

# PDP—11 DIAGNOSTIC DESIGN GUIDE

**For Internal Use Only**

**Do not remove from Digital Equipment Corporation property.**

**digital**



94-003/086/26

PDP-11 DIAGNOSTIC DESIGN GUIDE

Document Identifier: A-MN-ELENDIA-11-0, Rev. A  
20 January 1983

**ABSTRACT:** This document contains programming practices and standards for PDP-11/LSI-11 diagnostic programs. Pertinent information regarding operating environments, coding standards, design, and documentation is presented.

**APPLICABILITY:** This document is intended for diagnostic engineers and others who are developing diagnostic products for the PDP-11/LSI-11 family.

FOR INTERNAL USE ONLY  
DO NOT REMOVE FROM DIGITAL PROPERTY

TITLE: PDP-11 DIAGNOSTIC DESIGN GUIDE

DOCUMENT IDENTIFIER: A-MN-ELENDIA-11-0, Rev. A  
20-Jan-83

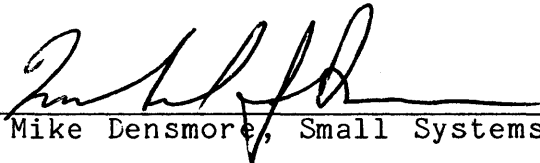
REVISION HISTORY: Initial Release

Responsible Department: Small Systems Diagnostics

Responsible Individual: Mike Densmore

Author: Compiled by Paul Niemi. Revised and expanded by: Mike Densmore, Tom Drake, Barry Irrgang, Bruce Luhrs, Joe Micali, Ron Parker, Bruce Ribolini, and Ray Shoop. Edited by Gunars Zagars.

ACCEPTANCE:

  
\_\_\_\_\_  
Mike Densmore, Small Systems Diagnostics Mgr.

Direct requests for further information to: Mike Densmore,  
MLO21-4/E20, DTN: 223-6162.

Copies of this document can be ordered from: Standards and Methods  
Control, MLO3-2/E56, DTN: 223-9475.



## PREFACE

This manual presents an overview of the PDP-11 diagnostic philosophy and procedures and an explanation of how to write diagnostic programs for the PDP-11 and LSI-11 families of computers. It is written for diagnostic engineers who are familiar with the PDP-11 hardware, operating system, Macro assembly language, and the hardware device to be tested. This manual can be used as an aid to diagnostic program development or as a reference for specific features of the PDP-11 Diagnostic Runtime Services and diagnostic macro library. It is not intended to be a self-teaching manual for a novice diagnostic engineer.

The manual consists of two parts. Part I (chapters 1 - 5) describes the PDP-11 diagnostic engineering philosophy. It deals with diagnostic goals, functions, methods, and the structure of the PDP-11 diagnostic operating system. Part II (chapters 6 - 11) presents system-wide guidelines, assisting in the development of diagnostic programs that will interface with the PDP-11 Diagnostic Runtime Services and that will conform to Diagnostic Engineering standards. The chapter titles are:

- Ch. 1 Diagnostic Users and Applications
- Ch. 2 Diagnostic Program Metrics
- Ch. 3 Diagnostic Strategy
- Ch. 4 Operating Modes and Capabilities
- Ch. 5 Diagnostic Development Process
- Ch. 6 XXDP+, the PDP-11 Diagnostic Operating System
- Ch. 7 DRS-Compatible Diagnostic Programs
- Ch. 8 Non-DRS Automated Environments
- Ch. 9 Structured Programming and Program Design Language 1
- Ch.10 Diagnostic Program Documentation
- Ch.11 General Coding Conventions

TABLE OF CONTENTS/REVISION STATUS

Subhead	Title	Page	Revision
	Title Page	i	20-Jan-83
	Document Management Page	ii	20-Jan-83
	Preface	iii	20-Jan-83
	Table of Contents/Revision Status	iv	20-Jan-83
<b>CHAPTER 1</b>	<b><u>DIAGNOSTIC USERS AND APPLICATIONS</u></b>		
1.1	DIAGNOSTIC USERS	1-1	20-Jan-83
1.1.1	Computer Design Engineers	1-1	20-Jan-83
1.1.2	Manufacturing Technicians	1-2	20-Jan-83
1.1.3	Field Service Engineers	1-2	20-Jan-83
1.1.4	End Users	1-3	20-Jan-83
1.2	DIAGNOSTIC APPLICATIONS	1-3	20-Jan-83
1.2.1	Local Operator Application	1-3	20-Jan-83
1.2.2	Automated Applications	1-4	20-Jan-83
1.2.2.1	Autotest	1-4	20-Jan-83
1.2.2.2	APT	1-4	20-Jan-83
1.2.2.3	APT-RD	1-5	20-Jan-83
1.2.2.4	ACT	1-5	20-Jan-83
1.2.2.5	XXDP+ Chain Mode	1-5	20-Jan-83
1.2.2.6	SLIDE	1-6	20-Jan-83
<b>CHAPTER 2</b>	<b><u>DIAGNOSTIC PROGRAM METRICS</u></b>		
2.1	FAULT DETECTION COVERAGE	2-1	20-Jan-83
2.2	FAULT ISOLATION AND TROUBLESHOOTING SUPPORT	2-1	20-Jan-83
2.2.1	Fault Isolation	2-2	20-Jan-83
2.2.2	Troubleshooting Support	2-2	20-Jan-83
2.3	DIAGNOSTIC PROGRAM SIZE	2-3	20-Jan-83
2.4	DIAGNOSTIC EXECUTION TIME	2-3	20-Jan-83
2.5	OPERATIONAL FUNCTIONALITY AND DOCUMENTATION	2-4	20-Jan-83
2.5.1	Test Mode Diagnostic Functionality	2-4	20-Jan-83
2.5.2	Troubleshooting and Repair Diagnostic Functionality	2-5	20-Jan-83
<b>CHAPTER 3</b>	<b><u>DIAGNOSTIC STRATEGY</u></b>		
3.1	SYSTEM CORE DIAGNOSTIC STRATEGY	3-1	20-Jan-83
3.1.1	System Core Definition	3-1	20-Jan-83
3.1.2	System Core Diagnostic Goals	3-2	20-Jan-83
3.1.3	System Core Diagnostic Implementation	3-2	20-Jan-83
3.2	CPU AND CPU OPTION DIAGNOSTIC STRATEGY	3-3	20-Jan-83
3.2.1	Hardcore Verification Tests	3-3	20-Jan-83
3.2.2	Basic CPU Cluster Tests	3-3	20-Jan-83
3.2.3	Extended CPU Cluster Tests	3-3	20-Jan-83

TABLE OF CONTENTS/REVISION STATUS

Subhead	Title	Page	Revision
3.2.4	Microdiagnostics	3-4	20-Jan-83
3.3	PERIPHERAL DIAGNOSTIC STRATEGY	3-5	20-Jan-83
3.3.1	General Requirements	3-5	20-Jan-83
3.3.2	Fault Detection and Isolation	3-6	20-Jan-83
3.4	SYSTEM EXERCISER DIAGNOSTIC STRATEGY	3-7	20-Jan-83
3.5	COMMUNICATIONS DIAGNOSTIC STRATEGY	3-8	20-Jan-83
3.5.1	Communications Strategy	3-9	20-Jan-83
3.5.2	Link Verification	3-9	20-Jan-83
3.5.3	System Performance	3-10	20-Jan-83
3.5.4	Standalone Communications Diagnostics	3-11	20-Jan-83
3.6	AUTOTEST DIAGNOSTIC STRATEGY	3-11	20-Jan-83
3.6.1	Commercial Maintenance Products Program	3-11	20-Jan-83
3.6.2	FTS-11	3-11	20-Jan-83
3.6.3	Auto Test Strategy For DECSA (Digital Ethernet Communications Service, ModelA)	3-11	20-Jan-83
CHAPTER 4	<u>OPERATING MODES AND CAPABILITIES</u>		
4.1	PROGRAM SELF IDENTIFICATION	4-1	20-Jan-83
4.2	TEST SELECTION CAPABILITY	4-1	20-Jan-83
4.3	PROGRAM EXECUTION MODES	4-2	20-Jan-83
4.3.1	Continue On Error	4-2	20-Jan-83
4.3.2	Halt On Error	4-2	20-Jan-83
4.3.3	Loop On Error	4-3	20-Jan-83
4.3.4	One Error Report Per Sub-Test	4-3	20-Jan-83
4.3.5	Inhibit Error Reports	4-3	20-Jan-83
4.3.6	Inhibit Progress Reports	4-3	20-Jan-83
4.3.7	Error Logging	4-3	20-Jan-83
4.3.8	Signal (Bell) On Error	4-4	20-Jan-83
4.4	INTERACTIVE PROGRAM EXECUTION	4-4	20-Jan-83
4.5	BASIC FUNCTIONAL TESTING AND RELIABILITY MODE	4-4	20-Jan-83
4.5.1	Basic Functional Testing	4-5	20-Jan-83
4.5.2	Reliability Mode	4-5	20-Jan-83
4.6	DEFAULTING PHILOSOPHY	4-5	20-Jan-83
4.7	SPECIAL OPERATING MODES	4-5	20-Jan-83
4.8	CHAIN MODE	4-6	20-Jan-83
CHAPTER 5	<u>DIAGNOSTIC DEVELOPMENT PROCESS</u>		
5.1	CONSULTATION PHASE (PHASE 0)	5-1	20-Jan-83
5.2	PLANNING PHASE (PHASE 1)	5-2	20-Jan-83
5.2.1	Diagnostic Project Plan	5-2	20-Jan-83
5.2.2	Diagnostic Functional Specification	5-3	20-Jan-83
5.2.3	Diagnostic Program Design Specification	5-3	20-Jan-83
5.3	IMPLEMENTATION PHASE (PHASE 2)	5-4	20-Jan-83

TABLE OF CONTENTS/REVISION STATUS

Subhead	Title	Page	Revision
5.3.1	Engineering Breadboard and Prototype Support	5-4	20-Jan-83
5.3.2	Final Diagnostic Implementation	5-5	20-Jan-83
5.4	DIAGNOSTIC QUALITY ASSURANCE AND RELEASE PHASE (PHASE 3A)	5-6	20-Jan-83
5.4.1	Introduction	5-6	20-Jan-83
5.4.2	Design Reviews	5-7	20-Jan-83
5.4.3	Manufacturing Installation	5-7	20-Jan-83
5.4.4	Performance Feedback	5-8	20-Jan-83
5.4.5	Configuration Compatibility Testing	5-8	20-Jan-83
5.4.6	XXDP+/ACT/APT/SLIDE Compatibility Testing	5-8	20-Jan-83
5.4.7	Diagnostic Program Verification	5-9	20-Jan-83
5.4.8	Quality Assurance Checklist	5-9	20-Jan-83
5.5	MAINTENANCE	5-9	20-Jan-83
CHAPTER 6	<u>XXDP+, THE PDP-11 DIAGNOSTIC OPERATING SYSTEM</u>		
6.1	INTRODUCTION	6-1	20-Jan-83
6.1.1	XXDP+ Monitor	6-1	20-Jan-83
6.1.2	Diagnostic Runtime Services	6-1	20-Jan-83
6.1.3	Utility Programs	6-2	20-Jan-83
6.1.4	Device Drivers	6-2	20-Jan-83
6.1.5	XXDP+ Nomenclature	6-2	20-Jan-83
6.1.5.1	Software Naming Conventions	6-2	20-Jan-83
6.1.5.2	File Naming Conventions	6-3	20-Jan-83
6.2	XXDP+ CONSTRUCTION	6-4	20-Jan-83
6.2.1	XXDP+ Monitor	6-4	20-Jan-83
6.2.2	Diagnostic Runtime Services	6-5	20-Jan-83
6.2.3	XXDP+ Utility Programs	6-5	20-Jan-83
6.2.3.1	UPD2	6-6	20-Jan-83
6.2.3.2	UPD1	6-6	20-Jan-83
6.2.3.3	PATCH	6-6	20-Jan-83
6.2.3.4	SETUP	6-6	20-Jan-83
6.2.3.5	XTECO	6-6	20-Jan-83
6.2.4	XXDP+ Device Drivers	6-7	20-Jan-83
6.2.5	Building XXDP+	6-8	20-Jan-83
6.3	BATCH CONTROL (CHAINING)	6-8	20-Jan-83
6.3.1	Batch Control of Diagnostics	6-9	20-Jan-83
6.3.2	Batch Control of Utilities	6-10	20-Jan-83
6.4	XXDP+ COMMANDS	6-10	20-Jan-83
6.4.1	Monitor Commands	6-10	20-Jan-83
6.4.1.1	Load Command	6-11	20-Jan-83
6.4.1.2	Start Command	6-11	20-Jan-83
6.4.1.3	Run Command	6-12	20-Jan-83
6.4.1.4	Chain Command	6-12	20-Jan-83
6.4.1.5	Directory Command	6-13	20-Jan-83
6.4.1.6	Fill Command	6-14	20-Jan-83

TABLE OF CONTENTS/REVISION STATUS

Subhead	Title	Page	Revision
6.4.1.7	Enable Command	6-14	20-Jan-83
6.4.1.8	Help Command	6-15	20-Jan-83
6.4.1.9	Test Command	6-15	20-Jan-83
6.4.2	DRS Commands	6-15	20-Jan-83
6.4.2.1	STA[RT] Command	6-16	20-Jan-83
6.4.2.2	RES[TART] Command	6-16	20-Jan-83
6.4.2.3	CON[TINUE] Command	6-17	20-Jan-83
6.4.2.4	PRO[CEED] Command	6-17	20-Jan-83
6.4.2.5	DRO[P] Command	6-18	20-Jan-83
6.4.2.6	ADD Command	6-19	20-Jan-83
6.4.2.7	DIS[PLAY] Command	6-19	20-Jan-83
6.4.2.8	FLA[GS] Command	6-19	20-Jan-83
6.4.2.9	ZFL[AGS] Command	6-20	20-Jan-83
6.4.2.10	PRI[NT] Command	6-20	20-Jan-83
6.4.2.11	EXI[T] Command	6-20	20-Jan-83
6.4.3	DRS Switches	6-21	20-Jan-83
6.4.3.1	TES[TS] Switch	6-21	20-Jan-83
6.4.3.2	PAS[S] Switch	6-22	20-Jan-83
6.4.3.3	FLA[GS] Switch	6-22	20-Jan-83
6.4.3.4	EOP Switch	6-22	20-Jan-83
6.4.3.5	UNI[TS] Switch	6-22	20-Jan-83
6.4.3.6	Combining Switches	6-23	20-Jan-83
6.4.4	DRS Flags	6-23	20-Jan-83
6.4.4.1	HOE (Halt On Error) Flag	6-24	20-Jan-83
6.4.4.2	LOE (Loop On Error) Flag	6-25	20-Jan-83
6.4.4.3	IER (Inhibit Error Reports) Flag	6-25	20-Jan-83
6.4.4.4	IBE (Inhibit Basic Errors) Flag	6-25	20-Jan-83
6.4.4.5	IXE (Inhibit Extended Errors) Flag	6-25	20-Jan-83
6.4.4.6	PRI (Printer) Flag	6-25	20-Jan-83
6.4.4.7	PNT (Print Number of Test) Flag	6-25	20-Jan-83
6.4.4.8	BOE (Bell On Error) Flag	6-26	20-Jan-83
6.4.4.9	UAM (Unattended Mode) Flag	6-26	20-Jan-83
6.4.4.10	ISR (Inhibit Statistical Reports) Flag	6-26	20-Jan-83
6.4.4.11	IDR (Inhibit DRopping of Units) Flag	6-26	20-Jan-83
6.4.4.12	ADR (Auto DRop) Flag	6-26	20-Jan-83
6.4.4.13	LOT (Loop On Test) Flag	6-27	20-Jan-83
6.4.5	XXDP+ Utility Commands	6-28	20-Jan-83
6.4.5.1	UPD2 Command Summary	6-28	20-Jan-83
6.4.5.2	UPD1 Command Summary	6-29	20-Jan-83
6.4.5.3	PATCH Command Summary	6-29	20-Jan-83
6.4.5.4	SETUP Command Summary	6-29	20-Jan-83
6.4.5.5	XTECO Command Summary	6-30	20-Jan-83

CHAPTER 7 DRS-COMPATIBLE DIAGNOSTIC PROGRAMS

7.1	INTRODUCTION	7-1	20-Jan-83
7.2	DRS PROGRAM BASICS	7-1	20-Jan-83

TABLE OF CONTENTS/REVISION STATUS

Subhead	Title	Page	Revision
7.2.1	Memory Layout	7-1	20-Jan-83
7.2.2	Different Operating Environment Versions	7-2	20-Jan-83
7.2.3	Interfacing to the Environment	7-2	20-Jan-83
7.2.3.1	Operator Commands	7-2	20-Jan-83
7.2.3.2	Switches	7-4	20-Jan-83
7.2.3.3	Hardware Parameterization	7-4	20-Jan-83
7.2.3.4	Software Parameterization	7-5	20-Jan-83
7.2.3.5	Passes and Sub-Passes	7-5	20-Jan-83
7.3	DRS PROGRAM STRUCTURE	7-5	20-Jan-83
7.3.1	Program Header (Required)	7-6	20-Jan-83
7.3.2	Dispatch Table (Required)	7-6	20-Jan-83
7.3.3	Default Hardware P-Table (Required)	7-6	20-Jan-83
7.3.4	Software P-Table (Optional)	7-6	20-Jan-83
7.3.5	Global Equates (Optional)	7-6	20-Jan-83
7.3.6	Global Data (Optional)	7-7	20-Jan-83
7.3.7	Global Text (Optional)	7-7	20-Jan-83
7.3.8	Global Error Reports (Optional)	7-7	20-Jan-83
7.3.9	Global Subroutines (Optional)	7-7	20-Jan-83
7.3.10	Statistical Report Coding (Optional)	7-7	20-Jan-83
7.3.11	Initialization Coding (Required)	7-7	20-Jan-83
7.3.12	Clean-Up Coding (Required)	7-8	20-Jan-83
7.3.13	Drop Units Coding (Optional)	7-8	20-Jan-83
7.3.14	Add Units Coding (Optional)	7-8	20-Jan-83
7.3.15	Hardware Tests (Required)	7-8	20-Jan-83
7.3.16	Hardware Parameter Coding (Required)	7-8	20-Jan-83
7.3.17	Software Parameter Coding (Optional)	7-9	20-Jan-83
7.4	DRS PROGRAM STRUCTURE MACROS	7-9	20-Jan-83
7.4.1	Optional Sections Selection (POINTER)	7-9	20-Jan-83
7.4.2	Header Call (HEADER)	7-9	20-Jan-83
7.4.3	Descriptive Text (DESCRIPT, DEVTYPE)	7-10	20-Jan-83
7.4.4	Last Address Generation (LASTAD)	7-10	20-Jan-83
7.4.5	Module Delimiters (BGNMOD, ENDMOD)	7-10	20-Jan-83
7.4.6	Test Delimiters (BGNTST, ENDTST)	7-11	20-Jan-83
7.4.7	Subtest Delimiters (BGNSUB, ENDSUB)	7-11	20-Jan-83
7.4.8	Segment Delimiters (BGNSEG, ENDSEG)	7-11	20-Jan-83
7.4.9	Hardcoded P-Tables	7-12	20-Jan-83
7.5	DRS SERVICE MACROS	7-12	20-Jan-83
7.5.1	Macro Package Initialization (SVC)	7-12	20-Jan-83
7.5.2	Global Equates (EQUALS)	7-12	20-Jan-83
7.5.3	Test Dispatch Table (DISPATCH)	7-13	20-Jan-83
7.5.4	Error Loop Control (CKLOOP)	7-13	20-Jan-83
7.5.4.1	Implied CKLOOP	7-13	20-Jan-83
7.5.4.2	Explicit CKLOOP	7-13	20-Jan-83
7.5.5	Error Loop Detection (INLOOP)	7-14	20-Jan-83
7.5.6	Abort Test Calls (ESCAPE, EXIT)	7-15	20-Jan-83
7.5.6.1	Escape Test (ESCAPE TST, SUB, SEG)	7-15	20-Jan-83
7.5.6.2	Exit Test (EXIT TST, SUB, SEG)	7-15	20-Jan-83

TABLE OF CONTENTS/REVISION STATUS

Subhead	Title	Page	Revision
7.5.6.3	Exit Routine (EXIT HRD, SFT, INIT, CLN, SRV, MSG)	7-15	20-Jan-83
7.5.7	Error Reporting (ERRSF, ERRDF, ERRHRD, ERRSOFT, ERROR, ERRTBL)	7-16	20-Jan-83
7.5.7.1	Error Report Classes	7-16	20-Jan-83
7.5.7.2	Report Call Arguments	7-17	20-Jan-83
7.5.7.3	Error Tables	7-17	20-Jan-83
7.5.8	Printing Messages (BGNMSG, ENDMSG, PRINTB, PRINTX, PRINTF)	7-18	20-Jan-83
7.5.8.1	Message Printout Format	7-18	20-Jan-83
7.5.8.2	Begin and End Message Calls (BGNMSG, ENDMSG)	7-19	20-Jan-83
7.5.8.3	Basic and Extended Print Message Calls (PRINTB, PRINTX, PRINTF)	7-19	20-Jan-83
7.5.9	Statistical Reporting (BGNRPT, ENDRPT, PRINTS, DORPT)	7-21	20-Jan-83
7.5.10	Branching (BERROR, BNERROR, BCOMPLETE, BNCOMPLETE)	7-21	20-Jan-83
7.5.11	Clock Macro	7-22	20-Jan-83
7.5.12	Event Flags (READEF)	7-23	20-Jan-83
7.5.13	Unit Selection (BGNAU, ENDAU, BGNDU, DODU, ENDDU)	7-23	20-Jan-83
7.5.13.1	Adding Units (BGNAU, ENDAU)	7-24	20-Jan-83
7.5.13.2	Dropping Units (BGNDU, ENDDU, DODU)	7-24	20-Jan-83
7.5.14	Default Hardware P-Table (BGNHW, ENDHW)	7-24	20-Jan-83
7.5.15	Software P-Table (BGNSW, ENDSW)	7-25	20-Jan-83
7.5.16	Hardware P-Table Questions (BGNHRD, ENDHRD)	7-25	20-Jan-83
7.5.17	Software P-Table Questions (BGNSFT, ENDSFT)	7-26	20-Jan-83
7.5.18	Parameter Coding Calls (GPRMD, GPRMA, GPRML)	7-26	20-Jan-83
7.5.18.1	GPRMD Call - Data	7-26	20-Jan-83
7.5.18.2	GPRMA Call - Address	7-28	20-Jan-83
7.5.18.3	GPRML Call - Logical	7-29	20-Jan-83
7.5.18.4	COUNT MACRO (COUNT arg)	7-29	20-Jan-83
7.5.18.5	DISPLAY Macro (DISPLAY Arg)	7-29	20-Jan-83
7.5.19	Transfer Calls (XFER)	7-29	20-Jan-83
7.5.20	Request Table (GPHARD)	7-31	20-Jan-83
7.5.21	Initialization (BGNINIT, ENDINIT)	7-31	20-Jan-83
7.5.22	Clean-Up Code (BGNCLN, ENDCLN, DOCLN)	7-32	20-Jan-83
7.5.23	Is Manual Intervention Allowed? (MANUAL)	7-33	20-Jan-83
7.5.24	Get Manual Parameters (GMANID, GMANIA, GMANIL)	7-33	20-Jan-83
7.5.24.1	GMANID Call	7-34	20-Jan-83
7.5.24.2	GMANIA Call	7-35	20-Jan-83
7.5.24.3	GMANIL Call	7-35	20-Jan-83

TABLE OF CONTENTS/REVISION STATUS

Subhead	Title	Page	Revision
7.5.25	Operator Interrupt Enable (BREAK)	7-35	20-Jan-83
7.5.26	Bus Reset (BRESET)	7-36	20-Jan-83
7.5.27	Memory Allocation (MEMORY)	7-36	20-Jan-83
7.5.	Interrupt Handling (SETVEC, CLRVEC, BGNSRV, ENDSRV)	7-36	20-Jan-83
7.5.29	Documentation Aids	7-37	20-Jan-83
7.5.29.1	Left Justified Graphics (COMMENT, ENDCOMMENT)	7-37	20-Jan-83
7.5.29.2	Right Justified Graphics (SLASH, STARS)	7-38	20-Jan-83
7.5.30	Program Priority (SETPRI, GETPRI)	7-38	20-Jan-83
7.5.31	Bus Type Check (READBUS)	7-38	20-Jan-83
7.5.32	Load Device Protection	7-39	20-Jan-83
7.5.33	File Control Services	7-39	20-Jan-83
7.5.34	Access to Flags	7-40	20-Jan-83
7.5.35	Autodrop Section	7-40	20-Jan-83
7.6	SAMPLE DIAGNOSTIC	7-41	20-Jan-83

CHAPTER 8 NON-DRS AUTOMATED ENVIRONMENTS

8.1	INTRODUCTION	8-1	20-Jan-83
8.2	AUTOMATED PRODUCT TEST (APT-11)	8-1	20-Jan-83
8.2.1	Introduction	8-1	20-Jan-83
8.2.2	APT Mailbox	8-2	20-Jan-83
8.2.3	APT Mailbox Fields	8-2	20-Jan-83
8.2.3.1	Message Type Code, Word 1 (SYSMAC/\$MSGTY)	8-2	20-Jan-83
8.2.3.2	Fatal Error Number, Word 2 (SYSMAC/\$FATAL)	8-3	20-Jan-83
8.2.3.3	Test Number, Word 3 (SYSMAC/\$TESTN)	8-4	20-Jan-83
8.2.3.4	Pass Count, Word 4 (SYSMAC/\$PASS)	8-4	20-Jan-83
8.2.3.5	Device Count, Word 5 (SYSMAC/\$DEVCT)	8-5	20-Jan-83
8.2.3.6	Unit, Word 6 (SYSMAC/\$UNIT)	8-5	20-Jan-83
8.2.3.7	Message Address, Word 7 (SYSMAC/\$MSGAD)	8-5	20-Jan-83
8.2.3.8	Message Length, Word 8 (SYSMAC/\$MSGLG)	8-6	20-Jan-83
8.3	AUTOMATIC COMPUTER TEST (ACT-11) SYSTEM	8-6	20-Jan-83
8.3.1	ACT Dump Mode	8-6	20-Jan-83
8.3.2	ACT Auto-accept Mode	8-6	20-Jan-83
8.3.3	ACT Station Test Mode	8-7	20-Jan-83
8.4	SERIAL LOADER IN DEMAND EVERYWHERE (SLIDE)	8-7	20-Jan-83
8.4.1	The SLIDE System	8-7	20-Jan-83
8.4.2	SLIDE Basic Software	8-8	20-Jan-83
8.4.2.1	Central Computer Memory Usage	8-9	20-Jan-83
8.4.2.2	Test Station Memory Usage	8-9	20-Jan-83
8.4.3	Using SLIDE	8-10	20-Jan-83
8.4.4	Obtaining a Directory	8-10	20-Jan-83
8.4.5	Time and Date Messages	8-11	20-Jan-83



TABLE OF CONTENTS/REVISION STATUS

Subhead	Title	Page	Revision
8.4.6	Chain Mode Operation	8-11	20-Jan-83
8.4.6.1	Making a Chain	8-12	20-Jan-83
8.4.6.2	Considerations when Making a Chain	8-13	20-Jan-83
8.4.7	Watchdog Timeout Feature	8-15	20-Jan-83
8.4.7.1	Watchdog Timer Commands	8-15	20-Jan-83
8.4.7.2	Using the Watchdog Timer	8-15	20-Jan-83
8.4.8	Issuing Commands to Another Terminal or Line Printer	8-17	20-Jan-83
8.4.9	Updating and Patching	8-17	20-Jan-83
8.4.10	SLIDE Help Commands	8-18	20-Jan-83
8.5	MACRO SUMMARY	8-19	20-Jan-83
8.5.1	Mandatory, Direct Support Macros	8-19	20-Jan-83
8.5.1.1	.\$ACT11 Macro	8-19	20-Jan-83
8.5.1.2	.\$APTHDR Macro	8-19	20-Jan-83
8.5.1.3	.\$APTBL5 Macro	8-20	20-Jan-83
8.5.1.4	.\$APTYPE Macro	8-20	20-Jan-83
8.5.1.5	REPORT Macro	8-20	20-Jan-83
8.5.1.6	.\$APTAT Macro	8-21	20-Jan-83
8.5.2	Indirect Support Macros	8-21	20-Jan-83
8.5.2.1	.\$EOP Macro	8-21	20-Jan-83
8.5.2.2	.\$CMTAG Macro	8-21	20-Jan-83
8.5.2.3	.EQUAT Macro	8-22	20-Jan-83
8.5.2.4	SETUP Macro	8-22	20-Jan-83
8.5.2.5	.NEWTST Macro	8-22	20-Jan-83
8.5.2.6	.\$SCOPE Macro	8-22	20-Jan-83
8.5.2.7	.\$ERROR Macro	8-22	20-Jan-83
8.5.2.8	.\$TYPE Macro	8-23	20-Jan-83
8.5.2.9	.TYPNAM Macro	8-23	20-Jan-83
8.5.3	Other Support Macros	8-23	20-Jan-83
8.6	SYSMAC.SML, THE DIAGNOSTIC MACRO LIBRARY	8-23	20-Jan-83
8.6.1	Definition Macros	8-24	20-Jan-83
8.6.2	In-line Code Macros	8-24	20-Jan-83
8.6.3	Handler Macros	8-25	20-Jan-83
CHAPTER 9	<u>STRUCTURED PROGRAMMING</u>		
9.1	INTRODUCTION	9-1	20-Jan-83
9.2	PROGRAMMING CONSIDERATIONS	9-2	20-Jan-83
9.2.1	Structured Design	9-2	20-Jan-83
9.2.2	Structured Programming	9-2	20-Jan-83
9.2.3	Module Structure	9-4	20-Jan-83
9.3	USING BLISS	9-5	20-Jan-83
9.4	USING PASCAL FOR SPECIFICATION AND DESIGN	9-5	20-Jan-83
9.5	USING BASIC TO WRITE DIAGNOSTICS	9-7	20-Jan-83
9.6	PROGRAM DESIGN LANGUAGE 1	9-7	20-Jan-83
9.6.1	Introduction	9-7	20-Jan-83
9.6.2	Purpose and History	9-8	20-Jan-83

TABLE OF CONTENTS/REVISION STATUS

Subhead	Title	Page	Revision
9.6.3	Guidelines	9-9	20-Jan-83
9.6.3.1	Design Guidelines	9-9	20-Jan-83
9.6.3.2	Guidelines for Translating PDL1 To Source Code	9-10	20-Jan-83
9.6.4	PDL1 Format	9-10	20-Jan-83
9.6.4.1	General Format	9-10	20-Jan-83
9.6.4.2	Block Structure	9-10	20-Jan-83
9.6.4.2.1	Sequential Blocks	9-11	20-Jan-83
9.6.4.2.2	Selective Blocks	9-11	20-Jan-83
9.6.4.2.3	Iterative Blocks	9-11	20-Jan-83
9.6.4.3	Internal Block Structure	9-11	20-Jan-83
9.6.4.3.1	Imperatives	9-11	20-Jan-83
9.6.4.3.2	The Underline Character	9-13	20-Jan-83
9.6.4.3.3	Assignment and Relational Operators	9-13	20-Jan-83
9.6.4.3.4	Parentheses and Brackets	9-13	20-Jan-83
9.6.4.3.5	Early Exit	9-14	20-Jan-83
9.6.4.4	Keywords	9-14	20-Jan-83
9.6.4.5	Block Structuring	9-15	20-Jan-83
9.6.4.6	Other Constructs	9-16	20-Jan-83
9.6.4.7	Example of a Program in PDL1	9-17	20-Jan-83
9.6.4.8	PDL1 Keywords in Alphabetical Order	9-18	20-Jan-83
CHAPTER	<u>10 DIAGNOSTIC PROGRAM DOCUMENTATION</u>		
10.1	DOCUMENTATION GUIDELINES	10-2	20-Jan-83
10.2	DOCUMENTATION SECTION	10-2	20-Jan-83
10.2.1	Documentation Cover Sheet	10-3	20-Jan-83
10.2.2	History Section	10-4	20-Jan-83
10.2.3	Table of Contents	10-5	20-Jan-83
10.3	PROGRAM ABSTRACT	10-6	20-Jan-83
10.4	SYSTEM REQUIREMENTS	10-6	20-Jan-83
10.5	RELATED DOCUMENTS AND STANDARDS	10-7	20-Jan-83
10.6	DIAGNOSTIC PREREQUISITES	10-7	20-Jan-83
10.7	PROGRAM ASSUMPTIONS	10-7	20-Jan-83
10.8	OPERATING INSTRUCTIONS	10-7	20-Jan-83
10.8.1	Loading and Starting Procedures	10-9	20-Jan-83
10.8.2	Special Environments	10-9	20-Jan-83
10.8.3	Program Options	10-9	20-Jan-83
10.8.4	Execution Times	10-9	20-Jan-83
10.9	ERROR INFORMATION	10-10	20-Jan-83
10.9.1	Error Reporting Procedures	10-10	20-Jan-83
10.9.2	Error Halts	10-12	20-Jan-83
10.10	OPTIONAL PERFORMANCE AND PROGRESS REPORTS	10-12	20-Jan-83
10.10.1	Performance Reports	10-12	20-Jan-83
10.10.2	Progress Reports	10-12	20-Jan-83
10.11	SUB-TEST SUMMARIES	10-13	20-Jan-83

TABLE OF CONTENTS/REVISION STATUS

Subhead	Title	Page	Revision
10.12	PROGRAM LISTING	10-13	20-Jan-83
10.13	SYMBOL TABLE AND AND CROSS REFERENCE LISTING	10-14	20-Jan-83
10.14	PROGRAM FUNCTIONAL DESCRIPTION	10-14	20-Jan-83
10.15	DESCRIPTIONS OF SUBROUTINES	10-16	20-Jan-83
CHAPTER 11	<u>GENERAL CODING CONVENTIONS</u>		
11.1	INTRODUCTION	11-1	20-Jan-83
11.2	RECOMMENDED CODING PRACTICE	11-3	20-Jan-83
11.2.1	Line Format	11-3	20-Jan-83
11.2.2	Comments	11-3	20-Jan-83
11.3	NAMING STANDARDS	11-4	20-Jan-83
11.3.1	Hardware Registers	11-4	20-Jan-83
11.3.2	Device Registers	11-4	20-Jan-83
11.3.3	General Purpose Registers	11-4	20-Jan-83
11.3.4	Processor Priority	11-5	20-Jan-83
11.3.5	Other Symbols	11-5	20-Jan-83
11.3.6	Program-Local Labels	11-5	20-Jan-83
11.4	PROGRAM MODULES	11-5	20-Jan-83
11.4.1	The Program Preface	11-6	20-Jan-83
11.4.2	Register Conventions	11-7	20-Jan-83
11.4.3	Argument Passing	11-7	20-Jan-83
11.4.4	Exiting	11-7	20-Jan-83
11.5	FORMATTING STANDARDS	11-8	20-Jan-83
11.5.1	Program Flow	11-8	20-Jan-83
11.6	FORBIDDEN INSTRUCTION USAGE	11-9	20-Jan-83
11.6.1	Instructions or Index Words as Literals	11-9	20-Jan-83
11.6.2	MOV Instead of JMP	11-9	20-Jan-83
11.6.3	Single-Word Instructions	11-10	20-Jan-83
11.6.4	PDP-11 Family Instruction Execution Differences	11-10	20-Jan-83
11.7	RECOMMENDED CODING PRACTICE - CONDITIONAL BRANCHES	11-11	20-Jan-83
	GLOSSARY	G-1	20-Jan-83
	INDEX	I-1	20-Jan-83



## CHAPTER 1

### DIAGNOSTIC USERS AND APPLICATIONS

This chapter describes the purposes of diagnostic programs for primary users and applications in the DIGITAL environment. An attempt is made to introduce the requirements placed on diagnostics by their users and applications.

#### 1.1 DIAGNOSTIC USERS

Diagnostic programs are used by computer design engineers, manufacturing technicians, field service engineers, and end users or customers. The common denominator of diagnostic users is their requirement for excellent fault detection coverage. Requirements concerning other diagnostic metrics such as program size, run-time, fault isolation, troubleshooting support, and operational documentation will vary with users and applications. Diagnostic users include:

- o Computer Design Engineers
- o Manufacturing Technicians
- o Field Service Engineers
- o End Users

##### 1.1.1 Computer Design Engineers

Computer design engineers rely on design verification test programs to detect functional or design implementation mistakes early in the hardware development phase. Fault (mistake) detection is their main concern. Design engineers have little or no concern for program size, run-time, fault isolation, troubleshooting support, or operational documentation. But poor or incomplete design verification test coverage (mistake detection) can result in costly Engineering Change Orders (ECOs) affecting manufacturing inventories, installed systems, and/or missed development schedules.

### 1.1.2 Manufacturing Technicians

Manufacturing technicians use diagnostics at several levels of the hardware test and repair processes. Diagnostic programs are used to screen (for defects) modules arriving from the module build process. This application requires excellent fault coverage but is usually sensitive to program run-time, thus forcing some design trade-offs between exhaustive testing and acceptable time-to-test. Fault isolation and troubleshooting support is generally not required in module screen diagnostics, since module repair is usually performed at a special purpose repair station utilizing repair tools (e.g., GenRad tester or microdiagnostics). Also, diagnostic operational documentation is not heavily emphasized because the module screen process is generally automated with the details of diagnostic execution and control masked from the technicians.

A second area of manufacturing diagnostic use is unit or system test, where central processing units, memory systems, input/output channels, and peripherals are tested either as components or as newly integrated systems. As in module screening, excellent fault detection coverage is required to minimize the number of faults slipping through to later system tests (possibly utilizing operating system software) or customer applications. Diagnostic programs used for unit or system test do not have the severe size and run-time constraints associated with the module screening diagnostics. However, unit and system test diagnostics must provide effective fault isolation and troubleshooting support, since repair is sometimes performed on-line, that is, at the time that the problem is detected. Diagnostic operational documentation becomes more important in this application because the technicians are directly involved with diagnostic execution and control. Technicians also must deal with a wide variety of hardware options; hence, a wide variety of diagnostic programs is needed.

### 1.1.3 Field Service Engineers

Field service engineers use diagnostic programs to install, maintain, and repair computer systems in countless configurations running countless applications. The goal should be isolation to the Field Replaceable Unit (FRU). Their diagnostic requirements may include the full spectrum of metrics: fault detection, fault isolation, troubleshooting support, and effective diagnostic operational documentation. The need for excellent fault detection coverage, fault isolation, and troubleshooting support is probably obvious from the repair objective of the field service engineer's task. The need for simple, effective diagnostic operational documentation is based on the variety and complexity of the systems that field service engineers support. Often the field service engineer is required to isolate and repair faults in equipment on which he or she has received little or no recent training. To further complicate the task, details of equipment configuration and options will seldom be known to the field

service engineer and, therefore, should not be required in order to execute the diagnostic programs. Default diagnostic test scripts are key elements in the PDP-11 diagnostic operational effectiveness goal. Several diagnostic metrics (such as program partitioning and run-time parameter definition) are heavily driven to achieve the diagnostic operational goals.

#### 1.1.4 End Users

End users use diagnostic programs in different configurations running various applications. The end user is concerned in determining whether the fault lies in the application being run or in the hardware running the application.

### 1.2 DIAGNOSTIC APPLICATIONS

Often, diagnostic programs are used in applications or processes that are quite independent of the ultimate test and repair mission. These applications impose requirements or constraints on the diagnostic programs which, in some cases, conflict with test and repair considerations. Since the ultimate effectiveness of a diagnostic program is a result of both mission effectiveness and process effectiveness, both sets of requirements must be addressed and effective compromise solutions engineered.

#### 1.2.1 Local Operator Application

The traditional and probably most important application for diagnostic programs is local operator controlled and directed testing, fault isolation, and repair verification. A major percentage of the PDP-11 Diagnostic Supervisor command functionality and the major diagnostic test design and documentation efforts are directed toward local operator effectiveness. Diagnostic scripting, ability to preconfigure diagnostic data structures, and default unit testing are examples of local operator test effectiveness tools.

Halt and loop-on-error control, multilevel error reporting, summary test reports, field replaceable unit callout, and listing troubleshooting documentation are examples of local operator fault isolation and repair effectiveness tools. Effective diagnostic programs must be designed and implemented to achieve excellence in test and repair support, operator ease of use and control.

## 1.2.2 Automated Applications

Over the past few years, diagnostic programs have been used in automated, often centrally controlled, applications. Automated diagnostic operation consists of the execution of predefined sequences, or scripts, of diagnostic programs. The scripting can be via local command files packaged on the diagnostic media and processed by XXDP+ or remote command files that are processed by the remote computer, or diagnostic host, and supplied to the diagnostic supervisor via a serial communication link. In the local script case, the diagnostic programs are usually loaded directly from the same local media, although there is at least one PDP-11 diagnostic application in which a local script requests program loads from the remote host. In remote script applications, the diagnostics can be loaded from the local diagnostic media or down-line loaded from the host via the serial communication link.

Automated diagnostic applications, whether locally or remotely controlled, have a definite impact on diagnostic design and packaging (see Chapter 7).

### 1.2.2.1 Autotest

There are many Automatic-Test Products within our sphere of influence. This Automatic-Test Philosophy is dealing with the functions that are designed to provide ease of use, improved time to diagnose, and software control of each device designed diagnostic test-sequence, enabling the ability to link each device diagnostic test sequence through software control to provide specific system automatic-test.

1.2.2.2 APT - APT is the acronym for an Automated Product Test application used throughout DIGITAL Manufacturing. APT employs remote diagnostic scripting with down-line diagnostic program load. Once APT loads a diagnostic program and starts diagnostic execution, it performs all monitoring and control functions (end of pass, error status collection) via an APT-unique software interface and protocol implemented in the diagnostic supervisor.

APT is sensitive to diagnostic operator intervention requirements, and to diagnostic program size and down-line load time. Diagnostic operator intervention, whether for configuration information or for hardware option information, is generally unacceptable to the APT application because of the need to create a finite set of test scripts that can be applied to a wide set of possible system configurations and hardware options.

Diagnostic program size and load-time considerations are obvious in time sensitive test processes. Although arbitrary program size reduction could reduce diagnostic fault coverage, thoughtful program partitioning (to allow selective hardware testing) and avoidance of verbose error and status messages (ASCII text) can benefit the diagnostic in an APT application.



1.2.2.3 APT-RD - APT-RD is an automated diagnostic control application utilized by DIGITAL field service to provide contract customers with quick response and effective on-site repair action. APT-RD becomes involved shortly after a customer requests a service call, by establishing a phone connection with the target system and initiating a diagnostic test session prior to the dispatching of a field service engineer. APT-RD effects Remote Diagnostic control by issuing diagnostic command sequences, via the phone link, to load (from local customer-mounted diagnostic media) and execute the appropriate diagnostics. Unlike APT, APT-RD will down-line load diagnostics only in rare situations (such as inability to boot or load from the local diagnostic media). APT-RD scripts use standard commands and key on ASCII message output (from the individual diagnostic programs) for all monitoring and control functions. As an application, APT-RD is extremely sensitive to the details of the supervisor and diagnostic program command and response messages. Also, as with APT, APT-RD is sensitive to diagnostic operator intervention requirements and, to a lesser degree, diagnostic program size and load time. Essentially the same diagnostic design considerations that are important to meet APT application requirements (program partitioning, no mandatory operator run-time intervention) are required for APT-RD. In addition, APT-RD requires well-defined, documented, and enforced (from program to program and version to version) command and message standards and implementation.

1.2.2.4 ACT - The Automated Computer Test system (ACT-11) provides three basic services that aid DIGITAL's manufacturing areas in testing PDP-11 computers.

1. Load and run a diagnostic into a Unit Under Test (UUT), as if it were run manually - called ACT "dump" mode.
2. Automatically load, run, and monitor a single diagnostic or sequence of diagnostics through one or more iterations - called ACT "auto-accept" mode; includes "quick verify" mode.
3. Directly perform a variety of UUT memory tests - called ACT "station test" mode.

1.2.2.5 XXDP+ Chain Mode - XXDP+, the diagnostic operating system for PDP-11s, consists of the monitor, diagnostic runtime services, utility programs, and loadable device drivers for the utilities. It has the facility to run programs without operator intervention, called chaining (see Chapter 5).

1.2.2.6 SLIDE - The "serial-line loader in demand everywhere" (SLIDE) system is used extensively in peripheral manufacturing areas to check out newly manufactured equipment. A SLIDE system consists of a PDP-11 computer system that communicates with the UUTs via serial asynchronous lines. The UUTs consist of a CPU that is known to be good and the peripheral to be tested. SLIDE can be used to load stand-alone programs. It can also chain diagnostics in the same manner as XXDP+.

## CHAPTER 2

## DIAGNOSTIC PROGRAM METRICS

In this chapter, the term diagnostic metrics refers to the characteristics, qualities, and attributes that affect the usefulness or effectiveness of diagnostics for their various users and applications. Chapter 1 introduced diagnostic metrics from the standpoint of the diagnostic users and applications. This chapter attempts to further define the metrics and relate them to the diagnostic design and development process.

Considered in this chapter are the following metrics:

- Fault detection coverage
- Fault isolation and troubleshooting support
- Diagnostic size
- Diagnostic execution time
- Operational functionality and documentation

## 2.1 FAULT DETECTION COVERAGE

Fault detection coverage is the common denominator or basic metric of all diagnostic uses. Inadequate, incomplete fault detection increases repair cost in either of two ways. First, it may defer detection of a fault to a later point in the computer manufacturing process. This results in higher repair or recycling costs. Or it may defer detection of a fault to a higher level diagnostic program (ultimately the customer's application). This results in longer troubleshooting and repair verification time and a reduction in customer confidence.

## 2.2 FAULT ISOLATION AND TROUBLESHOOTING SUPPORT

Fault isolation and troubleshooting support are the primary goals of diagnostic programs. Fault isolation is defined as explicit identification, via error reports, of one or more FRUs. A FRU may be an option, a subassembly (backplane and modules), one or a few modules, or one or a few integrated circuits (ICs). Troubleshooting support consists of error reports (short of FRU callout), listing documentation, and operational documentation intended to assist the technician in locating the failing components using a scope, logic prints, etc.

### 2.2.1 Fault Isolation

Fault isolation is an ambitious diagnostic undertaking that cannot be achieved without active cooperation from the design team. This cooperation must be in the form of well-defined and controlled FRU functional logic partitioning or FRU interconnect visibility.

FRU functional logic partitioning requires that the majority of the logic that implements a test function be physically and logically located on one FRU. The implication is that by detecting the fault, the diagnostic program has isolated it to an FRU.

FRU interconnect visibility requires diagnostic read access to logic states and signals that feed or control the test function. When the diagnostic program detects a failure, it gathers the appropriate inputs and control states to determine if the fault is within the test function, or reflecting into the test function from other interacting logic (which may be located in another FRU module or chip). Module interconnect visibility and function interconnect visibility are employed by the PDP-11 microdiagnostics (module level FRU).

Even with FRU interconnect visibility, fault insertion quality control is necessary to measure diagnostic FRU isolation effectiveness.

### 2.2.2 Troubleshooting Support

Troubleshooting support is a traditional component of virtually all diagnostic programs. Error reports provide the first level of troubleshooting information by supplying the failing test and subtest numbers, a brief statement of the function and test performed, and relevant test data and result data. Unless the user has extensive experience with the diagnostic and hardware failure symptoms, the error report information will not identify the repair action. However, the report should direct the user to the correct test listing section which, coupled with the test data and result data reported, should provide detailed troubleshooting assistance.

The test listing documentation, coupled with operational functionality such as loop-on-error, provides the user with a tool for determining the failure source. Unfortunately, effective use of test listing documentation and loop-on-error techniques requires a trained user and well-designed and structured documentation. It is not uncommon for one of these two prerequisites to be missing, resulting in extended troubleshooting and repair sessions. The diagnostic engineer cannot greatly influence the level of training and expertise of the diagnostic user. The engineer can, however, implement well-designed, well structured, informative error reports and test sections that maximize the potential transfer of troubleshooting assistance from the implementor to the user.

### 2.3 DIAGNOSTIC PROGRAM SIZE

Diagnostic program size is measured in kilowords (KW) of memory occupied by a diagnostic program at execution time. Diagnostic programs are comprised of test data, test execution code, environment interface code, and ASCII data. None of these components can be reduced arbitrarily without sacrificing test coverage, operational functionality, isolation, or troubleshooting support effectiveness. Program size must not exceed the minimum supported system memory size minus the size of the diagnostic control programs. Beyond this restriction, program size should be a function of the hardware test application. For example, a single program covering a total hardware subsystem maximizes local load media and test efficiency. Conversely, several small programs covering specific hardware subassemblies and modules will minimize APT down-line load time in a structured test process such as manufacturing module screening.

### 2.4 DIAGNOSTIC EXECUTION TIME

The execution time of a diagnostic program is the elapsed time from start to completion of one test pass. A test pass may consist of completion of all tests for each selected UUT (serial test), or completion of all tests for all selected UUTs (parallel test). Diagnostic execution time is primarily defined by the characteristics and test requirements of the UUT. The first pass is a fast verification (Quick Verify) of the UUT.

Pure logic tests usually execute at machine speed, thus allowing many test passes to occur in a few seconds or less. Electromechanical or data loop-back tests, such as disk head positioning tests or data communication tests, incur millisecond delays (pauses) resulting in test passes of a few minutes or less. Media testing (disk or tape) incurs a combination of data transfer, electromechanical, and media motion delays that can result in many minutes per test pass. Diagnostic program design should not impose unnecessary pass time requirements by building iterations into each test section.

The diagnostic engineer should specify first (Quick Verify) and also subsequent pass execution times (via the functional and program design specification), review them with the users, and employ thoughtful test algorithms to optimize electromechanical and media test execution times.

## 2.5 OPERATIONAL FUNCTIONALITY AND DOCUMENTATION

Diagnostic program operational functionality and documentation define the ease of loading and running the diagnostic program and the use and interpretation of the diagnostic program in a troubleshooting and repair situation.

Operational functionality is primarily what the diagnostic program and the diagnostic control software are capable of providing to the user. Operational functionality should be defined by the user's requirements. Documentation largely defines how easily and effectively the user can take advantage of the functionality.

Clearly, operational functionality is a prerequisite for easy, effective diagnostic program use. However, without effective documentation, the operational functionality will go unused.

Diagnostic programs are used in two modes:

- 1) test mode and
- 2) troubleshooting and repair mode

From an operational standpoint, these two modes have quite different requirements.

### 2.5.1 Test Mode Diagnostic Functionality

Test mode diagnostic use is typically an attempt to answer the question "Is there a hardware fault in the unit, subsystem, or system?" The goal of test mode diagnostic operation is to facilitate the running of all applicable diagnostic programs with as little system configuration, hardware option, and diagnostic knowledge as possible. Only when a fault is detected by a diagnostic program should it be necessary and appropriate for the operator to understand the hardware operation, diagnostic test algorithm, and troubleshooting functionality.

The PDP-11 diagnostic system (diagnostic control software plus unit diagnostic programs) utilizes configuration parameter and diagnostic execution scripts to automate, as much as possible, the test mode use of diagnostic programs. Diagnostic programs adhering to the PDP-11 Diagnostic Runtime Services interface conventions and XXDP+ chain mode programs will operate in script driven test mode (refer to Chapters 6 and 7).

## 2.5.2 Troubleshooting and Repair Diagnostic Functionality

Troubleshooting and repair support diagnostic functionality is important once a fault has been detected and reported by a diagnostic program. The effectiveness of the failure isolation and repair process depends on a combination of the diagnostic error report, diagnostic test algorithm and supporting documentation, and the diagnostic operator controls.

The error report must inform and direct the repair engineer without overwhelming him with superfluous data. PDP-11 diagnostics compatible with Diagnostic Runtime Services employ a three level error report structure -- header, basic, and extended. The intention is to provide essential test information and function or FRU callout (header), initial and final test status information (basic), and free-form troubleshooting information (extended). The reports should provide this information in structured, controlled packets that can be selectively enabled or disabled according to the ability or need of the diagnostic user to use the information.

Diagnostic test algorithms and their supporting documentation are often the final resort troubleshooting guide for the repair engineer. The diagnostic program must clearly define (through documentation and test structure, not through a reading of the code) what the test is doing, and how it is being done. Hardware initialization, initial test data, and test results (data and state) should be clearly identified and accessible. Although some formal diagnostic user training must be a prerequisite for effective diagnostic troubleshooting, the test algorithms and documentation must transfer as much as possible of the diagnostic engineer's hardware and test expertise to non-specialist diagnostic users.

The diagnostic control software's operator controls provide the final element of diagnostic troubleshooting functionality. Diagnostic troubleshooting controls such as loop-on-error, halt-on-error, test and subtest selection, bell-on-error, etc., are traditional functions long provided by diagnostics. In general, these troubleshooting control functions are generic to all diagnostics, and are standardized and implemented largely by the diagnostic control software. However, effective use of these functions depends on the diagnostic test design and proper program interface to (interaction with) these functions.





## CHAPTER 3

## DIAGNOSTIC STRATEGY

This chapter describes diagnostic strategy and structure of the PDP-11 diagnostic system.

## 3.1 SYSTEM CORE DIAGNOSTIC STRATEGY

The Diagnostic Strategy for PDP-11 central processor units, and compatible PDP-11 components, is a strategy designed to provide a test sequence from a top down approach on the system level and a bottom-up approach on the unit level. The bottom-up strategy is based on a building-block approach, initiating tests of the most primitive functions, utilizing the most primitive controls. This strategy continues building component and device confidence until the most complex functions of a device are tested.

The top-down strategy is based on attempting to determine which device on a system is suspected to be failing. This top-down approach is achieved by execution of prime device functions, and their ability to execute effectively interacting with all system components. The system exerciser programs are designed to perform the required system interaction within the constraints of the specification of the system under test.

This section explains today's System Core Diagnostic Strategy from the standpoint of:

- . System Core Definition
- . System Core Diagnostic Goals
- . System Core Diagnostic Implementation Process

## 3.1.1 System Core Definition

System Core is made up of those CPU cluster components which are essential for the loading and execution of the most basic macrolevel program. System Core typically includes the CPU micro-machine and data paths, memory-I/O bus, memory controller and some minimum memory storage, basic program load device functions, and finally, diagnostic control functions. In other words, System Core consists of all of the functions which are called into play from the pressing of the bootstrap button until the second program instruction is fetched from a predictable memory location.

The System Core Diagnostic facility plays an important role in the overall Diagnostic Strategy. Precise isolation of a fault requires consistent, predictable execution of the test procedure which detects the fault. Faults in the System Core will often result in catastrophic unpredictable behavior of macrolevel programs. A Diagnostic Strategy which assumes a fault free System Core, or attempts to test it by using it, will produce ineffective and misleading results whenever one or more core faults exist.

### 3.1.2 System Core Diagnostic Goals

The prime goals of Digital's Core Diagnostic Strategy are to reduce system mean time to repair (MTTR) by providing effective core fault detection, isolation and troubleshooting facilities, and to confront the Field Service/ Manufacturing training problem by simplifying core fault repair.

An effective Core Diagnostic facility benefits total system diagnostic effectiveness (e.g., MTTR) by establishing a base level of system operation from which the macrolevel diagnostic strategy can proceed.

The fault isolation and troubleshooting aids provided by the Core Diagnostic facility enable a wider range of repair personnel to deal with core failures which otherwise would require highly trained technicians employing highly deductive trouble-shooting techniques gained primarily through experience.

A secondary goal of the Core Diagnostic Facility, and one that is somewhat difficult to quantify, is to increase system reliability by protecting the system software from crashes resulting from undetected Core faults. This extra measure of reliability is guaranteed by invoking the Core Diagnostic facility as an integral part of the bootstrap sequence.

### 3.1.3 System Core Diagnostic Implementation

System Core Diagnostic facilities have been implemented, or are being planned, for the full range of Digital's computer systems. Although the systems range greatly in size and price, whether the system is the PDP-11/04 or the PDP-11/70, the core components are similar (i.e., CPU micro-machine, data paths, Mem-I/O bus, etc.), and the Core Diagnostic facilities are functionally the same (i.e., ROM based micro and macro level GO/NOGO tests, Core Test Control, Troubleshooting aids). However, the size and complexity of the core components and diagnostic facilities required vary significantly.

### 3.2 CPU AND CPU OPTION DIAGNOSTIC STRATEGY

The CPU Cluster consists of all the hardcore components and the additional components required for reliable program execution. Effective test and diagnosis of the CPU cluster requires thorough test and verification of the (hopefully small) System Core nucleus components. Progression from microlevel System Core verification to verification of the total CPU requires three levels of macrolevel testing:

- . Hardcore Verification Tests
- . Basic CPU Tests
- . Extended CPU Tests

#### 3.2.1 Hardcore Verification Tests

The Hardcore Verification Tests provide a macrolevel verification of the functions already tested from the microlevel and then expand to other hardcore functions which may not be candidates for thorough microlevel testing due to microdiagnostic size constraints. The Hardcore Verification Tests constitute the most basic macrolevel program execution and rely on a basic CPU "halt" function and primitive error-status display capability.

The role of the Hardcore Verification Tests with respect to the overall Diagnostic Strategy is to provide the base level of operation and to load and support the Basic CPU cluster Tests.

#### 3.2.2 Basic CPU Cluster Tests

The Basic CPU Cluster Tests verify the fundamental operation of CPU cluster components such as Memory Management, Data Cache, and basic CPU Interrupt and Trap functions.

These tests establish the hardware base for execution of the Extended CPU Cluster Tests and the memory and I/O Subsystem Tests.

#### 3.2.3 Extended CPU Cluster Tests

The Extended CPU Cluster Tests extend from the basic CPU Cluster Tests to include CPU options such as Extended or Floating Arithmetic Instructions, and all normal and "exception condition" CPU cluster functions. Specific diagnostic system implementations may perform basic and extended CPU cluster testing within the same diagnostic program. In other implementations -- such as one based on establishing a hardware base on which the diagnostic control software will run as soon as possible -- the basic and extended CPU Cluster Tests will be performed by separate diagnostic programs.

### 3.2.4 Microdiagnostics

The trend in peripheral subsystem design (terminals included) is towards microprogrammed, microprocessor-based control. The trend in peripheral (and terminal) diagnostics is towards diagnostic microcode, i.e., microdiagnostics. Microcoded diagnostics have three important, inherent advantages over macrocoded diagnostics:

1. Microdiagnostics can test hardware in very small incremental pieces. In this sense, the microprocessor is a diagnostic feature yielding very efficient tests and greatly improved fault detection and isolation.
2. Microdiagnostics provide complete, subfunction level control over the sequence and timing of test operations. In addition to affording more effective and efficient testing, this capability facilitates time measurements and time-critical testing in a way which is processor independent and operating system independent.
3. Since peripheral microdiagnostics are merely data files insofar as processors and software are concerned, peripheral microdiagnostics are inherently processor independent and operating system independent.

Diagnostic microcode will generally be implemented as a combination of resident and non-resident firmware. The resident firmware will reside in ROM or PROM and will contain the system core microdiagnostic. The system core microdiagnostic will test:

1. the hardware system core,
2. the load path for the non-resident microdiagnostics and
3. the ability of the resident microdiagnostics to communicate with software.

The non-resident microdiagnostics will take the form of RAM overlays and will functionally pick up where the resident words microdiagnostics left off. In general, there will be many words of non-resident microdiagnostics for each word of resident microdiagnostics. Resident microdiagnostics would be initiated by power-up, pushbutton, or software. Microdiagnostics would normally rely on software to analyze and interpret results, communicate with the operator, and load the non-resident microdiagnostic overlays. As an alternative, microdiagnostics could be made available in the form of optional plug-in modules.

Microdiagnostics are the most effective and efficient means of detecting and isolating hardware faults. They do not verify the design, test against functional specifications, or demonstrate the capacity to do useful work. For these reasons, higher level diagnostics must be provided in addition to microdiagnostics.

### 3.3 PERIPHERAL DIAGNOSTIC STRATEGY

In general, a peripheral diagnostic can assume that the central processor and memory are 100% functional. This is usually a safe assumption, although the extended, high-speed data transfers included in some peripheral diagnostics (e.g., disk) should be expected to sometimes bring out intermittent memory failures not detected by the memory diagnostics. Thus, in a sense, intermittent memory faults are a proper secondary target of certain peripheral diagnostic tests.

#### 3.3.1 General Requirements

Peripheral diagnostics should run on all members of a processor family. This requires that code be limited to a carefully chosen subset of the total family instruction set and that code which depends on instruction execution time for proper execution be avoided.

Peripheral diagnostics are used in a variety of environments -- standalone, APT, operating system, system exerciser, remote diagnosis, etc.

### 3.3.2 Fault Detection and Isolation

As with all types of diagnostics, the general trend will be towards better fault detection and towards automatic or semiautomatic fault isolation. Improved fault detection and isolation will be the result of six elements:

- a. More comprehensive and detailed status reporting, error detection, and error reporting by peripheral controllers and drives.
- b. More careful partitioning of hardware so as to create rational relationships between functions (or malfunctions) and the field replaceable units.
- c. More comprehensive "diagnostic hooks" including:
  1. data and control loopbacks (wraparounds) at all major interfaces:
    - . between the I/O bus and the central processor
    - . between the I/O bus and the peripheral controller
    - . between the peripheral controller and the drive
    - . between the drive and the media
  2. facilities allowing hardware to be exercised and tested incrementally and in small pieces at the "subfunctional" level consisting of:
    - . the capability to read and write all registers, status flip-flops, control memories, silos, buffer memories, etc.
    - . the capability to one-step, i.e., clock, logic under program control
    - . the capability to thoroughly exercise and test a peripheral controller without involving the drive
    - . the capability to thoroughly exercise and test a peripheral subsystem without involving the media (e.g., without head motion, without tape motion, without reading from or writing to the media, etc.)
    - . the capability to directly test all data and control paths and arithmetic and logical operations at the "subfunctional" level
    - . the ability to "force" errors under program control

- d. Diagnostic microcode.
- e. Physical fault insertion to verify fault detection and insulation and to create (or just to verify) fault dictionaries.
- f. Greatly increased emphasis and effort on fault isolation.

Peripheral subsystem diagnostics will provide one of three levels of automatic fault isolation:

- a. Faults will be isolated to the subsystem, i.e., the subsystem will be verified.
- b. Faults will be isolated to the controller or drive, i.e, controller faults will be distinguished from drive faults and drive faults will be distinguished from media faults.
- c. Faults will be isolated to one or a few field replaceable units.

For a given peripheral subsystem, the actual level of automatic fault isolation provided will be determined by both practical and philosophical considerations such as total number of field replaceable units, product cost, product life, schedule, volume, development cost, resource availability, etc.

#### 3.4 SYSTEM EXERCISER DIAGNOSTIC STRATEGY

The primary goals of a system exerciser are:

- 1. to detect and isolate failures not detectable at the unit or subsystem level of testing, and
- 2. to provide a level of confidence in the functionality and reliability of a system.

The system exerciser is especially designed to catch the following types of problems:

- 1. bus interconnection problems, such as those associated with latency, noise, crosstalk and transmission line phenomena,
- 2. device interaction problems such as those caused by bus or memory contention,
- 3. system loading and bus traffic problems such as data late, data overrun or underrun caused by latency, bandwidth, access rate, cycle time, peak transfer rates, etc.

4. other configuration-sensitive problems such as those of multi-port peripherals, shared busses, multiprocessors, etc.

A system exerciser is not designed to simulate normal, customer usage of a system. Rather, a system exerciser is designed to stress the system (within the specified limits of the hardware).

In general, the data transfer rate, subsystem interaction, bus traffic and general level of system activity produced by the system exerciser should exceed that produced by user tasks running under an operating system.

A system exerciser is designed for comprehensive error detection. Failures are much more likely to be detected and isolated via a system exerciser than via systems or applications programs. Often, faults which only appear intermittently when running systems or applications programs, appear frequently when running the system exerciser.

A system exerciser test module must be developed for each unique system component. The system exerciser is a configuration of subsystem-specific test modules executing concurrently (under control of an executive program). The run-time system exerciser is sysgen'd in the factory or field for each system configuration.

### 3.5 COMMUNICATIONS DIAGNOSTIC STRATEGY

Communications Diagnostic Engineering provides software and hardware expertise to Manufacturing, Engineering, and Field Service and acts as the "sounding board" on such issues as marketplace, networks, engineering design and implementation, front ends, I/O processor architecture, system engineering user mode diagnostics and the related supporting software.

It is important that the network diagnostic strategy is developed irrespective of processor type or architecture and that it supplies the guidelines for the network system. The goal is to provide:

1. Link Verification (node-to-node communication).
2. Node Verification (Modem & Line).
3. Controller Verification (communications device).



### 3.5.1 Communications Strategy

Communications diagnostics are structured to provide a high level of fault isolation in keeping with Field Service requirements. There are two basic approaches to developing hierarchies of programs designed to systematically detect and isolate failures in hardware. One approach is the "bottom up" process, which assumes that the failing unit is probably identified when the fault is initially detected. The other is the "top down" approach, which assumes that the fault is not isolated to a small segment of the system when the fault is initially detected. The first program to run on the system, in the top down approach will be a system exerciser whose purpose is to identify the failing subsystem or unit(s). Once the subsystem or unit is identified, a unit diagnostic will be run bottom up to identify the failing segment of logic. Most often this will be the module which will be replaced. In some cases, the maintenance philosophy may require identification of the failing component to accomodate field repair. In those cases another program may be run to analyze that segment of logic and identify the failing component. The "top down" approach has been established to minimize debug time in the field.

### 3.5.2 Link Verification

The following strategies are based on the "top down" approach.

DCLT (Data Communications Link Test) - These programs are meant to provide Field Service with a tool to maintain communication links. The programs provide the coverage necessary to detect failures in the computer equipment, the communication link, or the modems. Several modes of operation are available which can be used to check out the communication link using a building block approach.

NIE (The Network Interconnect Exerciser) will be used to determine the ability of nodes on the Network Interconnect (NI) to communicate. The NIE will also be used to aid other tools in troubleshooting the NI. The exerciser should be run whenever a problem is noticed and the ability of nodes to communicate is in question. It could also be used to test the ability of a new node to communicate on the network or to test the communication ability of a node which has been having problems.

ITEP (Interprocessor Test Program) is an over-the-line program to isolate faults primarily in the communications device, telephone line, or modem. It provides the Field Service operator with tight, repetitive loops for scope viewing of the modem leads and line. It also provides simple message transfer to a remote site running the same program or the Maynard Communications Turn-Around System. ITEP checks out terminals as well as remote processors.

CTAS (The Communication Turn-Around System) will provide the Field Service organization an easy to use method for establishing the data reliability and link integrity of a data communications network. Field Service personnel using CTAS will be able to perform installation checkout for new systems and fault isolation in existing systems.

Several modem types are supported as well as various baud rates. This allows the user to run against a known good interface, so that problems may be properly located, whether they occur in the software, interface hardware, modem, or communication line.

### 3.5.3 System Performance

DEC/X11 -- DEC/X11 modules are written for all communication devices to provide system performance critiqueing. The hardware design provides the programmable maintenance features needed to support DEC/X11 without requiring Field Service hardware reconfiguration. This is especially true for communication devices as they are usually connected to modems and the telephone system. Approximately 85% of the logic can be tested without operator intervention.

Exerciser Emulators -- Special Field Service diagnostic tools such as the IBM 2848 responder are provided to spot faulty customer design problems due to protocol, channel interface design and non-standard configurations. These "tools" find delicate and subtle micro-processor type problems as well as proving "on which side of the fence" the problem lies. Isolation of the customer's software, telephone line, and modem are critically important to a network's, as well as Digital's, credibility. These programs are submitted along with the normal SDC released programs.

Manufacturing Exercisers -- Because of the special requirements of Manufacturing (one all encompassing GO/NOGO diagnostic), a manufacturing-only diagnostic is provided only through internal distribution. These programs allow selection and deselection of multiple units while undergoing heat environment testing. They also provide the ability to configure a non-standard system to minimize operator intervention. Communication devices have a proliferation of strapping and switch options, and if all were required to be selected and cut, production test time would be greatly increased. Manufacturing Exercisers greatly increase manufacturing throughput and have served very well for the existing hardware architecture.

#### 3.5.4 Standalone Communications Diagnostics

After the failing unit has been identified, the bottom up approach is used. The diagnostics are operated in a standalone mode and stress timing internal to the device to maximum limits. Finding timing errors in the operating system environment would be much more subtle and difficult than using these tests. Usually five or six 8K word programs are involved as all code is straight line. Errors found in the lower level tests will be more descriptive in these simpler tests to help isolate problems to the component level.

### 3.6 AUTOTEST DIAGNOSTIC STRATEGY

#### 3.6.1 Commercial Maintenance Products Program

The Commercial Maintenance Products Program developed by Commercial Diagnostic Engineering fills a need for user friendly functional testing packages as part of the existing commercial system maintenance strategy. With no training or documentation required for operation, plain English prompts and responses, and test run times measured in seconds or minutes, this software has a broad range of applications.

#### 3.6.2 FTS-11

The Functional Test System for PDP-11 bounded computing systems is part of the Commercial Maintenance Products program. From a library of functional test modules, specific test packages can be rapidly assembled to fill the needs of novice commercial users.

#### 3.6.3 Auto Test Strategy for DECSA (Digital Ethernet Communications Service, Model A)

The DECSA system is tested by three sets of diagnostics. The sets are a self test, a Loadable Diagnostic Image, and a set of On-Line Diagnostics.

The Self-Test executes on power up or "init" and checks out the "hard core" and "load path". Then the Loadable Diagnostic Image (LDI) is brought in to do a thorough diagnosis of the DECSA. To complement these tests, some on-line tests are included that will test line cards and data links while the system is running the "operating software".

The Self-Test and LDI will run in an automatic mode and can be invoked by the operator depressing a TEST switch on the front panel of the DECSA.



## CHAPTER 4

## OPERATING MODES AND CAPABILITIES

The following operating modes and capabilities should be provided by diagnostic programs. Various diagnostics may require additional features to achieve their goals and maximize their utility. Program control should be consistent within a program, among programs for the same or similar hardware subsystems, among programs for a given product line or family, and should follow all applicable standards. Deviations from standards should be noted and completely described within the program listing and in the Program Users' Description.

In general, operator control of, and interaction with, the program should be via an interactive terminal, console switches or a software switch register and a printer. The present trend in CPU design is away from hardware switch registers. Thus emphasis should be placed on designs where these functions are implemented in software (see Chapter 6). Operator interaction with a program by explicitly interrogating or modifying memory locations is to be avoided. When unavoidable (due to memory limitations), the documentation should highlight this fact. Instruction for entering this information should be clearly stated.

## 4.1 PROGRAM SELF IDENTIFICATION

If a print device is available, the program should print its name, revision number, and release date as the first step of execution. In the case of programs with long run times, tests and possibly subtests should similarly identify themselves. The exceptions to this are unit diagnostics for CPUs and memories where the CPU and memory are assumed to be working. Consideration should be given to inhibit this function under certain special conditions (e.g., Quick Verify and chain mode in an automated environment).

## 4.2 TEST SELECTION CAPABILITY

Diagnostic programs should provide the operator with the capability to:

1. Select and run the complete set of tests comprising the program. On unit diagnostics this should be the normal default mode of operation. Note that sub-tests requiring operator intervention normally should not be executed in the default mode.
2. Begin program execution with the first sub-test and run the subset of tests between the first sub-test and the selected sub-test of the program, inclusively. The diagnostic engineer should also consider the option of allowing the operator to specify loops which start at a selected sub-test and recycle at the last sub-test in the program.
3. Loop on operator-selected tests. When looping on operator-selected tests, an initial error is not required to establish the loop. When looping because the loop-on-error switch was set, an initial error is required to establish the loop but additional errors are not required to sustain the loop.

In programs constructed incrementally, like unit diagnostics, starting the testing in the middle of the program should not produce erroneous or confusing error messages. The diagnostic engineer must design tests to be independent of one another.

### 4.3 PROGRAM EXECUTION MODES

Depending upon the program execution mode, the occurrence of an error will result in the following:

#### 4.3.1 Continue On Error

If an error occurs in this mode, the program will print the error message and continue unless error reports have been inhibited (4.3.5). Diagnostics default to this mode of operation.

#### 4.3.2 Halt On Error

If an error occurs in this mode, program execution will be halted or go back to the prompt in DRS. The capability to continue or to restart the program or to loop on the failing test must also be provided. Continuing is the default mode in which most programs will execute. The exception to this default condition may be certain unit diagnostics (e.g., CPU diagnostics).

### 4.3.3 Loop On Error

In general, the size and boundaries of error loops are determined by the diagnostic engineer. An initial error is required to establish the loop, but additional errors beyond the first should not be required to sustain the loop. The default condition should be for the program to continue after an error.

#### Note

In certain very special cases this rule may be violated (e.g, exercisers). However, when violated, the documentation should clearly state this fact.

### 4.3.4 One Error Report Per Sub-test

Programs should be designed so that if an error occurs in a subtest, only the first error report will be printed.

### 4.3.5 Inhibit Error Reports

This mode will inhibit printing error reports, except for a class of errors (fatal) which should be printed whether or not this mode is selected.

### 4.3.6 Inhibit Progress Reports

The operator must have the ability to inhibit printing progress reports. This mode will provide that capability.

### 4.3.7 Error Logging

An optional mode of operation can include the accumulation of various error types in memory during execution, with or without error type-outs. These memory locations could be later interrogated by the operator. This mode could be used in circumstances where an output device is normally not available or where prolonged, unattended operation is expected in normal circumstances. Report code section of DRS diagnostics support the logging of test summaries. (DEC STD 153, being updated, addresses Logging and Reporting of System Events.)

#### 4.3.8 Signal (Bell) On Error

This optional mode of operation is to provide the user with an audible or visual indication that an error has been detected. It is useful when adjustments are required or when intermittents are being tracked down.

#### 4.4 INTERACTIVE PROGRAM EXECUTION

Execution of certain tests requires operator action, such as recabling, powering down, opening doors, removing disk packs, inserting file protect rings, operating switches and pushbuttons, etc. With such interactive programs, the tests requiring operator action should:

1. print out clear, complete instructions,
2. pause until the operator responds via switches or keyboard, indicating the action has been taken,
3. verify the operator's actions, and
4. continue execution.

Lack of a timely response or failure to verify the operator action after a response should result in an appropriate message and retry by the program. If neither a terminal or printer is available, the program will take action similar to that described above (except, of course, no instructions or messages will be printed). Here it is doubly important that the operator's actions be thoroughly described in both the listings and the Operating Procedures section of the Program User Description. In summary, the diagnostic engineer should recognize that restart and retry capabilities are very important functions which must be carefully human engineered.

The diagnostic engineer should consider methods of defaulting program execution to its unattended mode (i.e., don't do any test requiring operator intervention). This is a human engineering consideration which is based on the type of program and the environment in which it will be used.

#### 4.5 BASIC FUNCTIONAL TESTING AND RELIABILITY MODE

When designing a system core diagnostic, the diagnostician should consider at least two modes of operation which may be required for the program.



#### 4.5.1 Basic Functional Testing

In this mode, the basic (gross) functional tests are executed. This mode, also known as quick verify, is used as a go/no-go type test. The purpose of this mode is to provide maximum coverage with minimum execution time. Sub-tests requiring operator intervention are never performed in the quick verify mode.

#### 4.5.2 Reliability Mode

In the reliability mode, all sub-tests in the program are executed. In some cases, the sub-tests may be iterated several times during a single pass. Finally, programs requiring operator actions may need a mode of operation where the sub-tests requiring operator action are skipped. The quick verify mode may be implemented by skipping manual intervention sub-tests and performing all other sub-tests but without all the iterations performed in the reliability mode.

#### 4.6 DEFAULTING PHILOSOPHY

As a general philosophy, the default mode of operation (load and go) should be the most rigorous and complete mode of execution of the program. The diagnostic engineer should consider the objectives of the program relative to its uses and operating environments when determining the sub-tests and iterations to be used in the default mode.

#### 4.7 SPECIAL OPERATING MODES

Exercisers are a special class of programs where the diagnostic engineer may see that more than two modes of operation are useful. For example, an acceptance mode may be needed for manufacturing. Thus, the program may have several different sequences of sub-tests (scripts) required for the various modes of operation. There are two basic approaches applicable. The first is to package the various scripts and use the switches or other indicators to determine which script to execute. The second is to allow the operator or an automated manufacturing system to specify the sub-tests to be run. The diagnostic engineer should address the question and understand the tradeoffs between using one or both of the two implementations. The prime factor to consider is the frequency with which the scripts would be changed. If they are expected to change rarely, the first approach is recommended.

#### 4.8 CHAIN MODE

Chain mode operation consists of the sequential execution of programs without operator intervention. Only programs that have been designed to run in chain mode can be chained. For additional information on the use of chain mode refer to Chapter 7.

## CHAPTER 5

## DIAGNOSTIC DEVELOPMENT PROCESS

This chapter identifies the activities that make up the diagnostic development process. The process presented is general in that it is appropriate for any diagnostic development effort -- large or small. The presentation is also specific in that it is heavily biased toward the DIGITAL diagnostic development process.

The diagnostic development process consists of the following major phases:

- Consultation
- Planning
- Implementation
- Testing, QA, and release
- Maintenance

Each phase involves objectives, time and staffing requirements, and external dependencies. Although specific objectives, requirements, and dependencies will vary from project to project, development of an effective diagnostic product requires thoughtful attention to each of the development phases. The Phase Review Process Manual (Order No. EL-EN356-00) and DEC STD 028 describe DIGITAL's corporate Phase Review Policy. Diagnostic projects will usually fit into the phases and use some of the milestones of the hardware systems for which the diagnostic programs are being developed.

### 5.1 CONSULTATION PHASE (PHASE 0)

The consultation phase of diagnostic development is an information gathering and exchange process. It is usually an effort requiring an experienced diagnostic project leader or technical supervisor to work with engineering, customer services, and manufacturing to formulate diagnostic strategy, key project milestones, and preliminary staffing requirements.

The consultation phase may start before project funding is negotiated and continues through the writing of a cursory project plan (strategy, key milestones, staffing).

Failure of a diagnostic engineer to be involved in the consultation phase reduces the opportunity for early diagnostic inputs and increases the chance of failure if there is a lack of understanding of the users' requirements.

The results of this phase should be published in a diagnostic requirements document. This document specifies the technical requirements of the diagnostic in terms of:

- . required capability of the product
- . required environment in which it will operate
- . required packaging, installability, ease-of-use, performance, reliability, maintainability, compatibility, evolvability, and serviceability (required quality)
- . required documentation (refer to the Software Development Policies and Procedures)

## 5.2 PLANNING PHASE (PHASE 1)

The diagnostic development planning phase can begin when the cursory diagnostic requirements document is reviewed and agreed upon and a diagnostic project leader is assigned. Then the diagnostic project plan or functional specification is written. For moderate to large projects (2 or more diagnostic engineers and/or 9 months or more duration), the following diagnostic planning documents should be developed:

- Diagnostic requirements document
- Diagnostic project plan
- Diagnostic functional specification
- Diagnostic program design specification

For smaller projects, it is appropriate to combine the relevant planning information into one or two documents. DIGITAL diagnostic engineers should follow the Software Development Policies and Procedures manual for the diagnostic engineering project plan (7C3-1), functional specification (7C3-2), and program design specification (7C3-3).

### 5.2.1 Diagnostic Project Plan

The diagnostic project plan lays the foundation for the total development effort. It presents, in a single document, an overview of the product and product goals, a statement of diagnostic goals and strategy, a summary of key project and diagnostic development milestones, and estimates of required resources (staff and computer facilities). Specifically state gating items, critical paths, requirements, and assumptions being made. Often, the project plan is developed in two stages. A Rev 0 project plan requiring from two to several weeks to develop may be followed later (often after functional specifications are written) by the Rev 1 or final project plan.

Thoughtful development and review of the project plan are prerequisites for all diagnostic development efforts, regardless of their size or complexity.

### 5.2.2 Diagnostic Functional Specification

The diagnostic functional specification is essentially a statement of how the diagnostic goals for each major diagnostic component will be achieved. The functional specification should be developed by the project leader or diagnostic engineer responsible for program implementation.

The diagnostic functional specification addresses three important facets of the product:

- a. Diagnostic Product Goals
  - . Intended users (design engineering, field service, manufacturing)
  - . Intended environments (local operator test, repair, APT, APT-RD, XXDP+, ACT-11, SLIDE)
  - . Diagnostic metrics (fault detection, isolation, and troubleshooting goals; program size and execution-time goals; operational functionality and documentation goals)
- b. Diagnostic Requirements
  - . Hardware test and isolation aids (special control logic, partitioning, test visibility)
  - . Hardware and software restrictions (hard-core error detection, minimum memory size and required hardware options, operating system driver)
  - . Development requirements (development resources: hardware and software, debug and evaluation resources, project staffing)
- c. Development Process
  - . Key project milestones (engineering breadboard and prototype support: what and when, preliminary release availability, final completed availability)
  - . Key process events (specification and implementation reviews, quality assurance procedure, post-release support)

### 5.2.3 Diagnostic Program Design Specification

The diagnostic program design specification describes, to the working design level, the internal diagnostic program implementation. It describes how the diagnostic functionality is to be implemented.

Several methods of program design representation are available:

- Detailed hierarchy charts
- Interface specification blocks
- HIPO diagrams (structured flowcharts)
- Programming design language 1 (PDL1)

Development of an appropriate program design representation -- from overview level hierarchy charts to detailed PDL1 descriptions -- is well worth the initial investment. It increases the probability of a high quality, accurately scheduled, program implementation and the timely development of useful program maintenance documentation.

### 5.3 IMPLEMENTATION PHASE (PHASE 2)

In theory, the transition between the diagnostic planning phase and the implementation phase should be clearly defined. Occasionally, however, both activities must go on in parallel. Such is the case for diagnostic efforts in support of new hardware products, where engineering breadboard and prototype support programs for hardware debug and design verification are needed well before the final diagnostic product is needed, or could be developed.

It is often necessary and desirable to plan the engineering breadboard and prototype diagnostic support phase as a semi-independent part of a project within the overall diagnostic effort. Based on the timing of engineering hardware support requirements, with respect to the startup of the diagnostic plan and specification effort, it may be necessary, and desirable, to defer detailed diagnostic functional and design specification completion until the engineering support programs are in place. Obviously the hardware dependent diagnostic capabilities and requirements must be specified during hardware design.

#### 5.3.1 Engineering Breadboard and Prototype Support

The objective of this part of the implementation phase is to provide the hardware engineers with basic hardware debug programs and design verification programs. These programs will be required within a few hours to a few days of initial hardware power-on. The level of hardware debug program support and diagnostic engineer support will vary from project to project. However, hardware design verification programs are normally essential to reduce the propagation of design mistakes into large numbers of prototypes or final systems.

The timely development of the correct (needed) set of engineering debug and design verification programs is an early, visible, and important phase in the diagnostic development process. Also, this phase enables the diagnostic engineer to develop the hardware functional understanding and the hardware implementation understanding that is essential for specification and implementation of effective diagnostics. The hardware debug and design verification effort requires planning and review to the same extent as the final diagnostic effort.

### 5.3.2 Final Diagnostic Implementation

To the same degree that the engineering breadboard and prototype support diagnostic effort must be focused on engineering hardware debug and design verification needs, the final diagnostic implementation effort must be focused on the diagnostic effectiveness and process needs of field service and manufacturing.

Since the needs of engineering debug and design evaluation are considerably different from those of manufacturing and field service (Section 1.1), the engineering diagnostic programs are not readily transportable to the manufacturing and field service environments. However, the diagnostic engineer's acquired knowledge and expertise are significant and transportable.

The final diagnostic implementation phase begins with review of the diagnostic functional and program design specifications and ends when the diagnostic programs are suitable for pre-release. Since this phase of diagnostic implementation is often a critical path for key product milestones (manufacturing startup, design maturity testing, and first customer shipment) it is important that the development tasks be well-defined, understood, scheduled in measurable stages, monitored, and reported. Good foresight in the planning phase will pay off here. For complex or critical product development efforts, trade-offs may have to be made between diagnostic program completion and the need to provide base level diagnostic programs. This is fine as long as deficiencies and incompleteness are well communicated. Schedules should be reviewed for possible early support interference with remaining development and test. Legitimate diagnostic program pre-release can occur when diagnostic development is complete (including debug and test), listing documentation and operational documentation are in final form, and a formal quality assurance (QA) checklist has been completed.

The diagnostic engineer should prepare the QA checklist according to the goals set up in the functional specification and this manual. It is good practice (required in DIGITAL diagnostic engineering) to conduct a pre-release review of the package (including the planned QA checklist) with hardware engineering, manufacturing, and customer services engineering representatives.

#### 5.4 DIAGNOSTIC QUALITY ASSURANCE AND RELEASE PHASE (PHASE 3A)

The final stage in the diagnostic development process is the QA effort leading to formal diagnostic release.

Quality assurance must be an ongoing process in the diagnostic development effort.

The QA process must involve members of customer services, manufacturing, and possibly, product lines, formal evaluation.

##### 5.4.1 Introduction

Diagnostic Quality Assurance is an aggregate of policies and procedures designed to ensure that a diagnostic achieves the desired level of quality. The desired level of quality must be determined before the project is underway and must be expressed as a set of project goals. The resources (people, time, equipment, technology) required to achieve the desired level of quality must also be determined. Included in these resources must be the quality assurance resources, i.e., the resources necessary to test the quality of the diagnostic and to correct discrepancies between specified and measured quality.

The diagnostic quality objectives for a given product are determined by manufacturing cost objectives, maintenance cost objectives, product life, product volume, competition, field and factory test and repair strategies, customer expectations and the resources available for diagnostic development. Included in these resources are the diagnostic features designed into the hardware.

The following diagnostic qualities must be assured to the specified level:

- . fault detection
- . error reporting precision, accuracy and comprehensiveness
- . fault isolation
- . necessary skill level of the intended user
- . precision, accuracy and comprehensiveness of the documentation
- . maintainability of the diagnostic
- . XXDP+/ACT/APT/SLIDE or other system compatibility
- . compatibility with all legal configurations
- . conformance to standards



The basic ingredients of the QA process are:

- . design review
- . design engineering acceptance
- . factory installation
- . performance feedback
- . configuration on other system compatibility testing
- . XXDP+/ACT/APT/SLIDE or other system compatibility testing
- . hardware or software fault insertion

It is important to note the QA testing of a diagnostic cannot be confined to the engineering laboratory. For reasons which are explained below, diagnostic QA testing is a process which continues through the first few months of production and delivery.

#### 5.4.2 Design Reviews

The review of diagnostic plans and specification is the first phase of the QA process. These reviews assure that the knowledge, insight and experience of the entire corporation are exploited, from the inception of the project, by the diagnostic engineers. Periodic design reviews during the design, coding and debug phases of the diagnostic development cycle minimize the number and scope of discrepancies discovered during the quality assurance testing phase.

#### 5.4.3 Manufacturing Installation

After debug on prototype hardware, the diagnostic engineer introduces the diagnostic program and assists in installation on the pilot-production line. At this point the program is really a prototype diagnostic and the engineer will instruct manufacturing and field personnel in its use, monitor its utility and effectiveness, solicit constructive criticism and react to the process by correcting and enhancing the diagnostic. This process is complete only when the production line is up and running to everyone's satisfaction, normally a matter of a few months.

#### 5.4.4 Performance Feedback

Feedback of diagnostic performance is vital to the diagnostic quality assurance process. Feedback should come primarily from two sources, Manufacturing and Field Service. Manufacturing uses the diagnostics in the factory installation process. Field Service uses the diagnostics in the field test sites and at first customer sites. Any bugs or deficiencies discovered by these groups should be reported back to the diagnostic developers. In the initial startup phase of a product, the problems should be reported by telephone to the developers. As the product life cycle continues, problems can be reported back using problem reporting systems such as AIDS or PRISM.

#### 5.4.5 Configuration Compatibility Testing

Engineering must connect prototype hardware to several configurations so that diagnostic software can be run against a variety of configuration possibilities.

Engineering laboratories seldom have sufficiently large and varied configurations to assure that a program will run on all members of a processor family and with all options but every effort should be made to audit diagnostic performance on varied system configurations.

Configuration compatibility testing is normally completed at a System Manufacturing site. The diagnostic engineer must monitor the configurations shipping out of Manufacturing before reaching a satisfactory level of confidence that the program will run properly on all legal configurations. Diagnostic problems which show up in system checkout are usually configuration related problems.

#### 5.4.6 XXDP+/ACT/APT/SLIDE and Other System Compatibility Testing

The ability of XXDP+ to load and chain diagnostics needs to be tested as part of the initial debug of the diagnostic.

ACT/APT/SLIDE and other system compatibility also need to be tested as part of the initial debug. This can best be accomplished by running APT lines to all the engineering labs. Pilot production time is too late to discover an APT incompatibility. Products intended to run on other operating systems (e.g., RSX-11M) must be fully tested for compatibility with those systems.

#### 5.4.7 Diagnostic Program Verification

Diagnostic program verification is the process of ensuring that the program actually performs as the developers intended. This can seem a monumental task. The combinations of error conditions, hardware configuration, and software environment are overwhelming. Currently, the optimum approach is to force the execution of all executable code in the program. The most desirable way to do this is by physically fault inserting the hardware under test. Normally, though, a diagnostic program tests many more fault conditions than can be physically inserted into the hardware. Therefore, hardware fault insertion must be augmented by other means. Temporary patches can be made to cause the code to execute conditional paths and error paths. All paths must be forced to ensure that errors are properly handled and printouts are correct. Careful structuring of the program in the design phase will simplify the verification process considerably.

#### 5.4.8 Quality Assurance Checklist

The QA process should be planned (via a QA checklist) and reviewed (via the pre-release review) to ensure that all specified diagnostic user applications (Chapter 1) and diagnostic program metrics (Chapter 2) have been achieved. Execution of the QA checklist involves detailed diagnostic effectiveness checks, operational functionality checks, and operating environment checks. Depending on hardware availability and diagnostic product complexity, the QA checklist process requires from two to six weeks to complete properly. Years of experience in diagnostic program development show that this final QA effort makes the difference between delivering prototype quality diagnostic products and delivering finished, production quality diagnostic products. From the perspective of the diagnostic end user, the difference between the product qualities (prototype vs. production) makes the QA process non-negotiable.

### 5.5 MAINTENANCE

Once a diagnostic has been released, the diagnostic goes into its maintenance phase. Any problems reported through the AIDS or PRISM problem reporting systems are fixed. It is in the maintenance stage that good planning and good documentation pay off. Diagnostic developers should schedule 10 to 15% of their time for maintenance of programs that they have developed.



## CHAPTER 6

## XXDP+, THE PDP-11 DIAGNOSTIC OPERATING SYSTEM

## 6.1 INTRODUCTION

XXDP+ is the diagnostic operating system for PDP-11s. It consists of four major components:

- . the monitor
- . the diagnostic runtime services
- . utility programs
- . loadable device drivers for the utilities

These four components work together to accomplish the system functionality.

## 6.1.1 XXDP+ Monitor

The monitor, which forms the core of the system, is the highest level software. All other components require monitor support for their operation. The monitor provides program load and execution, console terminals services, batch control and file services (loading and reading files) for the system storage medium only. The system storage medium is the storage medium on the device from which the monitor was loaded. All other components utilize the terminal services for operator communications and the file services for certain other operations. XXDP+ does not use interrupts.

## 6.1.2 Diagnostic Runtime Services

The Diagnostic Runtime Services (DRS) are an extension of the monitor. For certain types of diagnostic programs, commonly referred to as "DRS compatible", DRS provides non-diagnostic function support including:

- . standard operator interface
- . error message formatting
- . control of diagnostic

### 6.1.3 Utility Programs

The utility programs are used for file manipulation (e.g., moving files from one medium to another), diagnostic pre-parameterization (creating diagnostic files with hardware information included), file modification (e.g., patching), and to create batch control files. The utility programs use the monitor for typing and receiving messages and for loading the read/write device drivers required for file operations.

### 6.1.4 Device Drivers

The fourth component of the XXDP+ System is the collection of device handlers or drivers. These are used by the utility programs to access storage media and I/O devices. The drivers reside on the system storage medium and are loaded into memory as required.

### 6.1.5 XXDP+ Nomenclature

The term "XXDP+" is interpreted as follows:

- XX - 2 alpha characters which specify the media supported by any particular monitor. For example, DXDP+ is the XXDP+ System for the RX01, identified by the "DX" code (see Table 6-3).
- DP - Diagnostic Package, the primary function of the XXDP+ System.
- + - The plus sign differentiates XXDP+ from its forerunner, XXDP.

6.1.5.1 Software Naming Conventions - XXDP+ software components are named as illustrated in the following example:

CHMDXA0 XXDP+ DX MONITOR BIN

CHMDXA0 - first seven characters; the name of a program for a specific software component is derived from this by removing the first character.

These seven characters can be broken down into four groups as follows:

CH-M-DX-A0

The first two characters, CH, represent the XXDP+ family of software and documentation. Only XXDP+ related software has names beginning with CH.

The next character defines the type of product the component is, using the following codes:

- M - monitor
- S - runtime services
- U - utility programs
- Q - manuals

The third group refers to either the device supported (Ref. Table 6-3) or type of software (Ref. Table 6-4) using the following codes, for example:

- DK - RK05 Monitor
- DX - RX01 driver
- SU - SETUP utility

The last characters refer to the revision and patch level of the component, e.g., A0 is the first version of the component while B1 would be the first revision with one patch or temporary modification.

Examples:

- CHMDKB0 - B revision of RK05 monitor
- CHUSUA3 - A revision (with 3 patches) of SETUP utility
- CHDDXA0 - A revision of RX01 driver

### 6.1.5.2 File Naming Conventions

XXDP+ files are specified by a name and an extension. The name may be up to six alphanumeric characters in length and the extension may be up to three alphanumeric characters in length, with no imbedded spaces. The name and extension are separated by a dot(.). File names for all components, other than utilities, are based on the naming conventions discussed in Section 6.1.5.1. The file name is derived by dropping the "C" and taking the next six characters as the file name. The utility programs, for user convenience, are distributed under their common names like UPD2 and XTECO.

Some extensions are used to identify particular file types. For example, all components, other than utility programs, have .SYS extensions. Utility programs have .BIN or .BIC extensions. Batch control files have .CCC extensions. Table 6-1 lists those extensions having particular meanings.

Table 6-1. File Extensions

File Extension	Meaning
BAK	An XTECO backup file.
BIC	An executable program file that may be run or loaded through either operator or batch control operation.
BIN	An executable program file that may only be run or loaded through operator control.
CCC	A batch control file.
LIB	A library file.
OBJ	A DEC/X11 object module.
TXT	A text file.
SYS	An XXDP+ system file.

## 6.2 XXDP+ CONSTRUCTION

The purpose of this section is to give the diagnostic designer a general idea of how the components of XXDP+ are constructed and how they operate.

### 6.2.1 XXDP+ Monitor

The monitor which is about 4K words in size at load time consists of three major components:

- . secondary bootstrap
- . initialization code
- . runtime monitor code

The secondary bootstrap is loaded into memory at boot time and loads the remainder of the monitor into memory. The initialization code gathers certain information and relocates the runtime monitor to the last 2K words in the first 28K words of physical memory. The runtime monitor is the code used to carry out the operator functions described in Section 6.4.1, Monitor Commands. The runtime monitor, which is approximately 2K words in size, consists of five sections outlined in Table 6-2.



Table 6-2. Runtime Monitor Sections

Section	Function
Read Only Device Driver	Loads programs from the system medium and reads batch control files.
Operator Interface Handler	Processes operator commands from console terminal.
Batch Control Handler	Processes batch files from system medium.
Monitor Services Handler	Processes requests for monitor services which are made by utility programs via the EMT instruction.
Console Terminal Driver	Loads programs entered via console terminal

### 6.2.2 Diagnostic Runtime Services

The Diagnostic Runtime Services (DRS) are the portion of the XXDP+ System that control diagnostic programs. This program is an extension of the XXDP+ monitor that is automatically loaded into memory immediately below the monitor and started when a compatible diagnostic is run. DRS also provides non-test related services, such as console terminal support, to these diagnostic programs. All diagnostic programs that are compatible with DRS share some important common features, which are discussed in Chapter 7, DRS COMPATIBLE DIAGNOSTIC PROGRAMS. DRS commands are discussed in Section 6.3.2.

### 6.2.3 XXDP+ Utility Programs

There are five XXDP+ utility programs:

- . UPD2
- . UPD1
- . PATCH
- . SETUP
- . XTECO

All the utility program commands are discussed in Section 6.4.5.

#### 6.2.3.1 UPD2

UPD2 (Update Two) is a file manipulation utility used for building XXDP+ media, copying files from one medium to another, deleting files from a medium, modifying files, and other functions. UPD2 runs in the lowest 6K words of memory. All remaining memory between the top of UPD2 and the bottom of the monitor is used for buffers. UPD2 uses the runtime monitor to interface with the operator and to load the retrievable device drivers it uses for device related functions.

#### 6.2.3.2 UPD1

UPD1 (Update One), a file modification program that duplicates some of the functions of UPD2, is used exclusively for modifying binary files. Because of its small size, it can be used with larger programs than UPD2. UPD1 runs in upper memory and may overlay part of the monitor. You cannot exit from the utility to the monitor directly; you must reboot the monitor. UPD1 requires the retrievable device drivers that reside on the system medium, which must remain on-line and ready while UPD1 is used. UPD1 can only use one device and cannot transfer files.

#### 6.2.3.3 PATCH

The Patch utility can be used to modify any binary-formatted (.BIN or .BIC) file stored on an XXDP+ storage medium. This program can be used when a file which is to be modified is too large to be loaded into the memory space of the system being used.

#### 6.2.3.4 SETUP

SETUP allows the user to build the hardware and software tables for a DRS compatible diagnostic (.BIC or .BIN) and to store these tables with the diagnostic. SETUP also combines a special version of DRS with a diagnostic for use with ACT and SLIDE. These are DEC manufacturing systems discussed in Chapter 8. SETUP has the same memory requirements as DRS: 5.75K words. The total memory required ranges from 16K to 28K words depending on the diagnostic being parameterized. The SETUP commands are discussed in Section 6.4.5.4.

### 6.2.3.5 XTECO

The XTECO utility is a simple editor used to create and modify text files (.TXT or .CCC). It is a limited subset of TECO, a character editor supported by most of DEC's operating systems. As with the other XXDP+ utilities, XTECO commands are discussed in Section 6.4.5.

### 6.2.4 XXDP+ Device Drivers

XXDP+ supports most mass storage devices as well as some non-file structured devices such as paper tape. Table 6-3 lists all devices supported, the mnemonic used to specify the device and the name of the monitor and driver files. The ?? characters in the file name (as used in Table 6-3) refer to the revision and patch level which may vary over time. XXDP+ device drivers are, by necessity, small (maximum of 3400[10] bytes in length) and limited in function. They have limited error detection capabilities: read, write, and hard errors. These errors are reported and control is returned to the utility being used, which then takes any required further action. The user is required to run diagnostics on the device in question if an error persists.

TABLE 6-3. XXDP+ Supported Devices

Device	Mnemonic	Monitor	Driver
TU60	CT	HMCT??	HDCT??
RP04/5/6	DB	HMDB??	HDDB??
TU58*	DD	HMDD??	HDDD??
RK05	DK	HMDK??	HDDK??
RL01/2	DL	HMDL??	HDDL??
RK06/7	DM	HMDM??	HDDM??
RP02/3	DP	HMDP??	HDDP??
RM02/3	DR	HMDR??	HDDR??
RS03/4	DS	HMSD??	HDDS??
DECTAPE	DT	HMDT??	HDDT??
RX01	DX	HMDX??	HDDX??
RX02	DY	HMDY??	HDDY??
PRINTER	LP	--	HDLP??
TM02	MM	HMMM??	HDMM??
TS04	MS	HMMS??	HDMS??
TE10	MT	HMMT??	HDMT??
TR79	TR	HMTR??	HDTR??
PDT11	PD	HMPD??	HDPD??
HI SPD PT PCH	PP	--	HDPP??
LOW SPD PCH	PT	--	HDPT??
HI SPD PT RD	PR	--	HDPR??
LOW SPD PT RD	KB	--	HDKB??

\* The TU58 is also called DECTape II.

### 6.2.5 Building XXDP+

The minimum files that must reside on a bootable XXDP+ medium are the monitor for that medium, the device driver for that medium, the DRS (file name, HSAA??.SYS) and the directory utility (file name, HUDI??.SYS). The monitor file must be loaded up by UPD2 and then saved on the medium. The batch control file XXBLD.CCC will update or build XXDP+ media automatically. (The medium to be built must be in drive 0.) The file is started using the chain command. The file accepts switches which specify the media type to build and the mode in which to build. All supported XXDP+ media may be built. The available modes to build and update are listed in Table 6-4.

Table 6-4. XXDP+ Build and Update Modes

Mode	Description
DRIVER	A bootable medium with all XXDP+ drivers
MONITOR	A bootable medium with all XXDP+ monitors
UTILITY	A bootable medium with all XXDP+ utilities
SYSTEM	A combination of the above three modes

### 6.3 BATCH CONTROL (CHAINING)

XXDP+ has a facility for running programs without operator intervention called batch control or chaining. The commands that would normally be issued by an operator are put into a text file, using XTECO, and the monitor processes the commands in this file. This saves the operator from having to enter the commands manually, and once the batch control file has been created, it can be used over and over again. This process releases the operator from having to do repetitive tasks such as building new media or running a common set of diagnostics. More importantly, batch control allows a user to implement a test strategy and use it consistently. This is done by selecting the proper diagnostics and running them in a particular order and mode to achieve the best test process. Once this process is developed, it is put into a batch control file. Table 6-5 lists the available batch control functions. The batch feature allows one batch control file to call up another control file for execution. On completion of the commands in the second control file, control returns to the first control file.

Table 6-5. Batch Control Functions

COMMANDS	FUNCTION
Monitor Commands	Monitor commands: R, L, S, C, and E
Utility Commands	UPD2, SETUP, PATCH
DRS Commands	all DRS commands and diagnostic dialogue
Conditionals	sections of the batch file can be processed conditionally under operator control or runtime conditions
GOTO tag	begin processing at another section of the batch file designated by "tag"
QUIET	inhibit printing of batch file if printing or enable printing if printing was inhibited previously
PRINT	temporary override of QUIET
SMI/CM I	enable/disable manual intervention operations in specialized diagnostics
QUIT	terminate the batch operation
WAIT	stop batch operation until the operator types a control X

For a discussion of how to use these functions, refer to the XXDP+ User's Manual (CHQUSE).

### 6.3.1 Batch Control of Diagnostics

For the purposes of batch control, there are two types of diagnostics: chainable non-DRS-type (see Chapter 8) diagnostics and DRS-type (see Chapter 7) diagnostics. Chainable non-DRS type diagnostics can be batched by a simple run command: R DIAG[/n]; where n is an optional argument that specifies the number of passes the diagnostic will run (the default is one). DRS-type diagnostics require complete batch control. All commands normally entered by the operator must be in the batch file. The batch file is an INDIRECT COMMAND FILE for DRS. If the diagnostic program uses a software table, it is necessary to provide the commands required to support it. The user does not have to enter all the commands via the batch file, however. By using the SETUP utility (see Section 6.2.3.4) all hardware and software information can be supplied to the diagnostic prior to running the batch job.

### 6.3.2 Batch Control of Utilities

The UPD2, SETUP, and PATCH utilities may be used under batch control. To run one of these utilities under batch control, create a batch file that contains all the commands that would normally be entered by the operator. For example, to build an RX01 floppy diskette for XXDP+ under batch control, using UPD2:

```
R UPD2
LOAD HMDX??.SYS
SAVM DX0:
PIP DX0:=HSAA??.SYS
PIP DX0:=HUDI??.SYS
PIP DX0:=HDDX??.SYS
EXIT
```

The dialogue with UPD2 must end with an EXIT command in order to complete the batch job or to allow further batch functions.

## 6.4 XXDP+ COMMANDS

### 6.4.1 Monitor Commands

This section describes the XXDP+ monitor commands listed in Table 6-6. Some commands have optional "switches" which consist of a single character preceded by a slash(/). These are used to modify the command function.

Wildcard characters (? or \*) are permitted in the file specification. The first file found that fits the wildcard description will be run.

All XXDP+ monitor commands may be used in a chain file with the exception of the TEST command. The Chain command, when used in a chain, invokes nesting of chain commands to one level.

Table 6-6. XXDP+ Monitor Commands

Command	Function
L	Load a program
S	Start a program
R	Run a program
C	Run a batch job (Chain)
D	List directory of load medium
F	Set the terminal fill count
E	Enable alternate drive for system device
H	Type help information
Test	Run a batch file called SYSTEM.CCC
↑C	Return control to monitor

#### 6.4.1.1 Load Command

The Load command is used to load a file into memory. The program must be an executable file. The default extension is .BIN or .BIC (e.g., UPD2.BIN). The format of the load command is:

```
L filnam[.ext]
```

where the file name must be a standard XXDP+ file name. After the program is loaded, the full file name of the loaded program will be printed. Some examples of the load command are:

```
L DIAG (load DIAG.BI?)  
L ZDJCA2.NEW (load ZDJCA2.NEW)
```

#### 6.4.1.2 Start Command

The Start command is used to start a file that has been previously loaded into memory by a Load command. No commands should be issued between a Load and Start command. The purpose of this command sequence is to allow the user to load a program, halt the processor, modify memory contents, restart the monitor and start the program. The format of the Start command is:

```
S [addr]
```

The user may optionally enter a starting address. The monitor will start the program at the transfer address in the file if the operator does not enter a starting address. The default starting address for files without specific transfer addresses is 200 (octal). Some examples of the Start command are:

```
L RXDIAG      (load RXDIAG.BI?)
S              (start at transfer address)

L RXDIAG      (load RXDIAG.BI?)
S 204         (start at memory address 204 octal)
```

#### 6.4.1.3 Run Command

The Run command, which is used to load and start a program stored on the load (system) medium, is a combination of the Load and Start commands. The format of the Run command is:

```
R filnam[.ext] [addr]
```

The default extension is .BIN or .BIC. If there is a file with both extensions on the medium, the first file found is used. After the program is found and loaded, but before the program is started, the full file name is printed out to verify the load. This is useful in determining which of possibly several programs on a medium is being run after a wildcard specification. The file will be started at the transfer address in the file (or at 200 octal in the absence of a transfer address). The operator may optionally specify a starting address. Some examples of the Run command:

```
R UPD2        (load and start UPD2.BI?)
R SAMPLE.XXX  (load and start SAMPLE.XXX)
R RXDIAG 204   (load and start RXDIAG.BI? at location 204 octal)
```

#### 6.4.1.4 Chain Command

The Chain command is used to initiate execution of a batch (or chain) file. The file must be on the system medium and have a .CCC extension. Batch operations may accept user-defined switches. The format of the Chain command is:

```
C filnam[/switches]
```

User-defined switches are ASCII strings delimited by a / and another / or end-of-line. The monitor compares the conditional string in an "IF condition THEN" directive with all user-defined switches. If there is a match, the conditional portion of the chain file is executed.

Example:

```
C CHAIN/string would execute: IF string THEN
                                R DIAG1/Q
                                R DIAG2/Q
                                END
```



### 6.4.1.5 Directory Command

The Directory command is used to obtain a list of all the files on the system medium. This list contains the following five items of information:

- . entry number
- . complete file specification (name and extension)
- . date file created
- . length of file in 256 (decimal) word blocks
- . the number of the first block in the file

A few files are contiguous, i.e., their blocks are in order on the medium. Contiguous files are noted in the directory by a "C" following the date. Most files, however, are "linked", with their blocks not contiguous on the medium.

When the Directory command is given, the monitor must load the directory utility (HUDI???.SYS) which in turn requires the read/write device driver for the system medium type. These two files must be on the system medium in order for the Directory command to work. If one of these files is not on the medium, the monitor will type error message "? NOT FOUND: filename.ext". The format of the Directory command is:

D[/L][/F]

There are two optional switches for the Directory command. The /L switch will cause the directory to be printed on a line printer rather than on the console terminal. The /F switch causes the directory to be printed in a short form. This short form only gives the entry number and file name. Examples of both forms are shown below.

#### Directory Long Form

ENTRY#	FILNAM.EXT	DATE	LENGTH (in decimal)	START (addr. in octal)
1	HMDKA1.SYS	02-Jun-79	12	000100
2	HDDKA0.SYS	02-Jun-79	5	000120
3	HUDIA0.SYS	02-Aug-79	6	000066

#### Directory Short Form

1	HMDKA1.SYS
2	HDDKA0.SYS
3	HUDIA0.SYS

#### 6.4.1.6 Fill Command

The Fill command is used to control the number of non-printing (fill) characters that will be typed after a carriage return. The fill, or null, characters are typed to allow time for the carriage return before typing the next line, thus preventing overprinting. The following terminals require a fill count:

```
.ASR  
.LA30  
.VT05  
.VT50
```

The format of the Command is:

```
F
```

The monitor will print the current fill count in octal and then wait for the user to type the new fill count. If the user does not want to change the count, the user should not type a number, but simply a carriage return. Some examples of the Fill command:

```
F  
000005                (fill count = 5)
```

```
F  
000010 1              (old count = 10 octal, new count = 1)
```

The fill count is initially set to octal 14 in order to allow the monitor to start properly on a system that has one of the terminals that require a fill count. After start-up, the fill count is always reset to 0 since most terminals do not require fill counts. The user can immediately reset the fill count if desired.

#### 6.4.1.7 Enable Command

The Enable command is used to change the drive that the monitor considers to be the system device. For example, if the user had booted the system from drive 0 of an RK05 and later wanted to have the monitor use drive 1 as the system device (that is, as the default device), he or she could do this without re-booting the monitor by using the Enable command. This command is valid for multi-drive devices only and affects drives, not controllers. The format of the command is:

```
E n
```

Where n is the new drive number.

```
E 1    (enables drive 1)
```

#### 6.4.1.8 Help Command

The Help command is used to obtain a brief summary of XXDP+ commands. The contents of a file named HELP.TXT, which must be on the system medium, are printed. There is a switch to cause the summary to be printed on a line printer instead of on the console terminal. The format of the Help command is:

H[/L]

#### 6.4.1.9 Test Command

The Test command is a special case of the C (CHAIN) command. The TEST command invokes a specific chain file: SYSTEM.CCC. In all other respects, the TEST command functions exactly as the C command.

TEST[/SWITCHES]

### 6.4.2 DRS Commands

This section describes the eleven DRS commands listed in Table 6-7.

Table 6-7. DRS Commands .

Command	Function
START	Start the diagnostic and initialize
RESTART	Start the diagnostic and do not initialize
CONTINUE	Continue diagnostic at test that was aborted by a ^C
PROCEED	Continue from an error halt
DROP	Deactivate a unit
ADD	Activate a unit for testing
DISPLAY	Print a list of device information
FLAGS	Print status of all flags
ZFLAGS	Reset (clear) all flags
PRINT	Print statistical information
EXIT	Return to XXDP+ runtime monitor

Ref: XXDP+ System User's Manual (CHQUSE).

The commands are entered in response to the DRS prompt (DR>), which is issued after:

- . the DRS is loaded
- . all specified diagnostic operations are completed
- . a DRS detected error
- . a "halt-on-error" sequence
- . DRS has been aborted by a ^C(CTRL-C)

The sections that follow describe the effect of each command, which may be modified by the use of switches described in Section 6.4.3. DRS recognizes a command by the first three characters. The portion in square brackets may be omitted, i.e., the Start command may be entered as STA, STAR, or START.

#### 6.4.2.1 STA[RT] Command

The Start command, which is normally the first command issued to DRS, starts the diagnostic from its initial state. All initialization code is executed. The trap catcher code (code that allows DRS to handle any unexpected interrupts and report them to the user) is reloaded into the vector space. The format of the Start command is:

```
STA[RT][switch-list]
```

where "switch-list" is any valid combination of switches (modifiers) for the Start command. The default value is that all tests will run on all units, all flags (see section 6.4.4) will be cleared, and testing will continue until interrupted by the user (^C) or by a system error, and an end-of-pass message will be printed after each pass. A pass is defined to be all specified units tested once by all specified tests.

When the first START command is given, unless the P-tables have previously been set-up or coded in using SETUP, you must answer yes to the change hardware P-Table question (see Section 7.3.3).

#### 6.4.2.2 RES[TART] Command

The Restart command, like the Start command, starts the diagnostic from an initial state. The diagnostic initialization process for a Restart command, however, may differ. The user has the opportunity to change the contents of the software table only; the vector space is not changed. The format of the Restart command is:

```
RES[TART][switch-list]
```

where "switch-list" is any valid combination of switches for the Restart command. The default value is that all tests will be run on all units, all flags are cleared, testing will continue until aborted by the user (^C) or by a system error, and an end-of-pass message will be printed after each pass.

#### 6.4.2.3 CON[TINUE] Command

The Continue command is used to resume diagnostic operation after the user typed control-C (^C) to abort execution or after a halt-on-error. The diagnostic will be restarted at the beginning of the test that was aborted, not at the first test, as would be the case with the RESTART command. The unit being tested when the diagnostic was interrupted will remain as the unit being tested. The user will be given the opportunity to change the software table if desired. The hardware tables cannot be changed. The format of the Continue command is:

```
CON[TINUE][switch-list]
```

where "switch-list" is any valid combination of switches for the Continue command. The default operation of the Continue command is: the testing will run for the number of passes remaining in the pass count specified in the last Start or Restart command. All flags will remain set or clear as previously specified.

The Start and Restart commands can also be used to resume diagnostic execution, but diagnostic initialization will take place and testing will start with the first unit, first test.

#### 6.4.2.4 PRO[CEED] Command

The Proceed command is used exclusively with the halt-on-error feature in DRS. When halt-on-error is in force and the diagnostic reports an error to DRS, DRS returns to command mode. The user may issue any commands at this point. The Proceed command is special in that it restarts the diagnostic at the point where it reported the error. No initialization is done, the unit-under-test is not accessed and the vector space is unchanged. This process allows the user to examine the state of the unit being tested and then to continue testing without disturbing diagnostic operation. The format of the Proceed command is:

```
PRO[CEED][switch-list]
```

where "switch-list" is any valid combination of switches for the Proceed command. The default operation of the Proceed command is: the flags remain set or clear as specified with the previous command.

The following is a short summary of the effects of each command.

START

- trap catcher reloaded
- diagnostic initialize code executed
- user may change both hardware and software tables
- testing will start with first test on first unit
- all flags cleared
- units dropped by program are re-added; units dropped manually are not

RESTART

- trap catcher not reloaded
- some diagnostic initialize code may be executed
- user may change software table only
- testing will start with first test
- all flags are cleared
- units dropped by program are re-added; units dropped manually are not

CONTINUE

- trap catcher not reloaded
- initialize code not executed
- user may change software table only
- testing will start at beginning of interrupted test
- flags remain in previous state

PROCEED

- trap catcher not reloaded
- user may not change hardware or software tables
- test will be resumed immediately after error call
- flags remain in previous state

CONTROL-C

- Cleanup code executed, returns to DRS prompt

#### 6.4.2.5 DRO[P] Command

The Drop command is used to deactivate a unit from testing. The unit to be deactivated must be specified using the UNIT switch. All units are initially active and must be explicitly deactivated by the user or the diagnostic. The units to be deactivated must already be activated for testing. Units dropped by program are re-activated on a START or RESTART while units dropped by an operator using the DROP command are only re-activated by an ADD command.

The format of the Drop command is:

```
DRO[P][ /UNI[TS]:n]
```

where "n" is the number of the unit to be deactivated. The default operation of the Drop command (when the UNIT switch is not specified) is that all active units will be dropped from testing.

#### 6.4.2.6 ADD Command

The Add command is used to activate a unit for testing. The unit switch is used to specify the unit to be activated. All units are initially active and must be explicitly deactivated by the user of the diagnostic. The units to be activated must have already been deactivated. The format of the Add command is:

```
ADD[ /UNI[TS]:n]
```

where "n" is the unit to be activated. The default operation of the Add command (when the UNIT switch is not specified) is to return all deactivated units to active testing.

#### 6.4.2.7 DIS[PLAY] Command

The Display command is used to examine the contents of the hardware tables. All table data for the specified units are listed on the console terminal. Units that have been dropped are so designated. The format of the Display command is:

```
DIS[PLAY][ /switch-list]
```

where "switch-list" is any valid combination of switches for the Display command. The default operation of the Display command is that all units described in the hardware tables will be displayed on the console terminal.

#### 6.4.2.8 FLA[GS] Command

The Flags command is used to display the current status of the DRS flags. It cannot be used to change the state of the flags. This is done using the flags switch (see Section 6.4.3.3). The default flag setting is zero. The format of the Flags command is:

```
FLA[GS]
```

Examples (operator input underlined):

- a. No flags set:

```
DR>FLA  
FLAGS SET  
NONE
```

- b. Two flags set:

```
DR>FLA  
FLAGS SET  
IER  
LOE
```

The FLAGS command cannot be used to set or clear flags, only with START and RESTART commands.

#### 6.4.2.9 ZFL[AGS] Command

The Zflags command resets all DRS flags to their cleared states. The default flag setting is zero. The format of the Zflags command is:

```
ZFL[AGS]
```

When a program is loaded, the initial state of all flags is zero.

#### 6.4.2.10 PRI[NT] Command

The Print command causes the DRS to print the contents of the statistical tables kept by the diagnostic. (Statistical tables are an optional feature and not all diagnostics support them.) The Print command executes the code in the Report coding section. The data are printed on the console terminal in a format specified by the diagnostic program. There are no switches or arguments.

#### 6.4.2.11 EXI[T] Command

The Exit command returns control to the XXDP+ monitor. There are no arguments or switches.



### 6.4.3 DRS Switches

Switches are modifiers of commands. Many DRS commands affect units, with a command of this type affecting all units specified during hardware table build. A switch enables the user to limit the effect of the command to certain selected units. All switches cannot be used with all commands. Table 6-8 lists the DRS switches, their functions, and with which commands they can be used.

Table 6-8. DRS Switches

Switch	Function	DRS Commands Used with
TEST	Execute only specified tests	START, RESTART
PASS	Execute dddd (1 to 65536) passes	START, RESTART, CONTINUE
FLAGS	Set specified flags	START, RESTART, CONTINUE, PROCEED
EOP	Report end-of-pass after each pass	START, RESTART, CONTINUE
UNITS	Command will affect only specified units	START, RESTART, DROP, ADD, DISPLAY

#### 6.4.3.1 TES[TS] Switch

The Tests switch specifies which tests will be run. The default value is to run all tests. The format of the Tests switch is:

```
/TES[TS]:test-list
```

where "test-list" is a list of test numbers separated by colons (:). Sequential test numbers may be specified by giving the first and last test numbers separated by a dash. For example, if Tests 1,2,3, and 4 are to be specified, they may be entered as:

```
1:2:3:4 or 1-4
```

Although the test numbers may be entered in any order, the tests will always be executed in numeric order.

#### 6.4.3.2 PAS[S] Switch

The Pass switch is used to specify the number of passes a diagnostic will run; a pass being defined as all specified tests on all active units. The default value is "no limit". The format of the Pass switch, which allows the user to place a limit on the number of passes, is:

```
/PAS[S]:dddd
```

where dddd is a decimal number between 1 and 65536. When the PASS limit is reached, control is returned to DRS and a prompt is issued.

#### 6.4.3.3 FLA[GS] Switch

The Flags switch is used to set DRS operational flags. The default value is having all flags cleared.

The format for the Flags switch is:

```
/FLA[GS]:flag-list
```

where "flag-list" is a list of DRS flags (see Section 5.4.4) separated by colons (:).

#### 6.4.3.4 EOP Switch

The EOP switch is used to specify when end-of-pass messages will be printed. These messages indicate the number of passes completed and the number of errors found. Default operation is to print these messages after every pass. The format of the EOP switch is:

```
/EOP:dddd
```

where dddd is a decimal number between 1 and 65536. The end-of-pass message is printed after every dddd passes. For example, to have the EOP message printed after every 90 passes (user entry underlined):

```
DR>RES/EOP:90
```

#### 6.4.3.5 UNI[TS] Switch

The Units switch is used to specify which available units are to be tested. Default DRS operation is to encompass all units in the scope of any command. This switch is used to limit the effects of a command to certain units. The format of the Units switch is:

/UNI[TS]:units-list

where "units-list" is a list of unit numbers separated by commas. Unit numbers are decimal numbers from 1 to 64. A Unit is assigned a number based upon order of entry into the tables. The first unit is unit 1. If the units are sequential, they may be specified by the first and last unit number separated by a dash (-). For example, units 3, 4, 5, 6 and 7 may be specified as 3-7.

#### 6.4.3.6 Combining Switches

The user may specify as many valid switches, in any order, with a command as the user desires. Simply string out the switches, one after another, on the command line. For example, if the user wanted to start a diagnostic and:

1. test units 1 through 4 only,
2. execute tests 1, 5 and 15,
3. execute 100 passes and
4. only report the end-of-pass data after every 10 passes,

this is the command that would be given.

STA/UNI:1-4/TES:1:5:15/PAS:100/EOP:10

#### 6.4.4 DRS Flags

Flags are used to set up certain operational parameters such as looping on error. All flags are cleared at startup and remain cleared until explicitly set using the Flags switch. Flags are also cleared after a Start or Restart command unless set using the Flags switch. The Zflags command may also be used to clear all flags. No other commands affect the state of the flags.

Table 6-9 lists the DRS flags and their effects.

Table 6-9. DRS Flags

Flag	Effect
HOE	halt on error - control is returned to runtime services command mode
LOE	loop on error
IER	inhibit all error reports
IBE	inhibit all error reports except first level (first level contains error type, number, PC, test and unit)
IXE	inhibit extended error reports
PRI	direct messages to line printer
PNT	print test number as test executes
BOE	"bell" on error
UAM	unattended mode (no manual intervention)
ISR	inhibit statistical reports (does not apply to diagnostics which do not support statistical reporting)
IDR	inhibit program dropping of units
ADR	execute autodrop code
LOT	loop on test

#### 6.4.4.1 HOE (Halt On Error) Flag

The HOE flag, when set, will cause DRS to execute a "halt-on-error" sequence when an error is detected by the diagnostic. Execution of this sequence does not result in an actual processor halt, but returns DRS to command mode. The exact process is:

1. When the error is reported to DRS, the error message(s) will be printed (unless printing has been inhibited).
2. DRS will return to command mode.
3. The diagnostic will have been suspended at the point of the error report to DRS and the unit being tested will be left in the state that it was in at the time of the call.

After DRS has returned to command mode, the user may issue a Proceed command to resume diagnostic execution at the point where it was suspended. The user may also issue other commands as desired.

#### 6.4.4.2 LOE (Loop On Error) Flag

The LOE flag, when set, will enable DRS error looping. When error looping is in effect, DRS will cause the diagnostic to continually re-execute the code that detected the error. Looping remains in effect even if the symptoms that prompted the error report disappear. This allows for looping on intermittent errors. To stop the looping, the user must type CTRL-C (^C) to return DRS to command mode.

#### 6.4.4.3 IER (Inhibit Error Reports) Flag

The IER flag, when set, causes DRS to inhibit all error reporting to the console terminal. While in effect, no messages will be sent to the operator except system error reports such as ILL INT (illegal interrupt) and end-of-pass reports. This feature is usually used in conjunction with error looping. It speeds up the test process and, in the case of hard copy terminals, saves paper.

#### 6.4.4.4 IBE (Inhibit Basic Errors) Flag

The IBE flag, when set, causes DRS to inhibit the basic and extended portions of error reports. There are three levels of messages in an error report. This is illustrated below:

```
CZRLG DVC FTL ERR 00009 ON UNIT 00 TST 010 SUB 000 PC: 015626 ...error
BIT SET INSTRUCTION ON RLBA YIELDED WRONG RESULT           ...basic
CONTROLLER: 174400 DRIVE: 0                                 ...extended
EXP'D: 000000 REC'D: 000001                                 ...extended
```

#### 6.4.4.5 IXE (Inhibit Extended Errors) Flag

The IXE flag, when set, causes DRS to inhibit the extended error reporting only. The error message and basic reports will be printed.

#### 6.4.4.6 PRI (PRInter) Flag

The PRI flag, when set, causes DRS to redirect all messages to a line printer. This does not apply to command prompts.

#### 6.4.4.7 PNT (Print Number of Test) Flag

The PNT flag, when set, causes DRS to print the number of the test being executed.

#### 6.4.4.8 BOE (Bell On Error) Flag

The BOE flag, when set, causes DRS to issue a CTRL-G, or BELL character when an error is reported by the diagnostic. This will give an audible tone at the console terminal. This feature is usually used in conjunction with the message inhibit functions.

#### 6.4.4.9 UAM (UnAttended Mode) Flag

The UAM flag, when set, prevents the use of manual intervention during testing. Manual intervention assumes that an operator is present to undertake any necessary action. The use of this flag allows the operator to start the diagnostic and let it run unattended. When this flag is in effect, some testing will be inhibited. Refer to specific diagnostic documentation for a description of UAM flag effects in specific cases. See Section 7.5.23 for MANUAL macro for checking the setting of this flag.

#### 6.4.4.10 ISR (Inhibit Statistical Reports) Flag

The ISR flag, when set, causes DRS to inhibit the printing of statistics by the diagnostic. This is an optional feature and not all diagnostics support statistics. Consult specific diagnostic documentation to determine whether or not a diagnostic has this feature.

#### 6.4.4.11 IDR (Inhibit DRopping of units) Flag

The IDR flag, when set, causes DRS to inhibit the execution of the DROP code in the diagnostic, i.e., inhibit deselection of units by a diagnostic. Diagnostics may deselect a unit from the test process if an error threshold is reached or if a serious error is detected. This flag allows the user to keep the unit selected, usually for the purposes of tracing the error.

#### 6.4.4.12 ADR (AutoDRop) Flag

The ADR flag, when set, causes DRS to execute the "autodrop" code in a diagnostic. The purpose of this code is to test for "device ready" or "device available". If the unit being tested is not ready or available, it will be dropped (deselected). Not all diagnostics have autodrop code. Refer to specific diagnostic documentation to determine if a diagnostic does support this feature.

### 6.4.4.13 LOT (Loop On Test) Flag

The LOT flag, when set, causes DRS to continually execute the test(s) specified with the TEST switch. The initialize and end-of-pass code are not executed as in normal operation, however.

TABLE 6-10

#### DRS COMMANDS

```
START [/TESTS:##-#] [/PASS:#] [/FLAGS:XXX] [/EOP:#] [/UNITS:##..]  
RESTART [/TESTS:##-#] [/PASS:#] [/FLAGS:XXX] [/EOP:#] [/UNITS:##..]  
CONTINUE [/PASS:#] [/FLAGS:XXX]  
PROCEED [/FLAGS:XXX]  
DROP [/UNITS:##..]  
ADD [/UNITS:##..]  
PRINT  
DISPLAY [/UNITS:##..]  
FLAGS  
ZFLAGS  
EXIT
```

#### WHERE:

- /TESTS:## Specifies the test #'s to run when all tests in a diagnostic are not wanted.
- /PASS:# Specifies the number of passes to run; if omitted, the diagnostic runs continuously.
- /EOP:# Specifies the number of passes between End-Of-Pass printouts.
- /UNITS:# Specifies the number of the unit to be tested, dropped, or added.
- /FLAGS:XXX Specifies the flag(s) to be set. To specify more than one, use XXX:YYY.

#### Flags available are:

HOE	Halt on Error	LOE	Loop on Error
IER	Inhibit Error Reports	IBE	Inhibit Basic Error Reports
IXE	Inhibit Extended Error	PRI	Print Errors on Line Printer
PNT	Print Test Numbers	BOE	Bell on Error
UAM	Unattended Mode	ISR	Inhibit Statistical Reports
IDU	Inhibit Drop Units	ADR	Auto Drop absent Units
LOT	Loop on Test		

#### Note

Only first 3 characters of any string are needed to enter the command.

6.4.5 XXDP+ Utility Commands

This sections tabulates the various XXDP+ Utility commands as used in UPD2, UPD1, PATCH, SETUP, XTECO. For a complete discussion of these commands refer to the XXDP+ System User's Manual (CHQUSE).

6.4.5.1 UPD2 Command Summary

Table 6-11. Summary of UPD2 Commands

CATEGORY	COMMAND	FUNCTION
File Manipulation	DIR PIP FILE DEL REN	give directory of specified medium transfer a file or files transfer a file or files delete a file or files rename a file
File Modification	CLR LOAD MOD XFR HICORE LOCORE DUMP	clear UPD2 program buffer load a program modify file image in memory set transfer address set upper memory limit for dump set lower memory limit for dump dump a program image
New Medium Creation	ZERO SAVM SAVE COPY	initialize a medium save the bootable monitor image on a random-access device save the bootable monitor image on a sequential-access device copy entire medium
Miscellaneous	ASG DO READ EOT DRIVER	assign a logical name to a device execute an indirect command file read a file to check validity write logical end-of-tape mark on a tape load a device driver
Returning to Monitor	BOOT EXIT	bootstrap a device return control to the runtime monitor
Printing	PRINT TYPE	print a file on the line printer type a file on the console terminal



### 6.4.5.2 UPD1 Command Summary

Table 6-12. Summary of UPD1 Commands

COMMAND	FUNCTION
CLR	clear UPD1 program buffer
LOAD	load a program
MOD	modify file image in memory
XFR	set transfer address
HICORE	set upper memory limit for dump
LOCORE	set lower memory limit for dump
DUMP	dump a program image
DEL	delete a file
BOOT	bootstrap a device

### 6.4.5.3 PATCH Command Summary

Table 6-13. Summary of PATCH Commands

COMMAND	FUNCTION
BOOT	Boot specified device
CLEAR	Clear input table
EXIT	Return to XXDP+ monitor
GETM	Load DEC/X11 MAP file
GETP	Load saved input table
KILL	Delete address from input table
MOD	Enter address in input table
PATCH	Create patched file
SAVP	Save input table
TYPE	Print input table on terminal

### 6.4.5.4 SETUP Command Summary

They are only three commands in the SETUP utility. They are:

- . SETUP
- . LIST
- . EXIT

SETUP is used to build tables for DRS-compatible diagnostics. LIST types a list of DRS-compatible diagnostics on a specific medium, and EXIT returns control to XXDP+.

#### 6.4.5.5 XTECO Command Summary

The three commands used to put XTECO in an edit mode are:

- . TEXT
- . TECO
- . EDIT

TEXT is used to create a new file, TECO is used to modify files only on random-access devices, and EDIT can be used to modify files on any device (providing the input and output devices are different). Once in the edit mode, the twelve commands, listed in Table 6-14 can be used.

Table 6-14. Summary of EDIT Mode Commands

These are all terminated by ESC, ESC instead of Carriage Return.

CATEGORY	COMMAND	FUNCTION
Pointer Location	L	Move the pointer line by line
	C	Move the pointer character by character
	J	Move the pointer to the beginning of text
	ZJ	Move the pointer to the end of text
Search	S	Search for specified string in text now in memory
	N	Search for specified string in remainder of text file
Modify/ Display text	T	Type text
	D	Delete character(s)
	K	Delete line(s)
	I	Insert text
	A	Append text to that currently in memory
Terminating Edit Mode	EX	Exit edit mode

## CHAPTER 7

## DRS-COMPATIBLE DIAGNOSTIC PROGRAMS

This chapter describes the structure of the Diagnostic Runtime Services (DRS) module, as well as the DRS program structure macros and DRS service macros. These macros are defined by a macro library (SVC.MLB) which must be used when assembling a DRS-compatible diagnostic program.

## 7.1 INTRODUCTION

The DRS module is the result of efforts to gather together all the code which interfaces the diagnostic to the operational environment (system software and human operator) and which was formerly contained inside the diagnostic. The code was then put into a common front-end software product that can be appended to each diagnostic and through which the diagnostic may interface to the operational environment as it executes.

## 7.2 DRS PROGRAM BASICS

Any diagnostic, whether DRS-compatible or not, must be loaded and started by a load system of some kind. This load system can be XXDP+ (Field Service operation) or APT/ACT/SLIDE (Manufacturing Operation). DRS was designed as part of the XXDP+ system. Additional software provides DRS' functionality under the other loading systems by emulating XXDP+. In this chapter, the term DRS module will be used in preference to "supervisor".

## 7.2.1 Memory Layout

Table 7-1 illustrates the memory layout on a 16K word machine in a Field Service environment, where DRS runs under XXDP+. If the diagnostic program is less than 7.75K words in length, then the remaining space between DRS and the diagnostic may be used for data buffers.

Table 7-1. Memory Layout

XXDP+	1.5K words	72000 through 77777
DRS	6.25K words	41000 through 71777
Diagnostic	7.75K words	2000 through 40777
Stacks	0.25K words	1000 through 1777
Vector area	0.25K words	0 through 477

### 7.2.2 Different Operating Environment Versions

There are different versions of the DRS to run under various load systems. DRS is a part of the XXDP+ system. There are two additional versions which emulate XXDP+ under the various manufacturing load systems. The different versions of the DRS are illustrated in Table 7-2.

Table 7-2. DRS Versions

DRS Version	Operating Environment
CHSAA	XXDP+
CHSAB	APT
CHSAC	ACT/SLIDE

DRS is appended to the diagnostic at runtime for XXDP+ and APT applications. The DRS software is appended to the diagnostic by a special utility (SETUP) for ACT and SLIDE, the result being a single program image containing both DRS and the diagnostic.

### 7.2.3 Interfacing to the Environment

XXDP+ is the superset of DRS and is the basis for this chapter. Exceptions in the APT/ACT/SLIDE environments are noted where necessary.

**7.2.3.1 Operator Commands** - The Diagnostic must be loaded and started using standard XXDP+ commands. Once the Diagnostic is loaded, the DRS is loaded and given control and will issue its own prompt (DR>). This is known as the DRS Command Mode. In this mode, the operator can issue the commands listed (see Section 6.4.2):

- START
- RESTART
- CONTINUE
- PROCEED
- DROP
- ADD
- DISPLAY
- FLAGS
- ZFLAGS
- PRINT
- EXIT

Note

Only the first three characters of a command need be entered.

Table 7-3. DRS Command Mode Commands

	Function	Command
START	Build hardware and software P-tables and transfer control to the Diagnostic.	
RESTART	Transfer control to the Diagnostic, bypass building of hardware P-tables. Software P-tables may be modified.	
CONTINUE	Resume execution of Diagnostic at beginning of current hardware test. Issued after an operator CONTROL/C or an ERROR HALT.	
PROCEED	Resume execution of Diagnostic at instruction following the error call. Issued after an ERROR HALT.	
DROP	Flag the P-table for the specified logical unit as "Not Complete" and invoke the "Drop Code" in the Diagnostic. The effect of this command lasts until another START command or an ADD command is given. There is also a program drop macro (DODU) with the same effect, but it lasts only for the current command.	
ADD	Add a previously dropped unit back to the test cycle. The "Not Complete" flag is cleared from the P-table and the "Add Code" in the Diagnostic is invoked.	

7.2.3.2 Switches - The operator can, by way of switches on the above commands, affect the value of the following: pass count, test selection, unit selection, and flag settings. Under the supervisor, there is no access to the Hardware Switch Register. The flags (See Section 6.4.4) replace the use of the Hardware and Software Switch Registers. Available flags are:

HOE  
LOE  
IER  
IBE  
IXE  
PRI  
PNT  
BOE  
UAM  
ISR  
IDR  
ADR  
LOT

7.2.3.3 Hardware Parameterization - A major purpose of the DRS is to force parameterization by the operator. A DRS-compatible diagnostic does not autosize or assume standard device addresses but must request from the DRS a hardware parameter table (P-table) for each unit it desires to test. The P-tables are either built in memory by operator dialogue, at RUN-TIME, or they reside in the diagnostic image on the load medium. The latter is accomplished by either assembling the tables into the program or by attaching the tables using the SETUP utility.

Each device type has a different P-table format, but all diagnostics testing the same device must have P-tables of the same format. One diagnostic may have the RK05 Disk Drive as its target device, while another may have the RK11 Controller. The first diagnostic would see a unit as a drive and request an RK05-type P-table for the drive it tests. The second would see a unit as a controller and request an RK11-type P-table for each controller it tests. The P-table format is defined and controlled by the appropriate diagnostic group, using the following established format: the first N entries in any P-table must be the UNIBUS address down through the UNIT address, e.g., the first 3 entries in the TU77 P-table should be: UNIBUS address, MASSBUS address, and drive number.

7.2.3.4 Software Parameterization - Each diagnostic, optionally, has a set of behavioral parameters which are obtained by DRS by way of operator dialogue, before the diagnostic is given control. DRS places these into a software P-table that is hard-coded into the diagnostic. The diagnostic may then examine this table, during execution, to guide its behavior. The data in this table are distinguished from those in the hardware tables in that they relate to operational parameters and do not relate to hardware specific parameters.

7.2.3.5 Passes and Sub-Passes - When a diagnostic is executed, DRS selectively executes pieces of the diagnostic. The order of execution is as follows: initialization (INIT) code, hardware tests, and clean-up code. A single trip through these items is called a sub-pass. The required number of sub-passes, one or more, depends on whether it is a sequential diagnostic or a performance exerciser. When the necessary sub-passes have occurred such that all active units have been through all the hardware tests, a "pass" has been completed. If the DRS determines, after a certain number of sub-passes within a given pass, that all remaining units, as specified in the P-tables, have been dropped (are "non-active"), it will not return control to the INIT code but will declare "END-OF-PASS" at that point. This will also occur if the "/UNITS" switch has been used to select a subset of the units under test and this test does not include the last few units in the p-tables.

### 7.3 DRS PROGRAM STRUCTURE

A supervisor diagnostic has a very specific structure. The diagnostic program can consist of 17 separate sections; 7, which are required, and 10, predicated on expanded program requirements, which are optional. Each section may be contained in an independently assembled module, with the number of sections per module and the manner of linkage being determined by the program. The module calls BGNMOD and ENDMOD are used as initiating and ending directives, respectively. The argument (NAME) provides for a symbolic name to facilitate linkage. BGNMOD has an optional ARG = module name. The 17 sections, defining the structure or skeleton of the diagnostic, are discussed in this section.

### 7.3.1 Program Header (Required)

The Program Header section contains general information which describes the major characteristics of the diagnostic program, including the program name and revision and patch-order levels. It also provides space for an event flag register and pointer storage, by which the supervisor may find access to other key sections of the program (e.g., dispatch table, initialization and clean-up code, etc.). An optional argument allows the programmer to specify the default program priority. The default is zero (0) if not specified.

### 7.3.2 Dispatch Table (Required)

The Dispatch Table section contains address pointers to the various tests contained within the diagnostic program. This section requires the coding of only the Dispatch macro.

### 7.3.3 Default Hardware P-Table (Required)

The Default Hardware P-table coded by the programmer provides the format for the N (number of units) P-tables that DRS will build via operator dialogue, before transferring control to the diagnostic. This table is used as a template to give DRS the size of a P-table and, optionally, the default values of each entry in a table. The actual P-tables are added to the end of the diagnostic image, one table per unit.

### 7.3.4 Software P-Table (Optional)

There is just one Software P-table and it is coded by the programmer to contain any default behavioral parameters needed by the program. It may be modified by the DRS via operator dialogue before control is transferred to the diagnostic. Unlike the hardware P-table template used to create tables, this is the actual table used by DRS.

### 7.3.5 Global Equates (Optional)

This section consists of direct assignment statements which equate specific symbols, that are global in nature, with specific values. The EQUALS macro provides several of the direct assignments of a general nature, including bit definitions for a register and priority level definitions.



### 7.3.6 Global Data (Optional)

The Global Data section contains information that will be used by more than one test.

### 7.3.7 Global Text (Optional)

The Global Text section contains all the ASCII messages that will be used by more than one test and are, therefore, global in nature. The programmer may wish to put all text in this section in order to easily implement foreign languages.

### 7.3.8 Global Error Reports (Optional)

The Global Error Reports section contains the error message subroutines (BGNMSG, ENDMSG) which request the printing of both basic and extended error information for more than one test.

### 7.3.9 Global Subroutines (Optional)

The Global Subroutines section contains subroutines that will be used by more than one test.

### 7.3.10 Statistical Report Coding (Optional)

The Statistical Report Coding section contains the PRINTS macros and associated code that will be used to generate statistical reports. The BGNRPT and ENDRPT macros are used as beginning and ending directives for the coding contained in this section. A DORPT call or a PRINT command from the operator is used to request the execution of this section. If this section was executed as a result of a PRINT command, control is passed back to DRS.

### 7.3.11 Initialization Coding (Required)

The Initialization (INIT) code, executed at the beginning of every sub-pass, is primarily used for requesting P-tables. Any other set-up or initialization type functions may also be performed in the INIT code. There are event flags which the programmer may use to selectively execute code. The event flags indicate which conditions caused the INIT code to be involved: START, RESTART, CONTINUE, POWER FAIL OR NEW PASS (see 7.5.12).

### 7.3.12 Clean-Up Coding (Required)

The Clean-up Coding is executed at the end of every sub-pass, after a CTRL-C, and after a DOCLN macro. It performs any functions necessary to restore the device under test to a power-up state.

### 7.3.13 Drop Units Coding (Optional)

This section is not required for units to be dropped, but is there for the purpose of allowing the programmer to take any program-specific action in the event that a unit is dropped (e.g., making a notation in the statistics table). The Drop Unit code, invoked by a DODU macro or a DROP command, contains any code that needs to be executed in conjunction with the dropping of a unit from the test cycle. Units may still be dropped without having this code. The DRS reports the unit number of the device dropped. No coding is required in this section.

### 7.3.14 Add Units Coding (Optional)

The Add Unit Code, invoked by the ADD command, contains any code that needs to be executed in conjunction with adding a unit back to the test cycle. No coding is required in this section.

#### Note

All units are considered to be ADDED when a diagnostic is started.

### 7.3.15 Hardware Tests (Required)

The Hardware Tests must perform a series of independent free-standing blocks of code with no interdependencies.

### 7.3.16 Hardware Parameter Coding (Required)

The Hardware Parameter section contains the format for the questions that the DRS uses to build the hardware P-tables via operator dialogue and the information necessary to enter the data into the tables.

### 7.3.17 Software Parameter Coding (Optional)

The Software Parameter section contains the questions that the DRS will use to fill in the software P-table via operator dialogue in the same manner as the hardware P-tables.

## 7.4 DRS PROGRAM STRUCTURE MACROS

Since R0 is used by DRS in servicing macro calls, the programmer should not use R0.

### 7.4.1 Optional Sections Selection (POINTER)

A diagnostic program may contain any or all of the 10 optional sections. Five of the optional sections, however, require a pointer that is derived by and for the DRS and is located in the header block (see Section 7.4.2.). The effective use of these 5 pointers requires that the optional sections call must be coded to reflect usage (i.e, ANY, ALL, or NONE). The following coding possibilities exist:

POINTER BGNRPT, BGNSW, BGNSFT, BGNAU, BGNDU, ERRTBL, BGNSETUP (or any subset of the ARGS)

POINTER ALL - ALL provides pointers for all 5 sections.

POINTER NONE - NONE indicates to the DRS that no pointers are required; this is the default value.

### 7.4.2 Header Call (HEADER)

Each diagnostic program must contain a header block which describes the major characteristics of the program for the DRS. The following macro call is used to generate the program header:

HEADER FILNAM,REVLEV,DEPOREV,LONGST,TYPE[,PRIORITY]

FILNAM - FILNAM is ASCII name of program.

REVLEV - REVLEV is ASCII program release level.

DEPOREV - DEPOREV is DEPO (single decimal digit) of latest patch.

LONGST - LONGST is execution time of longest test in seconds (includes clean-up time).

TYPE - "0" for sequential diagnostic, "1" for Exerciser.

PRIORITY - PRIORITY (PSW mask) at which diagnostic will run when started (Default is PRI00).

Example: HEADER CZCLK,A,0,24.,0,#PRI04

This example is from CZCLK, Rev.A, DEPO Rev. 0. Execution time of longest test in 24 (decimal) seconds. It is a sequential diagnostic which starts executing at PRIORITY 4.

All the header arguments are required (except PRIORITY).

#### 7.4.3 Descriptive Text (DESCRIPT, DEVTYPE)

In addition to the Program Name, two lines of text will be printed to the operator. The first will come from the DESCRIPT macro at start-up time and will identify the diagnostics. The second will come from the DEVTYPE macro at hardware dialogue time and will identify the device under test. The arguments of both macros are 72-character ASCII strings enclosed in angle brackets.

DESCRIPT <Identification of Diagnostic Program>

Example: DESCRIPT <DATA COMM LINK TEST>

DEVTYPE <List of device types tested>

Example: DEVTYPE <DMP-11,DMV-11>

#### 7.4.4 Last Address Generation (LASTAD)

The final statement in the program (except for .END) must be the LASTAD macro. This call generates an even address, reflecting the first word of memory unused by the program. It is recommended to add a patch area of about 50 words just before the LASTAD macro call.

#### 7.4.5 Module Delimiters (BGNMOD, ENDMOD)

As discussed in section 7.3, BGNMOD and ENDMOD are used to initiate and end the individual sections.

#### 7.4.6 Test Delimiters (BGNTST, ENDTST)

A block of code which performs a specific diagnostic operation on a given unit is called a test. BGNTST and ENDTST are used to delimit this code, and tests may not be nested. Following the INIT code, a test is initiated by the DRS as an independent block of coding, although the use of global information may be required by any given test. Nothing done in one test must depend on anything done in any other test and no branching from one test to another is permitted.

If more than one test is contained within a module, the tests will be sequentially numbered and run as directed by DRS. By using the optional argument with a Begin Test (BGNTST) macro, a test number may be specifically assigned. DRS will always place the number of the test currently being executed into the header word L\$TEST.

Example 1: BGNTST

```
.  
. code  
. .  
. .  
ENDTST
```

Example 2: BGNTST 5

```
.  
. code  
. .  
ENDTST
```

#### 7.4.7 Subtest Delimiters (BGNSUB, ENDSUB)

A subtest is an optional block of code contained in a test. It is delimited by the directives BGNSUB, ENDSUB. BGNSUB takes no argument. Subtests may not be nested. Subtests are provided for tighter error loop control (see Section 7.5.4).

#### 7.4.8 Segment Delimiters (BGNSEG, ENDSEG)

A segment is an optional block of code within a test or subtest. Each segment utilizes an initiating (BGNSEG) and ending (ENDSEG) directive, in order to define minimum areas of responsibility within a test. Segments are the only blocks of test code that can be logically nested. Up to 8 segmented levels may be logically contained within a single segment. This provides considerable flexibility in the design of program loops for the monitoring of error conditions. Segments are also provided for tighter error loop control (see Section 7.5.4).

#### 7.4.9 Hard-Coded P-Tables

When present, these optional hardware P-tables are located between the LASTAD macro and the ".END" statement. These hardware P-tables, in contrast to the default hardware P-table located in the main body of the program, are appended to the binary image file of the diagnostic, just as though the DRS or the SETUP utility had built them there. Thus, the diagnostic can be "preparameterized" by the programmer. A sample is as follows (this is not code):

```
LASTAD
BGNSETUP 2 ;NUMBER OF P-TABLES
BGNPTAB
.WORD 176500 ;1ST UNIT CSR
.WORD 300 ;1ST UNIT VECTOR
.WORD 240 ;1ST UNIT PRIORITY
ENDPTAB
BGNPTAB
.WORD 176510 ;2ND UNIT CSR
.WORD 304 ;2ND UNIT VECTOR
.WORD 240 ;2ND UNIT PRIORITY
ENDPTAB
ENDSETUP
.END
```

#### 7.5 DRS SERVICE MACROS

All services that a diagnostic requires from DRS are called for by issuing an appropriate macro. This section describes these macros and provides examples. The interface between diagnostics and DRS is based on the TRAP instruction.

Use of R0: the programmer should not rely on the contents of register R0 across macro calls. R0 is used by DRS.

##### 7.5.1 Macro Package Initialization (SVC)

Macro Package Initialization (SVC statement) must be the first macro call of every program. The SVC statement initializes the DRS macro package, thereby allowing subsequent calls to assemble properly.

##### 7.5.2 Global Equates (EQUALS)

The Global Equates call equates specific symbols with specific values by way of direct assignment statements. It expands to give mnemonics for bit definitions, event flag definitions, and priority level definitions.

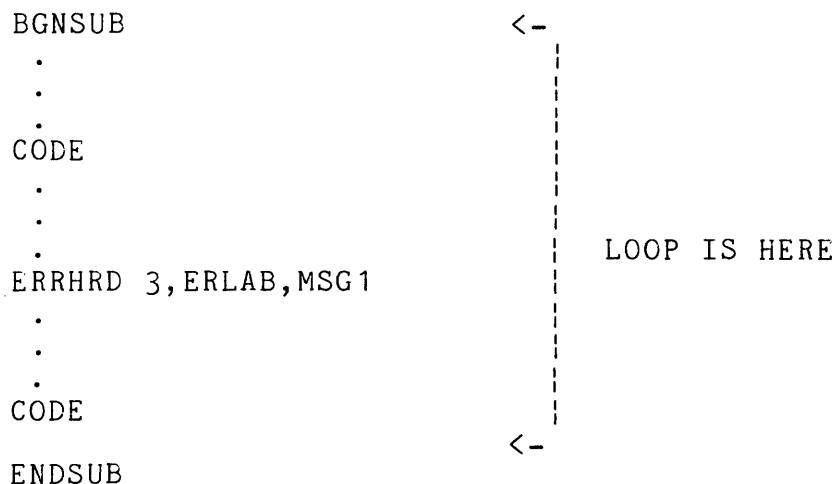
### 7.5.3 Test Dispatch Table (DISPATCH)

The DISPATCH call is used to produce a table containing the symbolic address (e.g., T1, T2, etc.) of each test contained in the diagnostic program. When the program is assembled, the addresses for the table are obtained from the Begin Test (BGNTST) statements. The Dispatch call must contain an argument to equal the decimal number of tests or additional tests will not be executed.

### 7.5.4 Error Loop Control (CKLOOP)

When the operator executes a diagnostic, the operator has the ability to set a "loop on error" flag. When this flag is set, the occurrence of an error will cause looping in one of two ways, depending on the type of CKLOOP statement that was coded or implied. Implied CKLOOP (7.5.4.2) is the normal usage.

7.5.4.1 Implied CKLOOP - This situation exists when the programmer does not code a CKLOOP statement. In this case, when an error call is issued, execution proceeds to the end of the smallest scope entity (test, subtest, or segment) that contains the error call, at which point a branch is created back to the beginning of that scope entity. This loop is permanently established, even if the error does not persist.



7.5.4.2 Explicit CKLOOP - This situation exists when the programmer has inserted a CKLOOP call (no ARGS) into the code. In this case, when an error call is issued, execution of the test code will continue down through the CKLOOP statement, at which point a branch is created back to the beginning of the smallest scope entity (test, subtest, or segment) that contains the error call. This loop is permanently established, even if the error that was reported does not persist.





### 7.5.6 Abort Test Calls (ESCAPE, EXIT)

There are two Abort Test calls. The first is a conditional escape call, while the second is an unconditional exit call. Both calls share the same arguments (TST, SUB, SEG) when directing an appropriate exit from test coding. Additional arguments (HRD, SFT, INIT, CLN, RPT, SRV, MSG) are available to the exit call to provide an unconditional exit from certain selected routines.

**7.5.6.1 Escape Test (ESCAPE TST, SUB, SEG)** - If an error is detected within a test, subtest, or segment; the error could invalidate results obtained from the execution of subsequent test code within the structure. To prevent this, the programmer has the option of including an ESCAPE statement which utilizes an argument (TST, SUB, SEG) to create a branch from the test code to the next ENDTST, ENDSUB, or ENDSEG statement. This escape may or may not occur if an error was detected, depending on the state of the Loop on Error flag. If the Loop on Error flag is set, the escape will cause a branch back to the beginning of that entity.

**7.5.6.2 Exit Test (EXIT TST, SUB, SEG)** - The EXIT macro used inside a test has the force of an unconditional escape: a branch is created to the next ENDTST/SUB/SEG statement, whether or not an error has occurred. If an error condition does exist when the exit statement is encountered, however, that error indication is cleared so looping will not occur even if the Loop on Error flag is set.

**7.5.6.3 Exit Routine (EXIT HRD, SFT, INIT, CLN, RPT, SRV, MSG)** - When EXIT is used with any one of additional arguments (HRD, SFT, INIT, CLN, RPT, SRV, MFG) an unconditional escape from the specified routine to its associated ending directive (i.e., ENDHRD, ENDSFT, ENDINIT, ENDCLN, ENDRPT, ENDSRV, ENDMSG) will occur, providing termination for one of the following routines:

- Hardware Parameter Coding (HRD)
- Software Parameter Coding (SFT)
- Initialization Coding (INIT)
- Clean-up Coding (CLN)
- Statistical Report Coding (RPT)
- Interrupt Service Coding (SRV)
- Print Message Coding (MSG)

### 7.5.7 Error Reporting (ERRSF, ERRDF, ERRHRD, ERRSOFT, ERROR, ERRTBL)

#### 7.5.7.1 Error Report Classes

Four error report calls are used within test coding to report individual errors. Errors are classified as soft or hard. A soft error is an error which is recoverable within a specific number of retries. There are two classes of hard errors:

- a. overflow of the acceptable limit of retries for a soft error
- b. a non-recoverable error

Neither of the above mean that the device has failed. In order for a diagnostic to determine device failure, based on the number of soft or hard errors, it must contain code to make a "DEVICE FATAL" error call when the acceptable limits are exceeded. The call must also be made where a single error indicates that the device has failed.

DEVICE FATAL - device failure

SYSTEM FATAL - system failure

Adherence to these definitions is important because soft and hard errors will be ignored when running under an automated system. Four error report calls are used within test coding to report individual errors (Table 7-5).

Table 7-5. Error Report Calls

Report Call		Function
ERRSF	NUM,MSGADR, POINTER	Reports a system fatal error
ERRDF	NUM,MSGADR, POINTER	Reports a device fatal error
ERRHRD	NUM,MSGADR, POINTER	Reports a hard error(unrecoverable)
ERRSOFT	NUM,MSGADR, POINTER	Reports a soft error (recoverable)

When an error is detected by the test code, the associated report call:

- reports the error by setting an error indicator, which provides one of the two enables required to allow a loop-on-error to occur (The other enable is the loop-on-error flag).

- . enables the printing, via optional parameter coding, of a combination of test identification information ( i.e., header; test, subtest, and error information; and any additional identifying information).

### 7.5.7.2 Report Call Arguments

#### 7.5.7.2.1 Explicit Arguments

ERROR NUMBER PARAMETER (NUM) requires one full word of memory and provides a unique number for each error message. The maximum error number is 32767 and it is passed as a decimal number.

MESSAGE ADDRESS PARAMETER (MSGADR), an optional argument, requires one full word of memory. The argument provides a symbolic address for a message, created via an assembler directive (.ASCIZ), that will be printed as part of the header, an to which DRS will add test and subtest numbers.

MESSAGE POINTER PARAMETER (POINTER), an optional argument, requires one full word of memory. The argument provides a symbolic address for specific subroutine coding that will be executed following the printing of the header. The subroutine coding, which could consist of macro calls that provide for the printing of additional error information, could also initialize a portion of a test for use in a test loop. The first and final instructions in the subroutine, however, must be BGNMSG and ENDMSG calls, respectively.

#### 7.5.7.2.2 Implicit Argument (Exerciser Diagnostics Only)

UNIT NUMBER REPORTING - DRS prints the error number of the failing device, so it is unnecessary for the diagnostics to do so. L\$LUN, in the header, contains the number of the unit currently being tested. However, in the case of a performance exerciser, DRS needs to be told the failing unit number, since several devices are being tested at once. This is done by entering the failing unit number into the header item L\$LUN (0 to 63, decimal) prior to each error call.

### 7.5.7.3 Error Tables

Another pair of macros is provided for the programmer who needs to dynamically modify either the error number, the message address, or the message blocks pointer. The error is reported by first filling in a table generated by the ERR\_TBL macros, which takes no argument and expands to the following:

	<u>ERRTYPE Values</u>
ERRTYP:: .WORD	0 = System Fatal
ERRNBR:: .WORD	1 = Device Fatal
ERRMSG:: .WORD	2 = Error Hard
ERRBLK:: .WORD	3 = Error Soft

After filling the table, the programmer must issue the macro "ERROR" with no arguments. This will cause the above table to be accessed. Only one "ERRTBL" macro can be coded per program.

### 7.5.8 Printing Messages (BGNMSG, ENDMSG, PRINTB, PRINTX, PRINTF)

7.5.8.1 Message Printout Format - An error message consists of four separate parts:

- . A header of test identification information (TEST I/D) from DRS.
- . A programmer specified error message (optional).
- . Basic error (BASIC ERROR) information (optional).
- . Extended error (EXTENDED ERROR) information (optional).

For example:

```
CZRLG DVC FTL ERR 00009 ON UNIT 00 TST 010 SUB 000 PC: 015626 ...error
BIT SET INSTRUCTION ON RLBA YIELDED WRONG RESULT           ...basic
CONTROLLER: 174400 DRIVE: 0                                ...extended
EXP'D: 000000 REC'D: 000001                                ...extended
```

Test Identification Information (TEST I/D):

The test I/D information is error message header data from DRS. This includes the error type, the test, subtest numbers, unit, pass, pass count, etc.

Programmer Specified Error Message:

One-line ASCII message describing the error in general terms.

Basic Error Information (BASIC ERROR):

The basic error information is provided by the diagnostic program via subroutines bounded by the BGNMSG, ENDMSG macros. This information must be printed with a PRINTB macro. These messages describe the nature of the error, including the contents of device registers.

Extended Error Information (EXTENDED ERROR):

The extended error information is provided by the diagnostic program via subroutines contained in the message section and bounded by the BGNMSG, ENDMSG macros. This information must be printed with a PRINTX macro. The messages describe certain program and/or hardware conditions that the diagnostic engineer deems necessary (e.g., number of bytes transferred, etc.).

7.5.8.2 Begin and End Message Calls (BGNMSG, ENDMSG) - The begin and end error message directives are used to initiate the code required to print the basic and extended error message information following the header print-out. BGNMSG requires an ARG = Name of This Message Routine. This is the MSGADR argument for error report calls (see Section 7.5.7.2).

7.5.8.3 Basic and Extended Print Message Calls (PRINTB, PRINTX, PRINTF) - The basic (PRINTB) and extended (PRINTX) and unconditional (PRINTF) error calls have the same format, so the following coding example will use the basic error print call. However, if the inhibit basic error flag is set, the PRINTB call cannot enable the print-out of basic information, and if either the inhibit basic error flag or the inhibit extended error flag is set, the PRINTX call cannot enable the print-out of extended information.

The print error message macro calls are as follows:

```
PRINTB #FORMAT,DAT1,...,DAT8
```

where DAT1 thru DAT8 are data arguments (up to a maximum of 8) to be printed in accordance with the specifications contained in the format statements. The data arguments are word items which may be specified by any addressing mode (i.e., R2, @R1, (R3)+, etc). The first argument placed on the stack is executed last and vice versa.

The format expression is as follows:

```
FORMAT: .ASCIZ /%D%D%D...../ ;% is a directive (D) delimiter
```

The format directives (D) may be coded as follows:

A = ASCII	;Pickup and copy characters from the ;format statement to the output buffer ;until the next print limit (%) is ;encountered.
T = ASCII	;Pickup and copy characters from an ;external area. The next data argument ;contains a pointer to the area.
N<N> = NEW LINE	;Insert N CR/LF's into the output buffer.

S<N> = SPACE ;Insert N spaces in output buffer  
;(1<=N<=9).

O<N> = OCTAL ;Convert next data argument to OCTAL  
;ASCII. If N is less than or equal to  
;6, only N ASCII characters will be  
;provided as output. If N is greater  
;than 6, leading ASCII spaces will be  
;supplied

B<N> = BINARY ;Convert next data argument to binary  
;ASCII. If N is less than or equal to  
;16, only N ASCII characters will be  
;provided as output. If N is greater  
;than 16, leading ASCII spaces will be  
;supplied.

D<N> = DECIMAL ;Convert next data argument to unsigned  
;decimal ASCII. If N is greater than the  
;actual number of characters output, and  
;that output is less than 5, leading  
;spaces will satisfy the difference  
;within the first 5 characters.

Z<N> = ZEROS ;Convert next data argument to unsigned  
;decimal ASCII, If N is greater than the  
;actual number of characters output, and  
;that output is less than 5, leading  
;zeros will satisfy the difference within  
;the first 5 characters; where N is  
;greater than 5, leading spaces up to N-5  
;will be supplied.

It should be noted that DRS allows a maximum line length of 72 characters. In addition, if any of the delimited directives are incorrectly coded, DRS will print an error message (?).

Example:

EXAMPLE: TO PRODUCE THE MESSAGE

ERROR AT CSR 177500 DRIVE 2

USE THE FOLLOWING CODE:

```
MOV          #177500,CSR
MOV          #2,DRIVE
PRINTB      #MSG,CSR,DRIVE
MSG:        .ASCIZ /%N%AERROR AT CSR %06%A DRIVE %01/
CSR:        .WORD 0
DRIVE:      .WORD 0
```

Print Forced Message call (PRINTF) - the forced message print call allows the programmer to override the print inhibit flags (e.g., to issue a warning message). The call utilizes the same format as the basic error print call (PRINTB), but does not require special subroutine coding. For printing a byte value, the data argument (DAT1 to DAT8) looks like <B,DAT1>, for example:

```
UNIT:          PRINTB          #FORMAT, <B,DRIVE>
DRIVE:         .BYTE 0
```

### 7.5.9 Statistical Reporting (BGNRPT, ENDRPT, PRINTS, DORPT)

If a report coding section is included in the design of the diagnostic program, a statistical report can be printed, via subroutine, which relates to the performance of the diagnostic tests (e.g., number of errors occurring per read or write operation, etc.). Thus, a report may be included, and formatted, at the programmer's discretion. In any case, the coding of a statistical print-out subroutine must be initiated (BGNRPT) and ended (ENDRPT) by report directives, while the required print (PRINTS) call is formatted in the same manner as the basic error print call (PRINTB) described above.

The Print Statistical Report calls are as follows:

```
PRINTS #FORMAT, DAT1,....,DAT8
```

It should be noted that if an Inhibit Statistical Report flag is set, a print-out will not occur when a PRINTS call is executed.

Do Report call (DORPT)

Do Report (DORPT) is a separate call that can be independently coded with a diagnostic structure to affect the printing of a statistical report. This call is inhibited via setting of the "ISR" control flag by the operator. Reports can be initiated by an operator via the PRINT command.

### 7.5.10 Branching (BERROR, BNERROR, BCOMPLETE, BNCOMPLETE)

DRS services performed for a diagnostic program can result in the return of an error indication via the setting of an error flag. To alter the sequencing of the diagnostic, dependent on the condition of the error indicator, a branch on error (BERROR) and branch on no error (BNERROR) call are provided which allow the program to predicate a branch on the set or reset condition of the indicator. Branch capability is provided by a single argument (Label), which provides each call with a symbolic label to which program sequencing is directed if the tested condition is true.

The Error Branch Macros are as follows:

```

BERROR LABEL           ;Branch to label if error indicator is
                        ;set (generates BCS inst.)

BNERROR LABEL          ;Branch to label if error indicator is
                        ;reset (generates BCC inst.)

```

DRS can also return a Function Complete indication via the setting of a complete flag. To alter program sequencing, dependent on the condition of the complete indicator, a branch on complete (BCOMPLETE) and branch on not complete (BNCOMPLETE) call are provided to predicate a branch on the set or reset condition of the indicator, respectively. Branch capability is provided by the argument (label), which provides for program redirection if the tested condition is true.

The Function Complete Branch Macros are as follows:

```

BCOMPLETE LABEL        ;Branch if function complete indicator is
                        ;set (generate BCC inst.)

BNCOMPLETE LABEL       ;Branch if function complete indicator is
                        ;reset (generate BCS inst.)

```

### 7.5.11 Clock Macro

If a programmer needs to do critical timing he can use either the "L" or the "P" clocks, if available, as follows:

CLOCK TYPE, PTR

```

For example:  CLOCK      L,R1           ;Look for a line clock.
               BNCOMPLETE CLK          ;Branch if none there.
               MOV       (R1)+,CLK      ;Save clock CSR address.

```

The ARG "Type" is either an "L" or a "P" and causes the address of the corresponding clock table to be passed back in the ARG "PTR". The format of this four word table is:

```

.WORD CSR           ;CLOCK ADDRESS
.WORD BR LEVEL      ;CLOCK BR LEVEL
.WORD VECTOR        ;CLOCK VECTOR
.WORD HERTZ         ;LINE FREQUENCY (L CLOCK ONLY)

```

Notes:

- a. The diagnostic should not call the DRS during a fine timing operation.



- b. The diagnostic should not use the clock for watchdog delays and should return to the DRS periodically during these delays.

### 7.5.12 Event Flags (READEF)

Read event flag call (READEF ARG) - the read event flag call is used to test the condition of an event flag bit specified by the associated argument. If the flag bit is set, a function complete indicator will be returned for testing via a branch complete (BCOMPLETE) call. The act of reading clears an event flag.

The read event flag macro is as follows:

```
READEF #ARG                ;Read condition of EF bit specified
                           ;by ARG.
                           ;ARG provides decimal number (1-32)
                           ;by any legal address mode.
                           ;If EF bit is set, return function
                           ;complete indicator and clear EF bit.
```

There are situations where the diagnostic needs to know what command invoked it, and for this purpose there are the following event flags which can be read by the program:

EF.START	Start command was issued
EF.RESTART	Restart command was issued
EF.CONTINUE	Continue command was issued
EF.NEW	New pass is being commenced after all
active units have been accessed (always	set on a new pass, even if EF.START or
	EF.RESTART is also set).
EF.PWR	Power is coming back up after failure
	(a power up message is printed by DRS)

DRS sets these flags to their correct values (and clears any that are no longer true) at each entry to the INIT code (i.e., at each sub-pass). The act of reading one of these event flags clears it. These flags are only valid during the execution of the INIT code.

### 7.5.13 Unit Selection (BGNAU, ENDAU, BGNDU, DODU, ENDDU)

If the diagnostic engineer wishes to include additional device-units in a test cycle, or delete specific units from a cycle, separate add unit (BGNAU, ENDAU) or drop unit (BGNDU, ENDDU) directives must be used to define the necessary coding (if any) while separate do drop unit (DODU) calls must be coded to effect execution. When the add or drop code is invoked, R0 will contain the logical unit number.

Note

All units are in the ADDED state when a diagnostic is started.

7.5.13.1 Adding Units (BGNAU, ENDAU) - Units may be added to the test sequence only through the use of operator add command (ADD). Each unit must have a P-table in memory due to an earlier hardware dialogue (i.e., the unit was previously dropped). The add code must be delimited by BGNAU, ENDAU. There is no particular coding required in the add code to cause the add to be effective: the section is just for program housekeeping.

7.5.13.2 Dropping Units (BGNDU, ENDDU, DODU) - The drop code, delimited by BGNDU...ENDU is invoked by either the operator drop command or by the issuance of the DODU macro. There is no particular coding required in the drop code to cause the drop to take effect in either case: the section is merely for program housekeeping.

Do Drop Unit Macro (DODU)

The effect of a DODU is dependent upon whether it is executed in the INIT code or in a hardware test. It invokes the drop unit coding and causes subsequent GPHARDS for the logical unit to be returned "not complete". This effect lasts only for the duration of the current command. See 7.5.20 for sample DODU. If the DODU is executed in the INIT section, the run is aborted and the DRS returns to prompt mode. If the DODU is executed in a test, control is returned to the diagnostic immediately after the DODU call. A DOCLN call is typically executed next (see section 7.5.22). This forces execution of the cleanup code.

7.5.14 Default Hardware P-Table (BGNHW, ENHWH)

The default hardware P-table serves as the template from which DRS builds identically formatted tables (via operator dialogue), one for each unit specified. The default hardware P-table is hard coded into the diagnostic (unlike the hardware P-tables which are built by DRS following the LASTAD macro (see Section 7.4.4) with beginning and ending directives BGNHW and ENHWH).

The hardware P-table may be structured differently for each device being tested. All diagnostic programs testing a particular device should use the same format for the hardware portion of the P-table. The standard P-table format requires that the first N words of the P-table be the device addresses from Unibus address down through unit address. The first N words of the P-table should look like this:

```
.WORD CSR1  
.WORD Next level of device address (e.g., massbus address)  
-----  
-----  
-----  
.WORD Unit address (e.g., drive number)
```

Additional words can contain additional hardware parameters if needed.

#### 7.5.15 Software P-Table (BGNSW, ENDSW)

The software P-table is coded into the program with initiating (BGNSW) and ending (ENDSW) directives. Unlike the default hardware P-table, this is an actual table used by DRS and not a template. Since table information relates to program behavior, each entry is formatted at the engineer's discretion. During dialogue with the operator, the assembled entries may be altered by operator input, or left intact via the execution of a carriage return, if the default-select on the GPRMxx call was YES. Thus, the assembled entries are the software default status. The BGNSW has an optional ARG = label of first entry of software P-table. This label can be used by the programmer to access the table.

#### 7.5.16 Hardware P-Table Questions (BGNHRD, ENDHRD)

Hardware parameter coding is assembled with the diagnostic program, utilizing initiating (BGNHRD) and ending (ENDHRD) directives. The parameter coding calls (GPRMD, GPRMA, GPRML, XFERs), that are required by the DRS to obtain desired hardware parameters from the operator, are placed in this section and are interpreted by the DRS in order to query the operator for hardware data during the initial load phase of the diagnostic. During this dialogue, the operator may enter ↑Z, which terminates dialogue and takes default values for the rest of the entire table. Also included in this section are the ASCIZ messages that are required to establish a dialogue with the operator. Data thus obtained by the DRS is placed in the P-tables.

### 7.5.17 Software P-Table Questions (BGNSFT, ENDSFT)

Optional software parameter coding may be assembled with the diagnostic program, utilizing initiating (BGNSFT) and ending (ENDSFT) directives. The parameter coding calls (GPRMD, GPRMA, GPRML, XFERs), that are required by the DRS to obtain desired software parameters from the operator, are placed in this section, and interpreted by the DRS in order to query the operator for hardware data during the initial load phase of the diagnostic. During this dialogue, the operator may enter ↑Z, which terminates dialogue and takes default values for the rest of the entire table. Also included in this section are the ASCIZ messages that are required to establish a dialogue with the operator. Data thus obtained by the DRS is placed in the software P-table.

### 7.5.18 Parameter Coding Calls (GPRMD, GPRMA, GPRML)

These calls are the means whereby questions are posed to the operator and the answers received are placed into the appropriate hardware or software P-table. There are three kinds of calls:

GPRMD	GET DATA
GPRMA	GET ADDRESS
GPRML	GET LOGICAL (YES/NO)

7.5.18.1 GPRMD Call - Data - The get parameter data call is as follows:

```
GPRMD MSGADR,OFFSET,RADIX,MASK,LOLIM,HILIM,DEFAULT-SELECT

MSGADR      ;Msgadr is address of message to be printed.
OFFSET      ;Offset is a relative data byte position in
            ;table.
RADIX       ;Radix is the base of the input value, decimal
            ;(D), octal (O), or binary (B).
MASK        ;Mask is bit position to which the data is
            ;justified prior to packing.
LOLIM       ;LOLIM is the lowest allowed unsigned value.
HILIM       ;HILIM is the highest allowed unsigned value.
DEFAULT-SELECT ;DEFAULT-SELECT provides a logical choice for
            ;default (YES), or no default (NO).
```

The information obtained by the get parameter data (GPRMD) call is placed in either a hardware or software P-table, as directed by the arguments. The call can obtain from the operator either even or odd valued parameters.

. The MSGADR argument provides DRS with the symbolic address of the ASCIZ message that is used to provide the question required to initiate dialogue with the operator.

. The OFFSET argument provides DRS with a byte location that is relative to the base location of the P-table by the offset value, and into which the current parameter will be placed.

. The RADIX argument symbolically defines the base value of the number system (D,O,B) that must be used by the operator in order to correctly input numerical information. To ensure proper operation, DRS checks the RADIX of the input data.

. The MASK argument provides DRS with the location of the consecutive bit positions, within a 16-bit word, into which the value of a parameter will be right justified in the P-table. For example, if the mask is a 36 octal (000036), the one bits contained in the 16-bit argument (0 000 000 000 011 110) define the desired location of the 4-bit parameter as bit positions 1-4 of the lower byte of the word.

. The LOLIM and HILIM arguments provide DRS with unsigned numerical values which define the minimum and maximum input values desired. To ensure proper operation, DRS will compare the limit values with the values entered by the operator. However, if the programmer desires to utilize a previous P-table entry as a limit, a special syntax (<<@VALUE>) may be used in place of the argument, the value of which is used to define the offset required to obtain the previous limit.

. The DEFAULT-SELECT argument provides DRS with a logical switch, which allows the supervisor to define either an actual operator input or a default value for the P-table. The actual coding of the default value must be YES or NO; if DEFAULT-SELECT is YES, the default value will be printed, and the operator may then enter a carriage return to select the default value, or a new value followed by a carriage return. If the default-select is NO, the default value will not be printed and the operator must enter a new value and carriage return. For the software P-table, the default value is what is contained in the table. For the hardware P-table, the default value is initially the value contained in the table. After that, the latest value is the default value.

7.5.18.2 GPRMA CALL - Address - The get parameter address call is as follows:

GPRMA MSGADR, OFFSET, RADIX, LOLIM, HILIM, DEFAULT-SELECT

MSGADR	;same as GPRMD
OFFSET	;same as GPRMD
RADIX	;same as GPRMD
LOLIM	;same as GPRMD
HILIM	;same as GPRMD
DEFAULT-SELECT	;same as GPRMD

The get parameter address (GPRMA) call is used to obtain even valued, unsigned parameter addresses from the operator. However, as previously implied, if odd valued addresses are desired the programmer must use the data call (GPRMD). The address call (GPRMA) arguments are used in the same manner as the data call arguments with one exception - the address call does not require the coding of a mask argument since all addresses will be word values.

#### Exception Capability

There is an additional feature available on the GPRMD and GPRMA calls, known as the "Exception Bit". By using it, the diagnostic is able to indicate that the desired low and high limits are not constants, but are in the form of relative offsets to previous entries in the P-table. In other words, these entries were put into the table as responses to previous questions. In the following example, the first hardware question asks for the low limit, the second asks for the high limit, and the third question is the one that uses those limits.

Note that the offsets in the exception coding are byte offsets from the beginning of the P-table to the location which contains the high or low limit.

BGNHW	
.WORD	0
.WORD	200
.WORD	100
ENDHW	
BGNHRD	
GPRMD	LOW, 0, D, 177777, 0, 100, YES
GPRMD	HIGH, 2, D, 177777, 101, 200, YES
GPRMD	VAL, 4, D, 177777, <@0>, <@2>, YES
ENDHRD	
LOW:	.ASCIZ /LOW LIMIT/
HIGH:	ASCIZ /HIGH LIMIT/
VAL:	.ASCIZ /VALUE DESIRED/
	.EVEN

7.5.18.3 GPRML Call - Logical - The Get Parameter, Logical call is as follows:

GPRML MSGADR,OFFSET,MASK,DEFAULT-SELECT

MSGADR	;same as GPRMD
OFFSET	;same as GPRMD
MASK	;same as GPRMD
DEFAULT-SELECT	;same as GPRMD

The Get Parameter Logical (GPRML) call is used to obtain a logical YES or NO response from the operator. Either lower or upper case values will be accepted by the supervisor. The arguments associated with the call are used in the same manner as the data call (GPRMD) arguments. However, the RADIX and LIMIT (LOLIM, HILIM) arguments are not required. The response is stored in a single bit. The bit will be set to a 1 for yes and 0 for no. Up to sixteen responses may be stored in a single word.

#### 7.5.18.4 COUNT Macro (COUNT arg)

An optional COUNT argument is permitted in the software dialogue only. By appending a COUNT value to the argument string of any of the above three macros, the program can cause the question to be asked a variable number of times. The actual value of the count argument is an offset into the software P-table. The entry so referred to must have been filled by a previous GPRMD call without a count argument. When the GPRMx macro with the count argument is processed, the value at (BGNSW plus COUNT) will be used to determine how many times the question will be asked. The answers will be placed (one word each) starting at the offset specified.

#### 7.5.18.5 DISPLAY Macro (DISPLAY arg)

The program may cause a one-time display of a piece of text by inserting a DISPLAY macro into his parameter gathering code. The argument is a pointer to a multi-line message where each line is an ASCII directive and the entirety is terminated by a .BYTE 0 or ASCIIZ directive.

#### 7.5.19 Transfer Calls (XFER)

Three transfer calls allow program control to be transferred to the location defined by the label associated with each transfer call. These calls may be used to sample the response received from the operator to the GPRML calls. Branching is forward only. Table 7-6 lists the transfer calls.

Table 7-6. Transfer Calls

Transfer Calls	Function
XFERT LABEL	If last GPRML input is true (Y), transfer control to LABEL
XFERF LABEL	If last GRPML input is false (N) transfer control to LABEL
XFER LABEL	Unconditionally transfer control to LABEL

The following provides an example of dialogue coding for construction of the P-table.

```

BGNHRD                                ;Begin hardware parameter code

GPRMD G1,0,0,160000,0,7,NO            ;Get a unit number (0-7). Place in
                                        ;upper 3 bits of P-table, at word 0,
                                        ;No default value is allowed.

GPRMA G2,2,0,177000,177776,YES        ;Get addr. of regs (177000-177776)
                                        ;place in word 2 of the table.
                                        ;Default value is from default
                                        ;table.

GPRMA G3,4,0,0,776,YES                ;Get the vector addr (octal 0-7).
                                        ;Place in word 4.
                                        ;Default value is from default
                                        ;table.

GPRMD G4,6,D,-1,0,6,YES               ;Get interrupt priority (0-6).
                                        ;Place in word 6.
                                        ;Default value is from default
                                        ;table.

GPRML G5,10,-1,YES                   ;Get logical switch (yes or no)
                                        ;place in word 10.
                                        ;Default value is from default
                                        ;table.

ENDHRD                                ;End hardware parameter code.

G1:  .ASCIZ    /UNIT NUMBER/
G2:  .ASCIZ    /DEVICE ADDRESS/
G3:  .ASCIZ    /INTERRUPT VECTOR ADDRESS/
G4:  .ASCIZ    /PRIORITY LEVEL/
G5:  .ASCIZ    /RK05F/

```



### 7.5.20 Request Table (GPHARD)

The GPHARD is issued in the initialization code with two arguments:

```
GPHARD LUNBR,PLOC
```

DRS places the address of the P-table for logical unit LUNBR to be placed at location PLOC. The arguments may be in the form of any addressing mode. Note that no movement of the P-table takes place.

The return from GPHARD indicates to the program whether the P-table requested was available or not by means of the "COMPLETE" or "NOT COMPLETE" indicator. The P-table will be unavailable if:

- A. This logical unit was dropped by the operator.
- B. This logical unit was dropped by the program (See DODU).
- C. The "/UNITS" switch was supplied on a restart command and this logical unit wasn't mentioned.

GPHARD calls must be issued in the INIT code section in order for the DRS to keep track of passes.

For a sample of GPHARD calls, see 7.5.21.

### 7.5.21 Initialization (BGNINIT, ENDINIT)

Diagnostic initialization coding is assembled with the diagnostic program utilizing initiating (BGNINIT) and ending (ENDINIT) directives.

The INIT code is invoked at the beginning of every sub-pass (see 1.2), when a START, RESTART, or CONTINUE command is issued and after a powerfail/autorestart. Following are two samples of INIT code, one for an ordinary sequential diagnostic and one for a performance exerciser.

```

        BGNINIT                ;Sequential example
        READF                  #EF.CONTINUE      ;Continue command?
        BECOMPLETE            END                ;Yes, get no p-table
        READF                  #EF.NEW          ;New pass?
        BNCOMPLETE            NEXT              ;No, skip setup
SETUP:  MOV                    #-1,LOGUNIT      ;Initialize logical unit NBR
NEXT:   INC                    LOGUNIT         ;Point to next logical unit
        CMP                    LOGUNIT,L$UNIT   ;Have we pass maximum?

        BEQ                    SETUP           ;Yes, abort the pass
        GPHARD                 LOGUNIT,PLOC    ;Get the p-table
        BNCOMPLETE            NEXT            ;If not available, get next
        MOV                    @PLOC,LOCAL     ;Move p-table contents to
                                ;local storage
END:    ENDINIT                ;Finished
    
```

INIT code is run at priority 0 or the value of the priority argument that was used in the HEADER macro. A SETPRI macro can be used to change the priority.

```

        BGNINIT                                ;Performance exerciser
                                                ;example
        READEF                                #EF.CONTINUE      ;Continue command?
        BCOMPLETE                             END                ;Yes, finished
SETUP:  MOV                                  #LOCAL,R4          ;Init local storage pointer
CLEAR:  MOV                                  #0,(R4)+            ;Clear next piece of local
        CMP                                  R4,#ENDLOC        ;End of local storage?
        BNE                                  CLEAR              ;No, go clear more
        MOV                                  #LOCAL,R4          ;Reinit local storage
                                                ;pointer
        MOV                                  #-1,LOGUNIT        ;Init logical unit number
NEXT:   INC                                  LOGUNIT            ;Point to next logical unit
        CMP                                  LOGUNIT,L$UNIT      ;Have we passed maximum?
        BEQ                                  END                ;Yes, finished
        GPWARD                               LOGUNIT,PLOC        ;Get the p-table
        BNCOMPLETE                           DROPPED           ;Not available, get next
        MOV                                  @PLOC,(R4)+        ;Move p-table contents to
                                                ;local storage
        BR                                    NEXT
END:    ENDINIT                                ;Finished

```

### 7.5.22 Clean-Up Code (BGNCLN, ENDCLN, DOCLN)

Cleanup coding is assembled with the diagnostic program, utilizing initiating (BGNCLN) and ending (ENDCLN) directives. The coding can be used by either the diagnostic program or DRS and must return the test device to a static (power-up) state. The clean-up code is invoked in three different ways:

- 1) at the end of every sub-pass
- 2) at the issuance of DOCLN macro
- 3) by operator CONTROL/C

Do Clean-Up (DOCLN) is a separate call that can be independently coded within a diagnostic structure to return a device to that static state via a call up of the BGNCLN structure. If invoked during the INIT code, the entire diagnostic will be aborted and control will return to the DRS command mode. If invoked during a hardware test, it aborts the current sub-pass with the message ABOPAS.

### 7.5.23 Is Manual Intervention Allowed? (MANUAL)

Each test requiring manual intervention must contain a Test, Allow Manual Intervention (MANUAL) call, to determine, via the return of a complete or not-complete indicator, if manual intervention may or may not be successfully performed. If a negative determination is made, the program must provide an exit from the test structure via the coding of an Exit Test (EXIT TST) macro. This is necessary since any attempt to execute manual intervention coding when the system is incapable of successfully performing the operation will cause the program to "hang" since no operator will be available to perform the required action that will enable the program to continue.

Example:

```
    BGNTST
    MANUAL
    BCOMPLETE 5$      ;Branch if manual intervention is allowed (UAM flag
    EXIT TST         ;set).
5$:.....
```

Manual intervention is not normally allowed in XXDP+ chain (batch) mode. The user may be a Clear Manual Intervention (CMI) or Set Manual Intervention (SMI) directive in a batch file to enable and disable manual intervention.

### 7.5.24 Get Manual Parameters (GMANID, GMANIA, GMANIL)

This should be done with care since diagnostics may not run properly. If used, the batch file must contain responses to manual intervention questions.

Ref: XXDP+ System User's Manual (CHQUSE)

Coding of a GMANI call provides a means for an operator to manually intervene in the operation of a program under predefined response conditions. When a GMANI call is executed, a message is delivered to the operator by DRS while the program awaits a manual response. The values (upper or lower case) entered by the operator are delivered to the program along with a function-complete indicator. The GMANI calls are similar in format to the GPRM calls except that the OFFSET argument is replaced by a DATADR argument. The XFER calls cannot be used with the GMANI calls.

### 7.5.24.1 GMANID Call

The Manual Intervention Input Data Call is as follows:

GMANID	MSGADR, DATADR, RADIX, MASK, MINCHR/LOLIM, MAXCHR/HILIM, DEFAULT-SELECT
MSGADR	;MSGADR is address of message to be printed.
DATADR	;DATADR is address of data storage area.
RADIX	;RADIX is number base of expected input ;values symbolized by: D(decimal), O(octal), ;B(binary), or A (ASCII).
MASK	;MASK is bit position to which the data is ;justified prior to packing.
MINCHR/LOLIM	;Define lower limit: ;MINCHR serves A radix. ;LOLIM serves D, O and B.
MAXCHR/HILIM	;Define upper limit: ;MAXCHR serves A radix. ;HILIM serves D, O, and B.
DEFAULT-SELECT	;to default values or strings.

- a) The Message Address Argument (MSGADR) is the symbolic address of the ASCII message which requests information from the operator.
- b) The Data Address Argument (DATADR) is the symbolic address of the storage area into which the data, received from the operator, will be placed.
- c) The Number Base Argument (RADIX) is a value which defines the radix the operator must use for input data. Legal radix values are D(DECIMAL), O(OCTAL), A(ASCII) and B(BINARY). Control characters may be input to perform desired control functions.
- d) The Mask Argument (MASK) provides DRS with the location of the consecutive bit positions, within a 16-bit word, into which the value of a parameter will be right-justified in the P-table. This argument does not apply to ASCII radix.
- e) The Minimum Character (MINCHR) and Maximum Character (MAXCHR) arguments apply to A radix usage only, with a maximum defined input of 72 characters. The Low Limit (LOLIM) and High Limit (HILIM) arguments apply to D, O, and B radix usage and provide the DRS with unsigned numerical values that define the number of input data characters desired.
- f) The Default-Select (DEFAULT-SELECT) argument provides the DRS with a logical switch which defines either an actual operator input or a default value for the P-table. The argument is coded either YES or NO to indicate, respectively, if default values should or should not be appended.

#### 7.5.24.2 GMANIA Call

The Manual Intervention Input Address call is as follows:

GMANIA MSGADR, DATADR, RADIX, MINCHR/LOLIM, MAXCHR/HILIM, DEFAULT-SELECT

MSGADR	;same as GMANID
DATADR	;same as GMANID except this is not a table address
RADIX	;same as GMANID
LOLIM	;same as GMANID
HILIM	;same as GMANID
DEFAULT-SELECT	;same as GMANID

The Get Manual Intervention Address (GMANIA) call is used to obtain even-valued, unsigned parameter addresses from the operator. If odd valued addresses are desired, the programmer must use the data call GMANID. The mask and ASCII character limits (MINCHR/MAXCHR), however, are not required since all addresses will be numeric values.

#### 7.5.24.3 GMANIL Call

The Get Manual Intervention Logical Call is as follows:

MSGADR	;same as GMANID
DATADR	;same as GMANID except this is not a table address
MASK	;same as GMANID
DEFAULT-SELECT	;same as GMANID

The Get Manual Intervention Logical (GMANIL) call is used to obtain a logical yes (YES) or no (NO) response from the operator. Either lower or upper case values will be accepted by DRS. The arguments are used in the same manner as those associated with the GMANID data call. The radix and character limits (MINCHR/MAXCHR and LOLIM/HILIM), however, are not required.

#### 7.5.25 Operator Interrupt Enable (BREAK)

The diagnostic program will sometimes run with priority levels that are so high that operator-initiated interrupts from the console device will be ignored. The possibility of this problem occurring is alleviated by the design of DRS, which allows an operator request flag to be set when communication occurs, and tested when a DRS call is executed by the program. For this reason, diagnostic program coding should contain periodic DRS calls that are predicated on approximately 2 seconds of run time. To serve this end, a special Program Break Call (BREAK), which will advise DRS to initiate a test of the operator request flag, is available to the programmer. This call is used only in the exceptional case where a given program loop contains no DRS calls (even ERROR calls) at all (usually in long software delays).

### 7.5.26 Bus Reset (BRESET)

The BRESET call is used to cause the execution of a RESET instruction to, for example, a unit under test (UUT) via DRS. This method of initializing error conditions allows DRS to implement self-protective procedures prior to executing a clear operation.

### 7.5.27 Memory Allocation (MEMORY)

When a program issues the MEMORY macro, DRS gets the address of the start of free memory.

#### MEMORY ARG

The address is placed into the location specified by ARG. The first word of free memory contains the length in words of free memory. The second word contains bits 0-15 of the physical address of free memory. The third word contains bits 16-31.

DRS, at start-up time, always puts the length of total memory (28K words and over) into the header word L\$HIMEM. The value is in page address register form.

### 7.5.28 Interrupt Handling (SETVEC, CLRVEC, BGNSRV, ENDSRV)

Since diagnostics usually have interrupt service routine coding, a device vector location should be configured with the address of the associated service routine, under appropriate interrupt priority level conditions. To accomplish this for each device, a Set Vector (SETVEC) macro is used, along with three associated arguments (ARG1, ARG2, ARG3), to provide the following:

- . ARG1, the first argument, provides the address of the vector.
- . ARG2, the second argument, provides the address of the associated interrupt service routine.
- . ARG3, the third argument, provides the priority level for the servicing of the associated interrupt.

The Set Vector macro is as follows:

```
SETVEC ARG1,ARG2,ARG3 ;Set-up a device interrupt.  
                      ;ARG1 provides the address of the  
                      ;vector.  
                      ;ARG2 provides the address of  
                      ;the service routine.  
                      ;ARG3 provides the appropriate priority  
                      ;level.
```

A vector location assigned to a device may be deallocated by the coding of a Clear Vector (CLRVEC) macro, in which an argument is used to provide the absolute vector address.

The Clear Vector macro is as follows:

```
CLRVEC ARG ;Return an interrupt vector to the unused pool.  
           ;The trap catching mechanism is replaced in the vector.  
           ;ARG provides the address of the vector.
```

The coding of the interrupt service routine requires an initiating (BGNSRV) directive and an ending (ENDSRV) directive.

The service routine macros are as follows:

```
BGNSRV LABEL      ;LABEL identifies the routine  
ENDSRV ARG        ;ARG is for optional interrupt priority change
```

RO is used by DRS and should not be used in the diagnostic, including the interrupt routine.

### 7.5.29 Documentation Aids

The formatting and readability of program listings may be enhanced by using the COMMENT, END COMMENT, SLASH, and STARS macros to provide left (COMMENT, ENDCOMMENT) and right (SLASH, STARS) justified 70-column line graphics for use as message and/or comment brackets.

#### 7.5.29.1 Left Justified Graphics (COMMENT, ENDCOMMENT)

The Left Justified Graphic macros allow printing of a 70-column line (0-69) of repetitive symbols (/\*\:). The associated argument may be used to define the number of lines desired. If the argument is not used, a single line will be printed. The left graphic macros are as follows:

```
COMMENT ARG      ;Print initial line of graphic (/*\:).  
                 ;ARG provides for specific number  
                 ;of lines.  
ENDCOMMENT ARG   ;Print final line of graphic (/*\:).  
                 ;ARG provides for specific number  
                 ;of lines.
```

### 7.5.29.2 Right Justified Graphics (SLASH, STARS)

Right Justified Graphic macros allow the printing of a 70-column line (32-96) of either forward slashes (/) or asterisks (\*). The associated argument may be used to define the number of lines desired. If the argument is not used, a single line will be printed. The right graphic macros are as follows:

```
SLASH ARG ;Print line of forward slashes (/).  
          ;ARG provides for specific number  
          ;of lines.
```

```
STARS ARG ;Print line of asterisks (*).  
          ;ARG provides for specific number  
          ;of lines.
```

### 7.5.30 Program Priority (SETPRI, GETPRI)

SETPRI is used to set the priority at which the diagnostic will run. The call format is SETPRI ARG, where ARG is the desired priority. GETPRI is used to determine what the current diagnostic program priority is. The call format is GETPRI ARG, where ARG is the location into which DRS will return the current program priority.

The priority argument is aligned with the priority field of the PSW. It is recommended that you use PRI07-PRI00 defined by the EQUATES macro. When SETPRI is used in the INIT code, the priority is permanent. When SETPRI is used in a test, the priority is only in effect for that test.

### 7.5.31 Bus Type Check (READBUS)

The READBUS call (no argument) is used to determine bus type (UNIBUS versus Q-BUS). A complete indication is given for Q-BUS, not complete for UNIBUS.

```
Example:  READBUS  
          BCOMPLETE  QBUS  ;Branch if Q-BUS.  
          ...        ;Bus is UNIBUS.
```



### 7.5.32 Load Device Protection

Load Device Protection is provided by DRS, using a table in the diagnostic to identify the P-table entries which are pertinent to the load device (i.e., Unibus addr., Massbus addr., unit/drive). The required three word table supplies the hardware off-sets of the CSR, the MASS BUS unit, and the drive number of the UTT. It is used by DRS to compare these hardware parameters to the corresponding values for the load device. On a match, a NOT-COMPLETE indication is returned after a GPHARD call for the load device.

If the programmer does not care to have a match attempted on a particular field (such as MASS BUS UNIT NBR), the programmer can code a "-1" in the appropriate slot in the protection table.

Example:

```
BGNPROT
.WORD 2      ;P-table offset of CSR.
.WORD -1     ;not a MASS BUS device.
.WORD 4      ;P-table offset of drive #.
ENDPROT
```

### 7.5.33 File Control Services

DRS provides the ability for a diagnostic to request the loading of a byte or word of a data file (from the load medium only) into an address specified by the diagnostic. The file must be opened by name (OPEN MACRO), read one byte or one word at a time (GETBYTE, GETWORD), and closed. It is the responsibility of the programmer, working with Release Engineering, to ensure that a file of the specified name resides on the load medium. Data will not be linked or relocated. A diagnostic which calls for data retrieval via DRS will not run under APT. Note that this says nothing about the APT compatibility of a diagnostic that does its own data retrieval from a medium that it knows about; such a case is not related to the data retrieval capability of DRS.

#### WARNING

Anyone writing a DRS diagnostic that calls for data retrieval via the DRS will have produced a non-APT-compatible diagnostic.

There are four macros involved in data retrieval:

OPEN ARG - Where ARG is the ASCII string containing the file name and extension. If the file is not found, the message "LOOKUP FAILURE FILE.EXT" will be printed and return to the monitor prompt mode will occur.

GETBYTE ADDR - The next sequential BYTE from the currently open data file is placed into the specified location. End-of-file is indicated by NOT COMPLETE.

GETWORD ADDR performs two GETBYTES.

CLOSE - The currently open data file is closed.

### 7.5.34 Access to Flags

A diagnostic may request to see the operator flag settings by issuing the RFLAGS macro. This is a Read-Only access. The settings will be passed in a 16-bit word to the diagnostic specified by the macro's argument (RFLAGS argument). The bit settings are as follows:

- 15 Halt on Error
- 14 Loop on Error
- 13 Inhibit Error Reporting
- 12 Inhibit Basic Error Reports
- 11 Inhibit Extended Error Reports
- 10 Direct All MSGS to Line Printer
- 9 Print Number of Test Being Executed
- 8 Bell on Error
- 7 Run in Unattended Mode
- 6 Inhibit Statistical Reports
- 5 Inhibit Dropping of Units by Diagnostic
- 4 Autodrop Units
- 3 Loop on Test
- 2 Reserved
- 1 Unimplemented
- 0 Unimplemented

### 7.5.35 Autodrop Section

Every diagnostic must have an AUTODROP SECTION delimited by BGNAUTO....ENDAUTO. This required section contains AUTODROP code with which the diagnostic checks each unit to see if it responds READY and drops it if it does not. If so desired, this section may contain nothing more than the delimiters. That is, the section may have no actual function. If the unit is dropped in a sequential diagnostic, it will be returned to the INIT code to fetch another P-table. In an "Exerciser-type" diagnostic, where more than one unit at a time is tested, it is always sent on to the first hardware test. The Autodrop Section is conditionally executed immediately after the INIT code, when the operator ADR flag is set.

## 7.6 SAMPLE DIAGNOSTIC

```
.TITLE Sample diagnostic
.MCALL SVC
SVC
.ENABLE ABS
.ENABLE AMA
.=2000

BGNMOD MOD1
;Program Header
POINTER BGNSW,BGNSFT,BGNDU,BGNRPT
HEADER CSAMP,A,0,10,0
DISPATCH 3

;Descriptive Text
DESCRIPT <SAMPLE DIAGNOSTIC>
DEVTYPE <RK05,RK06>

;Default Hardware P-Table
; Listed by DISPLAY, May change hardware P-tables on START
BGNHW DFPTBL
.WORD 0
ENDHW

;Software P-Table
; May change on START, RESTART, CONTINUE
BGNSW SFTBL
.WORD 0
ENDSW

;Global Equates and Global Data
EQUALS

LOGUNT: .WORD 0
LOCAL: .WORD 0
PLOC: .WORD 0

;Global Text
ERRMSG: .ASCIZ /ERROR MSG/
TS1: .ASCIZ /%N%AI AM TEST 1/
TS2: .ASCIZ /%N%AI AM TEST 2/
TS3: .ASCIZ /%N%AI AM TEST 3/
.EVEN
```

```
;Global Error Report Section
; PRINTB (Basic error info), PRINTX (Extended error info)
    BGNMSG ERRTN
    PRINTB #ERR1,CSR,DRIVE
    EXIT MSG
ERR1:  .ASCIZ  /%N%AERROR AT CSR %06%A DRIVE %01/
        .EVEN
CSR:    .WORD   177600
DRIVE:  .WORD   0
        ENDMSG

;Protection Table
    BGNPROT
        .WORD   -1
        .WORD   -1
        .WORD   0
    ENDPROT

;Global Subroutine Section (Such as Interrupt Service Routines)
; Interrupt Service Routine Addresses loaded in vector by SETVEC
    BGNSRV DEVINS
    MOV    @CSR,LOCAL
    ENDSRV

;Report Coding Section
; Invoked by DR> PRINT or DORPT macro
    BGNRPT
    PRINTS #STSMMSG,DRIVE
    EXIT RPT
STSMMSG: .ASCIZ  /%N%ADRIVE %01 %A WAS TESTED/
        .EVEN
        ENDRPT

;Initialization Code
; Executed by START, RESTART, CONTINUE, new subpass (check event
; flags)
    BGNINIT
    READEF #EF.CONTINUE
    BCOMPLETE END
    READEF #EF.NEW
    BNCOMPLETE NEXT

SETUP:  MOV    #-1,LOGUNT
NEXT:   INC    LOGUNT
        GPWARD LOGUNT,PLOC
        BNCOMPLETE NEXT
        MOV    @PLOC,LOCAL
END:    ENDINIT
```

;Autodrop Code

```
    BGNAUTO
    BIT      #2,@CSR
    BNE      ENDAT
    DODU     LOGUNT
ENDAT:     ENDAUTO
```

;Cleanup Code

```
; Executed on subpass, DOCLN, or ^C
    BGNCLN
    PRINTF  #CLEAN
    EXIT    CLN
CLEAN:     .ASCIZ  /%N%AI AM CLEANUP/
    .EVEN
    ENDCLN
```

;Drop Unit Code

```
; Executed by DROP or DODU
    BGNDU
    PRINTF  #DROP,LOGUNIT
    EXIT    DU
DROP:      .ASCIZ  /%N%AUNIT DROPPED WAS %06/
    .EVEN
    ENDDU
```

;Tests

```
    BGNTST  1
    PRINTF  #TS1
    ENDTST
    BGNTST  2
    PRINTF  #TS2
    BGNSUB
    BGNSEG
    BGNSEG
    "
    ENDSEG
    ENDSUB
    ENDTST

    BGNTST  3
    PRINTF  #TS3
    MOV     LOCAL,DRIVE
    ERRHRD  1,ERRMSG,ERRTN
    ENDTST
```

```
;Hardware P-Table Coding
      BGNHRD
      GPRMD   DR,0,0,-1,0,7,NO
      EXIT    HRD
DR:    .ASCIZ  /DRIVE/
      .EVEN
      ENDHRD
```

```
;Software P-Table Coding
      BGNSFT
      GPRML   SWQ1,0,-1,NO
      EXIT    SFT
SWQ1:  .ASCIZ  /SW QUESTION #1/
      .EVEN
```

```
;Patch Area
PATCH: .BLKW  50.
```

```
;The tail-end
      LASTAD
      ENDMOD
      .END
```

## CHAPTER 8

## NON-DRS AUTOMATED ENVIRONMENTS

## 8.1 INTRODUCTION

This chapter discusses the diagnostic programs that are used in automated, often centrally controlled, applications. The automated diagnostic operation consists of the execution of predefined sequences of diagnostic programs. Subjects covered include APT, ACT, SLIDE, Macro summary, and a discussion on SYSMAC. For a complete discussion of SYSMAC, refer to the PDP-11 MAINDEC SYSTEM PACKAGE document (SYSMAC.MAN, March 1975, MAINDEC-11-DZQAC-C-D).

## 8.2 AUTOMATED PRODUCT TEST (APT-11)

One of the environments in which diagnostics run is in manufacturing areas having an APT system. APT is a PDP-11-based computer system which loads and monitors one or more diagnostics into a PDP-11 computer, known as the unit under test (UUT).

## 8.2.1 Introduction

The diagnostic is loaded into the UUT over a serial asynchronous line by a specialized controller designed by test engineering. After loading a diagnostic, APT can start it running and monitor its progress. A diagnostic executing under APT continuously runs pass after pass until halted, externally by the operator or by APT when running a script file. When an error is detected by the diagnostic, the error is indicated in the APT mailbox and the appropriate path of execution is taken. It is preferable to loop, but in some cases, such as basic instruction diagnostics, the diagnostic can halt.

The APT system must communicate with the UUT diagnostic to ensure that the diagnostic is executing properly. If errors are detected, APT needs to be made aware of these errors. In addition, the diagnostic requires information about the environment in which it is to run. APT communicates with the UUT diagnostic via a polling mechanism. Periodically, APT reads the contents of a small number of UUT memory locations. This 8-word block is called the Mailbox. Based on the values read, APT performs one or more services. Services include verifying the proper execution of a diagnostic or ensuring that no error conditions exist. When it completes these services, APT modifies the Mailbox indicating to the diagnostic that the requested services have been completed.

### 8.2.2 APT Mailbox

The Mailbox, an 8-word block of memory within the diagnostic, provides the basic communications between APT and the diagnostic in the UUT. The address of this block is defined by the programmer and is passed to the APT system via the APT parameter block. The Mailbox locations contain status and control words, as well as a pointer to a diagnostic buffer. In general, the APT system monitors the first 5 words of the Mailbox for a request for service by the diagnostic and for the correct execution of the diagnostic (i.e., the diagnostic is not hung up without setting up an error condition).

### 8.2.3 APT Mailbox Fields

Table 8-1 illustrates the layout of the Mailbox fields.

Table 8-1. APT Mailbox Fields

Word	APT SPEC. Name	SYSMAC Name
1	Message Type Code	\$MSGTY
2	Fatal Error Number	\$FATAL
3	Test Number	\$TESTN
4	Pass Count	\$PASS
5	Device Count	\$DEVCT
6	Unit	\$UNIT
7	Message Address	\$MSGAD
8	Message Length	\$MSGLG

#### 8.2.3.1 Message Type Code, Word 1 (SYSMAC/\$MSGTY)

This field communicates diagnostic requests to the APT system. This field is set to a non-zero value by the diagnostic (.\$APTYPE MACRO) when service is required from the APT system. This field is reset to zero in all cases but fatal errors, after APT has serviced the request.



Contents - The following codes have been defined:

- a. Default and non-request value - Octal 000000
- b. Code 1, Octal 000001: Fatal error message - The product being tested is "not operable".
- c. Code 2, Octal 000002: Fetch statistics table - The diagnostic requests the APT system to copy a table of soft errors (e.g., tape or disk read errors, etc.).
- d. Code 3, Octal 000003: Spooling message - A message to be spooled is in the buffer and additional data will be spooled as part of the message.
- e. Code 4, Octal 000004: Spooling message - The last message or a complete message to be spooled is in the buffer.
- f. Code 5, Octal 000005: Script request message - The diagnostic currently running in the UUT is requesting the loading of another diagnostic.

SYSMAC Reference - The `.$APTYPE` macro will set up the above message type codes. Inline calls to the `.$APTYPE` macro are generated by the `REPORT` macro or diagnostic user code.

Programming Note - This must be the last field changed in the mailbox.

### 8.2.3.2 Fatal Error Number, Word 2 (SYSMAC/\$FATAL)

This field supplied the APT system with an error number that indicates why the product being tested is "not operable". When the diagnostic has discovered that the UUT is "not operable" then word 2 of the mailbox is loaded with an error number and `MSGTYP` is set to a one. The term "not operable" includes the case when the diagnostic determines that execution cannot continue and the case when a "soft error" occurs more frequently than a predetermined rate.

Contents - A binary number that can be related to the diagnostic and the reason for the error.

SYSMAC Reference - The `.$ERROR` macro will use the contents of the SYSMAC field `$ITEMB` when it reports fatal errors to the `.$APTYPE` macro.

Programming Note - This field must be set up before the fatal error code is set up in the Message Type field.

### 8.2.3.3 Test Number, Word 3 (SYSMAC/\$TESTN)

This location is assembled with a value of zero. During diagnostic execution, the number of the test currently being executed is stored in this location by the diagnostic. A diagnostic must be structured by test number, and all tests within a diagnostic must be sequentially numbered, starting with one. The number of the test currently being executed must be placed in this field before the test is started.

**Contents** - A binary number that can be related to the tests within the diagnostic.

**SYSMAC Reference** -

- a. The SYSMAC/NEWTST macro, upon option, will set up this field.
- b. The SYSMAC/.\$SCOPE macro routine will move field \$TSTNM to this field each time the scope routine is trapped to.

**Programming Note** - This field must be set up before test execution begins. The default value is octal 000000, which is not a valid test number.

### 8.2.3.4 Pass Count, Word 4 (SYSMAC/\$PASS)

This field is assembled with a value of zero and should be incremented each time the diagnostic completes the execution of a pass. A pass consists of the execution of all tests in a diagnostic. If more than one device is being tested, a pass consists of the execution of all tests on all devices controlled by the currently running diagnostic.

**Contents** - A binary number, that is set by default to octal 000000, and incremented by one (1) for each pass.

**SYSMAC Reference** - The \$PASS field has been moved from the SYSMAC Common Tags Area to the APT Mailbox when running under APT. Each time the SYSMAC/.\$EOP macro is entered, the pass count will be incremented by one (1).

**Programming Note** - Since the \$PASS field has been moved, a Clear instruction has been added to the code in the SYSMAC/SETUP macro to ensure that the field is set to zero when running in stand-alone mode.

#### 8.2.3.5 Device Count, Word 5 (SYSMAC/\$DEVCT)

This field, set to zero at assembly time, refers to the number of devices which have been tested and not to a device number. After the conclusion of the last test on a device, and only if another device is to be tested, this field should be incremented by one (1). If only one device is being tested, this field should remain zero.

**Contents** - A binary number that is set by default to octal 000000 and incremented by one (1) for each device tested.

**SYSMAC Reference** - None.

**Programming Note** - It is the responsibility of the diagnostic program to increment this field if more than one device is being tested.

#### 8.2.3.6 Unit, Word 6 (SYSMAC/\$UNIT)

This field, used by diagnostics that test multiple devices on a controller, must contain the device number as given in the Etable when a fatal or soft error is reported. This will allow the APT system to report the error to the correct device.

**Contents** - A binary value from 0 to 15 as related to one of the device descriptor words listed in the Etable.

**SYSMAC Reference** - None.

**Programming Note** - It is the responsibility of the diagnostic programmer to ensure that this field is correctly set up when testing multiple devices.

#### 8.2.3.7 Message Address, Word 7 (SYSMAC/\$MSGAD)

This field must contain the address of the buffer that contains the message, statistics, or script that is to be transferred to the APT system.

**Contents** - The address given must point to a word. Bits 17 and 18 will be obtained from the first word in the APT Parameter block. Thus, both the mailbox and message buffer must be addressable by the same bits 17 and 18.

**SYSMAC Reference** - The SYSMAC/\$APTTYPE macro will set up this field.

**Programming Note** - None.

#### 8.2.3.8 Message Length, Word 8 (SYSMAC/\$MSGLG)

This field must contain the length, in words, of the message in the buffer to be transferred to the APT system.

Contents - A binary number.

SYSMAC Reference - The SYSMAC/\$APTYPE macro will set up this field.

Programming Note - None.

### 8.3 AUTOMATIC COMPUTER TEST (ACT-11) SYSTEM

An ACT-11 system consists of a central computer and from one to thirty-two test stations. The central computer, or "mother", is made up of a PDP-11 CPU, one to eight RK disk drives, a high-speed paper tape reader, and a console terminal. Each test station, or "daughter", consists of a station console, the Unit Under Test (UUT), and the UUT's console terminal. The PDP-11 communicates via its UNIBUS with the UUT test stations. The UUT test stations communicate, in turn, via UNIBUS with the UUTs. ACT-11 may be used to load stand-alone programs. ACT-11, however, usually loads and monitors programs sequentially from a list known as a "sequence table". This sequencing capability requires that end-of-pass hooks be provided in the diagnostic programs themselves.

The ACT-11 system can operate in the following modes:

- o ACT Dump mode
- o ACT Auto-accept mode
- o ACT Station Test mode

#### 8.3.1 ACT Dump Mode

In this mode, ACT loads and runs a diagnostic into a Unit Under Test (UUT) as if it were run manually.

#### 8.3.2 ACT Auto-accept Mode

In this mode, ACT automatically loads, runs, and monitors a single diagnostic or series of diagnostics through one or more iterations. This mode includes the "quick verify" mode.

### 8.3.3 ACT Station Test Mode

In this mode, ACT directly performs a variety of UUT memory tests.

## 8.4 SERIAL LOADER IN DEMAND EVERYWHERE (SLIDE)

### 8.4.1 The SLIDE System

The SLIDE System can be thought of as an RKDP System with multiple users. (RKDP is an existing DEC software package that includes all PDP11 diagnostics on three RK05 disk packs.) SLIDE is composed of a central computer (mother) and up to 32 test stations (daughters). The central computer can have up to 8 RK05 disks attached to it. Normally, disks 0, 1, and 2 are used to store RKDP diagnostics and disks 3 through 7 are used to store preconfigured Run-Time Exercisers and other user programs. The central computer has a console terminal and in most cases each test station will have a user terminal.

The central computer is connected to each test station by a 4 wire communications line and the appropriate communications interface.

A monitor program called SLIDE.BIN is resident in the central computer and a Serial Line Interface Program (Y.BIN, sometimes referred to as the SLIP [Satellite Loader Interface Program] monitor) is resident in each test station CPU. These programs provide the software interface which permit communications between the central computer and each test station.

The main function served by SLIDE is diagnostic program loading. All diagnostics used by the test stations reside on the central computer's disk(s) and are serially output to selected test stations when requested. SLIDE is set up so that if more than one test station initiates a program load request at the same time, each request will, in effect, be handled simultaneously.

SLIDE can be run in either the dump mode or the chain mode. In the dump mode, one program is loaded into a test station. In the chain mode, a series of programs are executed with a single command. SLIDE can handle up to 64 user defined chains. Each chain consists of from 1 to 31 program and pass count selections to be executed in the order specified by the user. Pass counts of from 1 to 77777(octal) can be specified. It is also possible to branch from one chain to another chain, enabling SLIDE to run up to 1,984 programs with a single command.

SLIDE is best suited to be used in areas where options are to be tested on known good processors.

Some benefits obtained with SLIDE versus having a separate program loading device on each test station are:

1. By maintaining only one copy of each diagnostic, the job of ensuring that all users have the latest version is much simpler,
2. Reduced costs, since most paper tape readers can be eliminated,
3. Increased speed - SLIDE transfers programs at 10,000 cps while paper tape readers operate at 300 cps, and
4. Less operators - Using the chain mode, several programs can be run with a single command. (There is even a hard copy verification as each diagnostic is run).

#### Note

(1) When using multiple disks, the user must ensure that all drives are numbered sequentially. For example, if 4 disk drives were attached to the central computer then these drives would have to be drives 0, 1, 2, and 3.

(2) SLIDE has a watchdog timeout feature that can be used to monitor a test station running in the chain mode (refer to Section 8.4.7 for a description of the watchdog timeout feature).

The SLIDE system is composed of a central computer and up to 32 test stations.

#### 8.4.2 SLIDE Basic Software

The basic SLIDE software comes as part of the RKDP Diagnostic Package. The disks as received from the program library contain all SLIDE software, the PDP-11 diagnostics, the RKDP Monitor, and an update program. The SLIDE software is stored on the first RKDP disk pack. The SLIDE software consists of the following programs:

1. SLIDE.BIN      The SLIDE monitor program that will reside in the central computer.
2. SHELP.TXT     The SLIDE Help Text File that will reside on the system disk. It can be called in from the central computer's terminal or from any test station terminal.

3. A.OBJ           The modified PDP-11 absolute loader. A copy of A.OBJ will reside in each test station's memory.
4. Y.BIN           The PDP-11 Serial Line Interface Program. A copy of Y.BIN will reside in each test station's memory (Y.BIN is also referred to as the SLIP or test station monitor).

#### 8.4.2.1 Central Computer Memory Usage

SLIDE.BIN is stored in memory locations 0 through 70000 (octal), i.e., the lowest 16K words.

#### 8.4.2.2 Test Station Memory Usage

All SLIDE software at the test station is stored in the upper 1100 decimal words of memory. The size of the test station CPU will determine where A.OBJ and Y.BIN are stored. The end result is that A.OBJ is loaded in front of the bootstrap loader and Y.BIN is loaded in front of A.OBJ. (In effect, where the bootstrap loader is placed determines where A.OBJ and Y.BIN will be stored.)

#### Note

A.OBJ and Y.BIN are protected from being overwritten when running a diagnostic program at a test station by setting up two RKDP hooks. These are, the load medium indicator (location 41) and the automatic mode indicator (location 42). This is done at system initialization by the Y.BIN program. Y.BIN stores 377 in location 41 and XX7314 in location 42 (where XX is the memory size parameter). Together these two hooks flag the PDP-11 diagnostic program not to write into the upper 1500 words of the test station memory.

### 8.4.3 Using SLIDE

When using SLIDE, the user should be aware of the following special characters:

- CTRL/C to return user to command mode
- CTRL/S to suspend temporarily output to a terminal
- CTRL/Q to restart output to a terminal
- RUBOUT to delete the last character typed
- CTRL/L to clear the user terminal's input buffer

### 8.4.4 Obtaining a Directory

To obtain a directory, type one of the following:

/D#<CR> To obtain a full directory on the user terminal (where # is the selected disk drive number).

SAMPLE FULL DIRECTORY PRINTOUT OF DISK DRIVE 0.

```
/DO <CR>
FILENAME.EXT      LENGTH
RKDP  .BIN 017  TADP  .BIN 017  TCDP  .BIN 017  TMDP.  BIN 017
THDP  .BIN 017  RXDP  .BIN 017  XTECO .BIN 027  UPD1  .BIN 017
ACT   .BIN 032  HELP  .TXT 003  XQADF0.BIN 029  ZQUXBO.BIN 040
Y     .BIN 002  HELP8 .TXT 004  SLIDE .TXT 015  SLIP11.OBJ 002
HELP11.TXT 003  00    .CHN 001  01    .CHN 001  02    .CHN 001
03    .CHN 001  04    .CHN 001  06    .    001  05    .CHN 001
4002 BLOCKS USED.
```

/D#F<CR> To type a fast directory (no extension or blocks) on the user terminal.

SAMPLE FAST DIRECTORY PRINTOUT OF DISK DRIVE 1.

```
/D1F<CR>
RKDP      TADP      TCDP      TMDP      THDP      RXDP      XTECO      UPD1
ACT       HELP      XQADF0    ZQUXBO
Y         HELP8     SLIDE     SLIP11    HELP11
4002 blocks used
```



#### 8.4.5 Time and Date Messages

/T<CR> To print out the time and date on the test station terminal or console terminal.

To set the time and date, type one of the following:

##### Note

The time and date can only be changed from the console terminal.

/T###<CR> to set the time (HHMM)

/T/#####<CR> to set the date (MMDDYY)

/T###/#####<CR> to set the time and date

Sample time and date printouts. All user inputs are underlined.

/T TIME 02:17 01/01/79

/T1710 TIME 17:10 01/01/79

/T/020979 TIME 17:11 02/09/79

/T1800/021079 TIME 18:00 02/10/79

#### 8.4.6 Chain Mode Operation

Chain mode operation consists of the sequential execution of programs without operator intervention. Only programs that have been designed to run under XXDP+ chain mode can be chained using SLIDE. Chainable programs are identified in the directory by the extension .BIC.

##### Note

.BIC is a chainable binary file.

.CHN is a SLIDE chain command file.

To run chain mode, the SLIDE monitor requires a chain command file (.CHN) indicating the programs to run, and the number of times (pass count) each program must execute before going on to the next program in the table.

A chain file can have up to 31 program entries. Up to 32767 passes can be run for each entry, numbered in octal from 000001 to 077777. Up to 64 chain files can be generated, numbered in octal from 00 to 77.

#### 8.4.6.1 Making a Chain

Chains can be made up of programs stored on any disk. However, the .CHN file should be stored on drive 0. Making a chain file cannot be done on-line while SLIDE is running.

To make a SLIDE chain, the XTECO program must be used. The UPDATE 2 program resides on the RKDP disk and is used to add, delete, rename, or edit ASCII files on RKDP packages.

To make a chain, the following procedure should be used:

1. Start the RKDP monitor by booting the RK disk.
2. Run the XTECO program.

Once started, XTECO will respond with:  
\*

3. Write enable the disk
4. Type in:

a. TEXT DK\*:##.CHN<CR>

Where \* = disk drive number and ## = chain number

b. FFFF--/#####]<CR>

Where FFFF-- = The first four characters of the diagnostic name followed by 2 spaces or the six character diagnostic name and

##### = The number of passes to run it

- c. Repeat step B for additional programs
- d. /<CR> to terminate the list of programs in a chain

or

/## to call for execution of another chain at the end of the current chain.

- e. Type in a CTRL/Z to terminate the text input.

#### Note

The user can return to the RKDP monitor from XTECO by typing the following:

\*EXIT<CR>

Sample printout for making a chain follows:

```
CHMDKB1 XXDP+ DK MONITOR
BOOTED VIA UNIT 0
28K NON-UNIBUS SYSTEM
```

```
ENTER DATE (DD-MMM-YY): 8-JUN-82 <CR>
RESTART ADDR: 152010
THIS IS XXDP+. TYPE "H" OR "H/L" FOR HELP.
```

```
.R XTECO <CR>
CHUTEB2 XXDP+ XTECO UTILITY
RESTART: 005126
```

```
*TEXT DKO: 71.CHN <CR>
WRITE-ENABLE OUTPUT UNIT, THEN TYPE <CR>. <CR>
"ZKWA /000001]
"ZKWA /000003]
"ZKWA /000005]
"ZKWAB0/000002]
"ZKWA /000002]
"/
"^Z
```

#### Notes

1. Building a chain with ZKWA (clock test program) is an excellent way to verify that the chain mode works on a user system.
2. UPD2 cannot make a chain file.

#### 8.4.6.2 Considerations when Making a Chain

1. Before making a chain for SLIDE, the user should first, when possible, try making and running the chain under RKDP.

#### Note

Just because a program has a .BIC extension doesn't guarantee that a program is chainable.

2. When making a chain under SLIDE each program/passcount entry must contain the same number of characters. (i.e., each program entry must contain a four character entry followed by 2 spaces; or a 6 character entry and a slash (/). This is followed by a six digit pass count. This, in turn, is followed by a bracket (]) and a carriage return.

3. To verify that a chain has been typed correctly, use the TYPE command of UPD2. A chain made via instructions in section 7.4.6.1 can be verified by typing the following command string.

```
TYPE DK1:06----.CHN<CR>
```

A sample printout for verifying a chain follows:

```
*TYPE DK1:06----.CHN
```

```
ZKWA /000001]  
ZKWA /000003]  
ZKWA /000005]  
ZKWAB0/000002]  
ZKWA /000002!  
/  
^Z
```

#### Note

The ^Z may or may not be printed because some versions of UPD2 replace the ^Z with a null character.

To printout all chains on DK0, the following could be typed:

```
*TYPE DK0:*.CHN<CR>
```

4. When the user is typing a command or data under UPD2, he should be aware of the following special commands:

CTRL/C - returns user to command mode.

CTRL/Z - exits text mode and returns user to command mode.

RUBOUT - deletes the last character typed.

FILENAMES - considered to always be 6 characters long, plus a 3 character extension. The name and extension are left-justified with trailing blanks.

### 8.4.7 Watchdog Timeout Feature

The watchdog timeout feature is used to monitor the status of any test station running programs in chain mode. Normally, if a diagnostic fails while running in chain mode, all communications between the SLIDE monitor and the test station are terminated. When this condition occurs, the user has no way of knowing that chain mode operation has been terminated for a particular test station. Using the watchdog timer, this condition can be detected by the SLIDE monitor and an appropriate error message can be printed on the system console.

#### 8.4.7.1 Watchdog Timer Commands

The watchdog timer is activated by typing in the following command string at a test station terminal:

```
/W### <CR>
```

Where ### is a 1-3 digit decimal number that represents the watchdog timeout time. Timeout times of 1-255 minutes are valid.

The watchdog timer is deactivated by typing in:

```
/WC <CR>
```

If the user enters an invalid watchdog command, an ?INVALID ENTRY? message will be printed on the test station terminal.

#### 8.4.7.2 Using the Watchdog Timer

1. The user activates the test station for XXX minute timeouts.
2. The user starts a chain running at the test station.
3. If chain mode is terminated, the watchdog timer will timeout and print the following message on the console terminal:

```
LINE XX WATCHDOG TIMEOUT(XXX) AT HH:MM DD/MM/YY
```

Where XXX indicates a watchdog timeout time of XXX minutes.

Even if the chain runs successfully, the watchdog timer will remain activated. At this time the user has four options:

- a. Do nothing - the watchdog timer will timeout in XXX minutes.
- b. Start another chain - this chain will run using the same timeout time that was used for the previous chain.

- c. Enter a new /W command and start another chain - This chain will run using the timeout time that was specified in the new /W command.
- d. Disable the watchdog timer by entering a /WC command. This gives the user a way of eliminating unnecessary timeout messages from being printed on the console.

Two example watchdog timer printouts are given below:

Example 1.

Test station terminal printout:

```
/W03 WATCHDOG TIMER ACTIVATED  
/C32
```

```
[ZKWA      /000001]      BO LOADED      TIME 13:30    09/08/79
```

```
[ZKWA      /000002]      BO LOADED      TIME 13:31    09/08/79
```

```
CHAIN END
```

```
EXIT
```

Console terminal printout:

```
LINE 02 WATCHDOG(003) TIMEOUT AT 13:36 09/08/79
```

Example 2.

Test station terminal printout:

```
/W05 WATCHDOG TIMER ACTIVATED  
/C32
```

```
[ZKWA      /000001]      BO LOADED      TIME 13:37    09/08/79
```

```
[ZKWA      /000002]      BO LOADED      TIME 13:38    09/08/79
```

```
CHAIN END
```

```
EXIT
```

```
./WC WATCHDOG TIMER DEACTIVATED
```

#### 8.4.8 Issuing Commands to Another Terminal or Line Printer

With SLIDE the user has the capability of issuing commands to a particular test station or to the line printer from any other test station or from the console terminal. A few examples of this feature are given below.

1. To print out a full directory for drive 0 at test station number 07, the following command could be typed at the console terminal.

```
/#07=/D0<CR>
```

2. To load the clock program (ZKWAB0) into test station number 07, the following command could be typed at the console terminal.

```
/#07=ZKWAB0<CR>
```

3. To execute chain number 02 at test station number 07, the following command could be typed at any other test station terminal.

```
/#07=/C02<CR>
```

4. To print a fast directory for drive 1 on the line printer, the following command could be typed at any terminal.

```
/#LP=/D1F<CR>
```

5. To activate line 15 for 5 minute timeouts, the following command could be typed at any terminal.

```
/#07=/W5<CR>
```

#### 8.4.9 Updating and Patching

Diagnostic programs and supporting files may be updated (replaced) and/or patched using the UPD2 program. UPD2 is an offline facility, i.e., off-line programs cannot be run while SLIDE is executing.

UPD2 is a program that provides fairly powerful updating and patching capabilities. This program may be run by booting any RKDP disk and then by answering the RKDP's prompt character by typing `.R UPD2<CR>`. For complete details on UPD2, please reference the `XXDP+ USER'S GUIDE`. In summary, UPD2 provides the following updating and patching facilities:

- . Transferring an ABS format program file from any of the supported peripherals to any of the supported peripherals and reblocking the file in the process (requires 24K of memory for a 16K program).
- . Transferring (using the PIP command) any file from one drive to another (requires 12K of memory).
- . Patching a program by using the MOD command (requires 24K of core for a 16K program).
- . Patching a program by using the PATCH command (has no special memory requirements because the patching is done directly on the disk).
- . Batch operation (memory requirements depend upon operations performed).

#### 8.4.10 SLIDE Help Commands

/D#<CR> to print a full directory  
/D#F<CR> to print a fast directory (no ext. or blocks)  
(# is the drive number)

/T<CR> to print the time and date  
/T###<CR> to set the time (HHMM)  
/T/#####<CR> to set the date (MMDDYY)  
/T###/#####<CR> to set the time and date

AAAA--<CR> where AAAA-- is a 4 or 6 character program name.  
<LF> For a 4 character name, the first program that has  
the same first 4 characters will be sent.

For a <CR> the program must be started manually.  
For a <LF> the program will self start.

/W###<CR> will activate the watchdog timer for ###  
minute timeouts (### is a decimal number).

/C##<CR> will run a predefined chain of programs  
<LF> (## is the chain number).

-  
For a <CR> the chain will execute normally.  
For a <LF> a quick verify pass will be executed.  
For a - (minus) the chain will loop indefinitely.

/#XX=Y XX is (1) an octal line no. of any test station  
(2) LP for the line printer  
(3) CY for the console terminal

Y is any valid command



\* \* \* \* \* E X A M P L E S \* \* \* \* \*

ZKWABO<CR> sends program ZKWABO to the test station that initiated the program load request.

/#00=ZKWABO<LF> sends program ZKWABO to line #00 (program will self start).

/#25=/W005<CR> monitors line 25 for 5 minute timeouts.

/#03=/C01<CR> executes chain 01 on line #03.

/#LP=/DOF<CR> prints a fast directory of drive 0 on the LP.

## 8.5 MACRO SUMMARY

### 8.5.1 Mandatory, Direct Support Macros

The following macros are in direct support of XXDP+, ACT-11, APT-11 and are required when not using DRS.

#### 8.5.1.1 . \$ACT11 Macro

Purpose: Provide the ACT-11 System with the following information:

- a. Sets location 42 to zero.
- b. Sets up the address of the "end-of-pass" code into location 46.
- c. Sets up program needs information into location 52.
  - Power failure during run required.
  - Program memory size dependent.
  - Manual intervention required.

How-To-Use Reference: SYSMAC.MAN 10.1

#### 8.5.1.2 . \$APTHDR Macro

Purpose: Provides, at load time, the APT System with the following information:

- a. Setup location 24 with the start address of the program (default 200).
- b. Setup location 44 with the starting address of the APT parameter block.
- c. Generates the APT parameter block based upon known information and following programmer supplied data:
  - TSTM - Run time, in seconds, of the longest test in the diagnostic.
    - \*\*\* Define worst case time for maximum iteration count, memory size, and CPU speed.
  - PASTM - Run time, in seconds, of the 1st pass on one unit (quick verify).
  - UNITM - Additional run time, in seconds, of a pass for each additional unit being tested.

How-To-Use Reference: SYSMAC.MAN 10.2

#### 8.5.1.3 . \$APTBL S Macro

Purpose: Builds and sets up the APT mailbox and environment (ETABLE). The size of the mailbox is fixed. However, the ETABLE length may be defined by the programmer. (Minimum of four words).

All fields defined in the mailbox and ETABLE will be set to predefined values. The programmer may use assignment statements before calling this Macro to change these values.

How-To-Use Reference: SYSMAC.MAN 10.3

#### 8.5.1.4 . \$APTTYPE Macro

Purpose: Performs the communications with the APT system via the APT mailbox.

How-To-Use Reference: SYSMAC.MAN 10.4

#### 8.5.1.5 REPORT Macro

Purpose: Generates in-line code that calls the . \$APTTYPE Macro, permitting complete APT communication functionality.

Note

This macro is not required in converted diagnostics if the `.$ERROR` and `.$TYPE` macros are used.

How-To-Use Reference: SYSMAC.MAN 10.5

#### 8.5.1.6 `.$APTAT` Macro

Purpose: Generates space for the statistics table in a format compatible with the APT error reporting procedure.

Note

This macro is not required if statistics are not being collected.

How-To-Use Reference: SYSMAC.MAN 10.6

### 8.5.2 Indirect Support Macros

The following macros contain ACT-11, APT-11, or XXDP+ code. This code is required if diagnostics are to interface correctly.

#### 8.5.2.1 `.$EOP` Macro

Purpose: A code sequence required by ACT-11 and XXDP+ for end of pass control.

How-To-Use Reference: SYSMAC.MAN 9.3

#### 8.5.2.2 `.$CMTAG` Macro

Purpose:

- a. Sets up address words for the indirect switch and display registers (SWR and DISPLAY).
- b. Calls the `.$APTBL5` macro to generate the APT mailbox and ETABLE.
- c. Removes names from the common tag area that are now defined in the mailbox and ETABLE. (i.e., `$PASS`, `$SWREG`, etc.)

How-To-Use Reference: SYSMAC.MAN 9.2

### 8.5.2.3 .EQUAT Macro

Purpose: To define commonly used constants, registers, and symbols.

How-To-Use Reference: SYSMAC.MAN 7.4

### 8.5.2.4 SETUP Macro

Purpose: Sets up, at execution time, location SWR to ensure that the correct switch register is being used while running under APT systems control.

How-To-Use Reference: SYSMAC.MAN 8.1

### 8.5.2.5 NEWTST Macro

Purpose: Moves the test number generated by this macro to the APT mailbox test number field (\$TESTN).

How-To-Use Reference: SYSMAC.MAN 8.6

### 8.5.2.6 .\$SCOPE Macro

Purpose: Moves the test number used by this macro (\$TSTNM) to the APT mailbox test number field (\$TESTN) each time scope is trapped to.

How-To-Use Reference: SYSMAC.MAN 9.4

### 8.5.2.7 .\$ERROR Macro

Purpose: After this macro prints out the reported error, a fatal error will be reported to the APT system if running in the APT mode. The contents of field \$ITEMB will be reported as the fatal error number. Upon reporting the fatal error, the diagnostic will go into a single instruction loop.

How-To-Use Macro Reference: SYSMAC.MAN 9.5

#### 8.5.2.8 .\$.TYPE Macro

Purpose: If running in the APT system mode, this macro will spool all messages to the APT system via the mailbox. It will also suppress all messages to the UUT console if requested to do so by the APT system.

How-To-Use Macro Reference: SYSMAC.MAN 9.7

#### 8.5.2.9 TYPNAM Macro

Purpose: To type the name of the diagnostic, if the diagnostic is not running in the ACT-11 Auto-accept Mode.

How-To-Use Macro Reference: SYSMAC.MAN 8.20

### 8.5.3 Other Support Macros

The following macros are required to support the \$.EOP and \$CMTAG macros.

- \$.TRAP
- \$.TYPDEC
- PUSH
- POP
- STARS

## 8.6 SYSMAC.SML, THE DIAGNOSTIC MACRO LIBRARY

The System Macro Library (SYSMAC.SML) is a collection of supporting macros that are available to the PDP-11 diagnostic engineer. By using these macros, the engineer does not have to write the various "non-diagnostic" routines necessary to use the PDP-11 and the associated console terminal. This allows the engineer to place emphasis on the diagnostic techniques.

Based upon the type of code they generate, the macro routines in SYSMAC.SML are classified into 3 groups:

1. Definition macros
2. In-Line Code macros
3. Handler macros

### 8.6.1 Definition Macros

The macros in this group do not generate executable code but are used to define commonly used constants, symbols, registers, device addresses, and vectors. They are also used to aid in documenting a program's listing and are intended to be used at the very beginning of a program.

- .EQUAT
- .HEADER
- .KT11
- .SETUP
- .SWRHI
- .SWRLO

### 8.6.2 In-line Code Macros

These macros are used in the mainline code of a program to generate listing controls and subroutine and trap calls to the SYSMAC service routines.

- .CKSWR
- .COMMENT
- .ENDCOMMENT
- .ESCAPE
- .GETSWR
- .MUL
- .NEWTST
- .POP
- .PUSH
- .SETTRAP
- .SETUP
- .SKIP
- .STARS
- .TRMTRP
- .TYPBIN
- .TYPDEC
- .TYPNAM
- .TYPNUM
- .TYPOCS
- .TYPOCT
- .TYPTXT

### 8.6.3 Handler Macros

The Handler group is composed of those macros that actually service the subroutine and trap calls generated with the in-line macros. The handler macros are generally placed near the end of the program. The handlers within SYSMAC.SML are compatible with DEC-10-DECDOC.

- LINKAGE
- .SCATCH
- .\$4OCAT
- .\$CMTAG
- .\$DB2D
- .\$DB2O
- .\$DIV
- .\$EOP
- .\$ERROR
- .\$ERRTYP
- .\$MULT
- .\$POWER
- .\$R2AZ
- .\$RAND
- .\$READ
- .\$RDDEC
- .\$RDOCT
- .\$SAVE
- .\$SB2D
- .\$SB2O
- .\$SCOPE
- .\$SIZE
- .\$SUPRS
- .\$TRAP
- .\$TYPBIN
- .\$TYPDEC
- .\$TYPE
- .\$TYPOCT





## CHAPTER 9

## STRUCTURED PROGRAMMING

## 9.1 INTRODUCTION

A major change has occurred in the maintenance and repair philosophy of the entire computer industry over the past five years. Module and even subsystem replacement has replaced component level repair by electronic technicians. Although this change is due to a number of reasons, the primary reason is that components are no longer single function devices. Components now replace whole cabinets of logic, and component level repair in the field, done by highly skilled support engineers, only occurs under extreme emergency situations.

This change in repair philosophy affects diagnostic development in several ways. The technical training of field personnel has been greatly reduced since field diagnostics are no longer used for component-level troubleshooting. This requires tailoring the diagnostic interface to the non-technical user. The concept of customer maintenance also puts the running of DEC diagnostic programs in the user application realm, where human-engineering is critical. The human interface has to be in a natural, meaningful language which is helpful and non-intimidating. The obvious exception is in the area of module repair facilities, where dedicated test equipment is used to isolate defective components.

Although many diagnostics are still being written in assembly language, there is a growing use of higher level languages in diagnostic development. This has resulted from memory constraints being relaxed by changing the minimum system from 4K words of memory to 16K words. Another factor is that the development time for programs written in higher level languages is much less than that for pure assembly programming. Languages such as BLISS, PASCAL, BASIC and SPMACJ have been used for in-house diagnostic programs with much success.

## 9.2 PROGRAMMING CONSIDERATIONS

Diagnostic programmers must be aware of the three major requirements their diagnostic programs must satisfy. They are as follows:

- . the program must deliver the specified diagnostic coverage
- . the program has to be completed on schedule and within budget, and
- . the program must be maintainable and extensible for the entire life of the product.

By far, the largest payback to any development group is when the product they create is designed so that it is easy to test, maintain, and change. The concepts of structured design and structured programming have proven themselves as the best means to achieve the goals to testability, maintainability, and changeability in software.

### 9.2.1 Structured Design

The basic concept of structured design is to take a high level conceptual idea for a program and repeatedly break it into smaller and smaller pieces until each piece is concerned with performing one and only one function. Each piece is then coded as a module, or routine, in such a way that there is only one entry point into the module and only one exit from the module. In addition, all information passing into or out of the module must pass through a defined interface. Global data access is considered a poor programming practice. When these modules are diagrammed (see Section 9.2.2), a hierarchy becomes apparent with the top most module being the primary or main control module, which calls or causes the execution of the subordinate modules.

### 9.2.2 Structured Programming

Each module can be thought of as a block with an arrow going in and an arrow leaving the block (a, Figure 9-1). These blocks can be linked together in a number of ways, one of which is called a sequence. The sequence (b) involves two or more blocks where the output from one block feeds directly into the input of the next block. The entire sequence then has one entry and one exit point. The entire structure, therefore, can be represented as a block (c).

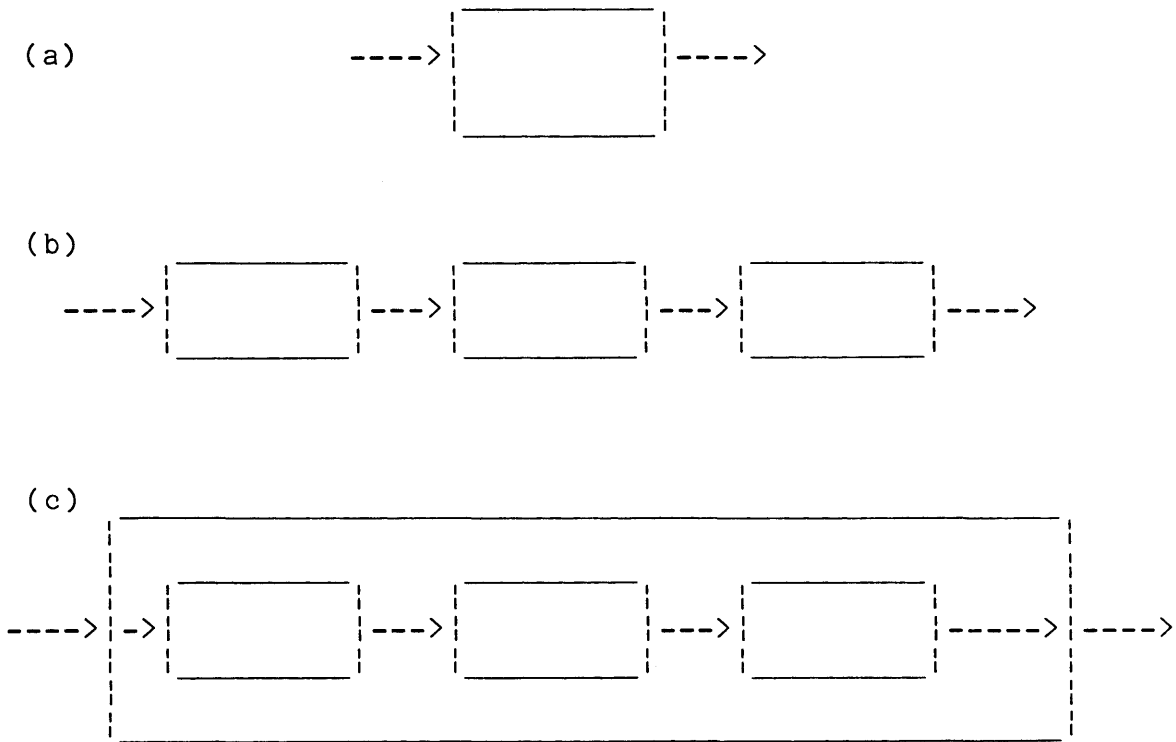


Figure 9-1. Block Structured Programming

This "block within a block" arrangement has led to the use of the term Block Structured Programming, or Structured Programming for short. The primary benefit of this type of structuring is that each block is responsible for one function. Any problem with that function can be isolated to that module or to the data being passed through the interface. In addition, each module can be modified without affecting any other module as long as the data interface remains the same. The modular structure should be documented with an overview diagram showing the interrelationship between the modules and with INPUT-PROCESS-OUTPUT diagrams for each module.

Any structure can be part of (nested within) any other structure as long as the single entry and exit is maintained. Figure 9-2 illustrates several common structures. The modular structure, which results from breaking the top level design into smaller pieces having a defined purpose, results in a hierarchy, with the topmost module being the primary or main module (Figure 9-3).



### 9.3 USING BLISS

BLISS is a middle-level language which has advantages of higher and lower level languages. It was designed to meet the following objectives.

#### Requirements of System Programming

- . Economy of memory space and execution time
- . Access to all relevant hardware features
- . Object code not dependent on elaborate run-time support

#### Characteristics of System Programming Practices

- . Control of data structure representation
- . Modularization of system into separately compilable sub-modules

#### Overall Good Language Design

- . Encourage program structuring for understanding
- . Economy of concepts, generality, and flexibility

BLISS relieves users of concern over a broad class of details while permitting access to hardware features and producing high quality object code.

### 9.4 USING PASCAL FOR SPECIFICATION AND DESIGN

PASCAL was originally designed as a language for teaching structured programming techniques in an educational environment. PASCAL is now becoming popular as a general-purpose language since PASCAL programs are structured and relatively easy to read and understand. PASCAL's English-like statements result in program code that is descriptive of what the program actually does.

PASCAL can be used in the development of diagnostics to describe software algorithms or to explain the functionality of a software procedure. PASCAL descriptions or models can clearly define the required behavior of sections of program code or routines. Ambiguity is undesirable in any specification or design document and PASCAL can also supplement carefully worded English prose and program flow diagrams.

The following example of a PASCAL procedural specification, derived from the Digital/Intel/Xerox Ethernet Specification, specifies an address recognition function or procedure which could be implemented in hardware or software:

```
                (* Recognize Address Function *)

CONST
  Address_Size = 48;      (* 48 bit address *)

TYPE
  Address_Mode = (Promiscuous, Normal);

VAR
  Physical_Address: Address_Value;
  Multicast_Address: Address_Value;
  Broadcast_Address: Address_Value;

FUNCTION      Recognize_Address (address: Address_Value):BOOLEAN;

BEGIN
  IF Address_Mode = Promiscuous OR
     address = Physical_Address OR
     address[1] = 1 AND address=Multicast_address OR
     address = Broadcast_Address
  THEN Recognize_Address = TRUE
  ELSE Recognize_Address = FALSE;

END;

                (* Recognize Address Function *)
```

This example clearly defines the recognition of an address. The address field of a receive message is checked. If the address field is the physical or multicast address of the station, the station is in promiscuous mode (receive everything), or the address field says it is a broadcast message that everyone should recognize, then this software would indicate that the message should be passed on to a higher level for examination as a received message.

PASCAL includes a variety of statements, data types, and predefined procedures and functions. Some are:

```
Data Types = INTEGER, REAL, CHAR, BOOLEAN, ARRAY, RECORD, SET, FILE
Statements = FOR, REPEAT, WHILE, UNTIL, CASE, IF-THEN, IF-THEN-ELSE,
              BEGIN...END, READ, WRITE
```

Refer to a PASCAL manual or text of your choice for more information.

## 9.5 USING BASIC TO WRITE DIAGNOSTICS

The BASIC language was developed at Dartmouth College so that students with no background in programming could learn to use a computer. Because of its simplicity, BASIC can be used to specify, design or even implement programs easily.

In the case of a Modem Manufacturing test for the LA12, two test programs were written in MINC BASIC to run on a MINC test station. The test station had the necessary hardware to test the analog signals in the modem. BASIC allowed manipulation and comparison of those signals.

## 9.6 PROGRAM DESIGN LANGUAGE 1

### 9.6.1 Introduction

Program Design Language 1 (PDL1) is a tool used to describe the design of a program. PDL1, which is a pseudo code, highlights control logic and functionality contained within a module (PDL1 is a block-structured language). It uses English expressions and reserved keywords. The advantages of designing with PDL1 are:

- . Implementation language independence
- . Provides an overview of the code
- . Use of structured design constructs
- . Machine maintainable documentation for code
- . Easy to learn
- . Replaces flow charts
- . Increased productivity during design and coding

PDL1 will not replace code or user documentation nor is it machine translatable to code. PDL1 is not, in itself, sufficient to completely design modules.

### 9.6.2 Purpose and History

Most computer programs written in the past, and even many that are being written today, are not very readable. In many cases, even the author of the program may find it difficult to obtain a particular piece of information about the program from the documentation produced some time in the past. The difficulty is greatly magnified when another programmer, besides the author, attempts to maintain a poorly documented program. This problem has been widely recognized (see Chapter 11, General Coding Conventions) and the usual solution is to produce more documentation.

Structured programming and structured design have been in existence for some time and programmers recognize the advantages they offer. If program documentation were to be similarly structured in a hierarchical manner, the information retrieval task would be simplified. Documentation delivered with a program should include overview diagrams, data flow diagrams, hierarchy charts, etc. The external, functional aspects of the program should be separated as much as possible from the internal, procedural details. Procedural descriptions may include detailed hierarchy charts, interface specification blocks, input-process-output charts, structured flowcharts, and a Program Design Language description of the program.

Pseudo code has been referred to as a Program Design Language or tight English in software engineering literature. Many pseudo code variations are patterned after block structured, higher level languages such as PASCAL or PL1. Many algorithms, in much of the recent software engineering literature, are presented using some type of pseudo code. Various design techniques use pseudo code as part of their process. For example, IBM's HIPO (Hierarchy plus Input-Processing-Output) approach uses pseudo code to describe the processing logic. Data flow diagramming techniques also use pseudo code to describe the content of processing blocks.

Pseudo code differs from language preprocessors, which translate from a structured language to a compilable source. The language preprocessor and the structured language must be maintained throughout the life of the software. FLECS and RATFOR are examples of pre-processors. Pseudo code is not compilable and cannot be imbedded in a language source except as comments.



### 9.6.3 Guidelines

PDL1 is to be used for logic and program design. There exists a natural transition from design of the system structured in PDL1 to the actual coding process, and PDL1's purpose is to allow communication from a diagnostic program designer to a diagnostic program developer. This section discusses guidelines for using PDL1 in both the design and coding of software. Since PDL1 is not compiled by a computer, its syntax rules do not have to be as stringent as they would be for a compilable language. Functionally, the description of a program in PDL1 is equivalent to a structured flowchart of the program. Just as there may be different levels of detail in flowcharts, there may be different PDL1 descriptions of the same program. A high level description, for example, might be included in a functional specification, while a lower level description could be included in a design specification. Whether flowcharts or PDL1 are used, descriptions of more than one level of detail should be provided for all but the simplest and shortest programs.

#### 9.6.3.1 Design Guidelines

PDL1 is an effective tool to describe the functionality and logic requirements of a module. The following guidelines are helpful:

- . Limit the size of blocks to one page.
- . Do not overdesign - continue until the code can be easily visualized. Do not duplicate the code, and do not use variable names or source code descriptions in the expressions.
- . Do top-down reviews to expand the detail of individual pseudo code lines.
- . Try to use similar structures and definitions to help identify common processing logic or functions.

The benefits of using the design guidelines are:

1. Easier to maintain designs.
2. The potential for identification of common subroutines and structures.

### 9.6.3.2 Guidelines for Translating PDL1 to Source Code

The following guidelines address the translation of PDL1 code to source code.

1. PDL1 code can be included in the designed module and maintained with the code.
2. PDL1 code can be left as a block in the front of a module. Generally, if the code is mixed with the source code, the PDL1 code replaces block comments only and does not replace line-by-line comments or additional non-PDL1 comments.
3. The PDL1 code and the block of source code it describes should be on the same page.

### 9.6.4 PDL1 Format

#### 9.6.4.1 General Format

The "rules" of PDL1 are intended to be only as stringent as necessary for two people, the writer and reader of the program, to communicate. A PDL1 code line starts with a keyword and may contain an expression. PDL1 is a block-structured language. The keywords are used to define the types of block structures and control structures used in a program. PDL1 does not allow any GOTO-like statements except for an early exit from an inner block or loop.

#### 9.6.4.2 Block Structure

PDL1 allows nesting of blocks to any depth, permitting the description of the most complex of programs. Any program may be written using the three classes of blocks:

1. sequential
2. selective
3. iterative

Each block starts with a keyword that defines the type of block and ends with the same keyword preceded by END (e.g., MODULE....ENDMODULE, DO....ENDDO, etc.)

#### 9.6.4.2.1 Sequential Blocks

Sequential blocks start with one of the following keywords:

- . PROGRAM
- . MODULE
- . MACRO
- . PROCEDURE
- . ROUTINE
- . SUBROUTINE
- . SUBTEST
- . SEGMENT

Alternate keywords that may be used to define a sequential block are the block name preceded by BGN, or BEGIN, and END (e.g. BGNMODULE....ENDMODULE). Since PDL1 will not be assembled or compiled, there are no restrictions in the number of characters used for these names. This allows the designer to create names truly descriptive of the functions performed by each sequential block. To show that a series of words is the names of a block, connect the words by using the underline character (\_). For example, ROUTINE RK01 LOGICAL FUNCTION\_TESTS. (For more detail, see Section 11.4.5, Block Structuring.)

#### 9.6.4.2.2 Selective Blocks

Selective blocks start with IF or SELECT and end with ENDIF or ENDSELECT, respectively. (For more detail, see Section 11.4.6, Other Constructs.)

#### 9.6.4.2.3 Iterative Blocks

Iterative blocks start with DO, REPEAT, or WHILE and end with ENDDO, ENDREPEAT, or ENDWHILE, respectively. (For more detail, see Section 9.6.4.6, Other Constructs.)

### 9.6.4.3 Internal Block Structure

#### 9.6.4.3.1 Imperatives

The keywords defining a block should be vertically aligned to start in the same print column and be connected by colons to clearly show the scope of the block. All statements to be included in the block are indented 4 spaces within the bracketing keywords. Within a block, each statement starts with an imperative. Some suggested imperatives are:

- . ATTEMPT
- . BEGIN
- . CALL (a subroutine)\*
- . CHANGE
- . CHECK (FOR)
- . DECREMENT (or DECR)
- . DELAY
- . DISABLE
- . DISPLAY
- . ENABLE
- . EXECUTE (a Macro)\*\*
- . FILL
- . HALT
- . INCREMENT (or INCR)
- . INITIALIZE (or INIT)
- . INPUT
- . INQUIRE
- . ISSUE
- . MOVE
- . OUTPUT
- . PRINT
- . READ
- . RECEIVE
- . REPORT
- . RESET
- . RESTART
- . RESTORE
- . SAVE
- . SEEK
- . SEND
- . SET
- . SET UP
- . START
- . STOP
- . STORE
- . TYPE
- . WRITE

\* The imperative CALL is only used to show where a subroutine is called.

\*\* The imperative EXECUTE is only used to show where a macro will be expanded.

#### 9.6.4.3.2 The Underline Character

In addition to connecting words in the name of a block (see Section 9.6.4.2.1), the underline character may be similarly used to concatenate words in order to form names of memory locations, arrays, buffers, functions, etc. It may also be used to separate groups of digits that form a number to improve readability (e.g., 177\_400, 4\_194\_304, or 01\_000\_001\_000 instead of 01000001000).

#### 9.6.4.3.3 Assignment and Relational Operators

The preferred assignment operator is = (equals sign).

The preferred relational operators are:

EQ for =

GT for >

LT for <

NE for ≠

GE for ≥

LE for ≤

#### 9.6.4.3.4 Parentheses and Brackets

Parentheses may be used to enclose:

1. Parameters - input to or output from a program, subroutine, etc.
2. Comments.
3. Arithmetic or logical subexpressions.

Brackets, braces, or both, if included in the character set, may be used in addition to or instead of parentheses. Consistency must be used in using parentheses, brackets, and braces throughout the program. When in doubt about this or any other "rule", document the exception or interpretation.

#### 9.6.4.3.5 Early Exit

Two methods of early exit are defined:

- . RETURN
- . EXITBLOCK (label)

The RETURN statement allows early exit from a subroutine, which has the effect of popping the last transfer address off the stack.

The EXITBLOCK (label) statement allows an exit from any level of nesting to any outer level. The outermost block to be skipped must be labeled with the "label" which is enclosed in parenthesis in the EXITBLOCK statement. When the EXITBLOCK statement is encountered, control passes to the next statement following the end of the labelled block.

#### 9.6.4.4 Keywords

##### 1. SEQUENTIAL BLOCK KEYWORDS

PROGRAM ENDPROGRAM  
MODULE ENDMODULE  
ROUTINE ENDRoutine  
MACRO ENDMACRO  
PROCEDURE ENDPROCEDURE  
SUBROUTINE ENDSUBROUTINE or SUBR ENDSUBR  
SUBTEST ENDSUBTEST  
SEGMENT ENDSEGMENT

##### 2. SELECTIVE BLOCK KEYWORDS

IF THEN ELSE ENDIF  
SELECT CASE DEFAULT ENDSELECT

##### 3. ITERATIVE BLOCK KEYWORDS

DO FOR TO DOWNTO BY ENDDO  
DO FOREVER ENDDO  
REPEAT UNTIL ENDREPEAT  
WHILE DO ENDWHILE

##### 4. OTHER KEYWORDS

EXITBLOCK  
CALL  
EXECUTE  
RETURN

#### 9.6.4.5 Block Structuring

```
PROGRAM program_name (param1, param2)
MODULE module_name
:  ROUTINE routine_or_test_name
:  :  SUBTEST subtest_name_or_number
:  :  :  SEGMENT segment_name
:  :  :  :  statement_1
:  :  :  :  statement_2
:  :  :  :  CALL subroutine_name (param1, param2, param3)
:  :  :  :  statement_4
:  :  :  ENDSEGMENT [segment_name]
:  :  :  statement_5
:  :  ENDSUBTEST [segment_name]
:  :  stmt_6
:  ENDROUTINE [routine_name]
:  stmt_7
ENDMODULE module_name
stmt8
ENDPROGRAM program_name
```

#### NOTES

1. Keywords are uppercase.
2. Brackets [] enclose optional names.
3. Some programs may not be nested this deeply.

#### 9.6.4.6 Other Constructs

```
IF expression
:   THEN
:     do this if true
:   ELSE
:     do this if false
ENDIF
```

```
SELECT case_variable
:   CASE a: do this
:   CASE b: or this
:   :
:   :
:   CASE x: or this
:   DEFAULT: or this
ENDSELECT
```

```
DO FOR variable = initial_value TO final_value BY increment
:   DOWN TO decrement
:   these
:   statements
ENDDO
```

```
REPEAT
:   these
:   statements
: UNTIL expression_is_true
ENDREPEAT
```

```
WHILE expression_is_true DO
:   these
:   statements
ENDWHILE
```

```
SUBR subroutine_name
:
:   statements
:
ENDSUBR subroutine_name
```

```
MACRO macro_name
:
:   statements
:
ENDMACRO macro_name
```



9.6.4.7 Example of a Program in PDL1

```
PROGRAM EXAMPLE
INITIALIZE STACK POINTER
OUTPUT START MESSAGE
ROUTINE TEST_01
:   READ STATUS INTO DEV_STAT (status of device should
:   :   initially equal zero)
:   IF DEV_STAT NEQ 0
:   :   THEN
:   :   :   OUTPUT ERROR MESSAGE
:   :   ELSE
:   :   :   REWIND DEVICE (to ensure that writing starts from
:   :   :   BOT)
:   ENDIF
ENDROUTINE TEST_01

LB:  ROUTINE TEST_02
:   SELECT BYTES PER INCH
:   :   CASE 6250: RECORDING_METHOD = GCR
:   :   CASE 1600: RECORDING_METHOD = PE
:   :   DEFAULT:  RECORDING_METHOD = NRZI
:   ENDSELECT
:   WRITE ONE 64 BYTE RECORD
:   SET UP 1 SECOND TIMEOUT (write command should be executed
:   :   in far less time than one second)
:   REPEAT
:   :   INPUT STATUS
:   :   UNTIL DEVICE NOT BUSY OR TIMEOUT OCCURS
:   ENDREPEAT
:   IF TIMEOUT OCCURRED
:   :   THEN OUTPUT ERROR MESSAGE
:   :   AND EXITBLOCK (LB)
:   REWIND DEVICE AND SET CTR = 0
:   WHILE REWINDING = 1 DO
:   :   INCREMENT CTR
:   :   IF CTR GTR 32767
:   :   :   THEN OUTPUT MSG. 'REWINDING'
:   :   ENDIF
:   ENDWHILE
ENDROUTINE TEST_02

ROUTINE TEST_03
:   READ ONE RECORD
:   DO FOR I = 126 DOWNT0 0 BY 2
:   :   IF BYTE [I] NEQ I
:   :   :   THEN OUTPUT ERROR (WAS, SHOULD_BE)
:   :   ENDIF
:   ENDDO
ENDROUTINE TEST_03
```

```
ROUTINE TEST_04
:
:
:
:
ENDROUTINE TEST_56
OUTPUT END MESSAGE
ENDPROGRAM EXAMPLE
```

#### 9.6.4.8 PDL1 Keywords in Alphabetical Order

##### PDL1 KEYWORDS IN ALPHABETICAL ORDER

BY	ENDSELECT	NE
CALL	ENDSUBR (=ENDSUBROUTINE)	
CASE	ENDSUBTEST	NOT
DEFAULT	ENDWHILE	PROCEDURE
DO	EQ	PROGRAM
DOWNTO	EXECUTE	REPEAT
ELSE	EXITBLOCK	RETURN
ENDDO	FOR	ROUTINE
ENDIF	FOREVER	SEGMENT
ENDMACRO	GE	SELECT
ENDMODULE	GT	SUBR (=SUBROUTINE)
ENDPROCEDURE	IF	SUBTEST
ENDPROGRAM	LE	THEN
ENDREPEAT	LT	TO
ENDROUTINE	MACRO	UNTIL
ENDSEGMENT	MODULE	WHILE

## CHAPTER 10

### DIAGNOSTIC PROGRAM DOCUMENTATION

Good documentation is important in all diagnostic programs. The programmer must always keep in mind that the documentation is for the program user and not the programmer. It is therefore vital that the User's Documentation be prepared so that a user with minimal knowledge of software or the language in which the diagnostic was written, be able to use the diagnostic.

Programs require several levels of documentation. Program documentation consists of user documentation, section and test descriptions, comment lines, cross references, etc. All important aspects of the program should be explained, from the overall purpose and structure of the program to the meaning of individual lines of code. Three important reasons for careful program documentation are:

1. Documentation is an important aid in the debugging phase of program development. Prefaces and comments tell what the code should do, so that unwanted side effects stand out.
2. Documentation helps the program user to understand the capabilities and requirements of the program. It also increases the value of the program as a fault isolation tool, if the user must troubleshoot with the program listing.
3. Documentation is essential to the individual who must maintain the program, particularly if the maintainer did not develop the program. Since the documentation tells what the code is intended to do, the maintainer can fix the code if the code does not perform as intended. If the maintainer wishes to alter the function of the code, the documentation will aid in determining what should be changed. The documentation may be thought of as consisting of two main parts. The first part (called external here) is aimed at describing the environment, structure, and functional features of the program to the users. The second part (called internal here) is the actual code and comments which comprise the assembly listing of the program. The discussion which follows is aimed predominantly at defining a standard format for the external part of the documentation.

You must document your program in each phase of its development. Do not leave it until the end when the program strategy and details may be difficult to recall.

## 10.1 DOCUMENTATION GUIDELINES

Although specific guidelines are impossible to give in all cases, it is important to follow some broad guidelines even in the internal parts of the documentation. Some of the factors to consider are the benefits of central listing (usually at the beginning of the listing) of all user MACROS. Central location allows the user to find explanations of MACROS without having to search the entire document. (Note: MACRO expansion is required each time the MACRO is used). Another useful practice is to locate variables, data buffers and constants at the beginning of the program. Finally, sequence the sub-tests in ascending order of normal execution in memory. It is confusing when the sub-tests are out of sequence or when error reports show PCs with random jumps in both directions. The way sub-tests are linked together requires some thought. Special test dispatchers or standard test linking routines (like those in SYSMAC) have unique advantages and disadvantages. As a general rule, use the standard sub-test linking packages. However, product specific sections of this document should provide some additional information on this entire area.

User documentation for all diagnostic programs should follow, as closely as possible, the general format discussed in the sections that follow.

## 10.2 DOCUMENTATION SECTION

The documentation section is the first item in the listing when you release a diagnostic program. This section should include all information necessary to running and using the program. This section identifies the name and function of the program, program operating instructions, run-time requirements, and a functional description of each test in the program.

Organize the documentation section under several headings. Refer to appropriate skeleton files for DRS and SYSMAC programs. The following headings are recommended:

Documentation cover sheet

History section

Table of contents

Program abstract

Hardware requirements

Software requirements

Prerequisites

Operating instructions

Program functional description

Test descriptions

### 10.2.1 Documentation Cover Sheet

The documentation cover sheet is the first page of the documentation file. Use the following format:

- a. PRODUCT CODE: Insert the AC-XXXXX-XX code assigned by PDP-11 Diagnostic Release Engineering.
- b. PRODUCT NAME: Up to 29 character description matching the title of the engineering change order (ECO). Although the description may be expanded on the cover sheet, the first 7 characters of the ECO description are unique and must be the first 7 characters of the product name.
- c. PRODUCT DATE: Not necessarily the release date, but whatever date the program is being created or revised.
- d. MAINTAINER: Maintaining group, such as Small Systems Diagnostic Engineering.
- e. DISCLAIMER: The disclaimer statement should appear as shown in Example 10-1.
- f. COPYRIGHT STATEMENT: DIGITAL engineers use the format shown in Example 10-1, giving the first and last copyright years. These copyright years should be the same as those on the ECO. They should be the first year that the program was released from SDC and the current year.

IDENTIFICATION

- a: PRODUCT CODE: AC-8781G-MC
- b: PRODUCT NAME: CZDZAGO DZ11 ASYNC MUX TEST
- c: PRODUCT DATE: 20 FEBRUARY 1981
- d: MAINTAINER: SMALL SYSTEMS DIAGNOSTIC ENGINEERING

e: THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED AND COPIED ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE AND WITH THE INCLUSION OF THE ABOVE COPYRIGHT NOTICE. THIS SOFTWARE, OR ANY OTHER COPIES THEREOF, MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY OTHER PERSON. NO TITLE TO AND OWNERSHIP OF THE SOFTWARE IS HEREBY TRANSFERRED

THE INFORMATION IN THE SOFTWARE IS SUBJECT TO CHANGE WITHOUT NOTICE AND SHOULD NOT BE CONSTRUED AS A COMMITMENT BY DIGITAL EQUIPMENT CORPORATION.

DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE OR RELIABILITY OF ITS SOFTWARE ON EQUIPMENT THAT IS NOT SUPPLIED BY DIGITAL.

THE FOLLOWING ARE TRADEMARKS OF DIGITAL EQUIPMENT CORPORATION:

DEC	DECUS	DECTAPE
MASSBUS	PDP	UNIBUS
VAX		

d	i	g	i	t	a	l
---	---	---	---	---	---	---

- f: COPYRIGHT (C) 1979, 1981 BY DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASSACHUSETTS ALL RIGHTS RESERVED.

Example 10-1. Documentation Cover Sheet

10.2.2 History Section

List the version numbers (including all DECOs and DEPOs), author, date, and reason for modification of each version, one per line in chronological order. The reasons for modification include problem reports closed (by number) or DEPOs closed.

The example below was taken from CZCLM, a communications link test program for the DMP-11 and DMV-11.

REVISION HISTORY:

<u>REV</u>	<u>DATE</u>	<u>AUTHOR</u>	<u>REASON</u>
A	14-Jan-81	Bruce Ribolini	Original issue, DCLT for the DMP-11, DMV-11
B	26-Oct-81	Ernie Cooper	Add - "SET E=T Command" Add - ID of device requesting downlineload. Added needed patches. General cleanup and enhancement of document.

Example 10.2 Sample History Section

10.2.3 Table of Contents

List of all the major subsection titles. For example:

- 1 GENERAL INFORMATION
  - 1.1 Program Abstract
  - 1.2 System Requirements
  - 1.3 Related Documents and Standards
  - 1.4 Diagnostic Hierarchy Prerequisites
  - 1.5 Terminal Setup
- 2 OPERATING INSTRUCTIONS
  - 2.1 Commands
  - 2.2 Switches
  - 2.3 Flags
  - 2.4 Hardware Questions
  - 2.5 Software Questions
  - 2.6 Extended P-Table Dialogue
  - 2.7 Quick Startup Procedure
- 3 ERROR INFORMATION
- 4 PERFORMANCE AND PROGRESS REPORTS
- 5 DEVICE INFORMATION TABLES
- 6 TEST SUMMARIES

Example 10.3 Table of Contents

### 10.3 PROGRAM ABSTRACT

This section is a short but clear description and summary of the diagnostic program. Include its scope (functional test, exerciser, repair level, etc.) and major features. Example 10-4 shows a typical program abstract.

This example was taken from CZCLM, a communication link test program for the DMP-11 and DMV-11.

#### PROGRAM ABSTRACT

This DCLT (Data Communication Link Test) program is meant to provide Field Service with a tool to maintain DMP-11 or DMV-11 to DDCMP multipoint communication links. This DCLT program will provide the coverage necessary to detect failures in the computer equipment, the communication link, or the modem. This diagnostic has been written for use with the Diagnostic Runtime Services software (DRS). These services provide the interface to the operator and to the software environment. This program can be used with XXDP+, ACT, APT, SLIDE and paper tape. For a complete description of the runtime services, refer to the XXDP+ System User's Manual (CHQUS?.SEQ where ? is rev. level of the manual). There is a brief description of the runtime services in section 2 of this document.

#### Example 10-4. Sample Program Abstract

### 10.4 SYSTEM REQUIREMENTS

This section should describe all hardware requirements. Remember, all PDP-11 DRS diagnostics require a minimum of 16K words of memory and a console terminal. See Example 10-5.

#### SYSTEM REQUIREMENTS

This program requires

- . a PDP-11 processor
- . at least 24K words of memory
- . a console terminal interfaced with the standard address and interrupt vector.

#### Example 10-5. System Requirements Documentation



## 10.5 RELATED DOCUMENTS AND STANDARDS

List any standards or other documentation related to the program (e.g., DEC STD 138 for ANSI sequences is related to terminal diagnostics). All DRS diagnostics must refer to the XXDP+ System Users' Manual.

## 10.6 DIAGNOSTIC PREREQUISITES

Specify what other diagnostics are assumed to have been run and found working before this diagnostic is run: for example, a controller diagnostic that should have been successfully completed prior to a drive test.

## 10.7 PROGRAM ASSUMPTIONS

If the program assumes that certain elements of the equipment under test (e.g., disc pack) are fault free or are free from certain types of faults (e.g., multiple faults), these assumptions should be described here. Further, any significant assumptions made regarding the required training of the user should be included.

## 10.8 OPERATING INSTRUCTIONS

Include in this category all instructions for loading and executing the diagnostic program. If the program can be loaded and run on-line as well as off-line (standalone), show both methods. Example 10-6 shows the operating instructions for the LSI-11 Extended Instruction Set Test program.

### a. Loading Procedure

Use standard procedure for loading diagnostic programs.

### b. Starting Procedure

#### b.1 Control Switch Settings

See Section c.1. All switches should be reset for worst case testing.

#### b.2 Starting Address

After loading the program, it should always be started at 200 octal. If it is desired to save the pass counter, then the program should be restarted at location RESTRT, otherwise the program can be restarted at octal 200.

b.3 Program and/or operator action

b.3.1 Stand Alone

- 1) Place LTC switch in Off position (if applicable).
- 2) Load program into memory (.L VKA???).
- 3) Set switches (see section c.1) all low for worst case.
- 4) Type .S 200.
- 5) The program will execute and END PASS will be typed after completion of first pass and every 4th pass. However, typeout will be suppressed if Bit 5 of location \$ENVM is high.
- 6) A minimum of two passes should always be run.

b.3.2 Under APT

Load the program and start after setting the desired switches (see section c.1). However, if the diagnostic is run under APT with Bit 5 of \$ENVM low, then it will be required that a SLU with TTY registers having addresses of 176560-66 and interrupt vectors of 70 for receiver and 74 for transmitter be present. It will also be required to change the pass time from 5 to 15 seconds and the test time from 3 to 10 seconds.

c. Operating Procedure

c.1 Switch Settings

A 16-bit location called \$SWREG (i.e., location 176 octal) has been used to give the following options by inserting a 1 in their respective bit positions.

Bit #	Octal Value	Function
15	1000000	Halt On Error
13	0200000	Inhibit Printout

8-bit byte \$ENVM has been used to define the operating mode. All typeouts can be suppressed by making Bit 5 of byte \$ENVM a one.

Note

Operator input is underlined.

Example 10-6. Operating Instructions.

### 10.8.1 Loading and Starting Procedures

This section should have a summary description of a standard loading and starting procedure here. Any special procedures should be documented here. Procedures for selecting a subset of the tests comprising the program should be described here.

### 10.8.2 Special Environments

Any special characteristics of the program when executed in other than the standalone mode (e.g., APT system, operating system for online mode) should be clearly stated here.

### 10.8.3 Program Options

This section will describe all operator parameters, their meanings and uses, default values, ways of inputting these parameters, and ways of changing default values.

Each family of processors has standard definitions of all control switches (switch register). The meaning and usage of all such switches is described here. When a deviation from the standard is necessary, it should be noted and the new usage clearly described.

The addresses, meanings, and uses of all memory locations which serve as program control registers, program switch registers, starting or restarting locations, etc. should be described here. Deviations from memory location standards will be clearly and completely described here.

If the program is interactively controlled via a terminal command language, this language will be described here.

### 10.8.4 Execution Times

A statement of the approximate program run times should be provided. Normally, this approximation is given for the case of a single error-free pass through the complete program. It is important that approximate run times be provided for meaningful configurations, operating modes, and other circumstances and that these configurations, modes, and circumstances be explicitly stated.

For example: First pass run time is 3 seconds, subsequent passes take 30 seconds, and longest test time is 5 seconds.

## 10.9 ERROR INFORMATION

### 10.9.1 Error Reporting Procedures

This section contains a description of what happens when the program detects an error. It describes the formats used in reporting the error.

The purpose of the error report is to provide concise and accurate information in order to facilitate the isolation and repair of the fault causing the error. A diagnostic program must provide an error report in every instance where an error is detected. The only permissible exception is when the operator or operating system (in the case of on-line diagnostics) has invoked the "no error report" mode. Generally speaking, error reports take the form of visual display or hard copy produced at the time the error is detected. Some systems, most notably on-line diagnostic or automated manufacturing test systems, may reproduce all or part of the error report on a system file (e.g., an error log) for failure data base creation, automatic analysis, or for deferred printing if no printer is available at the time of failure.

Sophisticated error reporting schemes may allow the operator (or operating system) to select the amount and nature of the information to be included in the error report. For certain diagnostics, it is not reasonable to assume that the system under test has the capability to print or log error reports; this is usually the case for basic central processor diagnostics. In these situations, the program may report an error merely by halting, with pertinent information stored in documented memory locations. When errors are detected within subroutines, special care must be exercised to ensure that meaningful information is reported. The printing of the PC within the subroutine has little or no meaning. Errors detected within subroutines must provide enough information to identify what caused the error and where in the main flow of the program the subroutine call was issued. This holds true for nested subroutines as well.

Error reports should address the following:

1. The location in the program documentation where the error information is explained. Usually this takes the form of test number and subtest number.
2. A concise description of the test operation which failed or produced the failure.
3. The correct (i.e., expected or specified) results of the test operation.
4. The actual (i.e., incorrect) results of the test operation.

5. Other significant data: register, flags, indicators, operands, addresses, important memory location contents, etc.
6. To the extent possible, identification of the failing hardware element or a list of probable failing elements listed in decreasing order of probability.
7. Where applicable, a pointer or index to an entry in a document where information, pertinent to the error but too voluminous to be included in the error report, can be found.

#### HEADER AND BASIC ERROR FORMATS

The following basic error formats have been adopted as minimum ways of displaying the specified error information.

##### A.1 Register Value Wrong

Expected: <expected-value>  
Received: <actual-value>

##### A.2 Register Dump of Unit Under Test

```
<Register-0>  : <Register-value>  ;<Description of set bits>
"
"
"
<Register-N>  : <Register-value>  ;<Description of set bits>
```

##### A.3 Data Compare Error in Buffer

Memory Buffer Address : <Starting Address of buffer>  
Record Size : <Length of Transfer>  
Words in Error : <Number of words that are incorrect>

Address	Expected	Received
<Memory Address>	<Expected Data>	<Actual Data>*
"	"	"
"	"	"
"	"	"

Note:

\* A maximum of 8 memory locations will be dumped.

### 10.9.2 Error Halts

An explanation of all halts designed into the program should be provided. The correct operator response to those halts should be included.

In general, error halts should only be used in cases where the error was fatal and the ability of the processor to print the error message is in doubt. The most common use of error halts is in unit diagnostics of basic processor functions. In such cases the failure detected may be in a function required to print the error.

In all cases where the error halts are used, the documentation should contain a list of all the halts in the program and a reference to the program documentation where additional data is available.

## 10.10 OPTIONAL PERFORMANCE AND PROGRESS REPORTS

These two sections describe what the programmer must do to provide optional performance and progress reports listing number of hard errors, number of soft errors, etc.

### 10.10.1 Performance Reports

There is a type of diagnostic program whose prime function is to provide hardware performance data. The performance data is required - in the form of visual display or hard copy performance reports - even if the hardware is operating normally and well within specifications. This type of diagnostic program is commonly required for peripheral equipment.

Since the format and contents of performance reports must vary widely according to the specifics of the hardware and the user environment, specific guidelines cannot be provided.

### 10.10.2 Progress Reports

Diagnostic programs must be designed so that it is not possible for a program to be halted, to be hung-up, or to skip selected tests without the operator being aware of the fact. This is especially important in the case of programs with long execution times. Progress messages in the form of console typeouts of the name of each test as execution of the test and subtest begins is a good way to ensure that "silent death" is noticed soon. Under DRS, the name of the subtest cannot be printed. For DRS-compatible diagnostics, use the PNT flag to get this option.

The size and frequency of progress messages must not be such as to increase program run time, reduce test effectiveness (e.g., by reducing the activity level of the equipment under test), or create an operator nuisance.

### 10.11 SUB-TEST SUMMARIES

A summary of each sub-test must be provided to allow for clearer understanding of the program by the users without having to read the entire source code. This does not put an additional burden on the Diagnostic Engineer if the guidelines indicated in Section 9.12 are followed. Each sub-test should carry a summary or heading. By identifying these headings with ;\* symbols, the Diagnostic Engineer can extract each of these headings to be used in this section.

### 10.12 PROGRAM LISTING

The program code and its listing should be sectioned into tests, subtests, subroutines, constants, variables, data, parameters, buffers, communication areas, etc. Each section within the listing should be preceded by a heading. The heading should describe the purpose, function, and methods used in the section it precedes.

All coding must be clearly and thoroughly commented. In general, each line of code requires a comment. The comment must convey the function, meaning, and role of the instruction it describes. A comment cannot be a simple translation of an instruction from alphanumeric symbols into English. If a particularly complex, obscure, or elegant instruction sequence is used, a paragraph of comments explaining what is being done shall precede the sequence. Code that implements a logical or mathematical algorithm is a good example of such a case.

Symbols, acronyms, mnemonics, and abbreviations used in the program listing must be clearly and completely defined by text within the listing. Names, abbreviations, acronyms, mnemonics, and symbols of hardware elements must be the same as used in the hardware specifications. Comments and sectional headings must clearly describe program actions when an error is detected.

Programs containing MACROs should expand the MACROs and explain in detail their functions and implementations. This should be done once in detail at the start of the source files. The explanation need not be repeated each time the MACRO is used in the program.

### 10.13 SYMBOL TABLE AND CROSS REFERENCE LISTING

Programs must contain symbol tables and cross reference listings when required by the diagnostic users.

### 10.14 PROGRAM FUNCTIONAL DESCRIPTION

This section of the documentation file should include the following information categories.

1. General information
  - a. Program abstract  
This is an overview of the diagnostic program
  - b. System requirements  
What hardware and software is needed for this diagnostic to run
  - c. Related documents and standards  
What other documents and standards should be read
  - d. Diagnostic prerequisites  
What diagnostics need to run before information from this one is valid. This is bottom up testing
  - e. Assumptions-restrictions  
What are the restrictions and assumptions used when running this diagnostic
2. Operating instructions  
Describe the command switches and modes of operation
3. Error information  
Describe the ERROR TYPES and the specific error information given by this diagnostic
4. Performance reports  
Describe any performance or progress reports that the diagnostic provides
5. Device information tables  
Describe device table information
6. Test descriptions  
Describe all tests in the diagnostic
7. Other text (e.g., Troubleshooting hints)

For each test (and subtest, if appropriate) explain the functions tested, possible failures, and actions that the operator should take on error detection. Example 10-7 shows the description of the first test in the LA34-VA Hardcopy Terminal functional diagnostic program.



```
.SBTTL REVERSE LINEFEED TEST
; MODULE NAME; REVL2.-1

BGNMOD

; Reverse linefeed / reverse index test 2
;
; This test will exercise the single line reverse linefeed and
; multiline reverse index features of the terminals under test.
;
; The test will verify correct operation at the following vertical
; pitches; 3, 8, 12, and 6 lines per inch.
;
; FA&T MODE will test 6 lpi, 10 cpi only.
;
; Each vertical pitch will be tested at the following horizontal
; pitches; 16.5, 5, and 10 characters per inch.
;
; The test will self modify for 8" paper if "wide" is less than
; 132.
;
; The pattern produced will appear as a zigzag pattern of *
; characters between two reference line of = characters spaced
; ten lines apart.

BEGIN TEST
: Print Test_ID
: Init Pointer to Table of Vert pitches
: Set Diagonal size = 8.
: Repeat for each entry in Vert Pitch Table
:   : Set Vert pitch
:   :   Init pointer to table of Horiz pitches
:   :   Repeat for each entry in Horiz Pitch Table
:   :   : Set Horiz pitch
:   :   : Select column count for current pitch & width
:   :   : Select pattern count for current pitch & width
:   :   : Set top at current line (66 line page)
:   :   : Skip 12 lines
:   :   : Print bottom ref line (column count long)
:   :   : Print a <CR>
:   :   : Send UP10 escape sequence
:   :   : Print top ref line (column count long)
:   :   : Send a <CR>
:   :   : Repeat pattern (Count times)
:   :   :   Create Diagonal down
:   :   :   Create Diagonal up
:   :   : End repeat
:   :   : Skip 12 lines
:   : End repeat
: End repeat
End test
```

Example 10-7. Test Description.

Notice that the test steps are listed in order. You should be able to use material from the functional and design specifications when you write the test descriptions for the documentation file.

### 10.15 DESCRIPTIONS OF SUBROUTINES

```
;++  
; FUNCTIONAL DESCRIPTION:  
;       Updates total char. count TOTCC based on CURCC. Last  
;       message is truncated to fit into the buffer. If total  
;       char. count exceeds "BUFLIM", a message is printed  
;       telling the operator that the truncation occurred.  
;  
; INPUTS:  
;       CURCC= Char. count of message being added  
;       TOTCC= Total char count of buffer it's being added to  
;  
; OUTPUTS:  
;       Message to operator if message truncated to fit  
;  
; FUNCTIONAL SIDE EFFECTS:  
;       Location "TEMP" used for calculations  
; CALLING SEQUENCE:  
;       JSR      PC,ADDCC      ;Updated total char. count  
;--
```

Example 10-8.

```
;++  
;FUNCTIONAL DESCRIPTION:  
; FACSIMILE: This routine is used to create a facsimile of  
; the transmit list and transmit buffer in the expect  
; list and expect buffer. The routine is normally  
; called when user command "SET E [XPECT]=  
; T [RANSMIT] is entered.  
;  
; CALLING SEQUENCE: JSR PC,FACSIMILE  
;  
;--  
; DEFINITIONS CMPBUF = Expected data buffer holds max 512 bytes  
; TXBUF = Transmit data buffer holds max 512 bytes  
; TTOTCC = Number of bytes in TXBUF  
; PTRTAB = Top of message list pointer table  
; CTOTCC = Number of bytes in expect message  
; CMPTOT = Number of expected messages  
; CMPPTR = Expected message list pointer  
; TXPTR = Transmit message list pointer  
; TXMTOT = Number of transmit messages  
; CCURAD = Storage address of message in CMPBUF  
; MSGLIN = Maximum number of messages that can be stored  
; BUFLIN = Number of bytes in buffer  
;  
; BEGIN FACSIMILE ROUTINE  
; (*COPY TXBUF ==> CMPBUF*)  
; ..SAVE R1  
; ..INIT R1  
; ..REPEAT  
; ....[CMPBUF]R1=[TXBUF]R1  
; ....R1=R1+1  
; ..UNTIL R1 = BUFLIM  
;  
; (*NOW CALCULATE EXPECT LIST MESSAGE POINTER*)  
; ..CMPPTR = PTRTAB + (2 * MSGLIM)  
;  
; (*NOW PRIME THE WHILE-DO LOOP*)  
; ..TXPTR = PTRTAB  
; ..CCURAD = CMPBUF  
; ..TXPTR = TXPTR + 2  
; ..CTOTCC = [TXPTR]  
; ..CMPTOT = 0  
; ..WHILE TXMTOT <> CMPTOT DO  
; ....[CMPPTR] = CCURAD  
; ....CMPPTR = CMPPTR + 2  
; ....[CMPPTR] = CTOTCC  
; ....TXPTR = TXPTR + 4  
; ....CCURAD = CCURAD + CTOTCC  
; ....CTOTCC = [TXPTR]  
; ....CMPPTR = CMPPTR + 2  
; ....CMPTOT = CMPTOT + 1  
; ..END WHILE DO
```

```
; ..CTOTCC = TTOTCC  
; END FACSIMILE ROUTINE
```

Example 10-9.

## CHAPTER 11

## GENERAL CODING CONVENTIONS

## 11.1 INTRODUCTION

In designing diagnostic programs, diagnostic engineers should keep in mind who the users and maintainers of the programs will be. Elegant software techniques and complicated coding subtract from, rather than add to, the quality and viability of a diagnostic program. Techniques which may be satisfactory for exercisers or maintenance programs may be too elegant for unit diagnostics. This chapter discusses some of the guidelines to follow to ensure good coding.

When developing basic logic diagnostics, straight line coding techniques should be followed since deviations from straight line coding call for extra documentation efforts. The use of standard subroutines is a justified and mandatory deviation from straight line coding. When standard subroutines are used, however, extra care must be taken to ensure that the program documentation is clear and will cause minimum confusion to the non-programmer user who must refer to the documentation.

Global parameters, constants, literals, tables, and data should be collected and concentrated into easily recognized, contiguous areas of the source code (and memory). Operator parameters should be collected into a single, contiguous, easily recognized area at the beginning (or end) of the source code (and memory). These areas should be clearly identified within the source listing by means of headings and comments.

All registers and vector addresses of peripheral equipment under test or used to output messages or control diagnostic execution should be readily changeable by the operator at program set-up time. Text within the source listing should clearly explain how to use this capability.

Subroutines should return control to the next executable instruction immediately following the subroutine call. This does not preclude the transfer of parameters with a subroutine call, nor does it preclude multiple return points (depending on conditions encountered in the subroutine). The objective is to keep the return address as close to the location that the call was made from as is practically possible.

The following practices are to be avoided:

1. Instructions which modify instructions.
2. Time-dependent code (i.e., code which depends on the execution time or response time of instructions or other operations). Note that this does not preclude the use of a real-time clock to measure the execution or response time of the hardware under test. Time-dependent code could be used if it is calibrated against a real-time clock or the line frequency.
3. Model-dependent code (i.e., code which will not correctly execute on all models of a product line). For example, a disk diagnostic should not be written that will execute on a PDP-11/40 but not on a PDP-11/05. Obviously, PDP-11/45 processor diagnostics need not execute on a PDP-11/05. Note that if a firm commitment is obtained to limit the types of processors the option will be used on, this constraint may not be applicable. This condition almost never exists.
4. Nested subroutines (i.e., subroutines which call subroutines). If nested subroutines are unavoidable, be sure that returns are made in reverse order (i.e., each subroutine returns control to the subroutine which last called it). Also clearly document (flow charts are good) and explain what is being done. Since unit diagnostic programs are the basic diagnostic tools, nested subroutines should be avoided wherever possible.

Values to be used as the basis of comparison for purposes of error determination, such as expected register contents, should be handled as program constants predetermined at coding time. These values should not be dynamically measured by the program itself during execution. There are exceptions to this rule. In multi-unit controllers, for example, the contents of the status/control register will depend on the number of units connected. In such cases the program would use information supplied by the operator as a basis of comparison.

If the contents of a dynamic register is to be processed (e.g., read, compared to a constant, and printed) first store the register and then process the stored value, not the register contents. A dynamic register is one whose contents can change without the explicit initiation of the change by the program. An example of such a register is the word count register of an NPR peripheral during a data transfer. This methodology is preferred even for static registers.

## 11.2 RECOMMENDED CODING PRACTICE

When developing coding, the guidelines discussed in this section should be followed.

### 11.2.1 Line Format

All source lines shall consist of from one to the maximum number of characters supported by the listing media (usually 80 columns). Assembly language code lines shall have the following format:

1. LABEL FIELD - If present, the label shall start at tab stop 0 (column 1).
2. OPERATION FIELD - The operation field shall start at tab stop 1 (column 9).
3. OPERAND FIELD - The operand field shall start at tab stop 2 (column 17).
4. COMMENTS FIELD - The comments field shall start at tab stop 4 (column 33) and may continue to the maximum number supported by the listing media.

Comment lines included in the code body shall be delimited by a line containing only a leading semicolon. The comment itself contains a leading semicolon and starts in column 1. Indents shall be 1 tab.

If the operand field extends beyond tab stop 4 (column 33), simply leave a space and start the comment. Comments which apply to an instruction but require continuation should always line up with the character position which started the comment.

Wherever possible, assembly pseudo-operation control statements should not be printed on the assembly listing.

### 11.2.2 Comments

Comment all coding to convey the global role of an instruction and not simply a literal translation of the instruction into English. In general, this will consist of a comment per line of code. If a particularly difficult, obscure, or elegant instruction sequence is used, a paragraph of comments shall immediately precede that section of code.

Preface text describing formats, algorithms, program-local variables, subroutines, etc. will be delimited by the character sequence `;` at the start of each line of text.

For example:

```
;*This routine accepts a list of  
;*random number and alphabetizes them.
```

### 11.3 NAMING STANDARDS

Names, symbols, and labels should adhere to the following standards:

#### 11.3.1 Hardware Registers

These registers must be named identically with the hardware definition: for example, PS and SWR.

#### 11.3.2 Device Registers

These are symbolically named identically to the hardware notation. For example, the control and status register for the RK disk is RKCS. Only this symbolic name may be used to refer to this register.

#### 11.3.3 General Purpose Registers

Only the following names are permitted as register names and may not be used for any other purpose:

Only the following names are permitted as register names; and may not be used for any other purpose:

```
R0=%0           ;Reg 0  
R1=%1           ;Reg 1  
R2=%2           ;Reg 2  
R3=%3           ;Reg 3  
R4=%4           ;Reg 4  
R5=%5           ;Reg 5  
R6=%6           ;Reg 6  
R7=%7           ;Reg 7  
SP=%6           ;Stack Pointer (Reg 6)  
PC=%7           ;Program Counter (Reg 7)
```

Note: To clarify, R0 can only be used to name register 0, R0 cannot be used to name register 5.



#### 11.3.4 Processor Priority

Testing or altering the processor priority is done using the following symbols:

PRI0, PRI1, PRI2, . . . ., PRI7

These symbols are equated to their corresponding priority bit pattern.

#### 11.3.5 Other Symbols

Frequently used bit patterns such as CR (carriage return) and LF (line feed) will be made conventional symbolics on an as-needed basis. Mnemonic symbol assignments should be defined for frequently used constants. This aids in the clarity of the program and ease of editing and maintaining.

#### 11.3.6 Program-Local Labels

Self-relative address arithmetic (. $+n$ ) is absolutely forbidden in branch instructions and should be used only where absolutely essential elsewhere. Indeed, no implication of adjacency is permitted without showing cause. Non-symbolic absolute references should be avoided.

Labels targetted for branches that exist solely for positional reference will use local labels of the form:

<NUM>\$: NOTE: This format defines "local labels".

Use of non-local labels is restricted, within reason, to those cases where reference to the code occurs external to the code. Local-labeling is formatted with the numbers proceeding sequentially down the page and from page-to-page. It should be noted that macros, when expanded, will violate this guide. This is acceptable since it is not avoidable.

### 11.4 PROGRAM MODULES

No other characteristic has more impact on the ultimate engineering success of a system than does modularity. It provides the means to lay out the program for ease of coding, understanding, and revising.

#### 11.4.1 The Program Preface

Programs must adhere to a strict format. This format adds to the readability and understandability of the program. The following sections are included in each program:

For the code section (in the source listing, NOT the assembly listing) of a SYSMAC diagnostic program:

1. Listing and assembly directives:
  - a. "No list" macro calls and definitions and unsatisfied conditionals (.NLIST MC,MD,CND).
  - b. List macro expansions (.LIST ME).
  - c. Enable, if appropriate, absolute addressing and absolute mode (.ENABL ABS,AMA).
2. A ".TITLE" statement that specifies the name of the program and the MAINDEC number with revision level.
3. The name of the principal author and the date on which the program was first created (Not to appear on the assembly listing).
4. The name of each modifying author, the date the modification was completed, and the purpose of the modification. The information concerning each modification shall occupy one line and the modifications shall be listed in chronological order (see Sec. 10.2.2).
5. A .SBTTL statement that defines the program section or subroutine that follows.
6. Switch Settings
7. Trap Catcher
8. Starting Address or Addresses (includes restart address).
9. A list of the definitions of all equated symbols used in the program. These definitions appear one per line and shall be broken into categories with each category in logical order (alphanumeric).
10. All local macro definitions, preferably in logical order by name. A description should be provided for each macro. Additionally, each macro should be adequately commented to allow easy understanding when it is expanded.

11. All constant, variable, and table data. The data should indicate:
  - a. Description of each element (type, size, etc.)
  - b. Organization (functional, alpha, adjacent, etc.)
  - c. Adjacency requirements
12. The program code. A detailed description of each test and sub-test shall precede the start of code. This description should describe what is being performed, what portion of the logic is being tested, any assumptions made, and the English language flow of the test itself.
13. All subroutines. A description should be provided for each subroutine. This description should include the calling sequence, the information retrieval procedures, the contents of registers (upon entry and exit), and the information delivery procedure.
14. ASCII messages
15. Any one-shot code
16. SYSMAC macros
17. Buffer area

For a DRS-compatible diagnostic, 13, 14, and 17 would occur before 12.

#### 11.4.2 Register Conventions

When a subroutine is entered, it minimally saves all registers (except for result registers) it intends to alter and, on exit, it restores these registers. State preservation is assumed across calls.

#### 11.4.3 Argument Passing

Any registers may be used, but their use should follow an orderly pattern. For example, if passing three arguments, pass them in R1, R2 and R3 rather than R1, R2, R5. Saving and restoring should not be scattered throughout the routine but each should happen in one place.

#### 11.4.4 Exiting

All subroutine exits occur through a single RTS Rn or RTI.

### 11.5 FORMATTING STANDARDS

The following formatting guidelines should be adhered to.

#### 11.5.1 Program Flow

Programs should be organized on the listing such that they flow down the page as the PDL flows. This organization makes the flow easier to follow. Code should exit the routine at only one point, preferably at the bottom.

```

; ++
; PDL description of a code fragment
; --

;           A = A + 5
;           IF (A .GT. 7)
;             THEN
;             :   D = D + 1
;             :   IF (D .GT. 10)
;             :     THEN
;             :       :   B = B / 2
;             :     ENDIF
;             :   ELSE
;             :     A = C
;             :   ENDIF
; 1$        C = C * 2
; 2$        RETURN
;

```

```

; ++
; Correct method of coding above PDL
; --

```

	ADD	#5, A	; A = A + 5
	CMP	A, #7	; (A > 7) ?
	BLOS	1\$	; No, branch
	INC	D	; D = D + 1
	CMP	D, #10	; (D > 10) ?
	BLOS	2\$	; No, branch
	ASR	B	; B = B / 2
	BR	2\$	;
1\$:	MOV	C, A	; A = C
2\$:	ASL	C	; C = C * 2
	RETURN		;

```
;++  
:Incorrect method of coding above PDL  
;--
```

```
        ADD      #5,A      ;  
        CMP      A,#7      ;  
        BHI     2$         ;  
        MOV      C,A       ;  
1$:     ASL      C         ;  
        RETURN                    ;  
2$:     INC      D         ;  
        CMP      D,#10     ;  
        BLOS    1$         ;  
        ASR     B         ;  
        BR      1$         ;
```

## 11.6 FORBIDDEN INSTRUCTION USAGE

Certain coding conventions, that may on the surface appear to be desirable, are forbidden.

### 11.6.1 Instructions or Index Words as Literals

The use of instructions or index words as literals of the previous instruction is forbidden. For example:

```
MOV      @PC,REGISTER  
BIC      SRC,DST
```

uses the bit clear instruction as a literal. This may seem to be a very "neat" way to save a word but what about maintaining a program using this trick? To compound the problem, this will not execute properly if I/D space is enabled on the 11/45. In this case @PC is a D bank reference.

### 11.6.2 MOV Instead of JMP

The use of the MOV instruction instead of a JMP instruction to transfer program control to another location is not allowed. For example:

```
MOV      #ALPHA,PC
```

transfers control to location ALPHA. Besides taking longer to execute (2.3 microseconds for MOV vs. 1.2 for JMP), the use of MOV instead of JMP makes it nearly impossible to pick up someone else's program and tell where transfers of control take place. As a more general issue, perhaps even other operations such as ADD and SUB from PC should be discouraged. Possibly one or two words can be saved by using these operations but this will not occur very often.

### 11.6.3. Single-Word Instructions

The seemingly "neat" use of all single-word instructions, where a double-word instruction could be used and would execute faster, should not be used. Consider the following instruction sequence:

```
CMP    -(R1),-(R1)
CMP    -(R1),-(R1)
```

The intent of this instruction sequence is to subtract 8 from register R1 (not to set condition codes). This can be accomplished in approximately 1/3 the time via a SUB instruction (3.8 vs. 9.4 microseconds) at no additional cost in memory space. Another consideration, what if R1 is odd? SUB always wins since it will always execute properly and is always faster!

### 11.6.4 PDP-11 Family Instruction Execution Differences

Because of differences in the way PDP-11 CPUs execute some instructions when certain register modes are used, care must be taken when using the same register in the source and destination operands and in using the PC as a source operand.

For example, avoid the use of the same register in both the source and destination such as:

```
MOV R1,(R1)+    or MOV R1,-(R1)
```

The register may or may not be auto-incremented or auto-decremented before use as the source.

The use of the PC can cause problems if used as a source operand, such as:

```
MOV PC,loc
```

because different CPUs update the PC at different times during fetch and execution of the instruction.

The use of auto-increment mode in a JMP or JSR may execute differently, such as:

JMP (R1)+ or JSR R5,(R1)+

R1 may or may not be incremented before use for the destination.

Other areas of difference involve RESET, RTT, RTI, the T-Bit, differences in Memory Management, HALT, Odd-Address references and others. Beware that differences exist and check your programs on all targeted CPUs to be sure you're not caught.

### 11.7 RECOMMENDED CODING PRACTICE - CONDITIONAL BRANCHES

When using the PDP-11 conditional branch instructions, it is imperative that the correct choice be made between the signed and the unsigned branches.

SIGNED	UNSIGNED
BGE	BHIS (BCC)
BLT	BLO
BGT	BHI
BLE	BLOS (BCS)

A common pitfall is to use a signed branch (e.g., BGT) when comparing two memory addresses. All goes well until the two addresses have opposite signs; that is, one of them goes across the 16kw (100000 octal) boundary. This type of coding error usually shows itself as a result of re-linking at different addresses and/or a change in size of the program.





## GLOSSARY

## A

**Absolute (ABS)** - A program section (psect) attribute. This attribute causes the code produced by the assembler to be nonrelocatable. All code is assigned fixed memory locations. The converse of absolute is relocatable (REL) where the code can be relocated to some other locations by the linker.

**Absolute loader** - The monitor for paper-tape-based systems.

## AIDS

**Alignment** - The address boundary at which a program section is based.

**Allocate a device** - To reserve a particular device unit for exclusive use. A user process can allocate a device only when that device is not allocated by any other process.

**Allocation** - The number of bytes of memory contributed by a program section to a particular module.

**Alphanumeric character** - An upper or lower case letter (A-Z, a-z), a dollar sign (\$), an underscore (\_), or a decimal digit (0-9).

**APT-RD** - An automated diagnostic control application used by DIGITAL field service to provide contract customers with quick response and effective on-site repair action.

**Argument** - An independent value within a command statement that specifies where, or on what, the command will operate (e.g., address, data).

**ARRAY** - An area of data storage in which all elemental parts are adjacent, identical in size, and sequentially accessed.

**ASCII** - American Standard Code for Information Interchange.

**ASH** - A PDP-11 instruction, Arithmetic Shift. A member of the Extended Instructions Set (EIS).

**ASHC** - A PDP-11 instruction, Arithmetic Shift Combined. A member of the Extended Instructions Set (EIS).

**Assembler** - A program that translates source language code, whose operations correspond directly to machine op codes, into object language code.

**Assembly** - Creation of a program written in Assembly language, using the MACRO-11 assembler program.

**Assignment** - To make one thing equivalent to another.

**Autodelete** - A possible effect of the file transfer process whereby a file from the input medium replaces a file of the same name on the output medium. In UPD2, only transfers initiated by a FILE command can result in autodeletion.

**Automated Product Test (APT) system** - System which loads and monitors one or more PDP-11 diagnostics into a PDP-11 computer.

**Automated Computer Test (ACT) system** - Used by DEC's manufacturing areas in testing PDP-11 computers.

## B

**Base register** - A general register used to contain the address of the entry in a list, table, array, or other data structure.

**Binary** - Consisting of two things, e.g., binary numbers are 0 and 1.

**Bit** - The smallest piece of data, a 0 or 1.

**Block** - 1. The smallest addressable unit of data that the specified device can transfer in an I/O operation (512 contiguous bytes for most disk devices). 2. An arbitrary number of contiguous bytes used to store logically related status, control, or other processing information (i.e., process control block).

**Boot (bootstrap)** - A program that loads another (usually larger) program into memory from a peripheral device.

**Boot block** - The first physical block on a medium (block zero). This block contains the XXDP+ secondary bootstrap for the device.

**Bootstrap** - The procedure that starts a CPU, which consists of loading and executing a very short program, the bootstrap loader, whose only function is to load and start a larger monitor program.

**BR** - Bus request

**Branch** - A transfer to another part of the program. A GOTO.

**Breakpoint** - In diagnostics, an address assigned through DRS. When the PC equals the value of the breakpoint, control returns to DRS.

Buffer - A section of memory reserved for storing data, usually from a file, as opposed to executable code. A temporary data storage area.

Byte - Eight bits of data, one half of a WORD, the smallest addressable memory location.

## C

Chain mode operation - The sequential execution of programs without operator intervention. Only programs modified to run in chain mode can be chained.

Clean-up Code - The last section of code within a diagnostic program, executed before the program is stopped, which brings all units into a non-ambiguous inactive, error-free state.

Command file - A file containing command strings.

Command string - A line or a set of continued lines (normally terminated by typing the carriage return key) containing a command) and, optionally, information modifying the command. A complete command string consists of a command; its qualifiers, if any; its parameter (file specifications, for example), if any; and their qualifiers, if any.

Condition codes - Four bits in the processor status word that indicate the results of the previously executed instruction.

Console terminal - The video or hardcopy terminal attached to the system via the DL interface at bus address 177560.

Control unit - Interface between drive units and the CPU.

CPU - Central processing unit (same as processor).

Cylinder - The tracks at the same radius on all recording surfaces of a disk pack.

## D

DEBUG - To find and remove errors from a program.

DEC/X11 - UNIBUS exerciser for the PDP-11 computer family which provides a means of testing the expected reliability of a particular system within a specified period of time.

**Default** - Assumed value supplied when a command qualifier does not specifically override the normal command function; also, fields in a file specification that the system fills in when the specification is not complete.

**Default disk** - The system disk to which the system writes all files that the operator creates, by default. The default is used whenever a file specification in a command does not explicitly name a device.

**Default hardware parameter table** - A mock table of specified size and format within the diagnostic program, created by the programmer, which DRS can copy when building the hardware P-tables.

**Delimiter** - A character or symbol used to separate or limit items within a command or data string. However, the delimiter is not a member of the string.

**Device** - The general name for any physical terminus or link connected to the processor that is capable of receiving, storing, or transmitting data. Card readers, line printers, and terminals are examples of record-oriented devices. Magnetic tape devices and disk devices are examples of mass storage devices. Terminal line interfaces and interprocessor links are examples of communications devices.

**Device driver** - That software which has the function of controlling the operation of a specific hardware component in a system. An RX01 driver, for example, is that software that accomplishes such tasks as selecting a physical block, reading a block of information, etc., on an RX01 disk.

**Device fatal error** - An error declared if the diagnostic program detects so many hard errors on the device being tested that it is pointless to continue testing the device or if there is something so catastrophically wrong with the device that it cannot be tested at all.

**Device handler** - see "device driver"

**Device name** - The field in a file specification that identifies the device unit on which a file is stored. Device names also include the mnemonics that identify an I/O peripheral device in a data transfer request. A device name consists of a mnemonic followed by a controller identification letter (if applicable), followed by a unit number (if applicable).

**Diagnostic** - Any program which isolates a hardware fault to the replacement level.

**Diagnostic program header** - The first part of the code of a diagnostic program, consisting of a set of memory locations used for communication between the program and DRS.

**Diagnostic program pass** - Execution of all of the selected tests once on all of the selected units.

**Diagnostic Runtime Services (DRS)** - A program that is loaded in memory to provide a framework for control and execution of diagnostic programs. It provides nondiagnostic services to diagnostic programs.

**Direct I/O** - A mode of access to peripheral devices in which the program addresses the device registers directly, without relying on support from the operating system drivers.

**DMA** - Direct memory access

**Drive** - The electro-mechanical unit of a mass storage device system on which a recording medium (disk cartridge, disk pack, or magnetic tape reel) is mounted.

**Driver** - The part of XXDP+ that handles physical I/O to a device.

**Dropping** - Deselecting a unit.

**Dump** - the process whereby an image of the contents of memory is placed on a storage medium.

## E

**ECO** - Engineering Change Order.

**Edit** - To modify text information in a file.

**Editor** - A utility program used to modify text files.

**EIS** - Extended Instruction Set.

**Entry mask** - A word whose bits represent the registers to be saved or restored on a subroutine or procedure call using the call and return instructions and which includes trap enable bits.

**Event** - A change in process status or an indication of the occurrence of some activity that concerns an individual process or cooperating processes. An incident reported to the scheduler that affects a process' ability to execute. Events can be synchronous with the process' execution or they can be asynchronous.

**Event flag** - A bit in an event flag cluster that can be set or cleared to indicate the occurrence of the event associated with that flag. Event flags are used to synchronize activities in a process or among many processes.

**Executable image** - An image that is capable of being run in a process. When run, an executable image is read from a file for execution in a process.

**Executive** - The generic name for the collection of procedures included in the operating system software that provides the basic control and monitor functions of the operating system.

**Exerciser** - A diagnostic program which runs all of the I/O units at once to test unit interactivity and stamina, along with data integrity causing strenuous and prolonged activity.

## F

**Field replaceable unit (FRU)** - A subassembly or a module or an integrated circuit that may be replaced in the field.

**File** - A logically related collection of data treated as a physical entity that occupies one or more blocks on a volume such as disk or magnetic tape. A file can be referenced by a name assigned by the user. A file normally consists of one or more logical records.

**File specification** - A unique name for a file on a mass storage medium.

## G

**GPR** - General purpose register.

**Global symbol** - A symbol defined in a module that is potentially available for reference by another module. The linker resolves (matches references with definitions) global symbols. Contrast with local symbol.

## H

**Hard error** - One that cannot be recovered from which is so serious that the process being performed cannot continue, e.g., a disk seek error.

**Hardware parameter tables (P-tables)** - A set of tables created by DRS, at program runtime, via operator interaction, containing specific drive-related information for each unit which is to be tested on the system, such as vector addresses, device priority, and baud rate.

Home block - A block in the index file that contains the volume identification, such as volume label and protection.

HW - Hardware

## I

I/O - Input/output

Image - An image consists of procedures and data that have been bound together by the linker. There are three types of images: executable, sharable, and system.

Index file - The file on a FILES-11 volume that contains the access information for all files on the volume and enables the operating system to identify and access the volume.

Init code - Initialization code - The section of the DRS diagnostic program, executed prior to every subpass of the diagnostic program, which retrieves the information contained in the hardware P-table for the unit in question and uses that information to set up the program parameters so the tests will reference that unit.

Interrupt - An event (other than an exception or branch, jump, case, or call instruction) that changes the normal flow of instruction execution. Interrupts are generally external to the process executing when the interrupt occurs.

Interrupt handler - A software routine that services an interrupt.

Interrupt priority level (IPL) - The interrupt level at which the processor executes when an interrupt is generated. There are 8 possible interrupt priority levels on the UNIBUS. IPL 0 is lowest, 7 highest. The levels arbitrate contention for processor service. The QBUS supports only one level. The Q22 bus supports levels 0,5,6, and 7. For example, a device cannot interrupt the processor if the processor is currently executing at an interrupt priority level greater than the interrupt priority level of the device's interrupt service routine.

Interrupt stack - The system-wide stack used when executing in an interrupt service context. At any time, the processor is either in a process context executing in user, supervisor, executive, or kernel mode; or in system-wide interrupt service context operating with kernel privileges, as indicated by the interrupt stack and current mode bits in the PSW. The interrupt stack is not context-switched.

Iterate - Repeatedly execute until a counter exceeds a limit.

## J

**Jump** - Continue execution at the specified address. A GOTO.

## K

**Keyword** - A word used in a language for understandability. These words are not reserved, but are required by the macro.

## L

**Library file** - A direct access file containing one or more modules of the same module type.

**Linked commands** - A group of independent commands connected together (linked) so as to form a single executable list of commands. Once initiated, the entire linked command list may be executed without further operator intervention.

**Linker** - A program that reads one or more object modules created by language processors and produces an executable image file, a sharable image file, or a system image file.

**Linking** - The resolution of external references between object modules used to create an image; the acquisition of referenced library routines, service entry points, and data for the image; and the assignment of virtual addresses to components of an image.

**Literal** - An operand which is used immediately, without being translated to some other value. An operand which specifies itself.

**Literal argument** - An independent value within a command statement that specifies itself.

**Load** - The process whereby the contents of a file containing a program image are placed in memory.

**Load unit protection** - Protection of the load unit medium, if the device being tested is also the load device for the system software, from possible destruction by the program.

**Local symbol** - A symbol that is meaningful only to the module that defines it. Symbols not identified to a language processor as global symbols are considered to be local symbols. A language processor resolves (matches references with definitions) local symbols. They are known to the linker and cannot be made available to another object module.



Logical block - A block on a mass storage device identified by using the volume-relative address rather than the physical (device-oriented) address or the virtual (file-relative) address. The blocks that comprise the volume are labeled sequentially starting with logical block 0.

Logical unit number (LUN) - The numerical designation of a device under test. LUNs are assigned in the order in which units are entered by the operator.

Loopability - The ability of the code contained in a test structure to be continually executed without failing.

LSI - Large-scale integration

## M

Macro - A word or symbol which represents a definition of some part of a program. Parameters may modify the code that is created by the macro.

MACRO-11 - The PDP-11 program which creates an executable program from a source program that was written in assembly language.

Maintenance programs - Any program specifically designed to be used only during preventive maintenance of the hardware.

Manual intervention - Operator intervention, during the execution of a diagnostic program, which can be accomplished via the console terminal or by a physical adjustment on the UUT, such as adding a cable or changing a switch position.

Medium - Physical storage such as a disk or magtape. In this manual, the term "medium" is equivalent to "XXDP+ medium".

Memory management - The system functions that include the hardware's page mapping and protection.

## MINC

Module - A part of a program assembled as a unit. Modular programming allows the development of large programs in which separate parts share data and routines.

MOS - Metal-oxide semiconductor.

**Mount a volume** - To logically associate a volume with the physical unit on which it is loaded (an activity accomplished by system software at the request of an operator). Or, to load or place a magnetic tape or disk pack on a drive and place the drive on-line (an activity accomplished by a system operator).

**MTTD** - Mean-time-to-detect

**MTTR** - Mean-time-to-repair

## N

**Network service protocol (NSP)** - The logical link control layer of DECNET architecture.

**NPR** - Non-processor-request

## O

**Object module** - The binary output of a language processor such as the assembler or a compiler, which is used as input to the linker.

**ODT-11** - On-line Debugging Technique for PDP-11 Computers.

**Operand** - A value (address or data) that is operated on by, or with, an instruction.

## P

**Parameter** - A parameter is the object of a command. It can be a file specification, a keyword option, or a symbol value passed to a command procedure. In diagnostics, parameters are usually operator-supplied answers to questions asked by a program concerning devices to be tested.

**Parameter switch** - A command qualifier. In diagnostics, it is preceded by a slash (/).

**Parser** - A procedure that breaks down on input string into its component parts.

**Pass** - A unit of diagnostic operation. A DRS-type diagnostic pass is defined to be execution of all specified tests on all active units.

**Patch** - A temporary remedy for a problem in a program that is accomplished by altering the program image stored on the XXDP+ medium.  
**Patch area** - A section of free memory which can be used for program patching when necessary.

**PC** - Program counter (register 7)

**Physical address** - The address used by hardware to identify a location in physical memory. A physical address consists of the 16, 18 or 22-bit memory address (depending on memory type).

**Physical block** - A group of data consisting of 256 (decimal) words. This is the standard size of data transmission to and from the XXDP+ media.

**Physical location** - An absolute memory reference (see "virtual location").

**POP** - A keyword meaning "remove from the stack". A macro used to remove data from the stack.

**Position independent code (PIC)** - A program section attribute. The contents of the psect do not depend on a specific location in virtual memory.

**Primary bootstrap** - Code, usually stored in a ROM, which loads the "boot block" (block 0) from a medium into the first 256 (decimal) words of memory and then transfers control to memory location 0.

**Priority** - The rank assigned to an activity that determines its level of service. For example, when several jobs contend for system resources, the job with the highest priority receives service first.

**PRISM**

**Program buffer** - A section of memory used by UPD2 for loading program images.

**Prompt** - A program's typed-out response to and request for operator action.

**PSW** - Processor status word. A register within the PDP-11 processor that contains the current condition of the processor and results of the last operation done (condition-code bits).

**PUSH** - A keyword meaning put onto the stack. A macro used to put data onto the stack.

## Q

**Qualifier** - A portion of a command string that modifies a command verb or command parameter by selecting one of several options. A qualifier, if present, follows the command verb or parameter to which it applies and is in the format: /qualifier:option. For example, in the command string "PRINT <filename> /COPIES:3", the COPIES qualifier indicates that the user wants three copies of a given file printed.

**Queue** - A list of commands or jobs waiting to be processed.

**Quick-verify (QV) application** - Way of running a diagnostic which verifies that all major components are present and functioning, tests all the logic at least once and indicates errors, indicates that no "hard" errors exist or that they no longer exist after a repair, and isolates the failing component in the shortest possible time.

## R

**Radix** - The base of the number system currently in use.

**Record** - A collection of adjacent items of data treated as a unit. A logical record can be of any length whose significance is determined by the programmer. A physical record is a device-dependent collection of contiguous bytes such as a block on a disk, or a collection of bytes sent to or received from a record-oriented device.

**Reserved word** - A word that is recognized as a part of the language or as a directive used by the program. Can not be used as a variable name or tag name.

**ROM** - Read-only-memory.

**Runtime environment** - System level software that is responsible for loading the diagnostic program and executing it: XXDP+, ACT, SLIDE, APT, or paper tape.

## S

**Scope loop** - A loop, containing the code which caused the error to be detected, executed by the diagnostic program providing electrical signals, which can easily be examined with an oscilloscope, on the hardware unit being tested.

**Script file** - A line-oriented ASCII file that contains a list of commands.

**Secondary bootstrap** - Code that resides in the boot block (block 0) of a medium. This code is loaded and started by the primary bootstrap and in turn loads and starts the XXDP+ monitor.

**Section** - A group of tests in a diagnostic program that may be selected by the operator.

**Sector** - A portion of a track on the surface of a disk.

**Segment** - Routines, which can be nested down to 8 levels in depth, contained within a test and delimited by the BGNSEG, ENDSEG macros. A segment causes some hardware activity to occur, checks for a resulting error condition, and reports an error, should it occur.

**Sequential diagnostic program** - A diagnostic program which tests all the units attached to a device, one at a time in sequence.

**Soft error** - An error that potentially can be recovered from, i.e. an error which may go away if the process which detected the error is repeated, e.g., the occurrence of a write-check error when writing data to a medium.

**Software parameter table** - A table used by the programmer to store all of the diagnostic program's runtime software variables, whose values must be obtained from the operator before test execution commences.

**SP** - Stack pointer (register 6)

**Spooling** - Output spooling: The method by which output to a low-speed peripheral device (such as a line printer) is placed into queues maintained on a high-speed device (such as disk) to await transmission to the low-speed device. Input spooling: The method by which input from a low-speed peripheral (such as the card reader) is placed into queues maintained on a high-speed device (such as disk) to await transmission to a job processing that input.

**Stack** - An area of memory set aside for temporary storage, or for procedure and interrupt service linkages. A stack uses the last-in, first-out concept. As items are added to (pushed on) the stack, the stack pointer decrements. As items are retrieved from (popped off) the stack, the stack pointer increments.

**Stack pointer** - General register 6 (R6). SP contains the address of the top (lowest address) of the processor-defined stack.

**Standalone mode** - A diagnostic program environment in which the program and DRS run without the operating system. The operator must use the console terminal when running diagnostics in the standalone mode, and no other users have access to the system.

**Statistical report** - An optional report containing a summary of the activity that occurred during the execution of a diagnostic program; e.g., the total number of disk reads and writes for each unit, enumeration of errors that occurred, etc.

**Sub-pass** - Execution of all of the selected tests once on one of the selected units.

**Subroutine** - A section of code that is executed via a Call to that subroutine. The subroutine accepts input parameters and sends back output parameters as a result of the code within the routine.

**Sub-system** - Logic controller along with its associated device or devices (e.g, disk controller and disk drives).

**Subtest** - Routines, which cannot be nested, contained within a test and delimited by the BGNSUB and ENDSUB macros. Code which tests the smallest logical element of the hardware and can produce one, and only one, type of error.

**SW** - Software.

**Switch** - A modifier for a command.

**Switch register (SWR)** - A set of switches that a program looks at in order to determine which paths of execution it should take.

**Symbolic argument** - An argument within a command that refers to another value.

**Syntax** - the rules governing a command language structure. The way in which command symbols are ordered to form meaningful statements.

**System** - Collection of sub-systems that are so related as to form a logical whole.

**System image** - The image that is read into memory from secondary storage when the system is started up.

**System medium** - The medium on the device from which the XXDP+ system was booted.

## T

**Test** - A unit of a diagnostic program that checks a specific function or portion of the hardware.

**text** - A collection of ASCII formatted data consisting of printing characters, tabs, carriage returns, and form feeds.

**Time stamp** - A statement of the time of day at which a specific event occurred.

**Top** - The most recent entry onto the stack.

**Track** - A collection of blocks at a single radius on one recording surface of a disk.

**Trap** - An exception condition that occurs at the end of the instruction that caused the exception. The PC saved on the stack is the address of the next instruction that would normally have been executed. All software can enable and disable some of the trap conditions with a single instruction. A software interrupt.

**Trap handler** - A software routine that services a trap.

**TSP** - Test-software-package utility (APT).

**TST** - Time-sharing terminal (APT).

## U

**Unattended mode (UAM)** - The running of a diagnostic program without the presence of an operator, selected when the operator sets the UAM flag.

**Unibus exercising** - Testing of several of the same or combinations of different units simultaneously which emphasizes detection over isolation.

**Unit (or device)** - A part of a sub-system (e.g., tape controller; tape drive).

**Unit record device** - A device such as a card reader or line printer.

**Unit test** - Test approach which tests a single unit in order to isolate the malfunction.

**UUT (unit under test)** - The device or portion of the computer hardware being tested by a diagnostic program.

**Utilities** - Tools used to aid in maintenance tasks which are directly aimed at maintaining the programs used for problem detection and isolation (e.g., XXDP+ disc to tape copy program).

## V

**Virtual block number** - A number used to identify a block on a mass storage device. The number is a file-relative address rather than a logical (volume-oriented) or physical (device-oriented) address. The first block in a file is always virtual block number one.

**Virtual location** - A relative memory reference. A program image that has been loaded into the program buffer by UPD2 uses virtual locations; that is, program location 0 is not physical memory location 0. It is the first physical memory location in the program buffer. The XXDP+ monitor does absolute loads and, in this case, program location 0 is not virtual, but is actually memory location 0.

## X

**XXDP+** - XX Diagnostic Package; "XX" is replaced by a two-letter device mnemonic.

**XXDP+ medium** - Physical storage, such as a disk pack, magtape, cassette, etc., that has been formatted for XXDP+ use.



## INDEX

	Page
A	
Abort Test Calls	7-15
Abstract, Program	10-5,6
Access to Flags	7-40
ACT	
Auto Accept Mode	8-6
Dump Mode	8-6
Station Test Mode	8-7
ADD Command	6-19
Add Units Coding	7-8
Adding Units	7-24
APT Mailbox Fields	8-2
Argument Passing	11-7
Arguments, Report Call	7-17
Assignment and Relational Operators	9-12
Assumptions, Program	10-7
Auto Drop (ADR) Flag	6-26
Autodrop section	7-40
Automated	
Applications	1-4
Computer Test (ACT)	1-5,7-2,8-6
Program Test (APT)	1-4,7-2,8-1
- Remote Diagnostic (APT-RD)	1-5
Autotest	1-4
Diagnostic Strategy	3-11
B	
BASIC	9-1,7
Basic	
CPU Cluster Tests	3-3
Error Information	7-16ff.
Functional Testing	4-4
Print Message call	7-18ff.
Batch Control	
Chaining	6-8
Functions	6-9
Handler	6-5
Of Diagnostics	6-8 to 11
Of Utilities	6-10
Begin Message Call	7-19
Bell On Error (BOE) Flag	6-26
BLISS	9-1

Block	
Structure, PDL1	9-2,3
Structured Programming	9-3
Boot Command	6-29
Bottom Up Process	3-1,9
Brackets	9-13
Branching	7-22
Building XXDP+	6-8
Bus	
Problems	3-7
Reset	7-36
Type Check	?
C	
Central Computer Memory Usage	8-9
Chain	
Command	6-12
Making	8-12
Considerations	8-14
Mode	8-7
Operation	8-11
Chaining (Batch Control)	6-8
CHSAA, CHSAB, CHSAC	7-2
Clean-Up Code (Coding)	7-8,32
CLEAR Command	6-29
Clock Macro	7-23
Coding Conventions, General	11-1
Combining Switches	6-23
Comments	9-9
Common Exits	?
Communications Diagnostic Strategy	3-8
Communications Turn-Around System	3-10
Compatibility Testing	5-8
Computer Design Engineers	1-1
Conditional Branches	11-11
Conditionals	6-9
Configuration Compatibility Testing	5-7
Console Terminal Driver	6-5
Consultation Phase	5-1
CONTINUE Command	6-17
Continue On Error	4-2
Control	
Software, Diagnostic	2-5
Transfer	11-7
Count Argument	7-29
CPU and CPU Option Diagnostic Strategy	3-3
CPU Cluster Tests	
Basic	3-3
Extended	3-3

## D

Data, Global	7-7
Default Hardware P-Table	7-6,25
Definition Macros	8-25
Descriptive Text	7-10
Design Reviews	5-7
Device	
Diagnostics	3-7
Drivers	6-2
Fatal Error	7-16
Registers	11-4
Development Process	5-1
Diagnostic	
Applications	1-3
Control Software	2-5
Development Process	5-1
Functionality	
Test Mode	2-4
Troubleshooting and Repair Support	2-4
Hooks	3-6
Macro Library, SYSMAC.SML	8-24
Metrics, Program	1-2,2-1
Microcode	3-4
Prerequisites	10-7
Program Size	2-3
Runtime Services (DRS)	6-1
Sample	7-41
Strategy	3-1
Users	1-1
Direct Support Macros	8-20
Directory Command	6-13
Dispatch Table	7-6
Display	7-29
Command	6-19
Text Command	6-30
Document File	10-2
Documentation	2-3
Aids	7-37
Cover Sheet	10-3
Guidelines	10-2
Drop Command	6-18
Drop Units Coding	7-8
Dropping Units	7-24
DRS	
Commands	6-15
Compatible Diagnostic programs	7-1
Flags	6-22,23,24
Program	
Basics	7-1
Structure	7-6
Switches	6-21

DUMP Command	6-29
E	
Early Exit	9-13
Edit Mode Commands	6-30
Enable Command	6-14
End	
Message Call	7-19
Users	1-3
Engineering	
Breadboard and Prototype Support	5-4
Change Orders (ECOs)	1-1
EOP Switch	6-22
Equates, Global	7-7
Error	
Halts	10-13
Information	10-10
Logging	4-3
Loop	
Control	7-13
Detection	7-15
Number Parameter	7-17
Report	
Calls	7-17
Classes	7-16
Reports, Global	7-7
Reporting	7-16, 10-10
Tables	7-18
Escape Test	7-15
Event Flags	7-23
Execution Times	10-10
Exerciser Emulators	3-10
Exit	
Command	6-20, 29
Routine	7-16
Test	7-16
Exiting	11-8
Explicit CKLOOP	7-14
Extended	
CPU Cluster Tests	3-3
Error Information	7-19
Print Message Call	7-19

## F

## Fault

Detection	1-1
Coverage	2-1
Isolation	2-2,3-6

## Field

Replaceable Unit (FRU)	1-2
Service engineers	1-1,2

## File

Control Services	7-39
Extensions	6-4
Manipulation	6-2
Commands	6-28
Modification Commands	6-28
Naming Conventions	6-3

Fill Command	6-14
--------------	------

Final Diagnostic Implementation	5-4
---------------------------------	-----

Firmware	3-4
----------	-----

First Customer Shipment	?
-------------------------	---

Flag Access	7-40
-------------	------

Flags Command	6-19
---------------	------

Flags, DRS	6-23,24
------------	---------

Flow, Program	11-8
---------------	------

Forbidden Instruction Usage	11-10
-----------------------------	-------

Formatting Standards	11-8
----------------------	------

Functional Specification, Diagnostic	5-2
--------------------------------------	-----

## G

## General

Coding Conventions	11-1
Purpose Registers	11-4

Get Manual Parameters	7-33
-----------------------	------

## Global

Data	7-7
Error Reports	7-7
Equates	7-7
Subroutines	7-7
Text	7-7

Go To Tag	6-9
-----------	-----

Go/No Go Test	4-5
---------------	-----

## H

Halt On Error	4-2
Flag (HOE)	6-24
Hard	
Core	3-3
Verification Tests	3-3
Errors	7-16
Hardcoded	
P-Table(s)	7-12
Hardware	
Parameter Coding	7-9
Parameterization	7-5
P-table Questions	7-25
Registers	11-4
Tests	7-9
Traps	8-28
Header Call	7-10
Header, Program	7-6
Help Command	6-15
Help Commands, SLIDE	8-18
Hlerarchy, Module	9-4
HIPO Diagrams	5-3

## I

Imperatives, PDL1	9-11
Implementation Phase	5-4
Implied CKLOOP	7-13
In-Line Code Macros	8-26
Indirect Support Macros	8-22
Inhibit	
Basic Errors (IBE) Flag	6-25
Dropping of Units (IDR) Flag	6-26
Error Reports (IER)	4-3
Extended Errors (IXE) Flag	6-25
Progress Reports	4-3
Statistical Reports (ISR) Flag	6-26
Initialization	7-31
Code (INIT)	7-31
Coding	7-8
Interactive Program Execution	?
Internal Block Structure	9-11
Interprocessor Test Program (ITEP)	3-9
Interrupt Handling	7-35
Is Manual Intervention Allowed?	7-33

## K

Keywords 9-13

## L

Last Address Generation (LASTAD) 7-11  
 Left Justified Graphics 7-37  
 Line Format 11-3  
 Line Printer, Issuing Commands To 8-17  
 LIST Command 6-29  
 Load  
     Command 6-11,29  
     Device Protection 7-39  
 Loading And Starting Procedures 10-9  
 Local Operator Application 1-3  
 Loop On  
     Error 4-3,7-13  
     Flag (LOE) 6-25  
     Switch ?  
     Test (LOT) Flag 6-27

## M

Macro  
     Package Initialization 7-13  
     Summary 8-20  
 Macros, DRS Program Structure 7-9  
 Mailbox 8-1  
 Manual Intervention, Is it Allowed? 7-33  
 Manufacturing  
     Diagnostic Use 1-2  
     Installation 5-7  
     Technicians 1-1  
 Memory  
     Allocation 7-36  
     Layout, DRS Program 7-1  
 Message  
     Address Parameter 7-17  
     Pointer Parameter 7-17  
     Printout Format 7-18  
 Metrics, Diagnostic 1-2,2-1  
 Microdiagnostics 3-4  
 Module  
     Delimiters 7-11  
     Hierarchy 9-4  
     Screen Diagnostics 1-2  
     Structure 9-5

Monitor	
Commands	6-11
Services Handler	6-5
N	
Naming Standards	11-4
Network Verification	3-9
Nomenclature, XXDP+	6-2
Non-DRS Automated Environments	8-1
O	
Obtaining a Directory	8-10
One Error Report Per Sub-test	4-3
Operating	
Environments	7-2
Instructions	10-8
Modes And Capabilities	4-1
Operational Functionality	2-3
Operator	
Commands, XXDP+	7-2
Interface Handler	6-5
Interrupt Enable (BREAK)	7-35
Optional Sections Selection (POINTER)	7-9
Options, Program	10-10
Other Support Macros	8-24
Over The Line Tests	3-9
P	
P-Table	
Hardware	7-6
Software	7-7
Parameter Coding Calls	7-26
Pass Switch	6-22
Passes And Sub-passes	7-5
PATCH Command Summary	6-29
PATCH Utility	6-6
Patching	8-18
PDL1	
Example	9-16
Format	9-9
Guidelines	9-8



Preformance	
Freeback	5-7
Reports	10-13
Peripheral Diagnostic Strategy	3-5
Physical Fault Insertion	5-8
Plannig Phase, Project	5-2
POINTER	7-9
Print	
Command(s)	6-9,20
Number Of Test (PNT) Flag	6-25
Printer (PRI) Flag	6-25
Printing	
Commands	6-28
Messages (BGNMSG,ENDMSG,PRINTx)	7-18
Proceed Command	6-17
Processor Priority	11-5
Project	
Design Specification	5-3
Goals	5-3
Plan, Diagnostic	5-2
Program	
Abstract	10-6
Assumptions	10-7
Design Language 1 (PDL1)	9-7
Documentation, Diagnostic	10-1
Execution Modes	4-2
Flow	11-8
Functional Description	10-15
Header	7-6
Listing	10-14
Local Labels	11-5
Modules	11-5
Options	10-10
Preface	11-6
Priority	7-38
Self Identification	4-1
Size, Diagnostic	2-3
Structure	
DRS	7-6
Macros, DRS	7-9
Programming	
Considerations	9-2
Languages	9-1
Progress Reports	10-13

## Q

Quality Assurance (QA)	
Checklist	5-5
And Release Phase, Diagnostic	5-5
Quick Verify (QV) Mode	4-5
Quiet Command	6-9
Quit Command	6-9

## R

Read Only Device Driver	6-5
Recommended Coding Practice	11-2
Register Conventions	11-7
Registers	11-4
Reliability Mode	4-5
Report	
Call Arguments	7-17
Coding, Statistical	7-8
Macro	8-21
Reporting, Statistical	7-21
Request Table	7-31
Requirements, Diagnostic	5-3
Restart Command (DRS)	6-16
Returning to Monitor Commands	6-28
Right Justified Graphics	7-38
RUN Command	6-12
Runtime Monitor Sections	6-5

## S

Sample	
Diagnostic	7-41
Program Abstract	10-6
Segment Delimiters	7-12
Selective Blocks, PDL1	9-11
Sequential Blocks, PDL1	9-10
Serial-line Loader In Demand Everywhere (SLIDE)	1-6,7-2
Service Macros, DRS	7-12
Setup	
Command Summary	6-29
Macro	8-22
Utility	6-6
Signal On Error	4-4
Single-Word Instructions	11-11
Size, Diagnostic Program	2-3

SLIDE	1-6,7-2,8-7
Basic software	8-8
Help Commands	8-18
Soft Errors	7-16
Software	
P-Table	7-7
Questions	7-26
Parameter Coding	7-9
Parameterization	7-5
Special	
Environments	10-9
Operating Modes	4-5
Start	
Command	6-12
DRS	6-16
Statistical	
Report Coding	7-8
Reporting	7-21
Strategy, Diagnostic	3-1
Structured	
Design	9-2
Programming	9-1
Subroutine Description	10-18
Subroutines, Global	7-7
Subtest(s)	
Delimiters	7-11
Summaries	10-14
Summary, Macro	8-20
Switches, DRS	6-21,7-4
Symbol Table and Cross Reference Listing	10-15
SYSMAC,SML, The Diagnostic Macro Library	8-24
System	
Core	
Definition	3-1
Diagnostic	
Goals	3-2
Implementation	3-2
Exerciser Diagnostic Strategy	3-7
Fatal Error	7-16
Performance	3-10
Requirements	10-7
Test	1-2

## T

Terminal, Issuing Commands To	8-17
Test	
Algorithms	2-5
Command	6-15
Delimiters	7-11
Description	10-17
Dispatch Table	7-13
Identification Information	7-19
Mode	2-4
Selection Capability	4-1
Station Memory Usage	8-9
Test Switch	6-21
Text, Global	7-7
Top Down Process	3-1
Transfer Calls (XFER)	7-29
Troubleshooting	
And Repair Mode	2-4
Support	2-2,3

## U

Unattended Mode (UAM) Flag	6-26
Underline Character	9-12
Unit	
Number Reporting	7-18
Selection	7-24
Under Test (UUT)	1-5; 2-3; 8-1
Units Switch	6-22
UPD1 command Summary	6-29
Update One (UPD1) Utility	6-6
Update Two (UPD2) Utility	6-6
Updating	8-18
Uisng SLIDE	8-10
Utility	
commands, XXDP+	6-27
Programs	6-2

## W

WAIT Command	6-9
Watchdog Timer	
Commands	8-15
Use	8-15
Wraparounds	3-6

## X

XTECO (Utility)	6-3,4,7
Command Summary	6-30
XXDP+	
Commands	6-11
Device Drivers	6-7
Environments	7-2
Monitor	6-1,4
Utility Commands	6-27
XXDP+, Building	6-8

## Z

ZFLAGS Command	5-21
----------------	------

\*\*\* END OF PDP-11 DIAGNOSTIC DESIGN GUIDE. \*\*\*

