STORED AT 03634
TTY4  ADDED TO <SD>
[T4]  STORED AT 01372
END OF LIBRARY TAPE .

*OPT-
```

Figure 4-1 GENDAC and I/O Handler Dialogue (Sheet 3 of 3)

```
*OPT-B
*IN-R:
↑↑P

INDAC 8/2 FUNCTION LIBRARY
[C1]  ADDED TO <**>

ANY OF: SIN, COS, EXP, LOG, ATAN?-I

TYPE 'Y' IF ANY OF THE ABOVE DESIRED.?-Y
[00]  ADDED TO <**>
[00]  STORED AT 07561
***SYSTEM BASE ALTERED!***
[L1]  ADDED TO <**>
[L1]  STORED AT 12000
[L0]  ADDED TO <**>
[L0]  STORED AT 06371

ANY DISK-RES: SIN, COS, EXP, LOG, ATAN?-Y
[D1]  ADDED TO <**>
[D1]  STORED AT 10753
[D0]  ADDED TO <**>
[D0]  STORED AT 07505

CR SIN, COS?-N

DR SIN, COS?-Y
SIN   ADDED TO <XS>
COS   ADDED TO <XS>
[SD]  STORED AT 17,7574

CR EXP?-N

DR EXP?-Y
EXP   ADDED TO <XS>
[ED]  STORED AT 17,7373

CR LOG?-N

DR LOG?-Y
[H1]  ADDED TO <**>
[H1]  STORED AT 11165
[H0]  ADDED TO <**>
[H0]  STORED AT 06574
LOG   ADDED TO <XS>
[GD]  STORED AT 17,6166

CR ATAN?-N

DR ATAN?-Y
ATAN  ADDED TO <XS>
[AD]  STORED AT 17,5765
END OF LIBRARY TAPE .

*OPT-
GENDAC COMPLETED.

.
```

Figure 4-2 GENDAC and Function Dialogue

CHAPTER 5
PREPARING THE PROGRAM

5.1 INTRODUCTION

After configuring a specific INDAC software system in accordance with Chapter 4, the user can prepare his source program on-line via the EDITOR and the Teletype. The EDITOR, COMPILER, and PIP are useful in preparing the source program. These programs are resident on the disk and can be called into operation simply by typing the assigned name (command string) in response to the Disk Monitor period (.). For example:

```
.EDIT
.PIP
.COMP
```

Other system programs that can be called into operation are:

```
.LOAD
.HELD
.LELD
.SGEN
.SPUT
```

5.2 MONITOR

INDAC uses the standard disk-oriented keyboard monitor, allowing the user to control the flow of INDAC 8/2 programs through the computer. The Disk Monitor also allows the user to LOAD, SAVE, or CALL any other PDP-8 program he wishes to run. (Refer to *Disk Monitor System Manual,* DEC-08-SDAB-D for a complete explanation of the Disk Monitor facilities.)

5.2.1 Monitor Residence

The Disk Monitor and INDAC system programs reside on disk. Part of the Monitor, called Monitor Head, also resides in core. It resides in the top page (locations 7600 through 7777) of field 0. The starting address of the Monitor is 7600. Nonresident portions of the Monitor, such as the Command Decoder, are automatically called into core as needed by the system programs.

5.2.2 Starting the Monitor

If the computer is turned off or stopped for any reason, the Monitor must be restarted at location 7600. This can be done by setting the switch register to 7600, pressing ADDR LOAD (EXTD ADDR, if necessary), CLEAR and CONT. If the Monitor is in the User mode, that is, a system program is running, CTRL/C may be typed to return to the Monitor mode. The rubout key may be used by the operator when his input command string is erroneous.

5.2.3 Bootstrapping the Monitor

If the Monitor Head is destroyed for any reason (Monitor cannot be started), the Monitor must be bootstrapped into core; turning the computer off and subsequently turning it on again does not destroy the contents of core. Use the following procedure to bootstrap the Monitor into core:

Step	Procedure
1	Toggle the following routine into core.

Address	Contents	Symbol
0200	6603	DMAR
0201	6622	DFSC
0202	5201	JMP .-1
0203	5604	JMP I .+1
0204	7600	7600
7750	7576	
7751	7576	

2 After toggling this routine into core, set switch register to 0200 and press ADDR LOAD, CLEAR and CONT.

5.2.4 Monitor Error Messages

If an illegal command string is entered, when the user types a carriage return, the Monitor responds with "?" to indicate invalid input.

Error messages output by the Command Decoder are given in Table 5-1.

Table 5-1
System Error Messages

Message	Meaning
?	Illegal syntax or miscellaneous error condition
D	Directory on the systems device is full
E	Too many inputs or outputs were entered
I	No such inputs
S	System I/O failure

Monitor time read or write errors cause a halt to occur. Persistence of this condition indicates a hardware failure or that write lock is enabled on the system device, as the system I/O routine attempts to read or write three times before halting.

5.3 SYSTEM PROGRAMS

The following INDAC System programs indicate other readiness to receive information by typing either an asterisk or a query:

.EDIT
.PIP
.COMP
.LOAD
.SPUT

The most common queries are:

*OUT-	Requests that the user specify one output device name. In the case of disk or DECtape, the filename to be assigned to the output data must also be specified.
*IN-	Requests that the user specify one or more (up to 5) input device names. For disk and DECtape, filenames of input files must also be specified.
*OPT-	Requests that the user specify one option or switch, entered as a single alphanumeric character; options available in each system program are described in system program descriptions (EDIT, PIP, COMP, etc.).
*	System program is ready to receive commands.

The following paragraphs describe the use and operation of EDIT, COMP, and PIP which are needed to prepare the INDAC source program. The system program, LOAD, is also discussed. This program is not needed in preparing the source program.

5.3.1 Command String Format

Command strings are composed of a few basic elements and follow certain rules of punctuation. To call a system program into operation and to give it necessary information, the following formats and conventions must be followed.

5.3.1.1 Device Names – Device names permitted in command strings are as follows.

S:	System device (disk or DECtape unit 0)
R:	High-speed paper tape equipment (reader or punch)
T:	Low-speed paper tape equipment on the Teletype (reader or punch)
Dn:	DECtape unit, if both disk and DECtape are present in the system (n = unit number, 0 through 7)

5.3.1.2 Filenames – Filenames are limited to four characters in length and can be composed of any combination of alphanumeric characters or special characters with the following exceptions.

a. Imbedded spaces should not appear in a filename.*

b. A filename cannot be one of the following words or symbols.

CALL SAVE ! , ; :

Extensions (n) to the filenames specified by the user are automatically appended by the system. They are used internally by the system and cannot be referred to or modified by the user.

SYS (n)	System program file in core bank n
ASCII	Source language program file (input to PAL-I Assembler or INDAC Compiler)
BINARY	Binary program file (output from PAL-I Assembler)

Filenames (and extensions) are meaningful only for file structured devices (disk and DECtape). If they are specified for other devices, they are ignored. Both the filename and extension name appear on directory listings produced by the list feature in PIP.**

*Note that the INDAC Executive is given the filename I OS; one reason for this unconventional use of an imbedded blank is to protect Executive from accidental destruction by the user (for example, deletion via PIP).

**AF in example means VERSION A, change F.

Example:

```
NAME  TYPE      BLK
AF
PIP   .SYS  (0)  0025
EDIT  .SYS  (0)  0016
LOAD.SYS    (0)  0011
.CD.  .SYS  (0)  0007
HGB   .ASCII (0) 0001
```

5.3.1.3 Punctuation – Punctuation within command strings is as follows.

, (comma)	Used to separate device names, when more than one is given in a command string.
:	Terminates each device name.

5.3.1.4 Special Characters – Special characters are used as follows.

↑C	If given while the system is in Monitor mode or while most system programs are running, control is returned to Monitor start (location 7600). Monitor responds with a dot. ↑C is typed by holding down the CTRL key and striking C. ↑C does not echo (does not print).
↑P	Typed in response to a ↑ typeout. Instructs the system to proceed with the next operation. ↑P can also be used to prematurely terminate certain operations while in progress (for example, the typing out of a file directory by the list option in PIP). ↑P is typed by holding down the CTRL key and striking P. ↑P does not echo (does not print).
)	Carriage return terminates current command string input. When typed alone, in response to a system query, it indicates that the user does not desire to specify the item (for example, device name) requested.
RUBOUT	Causes the current command string to be ignored, and the system returns to the beginning of the command string and is ready to receive a new command. RUBOUT does not echo.

5.3.2 Examples of Command Strings

These following examples illustrate the elements and rules previously explained. Samples of both Monitor mode and System mode operation are given.

a. Monitor Mode Commands:

.EDIT)	Call system program file, EDIT, from disk into core for execution
.PIP)	Call system program file, PIP, from disk into core for execution
.COMP)	Call system program file, COMP, from disk into core for execution

b. System Mode Dialogue:

*IN-S:PRO2)	Use the file PRO2 on the disk as the input file
*IN-S:TST1,R:)	Use the file TST1 on the disk and one file from the high-speed paper tape reader as the input files

(continued on next page)

b. (cont)

*OUT-S:SPEC ↵	Write the output file on the disk and assign it the filename SPEC
*OUT-T: ↵	Punch the output on the Teletype paper-tape punch
*OPT-B	Select option B

5.3.3 Editor

The Editor (Disk System Editor) enables the user to generate and edit INDAC source programs on-line from the teleprinter keyboard. The INDAC program may be either printed on the teleprinter, punched on paper tape using the high-speed punch, or saved on the disk as an ASCII file. To use the Editor, the user must call EDIT via the Monitor; this can be done only in response to a period. If a period is not present as the last system response, the user must type CTRL/C which should appear as follows:

```
.EDIT ↵
```

The Editor is transferred from disk into core and responds by typing:

```
*OUT-
```

The user selects one of the following output devices: (T:) for low-speed reader/punch; (R:) for high-speed reader/punch; (S:name) for output to the systems device on a file called name and types his choice immediately after OUT-. If the specified device is not valid, that is, not declared when building Monitor, Editor will respond with an error message (refer to Paragraph 5.3.3.6) and return control to Monitor. Thus, the user must call EDIT and respond to *OUT- with a valid device. When Editor recognizes a valid device, it responds with *↵ (asterisk, carriage return/line feed) and *IN-, as shown below.

```
*
*IN-
```

The user now specifies the input device by typing T:↵, R:↵, or S:name↵ or ↵ in the same manner as when replying to *OUT-, above.

The Editor now responds with:

```
*OPT-
```

asking the user to specify one of the following options.

B	Preserve blanks. Editor normally replaces multiple blanks (spaces) with tabs, resulting in considerable saving of space on the system device.
D	Enter dynamic deletion mode if input is from the system device. As the file is read, it is deleted from the system device, thus allowing space for output if desired. (Filename remains on the directory but without any assigned blocks.)
C	Combine the functions of B and D options.
↵	None of the above options; assume conversion of two or more blanks to tabs, and not D.

After the user has specified one of the options listed above, Editor responds with a carriage return/line feed and asterisk. The entire printout might appear as follows.

```
.EDIT )
*OUT-S:CLK1 )
*
*IN-T: )
*
*OPT-B
*
*
```

The appearance of the last asterisk in the preceding example indicates that Editor is ready to accept and operate on the user's symbolic program.

The user may now type the symbolic program directly into core by using the A command.

```
*A )
```

To list a file called CLK1 with a new output file (CLK) declared, the command string and printout would be as follows:

```
.EDIT )
*OUT-S:CLK )
*
*IN-S:CLK1 )
*
*OPT-B
*R )
*L )
            .STORAGE IA
#1          .PHASE
#10         DO SNAP #100 AT 15:00 PRIORITY 1
#11         DO SNAP #101 EVERY 1 SEC PRIORITY 2
            .ACTION
            DO SNAP #104 PRIORITY 5
#104        .SNAP
            .PROCESS
            TIMER (START,#10)
            TIMER (START,#11)
            EXIT
#100        .SNAP
#102        .FORMAT ("3:00:00 PM ALLS WELL")
            .PROCESS
            SEND (TTY,#102)
            EXIT
#101        .SNAP
#103        .FORMAT (0,/)
            .PROCESS
            LET IA=IA + 1
            SEND (TTY,#103) IA
            EXIT
            .END
```

5.3.3.1 Modes of Operation – To distinguish between editing commands and the actual text to be entered in the buffer, the Editor operates either in Command mode or Text mode. In Command mode, all input typed on the Teletype is interpreted as commands to the Editor to perform some operation, or to allow the operator to perform some operation on the text stored in the buffer. In Text mode, all typed input is interpreted as text to replace, be inserted into, or be appended to the contents of the text buffer.

After being loaded into core memory the Editor is in Command mode; that is, the program is waiting for a command. The user types the desired command code and terminates it by striking the carriage return (RETURN) key. This nonprinting character (represented by ↵) tells the Editor to carry out the command. The Editor then enters Text mode and responds with a line feed character (represented by ↓) as soon as it has processed the command and begun the operation.

With the Editor in Text mode, the user types the desired corrections or insertions to his text. To terminate the text he enters a form feed (CTRL/FORM combination) to tell the Editor to return to Command mode. The Editor answers by ringing a bell to indicate the transition back to Command mode.

5.3.3.2 Input Commands –

Command	Action and Explanation
R ↵	READ a page of text from the paper-tape reader or disk as specified. The Editor will read information from the input tape or disk until a form feed character (CTRL/FORM key combination) is detected. All incoming text except the form feed is appended to the contents of the text buffer. Information already in the buffer remains there. In the case of input via the photoelectric reader, the end of the tape will be interpreted as a form feed and the Editor returned to Command mode, if an actual form feed character does not appear on the tape. In the case of input via the Teletype reader, a form feed must be entered via the keyboard to return the Editor to Command mode, if an actual form feed character does not appear on the tape. If this is not done, the READ command is still in effect and all subsequent commands will be interpreted erroneously as text and appended to what was just read from tape. Any rubout encountered during a READ command will be ignored. (See RUBOUT.)
A ↵	APPEND the incoming text from the teleprinter keyboard to the information already in the buffer (the buffer may be empty initially). The Editor will enter the Text mode upon receiving this command and the user may then type in any number of lines of text. The new text will be appended to the information already in the buffer, if any, until the form feed (CTRL/FORM) key combination is struck. By giving the APPEND command with an empty buffer, a symbolic program tape may effectively be generated on-line by entering the program via the keyboard. The APPEND command must not be used to read paper tapes from the Teletype reader since every rubout on the tape will delete a character.

NOTE

In both of these commands, the Editor returns to the Command mode only after the form feed character.

5.3.3.3 Output Commands – Output commands are subdivided into List and Punch commands. List commands will cause the printout, on the Teletype, of all or any part of the contents of the text buffer to permit examination of the text. Punch commands provide for the output of leader/trailer, form feeds, corrected text, or for the duplication of pages of an input tape. List or Punch commands do not affect the contents of the buffer.

a. List Commands – The following commands cause part or all of the contents of the text buffer to be listed on the Teletype.

Command	Action and Explanation
L)	LIST the entire page. This causes the Editor to list the entire contents of the text buffer.
nL)	LIST line n. This line will be typed out, followed by a carriage return and a line feed.
m,nL)	LIST lines m through n, inclusive. Lines m through n will be printed on the Teletype.

The Editor remains in Command mode after a List command and the value of the current line counter is updated to be equal to the number of the last line printed.

b. Punch Commands – The following commands control the punching onto paper tape of leader/trailer, text and form feeds.

Command	Action and Explanation
P)	Proceed and output entire contents of the buffer followed by a form feed and return to Command mode.
nP)	Output line n, followed by a form feed, return to Command mode.
m,nP)	Output lines m through n, followed by a form feed, return to Command mode.
E)	Process entire file (perform enough NEXT commands to transfer the remaining input to the output file) and create an end-of-file indicator (legal only for output to the system device).
T)	TRAILER. This command causes about 4 in. of blank tape to be punched. If using low-speed punch, turn punch off before typing command then turn on immediately after typing carriage return.
N)	NEXT. This is a utility command that combines the functions of four commands. It punches the contents of the buffer, punches some blank tape, a form feed and more blank tape, kills the buffer, and reads in the next page of text from the input device specified (that is, it executes P), K), R)).
nN)	Execute the above sequence n times. If n is greater than the number of pages of input tape the command will proceed in the specified sequence until it reads the end of the input tape, then it will return to Command mode.

5.3.3.4 Editing Commands – The following commands permit deletion, alteration, or expansion of text in the buffer.

Command	Action and Explanation
K ↵	KILL the entire page in the buffer. The values of special characters "/" and "." are set to 0. The Editor remains in Command mode.
nD ↵	DELETE line n. Line n is removed from the text buffer. The numbers of all succeeding lines are reduced by one, as is the line count.
m,nD ↵	DELETE lines m through n, inclusive. The line following n becomes the new line m and the rest of the lines are renumbered accordingly. The value of the current line counter, ".", is equal to the number of the line preceding the deleted line or lines. The Editor remains in Command mode after all DELETE operations.
nI ↵	INSERT the typed text before line n, until a form feed (CTRL/FORM) is encountered. The Editor enters Text mode to accept input. The first line typed becomes the new line n. Rubouts are recognized. Both the line count and the numbers of all lines following the insertion are increased by the number of lines inserted. The value of "." is equal to the number of the last line inserted. To re-enter the Command mode, the CTRL/FORM key combination must be entered to terminate Text mode. If this is not done, all subsequent commands will be interpreted erroneously as text and entered in the program immediately after the insertion.
I ↵	INSERT without an argument will insert text before line 1.
nC ↵	CHANGE line n. Line n is deleted, and the Editor enters Text mode to accept input. The user may now type in as many lines of text as he desires in place of the deleted line. Rubouts are recognized during any CHANGE operation. If more than one line is inserted, all subsequent lines will be automatically renumbered and the line count will be updated appropriately.
m,nC ↵	CHANGE lines m through n inclusive (m must be numerically less than n). Lines m through n are deleted and the Editor enters Text mode allowing the user to type in any number of lines in their place. All subsequent lines will be automatically renumbered to account for the change and the line count will be updated. After any CHANGE operation, return to Command mode is accomplished by entering a form feed (CTRL/FORM key combination) to terminate input. After a CHANGE the value of the current line counter, ".", is equal to the number of the last line of the change. Lines which are changed or deleted do not physically disappear from the buffer area, thus the space they occupied is not recovered upon completion of the command. This being true, it is possible to overflow the buffer by changing, or deleting and inserting lines. This possibility may be effectively eliminated by logically segmenting a program on paper tape into "pages" of 50 to 60 lines. This is done by punching groups of 50 lines followed by a form feed character (see output commands). There is a way to retrieve lost space in some cases by use of the SEARCH command (refer to SEARCH command which follows).
m,n$kM ↵	MOVE lines m through n inclusive to before line k (m must be numerically less than n, and k may not be in the range between m and n). Lines m through n are moved from their current position and inserted before line k. The lines are renumbered after the move is completed, although the value of the current line pointer, ".", is unchanged. Moving lines do not use any additional buffer space.

(continued on next page)

Command	Action and Explanation
m,n$kM↵ (cont)	A line or group of lines may be moved to the end of the buffer by specifying k as "/+1". Example: 1, 10$/+M↵. Since the MOVE command requires three arguments, it must have three arguments to move even one line. This is done by specifying the same line number twice. Example: 5,5$23M↵. This will move line 5 to before line 23. The Editor remains in Command mode after a MOVE command.
G↵	GET and list the next line that begins with a tag or label. The Editor begins with the line following the current line (line .+1) and tests for a line which does not begin with a tab, slash, or a space. This will most often be a line beginning with a tag or label.
nG↵	GET and list the first line after line n which begins with a tag. The Editor begins with line n and tests it and each succeeding line as previously described. Both G and nG update the current line counter after finding the specified line. However, if either version of the GET command reaches the end of the buffer before finding a line beginning with other than a tab, slash, or space, the current line counter retains the value it had before the GET was issued and a "?" is typed to indicate that no tagged line was found. The Editor remains in Command mode after a GET command.
nS↵	SEARCH line n for the character specified after the carriage return. Allow modification of line when character is found. The SEARCH command is one of the most useful functions in the Editor. It is also structured somewhat differently from the other Editor commands. After terminating the command nS with a carriage return the user has told the Editor to SEARCH line n, but he has not specified what to search for. The Editor is, therefore, waiting for the user to type a character. The character he types is taken as the object of the search but is not echoed. The Editor instead immediately begins typing out the specified line. After typing the character for which it is searching the Editor stops. All of the editing features are then available to the user. He may proceed using any of the following: *a.* Delete the entire typed portion of the line by typing "←" (back arrow). *b.* Delete the entire untyped portion and terminate the line and the search by typing↵ (carriage return). *c.* Delete from right to left one of the typed characters for each "\" (rubout) typed. *d.* Insert characters after the last one typed simply by typing them. *e.* Insert a carriage return and line feed, thus dividing the line into two, by typing ↓ (line feed). *f.* Continue searching to the next occurrence of the search character by typing CTRL/FORM. When typing stops all options are again available. *g.* Change the search character and continue searching by typing CTRL/BELL followed by the new search character. Each time the Editor types the character for which it is searching, typing stops and all or any combinations of the above operation may be carried out.

(continued on next page)

Command	Action and Explanation
m,nS)	SEARCH lines m through n inclusive as previously described. The search character is input after the carriage return and all of the options are available. The only difference is in point b. Typing) (carriage return) deletes the entire untyped portion and terminates that line, but the search continues on the next line. By typing CTRL/BELL to change search characters, all editing of a single line may be done in one pass. Clearly, typing CTRL/BELL twice will cause the search to proceed to termination, since the search character will now be BELL, which is not stored in the buffer.
S)	An additional feature is available to the more sophisticated user: by typing S with no arguments the entire buffer may be searched for occurrences of a single character. It must be remembered, however, that as with every CHANGE command, every SEARCH command uses additional buffer space for storage of the new line. This is obviously necessary, since the program can have no prior knowledge of whether the size of the line will be less than, greater than, or equal to that of the old line, and it must therefore assume that it will be larger. As the entire buffer is searched, a new image of the text is created in core that is guaranteed to occupy the same or less space than previously, since all deleted spaces have been removed. The Editor recognizes this and immediately moves the text image back to the top of the buffer space. Thus, the only prerequisite to condensing the text image is that there be enough core space left to contain another image of the edited text.

5.3.3.5 Special Characters and Functions – A number of keys have special operating functions. These keys and their associated functions are listed below. The nonprinting characters are noted; the symbols for these are shown in parentheses. All others echo the character in parentheses.

a. ↑P (CTRL/P) – During output, processing stops and control is returned to the Command mode.

b. ↑C (CTRL/C) – Always returns to Monitor.

c. Carriage Return ( ) nonprinting) – In both Command and Text modes, striking the carriage return key (RETURN) signals the Editor to process the information just typed. In Command mode, it allows the Editor to execute the command just typed. A command will not be executed until it is terminated by striking the RETURN key (with the exception of = which needs no)). In Text mode, it causes the line of text which it follows to be entered in the text buffer. A typed line is not actually part of the buffer until terminated by a carriage return.

d. Back Arrow (←) – The back arrow (←) is used for error recoveries in both Command and Text modes. When used in Text mode, ← cancels everything to the left of itself back to the beginning of the line. The user then continues typing on the same line. When used in Command mode, ← cancels the entire command and the Editor issues a "?" and a carriage return/line feed (CR/LF). Back arrow cannot cancel past a CR/LF in either Command or Text mode.

A ← ? (CR/LF)
THIS ← "HERE IS A TEXT MODE EXAMPLE" (CR/LF)
only the part in quotes is entered in the buffer

e. Rubout (\) is also used in error recovery in both Command and Text modes with one exception. When executing a READ command from either the paper tape or Teletype reader, rubouts are ignored completely and do not go into the buffer.

(continued on next page)

e. (cont)

It is necessary for the READ command to disable the rubout function since all tab characters on paper tape are, for timing purposes, followed by rubouts which would destroy the tabs. Rubouts are not stored in the text buffer but are inserted by the Editor following all tab characters on the output tape.

At any other time in Text mode (specifically if Text mode was entered via the APPEND,CHANGE, INSERT, or SEARCH command) typing rubout echoes a back slash (\) and deletes the last typed character. Repeated rubouts delete from right to left up to, but not including, the CR/LF combination, separating the current line from the previous one. Example:

THE QUUICK \\\\ ICK BROWN FOX (CR/LF) will be entered in the buffer as
THE QUICK BROWN FOX

When used in Command mode, rubout is equivalent to back arrow and cancels the entire command. The Editor then issues a "?" and a CR/LF combination.

f. Form Feed (CTRL/FORM nonprinting) — Form feed signals the Editor to return to Command mode. A form feed character is generated by depressing and holding the CTRL key and hitting the FORM key. This combination is typed while in Text mode to indicate the desired text has been entered and the Editor should now return to Command mode. The Editor rings a bell in response to a CTRL/FORM to indicate the transition back to Command mode. If Editor is already in Command mode when CTRL/FORM is typed, no bell will sound. CTRL/BELL is equivalent to CTRL/FORM except in the case of a SEARCH command (see editing commands).

g. Period (.) — The Editor keeps track of the implicit decimal number of the line on which it is currently operating. At any given time the symbol period (.) stands for this number and may be used as an argument to a command. Example: .L) means list the current line. .-1,.+1L) means list the line preceding the current line, the current line, and the line following it.

After a READ or APPEND command, the current line counter (.) is the number of the last line in the buffer. After an INSERT or CHANGE command, (.) is equal to the number of the last line entered. After a LIST command, (.) is the number of the last listed line. After a DELETE command, (.) is the number of the line immediately before the deletion. After a KILL command, (.) is 0. After a GET command, (.) is the number of the line typed by GET. After a MOVE or SEARCH command, the current line counter (.) is not updated and remains at whatever it was before the command.

h. Slash (/) — The symbol slash (/) has a value equal to the decimal number of the last line in the buffer. It may also be used as an argument to a command. Example: 10,/L) means list from line 10 to the end of the buffer.

i. Line Feed (↓ nonprinting) — Commands are terminated by a carriage return/line feed (CR/LF) combination and the lines on each page of text are separated by a CR and LF. The user need only strike the RETURN key, however, to terminate a command or input line, since the Editor automatically generates a line feed to follow each carriage return.

On input from paper tape, line feed characters are completely ignored. On output the Editor automatically punches a line feed following each carriage return.

Typing a line feed while in Command mode is equivalent to typing ".+1L) " and will cause the Editor to type out the line following the current one and increment the value of (.) by one.

j. ALT MODE (ALT nonprinting) — Hitting the ALT MODE key (ALT MODE) while in Command mode will also cause the line following the current line to be typed out and (.) to be incremented by one. If the current line is also the last line in the buffer, typing either ALT MODE or Line Feed will be answered by a "?" from the Editor indicating there is no "next" line. Some Teletypes have an Escape (ESC) key in place of the ALT MODE. The function is identical for ESC or ALT MODE.

(continued on next page)

k. Left Angle Bracket (<) — Typing Left Angle Bracket (<) while in Command mode is equivalent to typing ".-1L↵ " and will cause the Editor to echo < and then type out the line preceding the current line. The value of (.) is decreased by one so that it still refers to the last line typed out.

l. Equal Sign (=) — The equal sign (=) is used in conjunction with the pointers (.) and (/). When typed in Command mode, it causes the Editor to print out the decimal value of the argument preceding it, followed by a CR/LF. In this way, the number of the current line may be found (.=XXX), or the total number of lines in the buffer (/=XXX) or the number of some particular line (/-8=XXX) may be determined without counting from the beginning.

m. Colon (:) — Colon is a lower case character with exactly the same function as (=).

n. Blank Tape and Leader/Trailer — Both Blank Tape and Leader/Trailer (code 200) are completely ignored on an input tape, as are line feed characters and rubouts. Line feeds and rubouts are automatically replaced wherever necessary on output; blank tape and leader/trailer are not.

o. Tabulation (→ nonprinting) — The Editor is written in such a way as to simulate "tab stops" at ten space intervals across the carriage. When the user holds the CTRL key and strikes the TAB key, the Editor produces a tabulation. A tabulation consists of from one to ten spaces, depending on the number needed to bring the carriage to the next tab stop. Thus, the user may use the Editor to produce neat columns on the hard copy.

5.3.3.6 Editor Error Messages — Editor will print an error message consisting of a question mark whenever the user requests nonexistent information or uses an inconsistent or incorrect format in typing a command. The question mark will be followed by a carriage return/line feed and the command will be ignored. If the computer halts at location 2330, a system error has occurred while reading from the disk. The user should, therefore, run the disk maintenance tests to determine the cause of the error.

5.3.4 INDAC Compiler

After the INDAC source program is prepared and saved on the disk or on paper tape using EDIT, the compiler is used to translate the INDAC program from ASCII (source) to vector (object) code for disk storage. To use the compiler, the user must call COMP via the Monitor. This can only be done in response to a period. If a period is not present as the last system response, the user must type CTRL/C, which should cause the Monitor to type the needed period. The printout should appear as follows:

```
.COMP↵
```

The compiler is transferred from disk into core and responds by typing:

```
*OUT-
```

The user must then respond with the output device name and the filename to be given the compiler program. The device name will always be the disk (S:). The filename can be any four printing characters except the exclamation mark and the space. When the compiler recognizes the device and filename as valid, it responds with (*) (asterisk, carriage return/line feed) and *IN-, as follows:

```
*
*IN-
```

The user must now respond with the input device name and the filename given to the INDAC source program if it was stored on the disk during the Editing operation. The compiler will accept an input from the disk (S:) or the high-speed reader (R:). Because the INDAC source file is in ASCII and the object file will be stored on the disk as a system file, no conflict is caused by using identical names if the input file is on the disk. No name need be specified if the source file is received from the high-speed reader. Examples of the command strings for the two input methods follow:

Disk	Reader
.COMP↵	.COMP↵
*OUT-S:TST1↵	*OUT-S:TST1↵
*	*
*IN-S:TST1↵	*IN-R:↵
*	*
*OPT-↵	*OPT-↵
	↑ type CTRL/P in response to up-arrow.

After the user responds to the input question correctly, the compiler types:

*OPT-

This question is asked to inquire if a listing of the entire compiled program and accompanying data is desired. A reply of L produces the list. No list is produced if the CR (↵) key is pressed.

NOTE

Complete listings are lengthy and are normally used for system diagnostic purposes only.

If the input file is on the disk, compiling starts when the CR (↵) key is pressed. In the case of inputs from the high-speed reader, the user must then type CTRL/P in response to the up-arrow (↑).

If, during compilation, the system halts with a value of 0721_8 in the accumulator, there is no more room available on the disk. Remove some files from the disk, via PIP, and recompile the program. Any extra user files should be deleted.

5.3.4.1 Compiler Output – The compiler prints all system level variables, those items declared in the job-level storage statement, regardless of the option used following termination of the *OPT- command. The format of these variables is:

Variable name @ storage location

The storage location addresses are required later to modify or inspect parameters by the Executive Command Decoder. A possible system level storage table would resemble the following format:

IHR @ 1 IMIN @ 2 ISEC @ 3 IHR1 @ 4
ES @ 5 R @ 8 PRES @ 11 VPR @ 17

Immediately after the storage variables, the compiler prints pairs of identical 4-digit numbers and then a final 4-digit number, in the format:

0162 0162
0221 0221
0342 0342 1236

These numbers are SNAP and external subroutine level requirements that refer to the highest location used in each SNAP and SUBROUTINE segment. The row containing the three 4-digit numbers is always printed last by the compiler. The third number in the last row is the core address for the start of the user's system. The sum of the third number in the last row and the largest of the other numbers printed cannot exceed 6600_8 for an 8K system consisting of one Teletype. For each additional Teletype subtract 200_8 from 6600_8 and, if FILE is used, subtract an additional 200_8 from it. Therefore, for a system composed of two Teletypes and FILE, the sum cannot exceed 6200_8. If the sum of the two numbers is greater than the calculated limit, the program must be rewritten to occupy less space in memory, by segmenting the largest SNAP into two SNAPS, for instance.

After compilation is complete, control is returned automatically to the Disk Monitor System, as indicated by its reply of a period.

5.3.4.2 Errors During Compilation – If any errors occur in the user's source level program during compilation, the compiler prints an error message that consists of a number or a letter and a number. The error messages are printed out as they are encountered during compilation and are, therefore, interspersed with the system level parameters. Each error message is always followed by an address indicating exactly where the error occurs in the program. For errors found before the first program tag, the notation STRT is used; for errors found after the first program tag, the tag number is printed. The exact address is determined by the +n value printed after the tag, where +n means n statement down from the tag statement,

```
E7 STRT+3
TEMP @ 01 ILOG @ 10
H2 #200+4
17 #600+1
0614   0614   5376
```

In this example, the first error is an E7 encountered three statements from the start of the program; the second error is an H2 located four statements from tag #200; and the last error is a 17 found one statement from program tag #600. Notice that the system level parameters TEMP and ILOG were printed as they were encountered in the program. Refer to Appendix D for the meaning of each error message.

5.3.4.3 Correcting Compilation Errors – The Editor is used to correct any errors in the INDAC program. Use the following procedure to correct the errors.

Step	Procedure
1	Call the Editor by typing EDIT after Monitor's period printed after compilation.
2	Answer Editor's *OUT- with a new name to be given to the corrected program in the form S: filename. A different name must be used from that of the error containing ASCII program already on the disk. As when the user's program was loaded initially, the compiler will accept input from the disk or from the high-speed reader. If the high-speed reader is to be used as the input for the Compiler, the corrected INDAC program must be on paper tape.
3	Respond to the Editor's *IN- with the present name of the program. Use the notation S: filename for the program stored on the disk with the errors.
4	Choose the appropriate Editor option for *OPT-; B is generally a good choice.
5	Type R) to cause the Editor to read in the first block (buffer) of the program.

(continued on next page)

Step	Procedure
6	If the exact location of the error is not known, type /L (slash and L) to list the last line of the first block. If the error is in this block, use the Editor commands to correct the error.
7	After the errors in the first block are all corrected, or if there are no errors in the first block, advance to the next block containing an error. Type N) to write the first block onto the device specified in Step 2 and to read in the next block. Type N) enough times to read in the program blocks up to the block containing the next error. The same technique can be used to locate the block containing the error, namely, type /L to list the last line of the block presently in the buffer. Use the Editor commands to correct all errors when they are located.
8	When all errors have been corrected and if output is not to the system device, type N until a ? is printed, indicating an end-of-file condition. If output is to the disk, type E) to output the rest of the program to the disk.

When all the errors have been corrected, the program must be recompiled. Be sure to use the new file name given to the Editor in response to the output question, Step 2, before correcting the errors. Most users may find it advantageous to call PIP to delete the incorrect file from the disk to save space.

5.3.5 PIP

PIP (Peripheral Interchange Program) performs general utility operations, such as listing the contents of specified directories, deleting unwanted files from the system device, and transferring files between devices, and copying specified files. PIP enables the user to do any of the above operations merely by typing commands from the teleprinter keyboard.

To use PIP, the user must call PIP via Monitor which can be done only in response to a period. If a period is not present as the last system response, the user must type ↑C, which causes Monitor to type the needed period. The printout appears as follows:

```
.PIP)
```

PIP is transferred from the disk into core and responds by typing:

```
*OPT-
```

The user now selects one of the following options.

L	List entire directory of device to be specified
D	Delete a file to be specified
A or)	Copy ASCII file (destination and origin(s) to be specified)
B	Copy binary file (destination and origin to be specified)
F	Copy FORTRAN binary file (destination and origin to be specified)
U	Copy user file (file structured origin and destination to be specified)*
S	Copy system file (file structured origin and destination to be specified)*

The user types only the option character, to which PIP immediately responds with a carriage return/line feed. The user does not terminate the line with the RETURN key; it is a meaningful option.

*User and system files may not be copied onto paper tape because they are core images and have no defined paper tape format.

If the user selects an option using any character other than one of those listed above, the option is illegal, and PIP ignores the request, types ? (question mark), and asks for another option character. The output would appear as follows:

```
*OPT-G
?
*OPT-
```

The L option lists the entire directory of the system device (disk) or DECtape on which a directory exists. For example,

```
.PIP )                      User calls PIP
*OPT-L                      list option of the
*IN-S: )                    system device directory
FB=0241                     PIP types number of free (unused)
                            blocks remaining on specified device

NAME    TYPE  BLK

AF                          followed by filename and descrip-
PIP    .SYS  (0) 0025       tion; for example, PIP is a system
EDIT   .SYS  (0) 0016       program in field 0 and occupies 25(8)
LOAD.SYS  (0) 0011          blocks of storage
.CD.   .SYS  (0) 0007
HELD.SYS  (1) 0001
LELD.SYS  (0) 0001
SGEN.SYS  (0) 0015
SPUT .SYS  (0) 0012
```

When the user specifies the D (delete a file) option, PIP responds with

```
*FILE TYPE (A,B,F,U,S)-
```

where A, B, F, U, and S are legal options from which the user may choose; indicating ASCII, binary, FORTRAN binary (compiler output), user, and system program, respectively. Options F and U are not used in INDAC 8/2.

If the user's reply is S), indicating a system file, PIP asks

```
REALLY?
```

PIP will not delete a system file unless the user answers by typing

```
Y )        (meaning yes)
```

to the question. Any reply other than Y) causes PIP to repeat the FILE TYPE request. When the user types Y), PIP responds with

```
*IN-
```

and waits for the user to specify the device and filename of the system file to be deleted. The printout would appear as:

*OPT-D	Delete option specifying
*FILE TYPE (A,B,F,U,S)-S)	system file, user must reply
REALLY?N)	with Y),
*FILE TYPE (A,B,F,U,S)-S)	PIP repeats request,
REALLY?Y)	user replied correctly,
*IN-S:SGEN)	PIP needs device and filename,
*OPT-	file is deleted and PIP asks for
	the next option.

When the file has been properly identified and deleted PIP returns to ask for another option. If filename SGEN, in the example above, had not been on the specified device, PIP would have ignored the request and typed a "?" before asking for another option. For example,

*IN-S:SGEN)	SGEN is not the name of
?	a file on the specified
*OPT-	device

The user should not try to delete system files .CD. or LOAD.

Options A, B, and S are used to transfer files from one device to another in the INDAC 8/2 System. When the user has requested any of these options PIP responds with

*OUT-

and waits for the user to specify the destination or output file and, if the destination is disk or DECtape, the name of the file. For example,

*OPT-A	copy an ASCII file option
*OUT-S:TST1)	specifying the destination and filename

Only one destination is legal; if the user specifies more than one, PIP will ignore the response, type the error message E, and return control to Monitor. For example,

*OPT-A	copy an ASCII file option
*OUT-S: TST1, E	PIP recognizes the comma, which
	is used to separate file names;
	control returns to Monitor

NOTE

The L and D options return to PIP's option request (*OPT-) when the user responds illegally, and all other options return control to Monitor.

PIP indicates acceptance of the user's destination by responding with *, carriage return/line feed, and *IN-, and waits for the user to specify the input, that is, to state from where the input is to originate. An attempt to specify more than one input to any but the A option will cause PIP to ignore the response, type tne error message E, and return control to Monitor. For example,

*OPT-S	copy a System file option
*OUT-S: TST1)	specifying system device and filename
*	PIP accepts user's destination
*IN-S: TST2, E	input to system device, comma is
	used to separate device names control returns
	to Monitor

The A option will allow any combination of up to 11 ASCII input files to be merged into one output file in the order specified by the input list. The user, therefore, may write generalized subroutines as separate files to do his often repeated operations and then, by combining these with each specialized program before assembly, eliminate the need to rewrite such operations for each program. PIP acknowledges each legal input file by printing an *. If, however, the input file specified to any option is not found on the specified device, PIP prints I in place of the * and returns to the Monitor. For example,

```
*IN-S:TST1)          the file does exist; when the user types CTRL/P,
*↑                   copying begins
*IN-S: TST2)
I                    the file does not exist
.                    control returns to Monitor
```

If the user requests the B option, indicating he wishes to copy a binary file but the filename he has specified appears as an ASCII file, it is not acceptable; therefore, PIP prints an I and control returns to Monitor. The user can ascertain file types by using the L option and checking the file directory.

A summary of the copy features of PIP is presented in the following table.

	Option	Number of Input Files	Disk	DECtape	High-Speed Reader/Punch	Teletype
ASCII	A	11	Yes	Yes	Yes	Yes
Binary	B	1	Yes	Yes	Yes	Yes
FORTRAN Binary	F	1	Yes	Yes	Yes	Yes
User	U	1	Yes	Yes	No	No
System	S	1	Yes	Yes	No	No

Examples

.PIP ↵	User calls PIP
*OPT-L	and requests the list option
*IN-S: ↵	of the system device directory
FB=0111	PIP types number of free (unused) blocks remaining on specified device
NAME TYPE BLK	followed by filename and description;
AF	e.g., PIP is a system program in field 0
PIP .SYS (0) 0025	and occupies 25_8 blocks of storage
EDIT .SYS (0) 0016	
LOAD.SYS (0) 0011	
.CD. .SYS (0) 0007	
HELD.SYS (1) 0001	
LELD.SYS (0) 0001	
SGEN.SYS (0) 0015	
SPUT .SYS (0) 0012	
<PZ>.SYS (0) 0001	
<CM>.SYS (0) 0001	
<IF> .SYS (0) 0001	
<SD>.SYS (0) 0003	
<XS>.SYS (0) 0001	
COMP.SYS (0) 0022	
OVAL.SYS (0) 0125	
COPS .SYS (0) 0014	
I OS .SYS (0) 0152	
FILE .SYS (0) 0106	
DDC .ASCII 0035	
*OPT-D	User requests the delete option and
*FILE TYPE (A,B,F,U,S)-A ↵	specifies type of file, A(ASCII) and
*IN-S:DCC ↵	device and filename; file is deleted
*OPT-D	User requests the delete option and
*FILE TYPE(A,B,F,U,S)-S ↵	specifies type of file, S (system)
REALLY?Y ↵	(PIP double checks); Y is only the meaningful answer
*IN-S:SGEN ↵	User specifies file and filename; file is deleted
*OPT-L	User requests list option
*IN-S: ↵	and system device directory,
FB=0163	Note increase of 52_8 free blocks (see above)

(continued on next page)

```
NAME  TYPE  BLK

AF
PIP    .SYS (0) 0025
EDIT .SYS (0) 0016
LOAD.SYS (0) 0011
.CD.  .SYS (0) 0007
HELD.SYS (1) 0001
LELD .SYS (0) 0001
SPUT .SYS (0) 0012
<PZ>.SYS (0) 0001
<CM>.SYS (0) 0001
<IF> .SYS (0) 0001
<SD>.SYS (0) 0003
<XS>.SYS (0) 0001
COMP.SYS (0) 0022
OVAL.SYS (0) 0125
COPS .SYS (0) 0014
I OS  .SYS (0) 0152
FILE .SYS (0) 0106
```

Note removal of two files:
ASCII file DDC
and
System file SGEN

```
*OPT-D
*FILE TYPE (A,B,F,U,S)-S )
REALLY?N )
*FILE TYPE (A,B,F,U,S)-S )
REALLY?W )
*FILE TYPE (A,B,F,U,S)-S )
REALLY?Y )
*IN-S:I OS )
?
*OPT-D
*FILE TYPE (A,B,F,U,S)-U
*IN-S:NONE )
?
*OPT-D
*FILE TYPE (A,B,F,U,S)-A )
*IN-S:EDIT )
?
*OPT-D
*FILE TYPE (A,B,F,U,S)-B )
*IN-S:EDIT )
?
*OPT-
```

User requests delete option

Y is only response for deletion of a system file; other responses cause PIP to repeat the file type request

Even if user responds to REALLY? with Y, PIP will not delete the Executive

PIP knows NONE is not an existing user filename on the system device and indicates by typing ?

User requests ASCII file option

PIP also knows when the filename and file type do not match; EDIT is a system program

Merge, into an ASCII file on disk "TST1", one tape from the reader, one tape from the Teletype, one file from disk called SRC, and one file from DECtape 7 called SRC1.

```
*OPT-A
*OUT-S:TST1)
*
*IN-R:,T:,S:SRC,D7:SRC1)
*
*
*
*↑↑↑↑                    (type CTRL/P after each file)
*OPT-
```

Copy the system file PIP from disk to DECtape 3 using filename PIPX.

```
*OPT-S
*OUT-D3:PIPX)
*
*IN-S:PIP)
*↑                       (type CTRL/P)
*OPT-
```

Try to merge two binary files onto disk called BIN from paper tape.

```
*OPT-B
*OUT-S:BIN)
*
*IN-R:,E                 (list exceeded)
.
```

Try to copy an ASCII paper tape from high-speed reader, a nonexistent file from DECtape 5, and a paper tape from Teletype to high-speed punch.

```
*OPT-A
*OUT-R)
*
*IN-R:,D5:FOO,T:)
*                        (R: accepted as legal)
I                        (D5:FOO rejected, no such file
.                        on D5:)
```

5.3.6 Loading Programs — Disk System Binary Loader

The Disk System Binary Loader takes as input the binary coding produced by the PAL-I Assembler or other DEC assemblers and loads it into core in executable form. When loading is completed, Loader "disappears" after first entering the loaded program at the starting address typed by the user just prior to loading. Loader accepts input from the system device or paper tape.

5.3.6.1 Binary Loader Operating Procedures

```
.LOAD)
```

Direct Monitor to print Binary Loader from the system device into core for execution.

(continued on next page)

*IN-	Loader requests source of input(s). Type one or more device names, separated by commas. If an input device is a file-structured device, include filename(s). Up to five files can be specified.
Examples	
*IN-R:)	Input one tape from the paper tape reader.
*IN-R:,R:,R:)	Input three tapes from the paper tape reader.
*IN-S:INPT)	Input the file INPT from the system device.
*IN-S:BIN2,R:)	Input the file BIN2 from the system device and one tape from the paper tape reader.
*IN-S:BIN1,S:BIN2)	Input the files BIN1 and BIN2 from the system device.
*	If device(s) are valid and filenames (if any) are actually found on the system device, Loader responds with one asterisk for each correct input.
*ST=	Loader requests the starting address to which control is to be transferred when loading is completed. The address is typed in the form fnnnn where f = field number[1] (omitted if field 0), and nnnn = location with field.
Examples	
*ST=)	Load into field 0.
*ST=7600)	Return to Monitor after loading.
*ST=0)	
*ST=30225)	Load into field 3. Jump to location 255, field 3, after loading.
*ST=10000)	Load into field 1. Return to Monitor after loading into field 1.
	Loader now types a series of up-arrows, one at a time, as explained below.
	Following each up-arrow typeout, the user is required to perform one or more actions.
↑↑	First up-arrow: Loader is ready to load. If paper tape input, put the tape in the reader. Type ↑P.[2] Second up-arrow: End of pass. Type ↑P to jump to previously specified starting address.

(continued on next page)

[1]The f-digit forces Loader to start loading into the specified field until a "field setting" is found in the input file or tape.

[2]If Teletype paper tape equipment is used, type ↑P before turning on the reader.

↑↑ (cont)	Multiple Input Files An up-arrow is typed out as the processing of each input file is completed. If paper tape input, insert the next file in the reader and type ↑P. Repeat the above step until all files given in response to the *IN- request have been processed. After all files have been entered, type ↑P to jump to the previously specified starting address.

NOTE

After each input paper tape is read, the high-speed paper tape version of Loader loops until the user types ↑P to continue.

At this point, Binary Loader disappears and control is transferred to the previously specified starting address.

5.3.6.2 Binary Loader Error Messages – An illegal checksum error condition causes Loader to type "?" and return to Monitor after the user types ↑P or ↑C. Error messages for illegal filenames or devices are as specified in Paragraph 5.2.5.

CHAPTER 6
EXECUTING THE PROGRAM

After the INDAC source program is compiled error free, the object program is stored on the disk as a system file. SPUT, the system-put-together program, converts the object program into absolute disk format so that the INDAC Executive can locate and execute the source program at the most efficient speed. After converting the source program, SPUT bootstraps the Executive into core. The Executive performs scheduling, interrupt control, disk/core management, input/output, handles operator terminal communication, and executes the compiler generated code. Once bootstrapped into core, the Executive waits for the operator to start the program by giving it a specific run command. These commands are interpreted by the Executive Command Decoder which can be called by typing CTRL/A on any command console. After the run instruction (CTRL/D R # [Phase tag]) is given and the Command Decoder is released, the console reverts to the system mode. The only way the Command Decoder can be called again is via the INDAC program placing the console in the operator mode with the command mode permitted. Then, the operator may call the Command Decoder to inspect and/or modify system level parameters without interfering with the execution of the INDAC program.

The INDAC object file is stored on the disk only after error-free compilation, as indicated by the compiler printout. Before running SPUT, make sure that no SNAPS are too large to be run (refer to Chapter 5 for details). After the compiler finishes with its printout, it will release control to the monitor which then types the monitor period. The user may then call SPUT by typing:

```
.SPUT )
```

SPUT is transferred from disk to core and responds by typing:

```
*IN-
```

The user must now respond with the input device name and the file name given to the INDAC object program during compilation. SPUT will only accept an input from the disk (S:). When SPUT recognizes the device and file name as valid it responds by typing:

```
*FILE START XXXX
EXEC LOADED
```

The Executive is called directly by SPUT and only the name given to the object program file during compilation needs to be specified. SPUT types a decimal block number after *FILE START indicating where the file may start. This information is useful when more than one job is to be run and data collected and stored in the FILE by one job is to be used by another job. Different jobs usually vary in size and therefore the start of the FILE area will be different. Therefore, to run two related jobs where the FILE is used, the start of the file must be changed for the shorter job to eliminate the conflict. The start of the file can be changed under the control of the Executive Command Decoder in the same way the system level variables printed by the compiler can be changed.

After the Executive is called into operation, indicated by SPUT typing EXEC LOADED, system level parameters may be altered and/or the job can be started via the Executive Command Decoder. To interact via the Command Decoder, a Teletype must be in the command mode. Any Teletype may be a command Teletype. To attach any Teletype to command mode, type CTRL/A. If an ↑A is echoed, the Teletype is now in command mode. A Teletype is released from command mode by typing CTRL/P and the RETURN key.

Commands to the Command Decoder are terminated by typing the RETURN key. A RUBOUT deletes the last character in the current line and echoes that character; when no character is echoed, no characters remain in the line. An entire line may be deleted by typing CTRL/U.

The Command Decoder has two basic operations: value modification and run commands. Value modification, initiated by CTRL/V, executes interlaced; that is, phases are allowed to execute when the Command Decoder is not processing commands. System level variables, including the FILE START parameters, may be displayed and modified while the system continues to execute.

The run command processor, initiated by CTRL/D, can flag phases for running. A run command, in the form ↑D R#n, is used to flag the phase n, where n is the phase tag, for execution.

When the user types the RETURN key, the system begins to execute the flagged phase and will release the command Teletype to the system output mode (refer to Appendix H for a list of the available commands and their meanings).

Such a series of commands appears as follows on the Teletype:

```
↑A              (CTRL/A)
*↑DR#1
```

Typing CTRL/V indicates to the Command Decoder that some value changes are going to be made to the system level variables. There are four number systems available for examining and modifying the contents of a particular variable in memory.

Number System	Abbreviation	Variable Type
clock	C	System Clock
octal	O	Integer
decimal	D	Integer
exponential (floating-point)	E	Real

It may be helpful to remember that for access to memory you have a memory "CODE". The combination of floating-point and integer arithmetic provides increased data acquisition capability. For an integer variable, one whose name starts with an I, such as IMIN or ISEC, decimal or octal numbers are used for inspection. For a real variable, one whose name does not start with I, such as VPR, exponential numbers must be used. To examine a particular storage variable, type the abbreviation for the number system to be used and the storage location number of the variable (the number after the variable's name in the storage table printed by COMP). For example:

```
*↑VD 3
+0003\+0000,
*E17
+0017\+0.350000E-0002,
```

The Executive types the storage location number, a back slash, its present value, and a comma.

To change the present value in a location, simply type in the new value after the comma is printed. A carriage return typed after the new value closes this location, leaving it with the new value. A line feed after the new value closes the location with the new value and then types the contents of the next location. If the present value in a particular location is not to be changed, but the user wants to check the next storage location, a line feed will cause the next location and its contents to be printed out in the same format and same number system as above. The value of the new location can then be changed after the comma is printed.

NOTE

The FILE START parameters are located at D-1, D-2 and D-3; they reflect the respective base of FILE addresses for FILE #1, #2, and #3. These addresses refer to the disk page address beginning at 0 for disk 0. The locations may be called in their appropriate number system(s) in any order.

For exponential notation, a number may be entered with or without the power of 10 portion. For example:

```
*E21
+0021/+0.230000E-0002                    (line feed)
+0024/+0.105000E-0001,  .01035           (line feed)
+0027/+0.780000E-0003,  73E-5            (carriage return)
```

The first location number, 21, was called by typing E21 and no changes were required, so a line feed was typed to examine the next variable. Note that the next location number has been increased by a value of 3. This is because each exponential (floating-point) number requires 3 memory locations for storage. Location 24 required a change which was incorporated by typing in the new value in decimal point notation. The new number will internally be changed to exponential notation. A line feed now causes another three locations to be advanced (still using the exponential number system) and this time a variable was entered followed by a power of 10 exponent.

These variable changes are automatically performed while the system is running by starting the value change sequence with CTRL/V and then typing the appropriate number system abbreviation and storage location variable after the response of a period.

Clock mode allows the user to inspect the value of the system clock and to preset to the time of day. Typing ↑C will display the system clock in "HH: MM: SS," and open the clock for modification. The system clock is closed by), or preset by XX:YY:ZZ).

```
*↑VC
  CLOCK 0:  1:35, 12:15:00
```

After the system is started, the keyboard is locked and will respond to the CTRL/A command only if the system designer opens the console (command mode permitted). All of the possible CTRL commands are listed below:

CTRL/A	Attach this Teletype to command mode
CTRL/C	Return to Monitor (available only if not disabled by GENDAC)
CTRL/D	Initiate a "RUN" command
CTRL/V	Initiate a value change
CTRL/P	Return to operator mode
CTRL/U	Delete this line (start line over)

If the same program is to be used again, only SPUT need be called because the program has already been compiled. Follow the same procedure for calling SPUT as already specified. Note that this may not apply if FILE is used.

CHAPTER 7
MODIFYING THE SYSTEM

7.1 INTRODUCTION

The INDAC Compiler (COMP) is unique among compilers in that the source-level language to be processed has been defined in tables. While this feature alone is not exceptional, the fact that these compiler tables may be modified by the user is unique.

The tables are called System Communication Tables, and they define, to the different programs in the INDAC system, the location of code, the names of devices, the disk areas, and other system parameters.

It is the function of GENDAC to interact with these tables (under user direction) and to update certain disk images to incorporate additions or modifications to the INDAC system (see Figure 7-1). GENDAC is a conversational program that accepts one or more formatted paper tapes called library tapes and/or accepts one or more disk images developed by the user. The entire acceptance or rejection procedure is under operator control through the Teletype keyboard.

7.2 SYSTEM COMMUNICATION TABLES

These tables provide the basic intercommunication link between the various component programs in the INDAC System. They are updated by GENDAC to contain all data relevant to the use of library routines within the system. These tables are described in the following paragraphs.

7.2.1 Intrinsic Functions ⟨IF⟩

The ⟨IF⟩ table contains the vector locations of the intrinsic functions of the INDAC language (the Executive routines that resolve the GOTO statement, the + in arithmetic statements, the DO statement, etc). The ⟨IF⟩ table is a by-product of the Executive Assembly and contains addresses generated by the assembler. In the INDAC System, the user is permitted to update this table (using GENDAC) to include certain core-resident functions in the INDAC Executive. Highly repetitive functions, such as bit shifting, thumbwheel input conversion, and output digital formatting, can be made core-resident parts of the language to save the overhead time of continual disk accessing.

7.2.2 External Subroutines ⟨XS⟩

The ⟨XS⟩ table contains the alphabetic names and descriptions of all external (disk-resident) subroutines and arithmetic functions. The INDAC Compiler uses this table to resolve such source-language statements as:

```
LET Y = SQRT (X)
DO IRONC (INPUT, IOUT, REF)
```

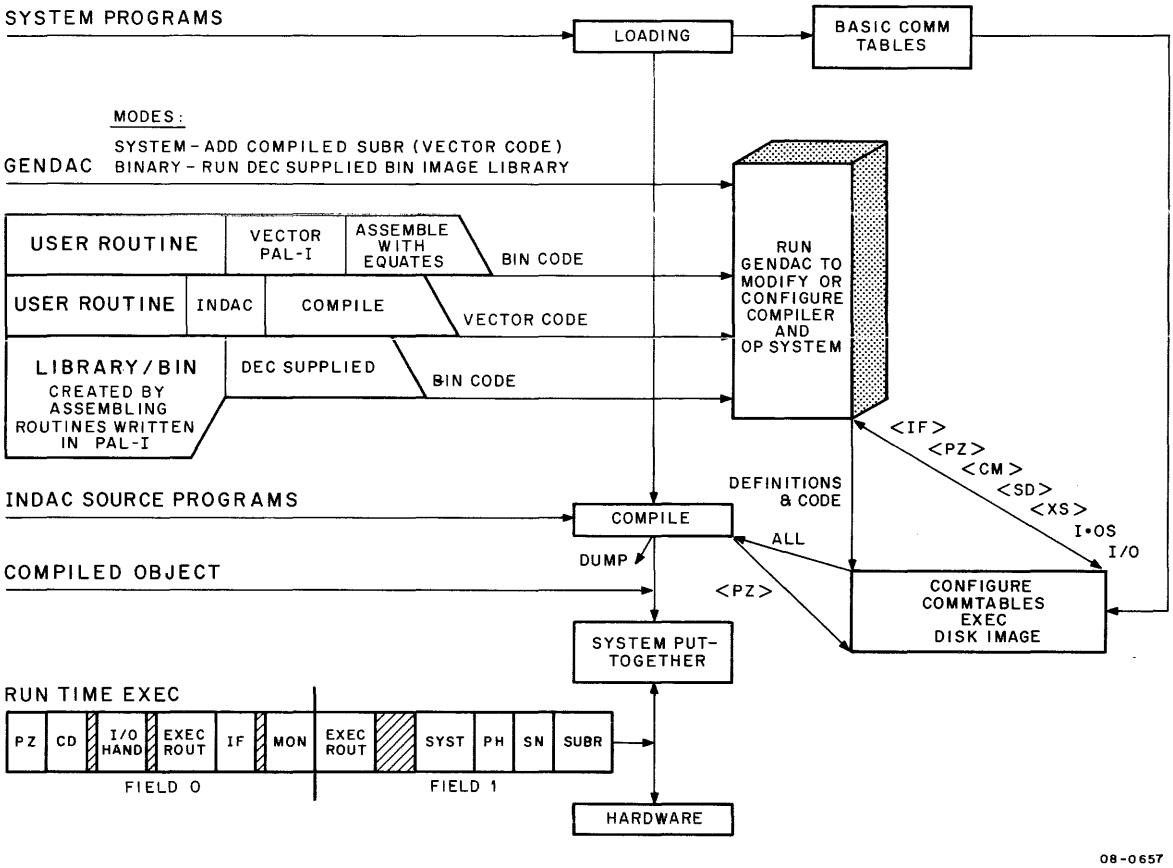


Figure 7-1 System Communication Tables — Interaction

7.2.3 System Devices ⟨SD⟩

The ⟨SD⟩ table contains the alphabetic names and definitions of all system hardware devices and pseudo-devices (see Appendix I). The INDAC Compiler uses this table to resolve all device requests such as the following:

```
GET (KEYS) IA
SEND (TTY, #101) ICLK
GET (CLOCK) ICLK
```

7.2.4 Core Map ⟨CM⟩

The ⟨CM⟩ table contains the core map of the Executive areas available for user insertions. This table is also generated through the assembly of the INDAC Executive. This table contains, for each page of core, the first address and the number of words available on that page.

7.2.5 Page Zero ⟨PZ⟩

The ⟨PZ⟩ table contains pointers, parameters, and general data for use by the INDAC Executive, the INDAC Compiler, and the Utility Support Programs. This table is an external table and is mentioned only to aid the explanation of how the INDAC System functions. The ⟨PZ⟩ table is one page in length and becomes Page 0 of Field 0 in the INDAC Executive during runtime. This page is updated and supplied by the Compiler as part of each compiled INDAC program. During the loading of the program, the page 0 table overlays the INDAC Executive page 0 image. For this reason, no direct modification of the disk image of Executive page 0 is possible.

7.2.6 Special GENDAC Table ⟨**⟩

GENDAC creates a special table ⟨**⟩ for its processing that contains tags, fixups, and entry points to support the "linking loader" function. This table is created for each separate run of GENDAC and exists for that run alone.

7.3 UPDATING THE SOFTWARE

The INDAC Compiler does not produce PDP-8 assembly-level code. The Compiler produces, instead, what we shall call "vector" code.

While most compilers producing an interpretive code point to a location in a table that, in turn, gives the location of the desired information in core; the INDAC Compiler's vector code points directly to the desired core location by using the ⟨IF⟩ table. Vector code is a highly compact form of information, allowing the INDAC Executive to receive its commands in an abbreviated form, and thereby reducing the time required to retrieve commands stored on disk and the time required to "interpret" normal interpretive code.

The INDAC Executive contains routines to process arithmetic and logical operations, routing, I/O, timing, and other system functions. The Compiler supplies the vector code to call these routines into action, based on the resolution of INDAC source-language statements. Two kinds of code appear in core while the Executive is operating:

a. Vector Code: The command-level code produced by the INDAC Compiler and used by the Executive to run the system.

b. PAL-I Assembly Code: The operating-level code performing the functions and commands specified by the vector code.

GENDAC can process both the vector code and the assembly-level code. Generally, vector code is a product of the INDAC Compiler and appears as a disk image; however, provision has been made to allow the user to produce his own vector code and to direct GENDAC to include this code in the system. The provision also exists for a user to create a new INDAC Executive function in the PAL-I code and to call this function into action through a new (user-created) vector command.

To more fully understand the vector code that the Compiler generates for source-language statements, code a simple INDAC program and run the Compiler under the List Option, as shown below:

```
.COMP
*OUT-S:TEST
*IN-S: TEST
*OPT-L
```

Paragraph 7.6, on Coding Details, contains the specifications and functional descriptions for each of the vector commands.

7.3.1 System Mode of Updating

The INDAC Compiler produces vector code as a normal part of its operation in compiling a program. Since GENDAC can process vector code, and since the Compiler can generate vector code, there ought to be some method of combining both capabilities so that code generated by the Compiler could become defined to the Compiler for future referencing. This would ensure that the next time the Compiler is run, the code previously generated can be referenced in source langauge.

The provision to define previously compiled code to the Compiler is known as the System Mode of Updating and can only be done when adding disk-resident subroutines. The procedures are summarized as follows.

Step	Procedure
1	Generate an INDAC program that consists of only one subroutine. This INDAC program is then compiled, using the INDAC Compiler.
2	Run GENDAC, using the "S" ("System") option.
3	Assign a particular name to the subroutine, give the current name of the compiled subroutine, and other pertinent information as requested by GENDAC via the Teletype keyboard.
4	GENDAC updates the appropriate system tables with the new name and moves the image of the subroutine to an area of the disk selected by a GENDAC algorithm.
5	Refer to the processed subroutine code with the INDAC statement: DO aaa where aaa is the name of the subroutine just processed.

7.3.2 Binary Mode of Updating

GENDAC operates in a second mode, termed the Binary Updating Mode. In this mode, GENDAC expects to find a formatted binary image produced by the PAL-I Assembler either as binary paper tape(s) or as binary disk image(s). In Binary Mode, the user can elect to manually simulate the INDAC Compiler output (vector code), develop complete assembly language routines, or produce some hybrid code, making use of the conveniences of both.

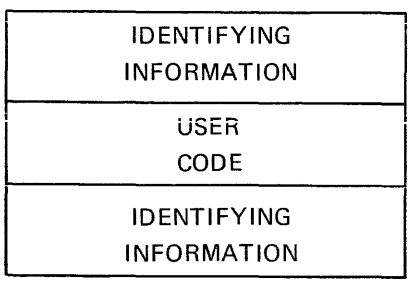
GENDAC requires information to update the System Communication Tables and disk images. One part of the information is the actual code for the library routine. Additional information is needed to define:

a. The function of that code (subroutine, device handler, function)

b. The area within the Executive that the code may exist

c. The information that the code is to be considered a disk-resident function or subroutine external to the INDAC Executive image.

STRUCTURE OF A BINARY IMAGE



Because GENDAC *sees* only an assembled binary image, a technique for dividing and identifying code and information is required.

GENDAC requires a specific set of origin declarations (*76XX, *7777) and FIELD declarations to separate and identify the library routines to be processed. Because these declarations are assembled by PAL-I as distinctive binary images, GENDAC is able to interpret them as declarations to separate and identify information and code. These declarations, therefore, are not to be considered origins for code or data, they are merely identification codes. (See Figure 7-2.)

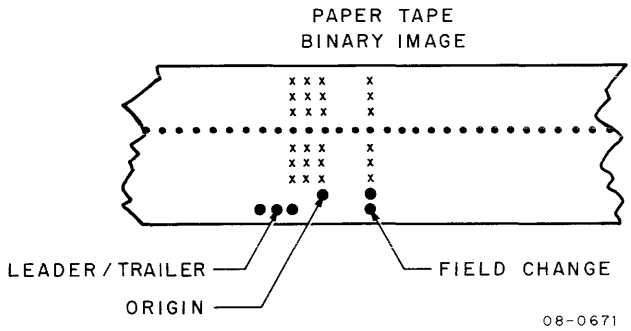


Figure 7-2 Paper Tape Binary Image

7.4 LIBRARY STRUCTURE

GENDAC Library Tapes may contain subroutines, functions, and I/O device handlers; the tapes may be in the form of punched paper tape or disk binary files. Library tapes consist of blocks of code referred to as "modules."

7.4.1 Basic Module

A module is a disk-resident or core-resident code that is assembled and located as a single unit. To identify these modules to GENDAC, the "type" of module (core or disk, function, subroutine, or device handler, etc.) is specified and linkage information and other pertinent system data is provided. The following PAL-I Assembler pseudo-ops are used to provide information regarding modules to GENDAC:

*76XX *7777 FIELD X

Because this area in core is prohibited (Monitor Bootstrap), these origins are considered by GENDAC to be descriptive information. The following origin numbers have English word equivalents (Equates) that are used in coding the modules:

7600 = GROUP
7610 = OLDMOD
7620 = NEWMOD
7630 = CALMOD
7640 = ENTRY
7650 = CODE
7660 = FIXUP
7777 = ENDGRP

Disk-resident modules are used for subroutines and functions and must be one page of code or less to be loaded by GENDAC. During Executive run-time, disk-resident modules are called into core when needed for execution and are placed into one of two (foreground/background) system buffers. Each such module contains one or more entry points defined in the ⟨XS⟩ table, for use by the compiler at compile time.

Core-resident modules are used for functions and device handlers. Each relocatable, core-resident module must be one page of code or less if coded in PDP-8 machine code, or two pages or less if coded in INDAC vector code (all vector code must reside in Field 1).

7.4.2 Extension Modules

Each function or device handler can be constructed from several modules, if necessary. This set of functional modules is considered a "functional group". In each case, one module contains the basic entry point(s) to the routine as named in the ⟨SD⟩ or ⟨XS⟩ system table, while the remaining modules are "extension" types. The module containing the entry point must be loaded by GENDAC into the appropriate part of Field 0 according to its purpose, that is, whether it is a function or device handler.

Any modules loaded, in addition to the initial module, as part of a core-resident function or device handler must be loaded by GENDAC and may contain an "extension entry point," named in the GENDAC ⟨**⟩ table, in order that the modules may reference each other. These modules need not be loaded into prespecified core locations. The user can specify the absolute location of any core-resident module or can allow GENDAC to allocate any relocatable code according to its function and the current ⟨CM⟩ table.

The modules of an inter-related group are bracketed on a library tape between a *7600 and a *7777 code; the entire group can be omitted (optionally) from any given GENDAC run. The *7600 is followed by a user-formulated text comment, which GENDAC types as the tape is read. The operator determines whether the group is to be loaded or bypassed. For example, consider the following code:

```
*7600
TEXT 'ADC'
```

When the above code is encountered on a library tape, GENDAC recognizes that it has encountered another library routine (*7600). GENDAC then types:

```
ADC?-
```

At operating time, the operator replies yes or no, depending on whether the routine is desired.

7.4.3 Call-Up Modules

If an extension module is common to each of several library routines, and the user desires to have GENDAC load the module only if one or more of the routines requiring it has been selected for loading, then such an extension module is defined as a "call-up" module. A definition of its allocation requirements, and a specially flagged entry in ⟨**⟩ must then precede all modules that may reference it. The actual body of code (together with any necessary global definitions) must follow all referencing modules.

7.4.4 Additional Notes

To insert an entry into a system table (either ⟨SD⟩ or ⟨IF⟩) pointing to code already present in the INDAC Executive, but omitted from the table, a pre-existing module may be declared with a *761x to avoid having GENDAC load the code again.

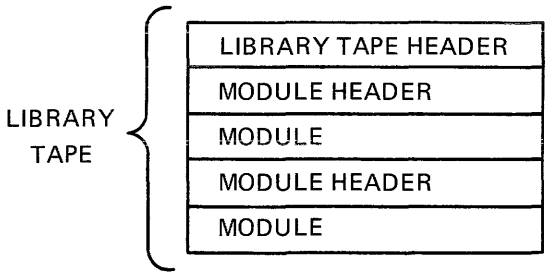
The special entry declaration *7642 is used to extend the Executive I/O interrupt skip-chain. It may be used only once in a module (any Field 0 module) to point to the first device skip IOT in the module. The last device skip IOT, of one or more in the module, must be followed by a jump indirect through the first word of the module. GENDAC inserts a connection address into the first word of an I/O interrupt module, joining that module into the interrupt skip-chain. The last module in the group points to a system error routine; thus, if no module responds to the interrupt, the interrupt chain defaults to the system error routine.

(This page intentionally left blank.)

```
 //EQUATES FOR GENDAC
GROUP=7600
OLDMOD=7610
NEWMOD=7620
CALMOD=7630
ENTRY=7640
CODE=7650
FIXUP=7660
ENDGRP=7777

       TEXT 'SPECIAL DEVICE HANDLER'

  //FUNCTIONAL GROUP DECLARATION
*GROUP
       TEXT 'MY DEVICE'
       TEXT 'SPECIAL DEVICE 1013 MOD 2'

  //PHYSICAL DESCRIPTION OF MODULE
*NEWMOD 4       /REQUEST FOR I-O MODULE ALLOCATION
        *275    /ORIGIN WITHIN PAGE (DETERMINED FROM <CM> TABLE)
        LAST-MYDEV+1      /SET UP LENGTH OF MODULE

  //LOGICAL DESCRIPTION OF MODULE
        1       /NUMBER OF ENTRY POINTS WITHIN MODULE
*ENTRY  3       /DEFINING AN I-O HANDLER ENTRY POINT
        TEXT 'MYDEVICE'   /NAME THAT COMPILER WILL RECOGNIZE
        0       /RELATIVE ENTRY POINT WITHIN MODULE
        2000    /BIT PATTERN FOR "GET" DEVICE
        6361    /IOT CODE FOR DEVICE
        0       /NO CONTROL WORDS

  //ACTUAL CODE OF MODULE
*CODE
        TEXT '[MY]'

        *275
MYDEV,  CALL
        SETUP               /INITIALIZE THE DATA ROUTINE
DEV2,   CALL
        DATA                /GET THE ADDRESS OF THE DATA ITEM
        JMP     NOMOR       /END OF DATA LIST
        SKP                 /INTEGER RETURN
        SYSOUT              /REQUEST FOR REAL VARIABLE-ERROR
        DCA     ADDR        /HOLD ADDRESS
        6362                /GET DATA FROM DEVICE
        DCA 1   ADDR        /STORE DATA-NOTE DATA FIELD SET TO ONE ON ENTRY
        JMP     DEV2        /GET NEXT ADDRESS
NOMOR,  CALL
LAST,   DEVRET              /RETURN FROM DEVICE HANDLER

  //FUNCTIONAL GROUP TERMINATION
*ENDGRP                     /END OF MODULE GROUP FOR MYDEVICE

*
```

7.5 LIBRARY TAPE FORMAT

The GENDAC Library Tape Format is graphically represented by the following diagram. A typical library tape comprises the elements indicated.



7.5.1 Definitions

The library tape is coded with PAL-I origin definitions of 76xx for communication purposes. Because this area is prohibited (Monitor Bootstrap), these origins are considered by GENDAC to be descriptive information. The equate statements listed in the Teletype example provide an English word equivalent that is more descriptive than the origin number. Refer to the coding sample shown on the foldout in the subsequent discussion.

```
// EQUATES FOR GENDAC

GROUP=7600
OLDMOD=7610
NEWMOD=7620
CALMOD=7630
ENTRY=7640
CODE=7650
FIXUP=7660
ENDGRP=7777

          TEXT 'SPECIAL DEVICE HANDLER'

//FUNCTIONAL GROUP DECLARATION
*GROUP
```

A PAL-I coded line such as:

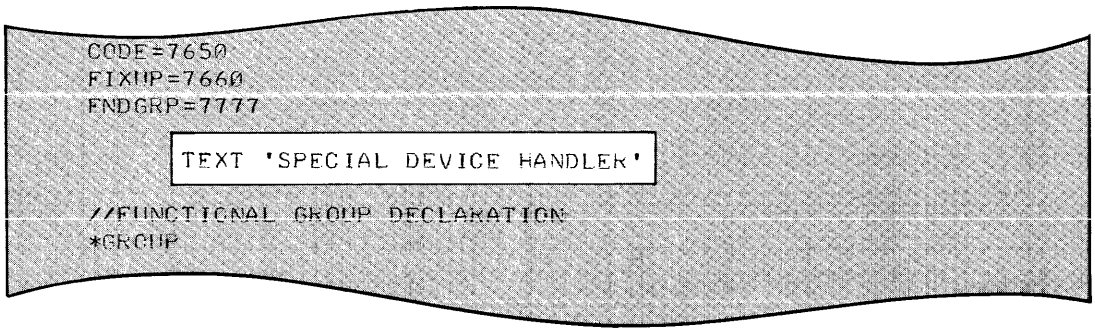
```
*ENTRY 3
```

results in *7643 as an assembler output.

7.5.2 Library Tape Header

The first element of the GENDAC Library Tape Format is the title for the entire library tape. With the PAL-I pseudo-op TEXT, the user creates a title for the tape as follows:

```
TEXT 'TITLE OF TAPE'        or 0
                            if no title is desired.
```



7.5.3 Module Header

The header data for modules is dealt with in subsequent paragraphs and consists of the following:

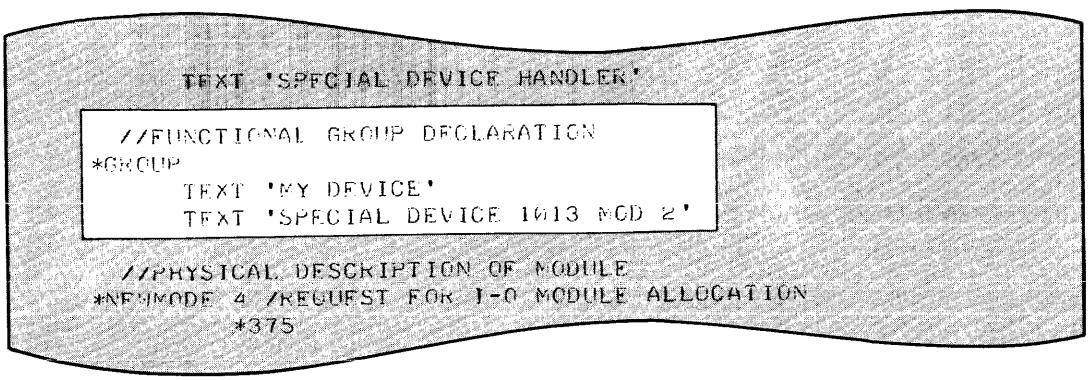
Functional Group Declaration . . . . . Defines a group of related modules (basic entry, extension, and call-up, which can be selected or bypassed).
*GROUP

Module Declaration

Physical Description . . . . . Declares the type of module and storage allocation requirements.
*OLDMOD Y
*NEWMOD Y
*CALMOD 2 or 3

Logical Description . . . . . Declares the purpose of the module, and consequently what system communication tables are to be updated.
*ENTRY

7.5.3.1 Functional Group Declaration – This data defines a group of modules, all of which may be either selected for loading at GENDAC run-time or bypassed at the user's discretion. A functional group begins with a *GROUP code and contains a description of the module to be printed at run-time. For example:



The *GROUP code indicates to GENDAC that it has come to a new group. When the tape is read into the computer by GENDAC, the following is typed on the Teletype:

MY DEVICE?-

At this point, the user indicates whether or not the group is to be included in the system at this time. If the operator requests additional information about the group (by typing "I"), the Teletype output at this point appears as follows:

MY DEVICE ? - I
SPECIAL DEVICE 1013 MOD 2?

Now the user must answer either "Y" if the group is to be included or "N" if it is to be bypassed.

There may be more than one line of text to be printed out as a description of the group. This is done by supplying the material on subsequent lines with a TEXT pseudo-op for each line. All such descriptive material is printed when and if "I" is typed by the operator.

7.5.3.2 Module Declaration — This data consists of two units of information:

a. A physical description of the module, defining the allocation requirements.

b. A logical description of entry point(s) within the module. There may be one or more such logical descriptions.

7.5.3.2.1 Physical Descriptions of Module — The first information to be placed after the Functional Group Declaration is one of the following module definition codes.

Code	Definition
*OLDMOD Y	The module is already core-resident, and Y cannot be 0.
*NEWMOD Y	The module is being supplied for addition to the INDAC System. The module may be a basic entry module or an extension module.
*CALMOD Y	A call-up module (where Y=2 or 3) is being defined at this point. This call-up module is some form of common routine and is to be loaded only if a request is subsequently made for this module.

The "Y" in the module definition codes above defines the allocation requirements of the module and is described in Table 7-1.

The allocation code is followed by a set of parameters that depend on the chosen value of Y. Table 7-2 lists the various parameter codes to be used. The user should insert the codes in the order listed, depending on whether the type of module being handled belongs to the module type listed (the value in the allocation code). If the code is required, the correct sequence must be observed.

Table 7-1
Allocation Codes

Y Code	Resident Area	Relocatability	Additional Information
0	Disk	–	External subroutine or function
1	Core	Nonrelocatable	User specifies absolute origin
2	Core	By page	User specifies page origin
3	Core	By word	–
4	Core	By page	I/O handler only
5	–	–	Illegal
6	Core	By page	Function
7	Core	By word	Function

Table 7-2
Allocation Parameters

Module Types	Actual Code Used	Reason for Code
1,2,3	FIELD X	Where x may be 0 or 1. FIELD 1 is required for any module written in vector code in an 8K system. FIELD 0 is required for any module containing a *7642 interrupt-chain extension.
1	*XXXX	Where XXXX is the absolute core origin of the module. (XXXX may not be in page 0 or of the form 76XX or 77XX.)
2,4,6	*0XXX	Where *0XXX is the origin within the page. All page-relocatable code must be assembled in page 200 – 377.
3,7	–	All word-relocatable code must be assembled in pages 200 – 577, and *200 should begin the body of code.
All cases except 0	XXXX	Where XXXX is the size of the module in words. There is a maximum of one page for absolute, call-up, and page-relocatable modules; a maximum of two pages for word-relocatable modules (except call-up); exactly one page is assumed for disk-resident modules.

In the following example, the physical description of the module, the origin within the page is arbitrarily selected as *275. In practice, the ESUP (Executive Support) program would be run against the current core map ⟨CM⟩ and an address chosen based on the available core.

```
        TEXT 'MY DEVICE'
        TEXT 'SPECIAL DEVICE 1013 MOD 2'

 //PHYSICAL DESCRIPTION OF MODULE
*NEWMOD 4       /REQUEST FOR I-O MODULE ALLOCATION
        *275    /ORIGIN WITHIN PAGE (DETERMINED FROM <CM> TABLE)
        LAST-MYDEV+1      /SET UP LENGTH OF MODULE
```

NOTE

The size of the module does not have to be counted. Because this code will be processed by the PAL-I Assembler, it is sufficient to generate an expression for the length of code and let the assembler generate the actual number.

7.5.3.2.2 Logical Description of Module – The next item in the module declaration is the collection of codes that specify the purpose of the module, and correspondingly specify what system tables must be updated to refer to points within this module.

Table 7-3 is a summary of the table update information, similar to Table 7-2. Each module to be considered uses all or part of this information for each entry point. If used, it must be in the sequence indicated.

Table 7-3
Table Update Information

Code	Significance
XXXX	XXXX is the number of entry points to be defined within this module, or 0 if none are defined. Each entry point definition is preceded by a *ENTRY X code.
*ENTRY X	Where X defines the system table(s) to be updated, and the nature of the table entry. X is one of the following codes: 0 Extension module or Extension Entry point names in ⟨**⟩ for use in global definitions only. 1 Call-up module, used if this is a *CALMOD X call-up module. Limit is one per module, also uses ⟨**⟩. 2 Device handler interrupt-chain extension. Uses no system table, word 0 of module code is filled in by GENDAC. 3 Device handler, called by source program, uses ⟨SD⟩. 4 Core-resident function, uses ⟨IF⟩. 5 Disk-resident function, uses ⟨XS⟩. 6 Disk-resident subroutine, uses ⟨XS⟩.

TEXT 'NAME OF ENTRY POINT'

For ENTRY 3, 4, 5 and 6, TEXT "name" specifies the name to be used in the INDAC source program.

0XXX — Where 0XXX is the location of the entry point relative to the base of the module. (If the module is of the type *ENTRY 2, then 0XXX points to the first IOT of the skip-chain segment in the module.)

When the module is of Type *ENTRY 3, a device handler called by the source program, the following additional data is required:

XXXX — Where XXXX is the bit pattern defining the device usage to the INDAC Compiler, bits may be ORed to give a combination of the following meanings:

4000	Allocated buffer required
2000	"GET" device
1000	"SEND" device
0400	Multiplexed device with channel assignments
0200	May reference a FORMAT statement

6XX1 — The device code.

00XX — Where XX is the number of device control words to follow, or 0 if there are no control words:

TEXT 'control word'

Where the control word is the text to appear in a control line statement, for example: X1000, GAIN 2, etc.

XXXX — Where XXX is the octal value for the control word just defined. (The TEXT and XXXX codes are repeated until the number of control words specified above are completed.)

The logical description of the sample I/O handler follows:

```
  //LOGICAL DESCRIPTION OF MODULE
          1       /NUMBER OF ENTRY POINTS WITHIN MODULE
*ENTRY    3       /DEFINING AN I-O HANDLER ENTRY POINT
          TEXT 'MYDEVICE'  /NAME THAT COMPILER WILL RECOGNIZE
          0       /RELATIVE ENTRY POINT WITHIN MODULE
          2000    /BIT PATTERN FOR "GET" DEVICE
          6361    /IOT CODE FOR DEVICE
          0       /NO CONTROL WORDS

  //ACTUAL CODE OF MODULE
*CODE
          TEXT '[MY]'
```

7.5.3.3 Module Body – When the header information required by GENDAC to define the module is complete, the code of the module itself can be included.

7.5.3.3.1 Module Title –

*CODE X	Where x has the following meaning:
0	code of standard (*NEWMOD X) module follows
1	code of call-up (*CALMOD X) module follows
TEXT 'module name'	Gives the name of the module. If a call-up module has been referenced by a *FIXUP X prior to this point, the code for the module is loaded, if the module has not been referenced, GENDAC skips to the next *ENDGRP separator and continues from there.

Details on preparing the code for a module are given in Chapter 3.

7.5.3.3.2 Fixup Declarations – GENDAC allows the use of relocatable code for modules. Such code cannot be completely defined at assembly time, because the final location of the code is unknown. In this sense, GENDAC operates as a linking loader, providing links within and between modules.

The basic link mechanism is the "FIXUP". A FIXUP is a declaration made before the actual body of code as follows:

*FIXUP X	Where X is the index to the following FIXUP definition TEXT 'AAAA'
TEXT 'AAAA'	Where AAAA is the name of an entry point of this or a previously loaded module and may take the following forms:

AAAA	Absolute reference
AAAA-.	Displacement reference
-AAAA	Negative absolute reference
-AAAA-.	Negative displacement reference
0-.	Null displacement FIXUP definition

Repeat this declaration for each FIXUP needed in this module.

GENDAC saves the text declaration stated and assigns a number to the saved text according to *FIXUP X.

This saved text, with its assigned number (X), is now a FIXUP. FIXUPs are called for within the actual code of the module as follows:

```
FIELD X
word
```

At the Field call, GENDAC uses the assigned FIELD as an index to retrieve the saved FIXUP text. The saved text is evaluated at the location in core, where the word following the FIELD call will be placed. The evaluated text is arithmetically added to the word following the FIELD call (normally the following word is zero). In most cases, the absolute reference format is used. The special cases are provided for generating relocatable, vector displacement code.

FIXUP declarations are limited to 8 (0–7) for each single module. A new module can use the same FIXUP index numbers again, because all saved text is deleted at the end of each module. The actual code of the sample I/O handler follows:

```
        0       /NO CONTROL WORDS
  //ACTUAL CODE OF MODULE
*CODE

        TEXT '[MY]'

        *275
MYDEV,  CALL
        SETUP                   /INITIALIZE THE DATA ROUTINE
DEV2,   CALL
        DATA                    /GET THE ADDRESS OF THE DATA ITEM
        JMP     NOMOR           /END OF DATA LIST
        SKP                     /INTEGER RETURN
        SYSOUT                  /REQUEST FOR REAL VARIABLE-ERROR
        DCA     ADDR            /HOLD ADDRESS
        6362                    /GET DATA FROM DEVICE
        DCA 1   ADDR            /STORE DATA-NOTE DATA FIELD SET TO
                                /ONE ON ENTRY
        JMP     DEV2            /GET NEXT ADDRESS
NOMOR,  CALL
LAST,   DEVRET                  /RETURN FROM DEVICE HANDLER
  //FUNCTIONAL GROUP
```

7.5.3.4 Group and Module Termination – When the module is complete, the separating *ENDGRP code (*7777) should be used in the following situations:

a. To terminate function groups of modules, and

b. To terminate a call-up module.

```
LAST,   DEVRET
  //FUNCTIONAL GROUP TERMINATION
*ENDGRP                         /END OF MODULE GROUP FOR MYDEVICE
```

7.6 CODING DETAILS

To illustrate the coding, application, and differences between subroutines and functions, a common industrial processing routine was selected and implemented both as a subroutine and as a function. The routine is a bit-shifting operation fixed at 4 bits (BCD) to the right. This limitation is imposed because a function can only

accept a single argument; the subroutine can accept several arguments and could be made more general. The structure of a library module is described in Paragraph 7.4.

7.6.1 Function (Core-Resident in Field 0)

The function is called by the language statement:

LET Ia = BCDSHF (e)

where BCDSHF may be isolated as a call or may be imbedded in an expression itself. The purpose of BCDSHF is to take the last value on the arithmetic stack, perform the shifting, and replace the value on the stack. In this sense it functions as a unary operator.

A core-resident function is entered in binary mode by the Executive. When the function completes its assignment, the function must return to the Executive. The actual code to perform the shifting is as follows:

```
TAD I STACK     /get the value on the stack
CLL RTR
RTR             /perform the shifting
DCA I STACK     /replace the value on the stack
VECTOR          /return to the user program in Vector mode
```

The ESUP program provides a paper tape of PAL-I Equate Statements to resolve the STACK and VECTOR terms. This tape must be loaded with the function module so that the terms can be resolved during assembly. The Executive enters the function routine with the data field set to Field 1. The arithmetic stack is located in Field 1; therefore, the indirect instructions reference the correct data. If any data is to be referenced, indirectly, in Field 0, then the field must be changed and later reset to 1 before exiting from the routine. The complete routine embedded in GENDAC descriptive information follows:

```
  TEXT 'SPECIAL FUNCTION'
*GROUP
  TEXT 'BCDSHF'
  TEXT 'BCD RIGHT-SHIFT FUNCTION (C-R)'
*NEWMOD 7          /FUNCTION-WORD RELOC
        5          /SIZE OF MODULE
        1          /ONE ENTRY POINT
*ENTRY  4          /FUNCTION, C-R
  TEXT 'BCDSHF'    /NAME COMPILER WILL RECOGNIZE
  0                /ENTRY POINT WITHIN MODULE
*CODE
  *200             /WORD-RELOC ORIGIN
  TAD I STACK      /GET VALUE FROM STACK
  CLL RTR
  RTR              /PERFORM BCD SHIFT
  DCA I STACK      /REPLACE VALUE ON STACK
  VECTOR           /RETURN TO VECTOR MODE
*ENDGRP
```

NOTE

This code is word-relocatable (unusual for PAL-level code), because all instructions are either page 0 references or nonmemory instructions.

7.6.2 Subroutine

The subroutine is called by the language statement:

```
DO BCDSHF (IX, IY)
```

where IX is the input item, and IY is the output item.

7.6.2.1 Subroutine Requirements and Analysis – The following is a list of subroutine requirements:

a. The first two words of the subroutine must be 0s. These words are used by the Executive to control the dynamic allocation scheme.

b. A subroutine cannot exceed one page.

c. A subroutine is entered in VECTOR mode and, therefore, must be located in FIELD 1. The subroutine is disk-resident and is automatically called into FIELD 1 by the Executive.

d. A function deals with information supplied by the arithmetic stack. A subroutine deals with data (generally located in FIELD 1); therefore, a subroutine must include some instructions to retrieve and update this data.

For this reason, a subroutine usually contains some initial Vector code to retrieve the data specified as the arguments of the subroutine call. There is also generally some final Vector code that updates the output data items of the call. As described in Paragraph 7.6.1, the ESUP paper tape provides the equate statements required to resolve the Vector code and must be loaded with the subroutine module for assembly.

7.6.2.2 Sample of Subroutine Coding

The following is an example of subroutine coding.

```
        /SAMPLE SUBROUTINE
        /DO BCDSHF(IX, IY)

//EQUATES FOR GENDAC
GROUP=7600
OLDMOD=7610
NEWMOD=7620
ENTRY=7640
CODE=7650
FIXUP=7660
ENDGRP=7777
        TEXT 'BCDSHF SUBROUTINE TAPE'

*GROUP  TEXT 'BCDSHF'
        TEXT 'BCD SHIFT SUBROUTINE'

*NEWMOD         /NEW-MODULE,DISK-RESIDENT

        1       /ONE ENTRY POINT
*ENTRY 6        /DISK-RESIDENT SUBROUTINE
        TEXT 'BCDSHF'
        2       /MODULE ENTRY POINT (MUST HAVE TWO WORDS OF ZEROS)
*CODE   TEXT 'BCDSHF'

        *200
        0           /TWO LEADING WORDS OF ZEROS REQUIRED
        0           /FOR ALL EXTERNAL SUBROUTINES OR FUNCTIONS
        ILE         /GET THE VALUE FOR THE FIRST ARGUMENT OF THE
        1           /CALL ON THE ARITHMETIC STACK
        IS
        ALPHA-.     /STORE THE VALUE INTERNAL TO THE SUBROUTINE
        GBIN        /ENTER BINARY MODE
        TAD  ALPHA
        CLL RTR
        RTR
        DCA ALPHA /PERFORM THE BDC SHIFT
        INTERP      /RETURN TO THE VECTOR MODE
        IL          /LOAD THE ADJUSTED VALUE ON THE
        ALPHA-.     /ARITHMETIC STACK
        ISE         /STORE THE VALUE ON THE STACK INTO
        2           /THE SECOND ARGUMENT OF THE CALL
        RETE        /RETURN FROM THE SUBROUTINE, BYPASSING
        2           /THE 2 ARGUMENTS OF THE CALL
ALPHA,  0

*ENDGRP
        $
```

7.6.3 I/O Handler

7.6.3.1 I/O Handler Operation – The compiler generates the following vector code for an I/O request:

```
5XXX          where XXX is an index to the handler
)control driver pointer
)format driver pointer
indicator     4000 – GET; 0000 – SEND
nr. data list words
  ...
  data list
  ...
```

7.6.3.1.1 The Control Driver Table – The control driver table contains the information supplied through the .EQUIPMENT statement device-control line, and the starting channel of each item in the data list. The Executive contains subroutines available to the user for extracting information from this table automatically.

7.6.3.1.2 The Format Driver Table – The format driver table contains information supplied through a .FORMAT or .HEADER statement. The processing of this table is reserved for the Teletypes and high-speed punch. This table is not available to the user.

7.6.3.1.3 Data Information and Calls – The remainder of the I/O call is processed by Executive subroutines through the "CALL" statements as shown in Table 7-4.

NOTE
The AC must be cleared before issuing a CALL.

Table 7-4
CALL Statements

ESUP CALL	Function
CALL SETUP	The first CALL in an I/O handler provides the initial setup and linkage to the I/O tables. Returns with the LINK set to 1 for a GET request and 0 for a SEND request. AC set to 0.
CALL ALOCAT	Returns with the base address of a page of FIELD 1 core. Used to obtain a page of buffer for dynamically allocated core. Not generally used.
CALL DATA a b c	Used to obtain addresses from the data list. Returns with the address of the data item in the AC at: *a.* no more data address *b.* request for integer variable *c.* request for real variable If the data item specified is an array, then consecutive DATA requests return with the consecutive addresses of the array requested, until the array is expired. The DATA routine then looks for additional data list items, until the list is expired.
CALL IOSUB a b	Request for the subroutine (if any) specified in the device control line to operate on the data supplied. A return is made at: *a.* subroutine did execute *b.* no subroutine requested
CALL CHANEL	Returns with the channel number in the AC. This is the channel associated with the last address supplied by the DATA routine.
CALL DEVRET	The I/O handler is finished. This is a return to the Executive in vector mode to continue processing the compiled program.

7.6.3.1.4 Executive Page Zero Parameters – Table 7-5 is a list of executive parameters.

Table 7-5
Executive Parameters

Field Location	ESUP Name	Use
0	ADDR	Loaded with the address supplied by DATA for use by the subroutine called through IOSUB.
0	VALUE	Loaded by the I/O handler with information derived from, or being sent to the I/O device. The processing subroutine (if any) loads VALUE from the data list (after processing) or takes VALUE, processes it, and stores it into the data list. The method of operation depends on whether the device is an input or output peripheral.
0	IOCTRL	After SETUP, this parameter contains the bit configuration, specified by the device control line, required to properly service the device (gain control, resolution, initializing requests, etc.).
0	STATUS	This parameter is available for I/O devices to communicate with language-level code. The parameter is retrievable by GET (STATUS) IX where IX contains whatever value STATUS was set to.

7.6.3.1.5 Interrupting Devices – A complete cross-reference listing of devices supported by DEC is included in the GENDAC library. For an analysis of interrupt processing, see the ADC handler included in the program listing (tag "ADCI" to "AFX = .").

7.6.3.2 A Typical I/O Handler – A typical I/O handler (DIGOUT) is presented below, showing the use of the I/O calls and parameters:

```
                    //DIGITAL OUTPUT DEVICE HANDLER-SYSTEM LEVEL
      0350          *350
0350  4465  DIGO,   CALL                /LINK BIT INDICATES DIRECTION
0351  0001          SETUP               / OF TRANSFER:"0"-OUTPUT;"1"-INPUT
0352  4465  DIGO2,  CALL
0353  0002          DATA                /GET ADDRESS AND TYPE OF ITEM
0354  5374          JMP      DIGO4      /NO MORE DATA ITEMS
0355  7410          SKP
0356  4510          SYSOUT              /ERROR-REAL VARIABLE REQUEST
0357  3117          DCA      ADDR       /SAVE ADDR FOR SUBROUTINE CALL
0360  4465          CALL
0361  0004          IOSUB               /CALL SUBROUTINE-IF ANY REQUESTED
0362  5365          JMP      DIGO3      /SUBROUTINE CALLED-AND EXECUTED
0363  1517          TAD I    ADDR       /SUBROUTINE DID NOT EXECUTE,
0364  3461          DCA I    VALUE      / HANDLER MUST STORE THE VALUE
0365  4465  DIGO3,  CALL
0366  0003          CHANEL              /GET CHANNEL ASSIGNED TO ITEM
0367  6366          6366                /SELECT CHANNEL
0370  7300          CLA CLL
0371  1461          TAD I    VALUE
0372  6365          6365                /SEND VALUE TO DIGITAL OUTPUT
                /NOTE: THIS DEVICE CLEARS THE AC AFTER SENDING THE ITEM;
                / THE AC MUST BE CLEAR BEFORE ISSUING A "CALL"
0373  5352          JMP      DIGO2      /GET NEXT DATA ITEM

3374  4465  DIGO4,  CALL
0375  0005          DEVRET              /DEVICE RETURN EXIT.
```

7.7 VECTOR CODE

All vectors are processed by the vector interpreter of the Executive and result in the execution of some intrinsic function or library routine of the Executive. The user, coding in assembly-level code, can make use of these routines by entering the vector mode and using these routines himself, perhaps in some combination not available through the Compiler. The following paragraphs describe the characteristics of vector code.

7.7.1 Numeric Formats

The internal format of numeric data is shown in Figure 7-3.

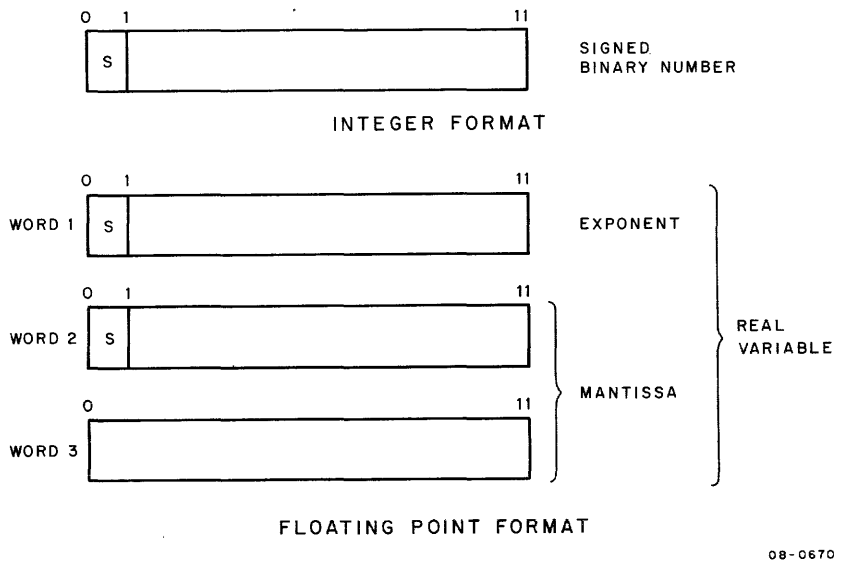


Figure 7-3 Internal Format of Numeric Data

Integer numbers can be positive or negative (stored as 2's complement) according to the sign (bit 0). Floating-point numbers, called "real numbers", can be positive or negative according to their sign and are each assigned three words of storage. Floating-point numbers are carried in normalized form. The exponent is a signed 2's complement quantity in one 12-bit word. The signed mantissa is stored in two 12-bit words, maintaining 23 bits of significance, making a total of three words of storage.

Arithmetic expressions are evaluated using a push-down arithmetic stack that grows in ascending locations (in FIELD 1 of an 8K system) with the current stack pointer (STK), pointing to word 3 of a real number or to the integer number.

7.7.2 Vector Code Example

An example of the vector code produced by the compiler to resolve the INDAC statement follows:

```
LET IA = IB + IC -ID
        IL
        )IB
        IL
        )IC
        ADD
        IL
        )ID
        SUB
        IS
        )IA
```

The term ")IB" means the displacement address from the point at which)IB occurs to the actual location of IB. In the PAL-I assembly, this term would be written as "IB-.". The code as written above means the following:

1. Load the integer, the displacement address of which follows, [)IB], onto the arithmetic stack.
2. Load the integer, the displacement address of which follows, [)IC], onto the arithmetic stack.
3. Take the last two values on the arithmetic stack and perform an integer addition. Replace these two values with the new sum (a single value).
4. Load the integer, the displacement address of which follows [)ID], onto the arithmetic stack.
5. Take the last two values on the arithmetic stack (the result of adding IB and IC, and the value of ID) and perform an integer subtraction. Replace these two values with the new value.
6. Take the integer value on the arithmetic stack and store it in the location the displacement address of which follows [)IA]. This last action collapses the arithmetic stack.

GENDAC provides for displacement fixups to allow vector code modules to interact. Displacement fixups may not be used in disk-resident modules, because a disk-resident module cannot reference another module. For the same reason, fixups cannot refer to a disk-resident module.

7.7.3 Load and Store Instructions

Load and store instructions are divided into categories:

a. Single-word operations (1-word)

b. "Real" variable operations (3-word)

Single-word operations are considered integer types. Three-word operations are considered floating types. Operations for integer and floating-point instructions take the same argument list, and deal with the arithmetic stack in the same fashion, with the single exception of the number of words handled.

7.7.3.1 Load a Simple Variable —

Form:	IL	FL
	)IB	)ALPHA
Function:	Load the simple variable, whose displacement address follows, on the arithmetic stack.	

7.7.3.2 Store a Simple Variable –

Form:

IS	FS
)IB	)ALPHA

Function: Store the last value on the arithmetic stack into the variable whose displacement address follows.

7.7.3.3 Load An Array Element (Indexed Variable) –

Form:

ILI	FLI
)IX	)IX
)IB	)ALPHA

Function: Load the arithmetic stack with the variable whose index value (displacement address) and base address (displacement) follow.

7.7.3.4 Store An Array Element –

Form:

ISI	FSI
)IX	)IX
)IB	)ALPHA

Function: Store the last value on the arithmetic stack into the variable whose index value (displacement) and base address (displacement) follow.

7.7.3.5 Load An External Argument –

Form:

ILE	FLE
k	k

Function: This vector is used within an external subroutine to load the arithmetic stack with the kth argument supplied by the subroutine call. Arguments range from 1 to n.

7.7.3.6 Store Into An External Argument –

Form:

ISE	FSE
k	k

Function: Store the last value on the arithmetic stack into the kth argument supplied by the subroutine call.

7.7.3.7 Load An External Argument, Indexed –

Form:

ILEI	FLEI
)IX	)IX
k	k

Function: Load the arithmetic stack with the kth argument supplied by the subroutine call, indexed by the internal subroutine variable whose address (displacement) is supplied.

7.7.3.8 Store Into An External Argument, Indexed –

Form:	ISEI	FSEI
	)IX	)IX
	k	k

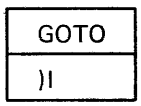
Function: Store the last value on the arithmetic stack into the kth argument supplied by the subroutine call, indexed by the internal subroutine variable whose address (displacement) is supplied.

7.7.4 GOTO Statements

7.7.4.1 Unconditional GOTO –

GOTO I

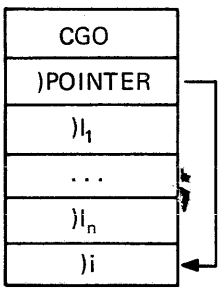
Compiles into:



7.7.4.2 Computed GOTO –

GOTO $(I_1, \ldots, I_n)$, i

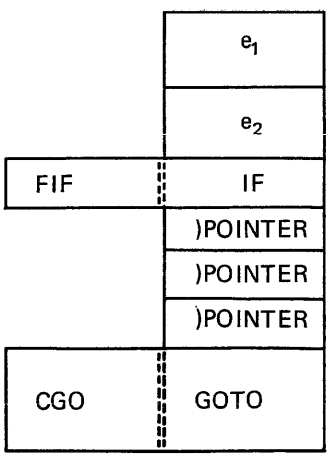
Compiles into:



7.7.5 IF Statement

IF (e_1) o (e_2) then S

compiles into:



evaluate expression e_1, and leave answer in stack

then evaluate expression e_2, leaving that answer on stack

FIF or IF depending on type

if e_1 greater than e_2

if e_1 equal to e_2

if e_1 less than e_2

generates GOTO statement according to whether S is unconditional GOTO or computed GOTO

The pointers are set to point to the appended GOTO statement if the relationship is true or to the next source statement if the relationship is false.

7.7.6 DO Statements

7.7.6.1 Internal Subroutine Call –

DO I

Compiles into:

GOSB
)I

7.7.6.2 External Subroutine Call –

Do s (v_1,v_2, . . .,v_n)

Compiles into:

4 xxx
)v_1
)v_2
. . .
)v_n

xxx – index into call driver table

NOTE: The Call Driver table is NOT available to GENDAC

pointers to the base arguments

7.7.7 Return Statement

The RETURN statement compiles into one of two forms, depending on whether it is an internal or external subroutine return.

An internal subroutine has the form,

RETURN

and compiles into:

RET

An external subroutine return has the form,

Return s

and compiles into:

RETE
*count

where the count is the number of arguments over which to jump in the return.

7.7.8 Loop Statements

7.7.8.1 FOR Statement –

FOR i = m_1 TO m_2 STEP m_3

Compiles into:

IL
)m_1
IS
)i

The parameters m_2 and m_3 are used in the NEXT statement. If Step m_3 does not appear in the statement, m_3 is assigned the value 1.

7.7.8.2 NEXT Statement –

NEXT i

Compiles into:

NEXT
)i
)m_3
)m_2
)POINTER

where m_3 and m_2 were defined in the corresponding FOR statement. The pointer is a displacement address to the cell following the corresponding FOR statement.

7.7.9 Arithmetic and Logical Operators

This class of vectors has no argument following the vector. The routines supporting the vectors deal only with the arithmetic stack. Within this class there are two types of operations.

a. Binary – operate on the last two values of the stack, replacing those values with the single result of the operation.

b. Unary – operate on the single, last value of the stack, replacing that value with the result of the operation.

7.7.9.1 Binary Vectors – Table 7-6 is a listing of the binary vectors.

Table 7-6
Binary Vectors

Integer	Floating	Function
MPY	FMPY	Multiply
DIV	FDIV	Divide
EXP	FEXP	Exponentiation

(continued on next page)

Table 7-6 (Cont)
Binary Vectors

Integer	Floating	Function
ADD	FADD	Addition
SUB	FSUB	Subtraction
OR	–	12-bit Boolean OR
LAND	–	12-bit Boolean AND

7.7.9.2 Unary Vectors – Table 7-7 is a listing of the unary vectors.

Table 7-7
Unary Vectors

Integer	Floating	Function
NOT	–	12-bit logical negation
NEG	FNEG	Arithmetical negation
FLT	–	Integer to floating-point conversion
–	FIX	Floating-point to integer conversion (truncate)

7.7.10 Special Vectors

The vectors described to this point are all normal compiler generations produced in response to some language-level statement. In this paragraph, two vectors are introduced that are not generated by the Compiler; however, they are available for PAL-I coding. Remember that all vectors must appear in FIELD 1 and are processed with the data field set to 1.

ESUP Name	Purpose
GBIN	Go-Binary: exit from interpretive mode, enter machine mode at the next instruction in FIELD 1 with data field set to 1.
XTRP xxxx	Exit from interpretive mode, enter machine mode in FIELD 0 at the instruction whose address follows the XTRP command. Data field set to 1.

7.7.11 Special PAL-I Code

Occasionally, the vectors already described are not sufficient to service user needs, and routine must be created to directly interact with the Executive. The following "macro-calls" are provided as equate statements through ESUP to ease communication with the Executive. The commands in Table 7-8 are all machine-mode instructions.

Another "macro-call" is available to allow communication for I/O handlers (FIELD 0). The call is "CALL" followed by an argument. Each argument references a subroutine within the Executive that supplies either information or a service. A complete discussion of "CALL" and the available arguments is presented in Paragraph 7.6.3 "I/O Handler".

Table 7-8
ESUP Commands

ESUP Name	Purpose
INTERP xxxx	Command given in FIELD 1 with data field set to 1, to enter vector mode at the next word. xxxx is interpreted as a vector.
GOTERP yyyy	Command given in FIELD 0 with data field set to 1, to enter vector mode in FIELD 1 at the address specified by yyyy.
VECTOR	Command given in FIELD 0 for Executive to continue processing the user program. This is normally the last instruction of any library routine or core-resident function created by a user.
ABSGO	Command given in FIELD 0 with data field set to 1 to continue operation of the user program (vector mode) at the address specified in location 0100_8 of FIELD 0. Normally used to simulate the operation of a GOTO . . . or a conditional test.
SYSOUT	Command given in FIELD 0, data field irrelevant, to abort the current SNAP. An error has been detected and no reasonable method of continuing is available.
INTRET	Used in the interrupt I/O handler to return to the AC and LINK restore routine, exiting from interrupt mode.

7.8 RUNNING GENDAC

GENDAC, in conjunction with the DEC-supplied or user-written library tape, is a powerful system configurator (see Chapters 2 and 4). Its main features are as follows:

a. Conversational operation – GENDAC requests the option required. If the user is uncertain as to the correct response, he strikes the question mark (?) key. GENDAC then prints an explanation to the user.

b. GENDAC links DEC-provided I/O handlers and math functions (sine, log, etc.) into the Executive.

c. GENDAC links user-developed I/O handlers, functions, and subroutines into the Executive. Subroutines can be written in INDAC or PAL languages.

d. GENDAC places subroutines in the top of the last supplied disk or on the first disk (disk 0) if available.

APPENDIX A
SUMMARY OF INDAC STATEMENTS

LEGEND:

a = print string

c = i SEC i MIN i HR / HH:MM (AT statement)

e = expression (logical or arithmetic)

i = integer variable

k = constant

l = statement label

m = control parameter

s = subroutine name

t = tag

u = device name

v = variable name

SCN = control names for device

A.1 EXECUTABLE STATEMENTS

I. ASSIGNMENT STATEMENTS

LET v = e

II. CONTROL STATEMENTS

1. GOTO Statements

GOTO l

GOTO ($l_1, l_2, \ldots, l_n$), i

2. IF Statement

IF (e_1) o (e_2) THEN S

o = Relational Operator

S = GOTO l

or

S = GOTO ($l_1, l_2, \ldots, l_n$), i

3. DO Statements

DO l

DO s ($v_1, v_2, \ldots, v_n$)

DO PHASE #t PRIORITY k

DO SNAP #t PRIORITY INTERRUPT

DO SNAP #t PRIORITY k

4. RETURN Statement

   RETURN
   RETURN s

5. Program Control Statements

   EXIT
   EXIT THEN DO SNAP #t PRIORITY k
   EXIT THEN DO PHASE #t PRIORITY k

6. Loop Statement

   FOR i = m_1 TO m_2 or FOR i = m_1 TO m_2 STEP m_3
   NEXT i

7. TIMER Statement

   TIMER (STOP, #t)
   TIMER (START, #t)

III. INPUT/OUTPUT STATEMENTS

1. GET (u, #t) list or GET (u) list
2. SEND (u, #t) list or SEND (u) list

IV. ACTIVITY STATEMENTS

1. #t_1 DO SNAP #t_2 EVERY c PRIORITY k
2. #t_1 DO SNAP #t_2 DELAY c PRIORITY k
3. #t_1 DO PHASE #t_2 DELAY c PRIORITY k
4. #t_1 DO SNAP #t_2 AT c PRIORITY k
5. #t_1 DO PHASE #t_2 AT c PRIORITY k

A.2 NON-EXECUTABLE STATEMENTS

I. SPECIFICATION STATEMENTS

1. .EQUIPMENT
   ↑*u (device code)
   ↑CHAN (m,n) . . . . . (multiplexer channels)
   ↑#t SCN_1, SCN_2, . . . SCN_n (control information)

II. ALLOCATION STATEMENTS

1. .STORAGE $v_1, v_2, \ldots, v_n$
2. #t .FORMAT
3. #t .HEADER

III. PROGRAM UNIT SEGMENTATION STATEMENTS

1. #t .PHASE
2. #t .SNAP
3. .SUBROUTINE s

IV. COMPILER DIRECTIVE STATEMENTS

1. .ACTION
2. .PROCESS
3. .END

APPENDIX B
EXAMPLE PROGRAM

This program shows the technique of making an array available to a subroutine.

```
                . ARRAY PASSING EXAMPLE
                .STORAGE II
#1              .PHASE
#10             DO SNAP #20 EVERY 30 SEC PRIORITY 2
                .ACTION
                TIMER (START, #10)
#20             .SNAP
                .STORAGE IK (1, 10)/1,2,3,4,5,6,7,8,9,0/,A(1,3)
#21             .FORMAT
                XX.XX   XX.XX   XX.XX
                .PROCESS
                LET I = II
                DO SUB (IK, I, A)
                SEND (TTY, #21) A(1), A(2), A(3)
                EXIT
                .SUBROUTINE  SUB (IDUM, IDM, DUM)
                .PROCESS
                LET IA = IDM
                LET IB = IA + 3
                FOR I = IA TO IB
                LET DUM(I) = IDUM(I)
                NEXT I
                RETURN SUB
                .END
```

APPENDIX C
USING THE DISK MONITOR SYSTEM

C.1 EDITOR COMMAND SUMMARY

Command	Format	Meaning
READ	R↵	Read incoming text and append to buffer until a form feed is encountered.
APPEND	A↵	Append incoming text to any already in the buffer until a form feed is encountered.
LIST	L↵	List the entire buffer.
	nL↵	List the line n.
	m,nL↵	List lines m through n.
PROCEED	P↵	Proceed and output the entire contents of the buffer and return to command mode.
	nP↵	Output line n, followed by a form feed.
	m,nP↵	Output lines m through n, followed by a form feed.
TRAILER	T↵	Punch four inches of trailer.
NEXT	N↵	Punch the entire buffer and a form feed; kill the buffer and read next page.
	nN↵	Repeat the above sequence n times.
KILL	K↵	Kill the buffer.
DELETE	nD↵	Delete line n.
	m,nD↵	Delete lines m through n.
INSERT	I↵	Insert before line one all text until a form feed is encountered.
	nI↵	Insert before line n until a form feed is encountered.
CHANGE	nC↵	Delete line n and replace it with any number of lines from the keyboard until a form feed is encountered.
	m,nC↵	Delete lines m through n, replace from keyboard as above until form feed is encountered.
MOVE	m,n$kM↵	Move and insert lines m through n before line k.
GET	G↵	Get and list the next line beginning with a tag.
	nG↵	Get and list the next line after line n which begins with a tag.
SEARCH	S↵	Search the entire buffer for the character specified (but not echoed) after the carriage return; allow modification when found.
	nS↵	Search line n, as above, allow modification.
	m,nS↵	Search lines m through n, allow modification.

(continued on next page)

Command	Format	Meaning
END FILE	E↵	Process the entire file (perform enough NEXT commands to pass the remaining input to the output file) and create an end-of-file indication; legal only for output to the system device. If the low-speed paper tape reader is used for input while performing an E command, the paper tape reader will eventually run out of tape, and at this point typing a form feed will allow the command to be completed.

C.2 EDITOR KEY FUNCTION SUMMARY

↵	(carriage return)	Text mode: Enter the line in the text buffer. Command mode: Execute the command.
←	(back arrow)	Text mode: Cancel the entire line of text and continue typing on same line. Command mode: Cancel command.
\	(rubout)	Text mode: Delete from right to left one character for each rub-out typed (is not in effect during a READ command). Command mode: Delete entire command.
FORM FEED		Text mode: End of input, return to command mode.
.	(period)	Command mode: Current line counter used as argument alone or in combination with + or - and a number.
/	(slash)	Command mode: Value equal to number of last line in buffer and used as argument.
↓	(line feed)	Text mode: Used in SEARCH command to insert a CR/LF into the line being searched. Command mode: List the next line.
>	(right angle bracket)	Command mode: List the next line.
<	(left angle bracket)	Command mode: List the previous line.
=	(equal sign)	Command mode: Used in conjunction with . and / to obtain their value (.=0027).
:	(colon)	Command mode: Lower case character, same function as =.
→	(tabulation)	Text mode: On output is interpreted as a tab-rubout combination.

C.3 ERROR MESSAGES

As an input command string is being typed, Monitor recognizes any incorrect syntax and responds with one of the error messages in the following table.

Message	Meaning
?	Illegal syntax or miscellaneous error condition
D	Directory on the systems device is full
E	Too many inputs or outputs were entered
I	No such inputs
S	System I/O failure

APPENDIX D
INDAC COMPILER ERROR MESSAGES

Error Message	Meaning
1	Incorrect representation for statement TAG.
2	Incorrect representation for statement LABEL.
3	This statement is not syntactically complete. The compiler needs more information.
4	Unrecognizable statement verb.
5	While scanning this IF statement, the Compiler could not find or recognize a relational operator (EQ, NE, GR, GE, LS, LE) or the THEN conjunctive.
6	The assignment variable (to the left of the equal sign) is not recognizable as a variable name; or, this variable is a dimensional array and must be subscripted.
8	The expressions in an IF statement cannot be real and integer (or logical). They must be of the same mode. A mixed mode of integer/logical is permitted, however.
9	The GOTO in this IF statement is missing or unrecognizable.
10	There is a subscripted variable in this statement whose subscript is bad. It is either (1) unrecognizable (2) a real constant or (3) the integer constant used as a subscript exceeds the declared size of the array.
11	The equal sign is missing or misplaced in this LET statement.
12	This arithmetic expression is not syntactically complete; the Compiler expects more information. Generally this means that an operand is missing following some operator. Occasionally, this condition is caused by some previously detected error in the same expression.
13	More than one unary minus in a row encountered while scanning this expression (for example, --A is not permitted, while - (-A) is O.K.).
14	There is a constant in this statement that is improperly written.
15	Only integer constants are permitted as exponents or within exponential expressions.
16	Mixed mode expressions are not permitted. Real and integer (logical) variables cannot be intermixed in one expression.
20	Illegal operator in arithmetic expression.
21	Illegal operator in logical expression.
22	Right parenthesis encountered adjacent to left parenthesis.
23	Extra left parenthesis (or parentheses) encountered.
24	Statement labels can only be used within the PROCESS section of a SNAP unit.
25	A logical expression cannot be equated to a real assignment variable.
26	Mixed mode expressions are not permitted. Arithmetic and logical operators cannot be intermixed in the same expression.
28	Unrecognizable label encountered during processing of a computed GOTO statement.

(continued on next page)

Error Message	Meaning
29	Missing or misplaced comma within a computed GOTO statement.
30	Missing or misplaced right parenthesis within a computed GOTO statement.
31	The transfer variable in a computed GOTO statement may only be (1) an unsubscripted integer variable or (2) an integer variable subscripted by an integer constant.
32	Priority verb missing, mispelled or not found in expected sequence. This error may be due to untagged activity statement not in activity section.
33	Array sizes must be declared by integer constants only.
34	Syntactial error in EXIT statement.
35	Time declaration in AT statement must be positive integer constants of form: HH:MM with HH less than 24, and MM less than 60.
36	Missing or unrecognizable DELAY or EVERY in an activity statement.
37	The time declaration in an activity statement must consist of (1) an integer constant or an unsubscripted integer variable or an integer variable subscripted by an integer constant (2) which must be followed by the contractions SEC, MIN, or HR.
38	Priority level stated is invalid. Level must be integer constant, between limits 1 to 11.
39	Variable name has been used more than once in STORAGE statements in this or parent program unit.
40	This tag has been previously defined, or this label has been previously defined within this SNAP.
41	The first character of a variable name must be alphabetic.
42	Could not make sense out of what to DO. Probably unrecognizable characters. Subroutine names must begin with alphabetic character.
43	Illegal FORMAT statement.
44	This statement is not permitted at PHASE level. .FORMAT, .HEADER, and .STORAGE are not permitted at PHASE level.
45	Error in declared FORMAT statement.
46	This statement must have a tag reference appended to it for other statements to reference.
48	Subroutine names must begin with an alphabetic character.
49	This symbolic device name does not exist in this INDAC 8 System.
50	The ACTION section can only appear in a PHASE unit.
51	Dot statements (for example, .FORMAT) are not permitted within executable code in the PROCESS section of a SNAP unit.
52	The word STOP, or START is missing, misspelled, misplaced, or otherwise unrecognizable.
53	Same as 52, except first character of the description word is not even alphabetic.
54	Missing comma in this statement.
55	This timer statement refers to an activity statement that was never declared.
56	A timer statement can only reference an activity statement.
57	Missing right parenthesis in this statement.

(continued on next page)

Error Message	Meaning
59	PRIORITY INTERRUPT is misspelled, misplaced or occurs more than once in ACTION section. Only one PRIORITY INTERRUPT per PHASE is permitted.
60	The .ACTION statement can only occur once within a PHASE program unit.
61	This statement can only appear in the ACTION section of a PHASE program unit.
62	DO PHASE may not appear as a freestanding statement. DO PHASE may be used only in activity or EXIT statements.
63	DO SNAP out of order. "DO SNAP #t PRIORITY n" may only appear in the action or process sections.
65	Activity statements may only appear immediately following the .PHASE statement and before the .ACTION statement.
66	This activity statement is incomplete.
67	The .PHASE statement is out of sequence. Generally this error occurs when a .PHASE follows a .PHASE without an intervening SNAP program unit.
68	The .SNAP statement is out of sequence. Generally this error occurs when a .SNAP erroneously follows a SUBROUTINE unit or appears in the system unit before a .PHASE statement occurs.
69	Only one .INTERRUPT device list is permitted during compilation.
70	The .INTERRUPT statement is out of sequence. It may appear only at system level.
71	The .SUBROUTINE statement is out of sequence. This error generally is due to no SNAP unit associated with the preceding PHASE unit.
76	Either (1) an executable statement cannot have a tag reference appended to it or (2) executable statements can only occur in the PROCESS section (with some minor exceptions in the ACTION section).
78	The first character of a device symbolic name must be alphabetic.
80	This subroutine name has been used before, either it appears in the subroutine library or the user has declared it twice in two SUBROUTINE statements.
82	The tag reference for this PHASE or SNAP has been used before.
84	Missing left parenthesis in this statement.
85	The .PROCESS statement may only appear only in a SNAP or SUBROUTINE program unit.
86	Subroutine size is exceeded. The compiler code for a subroutine cannot exceed 128 words.
87	While generating compiled code for the preceding program unit, an undefined label or tag has been found. The user evidently referred to it but never defined it (for example, GOTO 100, but statement label 100 never occurred within the unit).
88	The number of digits specified exceeds the format statement limit. Total width may not be greater than 15 digits, only the first six digits output are significant.
90	The line containing .HEADER must be blank following .HEADER. The header line(s) can only appear in the line(s) following.
91	There is an extraneous decimal point in this .FORMAT statement.
92	The system tables not on system – either ⟨PZ⟩, ⟨IF⟩, ⟨SD⟩, or ⟨XS⟩ do not appear on the system.

(continued on next page)

Error Message	Meaning
93	There is an illegal octal constant in this statement. Octal constants may not contain the digits 8 or 9 and must be four or less characters in length.
94	There must be at least one executable statement (for example, DO SNAP) in the ACTION section of a PHASE unit.
95	An array size declaration in a storage statement must appear as (1, N) where N is an integer constant greater than a positive one.
97	The right parenthesis is missing in a "repeat" command in a .FORMAT statement.
AA	The .EQUIPMENT statement. If present, must be the first statement in the system.
AB	The compilation has been aborted. No object code has been generated.
A1	The control variable in a FOR statement can only be an unsubscripted integer variable, or an integer variable subscripted by an integer constant.
A2	Missing equal sign in this FOR statement.
A3	One of the loop control parameters within this FOR statement is a real constant or not a valid constant.
A4	Nested FOR statements cannot share the same control variable.
A5	Internal DO statements (for example, DO 100) are only allowed in the PROCESS section of a SNAP or SUBROUTINE.
A6	This tag is missing from PHASE reference table.
A8	All loop control variables in a FOR statement must be integer mode.
A9	Activity statement tag is not defined in current PHASE.
C1	"DO PHASE #t EVERY i . . ." is not a legal activity statement.
C4	EXIT statements are not allowed in the ACTION section of a PHASE.
C5	An integer variable used as a time declaration in an activity statement must have been declared in system storage.
C6	Missing time dimension (HR, MIN, SEC).
C7	Unrecognizable subscript in activity statement.
C8	A time declaration in an activity statement can only be subscripted by an integer constant.
D1	This statement has a variable name which is in system storage. Subroutines cannot reference variables in system storage.
D2	Dummy variable cannot be subscripted by integer constant.
D4	Table storage capacity has been exceeded in the Compiler. Try breaking the expression into simpler expressions and recompile. This error can only occur only with unusually large arithmetic (or logical) expressions with enormous quantities of parentheses.
D5	This, too, is a rare error which can occur in an arithmetic (or logical) expression with very large quantities of function calls and parentheses. As with error D4, try breaking the expression into simpler expressions and recompile.
D6	Too many right parentheses in this expression.
D7	A dimensional variable which appears in the .EQUIPMENT statement may appear in a .STORAGE statement only as the name alone, with no dimension appended. The .STORAGE statement will allocate storage for the variable according to the implied dimension appearing in the .EQUIPMENT statement.

(continued on next page)

Error Message	Meaning
D9	A computed GOTO statement cannot have a variable which is subscripted by another variable. The subscript, if present, must be an integer constant.
E1	There is an unrecognizable line in the .EQUIPMENT statement.
E2	The second channel must be greater than the first channel in a multiplexer declaration.
E3	A variable cannot be declared twice in the .EQUIPMENT statement.
E4	The .EQUIPMENT statement contains two control lines having the same tag reference.
E5	This control action is not defined in this INDAC 8/2 System.
E6	Is this an attempt at a DO conversion subroutine?
E7	The first character of a conversion subroutine must be alphabetic.
E8	A comment line cannot have a tag reference or label appended to it.
E9	Dummy variable is previously defined (either in system storage or as another dummy in this statement).
G0	When addressing a multiplexed device, all variables must have appeared in the .EQUIPMENT statement.
G1	Missing left parenthesis in GET/SEND statement.
G2	This I/O direction is not possible (permitted) with this device.
G3	The tag reference in a GET/SEND statement must be to a .FORMAT or .HEADER statement or to a control line in the .EQUIPMENT statement.
G4	The tag reference in this GET/SEND statement is undefined.
G5	The control tag referred to in this GET/SEND statement is not associated with this device.
G6	Undefined conversion subroutine called for by the control tag referred to in this GET/SEND statement.
G7	This device is not formatable.
G8	A data list is not permitted when outputting a .HEADER.
H1	A subroutine dummy is not permitted as a subscript, or as variable in another subroutine call.
H2	The variable was not declared to be dimensioned, yet it is subscripted within this statement.
H3	The syntax of this FOR statement is incorrect (for example, the TO is missing or misspelled, or the STEP is wrong, etc.).
H4	Parameters in a FOR or DO subroutine statement must not be subscripted by other variables; however, subscripting by an integer constant is permitted.
H5	This NEXT statement refers to a loop control variable that either was never declared in a FOR statement or that was satisfied by a previously occurring NEXT statement.
H6	Same as H5, except that the difference appears in the subscript.
H7	Subroutine dummies cannot be used in a FOR statement.
H8	This RETURN statement references an unknown subroutine name.
H9	The preceding PHASE unit contains a reference to an undefined SNAP program unit.

(continued on next page)

Error Message	Meaning
J1	EXIT statements cannot request execution of an internal subroutine. (For example, EXIT THEN DO 100 is illegal.)
J2	The first character of the subroutine name must be alphabetic.
J3	Sometime during compilation there was a reference to a SUBROUTINE which never got defined.
J4	Sometime during compilation there was a reference to a PHASE unit that never got defined.
J5	The preceding SNAP or SUBROUTINE program unit contains a FOR statement that was never terminated by a NEXT statement.
J6	The SUBROUTINE name is illegal for one of the following reasons: (1) An external subroutine cannot be core-resident. (2) A subroutine name is used as a function call. (3) A function name is used as a subroutine call.
J7	Multiple base entries have been detected in the external subroutine table. GENDAC has set the "A" bit twice in the chain.
J8	The END statement is not in order. It must follow either a SNAP unit or a subroutine unit.
J9	This statement is complete so far as the Compiler is concerned, but extraneous characters occur at the end.
K2	Missing comma in GET/SEND statement.
K3	A subroutine dummy cannot appear in a GET/SEND statement.
K4	Error while unpacking data list in preset, unrecognizable constant.
K5	Integer variable cannot be preset with real constant.
K6	Array not large enough to preset with number of entries in data list.
K7	Real variable cannot be preset with integer constant.
K8	Real constant cannot specify the number of repeats, real constant encountered out of order.
K9	End of .STORAGE statement encountered within preset, terminate preset with a slash.
NO	There is no END statement. The END statement was inadvertently omitted.

NOTE

The errors that follow are internal compiler errors. They may occur when a user error has been encountered and recovery of compilation assumed a condition that was later incorrect. These errors may be ignored unless they are the only errors in a compilation. If this occurs contact nearest Software Support personnel.

S1	Attempt to move tag from System tag table to Phase tag table but unable to find in System tag table (tape 4).
S2	Entry in Executable Verb table exceeds size of table (tape 4).
S3	Label around ASCII string not present in .HEADER statement.
S4	Label for base of executable code or base of timer code not present.
S5	Error in Phase Tag table for this phase tag (tape 4).

APPENDIX E
SPUT ERROR CONDITIONS

SPUT has three classes of errors:

1. Single letter – Monitor message same as listed in Appendix C.3.
2. Message which states the error condition. Usually a disk error. If CDT ERROR occurs list source program and contact nearest software support personnel, because this is a system diagnostic.
3. ?! and Halt. A disk error has occurred while reading the Executive into core from disk unit 1. AC contains the disk error flags.

APPENDIX F
INDAC 8/2 EXECUTIVE ERROR MESSAGES

There are five general classes of Executive errors.

Error Message	Meaning
00–19	Interrupt level system error.
01	Interrupt level stack overflow.
20–39	Interrupt level user detected error.
21	Interrupt chain error. No device responded but a device interrupted.
22	Device not listed in compilation.
23	No devices specified for this interrupt. Interrupting device does not appear in .INTERRUPT statement.
40–59	System level user detected error.
41	No file number specified in file command.
42	No data list specified in file command.
43	Real variable specified page in file command.
44	Real variable in GET (device) . . . . where device is "CLOCK", "KEYS", "STATUS", or "PRIORITY".
45	Real variable output attempted via "0" format control.
46	Real variable in GET (TTY, . . . statement.
59	Negative subscript encountered.
60–79	System level system error.
60	Invalid interpretive operation code encountered.
61	Subroutine nesting limit exceeded.
62	System level stack overflow.
90–99	Miscellaneous errors.
99	User or I/O Handler call to error routine PC and AC listed to locate caller and reason.

APPENDIX G
GENDAC ERROR MESSAGES

Some error messages are of the form:

ERROR NN (additional data)

Tables G-1 and G-2 are a complete list of GENDAC error messages; the following notes apply to this table:

NO NOTE	Parentheses contain "debug" number, it can be ignored; it distinguishes possible sources of this error and can change with reassembly of GENDAC.
NOTE A	The above debug number is preceded by the content of the accumulator when the error was detected; in many cases this bears some relationship to the field or expression that is syntactically incorrect.
NOTE B	The above (both) are preceded by a code for the field or expression found on the library tape when the error was detected. Meaning of code: 0 = end of tape 1 = code 2 = origin symbol other than *7777 3 = Field pseudo-op 4 = *7777
NOTE C	The above (except Note B) are preceded by a second "debug" number of the same kind.

Table G-1
Numbered GENDAC Error Messages

NN	Notes	Meaning
01		DN full
02		Disk full
03		No free block above start of system table to be extended
04		Overlay not found in GOVL
05		Invalid ⟨PZ⟩
06		⟨XS⟩ pointed to nonsubroutine block (discovered at REPLACE? -Y)
07		"Impossible" system errors
10		HSR time-out
11	B	Library tape contains garbage

(continued on next page)

Table G-1 (Cont)
Numbered GENDAC Error Messages

NN	Notes	Meaning
12	C	EOT, *, FIELD found when code expected — following cases distinguishable by values dependent on assembly:
		TEXT, length (*761N), length (*762N, *763N), # entry points, Rel. addr. of e.p., device usage code, IOT, #ctl. wrds, Ctl. wrd. value.
13	A	FIELD follows tape leader
14	B	long name of *7660 group missing
15	A	*761N, *762N, *763N missing
16		*7610, *7630 illegal
17		modtyp = 5 illegal
20	B	FIELD after *761N, *762N, *763N missing
21	A	FIELD after *761N, *762N, *763N not 0 or 1
22	B	module origin requirement after *761N, & 762N, *763N missing
23	A	length of call-up module $> 200_8$
24	A	length 0 or too big
25	A	# entry points 0 or too big
26	B	missing control-origin after # entry points
27	B	missing *7650
30	A	# control words 0 or too big
31	A	more control words than declared
32	A	FIELD in place of control word or control-origin at end of control words
33		*7647 illegal
34		*7641 not associated with *763N
35		entyp incompatible with modtyp or two *7641 or *7642 in one module
36	A	Core-res. function entry point invalid
37	A	I/O device handler entry point invalid
40	A	IOT invalid
41		*7651 name not in ⟨**⟩
42		*7651 name refers to *7640 not *7641 or occurs twice on one tape
43	B	Missing *766N or actual origin of module code
44		Illegal syntax in *766N TEXT
45		Displacement fixups illegal in disk-resident module
46	B	Missing control-origin after *7777 separator (EOT is legal)
47		Manual repositioning of tape associated with "N" response did not leave tape on a *7777 separator

Table G-2
Unnumbered GENDAC Error Messages

Error	Description
?	*Not* an error message — indicates GENDAC awaiting keyboard input (printing of previous message was suppressed) Type '?' to discover.
?	Keyboard input not understood — (or illegal) — try again
↑C	User aborted run with CTRL/C from keyboard
NO SUCH FILE	No file by that name exists of type S if OPT=S or of type B if OPT=B
SUBROUTINE FILE INVALID	File specified does not have proper format for INDAC compiler output of a single subroutine
SUBROUTINE EXCEEDS ONE PAGE	Self-explanatory
AAAA ALREADY IN ⟨BB⟩	Name specified (only four characters printed) exists in specified system table — in general: logical error in constructing or loading library tape(s) — may need to regenerate system from scratch.
FIXUP ILLEGAL	Reference to disk-resident module in a *766N
FIXUP UNDEFINED AAA BBBB	*766N did not appear for this "FIELD" .AAAA = index of fixup (last digit of *766N) BBBB = value of "ICI" for word to be fixed up
AAAA EXCEEDS ALLOCATED AREA	Module code lies outside declared module length
TURN ON LSR	LSR time-out has occurred — if caused by jam, turn off until jam fixed, if caused by moving switch to stop, no data lost; if caused by moving switch to free, or if tape sprocket holes ripped, must restart system generation.
TURN OFF LSR	GENDAC is returning to disk monitor — follow instructions!
CHECKSUM ERROR AAAA	If AAAA changes or message does not readout on repeating entire system generation process, then hardware trouble with paper tape reader or punch which produced library tape. If AAAA reoccurs, then bad copy of library tape (probably punch error).
DISK READ/WRITE ERROR	Hardware trouble with disk
CANNOT CONTINUE	Self-explanatory
SYSTEM BASE ALTERED!	Base of INDAC program area in core has been altered by GENDAC due to addition of core-resident code. All INDAC main programs must be recompiled to incorporate this change.
USE 8K (XXXX)	A library tape option valid only for 8K INDAC system has been used when K=4
⟨AA⟩ MISSING OR INVALID	Self-explanatory
⟨IF⟩ FULL	Too many core-resident functions have been defined (limit is eleven) or ⟨IF⟩ is bad.
CANNOT ALLOCATE CORE	Either no room left in valid area for type of module now being allocated or ⟨CM⟩ is bad.

APPENDIX H
EXECUTIVE COMMAND DECODER OPERATION

H.1 OPERATOR COMMANDS

CTRL/A	Attach to Command mode if echo successful. If no echo or if ?, Command mode is not permitted.

All other commands active only in Command mode.

CTRL/C	Return to Monitor mode
CTRL/D	Initiate "RUN" command
CTRL/U	Delete entire line (Start line over)
CTRL/P	Release Teletype from Command mode back to Operator mode
CTRL/V	Value changes to be made in following modes
C	Inspect and open clock for preset HH:MM:SS
O, D, E	Inspect and open location for modification in Octal, Decimal, Exponential Notation
RUBOUT	Delete and echo last character in line; if none, no echo

H.2 COMMAND DECODER ERROR MESSAGES

BA	Bad Address:	The address requested is not within the defined system storage area
EP	Error in Phase:	The phase requested cannot be found
ES	Error in Syntax:	The input line cannot be resolved or the arithmetic information is not correct for the mode of operation.
?		Cannot resolve command.

APPENDIX I
THE GENDAC LIBRARY

The GENDAC library supplied with the INDAC software kit contains a library of I/O handlers and mathematical functions; they are:

1. **I/O Handlers**

PART OF EXECUTIVE:
- KEYS
- TTY
- STATUS
- PRIORITY
- CLOCK

UDCE (UDC)
UDCP (UDC)
ADC (AF01, AF02, AF03, AFC-8, AD01)
AF04
DAC (AA01A, AA50)
TTY2
TTY3
TTY4
PTP

2. **Mathematical Functions**

SIN
COS
ATAN
LOG
EXP

PART OF EXECUTIVE:
- SQRT

3. **Sample Call for Functions**

LET A = SQRT (B)

APPENDIX J
SYSTEM COMMUNICATION TABLES

Table J-1 lists and describes the Systems Communication Tables.

Table J-1
System Communication Tables

Name	Located In	Save Address	Used By	Content
Page Zero ⟨PZ⟩	Executive	0	Compiler	Page zero of eventual run-time system including various system communication parameters used by SPUT, GENDAC and the Compiler.
System Devices ⟨SD⟩	SGEN[1]	400–777	Compiler	Device handler names and pointers; legal .EQUIPMENT statement control words.
External Subroutines ⟨XS⟩	SGEN[1]	1600	Compiler	Subroutine names and pointers; function names and ⟨IF⟩ indexes.
Intrinsic Functions ⟨IF⟩	Executive	7000	Compiler	Vectors to operators and functions.
Core Map ⟨CM⟩	Executive	200	GENDAC	Defines the areas of Executive core available to core-resident modules.
⟨**⟩	GENDAC[2]		GENDAC	Allocation data for call-up modules and entry points used for fixup purposes alone.

[1] Created by SGEN and modified by GENDAC when the user configures a specific system. (The SGEN program examines the system configuration to determine the number of disks present and modifies internal Disk Monitor System tables to reflect the disks present.)

[2] ⟨**⟩ is cleared at entry to GENDAC.

APPENDIX K
ESUP OPERATION

K.1 INTRODUCTION

ESUP is a special support program for INDAC. ESUP produces an ASCII paper tape of equivalences (=) for use with PAL-I. One set of such equates is the actual interpretive code produced by the INDAC compiler for the run-time Executive. Those users who wish to code their own interpretive routines will find these equivalences useful. Other equates include addresses or instructions to link user code to the Executive. The file used to extract the equivalence information is the ⟨IF⟩ file. This is a communication table for the Executive and the Compiler; it defines the locations within the Executive of various calls that the Compiler or user will require. ESUP is also capable of producing a listing of available core in the executive area. This will be useful to users who wish to write I/O handlers or core-resident functions.

K.2 LOADING PROCEDURE

The following procedure will enable the user to load ESUP.

Step	Procedure
1	Place the ESUP paper tape in the high-speed reader.
2	Turn on the high-speed punch.
3	Load the program using the disk monitor system as follows:

```
.LOAD↵
*IN-R:↵
*
ST=200↵
↑↑  (User types CTRL/P after each ↑)
```

K.3 PRODUCING THE EQUATE TAPE

The following procedures will produce the equate tape.

Step	Procedure
1	The program types: *OUT- The user then responds to the output request with R:↵

(continued on next page)

Step	Procedure
2	The program types: *IN- The user then responds to the input request with S:⟨IF⟩↵.
3	The program types: *OPT- The user then responds to the option request with I (no CR is required).
4	The program then punches the equate tape, using the high-speed punch. When the output is finished, the program types: FINI and returns to request further outputs: *OUT-

K.4 PRODUCING THE CORE MAP

The following procedure will produce a printout of the core map.

Step	Procedure
1	The program types: *OUT- The user then responds to the output request with T:↵.
2	The program types: *IN- The user then responds to the input request with S:⟨CM⟩↵.
3	The program types: *OPT- The user then responds to the option request with C (no CR is required).
4	The program then types a table of the first free location on each page of the executive image and the length of the free area on that page. When the output is finished, the program types: FINI and returns to request further outputs: *OUT-

K.5 RETURNING TO MONITOR

Typing CTRL/C to a program request returns the user to the Disk Monitor.

K.6 RESTRICTIONS

The system must run under Disk Monitor control. The system assumes a high-speed punch for the ⟨IF⟩ table output.

K.7 REFERENCES

See *Disk Monitor System,* DEC-D8-SDAA-D for an explanation of the loading procedures and for more insight into the programmatic use of the Disk Monitor Command Decoder.

K.8 EXAMPLES

K.8.1 Loading and Usage

```
.LOAD
*IN-R:
*
ST=200
↑↑
*OUT-R:
*
*IN-S:⟨IF⟩
*
*OPT-I
FINI

*OUT-
```

K.8.2 Sample Equate Printout

A sample printout of a portion of the generated ASCII Equate tape.

NOTE

Equivalences may vary from one version of an Executive to another. The following is a sample of one version.

```
IL =     0201
ILI =    0200
ILE =    0277
ILEI =   0276
FL =     0236
FLI =    0233
FLE =    0307
FLEI =   0304
IS =     0211
ISI =    0210
ISE =    0302
ISEI =   0301
FS =     0253
FSI =    0250
FSE =    0314
FSEI =   0311
MPY =    0512
FMPY =   1200
DIV =    0437
FDIV =   1077
GOTO =   0361
CGO =    0400
IF =     0366
FIF =    1317
OR =     0556
LAND =   0552
NOT =    0564
NEXT =   0327
FIX =    1015
FLT =    1000
NEG =    0563
FNEG =   1012
GOSB =   0057
STOP =   0000
PAUZ =   0000
EXIT =   0037
ABRT =   0000
DUM1 =   0001
DUM2 =   0002
DUM3 =   0003
DUM4 =   0004
DUM5 =   0005
THLD =   0052
TMER =   0011
DUM6 =   0000
EXP =    1376
FEXP =   0110
RET =    0072
RETE =   0013
ADD =    0425
FADD =   0603
SUB =    0423
FSUB =   0600
```

INDAC HANDBOOK INDEX

A

AA01, 3-33
AA50, 3-33
Activity, 3-42, 3-47
Action, 3-42, 3-49
ADC, 3-30, 3-34
AD01, 3-34
Addition, 3-4
Address
  Base, 3-53
  Channel, 3-28, 3-39
  Displacement, 7-23
  Index, 3-12, 3-20, 7-26
  Pointer, 7-25
  System parameter, 3-53, 5-14, 6-2
  Start of user system, 6-1
AFC8, 3-34
AF01, 3-35
AF02, 3-35
AF03, 3-35
AF04, 3-36
Algorithm, 3-2
Analog to digital converter, 3-34
AND operator, 3-7
Arguments
  Arithmetic, 7-23
  MACRO calls, 7-20
  Subroutines, 3-44, 7-18
Arithmetic, 3-3
  Expression, 3-5
  Fixed point, 7-22
  Floating point, 7-22
  Operators, 3-4, 7-27
  Vector code, 7-23
Array, 3-20, 3-24
  Element, 3-25
  Integer, 3-25
  Name, 3-24
  Preset, 3-26
  Real, 3-25
  Subscript, 3-25
  Window, 3-25
ASCII Files, 5-3, 5-5, 5-13, 5-16
ASCII string, 3-22
Assembly-level Code, 4-7, 7-17

A (cont)

Assignment, 3-3
  Arithmetic, 3-3
  Boolean, 3-7
  Function, 7-17
Assembler, 5-3, 5-22, 7-3, 7-4
AT activity, 3-49
Atangent, 4-8

B

Background tasks, 3-53
BIN loader, 2-2, 4-3
Binary files, 5-3, 5-22
Binary mode, GENDAC, 4-7, 7-4
Binary operator, 7-27
Bit pattern, 3-6
Block (page)
  Disk, 3-38
  Paper tape, 2-3, 4-4, 7-5
Boolean, 3-6
  Bit pattern, 3-6
  Expression, 3-7
  Operators, 3-6, 7-27
  Vector code, 7-28
Bootstrap loader, Monitor, 5-2
Branching, 3-9
  Conditionally, 3-9
  Unconditionally, 3-9
Buffer
  I/O, 3-54, 7-14
  Text, 5-7

C

Calls
  Function, 7-17
  MACRO, 7-20
  PHASE, 3-47
  SNAP, 3-47
  SUBROUTINE, 3-44
CANCEL request, 3-61
Channel declaration, 3-39
  ADC, 3-34
  AF04, 3-34
  DAC, 3-34
  UDC, 3-37, 3-55
Checksum error, 2-3, 4-4

C (Cont)

CLOCK, 4-9
Display, 3-17
Get, 3-17
Set, 3-17, 6-3
Code
Interpretive, 7-3
INDAC statements, 3-2
MACRO calls, 7-20
PAL-I statements, 7-3, 7-28, K-1
Vector, 7-3, K-1
Cold start, 2-1, 4-2
Command Decoder,
Disk Monitor, 5-1
Executive, 3-51, 6-1
COMMAND request, 3-60
Comment, 3-43
Communication
Compiler tables, 7-1
Executive to SNAP, 3-54
Operator guidance, 3-54
COMP1-4, 2-5, 4-5
Compiler
Communication tables, 7-1
Dialogue, 5-14
Equates, 7-17, K-1
Error messages, 5-15, D-1
Loading, 2-4, 4-5
Printout, 5-14
Configuration, GENDAC, 4-2
Initial run option, 4-2
Re-run option, 4-2
Constant, 3-3, 3-20
Integer, 3-3
Octal, 3-6
Real, 3-3
Console, 3-59
Operator mode, 3-59
System mode, 3-59
Continuation line
EQUIPMENT statement, 3-28, 3-39
FORMAT statement, 3-18
Program statement (see .STORAGE statement)
STORAGE statement, 3-20

C (Cont)

Control declaration
ADC, 3-34
AF04, 3-36
FILE, 3-38
TTY, 3-60
UDC, 3-55
Control driver table, 7-20
Control word, 7-14
Conversion routine, 3-31
Core
Field, 5-23, 7-12
Functions, 4-8, 7-1, 7-12, 7-17
I/O handlers, 7-12
Job parameters, 3-53
Limitations, 3-54, 5-15, 6-1
Map, 7-2, 7-12, J-1, K-1
Monitor head, 5-2
Scheduling parameters, 3-53
SNAP, 3-53
Subroutine, 3-53
Swapping, 3-53
System parameters, 5-14
COS (change of state), 3-57
Cosine, 4-8

D

DAC, 3-28, 3-33
Data collection and control, 3-27
Data list, 3-17
GET statement, 3-35
SEND statement, 3-34
Decimal notation, 3-4, 3-24
DELAY activity, 3-48
Device, 3-33, 3-39, 7-14
Channel, 3-39, 7-14
Control, 3-39, 7-14
Declaration, 3-39
Hardware, 3-33, 4-9
I/O handler, 3-33, 4-9
Name, 3-33, 4-9, 7-14
Table, 2-5, 4-6
Dialogue
Compiler, 5-14
Editor, 2-7, 5-5

D (Cont)
GENDAC, 4-8, 7-29
PIP, 5-16
SPUT, 6-1
Digital input, 3-37
Digital output, 3-37
Digital to analog converter, 3-33
Dimensioned array, 3-20
Disk
Functions, 3-53, 4-8, 7-12
Job, 3-53
Monitor, 4-2, 5-1
Object file, 5-13
PHASE, 3-53
SNAP, 3-53
Source file, 5-5
Subroutine, 3-53, 7-12
System programs, 5-2
Division, 3-4
DO/RETURN statements
Language statement, 3-15
Vector code, 7-26
E
Editor, 2-7, 5-5
Commands, 5-7, C-1
Dialogue, 5-5
Error messages, 5-13
END statement, 3-1, 3-31
Equates
Compiler, 7-17, K-1
GENDAC, 7-5
EQUIPMENT, 3-28, 3-39, 3-53
Channel declaration, 3-28
Control declaration, 3-30
Subroutine declaration, 3-31
Error messages
Compiler, 5-15, D-1
Editor, 5-13
Executive, F-1, H-1
GENDAC, G-1
LOAD, 5-22
Monitor, 5-2, 6-2
PIP, 5-18
SPUT, E-1
ESUP, 7-12, 7-17, K-1
E (Cont)
EVERY activity, 3-47
Executive
Commands, 6-3, H-1
Loading, 2-5, 4-6
Operation, 3-50, 6-1
Parameters, 7-20
Explicit I/O handler, 3-37
Exponential notation, 3-19, 3-24
Exponentiation, 3-4
Expressions, 3-5
Arithmetic, 3-5
Boolean, 3-7
F
Field interrupt, 3-51, 3-55
Fields, 5-23, 7-12
File
Copying, 5-18
Creating, 5-5
Deleting, 5-17
Listing, 5-17
Merging, 5-19, 5-21
File name, 5-3
ASCII, 5-5, 5-14
Binary, 5-23
Input, 5-3
Output, 5-3
System, 5-14, 6-1
FILE, pseudo device, 3-31, 6-1, 6-3
Fixed point (integer), 7-22
Floating point (real), 7-22
Foreground tasks, 3-52
Format driver table, 7-20
Format, library tape, 7-9
Format, numeric, 7-22
FORMAT statement, 3-17, 3-22, 3-53
FOR/NEXT statements
Language statement, 3-12
Vector code, 7-27
Functions, 4-1, 7-1, 7-17
Atangent, 4-8
Coding details, GENDAC library, 7-16
Cosine, 4-8
Logarithm, 4-8
Sine, 4-8

G

GBIN vector, 7-28
GENDAC
  Binary mode, 2-7, 4-7, 7-4
  Dialogue, 2-7, 4-7, 7-29
  Error messages, G-1
  Equates, 7-5, 7-9
  Library tapes, 4-7, 7-5
  Library structure, 7-5
  Run options, 4-2
  System mode, 7-3
Generic code, 3-56
GET statement
  ADC, 3-31, 3-35
  AF04, 3-36
  CLOCK, 3-17, 3-20
  FILE, 3-38
  KEYS, 7-2
  PRIORITY, 3-54
  STATUS, 3-52, 3-54, 3-62
  TTY, 3-61
  UDCE, 3-38
  UDCP, 3-55
Global Parameters, 3-20, 3-24, 3-41
GOTO Statement
  Language, 3-9
  Vector Code, 7-25

H

HEADER Statement, 3-17, 3-22, 3-53
Headings, 3-22
Head of Monitor, 4-2, 5-1
HELD, 2-5, 4-6, 5-
HINDAC, 2-1, 4-2

I

IDACS, 3-1
Identification codes, library, 7-9
IDENTIFY request, 3-56
IDVM, 3-36
IF statement
  Language, 3-10
  Vector code, 7-25
INDAC language statements, 3-1, A-1
INDAC support programs, 2-4, 4-4
INDAC system tables, 2-4, 4-5, 7-1
INITIALIZE request, 3-56

I (Cont)

Input file, 5-3, 5-5, 5-13, 6-1
Integer constant, 3-3, 3-6
Integer quantity, 3-3, 3-6
  Constant, 3-3, 3-6
  numeric format, 7-22
  variable, 3-3, 3-6
Integer variable, 3-3
  Address, 3-22, 3-39
  Array, 3-20, 3-24, 3-39
  Data, 3-20, 3-24
Integrating digital voltmeter, 3-36
Iteration, 3-12
Interpretive Code, 7-3
Interrupt devices, 3-51
INTERRUPT statement, 3-51, 3-53
Interrupt task, 3-52
I/O handlers, 3-39, 4-1, 7-19
  ADC, 3-34
  AF04, 3-36
  CLOCK, 4-9
  Coding details, GENDAC library, 7-19
  DAC, 3-33
  FILE, 3-38, 4-1
  KEYS, 3-39, 4-1
  PRIORITY, 3-39, 4-1
  PTP, 4-9
  STATUS, 3-39, 4-1
  TTY, 3-39, 4-1
  TTY 2, 4-9
  TTY 3, 4-9
  TTY 4, 4-9
  UDCE, 3-37
  UDCP, 3-55

J

Job specification, 3-41

K

KEYS, 3-39, 7-2

L

Label, 3-9
Language statements
  Compiler code (vector) equates, 7-17, K-1
  INDAC, 3-2, A-1
  GENDAC equates, 7-5
  MACRO calls, 7-28

L (Cont)

PAL-I, 7-28, K-1
LELD, 2-6, 4-7
LET statement
Arithmetic, 3-3
Boolean, 3-6
Function call, 7-17
Vector Code, 7-23
Linearization routine
Call, 3-31, 3-44, 7-1
Coding details, 7-16
Definition codes, 7-9
Library codes, 7-9
Modules, 7-5
Paper tape, 7-5
Lists, 3-17, 3-20, 3-24, 3-3, 3-55, 3-60
LOAD, 2-8, 4-2, 5-22
Loading, 2-1, 4-1
Cold start, 4-2
Files, 5-5, 5-23
Monitor head, 5-1
Sample system, 2-1
User system, 4-1
Local parameters, 3-43
Logarithm, 4-8
Logs, 3-16
Looping, 3-12, 7-27
LSV (last scanner value), 3-57

M

MACRO calls, 7-28
MAKE, 2-4, 4-5
Mask, 3-13
Matrices, 3-21
Messages, 3-16
Modules, library
Basic, 7-5
Call-up, 7-6
Extension, 7-6
Size, 7-5
Monitor head, 4-2, 5-1
Monitor bootstrap, 5-2
Monitor mode, 5-1
Monitor system dump, 2-3, 4-4
MSUP, 2-2, 4-2, 4-3
Multiplex devices, 3-27, 7-14
Multiplication, 3-4

N

Name
Array, 3-20, 3-24
Device, 3-28, 5-3, 7-14
File, 3-38, 5-6, 5-16, 6-1
Function, 4-10, 7-17
I/O handler, 4-9, 7-14
Integer variable, 3-3, 7-22
Programs, 5-1
Real variables, 3-3, 7-22
Subroutine, 3-44, 7-18
Nesting, 3-12
NOT operator, 3-8
Numeric formats, 3-3, 3-24, 7-22

O

Object file, 6-1
Octal constant, 3-6
Octal notation, 3-6, 3-24
Operating mechanics, 2-1
Operator guidance, 3-16, 3-59
Operator mode, console, 3-59
Operators
Arithmetic, 3-4
Binary, 7-27
Boolean, 3-6
Relational, 3-10
Unary, 7-27
OR operator, 3-7
Output format, 3-17, 3-22, 7-14
Output file, 5-3, 5-5, 5-13

P

Page zero, 3-53, 7-2, 7-20
Page (Block)
Core, 3-54, 5-15, 7-5, 7-12
Disk, 5-15, 6-1, 7-12
Paper Tape, 2-3, 4-4, 5-7, 7-5
PAL-I, 7-3
Parameters, 3-3
Constant, 3-3
Executive, 7-20
Global, 3-41
Local, 3-43
Scheduling, 3-53
System, 3-53, 5-14, 6-7
Variable, 3-3

P (Cont)

PDP-8E, 2-1, 4-2
PDP-8I/L, 2-1, 4-2
PHASE, 3-1
  Parameters, 3-53
  Scheduling, 3-46, 3-53
  Structure, 3-42
PIP, 5-16
  Dialogue, 5-16
  Error messages, 5-17, 5-18, 5-19
  Options, 5-16
Point table I/O handler, 3-55
Priority level, 3-50, 3-54
  Background, 3-53
  Foreground, 3-52
  INTERRUPT, 3-51
PRIORITY pseudo device, 3-54
Process interface devices, 3-27
  ADC, 3-34
  DAC, 3-33
  Digital I/O, 3-37
  IDVM, 3-36
PROCESS statement, 3-1, 3-31, 3-43
Pseudo devices, 3-39
  FILE, 3-38, 5-15, 6-1, 6-3
  KEYS, 3-39, 7-2
  PRIORITY, 3-54
  STATUS, 3-54, 3-62
PTP, 3-17, 4-9

Q

Queue (Stack), 3-51

R

Real constant, 3-3, 3-6
Real quantity, 3-3, 3-6
  Constant, 3-3, 3-6
  Numeric format, 7-22
  variable, 3-3, 3-6
Real variable, 3-3
  Array, 3-20
  Data, 3-4
Relational operators, 3-10
Reports, 3-16
RIM loader, 2-2, 4-3

S

Scheduling, 3-1, 3-46, 3-51, 3-53

S (Cont)

  Immediate, 3-49
  Parameters, 3-53
  Priority, 3-50, 3-53
  Timed, 3-47
Segmentation, 3-40
  GENDAC equates, 7-5
  Phases, 3-42
  Snaps, 3-43
  Subroutines, 3-44
SEND statement
  DAC, 3-29, 3-34
  FILE, 3-38
  PTP, 3-18
  TTY, 3-18, 3-60
  UDCE, 3-30, 3-38
  UDCP, 3-55
SGEN, 2-5, 4-6, 5-1
Signal conditioning, 3-31
Sine, 4-10
SNAP, 3-1
  Background, 3-53
  Communication, 3-54
  Foreground, 3-52
  INTERRUPT, 3-51
  Scheduling, 3-46, 3-54
  Size, 5-15, 6-1
  Structure, 3-43
Source file, 5-1, 5-5, 5-13
SPUT, 2-9, 5-2, 6-1
Stack (Queue), 3-51
START, timer action, 3-42, 3-49
Statements, 3-1, A-1
  Data collection control, 3-27, 3-39
  Formatting, 3-17, 3-22
  Interrupt, 3-51
  Messages, 3-16
  Programming, 3-3
  Segmentation, 3-40
  Scheduling, 3-46
  Storage, 3-20, 3-24
STATUS pseudo device, 3-54, 3-62
STOP, timer action, 3-42, 3-49
Storage requirements, 5-15, 7-5
STORAGE statement, 3-20, 3-24, 3-53

S (Cont)

Subroutines
- Coding details, GENDAC library, 7-18
- External, 3-44, 7-1
- Implicit, 3-31, 3-45, 7-1
- Internal, 3-44
- Size, 5-15, 7-5

Subtraction, 3-4
Subscript, 3-21, 3-25
Swapping, 3-54
System files, 5-3, 5-14
System mode,
- GENDAC, 4-7, 7-3
- Console, 3-59

System parameters, 3-53, 5-14, 6-2
System programs, 5-2
- COMP, 5-13
- EDIT, 5-5
- PIP, 5-16
- SPUT, 6-1

T

Tables
- Communications, 7-1
- control driver, 7-20
- Format driver, 7-20
- STORAGE statement, 3-21

Tag, 3-17
- Activity, 3-47
- Control option, 3-30, 3-40
- FORMAT, 3-17, 3-22
- HEADER, 3-17, 3-22
- PHASE, 3-42
- SNAP, 3-43

Testing, 3-10
Time of day, 3-17
Timers, 3-49
- Delay (DELAY), 3-48, 3-49
- Interval (EVERY), 3-47, 3-49
- Time of day (AT), 3-48, 3-49

Titles, 3-17, 3-22
Trailer, 2-2, 4-3, 7-5
TRANSFER request, 3-57
TTY, 3-17, 3-59
TTY 2, 3-59
TTY 3, 3-59
TTY 4, 3-59

U

UDC, 3-30, 3-37
UDCE, 3-30, 3-37
UDCP, 3-37, 3-55
Unary operation, 7-27
User mode, 5-1

V

Variable, 3-3
- Integer, 3-24
- Real, 3-24

Vector code, 7-22
- DO/RETURN, 7-26
- Equates, 7-17
- FOR/NEXT, 7-27
- GBIN, 7-28
- GOTO, 7-25
- IF, 7-25
- I/O, request, 7-20
- LET, 7-23
- XTRP, 7-28

Vector mode, 7-22
Verify error, 2-3, 4-4

W

Word, computer, 3-3, 3-25, 7-22

X

XTRP vector, 7-28

< >

<CM> table, 4-2, 5-20, 7-2, J-1, K-1
<IF> table, 4-2, 5-20, 7-1, J-1, K-1
<PZ> table, 4-2, 5-20, 7-2, J-1
<SD> table, 4-2, 4-6, 5-2, 7-2, J-1
<XS> table, 4-2, 5-20, 7-1, J-1
<**> table, 4-2, 7-3, J-1

READER'S COMMENTS

Digital Equipment Corporation maintains a continuous effort to improve the quality and usefulness of its publications. To do this effectively we need user feedback – your critical evaluation of this manual.

Please comment on this manual's completeness, accuracy, organization, usability, and readability.

_____

_____

_____

_____

Did you find errors in this manual?

_____

_____

_____

_____

How can this manual be improved?

_____

_____

_____

_____

DEC also strives to keep its customers informed of current DEC software and publications. Thus, the following periodically distributed publications are available upon request. Please check the appropriate boxes for a current issue of the publication(s) desired.

☐ **Software Manual Update**, a quarterly collection of revisions to current software manuals.

☐ **User's Bookshelf**, a bibliography of current software manuals.

☐ **Program Library Price List**, a list of currently available software programs and manuals.

Please describe your position.

Name _____ Organization _____

Street _____ Department _____

City _____ State _____ Zip or Country _____

Fold Here

Do Not Tear - Fold Here and Staple

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

Postage will be paid by:

digital

Digital Equipment Corporation
Technical Documentation Department
146 Main Street
Maynard, Massachusetts 01754