VAX Diagnostic Design Guide

CONTENTS

CONTENTS (Cont)

CONTENTS (Cont)

CONTENTS (Cont)

CONTENTS (Cont)

CONTENTS (Cont)

CONTENTS (Cont)

CONTENTS (Cont)

CONTENTS (Cont)

CONTENTS (Cont)

CONTENTS (Cont)

FIGURES

TABLES

EXAMPLES

EXAMPLES (Cont)

EXAMPLES (Cont)

PREFACE

This manual presents an overview of the VAX diagnostic philosophy and procedures and an explanation of how to write diagnostic programs for VAX Family computers. It is written for diagnostic engineers who are familiar with the VAX-11 Macro assembly language, VAX hardware, the VAX/VMS operating system, and the hardware device to be tested. You can use the manual as a tutorial guide to diagnostic program development or as a reference for specific features of the diagnostic supervisor and diagnostic macro library.

The manual consists of two parts. Part I describes the VAX diagnostic engineering philosophy. It deals with diagnostic goals, functions, methods, and the structure of the VAX diagnostic system. Part II presents system-wide guidelines. It tells you how to write a diagnostic program that will interface with the diagnostic supervisor and that will conform to DIGITAL engineering standards.

Related documentation on VAX systems is listed in the following table.

Related Documents

Title	Document Number	Media
VAX-11 KA780 Central Processor Technical Description	EK-KA780-TD	Microfiche and hard copy
VAX-11 MS780 Memory System Technical Description	EK-MS780-TD	Microfiche and hard copy
VAX-11 DW780 Unibus Adaptor Technical Description	EK-DW780-TD	Microfiche and hard copy
VAX-11 RH780 Massbus Adaptor Technical Description	EK-RH780-TD	Microfiche and hard copy
VAX-11 KC780 Console Interface Board Technical Description	EK-KC780-TD	Microfiche and hard copy
VAX-11 Diagnostic System User's Guide	EK-DS780-UG	Hard copy only

Related Documents (Cont)

Title	Document Number	Media
VAX-11 Diagnostic System Technical Description	EK-DS780-TD	Microfiche and hard copy
VAX-11 Macro Language Reference Manual	AA-D032A-TE	Hard copy only
VAX-11 Linker Reference Manual	AA-D019A-TE	Hard copy only
VAX/VMS Guide to Writing a Device Driver	AA-H499A-TE	Hard copy only
VAX-11 Text Editing Reference Manual	AA-D029A-TE	Hard copy only
VAX/VMS System Services Reference Manual	AA-D01A8-TE	Hard copy only
VAX/VMS Command Language User's Guide	AA-D023A-TE	Hard copy only
VAX/VMS I/O User's Guide	AA-D028A-TE	Hard copy only
VAX-11/780 Architecture Handbook	EB-07466	Hard copy only
VAX-11/780 Hardware Handbook	EB-09987	Hard copy only
VAX-11 Software Handbook	EB-08126	Hard copy only
PDP-11 Peripherals Handbook	EB-07667	Hard copy only
Terminals and Communications Handbook	EB-15486	Hard copy only

NOTES

1. If you wish to order these manuals from within the United States, call Digital Equipment Corporation at either of the two numbers listed below.

   From all areas of the United States except New Hampshire, call (800) 258-1710.

PART I

VAX DIAGNOSTIC ENGINEERING
DESIGN PHILOSOPHY

Part I provides an overview of the VAX diagnostic system purposes, metrics, structure, and procedures. Diagnostic engineers must be familiar with this material in order to write effective programs that contribute substantially to the VAX diagnostic system.

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, and field service or customer engineers. 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, or troubleshooting support, and operational documentation will vary with users and applications.

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 and troubleshooting support, or operational documentation. But poor or incomplete design verification test coverage (mistake detection) can result in costly 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 (eg., GR or microdiagnostics). Also, diagnostic operational documentation is not heavily emphasized because the module screen process is generally automated with the details of diagnostic execution/control masked from the technicians.

A second area of manufacturing diagnostic use is unit or system test, where CPUs, memory systems, I/O 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 (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 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. Their diagnostic requirements include the full spectrum of metrics: fault detection, fault isolation and 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 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 VAX 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.

1.2 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 VAX diagnostic supervisor command functionality and the major diagnostic test design and documentation effort are directed toward local operator effectiveness. Diagnostic scripting, predefined configuration parameter files, 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 (FRU) callout, and listing troubleshooting documentation are examples of local operator fault isolation and repair effectiveness tools. To be totally effective, diagnostic programs must be designed and implemented to achieve excellence in test and repair support effectiveness, operator ease of use, and control effectiveness.

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 the diagnostic supervisor, 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 VAX 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.

1.2.2.1 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 the diagnostic supervisor) 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. This APT interface is totally indistinguishable, to diagnostic programs, from local operation, and totally insensitive to command or program output message content and syntax (associated with local diagnostic operation).

APT as an application, however, 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. (This issue of run-time diagnostic configuration and option selection also applies to local script and local operator diagnostic operation. VAX diagnostic standards specifically disallow mandatory hardware option or test sequence run-time selection.)

Diagnostic program size and load-time considerations are obvious in time sensitive test processes. Although arbitrary program size reduction will 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 APT application.

1.2.2.2 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 supervisor 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 supervisor commands and key on ASCII message output (from the supervisor and 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.

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 introduces 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.

2

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.

Effective diagnostic fault detection coverage is achieved through thorough planning and thorough, conscientious implementation.

Planning
Define the scope of desired testing. The scope of testing is often referred to as the unit under test (UUT). Avoid including functionality that is (or should be) tested by a higher level diagnostic program. For example, disk drive faults detected by a program intended to test just the controller will give misleading failure information and probably frustrate repair of the actual fault.

Define and minimize the diagnostic hard-core functionality (Paragraph 3.1, Chapter 3) which is used by the diagnostic program in testing the UUT. Faults in the hard-core will result in uncontrolled detection (i.e., program crash, unpredictable program operation) and render fault isolation or troubleshooting information ineffective. Hard-core functions should be provided with built-in error detection such as parity or limit checks. The diagnostic program should provide handlers or recovery routines for all predictable hard-core exception situations such as abnormal interrupts or machine checks.

Develop a diagnostic functional specification that details each logic function to be tested, and the proposed method of test. Carefully document the use of all hardware diagnostic aids such as loopback or special controls, and review the specification with the hardware designer. In addition to mapping out the diagnostic test strategy, the functional goals, and operational functionality, the functional specification also provides a reasonable basis for predicting program size and execution time.

Develop a design specification. Do a thorough job implementing the functional specification. Check-off the logic prints as the tests are designed and debugged, and double-check that all significant boundary conditions, exception cases, and interactions are tested. Document during design, not as an afterthought. Subject the diagnostic program to physical fault insertion if feasible and cost justified. (Note that fault insertion has shown that conscientiously planned and implemented diagnostics achieve 85 - 95 percent fault detection coverage. Fault insertion has the greatest importance and payback for diagnostic programs that provide field replaceable unit (FRU) callout or detailed listing-based troubleshooting information.)

2.2 FAULT ISOLATION AND TROUBLESHOOTING SUPPORT
Fault isolation and troubleshooting support are the primary functions of repair diagnostic programs. Fault isolation is defined as explicit identification, via error reports, of one or more FRUs. An FRU may be a subassembly (backplane and modules), one or a few modules, or one or a few 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 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 hardware designer. 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 all (or 90 percent) 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. Because of module density requirements, tristate bus designs, and function interactions, diagnostic isolation based on FRU functional logic partitioning rarely succeeds.

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 VAX-11/780 microdiagnostics (module level FRU).

Even with FRU interconnect visibility, fault insertion quality control (QC) is necessary to measure diagnostic FRU isolation effectiveness (invariably exposing some incorrect callouts).

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, in itself, suffice to 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 kilobytes (KB) 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, or isolation and troubleshooting support effectiveness. The only viable trade-off currently affecting diagnostic program size, is packaging: how test functions are grouped into loadable, executable entities.

Obviously, program size must not exceed the minimum supported system memory size. 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.

Program size can rarely be specified accurately until implementation is well underway. However, it is often necessary to make size estimates earlier than this. Diagnostic functional and program design specifications provide a useful basis for generating reasonably accurate size and execution time estimates. Once the diagnostic program is well-defined functionally and structurally, it is quite possible to compare it to other existing programs where visible similarities or differences (affecting size) can be compared.

2.4 DIAGNOSTIC EXECUTION TIME

Diagnostic execution time is typically the least controllable metric, and fortunately the least critical metric except in unusual test situations. 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.

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. It is not uncommon for large tape or disk units to require 30 minutes for one media test pass. Therefore, diagnostic program execution time is the least controllable metric. However, diagnostic program design should not impose unnecessary pass time requirements by building iterations into each test section.

Finally, the diagnostic engineer should estimate single pass execution time (via the functional and program design specification), review it with the users, and employ thoughtful test algorithms to optimize electromechanical and media test execution time.

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 supervisor are capable of providing to the user. 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: test mode and 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 VAX diagnostic system (supervisor 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 VAX diagnostic supervisor interface conventions, which are documented in Chapter 5 of this manual, will operate in script driven test mode.

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. The VAX diagnostics 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 doing it. 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 supervisor'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 supervisor. However, effective use of these functions depends on the diagnostic test design and proper program interface to (interaction with) these functions.

VAX DIAGNOSTIC SYSTEM: STRUCTURE AND STRATEGY

This chapter describes the structure of the VAX diagnostic system and the underlying strategy behind it. For completeness, the description will include all levels of VAX diagnostic programs (from console-based microdiagnostics through the VMS-based system diagnostic). However, the emphasis will be on the level 3 and level 2 I/O diagnostic programs, the predominant type required.

3.1 VAX FAMILY DIAGNOSTIC STRATEGY

The VAX architecture is intended to be implemented in a family of computer systems spanning a wide range of product cost, functionality, and diagnostic requirements. The VAX Family diagnostic strategy is intended to achieve consistent and appropriate diagnostic effectiveness and functionality across the family, and to minimize the need for redundant diagnostic development and support from implementation to implementation.

Achievement of the VAX diagnostic effectiveness and functionality goals is dependent on careful attention to the key diagnostic applications (Chapter 1), metrics (Chapter 2), and adherence to a sound diagnostic development process (Chapter 4). Awareness of the VAX diagnostic strategy and structure rationale will help ensure consistency of implementation. Six program levels make up the VAX diagnostic system, as follows.

Level 1 -- Operating system (VMS) based diagnostic programs [using logical or virtual queue I/O (QIO)]

Level 2R -- Diagnostic supervisor-based diagnostic programs (restricted) that can be run only under VMS (using physical QIO)

        Certain peripheral diagnostic programs
        System diagnostic program

Level 2 -- Diagnostic supervisor-based diagnostic programs that can be run either under VMS (on-line) or in the standalone mode (using physical QIO)

        Bus interaction program
        Formatter and reliability level peripheral diagnostic programs

Level 3 -- Diagnostic supervisor-based diagnostic programs that can be run in standalone mode only (using direct I/O)

        Functional level peripheral diagnostic programs
        Repair level peripheral diagnostic programs
        CPU cluster diagnostic programs

Level 4 -- Standalone macrodiagnostic programs that run without the supervisor

Hard-core instruction test

Console
Level -- Console-based diagnostic programs that can be run in the standalone mode only

Microdiagnostics
Console program
Octal Debugging Technique (ODT)
ROM resident power-up tests
LSI-11 diagnostic programs

These six program levels operate in the context of four environments: user, system, cluster, and console. These four environments, in turn, run within two operating modes: on-line (under VMS) and standalone (without VMS). Figure 3-1 gives a schematic representation of these relationships.

The four environments form the basis of the building block diagnostic approach. For each environment a portion of the hardware functions as a hard-core, which is assumed to be good. Specific diagnostic programs operate from the hard-core of this environment to test the hardware in an area beyond the hard-core. The hard-core for each environment consists of the hard-core of the next lower environment plus the area tested in that lower environment. Figure 3-2 shows the building block structure of the diagnostic environments.

ON-LINE MODE (CONTROL FROM ANY TERMINAL ON SYSTEM)	USER ENVIRONMENT		LEVEL 1 (VIRTUAL QIO)
		DIAGNOSTIC SUPERVISOR	LEVEL 2R (PHYSICAL QIO)
STAND ALONE MODE (CONTROL FROM CONSOLE TERMINAL, OFF-LINE)	SYSTEM ENVIRONMENT		LEVEL 2 (PHYSICAL QIO)
	CLUSTER ENVIRONMENT		LEVEL 3 (DIRECT I/O)
			LEVEL 4 (MACHINE-LEVEL)
	CONSOLE ENVIRONMENT		CONSOLE LEVEL (SUB-MACHINE LEVEL)

TK-3007

Figure 3-1 VAX Diagnostic System: Program Levels,
Environments, and Operating Modes

USER
MODE

STAND ALONE
MODE

CONSOLE
HARDWARE

CONSOLE
ENVIRONMENT
HARD-CORE

CONSOLE
ENVIRONMENT
AREA UNDER
TEST

CPU CLUSTER
ENVIRONMENT
HARD-CORE

CPU CLUSTER
ENVIRONMENT
AREA UNDER
TEST

SYSTEM
ENVIRONMENT
HARD-CORE

SYSTEM
ENVIRONMENT
AREA UNDER
TEST

USER
ENVIRONMENT
HARD-CORE

USER
ENVIRONMENT
AREA UNDER
TEST
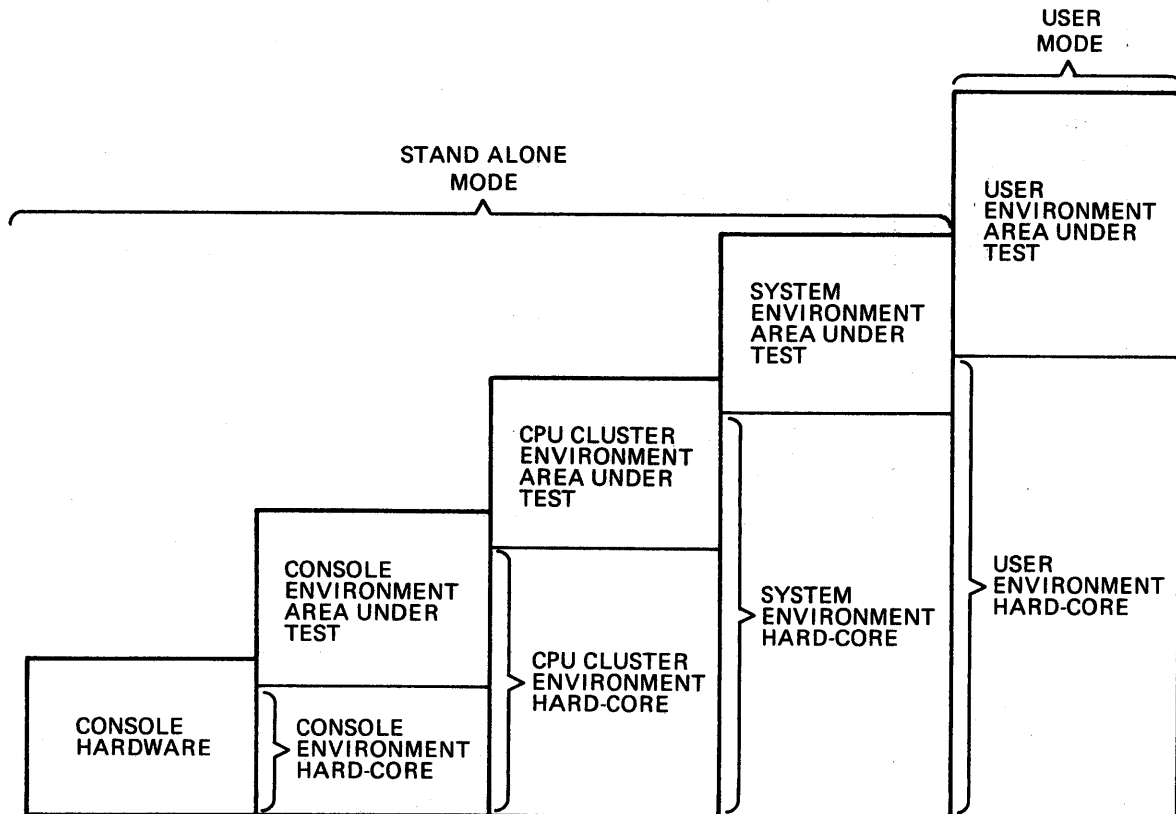
TK-3009

Figure 3-2 The Building Block Structure of the
Diagnostic Environments

3.1.1 Console Environment

The console environment operates in the standalone mode only. The operator controls the system from the console terminal. This environment consists of submachine level hardware, software, and firmware. It provides fundamental operator control functions, system programmer debugging functions, and basic (kernel) machine diagnostic functions. Figure 3-3 shows the console environment configuration. The console hardware forms the hard-core that must be good in order to run the microdiagnostics. Notice that the CPU microcode remains untested in the console environment.

From a diagnostic strategy standpoint, the console environment is the most basic, most implementation-specific piece of the diagnostic system. It ranges from the extensive capability and functionality of the VAX 11/780 console (LSI-11 subsystem) to totally ROM based quick verify tests in lower priced VAX CPUs.

The VAX console environment (also called kernel logic) diagnostic strategy is to implement, at minimum, thorough fault detection of the CPU kernel logic. Fault isolation to a replaceable module or integrated circuit (IC) will be performed on those VAX systems where justified by the system, price, market, and maintenance strategy.

3.1.2 CPU Cluster Environment

Like the console environment, the CPU cluster environment operates in the standalone mode only. This environment consists of the console environment plus the machine level components (complete CPU, memory, I/O channels) that support standalone, macro-level program execution. Figure 3-4 shows the CPU cluster environment configuration. The hardware tested in the console environment, by the microdiagnostics, forms the hard-core of the cluster environment.

The VAX CPU cluster environment diagnostic strategy is to implement a small number of level 4 and level 3 diagnostic programs which, in a building block fashion, test basic CPU/memory functionality, extended CPU/memory functionality, I/O channel operation, and CPU/memory/I/O channel/cluster interaction functionality. The I/O channel and cluster interaction diagnostic programs make full use of channel loopback capability and special plug-in exerciser units (manufacturing use) to maximize test effectiveness.
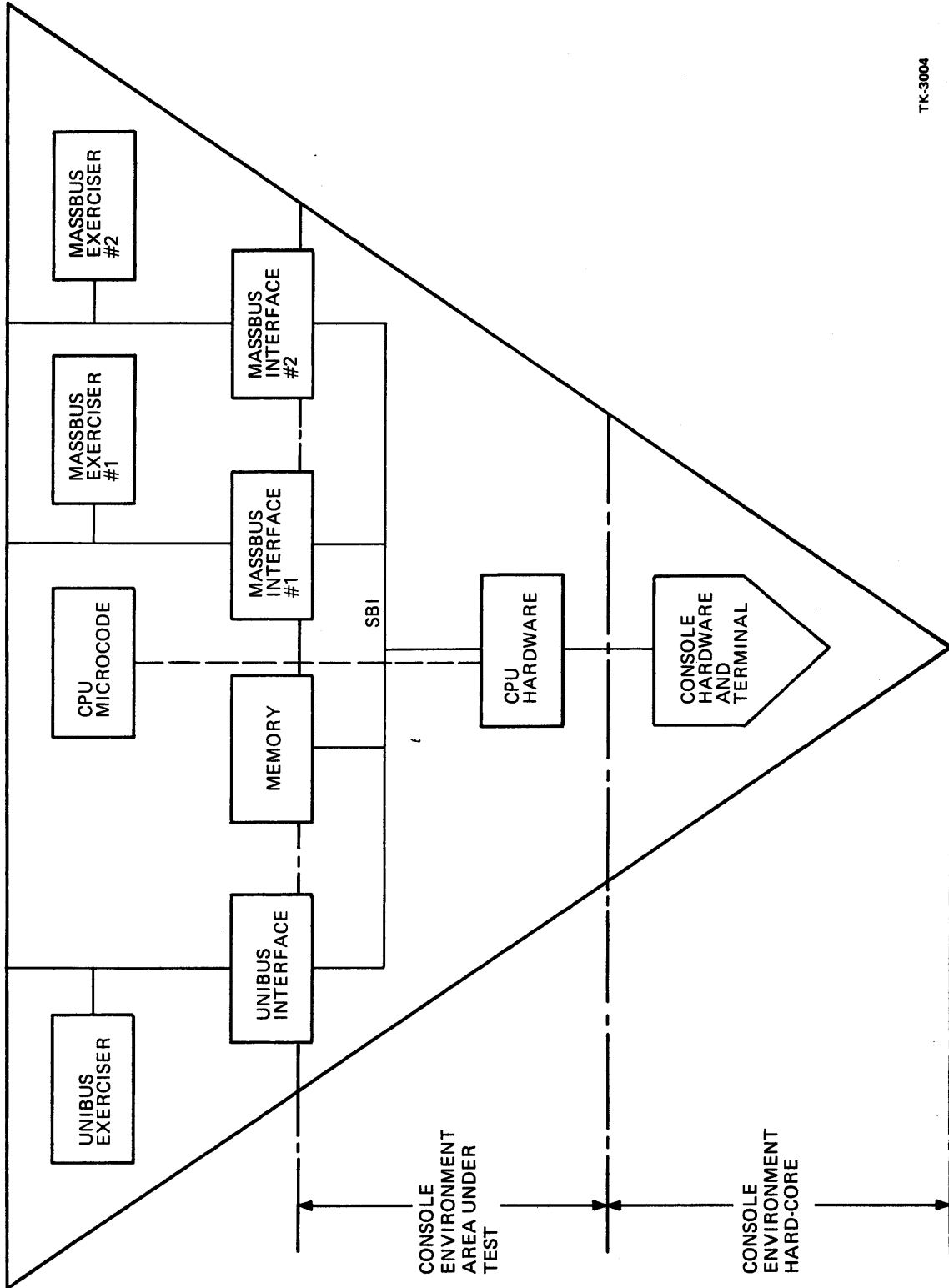
Figure 3-3 Console Environment
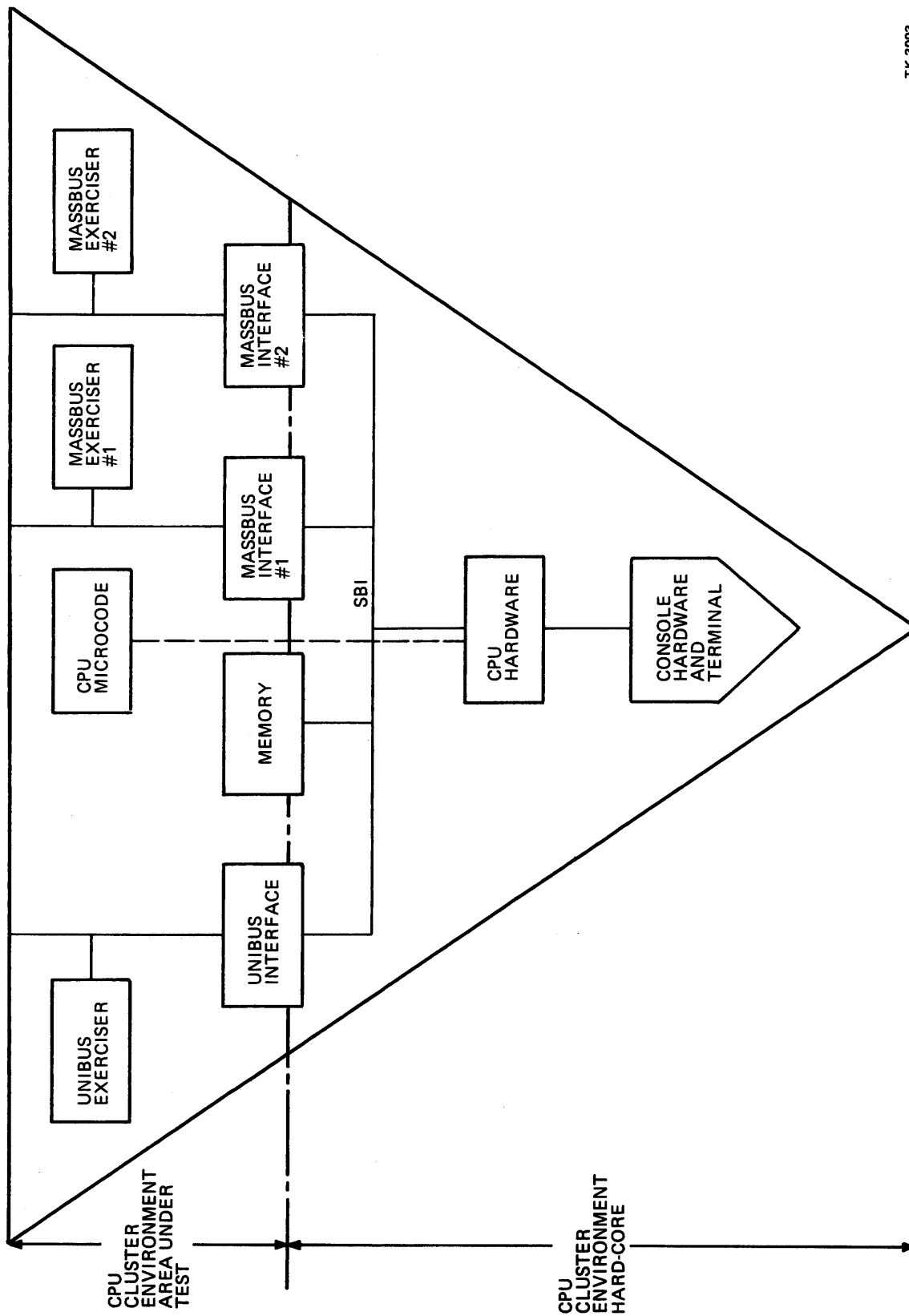
TK-3004

TK-3003

Figure 3-4 CPU Cluster Environment

VAX Diagnostic Design Guide

3.1.3 System and User Environments

The system and user environments consist of the CPU cluster environment plus the I/O subsystems (disk, tape, communication, application) that make up the useful, application solving system components. In the system environment, at the Unibus, Massbus, DRXXbus interconnect level, the proliferation and variety of I/O subsystems generate a strong demand for ongoing diagnostic program development.

3.1.3.1 System Environment - The system environment operates only in the standalone mode. The operator must use the console terminal. This environment contains a wide spectrum of diagnostic programs ranging from level 3 repair diagnostics through level 2 (QIO) device exercisers.

The VAX system environment level diagnostic strategy is to implement a series of level 3 repair diagnostic programs and level 2 functional diagnostic programs for each I/O subsystem. The diagnostic series (level 3 and level 2) for each I/O subsystem is designed to give building-block test coverage. This evolves from static logic and maintenance loopback tests (level 3) through basic function and electromechanical timing tests (level 3 or 2) to media reliability, acceptance and multidevice exercisers (level 2).

Figure 3-5 shows the system environment configuration in a typical VAX system. The hardware tested in the CPU cluster environment forms the hard-core for the system environment.

3.1.3.2 User Environment - The VAX user environment operates in the on-line mode, under VMS. The operator can control the diagnostic process from any terminal on the system, including the console terminal. The user environment includes the level 2 diagnostic programs, which run in the system environment, as well as the level 2R programs, which do not. Many of the diagnostic programs that run in the user environment will run simultaneously with user application programs. However, some, like the system diagnostic program, require exclusive use of the computer system.

3-8

SYSTEM
ENVIRONMENT
AREA UNDER
TEST

SYSTEM
ENVIRONMENT
HARD-CORE

LINE
PRINTER

MODEM

TERMINAL

TERMINAL

TAPE
DRIVE

TAPE
DRIVE

DISK
DRIVE

DISK
DRIVE

LINE
PRINTER
CONTROLLER

COMM
CONTROLLER

ASYNC
MUX

TAPE
FORMATTER

TAPE
FORMATTER

DISK
CONTROLLER

DISK
CONTROLLER

UNIBUS

MASSBUS
#1

MASSBUS
#2

UNIBUS
EXERCISER

CPU
MICROCODE

MASSBUS
EXERCISER
#1

MASSBUS
EXERCISER
#2

UNIBUS
INTERFACE

MEMORY

MASSBUS
INTERFACE
#1

MASSBUS
INTERFACE
#2

SBI

CPU
HARDWARE

CONSOLE
HARDWARE
AND
TERMINAL

TK-3012

Figure 3-5 System
Environment

3-9

3.1.4 Guidelines for the Use of the System and User Environments

The following guidelines should help diagnostic engineers to determine the sets of diagnostic levels and functions applicable for their target I/O subsystems and diagnostic effectiveness requirements. Each level possesses a set of diagnostic characteristics and capabilities. However, each level imposes constraints as well.

3.1.4.1 System Environment Level 3 Diagnostics

A. <u>Diagnostic Capabilities</u>
Direct I/O device access (maximum hardware visibility)
Unrestrained use of device maintenance logic, loopback, unorthodox command sequences, etc.
Microsecond range timer capability
Dedicated use of CPU cluster and I/O channel functions (i.e. standalone)
Tightest scope loops

B. <u>Constraints</u>
Execution in standalone mode only

C. <u>Considerations</u>
Performance of all I/O channel programming functions via the diagnostic supervisor channel services interface (Chapter 5) to achieve VAX system transportability
Standalone debug and test facilities are required
No VMS driver support is required

3.1.4.2 System/User Environment Level 2 Diagnostics

A. <u>Diagnostic Capabilities</u>
Physical (privileged) QIO device access
Full device functional test capability
Millisecond range timer capability
VMS (privileged user) and standalone operation
Device test in the VMS/application execution environment is possible
Execution with the VAX system diagnostic is possible.

B. <u>Constraints</u>
Device access restricted to QIO functions
No direct control over I/O channel functions such as interrupts
Share CPU cluster time and functions in user mode
Development of VMS physical I/O driver is required

C. <u>Considerations</u>
Appropriate for most functional level diagnostic programs where standard VMS support is planned
Especially appropriate for lengthy (run-time) media reliability or acceptance tests
VAX system transportability is achieved through VMS/diagnostic supervisor level 2 interface.

3.1.4.3 System Exerciser Tests (Level 2R)

A. <u>Capabilities</u>

Concurrent execution and full control of several level 2 or 2R diagnostic programs from one terminal are possible (Paragraph 3.1.5)

Three system diagnostic modes are provided:

Quick verify mode
Acceptance mode
Conversation mode

All standard level 2 diagnostic programs run without modification

B. <u>Constraints</u>

Execution under VMS only (level 2R)

Virtual dedication of system resources to testing is implied

C. <u>Considerations</u>

Same as for level 2 (Paragraph 3.1.4.2)

3.1.5   The VAX System Diagnostic Program (ESXBB)

The VAX System Diagnostic Program is a privileged process that runs with the VMS operating system. ESXBB performs a multiplexer function enabling a single VMS operator terminal to load, start, control (with the full set of diagnostic supervisor functions) and receive test results from any level 2 diagnostic program. Throughout the test session, ESXBB allows the operator to exercise individual diagnostic control and device test selection.

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
>   QA and release

4

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.

4.1 CONSULTATION PHASE

The consultation phase of diagnostic development is an informal information gathering and exchange process that begins as soon as engineering or product management admits to a project and is willing or anxious to talk about it. It is usually a part-time effort (less than 25 percent) requiring an experienced diagnostic project leader or technical supervisor to work with engineering, field service, and manufacturing to formulate diagnostic strategy, key project milestones, and preliminary staffing requirements.

The consultation phase typically starts 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 a project startup consultation phase reduces his opportunity for early diagnostic inputs and hinders project team building.

4.2 PLANNING PHASE

The diagnostic development planning phase can begin in earnest when the cursory diagnostic project plan 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 (3 or more diagnostic engineers and/or 9 month or more duration), the following diagnostic planning documents should be developed:

>   Diagnostic project plan
>   Diagnostic functional specification
>   Diagnostic program design specification

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

4.2.1 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). 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.

4.2.2 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: diagnostic goals, diagnostic requirements, and the development process:

A. Diagnostic Product Goals:
    Intended users (design engineering, field service, manufacturing)
    Intended applications (local operator test, repair, APT, APT-RD)
    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 environments (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)

4.2.3 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
(defined in the functional specification) 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 (PDL)

Development of an appropriate program design representation --
from overview level hierarchy charts to detailed PDL 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.

4.3 IMPLEMENTATION PHASE

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.

4.3.1 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.

4.3.2 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 (Paragraph 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 interim 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 prepared.

The diagnostic engineer should prepare the QA checklist according to the goals set up in the functional specification and the VAX diagnostic engineering standards and conventions (see Chapter 14 of this manual for details). 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 field service engineering representatives.

Support of early manufacturing startup and support of in-house and customer field test units normally require the pre-release of diagnostic programs.

4.4 DIAGNOSTIC QA AND RELEASE PHASE

The final stage in the diagnostic development process is the QA effort leading to formal diagnostic release. The QA process should be planned (via the QA checklist, Chapter 14 of this manual) and reviewed (via the pre-release review) to ensure that all specified diagnostic user applications (Chapter 1) and diagnostic user 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. Full fault insertion QA, required for fault isolating repair diagnostic programs, requires one to two weeks per hardware module. 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.

PART II

SYSTEM-WIDE GUIDELINES

Part II describes those features in the VAX diagnostic system common to all diagnostic programs, in particular, those which run under the VAX diagnostic supervisor (levels 2, 2R, and 3). In addition, Part II provides standards and guidelines for the diagnostic engineer concerning program interface with the diagnostic supervisor, proper use of macros and the supervisor library, program structure, and program debugging.

The diagnostic supervisor (ESSAA) is fundamental to the VAX diagnostic system. Most diagnostic programs developed to test the central processor, channel adapters, and peripheral devices on VAX Family computers should be designed to interface with the supervisor. The supervisor is a program that resides in memory together with a diagnostic program. It provides a framework for control and execution of diagnostic programs, and it provides nondiagnostic services to diagnostic programs.

In addition, the supervisor incorporates all system-specific features of the diagnostic system, enabling transportability of peripheral device diagnostic programs between VAX implementations. A disk diagnostic, for instance, should run on a small VAX system as well as on a VAX-11/780 system.

The supervisor runs in three environments:

> CPU Cluster Environments
> System Environments
> User Environments

The CPU cluster environment and the system environment operate in the standalone mode (without VMS). In this mode, the supervisor and the diagnostic program have exclusive control of the computer system. The user environment operates only in the on-line mode (under VMS), sharing the computer system with user applications (Figure 3-1 in Chapter 3). The CPU cluster environment supports only the level 3 diagnostic programs that test the central processor, memory, and the channel adapters. The system environment supports level 3 and level 2 peripheral device diagnostic programs. The user environment supports level 2 and level 2R diagnostic programs (refer to Chapter 3 for details).

In addition, the CPU cluster environment and the system environment can be modified for automated product testing (APT).

5.1 SUPERVISOR FUNCTIONS FOR THE DIAGNOSTIC ENGINEER AND THE USER

The common services provided by the three operating environments of the supervisor are necessary for test operation, but they are not directly related to the testing of a device. Incorporation of these functions in the supervisor leaves the diagnostic engineer free to concentrate on the device. In addition, a subset of the supervisor commands includes debug and utility features such as deposit, examine, and breakpoints.

The framework that the supervisor provides for VAX level 2, level 2R, and level 3 diagnostic programs frees the operator from the need to have a detailed knowledge of each program. A general knowledge of the programs to be run and a familiarity with the supervisor commands are sufficient to make good use of the diagnostic programs.

The supervisor commands enable the operator to load and run the diagnostic programs and to set flags that control program execution. The control flags and commands are program independent and, therefore, are consistent across the range of diagnostic programs.

5.2 SUPERVISOR MACRO LIBRARY

The macros in the supervisor macro library (DIAG.MLB) function as a high level diagnostic language supplement to the language (VAX-11 Macro or Bliss) used by the programmer. These macros fall into three categories, according to the functions they perform and the ways they are implemented.

5.2.1 Utility Macros

The utility macros provide a variety of services for the program. The program format utility macros provide assembler and linker directives that aid in the interface between the diagnostic program and the supervisor.

The program control utility macros enable the program to test specific conditions and to alter the flow of the program. Some of these macros call supervisor services to perform the required functions. Others merely generate in-line executable code.

The symbol definition macros save the programmer great effort by defining many of the global symbols required by most programs. These macros generate assembler directives.

Refer to Chapter 7 for a more complete explanation of the utility macros.

5.2.2 Supervisor Service Macros

The supervisor service macros call supervisor service routines to perform specific functions. They do not, generally, alter the flow of the program. Most of the supervisor routines return status codes. The supervisor service macros call routines that provide the following functions:

        Program control
        Channel control
        Memory management
        Program delay
        Error reporting
        Program-operator dialogue control
        System control
        Hardware P-table address retrieval

Chapter 8 gives a more complete description of the supervisor service macros.

5.2.3   VMS Service Macros
A subset of the VMS service macros is available to level 2 and level 2R diagnostic programs. A small number of these macros are also available to level 3 programs.   When the supervisor runs on-line, the service calls are mapped through the supervisor to the required VMS routines.   In the standalone mode, however, the supervisor emulates these services. Six types of VMS services are available to diagnostic programs:

> I/O services
> Event flag services
> Timer services
> Formatted ASCII output services
> Memory management services
> Hibernate and wake services

Refer to Chapter 9 for a more compete explanation of the VMS service macros.

5.3   DIAGNOSTIC SUPERVISOR COMMANDS
The diagnostic supervisor commands are grouped in four sets:

> Program and test sequence control
> Scripting control
> Execution control
> Debug and utility control

The debug and utility features are listed in Chapter 14. Commands, switches, and literal arguments may be abbreviated to the minimum number of characters necessary to retain their unique identity.   For example, the Load command can be specified by a single L, whereas the Start command requires a minimum of ST.

In the symbolic command descriptions that follow, certain special characters are employed which require some explanation.   Angle brackets, < >, are used to enclose symbolic arguments that are satisfied by a numeric expression or character string.   Optional arguments are enclosed by square brackets, [ ].   An OR function is indicated with an exclamation point,!.   Literal arguments such as ALL, OFF, and FLAGS are capitalized.

Use the hyphen, -, as a continuation character at the end of a line to continue a command from one line to the next.   Use an exclamation point, !, to separate a comment from a command in a command line.

Notice that operator input is underlined in the examples that follow.

5.3.1   Program/Test Sequence Control Commands

These commands enable the operator to select programs and portions of programs and to control the sequence of test execution.

Set Load Command

SET LOAD <device>:[directory]<CR>

The Set Load command establishes the storage device from which the supervisor will load diagnostic programs.   Use the Set Load command in combination with the Load command or the Run command.

```
DS> SET LOAD DMA0:[SYSMAINT]
DS> LOAD ESDXA

DS> SET LOAD DMA0:[SYSMAINT]
DS> RUN ESDXA
```

Example 5-1   Set Load Command

NOTE
The directory name, and the square brackets around it, are necessary in the Set Load command.

Show Load Command

SHOW LOAD<CR>

The Show Load command causes the supervisor to display the storage device from which diagnostic programs are to be loaded when the Load command is given.

```
DS> SHOW LOAD
DMA0:[SYSMAINT]
DS>
```

Example 5-2   Show Load Command

Load Command

LOAD <file-spec><CR>

This command loads the specified file into main memory from the default load device.   The default file extension is .EXE.   The storage device from which the program is loaded is the device established on the previous Set Load command.

Note that you need supply only the five-letter code that identifies each diagnostic program for the command line argument <file-spec>.

For example:

<u>LOAD ESTAA</u>	! Load the local terminal ! diagnostic program.

<div align="center">Example 5-3   Load Command</div>

Attach Command

ATTACH <UUT-type> <link-name> <generic-device-name> . . .<CR>

The operator should use several Attach commands, before starting a diagnostic program, to define each unit under test (UUT), and the devices which link it to the SBI, for the supervisor. If you are testing several units at once, repeat the Attach command for each device. Every unit under test is uniquely defined by a hardware designation and a line.

The first parameter <UUT-type> is the hardware designation of the unit under test. For example, RH780, TM03, TE16, and DZ11 are hardware designations.

The second parameter <link-name> is the generic name of the piece of hardware that links the unit under test, in most cases through intermediate links, to the main system bus. For example, an RH780 is linked to the SBI. A TU45 is linked to an MTa; and a DZ11 is linked to a DWn. You must attach each piece of hardware (with the exception of the SBI) before you can use it as a link in an Attach command.

The third parameter is the generic device name, which identifies to the supervisor the particular unit to be tested. Use the form "GGan" for the device name. "GG" is a 2-character generic device name (alphabetic). "a" is an alphabetic character, specifying the device controller. "n" is a decimal number in the range of 0-255, specifying the number of the unit with respect to the controller.

Use the unit number, "n" or "a", only if it is applicable to the device. You must supply additional information for some types of hardware to enable the diagnostic program to address the device. For example, you must supply the TR and BR numbers for an RH780, the controller number for a TM03, and the CSR vector and BR for a Unibus device. If you include such additional information in the Attach command line, use the order and format shown in Table 5-1. If you do not include additional information, but the information is necessary, the supervisor will prompt you for it.

Table 5-1   Device Naming Conventions

Type	Link	Generic	Additional Information
KA780	SBI	KAa	\<G-floating> \<H-floating> \<WCS-last-address>
MS780	SBI	MSa	\<tr>
RH780	SBI	RHa	\<tr> \
DW780	SBI	DWa	\<tr> \
RP07	RHa	DBan	
RP06	RHa	DBan	
RP05	RHa	DBan	
RP04	RHa	DBan	
RM03	RHa	DRan	
RK611	DWa	DMa	\<ucsr> \<uvector> \<ubr>
RK07	DMa	DMan	
RK06	DMa	DMan	
TM03	RHa	MTa	\<drive>
TE16	MTa	MTan	
TU45	MTa	MTan	
TU77	MTa	MTan	
DZ11	DWa	TTa	\<ucsr> \<uvector> \<ubr> \<EIA> ! \<20MA>
DUP11	DWa	XJan	\<ucsr> \<uvector> \<ubr>
DMC11	DWa	XMan	\<ucsr> \<uvector> \<ubr>
KMC11	DWa	XMan	\<ucsr> \<uvector> \<ubr>
LP11	DWa	LPa	\<ucsr> \<uvector> \<ubr>
CR11	DWa	CRa	\<ucsr> \<uvector> \<ubr>
DR11B	DWa	??a	\<ucsr> \<uvector> \<ubr>
PCL11	DWa	??a	\<ucsr> \<uvector> \<ubr>
TS04	DWa	MTan	\<ucsr> \<uvector> \<ubr>
RL02	??a	??an	\<ucsr> \<uvector> \<ubr>
RL11	DWa	??an	\<ucsr> \<uvector> \<ubr>

The definitions for the additional fields are:

\<tr>	Adapter TR number	decimal	1-15
\ 	Adapter br level	decimal	4-7
\<drive>	Massbus drive	decimal	0-7
\<ucsr>	Unibus CSR address	octal	760000-777776
\<uvector>	Unibus vector	octal	2-776
\<ubr>	Unibus BR level	decimal	4-7
\<EIA>	EIA terminal interface		
\<20MA>	20 mA terminal interface		

In the generic name:

"a" is a letter from A to Z.
"n" is a decimal number in the range 0-255.
"??" is a generic device name which may be any two letters.

```
DS> ATTACH DW780 SBI DW0 3 4   ! Attach the DW780.
DS> ATTACH DZ11 DW0 TTA        ! Attach the DZ11 TTA.
CSR? 760120                    ! The supervisor prompts
VECTOR? 320                    ! for information not
BR? 4                          ! supplied in the command
                               ! line.
DS>
```

<p align="center">Example 5-4   Attach Command</p>

Select Command

```
SELECT <generic-device-name>[:],-<CR>
[<generic-device-name>[:] . . . ] ! ALL<CR>
```

The operator must select each unit to be tested with the Select command, after attaching it. For each unit, supply the appropriate generic device name, as shown in Table 5-1. The Select command adds the specified device to the list of units to be tested. The command takes effect when the next diagnostic program is started.

```
DS> SELECT   TTA:
DS>
```

<p align="center">Example 5-5   Select Command</p>

Deselect Command

```
DESELECT <generic-device-name>[:] [,<generic-device-name>[:] -
. . .] ! ALL<CR>
```

Use the Deselect command to remove the name of one or more devices from the list of units to be tested.

```
DS> DESELECT   TTA:
DS> DESELECT   ALL
DS>
```

<p align="center">Example 5-6   Deselect Command</p>

Show Device Command

```
SHOW DEVICE  <generic-device-name>[:]-<CR>
[,<generic-device-name>[:] . . .]<CR>
```

The Show Device command causes the supervisor to display the characteristics of the specified devices on the operator's terminal. If you omit the device name, the supervisor will list the characteristics of all attached devices (Example 5-7).

Show Select Command

SHOW SELECT<CR>

The Show Select command causes the display of information in the
same format as the Show Device command. However, the information
is shown only for the devices that have been previously selected.

```
DS> SHOW DEVICE
_DW0     DW780             60006000    TR=3. BR=4. NUMBER=0.
_DMA     RK611     _DW0    6013FF20    CSR=00000777440(O) VECTOR=00000000210(O) BR=5.
_DMA0    RK07      _DMA    00000000
_TTA     DZ11      _DW0    6013E050    CSR=00000760120(O) VECTOR=00000000320(O) BR=4.

DS> SHOW SELECT

DS> SELECT TTA:
DS> SHOW SELECT
_TTA     DZ11      _DW0    6013E050    CSR=00000760120(O) VECTOR=00000000320(O) BR=4.
DS> DESELECT TTA:
DS> SHOW SELECT
DS>
```

Example 5-7   Show Device and Show Select Commands

Start Command

START   [/SECTION:<section-name>]-<CR>
        [/TEST:<first>[:<last>!/SUBTEST:<num>]]-<CR>
        [/PASSES:<count>]<CR>

The Start command causes the diagnostic supervisor to pass control
to the initialize routine in the diagnostic program in memory,
thus beginning program execution.

Each diagnostic program is organized in discrete tests. The tests
are grouped in sections, according to their functions, execution
times, and whether or not there is need for operator interaction.

If the Start command is given without switches, the program will
run the tests in the default section. In other words, the initial
setting for Section is DEFAULT. The supervisor calls only those
tests that have been designed by the diagnostic engineer to run in
the default section. Default section tests should not require
operator intervention.

The SECTION switch, if required, must be set up by the programmer
in the data structures section of the program (Chapters 6 and 7).
When a section is selected in conjunction with the Start command,
only the tests that it contains will be executed. Default section
tests do not require operator intervention.

The TEST switch is used in two distinctly different ways. If the
first and _last_ arguments are specified, the supervisor
sequentially passes control to tests _first_ through _last_,
inclusively. If the _first_ argument is combined with the SUBTEST
switch, program execution begins at the beginning of the _first_
test and terminates at the end of the subtest _num_. If the SUBTEST
switch is used in conjunction with the PASSES switch, the operator
is provided with a loop-on-subtest capability. In this case, only
the subtest named in the command line is executed, once looping
begins.

If the TEST switch is not specified, all tests within the named
section of the program are executed. In other words, the default
for TEST is TEST 1 through TEST n, where TEST n is the highest
numbered test in the section. If only the first argument is
specified with the TEST switch, the last argument is assumed by
default to be the highest numbered test within the program.

Tests are run only if they are included in the section named.

If the PASSES switch is not used, the default value is 1. Test
and pass numbers are decimal, the minimum value for passes is 1.
The maximum value is 0, which means infinity in this context.

For example:

DS> <u>START</u>	! Start execution of the ! diagnostic program in memory.
DS> <u>START/SEC:MANUAL</u>	! Start execution of the ! manual section of the program.
DS> <u>START/SEC:MANUAL/TEST:32:33</u>	! Run tests 32 and 33 if they are ! in the manual section. Some ! tests may not be executed ! unless the section is ! specified.
DS> <u>START/TEST:6:12</u>	! Run tests 6, 7, 8, 9, 10, ! 11, 12.
DS> <u>START/TEST:9/SUBTEST:5</u>	! Run test 9, subtests 1, 2, ! 3, 4, 5.
DS> <u>START/TEST:9</u>	! Run tests 9 through n, ! where n is the last test in ! the default section.
DS> <u>START/PASS:3</u>	! Run 3 passes of the ! default section.
DS> <u>START/TEST:9/SUBTEST:5/PASS:0</u>	! Execute test 9, subtests 1, 2, ! 3, 4, and then loop on subtest 5 ! indefinitely.

Example 5-8   Start Command

Run Command

```
RUN      <file-spec>[/SECTION:<section-name>]-<CR>
         [/TEST:<first>[:<last>!/SUBTEST:<num>]]-<CR>
         [/PASSES:<count>]<CR>
```

Run is equivalent to a Load and Start command sequence. The Run command switches are identical to those in the Start command.

For example:

DS> RUN ESTAA	! Load and run the local ! terminal diagnostic.
DS> RUN ESTAA/SEC:MANUAL	! Load the local terminal ! diagnostic and run the ! manual section.
DS> RUN ESTAA/SEC:MANUAL/TEST:32:33	! Load the local terminal ! diagnostic and run tests ! 32 and 33 in the manual ! section.
DS> RUN ESTAA/TEST:6:12	! Load the local terminal ! diagnostic and run tests ! 6, 7, 8, 9, 10, 11, 12.
DS> RUN ESTAA/TEST:9/SUBTEST:5	! Load the local terminal ! diagnostic and run test 9, ! subtests 1, 2, 3, 4, 5.
DS> RUN ESTAA/TEST:9	! Load the local terminal ! diagnostic and run tests 9 ! through n, where n is the ! last test in the default ! section.
DS> RUN ESTAA/PASS:3	! Load the local terminal ! diagnostic and run three ! passes.
DS> RUN ESTAA/TEST:9/SUBTEST:5/PASS:0	! Load the local terminal ! diagnostic, execute test 9, ! subtests 1, 2, 3, 4, and ! then loop on subtest 5 ! indefinitely.

Example 5-9   Run Command

Summary Command

SUMMARY<CR>

This command causes the execution of the program's summary report code section, which prints statistical reports. Note that this command is generally used only after running a pass of a diagnostic program. However, the summary command can be used at any time, and would be useful, for example, when the Disk Reliability Program is run. Type Control C first to return control to the command line interpreter (CLI). Then type SUMMARY to obtain a statistical report on the program. CONTINUE may be typed at this point, if the operator wishes to resume program execution.

Control C Command

^C<CR>

Normally Control C returns control from a diagnostic program to the CLI in the diagnostic supervisor. The supervisor then enters a command wait state and displays the DS> prompt on the operator's terminal. The operator may then issue any valid command. Control C is the only diagnostic supervisor command that may be issued while a program is running. When a diagnostic program is running in conversation mode, Control C returns control to a command interpreter within the program for the conversation mode.

Continue Command

CONTINUE<CR>

This command causes program execution to resume at the point at which the program was suspended. This command is used to proceed from a breakpoint, error halt, summary, or Control C situation.

The following example shows how Control C, Summary, and Continue can be used together to obtain statistics on the program being run and to then resume execution.

```
                      ...Program is running...

^C                                    ! Operator types Control C.
DS> SUMMARY                           ! supervisor prompt
                                      ! Operator requests
                                      ! statistical report.

                         Statistical
                           Report

DS> CONTINUE                          ! supervisor prompt
                                      ! Operator requests
                                      ! resumption of program.

                      ...Program is running...
```

Example 5-10   Use of Control C, Summary, and Continue Commands

Abort Command

ABORT<CR>

This command passes control to the program's cleanup code and then returns control to the supervisor, which enters a command wait state and displays the supervisor prompt, DS>. At this point the operator may issue any command except Continue. Example 5-11 shows how the Abort command can be used together with Control C and Summary.

```
                      ...Program is running...

^C                                    ! Operator types Control C.
DS> SUMMARY                           ! supervisor prompt
                                      ! Operator requests
                                      ! statistical report.

                         Statistical
                           Report

DS> ABORT                             ! supervisor prompt
                                      ! Operator requests program
                                      ! cleanup and termination.

DS>                                   ! supervisor prompt
```

Example 5-11   Use of Control C, Summary, and Abort Commands

5.3.2   Scripting
The scripting feature in the supervisor enables the computer
operator to run predefined sequences of diagnostic programs
automatically.   Supervisor commands normally solicited from the
operator's terminal are instead taken from a text file.

5.3.2.1 Scripting Command

@[load-device:[directory]]<file-spec><CR>

This command causes the supervisor to execute the commands that it
finds in the command file specified.   You should build the command
file with a text editor before starting the supervisor, and then
copy the command file on the diagnostic program load device.   When
you execute the command file from the supervisor, the supervisor
assumes that the load device for the command file is the device
from which the supervisor was loaded.   If the load device is
different, specify the device and the directory for the file
either with the scripting command or with a preceding Set Load
command.

Example 5-12 shows a typical command file.   Example 5-13 shows how
the file can be used.   Notice that in Example 5-13 the load device
is specified, but the file type and version are not specified.
When the operator does not supply the file type and version
number, the supervisor applies the defaults ".COM;0".

```
DS> ATTACH DW780 SBI DW0 3 4
DS> ATTACH DZ11 DW0 TTA 760120 320 4
DS> SELECT TTA:
DS> RUN ESDAA/PASS:3
```

Example 5-12   A Typical Command File

NOTE
The author of the command file must
supply the DS> at the beginning of each
line.

```
$ COPY CMD.COM DMA0[TEST]
$ RUN ESSAA
DS> @DBA0[TEST]CMD
```

Example 5-13   Execution of a Typical Command File

NOTE
The square brackets around the directory
name, [TEST], are necessary.

Diagnostic programs should not solicit information from the operator, except under unusual circumstances. Exceptions are manual intervention tests and volume verification failures for programs that write on disks. Responses to questions of this nature should come from the operator, not from a script. Therefore, script files contain only supervisor commands.

All of the $DS_ASKxxxx_x supervisor services (Chapter 8) will prompt the operator at the terminal regardless of the state of scripting. Synchronization between the script and the supervisor is not a problem, since each line in the script is a separate and complete supervisor command. The supervisor interprets each command exactly as if it had been typed on the operator's terminal.

5.3.2.2 @ Command Processing – The supervisor processes the @ command roughly as follows.

1. The supervisor aborts the current program if necessary.

2. A DS$LOAD command reads the whole script at once into a buffer. This prevents interaction between the unit under test and the load unit. Any interaction might cause incorrect interpretation of unit, controller, or channel status.

3. The supervisor initializes a pointer to the first line of the script.

4. The supervisor sets a flag to indicate that the next command is to be taken from the script.

5. As the supervisor processes the commands in the script, it displays the prompt and command text on the operator's terminal.

6. When the script has been exhausted, the supervisor types "@ <EOF>".

5.3.2.3 Buffer Allocation and Script Nesting – The supervisor dynamically allocates the memory buffer for script text and control and position information. Each script descriptor is linked to previous script descriptors. This allows you to nest scripts. The amount of memory available on a given VAX computer system limits the number of nesting levels possible.

You can invoke script nesting with an "@<file-spec>" command within a script. The supervisor processes commands from the second script file until it reaches the end of the script. The supervisor then releases the second script and resumes processing commands from the first script. If no previous script is left unprocessed, control returns to the operator's terminal.

5.3.2.4 Interrupting the Script - The operator may type Control C on the terminal to interrupt the script, if necessary. Control C causes the supervisor to suspend the script and stop the current program, if a program is running. The operator can issue any command while the script is suspended. However, if the operator wants to resume the script, eventually, by typing CONTINUE, the selection of commands is limited.

These commands can be followed by resumption of the program.

        Set
        Clear
        Examine
        Deposit
        Show
        Summary
        Next
        Continue

The following commands flush all scripts and return control to the command line interpreter in the supervisor:

        Attach
        Select
        Deselect
        Load
        Start
        Run
        Abort

In general, a command flushes scripts if it would be meaningless to continue the script after the command has been executed.

5.3.2.5 Command File Format - A command procedure must be a contiguous ASCII file created by VAX-11 RMS (record management services) on an ODS-1 or ODS-2 disk file structure. The file must be line oriented and records must not exceed 72 characters. You can create a command procedure file with any editor or with the VMS Create command. The supervisor treats all records as supervisor commands. Any legitimate supervisor command is valid in a script.

5.3.3 Execution Control Functions
The execution control functions allow the operator to alter the characteristics of the diagnostic programs and the diagnostic supervisor. These functions are implemented by command flags and event flags. The command flags are used to control the printing of error messages, ringing the bell, halting and looping of the program, and so on.

Set Flags Command

SET [FLAGS] <arg-list><CR>

This command results in the setting of the execution control flags specified by <u>arg-list</u>. No other flags are affected. <u>Arg-list</u> is a string of flag mnemonics from the following table, separated by commas.

HALT   Halt on error detection. When the program detects a failure and if this flag is set, the supervisor enters a command wait state after all error messages associated with the failure have been output. The operator may then continue, restart, or abort the program. This flag takes precedence over the Loop flag.

LOOP   Loop on error. When set, this flag causes the program to enter a predetermined scope loop on a test or subtest that detects a failure. Set the IE1 flag if you want to inhibit error messages. Looping will continue until the operator returns control to the supervisor by using the Control C command. The operator may then continue, clear the flag and continue, or abort the program.

BELL   Bell on error. When set, this flag will cause the supervisor to send a bell to the operator whenever the program detects a failure.

IE1   Inhibit error messages at level 1. When set, this flag suppresses all error messages, except those that are forced by the program or supervisor.

IE2   Inhibit error messages at level 2. When set, this flag suppresses basic and extended information concerning the failure. Only the header information message (first three lines) is output for each failure.

IE3   Inhibit error messages at level 3. When set, this flag suppresses extended information concerning the failure. The header and basic information messages are output for each failure.

IES   Inhibit summary report. When set, this flag suppresses statistical report messages.

QUICK   Quick verify. When set, this flag indicates to the program that the operator wants a quick verify mode of operation. The interpretation of this flag is program dependent. (Refer to Chapter 7, Paragraph 7.3.2.)

TRACE   Report the execution of each test. When set, this
                flag causes the supervisor to report the execution of
                each individual test within the program as the
                supervisor dispatches control to that test.

OPERATOR   Operator present. When set, this flag indicates to
                the supervisor that operator interaction is possible.
                When cleared, the supervisor and the program take
                appropriate actions to ensure that the test session
                continues without an operator. (Refer to Chapter 7,
                Paragraph 7.3.3.)

PROMPT   Display long dialogue. When set, this flag indicates
                to the supervisor that the operator wants to see the
                limits and defaults for all questions printed by the
                program.

ALL   All flags in this list.

Clear Flags Command

CLEAR [FLAGS] <arg-list><CR>

This command results in the clearing of the flags specified by
arg-list. No other flags are affected. Arg-list is a string of
flag mnemonics separated by commas. See the Set command for
supported arguments.

Set Flags Default Command

SET FLAGS DEFAULT<CR>

This command returns all flags to their initial default status.
The default flag settings are OPERATOR and PROMPT.

Show Flags Command

SHOW FLAGS<CR>

This command displays all the execution control flags and their
current status. The flags are displayed as two mnemonic lists;
one list is for those flags that are set, the other for those that
are clear.

The following example shows how the Set Flags, Clear Flags, and
Show Flags commands can be coordinated.

```
DS> SET FLAG TRACE                ! Set the TRACE flag.
DS> CLEAR FLAG QUICK              ! Clear the QUICK flag.
DS> SHOW FLAGS
CONTROL FLAGS SET: PROMPT, OPER, TRACE
CONTROL FLAGS CLEAR: QUICK, IES, IE3, IE2, IE1, BELL, LOOP, HALT

DS>
```

Example 5-14   Use of the Flag Control Commands

Set Event Flags Command

SET EVENT [FLAGS] <arg-list> ! ALL<CR>

This command results in the setting of the event flags specified
by arg-list. No other event flags are affected. Arg-list is a
string of flag numbers in the range of 1-23, separated by commas.
ALL may be specified instead of arg-list.

Event flags are status posting bits maintained by VMS and the
supervisor. Diagnostic programs can use event flags to perform a
variety of signaling functions, including communication with the
operator.

The diagnostic engineer should follow three event flag conventions
in particular.

>   a.   Event Flag Ø (EVØ) is reserved for the supervisor in
>   connection with QIO functions.
>
>   b.   Event Flag 1 (EV1) enables (when set) or disables (when
>   clear) error logging under VMS.
>
>   c.   Event Flag 2 (EV2) enables (when set) or disables (when
>   clear) retries under VMS.

Clear Event Flags Command

CLEAR EVENT [FLAGS] <arg-list> ! ALL<CR>

This command results in the clearing of the event flags specified
by arg-list. No other event flags are affected. Arg-list is a
string of flag numbers in the range of 1-23, separated by commas.
An optional ALL may be specified instead of arg-list.

Show Event Flags Command

SHOW EVENT [FLAGS]<CR>

This command causes the supervisor to display a list of the event
flags that are currently set.

Example 5-15 shows how the Set Event Flags, Clear Event Flags, and
Show Event Flags commands can be coordinated.

```
DS> SET EVENT FLAGS 1, 9, 15
DS> CLEAR EVENT FLAGS 2, 6
DS> SHOW EVENT FLAGS
EVENT FLAGS SET: 15, 9, 1
DS>
```

Example 5-15   Event Flags Control Commands

5.4   SUPERVISOR FUNCTIONAL DESCRIPTION

Most functions of the diagnostic supervisor fall into two categories: command line interpreter (CLI) and program interface (PGI). Together these categories of functions form the framework within which level 2, 2R, and 3 diagnostic programs must be executed. The CLI forms the interface between the operator's terminal, the supervisor, and the program to be run. The program interface handles communication between the diagnostic program and the supervisor. Figure 5-1 shows the relationship of the CLI and the PGI to the operator's terminal and the device under test.

The CLI implements the supervisor commands listed previously, together with the supervisor debug commands. The CLI displays a prompt symbol, DS>, when it is waiting for a command from the operator. When the operator types a command, a parser in the CLI interprets it and dispatches control to an appropriate action routine.

Depending on the command, the called routines will perform the desired function and then pass control back to the CLI or to the diagnostic program. For instance, in response to a SET FLAG IE2 command, the parser will call a routine that sets the flag. Control then returns to the CLI, which causes the display of another prompt on the operator's terminal. In response to a START command, however, the CLI calls the dispatch routine, which in turn initiates diagnostic program execution.
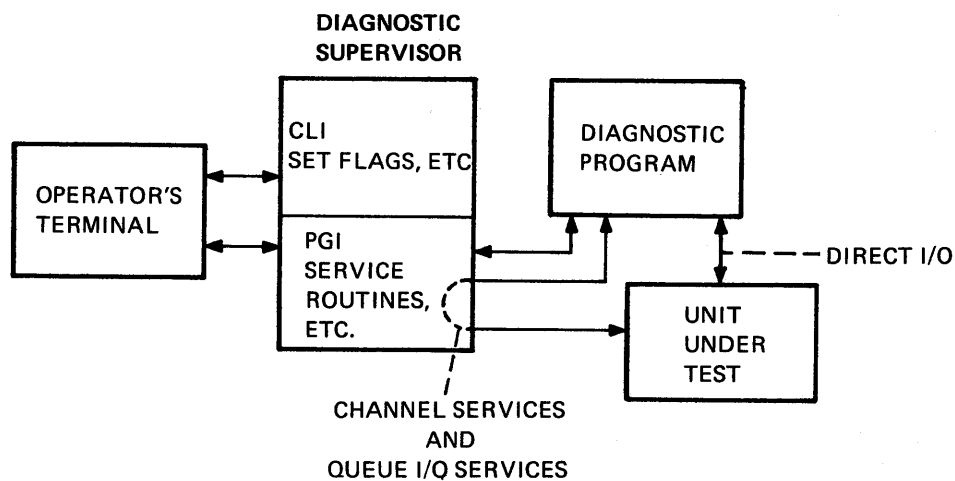
Figure 5-1   Diagnostic Supervisor Functional Block Diagram

The program interface implements the program control services, message handling services, memory management services, channel services, and I/O services. The dispatch routine calls the various routines of the diagnostic program in the proper sequence (initialization, test 1, test 2,...test n, and cleanup). The routines that make up the diagnostic program, in turn, call different service routines in the PGI as needed.

Figure 5-2 is a simplified flowchart showing how the CLI and the PGI portions of the supervisor interface with a diagnostic program.

This figure highlights two features of the diagnostic system in particular. First, control begins and ends with the DS> prompt in the CLI portion of the supervisor. Second, the diagnostic program is not an independent program. It consists of a series of routines (initialization, test 1, etc.) that are called by the dispatch routine in the supervisor.

Figure 5-3 shows the diagnostic system memory allocation. The diagnostic program space always starts at virtual address 200. The supervisor space starts at virtual address 10000. The APT mailbox extends from FC00 to FFFF. The spaces between the diagnostic program and the APT mailbox and above the supervisor are available for use as memory buffers.
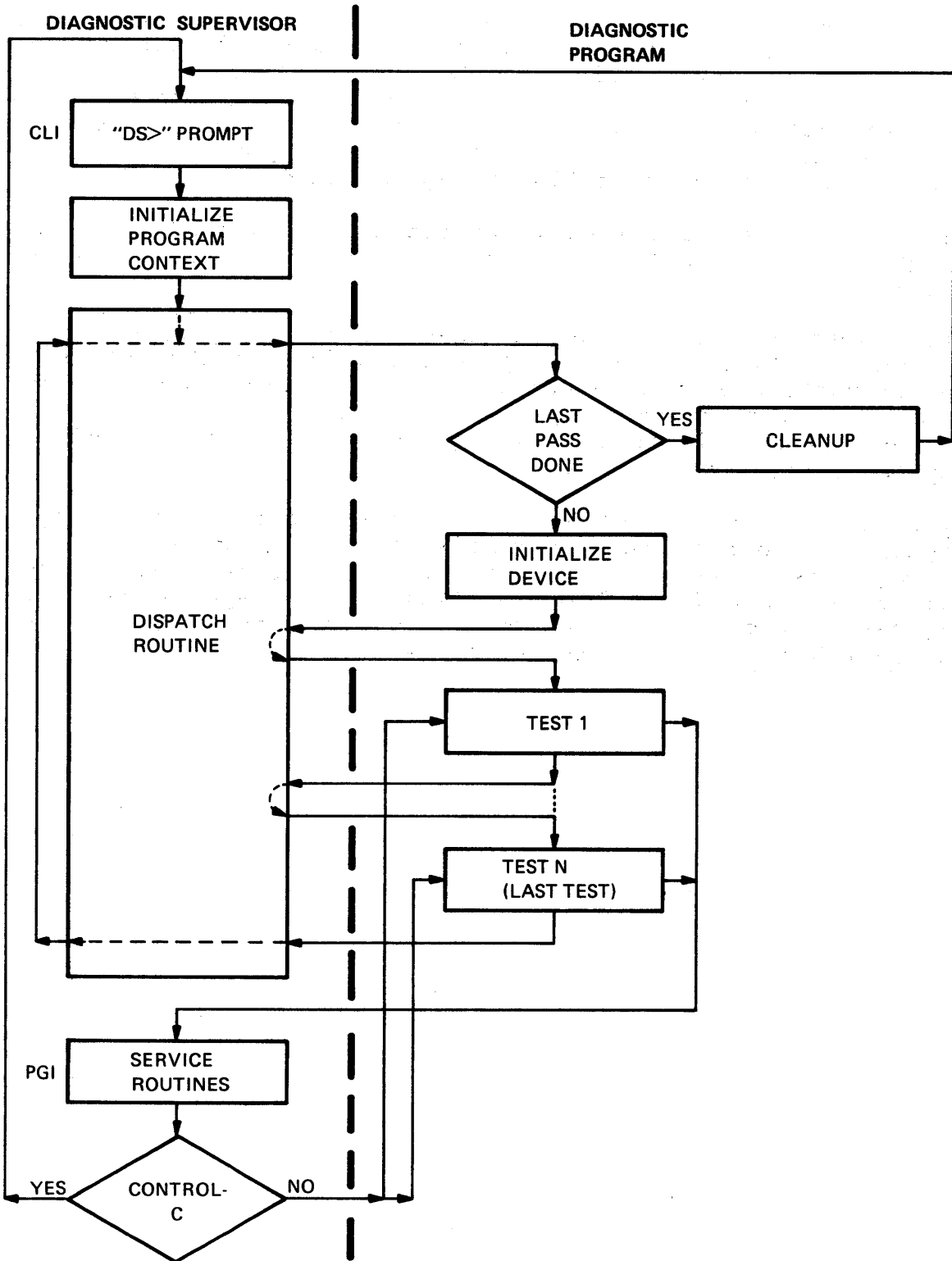
Figure 5-2   Diagnostic Supervisor and Diagnostic
Program Interaction

```
                              ┌──────────────┐ 0
                              │              │
                              │              │
                              ├──────────────┤ 200
                              │ DIAGNOSTIC   │ │
                              │ PROGRAM      │ │
                              │              │ │
                              │              │ │
                              │              │ ▼
                              ├ ─ ─ ─ ─ ─ ─ ─┤
                              │ SPACE AVAILABLE│
                              │ FOR MEMORY   │
                              │ BUFFERS      │ FC00
                              ├──────────────┤ │
                              │ APT          │ │
                              │ MAILBOX      │ ▼
                              ├──────────────┤ 10000
                              │ DIAGNOSTIC   │ │
                              │ SUPERVISOR   │ │
                              │              │ │
                              │              │ ▼
                              ├ ─ ─ ─ ─ ─ ─ ─┤
                              │ SPACE AVAILABLE│
                              │ FOR MEMORY   │
                              │ BUFFERS      │
                              └──────────────┘
```
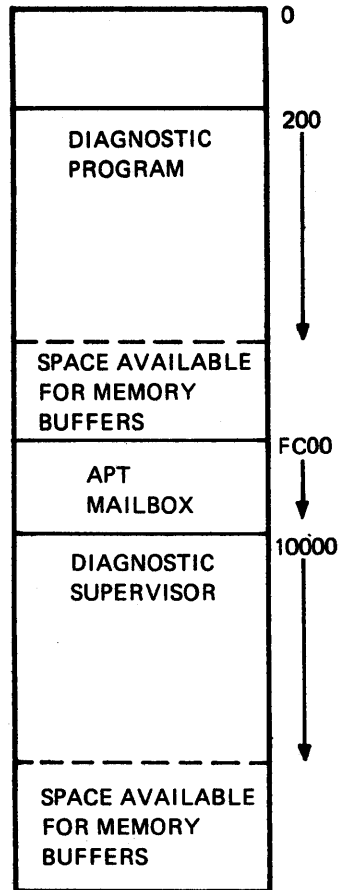
TK-1737

Figure 5-3   Diagnostic System Memory Allocation

6.1 OVERALL PROGRAM STRUCTURE

Like all VAX software, diagnostic programs are organized in modules, program sections, routines, subroutines, and data structures. Once a diagnostic program has been developed, and the source modules have been assembled and linked, the listing should have the following format.

Header Module

Section	Function
Nonexecutable Code	
Program Header Section	Module preface.
Declaration Section	
Include Files	Macro library declarations.
Program Header Data Block	Parameters that define programs to supervisor.
Dispatch Table	Table of test addresses.
Program Equates	Area for macro and symbol definitions.
Program Data	Area for data used by more than one test.
Device Register Contents Table	Working storage.
Statistics Table	Statistics to be used in conjunction with the summary report routine.
Character String Type Data	ASCII strings.
Program Section Names	Information for supervisor.
Device Mnemonics List	ASCII strings.
Names of Device Registers and their Bits	ASCII strings.
ASCII Messages	Message texts for error reports.
Executable Code	
Initialization Routine	Initialize device.
Cleanup Routine	Restore device to original condition.
Summary Report Routine	Print statistics.

Subroutine and Test Modules

Section	Function
Nonexecutable Code	
Global Subroutine Module	

Module Header Section
Declaration Section
  Equated Symbols
  Own Storage	

Executable Code	
Program Global Subroutines	Common services.
Test Modules	
Nonexecutable Code	

Module Header Section
Declarations Section
  Equated Symbols
  Own Storage	

Executable Code	
Test 1 -- description Test 1 -- code Test n -- description Test n -- code	

Notice that the tests are placed at the end of the listing. However, tests and subtests form the heart of any diagnostic program. In a functional level program, each test exercises or checks a major function of the device under test. Subtests exercise or check separable parts of the function to be tested. In a repair level program, tests check major logic areas, while subtests check subordinate logic areas. The sequence of tests should be designed and arranged in a building block structure. A minimum set of logic or functions should be tested first. After basic operations have been verified, a larger and more complex set of logic or functions should be tested, using the previously tested block or a base area that is known to be good. This strategy enables the programmer to define errors as precisely as possible.

You should group the tests into sections, according to their functions and run-time requirements. Each diagnostic program should have a default section, which contains the basic tests and will run without operator intervention. Examples of other section types follow:

    manual intervention section
    loopback section
    media test section

The remainder of the diagnostic program provides support for the tests. The initialization and cleanup code sections form an envelope around test execution. The global subroutines can be called from the tests to perform common services such as error reporting. The data structures at the beginning of the program provide supervisor interface information, test patterns, message texts, symbol definitions, other test-related information, and working storage areas in memory.

Files containing templates for the header module and a test module, the diagnostic macro library, and other diagnostic program development tools are available from VAX Diagnostic Engineering. In developing a program, the programmer should first complete a project plan, a functional specification, and a design specification (refer to Chapter 4 of this manual). He should then build on the header and test module templates to code the program.

Diagnostic programs that follow this standard, modular format are easy to develop, debug, and maintain. The format has proven to be reliable. It provides a good interface with the supervisor, and it conforms to standard DIGITAL software methodology.

6.2   PROGRAM HEADER MODULE

6.2.1   Program Header Section (Module Preface)
The program header section is the first item in the first program module. This section defines the functions and the uses of the program for the user. Except for the first two lines, each line begins with a semicolon to indicate that it is a comment, not data or executable code. The program header section consists of the following items in the order given.

    a.   A .TITLE statement specifying the program name. The title is a symbol of up to 15 characters in length. A phrase indicating the program function should follow on the same line. The TITLE statement is always the first line of the file.

    b.   An .IDENT statement indicating the program's current version number. The IDENT statement is always the second line of the file.

c.   Standard DIGITAL legal notices. These should be typed in capital letters in the source program.

d.   A facility statement: VAX Diagnostic System.

e.   A short functional description of the program. A more extensive program description should be given in a separate program abstract that should be attached to the listing as a preface.

f.   Environment statement: VAX Diagnostic Supervisor.

g.   The author, program date, and program version number.

h.   A detailed current edit history. This item specifies the versions, the modifier, and the last date of each version. This item also lists the specific changes made between base levels (during production) or releases, providing a short, functional description of each problem and its solution, as well as appropriate reference information, such as SPR number(s), etc. The comments include the full name of the person responsible for each version. If several people modify the module, the initials of the others appear in each edit line.

Example 6-1 shows a program header module preface.

```
 1                    .TITLE   ZZ_ABCDE   DEVICE-X REPAIR LEVEL DIAGNOSTIC
 2                    .IDENT   /V1.1-2/
 3
 4
 5
 6
 7      ;
 8      ; COPYRIGHT (C) 1978, 1979
 9      ; DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASSACHUSETTS 01754
10      ;
11      ; THIS SOFTWARE IS FURNISHED UNDER A LICENSE FOR USE ONLY ON A
12      ; SINGLE COMPUTER SYSTEM AND MAY BE COPIED ONLY WITH THE
13      ; INCLUSION OF THE ABOVE COPYRIGHT NOTICE.  THIS SOFTWARE, OR
14      ; ANY OTHER COPIES THEREOF, MAY NOT BE PROVIDED OR OTHERWISE
15      ; MADE AVAILABLE TO ANY OTHER PERSON EXCEPT FOR USE ON SUCH
16      ; SYSTEM AND TO ONE WHO AGREES TO THESE LICENSE TERMS.  TITLE
17      ; TO AND OWNERSHIP OF THE SOFTWARE SHALL AT ALL TIMES REMAIN
18      ; IN DEC.
19      ;
20      ; THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE
21      ; WITHOUT NOTICE AND SHOULD NOT BE CONSTRUED AS A COMMITMENT
22      ; BY DIGITAL EQUIPMENT CORPORATION.
23      ;
24      ; DEC ASSUMES NO RESPONSIBILITY FOR THE USE OR RELIABILITY OF
25      ; ITS SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DEC.
26      ;
27      ;
28      ;++
29      ; Facility:   VAX-11 Diagnostic System.
30      ;
31      ; Abstract:       The program consists of two subtests in one
        ; test. The subtests are executed via commands by the user.
33      ; The test routine uses the command parser in conjunction with
34      ; a command syntax tree.
35      ;
36      ; Environment: VAX Diagnostic Supervisor.
37      ;
38      ; Author:     Ted Bear      1-NOV-78 Version V1.0.
39      ;
40      ; Modified By: Jim Skunk    1-DEC-78 Added Quick Verify. New
41      ; Version V1.1.
42      ;--
```

Example 6-1   A Program Header Module Preface

6.2.2   Module Declarations Section

The module declarations section contains the data structures for the entire program. The following items should be given.

- The Include Files section names the diagnostic macro library (DIAG) and any other libraries referenced by the program.

- Definition of local macros (user-defined macros).

- Equated symbols. This section consists of the $DS_BGNMOD macro and a set of other macros that define the keyword symbols for macros in the diagnostic macro library. You may also equate local symbols to literal values.

Example 6-2 shows a typical declaration section for the header module.

```
39          .SBTTL Declarations
40   ;
41   ;  Include Files:
42   ;
43   .Library \DIAG\              ; VAX Diagnostic Macro Library.
44
45   ;
46   ;  Macros:
47   ;
48
49   ;
50   ;  Equated Symbols:
51   ;
52          $DS_BGNMOD     <SEP_REPAIR>
     ;DIAGNOSTIC MACRO LIBRARY V4.Ø3 "DIAG.MLB (516)"
53          $DS_BITDEF     GLOBAL ; mnemonic bit definitions
54          $DS_CHDEF      GLOBAL ; channel service symbols
55          $DS_DSADEF     GLOBAL ; offsets in supervisor/APT mailbox
56          $DS_DSSDEF     GLOBAL ; supervisor service entry vectors
57          $DS_ERRDEF     GLOBAL ; error call parameter offsets
58          $DS_PARDEF     GLOBAL ; parameter code symbols
59          $DS_CLIDEF            ; syntax tree symbols and literals
60          $DS_PRINTX_DEF       ; print call parameter offsets


61   NOP  =  Ø                    ; no action
62   SUB1 =  1                    ; select subroutine #1
63   SUB2 =  2                    ; select subroutine #2
```

Example 6-2   Declarations Section

Notice that the line between lines 52 and 53 is not numbered. The assembler expands the $DS_BGNMOD mode to generate the line that follows.

- The Program Header Data Block program section contains program parameters that allow the diagnostic supervisor to control the program. The diagnostic supervisor looks for this information beginning at virtual address 200 (hex). The $DS_HEADER macro, together with appropriate arguments, creates the header data block. Example 6-3 shows a typical program header data block after module assembly.

```
64   .SBTTL Program Header Data Block.
65 ;++
66 ; Functional Description
67 ;
68 ; The program header data block contains the parameters
69 ; which allow the diagnostic supervisor to control the
70 ; program.  The diagnostic supervisor looks for the header
71 ; information beginning at virtual address 200 (hex).
72 ;
73 ;--
74   $DS_HEADER    <DEVICE-X REPAIR LEVEL DIAGNOSTIC>, REV=01, DEPO=0, NUNIT=4
                   .SAVE
                   .PSECT $HEADER, PAGE, NOEXE, NOWRT
     L$L_HEADLENGTH:                          .LONG A_HEADEND-. ;LENGTH OF THE HEADE
                                                                ;DATA BLOCK.
     L$L_ENVIRON:    .LONG $ENV            ;PROGRAM ENVIRONMENT.
     L$A_NAME:       .ADDRESS T_NAME       ;PROGRAM NAME TEXT ADDRESS.
     L$L_REV:        .LONG 01              ;PROGRAM REVISION LEVEL.
     L$L_UPDATE:     .LONG 0               ;DIAGNOSTIC ENGINEERING PATCH ORDER.
     L$A_LASTAD:     .ADDRESS LASTAD       ;FIRST FREE LOCATION AFTER PROGRAM.
     L$A_DTP:        .ADDRESS DISPATCH     ;TEST DISPATCH TABLE POINTER.
     L$A_DEVP:       .ADDRESS AL_DEVTYP    ;DEVICE TYPE LIST POINTER.
     L$L_UNIT:       .LONG 1               ;NUMBER OF UNITS THAT CAN BE TESTED.
     L$A_DREG:       .ADDRESS DEV_REG      ;DEVICE REGISTER CONTENTS TABLE POINTER
                     .LONG 0[5]
     L$A_ICP:        .ADDRESS INITIALIZE   ;INITIALIZATION CODE POINTER.
     L$A_CCP:        .ADDRESS CLEAN-UP     ;CLEAN-UP CODE POINTER.
     L$A-REPP:       .ADDRESS SUMMARY      ;SUMMARY REPORT CODE POINTER.
     L$A_STATAB:     .ADDRESS 0            ;STATISTIC TABLE POINTER.
     L$L_ERRTYP:     .LONG 0               ;# OF TYPES OS $ERRSOFT AND $ERRHARD.
     L$A-SECNAM:     .ADDRESS SECTION      ;LIST OF SECTION NAME ADDRESSES.
     L$A_TSTCNT:     .ADDRESS L-TSTCNT     ;POINTER TO NUMBER OF TESTS.
     A_HEADEND:
     T-NAME:         .ASCIC \DEVICE-X REPAIR LEVEL DIAGNOSTIC\
                     .PSECT _LAST, PAGE
     LASTAD:         .PSECT $TSTCNT, NOEXE, NOWRT, OVR, LONG
     L-TSTCNT:       .RESTORE
```

Example 6-3   Program Header Data Block

Notice that the line numbers stop at 74; the Program Header Data Block is created by the $DS-HEADER macro at assembly time.

- The Dispatch program section contains the $DS_DISPATCH macro. This macro generates a new psect, a beginning tag, address label, and ending tag for the dispatch table. The actual entries in the dispatch table are created at link time by use of the $DS_BGNTEST macro at the beginning of each test. When fully put together, the dispatch table consists of a list of addresses pointing to the beginning of each test. The supervisor accesses this table when sequencing through the tests in the program. Example 6-4 shows how the $DS_DISPATCH macro is used to create the dispatch table psect.

```
76          .SBTTL Dispatch Table.
77    ;++
78    ;  Functional Description:
79    ;
80    ;     The dispatch table is a collection of information
81    ;     describing each test and grouped together into a
82    ;     contiguous list by the linker.  This is done by defining
83    ;     a psect called dispatch.
84    ;
85    ;--
86
87          $DS_DISPATCH
                    .SAVE
                    .PSECT DISPATCH, LONG, NOWRT, NOEXE
      DISPATCH:
                    .PSECT DISPATCH_X, LONG, NOWRT, NOEXE
                    .LONG 0[6]

                    .RESTORE
```

Example 6-4   Dispatch Table Psect

- The basic hardware P-table entry are provided by the supervisor. One entry is dynamically built for each operator-selected device.

   Each entry contains nine items. P-table entries are located by using the $DS_GPHARD supervisor service call; and each of the items can be accessed with symbolic offsets. The symbolic offsets are defined by either the $DS_BGNHARD or $DS_HPODEF macro statement. Example 6-5 shows the P-table format. Table 6-1 shows the symbolic offsets.

device name quadword descriptor	
drive	size
ASCII device name including a "_"; maximum of 11 characters in device name	
device address	
direct virtual address	
address of P-table for link to the CPU	
ASCII	Vector
device type; maximum of 11 characters	
device dependent information	

Example 6-5   P-Table Format

Table 6-1   P-Table Symbolic Offsets

Symbol	Function
HP$Q_DEVICE	Quadword descriptor of device name
HP$W_SIZE	Total size of P-table
HP$B_DRIVE	Unit number
HP$T_DEVICE	ASCII device name with leading "_", max length = 11 characters
HP$A_DEVICE	Device address for this UUT
HP$A_DVA	Address used to directly address another UUT through this device
HP$A_LINK	Address of P-table for the device linking this to the CPU
HP$W_VECTOR	Primary interrupt vector for device
HP$T_TYPE	ASCIC hardware type, max length = 11 characters

●   The Program Global Data section contains all dynamically modified data. Address pointers and buffers should be placed here.

●   A device register contents table should be set up to contain information obtained from the I/O system services. The $DS_BGNREG and $DS_ENDREG macros should be used to define the table boundaries.

●   A statistics table may be set up for the tabulation of hardware and software errors. A separate table should be created for each device under test. The $DS_BGNSTAT and $DS_ENDSTAT macros define the boundaries of the table. The summary routine accesses this table when printing the statistical report.

Example 6-6 shows a typical program global data section.

```
120        .SBTTL Program Global Data Section.
121        .PSECT DATA, LONG
122   ;++
123   ; Functional Description:
124   ;
125   ;
126   ;
127   ;
128   ;--
129
130   ;++
131   ;     This section contains global addresses of device
132   ;     registers.
133   ;--
134
135   DRWC::          .ADDRESS 0      ; word count register (device
136                                   ; base address)
137   DRBA::          .ADDRESS 0      ; buffer address register
138   DRST::          .ADDRESS 0      ; command and status register
139   DRDB::          .ADDRESS 0      ; data buffer register
140
141   HDW_PTBL::
142   PHYSICAL_ADR::  .ADDRESS 0
143                   .ADDRESS 0
144   VIRTUAL_ADR::   .ADDRESS 0
145                   .ADDRESS 0
146
147   ;--
148   ; Device Register Contents Table.
149   ;--
150       $DS_BGNREG
      DEV_REG:
151
152   REG_BUF:        .BLKL 40
```

```
153        $DS_ENDREG
154   ;++
155   ;Statistics Table.
156   ;--
157        $DS-BGNSTAT                 ; statistical data supplied
      STATISTIC:                       ; by the summary routine
158        $DS_ENDSTAT
159
160   ;*** Other global data ***
161
162   CVTREG_CHARBUF:: .BLKB 132       ; memory buffers
163   DIOMEM::          .WORD ^0160000
164   INTER_FLAG::      .LONG 0
165   LOG_UNIT::        .LONG 0
166   STATUS1::         .LONG 0
167   STATUS2::         .WORD 0
168   STATUS_RVR::      .WORD 0
169   TEMP::            .WORD 0
170   UNIT::            .LONG 0
```

Example 6-6   Global Data Section

- The Program Text section contains all common character
  string type data entries. Program section names, device
  types, device registers, and register bit mnemonics are
  listed here. Example 6-7 shows a typical program text
  section.

```
171        .SBTTL Program Text Section.
172   ;++
173   ; Functional Description:
174   ;
175   ;    This section contains all the common character string
176   ;    type data entries.
177   ;
178   ;--
179
180   ;++
181   ; Program Section Names:
182   ;--
183        $DS_SECTION <QUICK>
      S_DEFAULT:
                        .ASCIC \DEFAULT\

      S_DRB:
                        .ASCIC \QUICK\

      SECTION:
                        .LONG 2
                        .LONG S_DEFAULT
                        .LONG S_QUICK
184
185   ;++
186   ; Device Mnemonics List.
187   ;--
```

```
188
189   T_DEVICE
190         $DS_DEVTYP   <DR11C>,< >
191
192   ;++
193   ; Device Register Bit Mnemonics
194   ;
195
196   CSR_BIT_DEF::     .ASCIC      \ERR,NEX,ATTN,MAINT,DSTATA,\-
197                                 \DSTATB,DSTATC,CYCLE,READY,\-
198                                 \IE,XBA17,XBA16,FNCT3,FNCT2,\-
199                                 \FNCT1,GO\
200   REG_BIT_DEF::     .ASCIC      \BIT15,BIT14,BIT13,BIT12,\-
201                                 \BIT11,BIT10,BIT9,BIT8,BIT7,\-
202                                 \BIT6,BIT5,BIT4,BIT3,BIT2,\-
203                                 \BIT1,BIT0\
```

Example 6-7   Program Text Section

- Formatted ASCII output statements. Messages to the operator, such as instructions and questions, should be listed here.

- Error report statements. ASCII texts for all error messages and error message components should be listed here.

Example 6-8 shows some typical messages.

```
203  ;++
204  ;
205  ;
206  ;
207  ; Error Report Statements.  ASCII texts of error messages
208  ;
209  ;--
210
211  MSG1_PTBLE_FAIL::   .ASCIC    \FAILED TO READ P_TABLE -ABORTING\

212  MSG2_UBA_FAIL::     .ASCIC    \FAILED TO INITIALIZE UBA\-
213                                \ - ABORTING\

214  MSG3_UNI_FAIL::     .ASCIC    \FAILED TO INITIALIZE UNIBUS\-
215                                \ - ABORTING\

216  MSG4_CLEAR_FAIL::   .ASCIC    \FAILED TO CLEAR UBA STATUS\-
217                                \ - ABORTING\

218  MSG5_INTER_FAIL::   .ASCIC    \UNEXPECTED DW780 INTERRUPT\-
219                                \ - ABORTING\

220  MSG6_INTER_UNKN::   .ASCIC    \UNKNOWN INTERRUPT ENCOUNTERED\-
```

```
221                                      \ - ABORTING\

222 MSG7_STAT_FAIL::    .ASCIC    \FAILED WHILE CHECKING UBA\-
223                                      \STATUS - ABORTING\

224 MSG8_ENINT_FAIL::   .ASCIC    \CHANNEL SERVICES INTERRUPT\-
225                                      \ENABLE FAIL - ABORTING\

226 MSG9_DSINT_FAIL::   .ASCIC    \CHANNEL SERVICES INTERRUPT\-
227                                      \DISABLE FAIL - ABORTING\

228 MSG10_BRLV_FAIL::   .ASCIC    \EXPECTED/RECEIVED BR LEVELS\-
229                                      \DON'T MATCH\
```

Example 6-8   Error Report Statement

6.2.3   Initialization Routine

The initialization routine is the first part of the diagnostic program to be called by the dispatch routine in the supervisor. The routine initializes the device or devices to be tested. It then sets up conditions in the CPU and the rest of the computer system that are necessary for diagnostic program execution such as making a channel assignment for each unit to be tested.

The initialization routine is executed at the beginning of each execution of the test sequence. It checks to determine whether the last pass to be executed has been completed. Following the end of the initialization routine, control returns to the dispatch routine in the supervisor, which calls the next appropriate routine (e.g., test 1).

Print routines should not be called from the initialization routine except on the detection of a device fatal or system fatal error. If a system fatal error is detected, the $DS_ERRSYS_x macro should be used. If a device fatal error is detected, the $DS_ERRDEV_x macro should be used.

A program that is designed to test more than one unit may test one unit at a time (serial testing), or it may test several units at a time (parallel testing). The initialization code reflects some of the differences between these two designs.

Example 6-9 shows a program design language (PDL1) description of an initialization routine for a program that performs parallel testing.

```
ROUTINE INITIALIZATION
:   IF PASS 0
:   :   THEN
:   :   :   !****PROGRAM INITIALIZATION****
:   :   :   SET LUN=0
:   :   :   REPEAT                          ! For each unit
:   :   :   :   :   CALL $DS_GPHARD         ! get hardware parameter
```

```
:  :  :  :  :     CALL $DS_ASSIGN              ! assign I/O channel.
:  :  :  :  :     INCREMENT LUN
:  :  :  :     UNTIL LUN INVALID              ! last unit
:  :  :     END REPEAT
:  :  ELSE
:  :  !****END OF PASS HOUSEKEEPING****
:  :  :  UPDATE STATISTICS TABLE
:  :  :  CALL $DS_ENDPASS
:  ENDIF
:  :  !****EVERY PASS INITIALIZATION****
:  CLEAR BUFFERS
:  CLEAR COUNTERS
END ROUTINE INITIALIZE
```

Example 6-9   Initialization Routine for Parallel
Device Testing

On pass Ø (program startup) the routine fetches the P-table
address and assigns an I/O channel for each device to be tested.
On subsequent executions, the initialization routine performs end
of pass housekeeping functions.   It also calls the end pass
supervisor service with $DS_ENDPASS_X.   If no more passes are to
be executed, the end pass service terminates program execution by
calling the summary and cleanup routines.

If the last pass has not been completed, the routine performs some
every-pass initialization functions, clearing buffers and
counters.

Example 6-1Ø shows a PDL1 description of an initialization routine
for a program that performs serial testing.   On pass Ø the routine
allocates buffers and sets the logical unit number to Ø, to test
the first unit.   On subsequent executions of the routine, the code
increments the logical unit number.   Each time through the
routine, the code selects a higher numbered unit.

```
ROUTINE INITIALIZE
:  IF PASS Ø
:  :  THEN
:  :  :  !****PROGRAM INITIALIZATION
:  :  :  ALLOCATE BUFFERS
:  :  :  SET LUN=Ø
:  :  ELSE
:  :  :  INCREMENT LUN
:  :  :  IF LAST UNIT DONE
:  :  :  :  THEN
:  :  :  :  :  !****END OF PASS HOUSEKEEPING****
:  :  :  :  :  CALL $DS_ENDPASS
:  :  :  :  :  SET LUN=Ø̄
:  :  :  END IF
:  ENDIF
:  !****TEST SEQUENCE INITIALIZATION****
```

```
:   CALL $DS_GPHARD
:   ASSIGN CHANNEL
:   CLEAR BUFFERS
:   CLEAR COUNTERS
END ROUTINE INITIALIZE
```

Example 6-10   Initialization Routine for Serial
Device Testing

If all the units selected by the operator have been tested, the routine performs end of pass housekeeping functions and calls the end pass supervisor service.   If the last pass has not been completed, the code sets the logical unit number to 0, beginning another pass.

The test sequence initialization functions follow the housekeeping functions.   The routine performs these functions with each execution except at program termination.

6.2.4   Cleanup Routine
The dispatch routine in the supervisor calls this routine following execution of the last test in the last pass of the program sequence. The dispatch routine will also call cleanup any time the program is terminated abnormally. The cleanup routine should force initialization of the device under test without error checking. The routine should stop any data transfers in progress, release memory buffers, and cancel timers. If the program is a level 2 diagnostic program, assigned channels should be deassigned to return control of the device or devices under test to the operating system. In addition, the cleanup routine should be set up to handle machine checks, in case the device under test is powered down. Example 6-11 is a PDL1 description of a typical cleanup routine.

```
ROUTINE CLEANUP
:   STOP I/O IN PROGRESS
:   IF MEMORY PAGES IN USE NOT EQUAL TO ZERO
:   :   THEN
:   :   :   CALL $DS_RELBUF                      ! release buffers.
:   ENDIF
:   CALL $CANTIM                                 ! cancel timers.
:   SET LUN = 0
:   REPEAT
:   :   :   CALL $DASSGN                         ! deassign I/O channel.
:   :   :   INCR LUN
:   :   UNTIL LUN GREATER THAN OR EQUAL TO TEST UNITS
:   :                                            ! TEST_UNITS  =  #  of
:   :                                            ! units under test.
:   END REPEAT
END ROUTINE CLEANUP
```

Example 6-11   Typical Design Specification for a Level 2
Program Cleanup Routine

6.2.5   Summary Routine

The summary report routine is optional. In conjunction with the statistics table it provides the operator with information describing program activity. The summary report is unnecessary in programs that execute in a few seconds. But it is very useful in programs with long execution times, such as a magnetic media reliability program. When a summary report routine is provided, the operator can interrupt program execution at any time, obtain a summary, and then continue execution or abort. Example 6-12 is a PDL1 description of a typical summary routine.

```
ROUTINE SUMMARY
:     SET TEST_UNITS = # OF UNITS
:     SET LUN = 0
:     REPEAT
:     :     IF STATISTICS ARE VALID
:     :     :   THEN
:     :     :     :    PRINT SUMMARY HEADER MESSAGE
:     :     :     :    PRINT TOTAL TRANSFERS
:     :     :     :    PRINT TOTAL READS
:     :     :     :    PRINT TOTAL WRITES
:     :     :     :    PRINT TOTAL ERRORS
:     :     :     :    PRINT TOTAL READ ERRORS
:     :     :     :    PRINT TOTAL WRITE ERRORS
:     :     :     :    INCR LUN
:     :     :   ELSE
:     :     :     :    INCR LUN
:     :     ENDIF
:     :     IF LUN GREATER THAN OR EQUAL TO TEST_UNITS
:     :     :   THEN
:     :     :     :    RETURN
:     :     ENDIF
:     END REPEAT
END ROUTINE SUMMARY
```

Example 6-12   Typical Design Specification for a Summary
Report Routine

This routine prints statistics on each device under test, if the statistics are valid.

6.2.6   Initialization, Cleanup, and Summary Documentation

The initialization, cleanup, and summary routines should be fully documented. The documentation consists of a routine preface, block comments for each logical block of code, and line comments. The routine preface should be organized a follows:

● A .SBTTL statement specifying the name of the routine.

- A functional description of the routine.

- A list of the routine calling sequence.

- A list of the routine input and output parameters (normally none).

- A list of implicit inputs, outputs, and side effects (normally none).

Refer to Chapter 11 for details. Each routine should have one entry point and one exit point, so that it will behave in the expected manner under all circumstances. An entry point is a label that points to the first location in a routine. The first location contains a register save mask that will save the contents of the registers used by the routine. An exit point is a label pointing to a return instruction (RET).

6.3 GLOBAL SUBROUTINES AND THE GLOBAL SUBROUTINE MODULE

Normally, global subroutines should be grouped in a separate module. However, if the program is small and the global subroutines are few and short, they may be included in the header module. Global subroutines should be used to perform functions that are performed two or more times in the test routines. The procedure performed should be general enough to be widely applicable.

Like the header module, the global subroutines module should contain a module preface and an equated symbol section. Each global subroutine, like the initialization routine, should include a routine preface and be fully documented with block comments and line comments. Each global subroutine should have one entry point and one exit point only.

A list of functions typically performed by global subroutines follows:

- Print channel status if a channel error has been detected.
- Print expected and received data.
- Print device and channel register contents.
- Alter baud rate on a communications device.
- Compare data.
- Check channel status.
- Perform housekeeping functions.
- Set up specific test conditions on a device.
- Handle device interrupts.

Example 6-13 is a PDL1 description of a typical routine that prints expected and received data.

```
ROUTINE PRINT_EX_REC
:    GET REGISTER NAME
:    GET RECEIVED DATA
:    GET EXPECTED DATA
:    FIND BITS IN ERROR
:    PRINT REGISTER NAME
:    PRINT EXPECTED DATA
:    PRINT RECEIVED DATA
:    PRINT BITS IN ERROR
END ROUTINE PRINT_EX_REC
```

Example 6-13   Typical Design Specification for a Print
Expected and Received Data Routine

Parameters passed by the calling routine to a global subroutine
are normally pushed on the stack by the caller in the order
specified by the global subroutine. The global subroutine then
accesses these parameters by adding offsets to the argument
pointer and moving the data into the general registers. Example
6-14 shows how parameters are passed with a CALLS instruction in a
VAX-11 Macro program. The first argument is offset 4 bytes from
the argument pointer. The second argument is offset 8 bytes. In
an actual diagnostic program, however, the $DS_ERRHARD_S macro
would push the information on the stack and the print routine
would be called from DS$ERRHARD routine in the supervisor. (Refer
to Example 8-25 in Chapter 8.)

Calling Routine	
TEST_N::	
PUSHL R9	; expected data
PUSHL R8	; received data
PUSHAL ADR_NAME	; address of register name
CALLS #3, PRINT_EX_REC	; Call subroutine.
.	
.	
.	
Global Subroutine	
PRINT_EX_REC::	
.WORD ^M <R2, R3, R4, R5, R6, R7>	
	; Save registers.
MOVL 4(AP), R2	; Save address of register name.
MOVL 8(AP), R3	; Save received data.

```
MOVWL 12(AP), R4                    ; Save expected data.
    .
    .
    .
RET                                 ; Return to calling routine.
```

Example 6-14   The Passing of Parameters from a Calling
                Routine to a Global Subroutine

Example 6-15 shows the arrangement of data as it is placed on the stack.

3	argument pointer (AP)
ADR_NAME	4(AP)
received data	8(AP)
expected data	12(AP)

Example 6-15   Arrangement of Arguments on the Stack

Notice that the general registers used by the global subroutine (R2 to R7) are saved at the beginning of the routine with a register save mask. The RET instruction at the end of the global subroutine restores the original contents of these registers before returning control to the calling routine. See the VAX-11/780 Architecture Handbook, Appendix C, for details on subroutine calls. If the global subroutine must pass data back to the calling routine, the calling routine specifies a buffer in which the global subroutine can write the data.

6.4   TEST ROUTINES AND TEST MODULES

The test portion of level 2, 2R, and 3 programs should be arranged in one or more separate modules. Each module should contain the following elements:

    a module preface
    an equated symbol section
    test code.

The test code in each module should be organized in tests and subtests. The tests divide the program into major functional or logic areas.

A test can be part of more than one section. Each test should be coded as an independent routine to be called by the dispatch routine in the supervisor. It must be able to stand by itself between the execution of initialization and cleanup routines.

Ideally, each test should have one entry point and one exit point. This structure simplifies program debugging procedures. It also helps to ensure proper program flow under unforeseen error conditions. Each test can contain one or more subtests within it. Subtests are blocks of code within the test routine. They check portions of the functional areas or logic areas covered by the test. Although each test in a program must be independent of the tests which precede and follow it, subtests have no such requirement. For example, the fourth subtest in a given test may depend on functions performed in the preceding three subtests. However, each subtest should be constructed so that it can be executed as a tight loop after the preceding subtests have been executed once. A set of utility macros and supervisor service macros are used to define the boundaries of each test and subtest, as shown in Example 6-16.

```
$DS_BGNTEST                                    ; test 1
    $DS_BGNSUB                                 ; test 1, subtest 1
    ;(text code for test 1, subtest 1)
    $DS_ENDSUB
    $DS_BGNSUB
    ;(test code for test 1, subtest 2)         ; test 1, subtest 2
    $DS_ENDSUB
$DS_ENDTEST
$DS_BGNTEST                                    ; test 2
    $DS_BGNSUB                                 ; test 2, subtest 1
    ;(test code for test 2, subtest 1)
    $DS_ENDSUB
$DS_ENDTEST
```

Example 6-16   Use of Structural Macros to Define Test
and Subtest Boundaries

Each test and subtest should be fully documented. Each test must provide a preface that describes the whole test and lists

assumptions. The programmer should explain what functional or logic areas are assumed to be working properly when the test starts.

A complete description and list of assumptions must be included in the preface to each subtest. This description should explain what areas the subtest checks, how the subtest works, and what error isolation procedures should be used when the subtest detects a hardware failure.

Example 6-17 shows part of the documentation for a typical test and subtest.

```
$DS_BGNTEST

        ;++
        ;
        ; Test Description:
        ;
        ; This test checks the map registers in the UBA.  It performs
        ; this test by checking whether all registers hold zeros and
        ; ones.  Then the test will write patterns to the map
        ; registers to check for bits tied together.
        ;
        ; Assumptions:
        ;
        ; Test12
        ; This test assumes that the data path from the CPU to the
        ; UBA has been checked and that register addressing works
        ; correctly.
        ;
        ;--


$DS_BGNSUB

        ;++
        ;
        ; Subtest Description:
        ;
        ; This subtest checks whether UBM000--UBM496 will hold an all
        ; zeros data pattern and an all ones data pattern.
        ;
        ; Assumptions:
        ;
        ; Subtest 1
        ;
        ; Test Steps:
        ;
        ; 1. Init map register index to zero (R3).
        ; 2. Clear selected map register-MP (R3).
        ; 3. If MP (R3) .EQU 0 then continue else report error.
        ; 4. Complement selected register-MP (R3).
```

```
;  5. If MP (R3) .EQU -1 then continue else report error.
;  6. Select next register (update R3).
;  7. If R3 .GTR 496 then exit else GOTO step 2.
;
;  Errors:
;
;  1. Timeout - UBA failed to respond
;  2. Zeros data failure
;  3. Ones data failure
;
;  Debug:
;
;  Error #1-
;  This error could mean power failure.  Check supplies.
;
;  Error #2-
;  Check bit(s) that failed for stuck at one state.
;
;  Error #3-
;  Check bit(s) that failed for stuck at zero state.
;
;--


;  (test code)


$DS_ENDSUB
     .
     .
     .
$DS_ENDTEST
```

Example 6-17   Typical Test and Subtest Documentation

In addition to the macros shown in Examples 6-16 and 6-17, the macros in the diagnostic library and certain VMS service macros are required for many test and subtest operations (Chapters 7-10).

6.5   GUIDELINES FOR LEVELS 1, 2, 2R, 3, 4

In general, the diagnostic programmer should avoid any code that requires the use of implementation specific hardware features (level 4 excepted).   Level 1, 2, 2R, and 3 diagnostic programs should rely strictly on VAX architectural features, so that they are transportable to all VAX implementations.

6.5.1 Guidelines for Writing Level 1 Programs

Level 1 is reserved for high-level diagnostic programs that require use of the operating system. They may or may not require use of the diagnostic supervisor.

A level 1 program may use virtual I/O transfers to perform non-privileged file access.   It may test the operating system software as well as the hardware.

6.5.2   Guidelines for Writing Level 2 Programs

Level 2 diagnostic programs use device driver routines built into the VMS operating system and the supervisor to handle all I/O transfers to the device under test. Device registers must not be accessed directly. In addition, hardware should be totally transparent to the program. The supervisor channel services should not be used.

The I/O driver routines dump the device registers into a buffer allocated by the programmer. The format for the dump varies with each device. Use this buffer (specified by the keyword P6) to check the conditions on the unit under test following QIO completion.

Any service that provides an optional event flag must be given an event flag in the range of 32-63 or a parameter.

Level 2 programs must use physical I/O transfer. Use of virtual I/O transfers will change the program level to 2R.

6.5.3   Guidelines for Writing Level 2R Program

Diagnostic programs should not be designed as level 2R programs unless there is no alternative. However, certain program types, such as a system exerciser, must create subprocesses and allocate and deallocate devices. Such functions require the user environment (on-line mode), making the program level 2R.

In addition, VMS supports some devices (with driver routines) that the supervisor does not support. Diagnostic programs that use QIO to test such devices are designated as level 2R.

6.5.4   Guidelines for Writing Level 3 Diagnostic Programs

Level 3 diagnostic programs make use of VAX system features such as direct I/O and the privileged instructions (e.g., MTPR) unavailable to programs running in the VMS environment. In this way, level 3 programs can define errors more precisely than level 2 programs.

Level 3 programs should use the supervisor channel services to perform all channel functions when accessing a peripheral device. The map registers and memory should not be handled directly. Hardware between the device under test and the program should not be manipulated except through supervisor services. For example, the $DS_GETBUF_x and $DS_RELBUF_x macros should be used when it is necessary to allocate memory.

Level 3 diagnostic programs use maintenance mode type features where the hardware permits, in order to step through certain functions.

Avoid analysis of unexpected errors. The supervisor will handle these.

In general, level 3 programs perform serial testing instead of parallel testing.

6.5.5 Guidelines for Writing Level 4 Diagnostics

A level 4 program must use the VAX native instruction set, and it must run without the supervisor. This means that level 4 programs stand between the microdiagnostic program and the level 3 programs in the diagnostic strategy. Some problems may prevent execution of the supervisor. However, they may go undetected by the microdiagnostics. Level 4 programs may test for these problems.

For example, level 4 programs are necessary to test the instructions required by the supervisor (hard-core) and to test the translation buffer and cache in the VAX CPU.

Avoid developing level 4 diagnostic programs when they are not really necessary. Level 4 programs are more difficult for the operator to run and interpret than level 2, 2R, and 3 programs.

Make your level 4 program as simple as possible. Use straight line code instead of modular program structure. With straight line code, the listing is simple and the program counter (PC) is never reset. Make the instruction set simple, particularly at the beginning of the program.

Write the program in position independent code (PIC) so that it can be loaded into any area in memory if there is a known memory problem. Build in a section of code in the program (preferably at the beginning) that will handle traps, interrupts, and machine checks.

Use the processor Halt instruction on error detection. No print routines should be used. The PC printed on the console points to the failing location plus one. In addition, the programmer should use a location (e.g., location 0) to hold the numbers of the current test and subtest.

Since no error message is printed on error detection, the operator must be able to identify the failing test, subtest, and location through commands to the console.

Use consistent test patterns throughout the program when possible, so that the operator can deal with new errors through the use of techniques already familiar to him. For example, when a verification process involves comparison of expected and received data, the data patterns can be placed in the general registers for easy access, as follows:

    R0 = expected data
    R1 = received data
    R2 = exclusive OR.

Level 4 program documentation must be excellent, since no error messages can be printed. Block comments and line comments should explain fully and precisely what the code is doing.

6.6 GUIDELINES FOR REGISTER TESTING

Diagnostic programs that test hardware must often check registers, buses, and memory locations to detect failures such as bits stuck at one (S-A-1), bits stuck at zero (S-A-0), or bits tied together. One of the most commonly used methods is the writing and reading of the floating one and floating zero patterns (pattern set one, PS1). These patterns show up solid errors in a way apparent to a human observer. However, this pattern set requires far more patterns than necessary. 2 x N patterns are needed, where N is the number of bits in the register. For a 16-bit register, 32 patterns are required. For a 32-bit register, 64 patterns are required.

Another set of patterns, (pattern set 2, PS2), can be generated which requires only $(\log_2 N) + 1$ patterns. Therefore, 4 patterns may be used to check out 8 bits; 5 patterns for 9 to 16 bits; 6 patterns for 17 to 32 bits; etc. This set of patterns can be generated as follows, starting from either end of a register:

1.	Set every other bit.	10101010	1010	10	10
2.	Use the 1's complement of the first pattern.	01010101	0101	01	01 2-bit set
3.	Double the number of adjacent ones and zeros	00110011	0011	0011	4-bit set
	until the last pattern,	00001111	00001111		8-bit set
	(if N is a power of 2)				
	contains half ones and	0000000011111111			16-bit set
	half zeros. Starting				
	with the 3rd pattern,				
	the ones or zeros must				
	be inserted into the				
	same end.				

If pattern 2 ends in . . . .01_2, then
pattern 3 ends in . . . 011_2,
pattern 4 ends in . . 01111_2, etc.

Conversely, if pattern 2 ends in . . . . 10_2, then
pattern 3 ends in . . . .100_2,
pattern 4 ends in . . 10000_2, etc.

These patterns will detect all S-A-1s, all S-A-0s, and bits tied together. Using 5 patterns instead of 32 or 6 instead of 64 could save a lot of time, especially in Quick Verify mode. Consider a routine that tests a RAM containing 256 32-bit registers.

PS1:   256 (32-bit registers) X 10 (microseconds per test) X 64 (patterns) = approximately 164 milliseconds.

PS2: 256 (32-bit registers) X 10 (microseconds per test) X 6
(patterns) = approximately 15 milliseconds.

Whether PS1, PS2, or some other pattern set is used will depend
upon the application. Plug in your own numbers above and then
decide. A saving of 149 milliseconds will not mean anything to a
human operator, but a saving of 149 seconds certainly would.

Another point, if a scope loop is set up when PS1 is used, the
pattern generating algorithm may be inside the scope loop. With
PS2, it is more efficient to store the patterns in memory and use
them one after another. The scope loop should include the writing
and reading of only one pattern at a time.

```
Example 6-18 shows the two PS2 patterns for 32 bits.
32-bit patterns (in hex)        Alternate 32-bit patterns (in hex)
     AAAA AAAA                        5555 5555
     5555 5555                        AAAA AAAA
     3333 3333                        CCCC CCCC
     0F0F 0F0F                        F0F0 F0F0
     00FF 00FF                        FF00 FF00
     0000 FFFF                        FFFF 0000
```

Example 6-18   32-Bit Register Test Patterns

Four types of utility macros are available in the diagnostic macro
library:

    Program format utility macros
    Program control utility macros
    P-table control utility macros
    Symbol definition utility macros.

7.1 CODING UTILITY MACROS

Some of the utility macros call supervisor routines, while others
do not. However, the format for each utility macro call is the
same:

$DS_macro-name   argument list

In the paragraphs that follow, optional arguments are enclosed in
square brackets. All arguments should be specified by position or
keyword name. The format for the header directive macro is shown
in the following example.

```
$DS_HEADER pname, rev, [update], [nunit], [errtyp], [stat]
```

Example 7-1 Header Macro Format

The arguments for PNAME, REV, NUNIT, and ERRTYP can be specified
by position, as follows:

```
$DS_HEADER <SAMPLE TEST>, 01, , 8, 6
```

Example 7-2 Specification of Arguments by Position

Notice that the UPDATE and STAT arguments are omitted, so that the
default arguments will be used. However, an omitted argument must
be indicated with a pair of commas. The commas function as a place
holder. The omitted STAT argument requires no commas, since
nothing follows it.

The same argument list can be specified with keyword names as
shown in the following example.

```
$DS_HEADER        PNAME = <SAMPLE TEST>, -
                  REV = 01, -
                  NUNIT = 8, -
                  ERRTYPE = 6
```

Example 7-3 Specification of Arguments by Keyword Names

Notice that when the argument list is continued from one line to
the next, a hyphen must terminate each line except the last.

Specification of arguments by position and keyword name in combination is a third alternative, as shown below.

```
$DS_HEADER        <SAMPLE TEST>, REV = 01, NUNIT = 8, -
                  ERRTYPE = 6
```

Example 7-4   Specification of Arguments by Position
and Keyword Name

7.2   PROGRAM FORMAT UTILITY MACROS

The program format utility macros generate assembler and linker directives that clarify the diagnostic program structure.

Macros of this type provide one or more of the following functions.

- Generation of subtitles

- Generation of psects

- Generation of routine entry points and masks

- Generation of beginning and ending tags

- Generation of address labels for routines and storage areas

The statements that these macros generate provide listing controls and labels. The supervisor, other program routines, and other macros can use these labels to control the program flow and to access data storage areas.

A list of program format macros follows. Functional explanations and examples are provided.

7.2.1   Program Module Directive Macros

$DS_BGNMOD env, [test], [subtest]

env   =   CEP_FUNCTIONAL, CEP_REPAIR, SEP-FUNCTIONAL or SEP_REPAIR
test   =   number of the first test in this module.
subtest =   number of the first subtest in this module.

NOTES
1. $DS_BGNMOD is absolutely required in every module.

2.   CEP_FUNCTIONAL = CPU cluster environment functional diagnostic with failing function callout on error detection (level 3 only).

3. **CEP_REPAIR** = CPU cluster environment repair diagnostic with failing module callout on error detection (level 3 only).

4. **SEP_FUNCTIONAL** = system or user environment functional diagnostic with failing function callout on error detection (level 2, 2R, or 3).

5. **SEP_REPAIR** = system or user environment functional diagnostic with failing module callout on error detection (level 2, 2R, or 3).

6. Default **TEST** argument = 1.

7. Default **SUBTEST** argument =1.

$DS_ENDMOD (no arguments)
This pair of macros must be used to define the beginning and end of every program module, as shown in Example 7-5.

```
$DS_BGNMOD          SEP_REPAIR
; (module text)
$DS_ENDMOD
```

Example 7-5   Use of Begin and End Module Macros

7.2.2   Subtitle Directives

$DS_SBTTL

$DS_SBTTL generates an assembler directive to provide the given subtitle, with the diagnostic test or subtest number included, for the assembly listing. Use this macro only at the beginning of each test and subtest. Use the normal assembler subtitle directive (.SBTTL) in all other cases.

$DS_SBTTL ascii, [align]

ascii = subtitle string, maximum length = 50.
align = psect alignment of the test.

$DS_PAGE

The $DS_PAGE macro should be used in conjunction with the $DS_SBTTL macro. It causes the .SBTTL assembler directive to appear as the first output line of the next assembler listing page by suppressing the next macro call ($DS_SBTTL) and listing the expansion.

VAX Diagnostic Design Guide

$DS_PAGE (num)
num = 0 or 1.

> 1 indicates that a .PAGE directive should be generated. 0
> indicates that a .PAGE directive should not be generated.

Example 7-6 shows how the $DS_PAGE and $DS_SBTTL macros should be
coordinated.

```
        .
        .
        .
    ; (code)
    $DS_PAGE        NUM = 1
    $DS_SBTTL       <WIDGET TEST>
```

Example 7-6 Subtitle Directives

7.2.3 Header Directive Macro
The $DS_HEADER macro should be used at the beginning of a
diagnostic program to generate a table that defines the program
structure to the supervisor.

$DS_HEADER pname, rev [update], [nunit], [errtype], [stat]

pname   = program name string.
rev   = program release level number.
update   = DEPO patch level number.
nunit   = the maximum number of units that can be tested
           concurrently.
errtype = the number of types of error that can occur on the
           unit under test.
stat   = a pointer to the statistics table.

NOTES
1. REV and UPDATE are used as major and
   minor revision numbers and will be
   changed to MAJREV and MINREV in the
   future.

2. If STAT is specified, it must be
   STATISTIC; refer to $DS_BGNSTAT.

Example 7-7 shows a typical use of the $DS_HEADER macro together
with the macro expansion.

```
79 ; --
80        $DS_HEADER                    <DZ11 8 LINE ASYNC MUX TEST>, REV = 01, DEPO = 0, NUNIT = 8
                           .SAVE
                           .PSECT      $HEADER, PAGE, NOEXE, NOWRT
L$L_HEADLENGTH:            .LONG       A_HEADEND -.         ; LENGTH OF THE HEADER DATA BLOCK.
L$L_ENVIRON:              .LONG       $ENV                 ; PROGRAM ENVIRONMENT.
L$L_NAME:                .ADDRESS T_ NAME                  ; PROGRAM NAME TEXT ADDRESS.
L$L_REV:                 .LONG       01                   ; PROGRAM REVISION LEVEL
L$L_UPDATE:              .LONG       0                    ; DIAGNOSTIC ENGINEERING PATCH ORDER.
L$A_LASTAD:              .ADDRESS LASTAD                   ; FIRST FREE LOCATION AFTER PROGRAM.
L$A_DTP:                 .ADDRESS DISPATCH                 ; TEST DISPATCH TABLE POINTER.
L$A_DEVP:                .ADDRESS AL_DEVTYP                ; DEVICE TYPE LIST POINTER.
L$A_UNIT:                .LONG       8                    ; NUMBER OF UNITS THAT CAN BE TESTED.
L$A_DREG:                .ADDRESS DEV_REG                  ; DEVICE REGISTER CONTENTS TABLE POINTER.
                         .LONG       0[5]
L$A_ICP:                 .ADDRESS INITIALIZE               ; INITIALIZATION CODE POINTER.
L$A_CCP:                 .ADDRESS CLEANUP                  ; CLEAN-UP CODE POINTER.
L$A_REPP:                .ADDRESS SUMMARY                  ; SUMMARY REPORT CODE POINTER.
L$A_STATAB:              .ADDRESS 0                        ; STATISTIC TABLE POINTER.
L$L_ERRTYP:              .LONG       0                    ; # OF TYPES OF $ERRSOFT & $ERRHARD.
L$A_TSTCNT:              .ADDRESS SECTION                  ; LIST OF SECTION NAME ADDRESSES.
A_HEADEND:
T_NAME:                  .ASCIC      \DZ11 8 LINE ASYNC MUX TEST\

                         .PSECT, _LAST, PAGE
LASTAD:                  .PSECT, SYSTCNT, NOEXT, NOWRT, OVR, LONG
```

Example 7-7 Expansion of the
$DS_HEADER Macro in the DZ11
Diagnostic Program (ESDAA)

7.2.4   Dispatch Table Initialization

$DS_DISPATCH (no arguments)

This macro should be used following the $DS_HEADER macro. It generates the psect, beginning tag, address label, and ending tag for the dispatch table. The actual entries in the dispatch table are generated (at link time) by use of the $DS_BGNTST macro at the beginning of each test. Example 7-8 shows the standard use of the $DS_DISPATCH macro.

```
$DS_HEADER    <SAMPLE TEST>, REV = 01, DEPO = 0, NUNIT = 8
.
.
.
$DS_DISPATCH
```

Example 7-8   Use of the $DS_DISPATCH Macro

7.2.5   Data Section Directives

$DS_BGNDATA [align]

align =   Psect alignment. The alignment may be for any of the following data types.

        BYTE
        WORD
        LONG
        QUAD
        PAGE

$DS_ENDDATA (no arguments)

These macros can be used to set up a data section for a specific test. The data section should follow the .SBTTL statement that identifies the test. The data to be used in the test should follow, coming between the $DS_BGNDATA macro and the $DS_ENDDATA macro. The test will be executed once for every argument list in the table.

The $DS_BGNDATA macro generates a label (DATA_00n), where n corresponds to the number of the test. Several argument lists can follow the $DS_BGNDATA macro call. The $DS_ENDDATA macro generates a longword of zeros that functions as the test argument table terminator.

The test code should reference the argument lists with offsets from the argument pointer (AP). Each time the test is repeated, the supervisor advances the argument pointer to point to the next argument list. When the longword of zeros, provided by $DS_ENDDATA, is accessed after the last argument list has been used, control passes through the dispatch routine to the next test. In Example 7-9, three argument lists are given.

```
              $DS_SBTTL          <test name>
                  .
                  .
                  .
              $DS_BGNDATA
                    .LONG 4, 0, M_NRZ, M_ODD, M_RANBYT
                               ; first argument list
                    .LONG 4, 0, M_PE, M_ODD, M_RANBYT
                               ; second argument list
                    .LONG 4, 0, M_NRZ, M_EVEN, M_RANBYT
                               ; third argument list
              $DS_ENDDATA
              $DS_BGNTST
                  .
                  .
                  .
10$           ; (test code)
```

Example 7-9   Test Argument Table Directives

The supervisor calls the test three times, once with each argument
list. Then, when the supervisor encounters the zero longword
supplied by the $DS_ENDDATA macro, control passes to the dispatch
routine in the supervisor.

7.2.6   Device Register Storage Area Directives

$DS_BGNREG (no arguments)
$DS_ENDREG (no arguments)

The device register storage directive macros can be used to mark
the beginning and ending of the memory storage area used for
device register contents. The $DS_BGNREG macro produces a label,
DEV_REG:, as follows.

```
    $DS_BGNREG
        DEV_REG:                        ; label created by $DS_BGNREG
        CSR_REG::     .LONG 0           ; CSR data
        TCR_REG::     .LONG 0           ; TCR data
    $DS_ENDREG
```

Example 7-10   Device Register Storage Area Directives

7.2.7   Statistics Table Directives

$DS_BGNSTAT (no arguments)
$DS_ENDSTAT (no arguments)

The statistics table directives should be used to set up an area
in memory for each logical unit being tested. The hardware and
software errors may be tabulated here and accessed by the summary

report routine. The $DS_BGNSTAT macro sets up a base address with a label (STATISTIC:). The programmer can build the statistics table by storing error information in locations offset from STATISTIC.

The statistics table directives should follow the device register storage area directives in the header module.

```
        $DS_BGNSTAT
STATISTIC:
        $DS_ENDSTAT
```

Example 7-11   Statistics Table Directives

7.2.8   Section Definition Table

$DS_SECTION arg, arg, arg...

This macro should be used in conjunction with the $DS_SECDEF macro to define the program section names. The $DS_SECTION macro should be used in the program text section of the header module. The arguments for $DS_SECTION must appear in the same order as the arguments for $DS_SECDEF. These section names are used with the $DS_BGNTEST macro to show which section a test belongs to. Example 7-12 is typical.

```
;Header Module

        .SBTTL Program Test Section
        ; Program section names
        $DS_SECTION      ALL,H237,H3190,CONVERSATION

;Test Module

        ; Equated symbols
        $DS_BGNMOD       SEP_REPAIR
        .

        .

        .
        $DS_SECDEF       ALL,H327,H3190,CONVERSATION
```

Example 7-12   Use of the $DS_SECTION and $DS_SECDEF Macros

7.2.9   Quadword Descriptor Directive

$DS_STRING text, [locsyml], [locsym2]

text   = an ASCII string to be generated.
locsyml = address of an ASCIC string.
locsym2 = address of an ASCIZ string.

The $DS_STRING macro generates a quadword descriptor for a given character string. The macro provides an ASCII string count and an ASCII string terminated by a zero byte. This macro can be used in the header module or in one of the tests or program subroutines, as appropriate. Example 7-13 shows the usual format.

```
          $DS_STRING <message>
```

Example 7-13   Quadword Descriptor Directive

7.2.10   Initialization Code Directives

$DS_BGNINIT [regmask], [psect]

regmask =   the entry point register save mask.
psect   =   any legal arguments for the .PSECT directive.

$DS_ENDINIT (no arguments)

These macros should be used to provide a frame for the
initialization code. The $DS_BGNINIT macro provides an entry point
to the routine. The $DS_ENDINIT macro produces an exit label, a
call to the DS$BREAK routine in the supervisor (to check for
Control C), and a return instruction (Example 7-14).

```
.SBTTL INITIALIZATION CODE
.
.
.
$DS_BGNINIT
; (init code)
$DS_ENDINIT
```

Example 7-14   Initialization Code Directives

7.2.11   Cleanup Code Directives

$DS_BGNCLEAN [regmask], [psect]

<u>regmask</u>   =   the entry point register save mask.
<u>psect</u>   =   any legal arguments for the .PSECT directive.

$DS_ENDCLEAN (no arguments)

These macros provide a frame for the cleanup routine. The
$DS_BGNCLEAN macro generates an entry mask for the routine. The
$DS_ENDCLEAN macro generates an exit label, a call to the
DS$BREAK routine in the supervisor, and a return instruction.

```
.SBTTL CLEANUP CODE
.
.
.
$DS_BGNCLEAN
; (cleanup code)
$DS_ENDCLEAN
```

Example 7-15   Cleanup Code Directives

7.2.12   Summary Code Directives

$DS_BGNSUMMARY [regmask], [psect]

<u>regmask</u>   =   the entry point register save mask.

psect   =   any legal arguments for the .PSECT directive.

$DS_ENDSUMMARY (no arguments)

These macros provide a frame for the summary routine. The
$DS_BGNSUMMARY macro produces an entry point. The $DS_ENDSUMMARY
macro provides an exit label, a call to the DS$BREAK routine in
the supervisor, and a return instruction.

The summary routine should print out the contents of the
statistics table.

```
        .SBTTL SUMMARY REPORT CODE
        .
        .
        .
        $DS_BGNSUMMARY
        ;(summary report code)
        $DS_ENDSUMMARY
```

Example 7-16   Summary Code Directives

7.2.13   Interrupt Service Routine Directives

$DS_BGNSERV label

label   =   the identifying label for this routine.

$DS_ENDSERV (no arguments)

These macros provide a frame for the interrupt service routine.
The $DS_BGNSERV macro generates an entry point, ensures longword
alignment, and generates push instructions to save R0 and R1 on
the stack. The $DS_ENDSERV macro generates instructions that
restore R0 and R1 and then perform a return from interrupt
instruction.

```
        .SBTTL DEVICE INTERRUPT SERVICE HANDLER
        .
        .
        .
        $DS_BGNSERV     SERVICE
        ; (interrupt routine code)
        $DS_ENDSERV
```

Example 7-17   Interrupt Service Routine Directives

7.2.14   Test Routine Directives

$DS_BGNTEST [section], [regmask], [align]

section =   the test section name(s). If a test belongs to more
              than one section, place the section names in angle

7-11

brackets, < >, and separate the section names with commas.

regmask = entry point register save mask.
align   = boundary alignment.

$DS_ENDTEST (no arguments)

The $DS_BGNTEST and $DS_ENDTEST macros provide a frame for each test routine in a diagnostic program. $DS_BGNTEST produces an entry mask, an entry point, and an entry in the dispatch table. The $DS_ENDTEST macro produces instructions which move a #1 into R0 to indicate normal test completion, call the DS$BREAK routine in the supervisor, and return to the dispatch routine. The test directives should be used in conjunction with the $DS_BGNSUB, $DS_ENDSUB, and $DS_CKLOOP macros to provide program control (Examples 7-18 and 7-24).

```
$DS_SBTTL <WIDGET TEST>
$DS_BGNTEST <DEFAULT,ALL>,ALIGN=LONG
$DS_BGNSUB
; (subtest 1 code)
.
.
.
$DS_ENDTEST
```

Example 7-18   Test Directives

7.2.15   Message Routine Directives

$DS_BGNMESSAGE [regmask]

regmask = the entry point register save mask.

$DS_ENDMESSAGE (no arguments)

These macros should be used to provide a frame around each global print subroutine in a diagnostic program. The $DS_BGNMESSAGE macro produces an entry mask for the routine. The $DS_ENDMESSAGE produces a return instruction.

```
; (data storage)
TAG:
$DS_BGNMESSAGE
; (print routine code)
$DS_ENDMESSAGE
```

Example 7-19 Message Routine Directives

7.2.16   Numeric Error Header Information Directive

The $DS_ERRNUM macro inserts an error number into an argument list at the label given. If an error number is not given, the next sequential error number will be used. The macro does not call a supervisor routine.

$DS_ERRNUM label, [num]

label = label on argument list.
num   = error number to insert.

NOTE

This macro generates executable code that uses (destroys) R0. R0 is left pointing to LABEL.

7.3   PROGRAM CONTROL UTILITY MACROS

The program control utility macros produce executable code which, in some cases, alters the flow of the program. Some of the macros, like $DS_BREAK, generate calls to supervisor routines. Others, like $DS_BCOMPLETE, merely expand to a few instructions without calls to any routine. When program flow is altered with a branch or jump, successful execution of the instructions generated often involves correct use of related macros. For example, $DS_CKLOOP, the check loop macro, must be coordinated with the $DS_BGNSUB and $DS_BGNTST macros.

Not all of the program control utility macros require arguments. The supervisor routines called by utility macros do not provide return status codes.

7.3.1   Pass Control Macros

$DS_BPASS0 label

label = the address for transfer of program control.

$DS_BNPASS0 label

label = the address for transfer of program control.

Either $DS_BPASS0 or $DS_BNPASS0, together with the $DS_ENDPASS_x supervisor service macro, must be used in the initialization routine of a diagnostic program. The $DS_BPASS0 and $DS_BNPASS0 macros check the status of a one time switch used for program initialization. These macros enable the programmer to omit execution of certain portions of the initialization code in order to keep track of test and pass iterations. The $DS_ENDPASS_x supervisor service macro calls a routine in the supervisor (DSX$DSENDPASS) to mark the end of a pass of the diagnostic program. If the number of passes run corresponds to the number of

passes selected by the operator, the routine causes execution of
the summary and cleanup routines. Example 7-20 shows how the
$DS_BPASS0 and $DS_ENDPASS_x macros can be coordinated.

```
1            .SBTTL INITIALIZATION CODE
2            $DS_BGNINIT
3            $DS_BNPASS0 10$              ; First time through?
4            CLRL LOG_UNIT                ; Yes, start with first unit.
5            BRB 20$                      ; Begin device
                                          ; initialization.
6    10$:    INCL LOG_UNIT               ; No, start with next unit.
7            CMPL DSA$GL_UNITS,   LOG_UNIT
8                                         ; Is last unit tested?
9            BNEQ 20$                     ; No.
10           $DS_ENDPASS_G                ; Yes, check for last pass.
11           CLRL LOG_UNIT                ; Last pass not done.
12   20$:    $DS_GPHARD . . .             ; Get P-table address.
13           $ASSIGN . . .                ; Assign a channel.
14              .
15              .
16              .

17           CLRQ    CSRW                 ; Initialize device under
                                          ; test.
```

Example 7-20   Pass Control in Initialization Code

In Example 7-20, on the first execution of the first pass of the
program, the unit number is set to 0 at line 4 and then control
passes to line 12. On the second execution of the first pass, the
pass 0 flag is clear. Control passes from line 3 to line 6, and
the logical unit number is incremented. When the last unit
selected has been tested, the incremented logical unit number
equals MAX_UNITS, and the $DS_ENDPASS_G macro is executed.

If the last pass has been completed, the supervisor routine called
by $DS_ENDPASS_G passes control to the summary and cleanup
routines. If the last pass has not been completed, control returns
to line 11 for selection of the first device and the execution of
another pass.

When the logical unit number is incremented in line 6 and the last
unit has not been tested, the BNEQ instruction in line 9 causes a
branch to line 12, beginning the initialization of the device
under test. Compare this code with the PDL1 description of an
initialization routine for serial testing in Example 6-10.

7.3.2   Quick Flag Macros

$DS_BQUICK label
$DS_BNQUICK label

label = the address for transfer of program control.

These macros check the status of the Quick control flag. The macros enable the programmer to provide branches around certain lengthy tests when the operator desires a quick execution of the program. The tests that are executed when the Quick flag is set should contain the core of the program and provide at least a cursory check of the device under test. The $DS_BQUICK and $DS_BNQUICK macros can be used in conjunction with the $DS_EXIT macro and the $DS_ENDTEST macro, as shown in Example 7-21.

```
        .SBTTL TEST 3
        .
        .
        .
        $DS_BGNTST
        $DS_BNQUICK 10$        ; Branch if not quick
                               ; verify mode.
        $DS_EXIT TEST          ; Branch to $DS_ENDTEST macro to
                               ; leave test 3.
10$:    CLRL R6                ; Clear character counter.
        ; (test code)
        $DS_ENDTEST
```

Example 7-21   Quick Flag Macros

7.3.3   Operator Flag Macros

$DS_BOPER label
$DS_BNOPER label

label =   the address for transfer of program control.

This pair of macros enables the programmer to force execution or omission of certain tests, depending on the state of the Operator control flag. The $DS_BOPER macro generates instructions that check the status of the Operator flag and cause a branch to the label specified by the programmer if the flag is set. In a similar fashion, the $DS_BNOPER macro generates a branch if the Operator flag is clear.

These flags may be used in tests or portions of tests that require operator intervention, as shown in Example 7-22.

```
        .SBTTL TEST 4
        $DS_BGNTST
        $DS_BOPER 10$          ; Branch if operator is present.
        $DS_EXIT TEST          ; Branch to $DS_ENDTEST
                               ; macro to leave test 4.
```

```
10$:        $DS_BGNSUB
              .
              .
              .
            $DS_ENDSUB
              .
              .
              .
            $DS_ENDTEST
```

Example 7-22   Operator Flag Macros

7.3.4   Program Subtest Control Macros

$DS_BGNSUB (no arguments)
$DS_ENDSUB (no arguments)

This pair of macros should be used to form a frame around each subtest in a test. The $DS_BGNSUB macro provides an entry point to the subtest and generates a call to a supervisor routine to indicate the beginning of a subtest. The supervisor routine ensures that the diagnostic program is sequencing through the subtests in each test in numerical order. If the routine detects a sequencing error, it notifies the operator and returns control to the command mode.

The $DS_ENDSUB macro generates an exit label for the subtest and generates a call to a supervisor routine (RENDSUB). This routine checks both the test numbers and the subtest numbers within each test for correct sequence. On error detection the supervisor routine notifies the operator of the error and returns control to the command mode. Example 7-23 shows coordination of the test and subtest control macros.

```
            $DS_BGNTST
            $DS_BGNSUB
            ; (subtest 1 code)
            $DS_ENDSUB
            $DS_BGNSUB
            ; (subtest 2 code)
            $DS_ENDSUB
            $DS_ENDTEST
```

Example 7-23   Program Subtest Control Macros

7.3.5   Loop Control Macros

$DS_CKLOOP label

label = the address for transfer of program control.

This macro generates a call to a supervisor routine that controls the subtest looping mechanism. If an error has been detected (Error flag is set) and the Loop flag is set, the supervisor routine sets up a scope loop on the error by causing a branch back to the label specified.

THE $DS_CKLOOP should be used in conjunction with the following macros:

```
$DS_BGNTEST
$DS_ENDTEST
$DS_BGNSUB
$DS_ENDSUB
```

Example 7-24 shows the test structure that should be used.

```
        .SBTTL TEST 5
        $DS_BGNTEST
        $DS_BGNSUB
LABEL_1:
        ; (test code 1)
        $DS_CKLOOP    LABEL_1
LABEL_2:
        ; (test code 2)
        $DS_CKLOOP    LABEL_2
        $DS_ENDSUB
        $DS_BGNSUB
LABEL_3:
        ; (test code 3)
        $DS_CKLOOP    LABEL_3
        $DS_ENDSUB
            .

            .

            .
        $DS_ENDTEST
```

Example 7-24   Loop Control Macro Use

If an error occurs in test code 1, a loop will be made from the first $DS_CKLOOP macro back to LABEL_1. In the same way, if an error occurs in the test code 2, a loop will be made from the second $DS_CKLOOP macro back to LABEL_2.

However, if the $DS_CKLOOP statements were omitted, only one loop would occur in the first subtest (from $DS_ENDSUB to $DS_BGNSUB), no matter how many errors were detected. Note that the LABEL argument supplied with the $DS_CKLOOP macro must refer to an address within the same subtest. If the test has no subtests, LABEL must refer to an address within the same test.

Program loops may be nested and/or overlapped in order to satisfy local program requirements. The supervisor stores the addresses of both the error call and the $DS_CKLOOP macro statement, so that subsequent repetitions of the loop will be consistent with the first. However, if a different error should then occur within the loop, the loop range will be modified to conform to the latest error condition.

7.3.6   Escape Control Macro

$DS_ESCAPE arg

<u>arg</u> = TEST or SUB

This macro provides a conditional exit from a diagnostic program test or subtest. The macro generates a call to a supervisor routine that checks the status of the Error flag. If the Error flag is set, control passes to the end of the current subtest or test, depending on the argument supplied. The $DS_ESCAPE macro enables the programmer to eliminate execution of certain portions of a program if prior testing indicates that those portions would be redundant and bound to fail. Note that the escape sequence preserves the contents of RØ.

If the $DS_ESCAPE macro is to be used with test code that is part of an error loop, it should be placed after the macro that causes the loop, as shown in Example 7-25.

```
           .SBTTL TEST 6
           $DS_BGNTST
           ; (set up code)
           $DS_BGNSUB           ; subtest 1
LABEL_1:
           ; (test code 1)
LABEL_2:
           ; (test code 2)
           $DS_CKLOOP    LABEL_2
           $DS_ESCAPE    SUB
LABEL_3:
           ; (test code 3)
           $DS_ENDSUB
           $DS_BGNSUB           ; subtest 2
           ; (test code 4)
           $DS_ENDSUB
           $DS_ENDTEST
```

Example 7-25 Escape Control Macro

If an error occurs in test code 2 and the Loop flag is set, a loop from the $DS_CKLOOP macro statement to LABEL 2 will occur. However, if an error in test code 2 occurs and the Loop flag is not set, the $DS_ESCAPE macro will give program control to the second subtest, thus bypassing test code 3.

7.3.7   Exit from Program Phase Macro

$DS_EXIT arg

arg = TEST, SUB, INIT, CLEAN, SERV, MESSAGE, SUMMARY

This macro generates an unconditional branch statement, causing a branch to the last statement in the current section of the program, depending on the argument. This macro can be used to omit execution of code requiring some condition that is not present, as shown in Example 7-22.

7.3.8   Break in Diagnostic Program Macro

$DS_BREAK (no arguments)

This macro calls the DS$BREAK routine in the supervisor to check for Control C. If the operator has typed Control C, setting a flag, the routine inserts a breakpoint and passes control to the command mode in the CLI. Note that the DS$BREAK routine is also called whenever a supervisor service routine is called. The DS$BREAK routine is the only mechanism provided in the supervisor that allows a diagnostic program to check the status of the Control C flag. However, if the diagnostic program performs loops that do not contain any supervisor calls, the operator will be unable to interrupt the program with a Control C, once the program has gone into a tight loop. In order to prevent this condition, any potential loop that does not include a supervisor service call should contain a $DS_BREAK macro call to check for Control C.

7.3.9   Branch on Complete Utility Macros

$DS_BCOMPLETE label
$DS_BNCOMPLETE label

label = the address for transfer of program control.

These macros generate instructions that test R0 and branch conditionally to the address specified by LABEL. The $DS_BCOMPLETE macro call causes a branch if the low bit of R0 is set, indicating successful completion of some function.

The $DS_BNCOMPLETE macro causes a branch if the low bit of R0 is clear, indicating the failure of some function.

By convention, when a test routine or service routine is executed, a return status code is placed in R0. The low-order bit indicates success or failure. Other bits in the register may be used to qualify the nature of the success or failure, but a test of bit 0 should be performed before the other bits are analyzed. The branch on complete macros simplify the testing of bit 0, as shown in Example 7-26.

```
10$:      $DS_CHANNEL_S -           ; channel call
               UNIT = LOG_UNIT-     ; device
               FUNC = # CHC$_DSINT  ; Disable interrupts.
          $DS_BCOMPLETE 20$         ; success?
          $DS_ERRSYS_G LIST         ; Print error message.
          $DS_ABORT TEST            ; Leave test.
20$:      INCL R5                   ; proceed
```

Example 7-26   Branch on Complete Utility Macro Use

7.3.10   Branch on Error Utility Macros

$DS_BERROR label
$DS_BNERROR label

label = the address for transfer of program control.

These macros, like the branch on complete macros, generate instructions that test R0 and branch conditionally to the address specified by LABEL.

The $DS_BERROR macro call causes a branch if the low bit of R0 is clear. The $DS_BNERROR macro call causes a break if the low bit of R0 is set.

One of these macros should be used to test for successful completion of a previous service call or test routine (Example 7-27).

7.3.11   Aborting the Test Sequence Macro

$DS_ABORT arg

<u>arg</u> = the level of abort (PROGRAM or TEST). The default argument
        is PROGRAM.

This macro enables the programmer to abort execution of the
current test or the entire program, depending on the argument. If
the argument is PROGRAM, the macro generates a call to a
supervisor routine. This routine prints an abort message on the
operator's terminal, executes the program's cleanup code, and then
transfers control to the supervisor.

If the argument is TEST, the macro clears R0 indicating test
failure and then does an RET to call the supervisor dispatch
routine and start the next test.

The $DS_ABORT macro should be used to abort the current test or
the program at any point where it is clear that further execution
will be invalid. For example, if a system error is detected in the
initialization routine, the program should be aborted.

```
        .SBTTL INITIALIZATION CODE
        $DS_BGNINIT
        .
        .
        .
10$:    $DS_GPHARD_S -                  ; Get hardware P-table
                                        ; address.
            DEVNUM = LOG_UNIT, -        ; device under test
            ADRLOC = HDW_PTBLE          ; address of pointer to
                                        ; P-table
        $DS_BNERROR 20$                 ; Branch if no error.
        $DS_ERRSYS_S -                  ; system fatal error
            UNIT = LOG_UNIT, -          ; device
            MSGADR = MSG40              ; system services
                                        ; error message
        $DS_ABORT PROGRAM               ; Fatal error, abort program.
20$:    MOVL HDW_PTBLE, R2              ; Move P-table base to index.
```

Example 7-27   Program Abort Macro

In Example 7-27, a supervisor service macro ($DS_GPHARD) is used
to obtain the hardware P-table base address for a specific device.
The $DS_BNERROR macro is then used to test the return status code.
If no error is detected, the program performs a branch to location
20$. Otherwise, an error message is printed and the program is
aborted.

7.4 $DS_CLI Command Line Interpreter Tree Macro
$DS_CLI is a utility macro. The macro is used to build parsing
trees. Each call generates one node in the tree. The parser goes
down the tree until a mismatch or branch directive is encountered.

down the tree until a mismatch or branch directive is encountered.
The macro does not call a supervisor routine. Example 8-27 in
Chapter 8 shows how the macro can be used to build a tree.

$DS_CLI char, [action], [miss], [ascii]

char   = comparison character. See Note 1.
action = a code to be passed to the action routine. See note 2.
miss   = a mismatch or branch displacement in the tree.
ascii   = an ASCII string to use for comparison.
Return status codes: not applicable.

NOTES
1. Special character codes defined by
$DS_CLIDEF to be optionally used as
the CHAR argument:

Symbolic	Function
CLI$K_ERROR	Action/Parser return (bit 0 of R0 = 0).
CLI$K_EXIT	Action/Parser return (bit 0 of R0 = 1).
CLI$K_BR	Unconditional branch within the tree using MISS.
CLI$K_BIF	Branch if. Checks bit 0 of R0. Bit 0 = 0, fall through to next node. Bit 0 = 1, branch using MISS.
CLI$K_SPACE	Traverse spaces and/or tabs and call ACTION if any were found (R8 points to next non-space CHAR).
CLI$K_NUM	Traverse numeric fields and call action with numeric value in R10. This function uses the default radix. It branches using MISS if no numeric data is found.
CLI$K_ALPHA	Traverse alphabetic fields. (Uppercase A-Z).
CLI$K_ALNUM	Traverse alphanumeric fields. (Uppercase A-Z or 0-9).
CLI$K_OCT	Same as CLI$K_NUM, except that the octal radix is forced.
CLI$K_DEC	Same as CLI$K_NUM, except that the decimal radix is forced.
CLI$K_HEX	Same as CLI$K_NUM, except that the hex radix is forced.
CLI$K_STRING	ASCII argument used for match. (Note: Only the first character of the string need match).

2. Upon entry to the action routine the
registers contain:

Register	Content
R0	Action code parameter from tree
R7	Parse tree pointer
R8	Input string pointer
R9	Input string count remaining
R10	Numeric data buffer

7.5   P-TABLE CONTROL MACROS

The VAX diagnostic system provides for the addition of diagnostic programs that test special devices that are not parts of any standard VAX configuration. However, the types of parameters that describe special devices vary. The supervisor does not automatically know what parameters to prompt the operation for in such cases.

Therefore, diagnostic programs that test special devices must include macros that define a P-table descriptor for each unit type to be tested. The resulting P-table descriptor defines the device dependent information offsets in the P-table, and it defines the fields required and their locations.

At run time, as the computer operator runs the supervisor and prepares to run the program that tests the special device, he must load the program before attaching the device to be tested with the ATTACH command. In response to the ATTACH command, the supervisor looks into the program already loaded for the P-table descriptor. It then prompts the operator for the required device parameters and adds the retrieved information to the P-table.

When the supervisor searches for a P-table descriptor, it checks the DEVTYP list in the program first. If the search is unsuccessful, the supervisor checks a list within the supervisor itself. In this way, the program can define a device without requiring an update to the supervisor.

You must use two types of macros to build P-table descriptors:

> P-table descriptor macros
> Structure definition macros

7.5.1   P-Table Descriptor Macros

The following macros are available to build P-table descriptors.

7.5.1.1 $DS_$INITIALIZE Start P-Table Processing

$DS_$INITIALIZE device, length, max, driver

device =   the hardware name for the device, e.g., RP06 or DW780.
length =   the total length of the associated P-table.
max   =   the maximum allowable unit number in the device name. This number should be zero if a unit number is not allowed (as on a TM03).
driver =   the two-character QIO device driver name. This argument should be null if it is not applicable.

Use this macro as the first of the P-table descriptor macros.

7.5.1.2 $DS_$DECIMAL Retrieve and Range Check a Decimal Number

$DS_$DECIMAL prompt, low, high

prompt = the ASCIC name of the field used as a prompt for the
          operator, if needed.
low   = the low limit for the parameter supplied by the
          operator.
high   = the high limit for the parameter supplied by the
          operator.

Use this macro to prompt the operator for a decimal value. After
the range check, the value becomes the current VALUE.

7.5.1.3 $DS_$OCTAL Retrieve and Range Check an Octal Number

$DS_$OCTAL prompt, low, high

prompt = the ASCIC name of the field used as a prompt for the
          operator, if needed.
low   = the low limit for the parameter supplied by the
          operator.
high   = the high limit for the parameter supplied by the
          operator.

Use this macro to prompt the operator for an octal value. After
the range check the value becomes the current VALUE.

7.5.1.4 $DS_$HEXADECIMAL Retrieve and Range Check a Hexadecimal
Number

$DS_$HEXADECIMAL prompt, low, high

prompt = the ASCIC name of the field used as a prompt for the
          operator, if needed.
low   = the low limit for the parameter supplied by the
          operator.
high   = the high limit for the parameter supplied by the
          operator.

Use this macro to prompt the operator for a hexadecimal value.
After the range check the value becomes the current VALUE.

7.5.1.5 $DS_$STRING Retrieve and Verify an ASCII String

$DS_$STRING prompt, strings

prompt = the ASCIC name of the field used as a prompt for the
           operator, if needed.
strings = a list of valid strings.

Use this macro to prompt the operator for an ASCII string. The
macro scans the input stream for a matching string.

7.5.1.6 $DS_$LITERAL Define a Constant Value

$DS_$LITERAL value

value =   a constant.

Use this macro to create a constant value for VALUE from the
program, if there is no need to retrieve a value from this
operator.

7.5.1.7 $DS_$FETCH Extract a Field from the P-Table

$DS_$FETCH offset, bit, size

offset =   a byte offset from the base of the P-table.
            O<OFFSET<65536.
bit   =   a bit displacement from the beginning of the OFFSET
            parameter. Ø<BIT<255.
size   =   the size of the field (in bits) to be extracted from the
            P-table. Ø<SIZE<32.

Use this macro to extract a field from the P-table. The parameters
OFFSET, BIT, and SIZE specify the field in the P-table.

7.5.1.8 $DS_$STORE Insert a Value into the P-Table

$DS_$STORE offset, bit, size

offset =   a byte offset from the base of the P-table.
            Ø<OFFSET<65536.
bit   =   a bit displacement from the beginning of the OFFSET
            parameter. Ø<BIT<255.
size   =   the size of the field (in bits) to be inserted into the
            P-table. Ø<SIZE<32.

7.5.1.9 $DS_$END Finish Processing P-Table

$DS_$END (no arguments)

Use this macro to mark the end of the P-table descriptor.

7.5.2   Structure Definition Macros
Use the three structure definition macros to define the structure
of the addition to the P-table that you wish to create. These
macros are available through LIB.MLB in VMS.

7.5.2.1 $DEFINI Start a Data Structure

$DEFINI struc, gbl, dot

struc =   the structure name (e.g., the name of the P-table).
gbl   =   GLOBAL or LOCAL.
dot   =   the value of the first symbol to be generated.

Use this macro to start the definition of a data structure.

7.5.2.2 $DEF Define Some Fields Within the Structure

$DEF sym, alloc, siz

sym   = the name of the symbol to be defined.
alloc = an assembler directive indicating the block size to be
        used:
        .BLKB
        .BLKW
        .BLKL
siz   = the number of blocks to be allocated.

Use this macro to generate an offset for the symbol given.

7.5.2.3 $DEFEND Finish Definitions

$DEFEND struc, gbl

struc = the structure name (e.g., the name of the P-table).
gbl   = GLOBAL or LOCAL.

Use this macro to clean up the data structure definition process
after all of the symbols for the structure have been defined.

7.5.3   P-Table Control Macro Examples

Example 7-28 shows how the supervisor uses the data structure
macros to define device dependent offsets for the P-table for the
RH780 interface.

```
$DEFINI     RH780,$GBL,HP$A_DEPENDENT
$DEF        RH780$B_TR,.BLKB,1          ; TR number of adapter
$DEF        RH780$B_BR,.BLKB,1          ; BR level of adapter
$DEF        RH780$K_LEN
$DEFEND     RH780,$GBL,DEF
```

Example 7-28 Building a Data Structure
for the RH780 P-Table Offsets

In this example the programmer has created a data structure called
RH780, which defines the symbolic offsets for the P-table
describing an RH780. The symbols are global, and the value of the
first symbol to be generated is HP$A_DEPENDENT. The symbols
RH780$B_TR and RH780B_BR are both allocated one byte of storage
space. The symbol RH780$K_LEN points to a value containing the
length of the P-table.

Example 7-29 shows how the supervisor uses the P-table descriptor
macros to create a P-table descriptor that fills in the P-table at
run time.

```
1    $DS_$INITIALIZE  RH780,RH780$K_LEN,8  ; Start P-table
                                           ; descriptor.
2    $DS_$DECIMAL     TR,1,15              ; Request TR number.
3    $DS_$STORE       RH780$B_TR,0,8       ; Store TR in
                                           ; RH780$B_TR.
4    $DS_$STORE       HP$A_DEVICE,13,4     ; Store TR in bits 13-16
                                           ; of device address.
5    $DS_$STORE       HP$W_VECTOR,2,4      ; Store TR in bits 2-5
                                           ; of vector.
6    $DS_$DECIMAL     BR,4,7               ; Request BR.
7    $DS_$STORE       RH780$B_BR,0,8       ; Store BR in
                                           ; RH780$B_BR.
8    $DS_$STORE       HP$W_VECTOR,6,2      ; Store BR in bits 6-7
                                           ; of vector.
9    $DS_$LITERAL     6                    ; Get a literal 6.
10   $DS_$STORE       HP$A_DEVICE,28,4     ; Set bits 29-30 of
                                           ; device address.
11   $DS_$LITERAL     1                    ; Get a literal 1.
12   $DS_$STORE       HP$W_VECTOR,8,1      ; Set bit 8 in SCB
                                           ; vector offset.
13   $DS_$END                              ; End P-table
                                           ; descriptor.
```

Example 7-29 Building a P-Table Descriptor

In this example from the supervisor, line 1 initializes the
P-table descriptor for the RH780. As many as eight units may be
selected by the operator. The $DS_$DECIMAL macro in line 2 will
cause the supervisor to prompt the operator with the symbol TR.
The supervisor will accept a number within the range of 1 to 15.

The $DS_$STORE macro in line 3 causes the supervisor to store the
value retrieved from the operator (the TR number) in the location
specified by the offset symbol RH780$B_TR. The bit displacement
from the beginning of the location is zero, and the size of the
field is eight bits. Line 4 stores the same TR value in bits 13 to
16 of the device address and line 5 stores the TR value in bits
2-5 of the device vector.

Lines 6 to 8 prompt for and store the BR number in the same way.
Lines 9 and 10 use a literal 6 to set bits 29 and 30 of the device
address. Lines 11 and 12 set bit 8 in the vector; and the $DS_$END
macro terminates the P-table descriptor.

7.6   SYMBOL DEFINITION UTILITY MACROS
The symbol definition utility macros define symbols common to the
VAX diagnostic system. Many of the symbols defined are keyword
names for other macros. These symbols are necessary to the
supervisor-program interface. Other definition macros define
useful symbols such as bit names, table offsets, and return status
codes. The symbol definition utility macros should be used
primarily in the program equates section of the header module. If
other sets of symbols are confined to any specific module, the
symbols should be defined in the equates section of that module.
Example 6-2 in Chapter 6 shows a typical use of some of the
definition macros in the header module template. See the channel
services description, Chapter 8, Paragraph 8.4, for the symbols
used in the channel service macros.

7.6.1 $DS_BITDEF Define Bit Mask Mnemonics

$DS_BITDEF [gbl]

g̲b̲l̲ = GLOBAL or LOCAL.

This macro defines the mnemonics BIT0 to BIT31 and the masks
corresponding to each bit.

7.6.2 $DS_CFDEF Call Frame Definitions

$DS_CFDEF [gbl]

g̲b̲l̲ = GLOBAL or LOCAL.

This macro provides symbolic definitions for call frame offsets
from the frame pointer (FP).

The defined symbols are:

CF$L_ONCOND	Condition Handler
CF$W_PSW	Processor Status Word
CF$W_MASK	Register Mask
CF$L_AP	Old Argument Pointer
CF$L_FP	Old Frame Pointer
CF$L_PC	Return PC
CF$L_REG	Saved Registers

7.6.3 $DS_CHCDEF Channel Function Definition

$DS_CHCDEF [gbl]

g̲b̲l̲ = GLOBAL or LOCAL.

This macro provides symbolic definitions for the functions used in
the channel call macro $DS_CHANNEL_x.

7.6.4 $DS_CHIDEF Interrupt Status Definitions

$DS_CHIDEF [gbl]

gbl = GLOBAL or LOCAL.

This macro defines the codes that are used by the channel service, $DS_CHANNEL_x, to indicate adapter status following an interrupt. The $DS_CHIDEF macro is used in conjunction with the CHC$_ENINT function of the $DS_CHANNEL_x macro.

7.6.5 $DS_CHMDEF Channel Mapping Function Definition

$DS_CHMDEF [gbl]

gbl = GLOBAL or LOCAL.

This macro provides symbolic definitions for the $DS_SETMAP_x functions.

7.6.6 $DS_CHSDEF Channel Adapter Status Definitions

$DS_CHSDEF [gbl]

gbl = GLOBAL or LOCAL.

This macro provides symbolic definitions for the return status codes for the CHC$_STATUS function of the $DS_CHANNEL_x macro.

7.6.7 $DS_CHDEF Channel Symbol Definitions

$DS_CHDEF [gbl]

gbl = GLOBAL or LOCAL.

This macro invokes four other macros ($DS_CHCDEF, $DS_CHIDEF, $DS_CHMDEF, and $DS_CHSDEF) to define all of the channel symbols used by the channel service calls.

7.6.8 $DS_CLIDEF Command Line Interpreter Definitions

$DS_CLIDEF [gbl]

gbl = GLOBAL or LOCAL.

This macro generates special character code definitions for the CHAR parameter of the $DS_CLI macro. A list of these symbols follows:

        CLI$K_ERROR
        CLI$K_EXIT
        CLI$K_BR

```
CLI$K_BIF
CLI$K_SPACE
CLI$K_NUM
CLI$K_ALPHA
CLI$K_ALNUM
CLI$K_OCT
CLI$K_DEC
CLI$K_HEX
CLI$K_STRING
```

7.6.9 $DS_DEVTYP Device Types

$DS_DEVTYP [<name, name,...>][,<addresses of P-table descriptors separated by commas>]

name = 2-character generic device type.

This macro generates a string of device type names and a string of P-table descriptors known to the program. The device type names listed below are the type names recognized in the Attach command (see Chapter 5, Paragraph 5.3.1).

Device Type Name

KA780	RK06
MS780	TM03
RH780	TE16
DW780	TU45
RP07	DZ11
RP06	DMC11
RP05	LP11
RP04	CR11
RM03	DR11B
RK611	PCL11
RK07	

7.6.10 $DS_DSADEF Define APT Command and Mailbox Flag Offsets

$DS_DSADEF [gbl]

gbl = GLOBAL or LOCAL.

This macro provides symbols that define CLI flags, command areas, and APT mailbox areas. DSA$AL_APTMAIL is the base address of the APT mailbox.

Symbol	Meaning
DSA$GL_FLAGS	longword containing the following flag bits

Flags

DSA$M_HALTD	halt on error detection
DSA$M_HALTI	halt on error isolation
DSA$M_LOOP	loop on error flag
DSA$M_BELL	bell on error
DSA$M_IE1	inhibit all error reports
DSA$M_IE2	inhibit basic error reports
DSA$M_IE3	inhibit extended error reports
DSA$M_IES	inhibit summary reports
DSA$M_QUICK	quick verify
DSA$M_SPOOL	spool output messages
DSA$M_TRACE	trace tests
DSA$M_LOCK	lock in physical memory
DSA$M_OPER	operator present
DSA$M_PROMPT	display long dialogue
DSA$M_NORPT	suppress all output to the terminal
DSA$M_USER	user environment
DSA$M_PASS0	pass 0 flag
DSA$M_APT	APT mode

Command Area Definitions

DSA$GL_APTCOM	APT command
DSA$GL_PASSES	passes to run
DSA$GL_UNITS	number of units to be tested
DSA$GL_SECTNO	section number
DSA$GL_CPUTYP	VAX CPU type code

APT Mailbox Area Definitions

DSA$GL_MSGTYP	message type
DSA$GL_ERRNO	error number
DSA$GL_EVENT	event counter
DSA$GL_SUBTNO	subtest number
DSA$GL_TESTNO	test number
DSA$GL_PASSNO	pass number
DSA$GL_DEVLEN	device descriptor length
DSA$GL_DEVNAM	device descriptor
DSA$GL_MSGPTR	message descriptor

7.6.11 $DS_DSDEF Define Supervisor Status and Condition Codes

$DS_DSDEF [gbl]

gbl = GLOBAL or LOCAL.

This macro generates symbols that denote error conditions. The supervisor service macros call supervisor service routines that return one of the codes represented by these symbols in R0 to indicate the return status. The following symbols are generated.

Symbol	Meaning
DS$_WARNING	warning
DS$_NORMAL	normal
DS$_ERROR	error condition
DS$_SEVERE	severe error condition
DS$_OVERFLOW	overflow
DS$_NULLSTR	null string
DS$_PROGERR	program error
DS$_TRUNCATE	data truncation
DS$_NOTDON	not done
DS$_IVVECT	invalid vector
DS$_IVADDR	invalid address
DS$_VASFUL	virtual address space full
DS$_INSFMEM	insufficient memory
DS$_MMOFF	memory management is off
DS$_IHWE	initial hardware error
DS$_FHWE	final hardware error
DS$_LOGIC	interface error
DS$_ILLPAGCNT	illegal page count
DS$_FRAGBUF	buffer was fragmented when released
DS$_MCHK	machine check
DS$_KRNLSTK	kernel stack not valid
DS$_POWER	power fail
DS$_TRANSL	translation not valid
DS$_CHME	change mode error
DS$_NOTIMP	not implemented
DS$_IPL2HI	IPL is too high
DS$_ICERR	interval clock error
DS$_ICBUSY	interval clock busy
DS$_ARITH	arithmetic trap

7.6.12 $DS_DSSDEF Define Supervisor Service Entry Points

$DS_DSSDEF [gbl]

gbl = GLOBAL or LOCAL.

This macro generates symbols that define the supervisor service entry points for the diagnostic program. When a supervisor service macro is expanded, it generates a call to one of these entry points in the supervisor entry module. The $DS_DSSDEF macro generates the following symbols.

DS$ENDPASS	DS$MMOFF
DS$GPHARD	DS$ABORT
DS$SUMMARY	DS$SETVEC
DS$CLRVEC	DS$INITSCB
DS$SETIPL	DS$CHANNEL

DS$SETMAP	DS$SHOCHAN
DS$BGNSUB	DS$ENDSUB
DS$CKLOOP	DS$INLOOP
DS$ESCAPE	DS$BREAK
DS$WAITMS	DS$WAITUS
DS$CANWAIT	DS$CNTRLC
DS$ASKDATA	DS$ASKVLD
DS$ASKADR	DS$ASKLGCL
DS$ASKSTR	DS$CVTREG
DS$PARSE	DS$ERRSYS
DS$ERRDEV	DS$ERRHARD
DS$ERRSOFT	DS$PRINTB
DS$PRINTX	DS$PRINTF
DS$PRINTS	DS$ELOGON
DS$ELOGOFF	DS$GETBUF
DS$RELBUF	DS$GETMEM
DS$MOVVRT	DS$MOVPHY
DS$MMON	SYS$QIOW
SYS$ALLOC	SYS$ASSIGN
SYS$BINTIM	SYS$CANCEL
SYS$CANTIM	SYS$CLREF
SYS$DALLOC	SYS$DASSGN
SYS$GETTIM	SYS$QIO
SYS$READEF	SYS$SETEF
SYS$SETIMR	SYS$SETPRT
SYS$WAITFR	SYS$WFLAND
SYS$WFLOR	SYS$GETCHN

7.6.13 $DS_ENVDEF Define Environment Codes

$DS_ENVDEF (no arguments)

This macro defines the symbols that can be used for the ENV argument of the $DS_BGNMOD macro. The $DS_BGNMOD macro invokes the $DS_ENVDEF macro.   The following symbols are defined:

        CEP_FUNCTIONAL
        CEP_REPAIR
        SEP_FUNCTIONAL
        SEP_REPAIR

7.6.14 $DS_ERRDEF Define Error Call Arglist Offsets

$DS_ERRDEF (no arguments)

This macro defines symbolic arguments for the $DS_ERRDEV_X, $DS_ERRHARD_X, $DS_ERRSOFT_X, and $DS_ERRSYS_X macros. The following symbols are defined.

Symbol	Meaning
ERR$_NUM	error number
ERR$_UNIT	logical unit number
ERR$_MSGADR	header message address
ERR$_PRLINK	extended error print routine
ERR$_P1	additional data
ERR$_P2	additional data
ERR$_P3	additional data
ERR$_P4	additional data
ERR$_P5	additional data
ERR$_P6	additional data

7.6.15 $DS_HDRDEF Define Header Section Area

$DS_HDRDEF [gbl]

gbl = GLOBAL or LOCAL.

This macro defines absolute addressable symbols for the header section area (the area defined by the $DS_HEADER macro). The macro defines the following symbols:

Symbol	Meaning
L$L_HEADLENGTH	length of the header data block
L$L_ENVIRON	program environment
L$A_NAME	program name test address
L$L_REV	program revision level
L$L_UPDATE	diagnostic engineering patch order
L$A_LASTAD	first free location after program
L$A_DTP	test dispatch table pointer
L$L_DEVP	device type list pointer
L$L_UNIT	number of units that can be tested
L$A_DREG	device register contents table pointer
L$A_ICP	address initialize
L$A_CCP	cleanup code pointer
L$A_REPP	summary report code pointer
L$A_STATAB	statistics table pointer
L$A_ERRTYP	number of types of $ERRSOFT and $ERRHARD
L$A_SECNAM	list of section name addresses
L$A_TSTCNT	pointer to number of tests

7.6.16 $DS_HPODEF Define Hardware P-Table Entry Offsets

$DS_HPODEF (no arguments)

This macro defines offsets for items in the hardware P-table, allowing the programmer to access the items with their symbolic offsets.

NOTES

1. Only device independent offsets are defined.

2.

Symbol	Description
HP$Q_DEVICE	quadword descriptor of device name
HP$W_SIZE	total size of P-table
HP$B_DRIVE	unit number
HP$T_DEVICE	ASCII device name with leading "_", max length = 11 characters
HP$A_DEVICE	device address for this UUT
HP$A_DVA	address used to directly address another UUT through this device
HP$A_LINK	address of P-table for device linking this to the CPU
HP$W_VECTOR	primary interrupt vector for device
HP$T_TYPE	ASCIC hardware type, max length = 11 characters

7.6.17 $DS_PARDEF Parameter Definitions

$DS_PARDEF [gbl]

gbl = GLOBAL or LOCAL.

This macro defines the radix and exception mask arguments for the macros of the form $DS_ASKxxxx. The following symbols are defined.

        PAR$_BIN
        PAR$_OCT
        PAR$_DEV
        PAR$_HEX
        PAR$_NO
        PAR$_YES
        PAR$_NODEF
        PAR$_ATLO
        PAR$_ATHI
        PAR$_ATDEF

7.6.18 $DS_SCBDEF System Control Block Definitions

$DS_SCBDEF [gbl]

<u>gbl</u> =   GLOBAL or LOCAL.

This macro defines symbols for the system control block vector offsets.

The following symbols are defined:

```
SCB$L_ZERO          SCB$L_MACHCHK       SCB$L_KNLSTK
SCB$L_POWER         SCB$L_OPCDEC        SCB$L_OPCCUS
SCB$L_ROPRAND       SCB$L_RADRMOD       SCB$L_ACCESS
SCB$L_TRANSL        SCB$L_TBIT          SCB$L_BREAK
SCB$L_COMPAT        SCB$L_ARITH         SCB$L_CHMK
SCB$L_CHME          SCB$L_CHMS          SCB$L_CHMU
SCB$L_SFTLVL1       SCB$L_SFTLVL2       SCB$L_SFTLVL3
SCB$L_SFTLVL4       SCB$L_SFTLVL5       SCB$L_SFTLVL6
SCB$L_SFTLVL7       SCB$L_SFTLVL8       SCB$L_SFTLVL9
SCB$L_SFTLVL10      SCB$L_SFTLVL11      SCB$L_SFTLVL12
SCB$L_SFTLVL13      SCB$L_SFTLVL14      SCB$L_SFTLVL15
SCB$L_TIMER         SCB$L_RXDB          SCB$L_TXDB
```

7.6.19 $DS_SECDEF Section Definitions

$DS_SECDEF <arg,arg,arg...>

<u>arg</u> =   section name.

This macro should be used by the diagnostic programmer to define the test sections in each program source module. Use this macro in all program modules except the header module.

> **NOTE**
> The section names used as arguments must
> appear in the same order as they do in
> the $DS_SECTION macro in the header
> module.

7.6.20 $DS_DEFDEL Delete Macro Expansion

$DS_DEFDEL (no arguments)

This macro is used to conserve memory space during assembly. When this macro call follows a group of the symbol definition macros, it deletes the macro expansions from memory after the assembler has defined the corresponding symbols.

In addition to the utility macros, the diagnostic macro library in the supervisor provides a variety of macros that call supervisor routines to perform functions. Unlike the utility macros, the supervisor service macros generally return status codes. Furthermore, the supervisor service calls return control to the location following the instruction that called the supervisor service routine. Generally, they do not modify the flow of the program. Eight types of supervisor service macros are available.

> Program control service macros
> Channel service macros
> Memory management service macros
> Delay service macros
> Error message service macros
> Program-operator dialogue service macros
> System control service macros
> Hardware P-table access service macro

8.1   CODING SUPERVISOR SERVICE MACROS

The supervisor service macros take the form $DS_xxxx_x. The $DS_ prefix signifies the diagnostic supervisor. The suffix, _x, shows that the macro may end in any of four ways, depending on the function required.

> _DEF--generate symbols and offsets
> _L--generate an argument list
> _S--call the service with CALLS
> _G--call the service with CALLG

The RELBUF service (release buffer space) is typical. The following four macros relate to this service:

> $DS_RELBUF_DEF
> $DS_RELBUF_L
> $DS_RELBUF_G
> $DS_RELBUF_S

The listings for these macros are available in DIAG.LST. Paragraph 8.5.4 provides an explanation of the service performed and the arguments required. That section gives the following format for the RELBUF service.

> $DS_RELBUF_x pagcnt, [retadr], [region]

The suffix _x following $DS_RELBUF shows that the RELBUF service may be called with _G or _S suffixes. The arguments that follow it show the positional dependence and the keyword names of each argument. The arguments enclosed in square brackets are optional. They may be omitted. When the programmer does not specify an optional argument, the macro supplies a default value. The argument list supplied to any supervisor service must have a format in memory like that shown in Figure 8-1.

```
┌─────────────────────────┬───┐
│                       ┆ │ 3 │
├─────────────────────────┴───┤
│          PAGCNT             │
├─────────────────────────────┤
│          RETADR             │
├─────────────────────────────┤
│          REGION             │
└─────────────────────────────┘
```
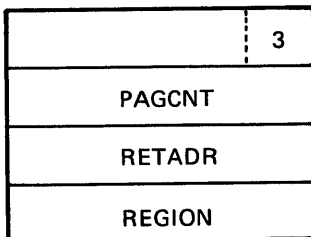
TK-1884

Figure 8-1   Memory Format for RELBUF Supervisor
             Service Macro Arguments

All arguments are longwords. However, they should be coded as
symbols. The symbols must specify addresses or data, except for
the first longword. The first longword must always contain, in its
low-order byte, the number of arguments in the remainder of the
list.

8.1.1 $DS_name_G Macro Call Format
The $DS_name_G macro call format is useful when a supervisor
service routine is to be called repeatedly with the same (or
nearly the same) argument list, or when there is no argument list.
This macro format generates a CALLG instruction. It requires the
programmer to construct an argument list elsewhere in the program
and to specify the address of the list as an argument to the
$DS_name_G macro call, as follows:

        $DS_name_G label

The programmer should use the $DS_name_L macro format to generate
the list of arguments referenced by LABEL. In general, the
$DS_name_L macro format should be used in the data section of the
program, in the first module, as shown in Example 8-1.

```
LIST::
        $DS_RELBUF_L -              ; arglist for RELBUF
            PAGCNT = 10, -          ; 10 pages
            RETADR = BUFLIM, -      ; return virtual location
            REGION = 0             ; Buffer is in P0 space.
.
.
.
BUFLIM:   .BLKL   2                 ; buffer address
```

Example 8-1   Use of the $DS_name_L Macro Format
              to Build an Argument List

The supervisor service associated with that argument list can then be called with a $DS_name_G macro format, as shown in Example 8-2.

```
$DS_RELBUF_G LIST          ; Call RELBUF supervisor service
                           ; with arguments specified in
                           ; LIST.
```

Example 8-2 Calling a Supervisor Service with
the $DS_name_G Macro Format

The argument, LIST, enables the macro to pass the items following the LIST label in the data section of the program to the supervisor service being called.

The VAX-11 macro assembler expands the $DS_name_G macro format as shown in Example 8-3.

```
$DS_RELBUF_G LIST                    ; Call RELBUF.
             CALLG LIST, @#DS$RELBUF
```

Example 8-3 Expansion of $DS_name_G Macro Form

Sometimes it is necessary to alter one or more of the arguments in the list to be passed to the supervisor service, leaving the rest unchanged. For example, the programmer may want to use the RELBUF supervisor service another time with a page count argument of 7. Several steps are required.

First, the programmer must use the $DS_name_DEF macro in the data section of the test module that contains the supervisor service call. This macro provides a set of symbols that describe offsets from the base address of the argument list. The $DS_RELBUF_DEF macro creates the following symbols and offsets:

```
        RELBUF$_NARGS  = 3
        RELBUF$_PAGCNT = 4
        RELBUF$_RETADR = 8
        RELBUF$_REGION = 12
```

Second, the programmer must replace the old value in the argument list with a new value, as shown in Example 8-4.

```
MOVAL   LIST, R2                    ; base address of arglist
MOVL    #7, RELBUF$_PAGCNT(R2)      ; Replace the page count in
                                    ; LIST with the number 7.
```

Example 8-4 Modification of an Argument List

Third, the supervisor service can then be called with the
$DS_name_G macro. The modified argument list will be passed.
Example 8-5 shows the entire process.

```
; Header Module

LIST::
        $DS_RELBUF_L -                  ; argument list for RELBUF
            PAGCNT = 10, -              ; 10 pages to be released
            RETADR = BUFLIM, -          ; Return virtual location here.
            REGION = 0                  ; Buffer is in P0 space.
        .
        .
        .
BUFLIM: BLKL 2                          ; address of released buffer
        .
        .
        .
```

```
; Test Module

        $DS_RELBUF_DEF                  ; Create offset symbols for
    .                                   ; argument list for RELBUF.
    .
    .

        $DS_RELBUF_G LIST               ; Call RELBUF supervisor
                                        ; service with unmodified
    .                                   ; argument list.
    .

        MOVAL LIST, R2                  ; base address of argument list
                                        ; for RELBUF
        MOVL #7, RELBUF$_PAGCNT (R2)
                                        ; Replace the page count in
                                        ; list with the number 7.
    .
    .

        $DS_RELBUF_G LIST               ; Call RELBUF supervisor
                                        ; service with modified
                                        ; argument list.
```

<p align="center">Example 8-5   Uses of $DS_name_G, $DS_name_L,
and $DS_name_DEF Macro Formats</p>

8.1.2 $DS_name_S Macro Call Format

The $DS_name_S macro call format is useful when a supervisor
service routine is to be called infrequently. It is also useful
when the supervisor routine is to be called repeatedly, but with a
different argument list each time. This macro format generates a
CALLS instruction. It requires the programmer to construct the
argument list as a part of the macro call. At assembly time, the
macro is expanded to create code that pushes the argument list
on the stack during program execution.

Arguments for the $DS_name_S macro format can be specified in either of two ways.

1.   Keywords may be used to describe the arguments. All keywords must be followed by an equal sign (=) and the value of the argument, as shown in Example 8-6. Default values will be supplied, by the macro, for unspecified arguments.

```
$DS_RELBUF_S -                    ; Call RELBUF service.
     REGION = 1, -                ; Buffer is in P1 space.
     PAGCNT = 12                  ; 12 page buffer
```

Example 8-6   Use of the $DS_name_S Macro Format with Keywords

Notice that the order of arguments has changed and that the RETADR argument is omitted altogether.

2.   Position, with omitted arguments indicated by doubled commas in the argument position, may be used to describe the arguments. Commas for optional trailing arguments may be omitted. Example 8-7 shows how position should be used.

```
$DS_RELBUF_S #12, , #1     ; Call RELBUF supervisor
                          ; service with PAGCNT = 12.
                          ; RETADR = default value
                          ; REGION = 1
```

Example 8-7 $DS_name_S Macro Format with Arguments
Specified by Position

These two macro calls (Examples 8-6 and 8-7) are expanded in the same way by the assembler. In each case, the arguments are pushed on the stack as shown in Example 8-8.

```
PUSHL #1                  ; region
PUSHAQ RETADR             ; pointer to buffer description
PUSHL #12                 ; page count
CALLS #3, @#DS$RELBUF
```

Example 8-8   Expansion of the $DS_RELBUF_S Macro

Figure 8-1 shows the arrangement of data on the stack.

8.2   RETURN STATUS CODES

Most of the supervisor services provide a return status code in general register R0, when execution of the service has been completed. When bit 0 of R0 is set, it indicates that the supervisor service has been completed successfully. The low order three bits, taken together, indicate the severity of the error, as follows:

0	warning
1	success
2	error
3	reserved
4	severe error
5-7	reserved.

The remaining bits in the low-order word of R0 (bits 3-15) classify the particular return condition. Return status codes for each supervisor service macro are indicated in the entries for each macro. Each return status code has a unique symbolic name in the format, DS$_code, where code is a mnemonic describing the return condition. For example:

DS$_NORMAL indicates successful return.

DS$_IHWE indicates an initial hardware error on a channel call. If this status code is returned, it shows that the function has not been performed, because the adapter hardware status was found to be in error.

When you code a supervisor service macro, you should examine the explanation of the macro in order to determine what return codes must be checked.

The program can test for successful completion of a supervisor service call by checking the low-order bit of R0, as shown in Example 8-9.

```
    BLBC R0, ERRLABEL                 ; error if low bit clear
```

Example 8-9   Testing for Successful Return Status

The error checking routine at ERRLABEL should check R0 for specific values with a compare instruction as shown in Example 8-10.

```
    CMPW R0, DS$_IHWE                 ; initial hardware error?
```

Example 8-10   Identifying the Return Status Code

8.3 PROGRAM CONTROL SERVICE MACROS

The program control service macros enable the programmer to control the program flow during execution. However, with the exception of the $DS_ENDPASS_x macro, these supervisor service macros do not directly change the course of the program.

8.3.1 $DS_CNTRLC_x Control-C

This macro calls a supervisor routine that enables the diagnostic program to intercept the next Control C typed by the operator. It provides the address of a routine for the supervisor to call on Control C. The routine sets an Enable flag. This flag remains set until a Control C is typed or until the flag is canceled by a call with a routine address of zero. If a routine has been specified when the next Control C is typed, the Enable flag is cleared and the specified routine is called.

$DS_CNTRLC_x label

label = the address for transfer of program control.

Return Status Codes
SS$_NORMAL: Service completed successfully.

```
$DS_CNTRLC_S CNTRLC_HANDLER
```

Example 8-11 $DS_CNTRLC_x Macro Usage

In Example 8-11, the $DS_CNTRLC_S macro call causes a transfer of program control to a Control C handler, which can call the summary routine and then rearm itself. This feature is useful in programs that run for a long time.

8.3.2 $DS_INLOOP_x Check for a Loop

This macro calls a supervisor routine that tests whether or not the program is looping on an error. The routine sets the low bit of R0 if the program is in a loop. It clears the low bit of R0 if the program is not in a loop.

$DS_INLOOP_x (no arguments)

Return Status Codes
DS$_NORMAL: The program is in a loop.
DS$_ERROR: The program is not in a loop.

```
            .SBTTL        TEST 5
            $DS_BGNTST
            $DS_BGNSUB
LABEL_2:
            ; (test code 1)
            $DS_BNERROR LABEL_3
            $DS_INLOOP_S
            BLBS R0, LABEL_3                    ; Program is in a loop.
            $DS_ABORT
LABEL_3:
            $DS_CKLOOP LABEL_2
            .
            .
            .
            $DS_ENDSUB
```

Example 8-12   Use of the $DS_INLOOP_x Macro

In Example 8-12 the $DS_INLOOP macro call enables the program to abort on error detection unless the program is in an error loop.

8.3.3 $DS_SUMMARY_x Execute the Summary Report Section
This macro calls a supervisor service routine. This, in turn, calls the user summary report routine to print out a summary message. The summary report routine is normally called indirectly from the initialization code with the $DS_ENDPASS_x macro at the completion of program execution.

$DS_SUMMARY_x (no arguments)

8.3.4 $DS_ENDPASS_x Indicate to the Supervisor that a Logical Pass is Completed
This macro calls a routine in the supervisor to mark the end of a pass of the diagnostic program. If enough passes have been run, the routine causes execution of the user summary and cleanup routines.

$DS_ENDPASS_x (no arguments)

Return Status Codes: None.

Example 8-13 shows how the $DS_ENDPASS_x macro can be coordinated with the $DS_BPASS0 macro in the initialization code.

```
1           .SBTTL INITIALIZATION CODE
2           $DS_BGNINIT
3           $DS_BNPASS0 10$                ; First time through?
4           CLRL LOG_UNIT                  ; Yes, start with first unit.
5           BRB 20$                        ; Begin device
                                           ; initialization.
6   10$:    INCL LOG_UNIT                  ; No, start with next unit.
7           CMPL DSA$GL_UNITS, LOG_UNIT
8                                          ; Is last unit tested?
```

```
9           BNEQ 20$                      ; No.
10          $DS_ENDPASS G                 ; Yes, check for last pass.
11          CLRL LOG_UNIT                 ; Last pass not done.
12  20$:    $DS_GPHARD . . .              ; Get P-table address.
13          $ASSIGN . . .                 ; Assign a channel.
14              .
15              .
16              .
17          CLRQ    CSRW                  ; Initialize device under
                                          ; test.
```

Example 8-13   Use of the $DS_ENDPASS Macro Call

Notice that the $DS_ENDPASS_x macro call is executed at the end of each pass. Notice also that the $DS_ENDPASS_x macro is not executed between passes.

8.4 CHANNEL SERVICE MACROS

The channel service macros are designed to promote the transportability of diagnostic programs across various VAX implementations. These macros enable level 3 diagnostic programs to interface with the channel adapters in a general way. The macros also make design changes in the channel adapters as transparent as possible. The channel service macros are not available to level 2 and 2R diagnostic programs.

8.4.1 $DS_CHANNEL_x Channel Service

This macro calls a supervisor routine that provides a channel adapter interface service, enabling general control over the hardware status of the channel adapter. Any of ten functions (such as purge or clear) may be specified. The call accomplishes the function specified by writing or reading bits on the adapter registers.

$DS_CHANNEL_x unit, func, [vecadr], [stsadr]

unit   =   the logical unit number.

func   =   the function code specifying the operation to be performed. The code is expressed symbolically. The function argument should be preceded by a number sign (#).

vecadr =   the entry point address for interrupt service when an interrupt occurs. Required for the CHC$ENINT function.

stsadr =   the address of a quadword that stores adapter status when the CHC$_STATUS function is used and when an interrupt occurs. This argument is required when the CHC$_STATUS and CHC$_ENINT functions are specified for the channel call.

$DS_CHANNEL Functions:

CHC$_INITA   Initialize channel adapter
    Return status codes:
    DS$_NORMAL: Service successfully completed.

CHC$_INITB   Initialize device bus (UBA) only
    Return status codes:
    DS$_NORMAL: Service successfully completed.

CHC$_ABORT   Adapter abort (MBA only)
    Return status codes:
    DS$_NORMAL: Service successfully completed.

    DS$_LOGIC: Bit set/clear failure. DTABT did not set (MBA).
    ABORT did not clear (MBA).

CHC$_PURGE   Purge data path (UBA only)
    Return status codes:
    DS$_NORMAL: Service successfully completed.

    This function purges the data path specified by the last
    $DS_SETMAP_x macro call.

CHC$_ENINT   Enable interrupts
    Return status codes:
    DS$_NORMAL: Service successfully completed.

    DS$_IHWE: Initial hardware error. An adapter error condition
    was found before the function was performed.

    DS$_LOGIC: Bit set/clear failure. The Interrupt Enable bit
    failed to set.

    DS$_IVVECT: An invalid vector was found by $SETVEC.

CHC$_DSINT   Disable interrupts
    Return status codes:
    DS$_NORMAL: Service successfully completed.

    DS$_IHWE: Initial hardware error. An adapter error condition
    was found before the function was performed.

    DS$_LOGIC: Bit set/clear failure. The Interrupt Enable bit
    failed to clear.

    DS$_IVVECT: An invalid vector was found by $CLRVEC.

    CHC$_CLEAR   Clear adapter status
        Return status codes:
        DS$_NORMAL: Service successfully completed.

        DS$_LOGIC: Bit set/clear failure. One of the status bit(s)
        failed to clear.

CHC$_STATUS   Request adapter status
    Return status codes:
    DS$_NORMAL: Service successfully completed.

CHC$_SETDFT   Set defeat parity (UBA only)
    Return status codes:
    DS$_NORMAL: Service successfully completed.

CHC$_CLRDFT   Clear defeat parity (UBA only)
    Return status codes:
    DS$_NORMAL: Service successfully completed.

NOTES

1. The interrupt enable function (CHC$_ENINT) will enable adapter interrupts and, in the case of the UBA, device interrupts. After Unibus initialization or UBA initialization, perform a clear-function before enabling interrupts.

2. Adapter status, returned in response to the CHC$_STATUS function, is stored in a location specified by the argument STSADR, as shown in Note 3.

3. Returned status description: Two longwords of status are returned for each request status function (CHC$_STATUS) of the channel call. This status, along with certain specific interrupt information, is also supplied on all interrupts that are to be passed to a program that has enabled interrupt processing. The two longwords returned have the following format:

STATUS 1	
RVR	STATUS 2

STATUS 1 =   Adapter status as defined in Note 4.

STATUS 2 =   One word of interrupt status as defined in Note 5.

RVR =   Receive Vector Register for those devices that interrupt through the UBA.

4.   Adapter Status Definitions (Status 1)
          Symbol   Definition

          CHS$M_SYSERR   System error (category)
          CHS$M_CHNERR   Channel error (category)
          CHS$M_DEVERR   Device error (category)
          CHS$M_PGMERR   Program error (category)
          CHS$M_PGMHDE   Program error (hardware
                                          detected)
          CHS$M_DEVBUS   Unibus/Massbus error
          CHS$M_DEVTO   Device time-out
          CHS$M_CHNDPE   Channel data parity error
          CHS$M_CHNMPE   Channel memory parity error
          CHS$M_CHPFOT   Channel power fail/overtemp
          CHS$M_SYSMEM   System memory error
          CHS$M_SYSSBI   System SBI error
          CHS$M_MBAEXC   Massbus exception
          CHS$M_MBANED   MBA non-existent drive
          CHS$M_MBADTB   MBA data transfer busy
          CHS$M_MBADTC   MBA data transfer complete
          CHS$M_MBACPE   MBA control parity error
          CHS$M_MBAWCK   MBA write check
          CHS$M_BUSIC   Bus init clear (deassertion)
          CHS$M_BUSINIT   Bus init (assertion)
          CHS$M_BUSPDN   Bus power down
          CHS$M_ERRANY   Any error category
                                          (CHSM_SYSERR, CHSM_CHNERR,
                                          CHSM_DEVERR, CHSM_PGMERR)

5.   Interrupt Status Definitions (Status 2)

          CHI$M_CHNINT   Channel interrupt
          CHI$M_DEVINT   Device interrupt
          CHI$M_IPL   Interrupt priority level

When CHC$_ENINT is used, $DS_CHIDEF should be used in the program
header module to define the status codes returned at STSADR.

A complete list of return status codes for $DS_CHANNEL_x follows:

     DS$_NORMAL: Service successfully completed.

     DS$_PROGERR: Program error encountered.

DS$_IHWE: Initial hardware error encountered. The adapter hardware status was found to be in error before performance of the requested function. The function will not be performed.

DS$_FHWE: Final hardware error status encountered. The adapter hardware status was found to be in error after performance of the requested function. The capability of the adapter to correctly operate is in question.

DS$_LOGIC: An adapter function that sets or clears an adapter status bit has failed. The capability of the adapter to correctly operate is in question.

DS$_IVVECT: An invalid vector has been given as an argument.

DS$_IVADDR: An invalid address has been given as an argument.

DS$_ERROR: An error has been found in trying to associate a hardware P-table with the logical unit argument.

Example 8-14 shows how the $DS_CHANNEL_x macro can be used to reset the Unibus channel (DW780).

```
.SBTTL INITIALIZATION CODE
.
.
.

    $DS_BGNINIT

.
.
.

    $DS_CHANNEL_S -                 ; channel call
        UNIT = LOG_UNIT, -          ; device under test
        FUNC = #CHC$_INITA          ; Reset UBA.
    $DS_BNERROR 10$                 ; Branch if successful.
        ; (error message)
    $DS_ABORT PROGRAM               ; Failure, abort program.
10$:
```

Example 8-14   Resetting the Unibus Channel
with the $DS_CHANNEL_x Macro

8.4.2 $DS_SETMAP_x Set Channel Adapter Mapping

This macro enables the diagnostic engineer to set up channel adapter mapping for I/O transfers.

$DS_SETMAP_x unit, func, phyadr, [mapbas], [bytcnt], [datpth]

<u>unit</u>   =   the logical unit number.

<u>func</u>   =   the function code symbol specifying the type of mapping to be performed. See the symbols listed below.

<u>phyadr</u> =   the address of a two longword array describing the physical buffer start and end addresses. Normally these will be the start and end addresses returned by a $DS_GETBUF_x macro call.

<u>mapbas</u> =   adapter map register select. For the UBA (DW780) this parameter corresponds to the upper 9 bits of the Unibus address (bits 17:09) to be used in the base address (BA) registers of the desired device. The number supplied for the MAPBAS parameter to the UBA should be in the range of 0 to 495 (decimal).

For the MBA (RH780) the MAPBAS parameter selects the current map register through bits 16:09 of the virtual map register. The default for MAPBAS is zero.

<u>bytcnt</u> =   the positive byte count (used for the (RH780) only). This field is ignored when the UBA (DW780) is used. However, the field is checked for validity regardless of the adapter type to be used.

<u>datpth</u> =   UBA (DW780) data path number (0-15). This field is ignored when the MBA (RH780) is used.

Function Argument Symbols

Symbol	Function
CHM$_FORWARD	Prime the adapter for a forward operation.
CHM$_REVERSE	Prime the adapter for a reverse operation.
CHM$_INVALIDATE	Invalidate all map entries.
CHM$_MAP	Set requested mapping.
CHM$_OFFSET	Mapping with byte offset.

CHM$_MFWDV	Invalidate all map entries, set requested mapping, prime the adapter for a forward operation.
CHM$_MFWDN	Do not invalidate any map entries, set requested mapping, prime the adapter for a forward operation.
CHM$_NFWDN	Do not invalidate any map entries, do not set mapping, prime the adapter for a forward operation.
CHM$_MREVV	Invalidate all map entries, set requested mapping, prime the adapter for a reverse operation.
CHM$_MREVN	Do not invalidate any map entries, set requested mapping, prime the adapter for a reverse operation.
CHM$_NREVN	Do not invalidate any map entries, do not set mapping, prime the adapter for a reverse operation.
CHM$_MFWDVO	Invalidate all map entries, set requested mapping with byte offset, (UBA only), prime the adapter for a forward operation.
CHM$_MFWDNO	Do not invalidate any map entries, set requested mapping with byte offset, (UBA only), prime the adapter for a forward operation.
CHM$_MREVVO	Invalidate all map entries, set requested mapping with byte offset, (UBA only), prime the adapter for a reverse operation.
CHM$_MREVNO	Do not invalidate any map entries, set requested mapping with byte offset, (UBA only), prime the adapter for a reverse operation.

Return Status Codes
DS$_NORMAL: Service successfully completed.

DS$_PROGERR: Program error. The buffer address is out of range, the base address is out of range (0 through 255-MBA, 0 through 495-UBA), or the specified byte count is too large.

DS$_IHWE: Adapter hardware error status was encountered before the mapping was performed. The error condition must be cleared before the mapping will be performed.

DS$_ERROR: An error has been found trying to associate a hardware P-table with the logical unit argument.

Example 8-15 shows how the set map service can be set up and executed.

```
          MOVZBL #1, CYLN                ; cyl = 1
          CLRL TRACK                     ; track = 0
          CLRL SECTOR                    ; sector = 0
          MOVZWL #256, WORD_COUNT        ; Specify word count for
                                         ; compare.

          ;
          ; Initialize disk drive.
          ;

          CLRL R3                        ; Clear index.
20$:      MOVQ #^XFFFF0000FFFF0000, -    ; Fill write buffer
          @WRITE_BUFFER[R3]              ; indexed by R3.

          AOBLSS #64, R3, 25$            ; Loop until done.


          ;
          ; Load disk address and map buffers.
          ;
          MOVW CYLN, RKDCYL(R2)          ; Load cylinder.
          MOVB TRACK, RKDA+1(R2)         ; Load track.
          MOVB SECTOR, RKDA(R2)          ; Load sector.
          MNEGW #256, RKWC(R2)           ; Word count = 256.
          MOVW #^01000                   ; Set transfer address to
                                         ; 1000 (8).

          CLRW RKBA(R2)                  ; Clear BA.
          $DS_SETMAP_S -
              UNIT = L$UNIT, -           ; device
              FUNC = #CHM$_MFWDV,-       ; Invalidate all map
                                         ; entries, set requested
                                         ; mapping, prime adapter
                                         ; for a forward operation.

              PHYADR = WRITE_BUFFER, -
              MAPBAS = #1, -             ; Unibus page 1
              DATPTH = #1                ; direct data path
```

```
          $DS_BNERROR 30$
          $DS_ERRSYS_S, L$UNIT, UBAMAP, -
          DUMP_UBA                              ; error reporting code
          $DS_ABORT TEST
30$:      ; Initiate NPR transfer,
          ; read from memory, write
          ; to disk.
          MOVW #WRITE, HOLD_CS1                 ; Define command
                                                ; to be executed.

          JSB @START_XFER                       ; Execute command.
          BLBS R0, 40$
          ; (error message)
40$:      ; (compare expected and received data)
```

Example 8-15 Use of the $DS_SETMAP_x Macro

In Example 8-15 the $DS_SETMAP_x macro enables the program to set up map registers on the channel adapter. Notice that a failure of this function is a system error and is considered to be fatal. After the transfer has been successfully set up, the disk drive will perform NPR read transfers to retrieve data from the WRITE_BUFFER in memory through the direct data path.

After successful completion of the NPR transfer, the disk drive under test should perform an interrupt. The interrupt service routine can then return control to the program.

8.4.3 $DS_SHOWCHAN_x Show Channel Registers

This macro calls a supervisor routine (DSX$SHOWCHAN) which displays on the operator terminal the configuration register contents and the status register contents for the channel adapter in use. If the status indicates errors that require the display of additional registers, those registers will also be displayed. For example, if the status register indicates an invalid map, the relevant map register will be displayed. The display for each register contains the register mnemonics, the physical address of the register, the register contents, and a mnemonic description of the register contents.

$DS_SHOWCHAN_x unit

unit = logical unit number.

Return Status Codes
DS$_NORMAL: Service successfully completed.

DS$_PROGERR: An error has been found while trying to associate a hardware P-table with the logical unit argument.

Example 8-16 shows how the $DS_SHOWCHAN_x macro should be used to display channel information.

```
CSRERR:      .ASCIC \ERROR IN CSR\
    •
    •
    •
MSG_TAG:     $DS_BGNMESSAGE
             $DS_PRINTB_S...                   ; error description
             $DS_SHOWCHAN -                    ; Display channel
                 UNIT = LOG_UNIT               ; information.
             $DS_PRINTB_S...                   ; Display device CSR.
             $DS_PRINTB_S...                   ; Display device SR status
                                                 register.

                 •
                 •
                 •
             $DS_ENDMESSAGE
             •
             •
             •

; TEST N
             $DS_BNERROR 10$                   ; Is there an error?
             $DS_ERRHARD_S -                   ; Yes, print.
                 UNIT = LOG_UNIT, -
                 MSGADR = CSRERR, -
                 PRLINK = MSG_TAG
10$:         ; continue with test
```

Example 8-16   Use of the $DS_SHOWCHAN_x Macro

In this example, the $DS_SHOWCHAN_x macro is incorporated in a print subroutine called through the PRLINK parameter in the $DS_ERRHARD_x macro in Test N (refer to Paragraph 8.7.1.3).

8.5   MEMORY MANAGEMENT SERVICE MACROS

All memory management for the diagnostic system is provided by either the supervisor or VMS. These memory management services ensure system integrity and operational consistency across the various operating environments and processor types. In the stand-alone environment, memory management is normally off. In the VMS environment, memory management is totally controlled by VMS and is always on.

The supervisor provides services that allocate memory, both virtual and physical. Physical memory allocation is restricted to level 3 diagnostic programs.

8.5.1 $DS_MMON_x Turn Memory Management On

This macro calls a supervisor routine (DSX$MMON) that turns memory management on. Use this macro in level 3 programs only.

$DS_MMON_x (no arguments)

Return Status Codes: None.

8.5.2 $DS_MMOFF_x Turn Memory Management Off

This macro calls a supervisor routine (DSX$MMOFF) that turns memory management off. Use this macro in level 3 programs only.

$DS_MMOFF_x (no arguments)

Return Status Codes: None.

8.5.3 $DS_GETBUF_x Get Virtual Memory Space

This macro calls a supervisor service routine (DSX$GETBUF) to obtain memory space for buffer areas. The service allocates memory space at the logical end of the program (Figure 5-3). In the standalone environment, the physical memory allocation is contiguous. The routine also calls DS$BREAK to check for Control C.

$DS_GETBUF_x pagcnt, [retadr], [phyadr], [region]

pagcnt = the number of pages of memory desired.

retadr = the address of a 2 longword array to receive the virtual buffer limits. .

phyadr = the address of a 2 longword array to receive physical start and end addresses.

region = that part of memory to which the buffer is to be allocated. 0 = (default) P0 space. 1 = P1 space. 2 = system space.

Return Status Codes
SS$_NORMAL: Service successfuly completed.
SS$_ILLPAGCNT: Page count is less than 1
SS$_VASFULL: Virtual address space full.

You should use $DS_GETBUF_x, together with $DS_RELBUF_x, to obtain memory as it is needed for specific tests. When large buffers are needed, this service enables the programmer to avoid loading a large block of zeros at the time that the program is loaded into memory.

8.5.4 $DS_RELBUF_x Release Buffer Space

This macro calls a supervisor routine to release a buffer area in virtual memory from program control.

$DS_RELBUF_x pagcnt, [retadr], [region]

pagcnt = the number of pages.

retadr = the address of a 2 longword array to receive de-allocated buffer limits.

region = the space from which the buffer is to be released. 0 = (default) PØ space. 1 = P1 space. 2 = system space.

Return Status Codes
SS$_NORMAL: Service successfully completed.

DS$_FRAGBUF: Buffer was not contiguous.

SS$_ILLPAGCNT: Page count is less than 1.

SS$_PAGOWNVIO: Page owned by a more privileged access mode.

8.6   DELAY SERVICE MACROS

A diagnostic program can delay itself a specified number of microseconds or milliseconds through the use of a delay service call. For example, if your program is testing a Unibus device and you suspect the possibility of a Unibus timeout, you can use a delay service macro. You should specify the greatest allowable time as an argument. When the return from the call is made, the service returns the increment of the time delay not used via the optional RETTIM argument.

You can abort the operation of the delay macros with the $DS_CANWAIT_x macro.

8.6.1 $DS_WAITUS_x Microsecond Delay

This macro calls a supervisor routine that suspends program operation for a number of microseconds equal to ten times the number specified.

$DS_WAITUS_x time, [rettim]

time   =   the number of 10 microsecond units.

rettim =   the longword address to receive unused time in 10 microsecond units.

Return Status Codes
DS$_NORMAL: Service successfully completed.

DS$_ICERR: Interval clock error.

DS$_PROGERR: Negative (or less than overhead) time interval specified.

SS$_QUOTA: Multiple time function error.

Example 8-17 shows how you can set up a delay with $DS_WAITUS_x while waiting for completion of a Unibus read function.

```
        MOVL BASE_ADR, R2              ; Get RKCS1 address.
        .
        .
        .
        MOVW R3, (R2)                  ; Write RKCS1.
        MOVZWL (R2), R3               ; Read RKCS1.
        $DS_WAITUS_S #10             ; Stall for 100 us for an event
                                      to occur.
```

Example 8-17   Use of the $DS_WAITUS_x Macro

NOTE
Because of overhead, the minimum delay is
about 100 microseconds.

8.6.2 $DS_WAITMS_x Millisecond Delay

This macro calls a supervisor routine that suspends program execution for a number of milliseconds equal to ten times the number specified.

$DS_WAITMS_x time, [rettim]

time   =   the number of 10 ms units.

rettim =   the longword address to receive unused time in 10 ms units.

Return Status Codes

$DS_NORMAL: Service successfully completed.

DS$_ICERR: Interval clock error.

DS$_PROGERR: Negative time interval specified.

SS$_EXQUOTA: Multiple time function error.

8.6.3 $DS_CANWAIT_x Cancel Wait

This macro calls a routine in the supervisor that cancels the effect of a previous $DS_WAITMS_x or $DS_WAITUS_x macro by invoking the $WAKE_x system service macro.

$DS_CANWAIT_x (no arguments)

Return Status Codes
DS$_NORMAL: Service successfully completed.

NOTE
This macro normally occurs in an
interrupt service routine.

You should coordinate the $DS_CANWAIT_x macro with one of the delay service macros to determine whether or not a timeout has occurred. For example, in an interrupt service routine, the program could delay longer than required for the interrupt to occur. When the interrupt does occur, the interrupt service routine could set a flag bit and execute the $DS_CANTIM_x macro call. Control would then return to the supervisor, which would, in turn, return control to the diagnostic program. However, if the flag bit were cleared upon return to the program, the program would know that an interrupt timeout had occurred.

Notice that the actual delay time achieved may be longer than the time requested. This is especially true of level 2 and level 2R programs, since other programs may be competing for system resources in the VMS environment.

Example 8-18 shows how you may coordinate the $DS_WAITMS_x macro with $DS_CANWAIT_x.

Test routine.

```
START_XFER::
        PUSHR #^M<R3, R4>              ; Save registers.
        MOVZBL #1, R0                  ; Initialize return status
                                       ; for optimistic start.
        MOVZWL DRVTYP, R4              ; Set up drive type.
        MOVB HOLD_CS1, R4             ; OR in command.
        MOVW R4, RKCS1(R2)            ; Load command and start
                                       ; transfer.
        $DS_WAITMS_S #10              ; Wait 100 ms.
        BLBS DN_FLG, 10$             ; Did an interrupt occur?
        $DS_ERRHARD...                ; No, timeout.
10$:    ; Yes, interrupt occurred, check data transferred.



Interrupt service routine.

        BBC #CHNINT, UBASTATUS$W_2,20$
```

```
                                              ; Branch if not a UBA
                                              ; interrupt.
               $DS_ERRSYS...
               $DS_ABORT PROGRAM
20$:           BBS DEVINT, UBASTATUS$W_2,30$
                                              ; Branch if device
                                              ; interrupt.

               $DS_ERRSYS_S...
               $DS_ABORT PROGRAM
30$:           $DS_CANWAIT_S
               MOVB #1, DN_FLG                ; Set interrupt done flag.
               .
               .
               .
               REI
```

Example 8-18   Coordination of the $DS_WAITMS_x
and $DS_CANWAIT_x Macros

In Example 8-18, the test routine starts a non-processor request
(NPR) transfer. The $DS_WAITMS macro then causes the program to
delay for 100 ms.

When the device completes the transfer required, it causes an
interrupt. The interrupt service routine fields the interrupt. The
interrupt service routine then cancels what remains of the delay,
sets a flag to indicate that the interrupt occurred, and then
returns control to the test routine. The test routine, in turn,
checks the flag to determine whether or not the interrupt did
occur.

8.7   ERROR MESSAGE SERVICE MACROS

The VAX diagnostic system provides three categories of error
information. These correspond to the three inhibit error message
flags (IE1, IE2, IE3) in the supervisor command language.

First, both the supervisor and the diagnostic program provide the
header. The header includes the program name, version and update;
the test, subtest, and error numbers; the error type; the device
under test; and a brief message.

Second, the diagnostic program provides basic information. The
basic information includes a description of the error, the
contents of the device registers, and expected and received data
patterns.

Third, the diagnostic program provides extended error information
to describe certain program conditions and/or hardware conditions.
The type of information printed varies with the program, but might
include statistics such as the number of bytes transferred.

If the Bell flag is set, the supervisor will ring the bell for
each error that the program encounters.

8-23

In addition, a convert register information macro is available to print subroutines. This enables the programmer to specify the mnemonics of failing register bits.

8.7.1   Header Information Message Macros

You can use four macros to request the printing of header information. These macros are identical in format, but you should use each to report a particular type of error. Each of the macros calls a supervisor service routine. The routine produces a three line error message for the operator. The message shows the program title and version number, pass number, test and subtest numbers, error number, and time stamp. In addition, the service routine prints a brief message specified by the programmer.

Example 8-19 shows the format for the header information for each of these macros.   Example 8-20 shows a typical error message header printout.

```
******** PROGNAME -- VER.UPD. ********

PASS #   TEST #   SUBTEST #   ERROR #   daytim

ERRTYP -- UUT -- message
```

Example 8-19   Error Message Header Format

```
********   ZZ-ESRCA RPOX/DCL DIAGNOSTIC - 4.1   ********
PASS 1    TEST 1    SUBTEST 0    ERROR 2    10-MAR-1978    08:26:20.26
DEVICE  FATAL  WHILE  TESTING  DBA0:    CONTROL  BUS  PARITY  ERROR
DETECTED
```

Example 8-20   Sample Error Message Header Printout

The programmer must specify the address of a print routine as the PRLINK parameter (Paragraph 8.7.1.1) if supplemental information must be printed. Otherwise, if the operator has set the Halt flag, no error message will be printed.

Note that you should use a Print macro within the print routine to print the basic or extended information.

Note also that the IE1 flag, when set, inhibits header messages.

8.7.1.1 $DS_ERRSYS_x Print System Fatal Error Header Information

You should use this macro after detecting a system fatal error.

$DS_ERRSYS_x [num], [unit], [msgadr], [prlink], [p1--6]

num   =   the unique error number within the current subtest. NUM is initialized by the $DS_BGNTEST and $DS_BGNSUB macros and automatically sequenced when not specified. The automatic sequencing of this parameter also includes all its default uses in the macros $DS_ERRHARD_x, $DS_ERRSOFT_x, and $DS_ERRDEV_x.

unit   =   the logical unit number of the unit under test.

msgadr =   the address of a counted ASCII string. This message is
            included in the third line of the error header message.
            It should be a brief description of the error or a
            module call out message.

prlink =   the address of an error reporting routine. This is the
            address of a closed routine to print supplemental
            information about the error that has occurred. The error
            reporting code at this address must be surrounded by
            $DS_BGNMESSAGE and $DS_ENDMESSAGE. Execution of this
            section of code is not contingent on the Halt On Error
            flag.

p1--6   =   one to six optional parameters that may be passed to the
            error print routine. If specified, they must be used in
            sequence.

Return Status Codes: None.

NOTES
1. **R2 through R11 are preserved within
    this service by the supervisor and
    are intact when the call to PRLINK
    is executed.**

2. **$DS_ERRSYS_x may not be used
    between subtests.**

The failure of a channel call is a system fatal error. You should
follow it with a $DS_ERRSYS_x macro call and, generally, a
$DS_ABORT macro call, as shown in Example 8-21.

```
        .SBTTL INIT
        $DS_BGNINIT
        .
        .
        .
        $DS_CHANNEL_S #0, #CHC$_INITA
                                    ; Initialize the UBA.
        $DS_BNERR 10$               ; Is there an error?
        $DS_ERRSYS_S -              ; Yes, a system fatal
                                    ; error.
            UNIT = 0,-
            MSGADR = UBA_INIT_ERR,- ; message
            POINTER = DUMP_UBA      ; error reporting code
        $DS_ABORT PROGRAM           ; Terminate program
                                    ; execution.
10$:        ; Continue with program.
```

Example 8-21   Use of the $DS_ERRSYS_x Macro

8.7.1.2 $DS_ERRDEV_x Print Device Fatal Error Header Information
You should use this macro after detecting a device fatal error.

$DS_ERRDEV_x [num], [unit], [msgadr], [prlink], [p1--6]

num   =   the unique error number within the current subtest. NUM
            is initialized by the $DS_BGNTEST and $DS_BGNSUB macros
            and automatically sequenced when not specified. The
            automatic sequencing of this parameter also includes all
            its default uses in the macros $DS_ERRHARD_x,
            $DS_ERRSOFT_x, and $DS_ERRSYS_x.

unit   =   the logical unit number of the unit under test.

msgadr =   the address of a counted ASCII string. This message is
            included in the third line of the error header message.
            It should be a brief description of the error or a
            module call out message.

prlink =   the address of an error reporting routine. This is the
            address of a closed routine to print supplemental
            information about the error that has occurred. The error
            reporting code at this address must be surrounded by
            $DS_BGNMESSAGE and $DS_ENDMESSAGE. Execution of this
            section of code is not contingent on the Halt On Error
            flag.

p1--6   =   one to six optional parameters that may be passed to the
            error print routine. If specified, they must be used in
            sequence.

Return Status Codes: None.

NOTES
1. R2 through R11 are preserved within
    this service by the supervisor and
    are intact when the call to PRLINK
    is executed.

2. $DS_ERRDEV_x may not be used
    between subtests.

A fatal device error is one that prevents further testing of the
device. Following the error detection and message, the programmer
may want to drop the testing of the device under test and proceed
to the next device to be tested, or the programmer may want to
abort the program.

8.7.1.3 $DS_ERRHARD_x Print Hardware Error Header Information – You should use this macro after detecting a hardware error.

$DS_ERRHARD_x [num], [unit], [msgadr], [prlink], [p1--6]

num   =   the unique error number within the current subtest. NUM is initialized by the $DS_BGNTEST and $DS_BGNSUB macros and automatically sequenced when not specified. The automatic sequencing of this parameter also includes all its default uses in the macros $DS_ERRDEV_x, $DS_ERRSOFT_x, and $DS_ERRSYS_x.

unit   =   the logical unit number of the unit under test.

msgadr =   the address of a counted ASCII string. This message is included in the third line of the error header message. It should be a brief description of the error or a module call out message.

prlink =   the address of an error reporting routine. This is the address of a closed routine to print supplemental information about the error that has occurred. The error reporting code at this address must be surrounded by $DS_BGNMESSAGE and $DS_ENDMESSAGE. Execution of this section of code is not contingent on the Halt On Error flag.

p1--6   =   one to six optional parameters that may be passed to the error print routine. If specified, they must be used in sequence.

Return Status Codes: None.

NOTES

1.   R2 through R11 are preserved within this service by the supervisor and are intact when the call to PRLINK is executed.

2. $DS_ERRHARD_x may not be used between subtests.

The $DS_ERRHARD_x macro is the error message macro most commonly used upon error detection. Example 8-22 is typical.

```
10$:
         MOVW #^X5068, @DZCSR            ; Write to CSR.
         MOVW @DZCSR, CSRW               ; Read CSR.
         XORW3 #^X5068, CSRW, XORW       ; Check data.
         BEQL 20$                        ; success
         $DS_ERRHARD_S -
              UNIT = LOG_UNIT, -         ; device under test
              MSGADR = REGERR            ; message address
20$:     $DS_CKLOOP
```

Example 8-22   Use of the $DS_ERRHARD_x Macro

8-27

VAX Diagnostic Design Guide

8.7.1.4 $DS_ERRSOFT_x Print Soft Error Header Information - When
a test fails initially but works correctly when retried, you
should use this macro.

$DS_ERRSOFT_x [num], [unit], [msgadr], [prlink], [pl--6]

num   = the unique error number within the current subtest. NUM
           is initialized by the $DS_BGNTEST and $DS_BGNSUB macros
           and automatically sequenced when not specified. The
           automatic sequencing of this parameter also includes all
           its default uses in the macros $DS_ERRHARD_x,
           $DS_ERRDEV_x, and $DS_ERRSYS_x.

unit   = the logical unit number of the unit under test.

msgadr = the address of a counted ASCII string. This message is
           included in the third line of the error header message.
           It should be a brief description of the error or a
           module call out message.

prlink = the address of an error reporting routine. This is the
           address of a closed routine to print supplemental
           information about the error that has occurred. The error
           reporting code at this address must be surrounded by
           $DS_BGNMESSAGE and $DS_ENDMESSAGE. Execution of this
           section of code is not contingent on the Halt On Error
           flag.

pl--6   = one to six optional parameters that may be passed to the
           error print routine. If specified, they must be used in
           sequence.

Return Status Codes: None.

                              NOTES
        1.   R2 through R11 are preserved within
             this service by the supervisor and
             are intact when the call to PRLINK
             is executed.

        2. $DS_ERRSOFT_x may not be used
             between subtests.

8.7.2   Information Print Macros
The four basic and extended information print macros all use the
same format. Each calls the extended print routine in the
supervisor to check a different set of the inhibit error message
flags before printing the message, as follows:

                         Flags checked

        $DS_PRINTB_x   IE1, IE2
        $DS_PRINTX_x   IE1, IE2, IE3
        $DS_PRINTS_x   IES
        $DS_PRINTF_x   none

Example 8-23 shows the three standard formats for basic error
messages.

8-28

```
1        REGISTER VALUE WRONG

         GOOD:     <EXPECTED-VALUE>
         BAD :     <ACTUAL-VALUE>
         XOR :     <XOR-VALUE>            ;<MNEMONICS OF XOR BITS>

2        REGISTER DUMP OF UNIT UNDER TEST

<REGISTER-0>          :  <REGISTER-VALUE><RADIX>      ;<MNEMONICS OF SET BITS>

                               .
                               .
                               .

<REGISTER-N>          :  <REGISTER-VALUE><RADIX>      ;<MNEMONICS OF SET BITS><SEE NOTE 1>

3        DATA COMPARE ERROR IN BUFFER

DEVICE STARTING ADDRESS:      <PHYSICAL STARTING ADDRESS ON UUT>
MEMORY BUFFER ADDRESS  :      <MEMORY ADDRESS>
RECORD SIZE            :      <LENGTH OF TRANSFER>
WORDS IN ERROR         :      <NUMBER OF WORDS THAT ARE BAD>

ADDRESS               GOOD                   BAD              XOR

<MEMORY ADDRESS>      <EXPECTED DATA>  <ACTUAL DATA>      <XOR OF GOOD AND BAD>

                               .
                               .
                               .

<MEMORY ADDRESS>      <EXPECTED DATA>        <ACTUAL DATA>      <XOR OF GOOD AND BAD><SEE NOTE 2>
```

Example 8-23 Standard Formats
for Basic Error Messages

NOTES

1. All mnemonics must be identical to the ones in the hardware engineering specification for the unit under test.

2. A maximum of eight memory locations will be dumped.

The extended print routine uses formatted ASCII output (FAO) to compile a string and then print it. You should use the $DS_PRINTB_x and $DS_PRINTX_x macros in the print routine (PRLINK argument), rather than in the test routines. In this way only one macro call (e.g., $DS_ERRHARD_x) must be made for each error detected in a test routine, as shown in Example 8-24.

```
Program data section:

        MSG5::
                .ASCIC \CSR IN ERROR\
        MSG6::
                .ASCIC \TRANSMIT READY IS CLEAR, SHOULD BE SET\

Print subroutine containing $DS_PRINTB_x macro:

PRINT_ROUTINE::
    $DS_BGNMESSAGE -
        REGMASK = <R2, R3, R4>
    MOVL ERR$_P1(AP), R2                 ; Save address of basic
                                         ; error message.
    $DS_PRINTB_S -
        FORMAT = (R2)                    ; Print basic message.
    .
    .
    .
    $DS_ENDMESSAGE

Test routine which calls the print subroutine:

10$:
    $DS_BGNSUB
    .
    .
    .
        ; transfer data and compare.
        $DS_BNERROR 20$                  ; Is there an error?
        $DS_ERRHARD_S -                  ; yes
            UNIT = LOG_UNIT, -           ; device under test
            MSGADR = MSG5, -             ; header message
                                         ; text
            PRLINK = PRINT_ROUTINE, -    ; basic error information
                                         ; print routine
            P1 = MSG6                    ; basic print message
    .
    .
    .
20$: $DS_CKLOOP 10$
```

Example 8-24 Use of the $DS_PRINTB_x Macro

In Example 8-24 the $DS_ERRHARD_S macro takes as arguments MSG5, PRINT_ROUTINE, and MSG6. The macro first calls a supervisor routine to produce a three-line error header message. The MSGADR argument (MSG5) points to an ASCII string that makes up a part of the header message. The PRLINK argument identifies the basic print routine, to which control passes from the supervisor routine. The print routine saves the P1 argument from the $DS_ERRHARD_S macro call in R2. The $DS_PRINTB_S macro in the print subroutine then causes MSG6 to be printed.

However, you will generally need to print a message containing information which varies, such as the value in a register or the mnemonic of a bit in error. In this case, instead of providing a complete message in the data section, you must provide a control string that includes FAO directives. Variable data and character strings can be represented with the FAO directives. Table 8-1 gives a summary of these directives.

Table 8-1   Summary of FAO Directives

Character String Substitution

Directive	Function	Parameter(s)*
!AC	Insert a counted ASCII string.	Address of the string; the first byte must contain the length.
!AD	Insert an ASCII string.	1. Length of string 2. Address of string
!AF	Insert an ASCII string. Replace all nonprintable ASCII codes with periods (.).	1. Length of string 2. Address of string
!AS	Insert an ASCII string.	Address of quadword character string descriptor pointing to the string.

Numeric Conversion (zero-filled)

!OB	Convert a byte to octal.	Value to be converted
!OW	Convert a word to octal.	to ASCII representation.
!OL	Convert a longword to octal.	
!XB	Convert a byte to hexadecimal.	For byte or word conversion, FAO uses
!XW	Convert a word to hexadecimal.	only the low-order byte or word of the longword

Table 8-1   Summary of FAO Directives (Cont)

!XL	Convert a longword to hexadecimal.	parameter.
!ZB	Convert an unsigned decimal byte.	
!ZW	Convert an unsigned decimal word.	
!ZL	Convert an unsigned decimal longword.	

Numeric Conversion (blank-filled)

!UB	Convert an unsigned decimal byte.	Value to be converted to ASCII representation.
!UW	Convert an unsigned decimal word.	
!UL	Convert an unsigned decimal longword.	
!SB	Convert a signed decimal byte.	For byte or word conversion, FAO uses only the low-order byte or word of the longword parameter.
!SW	Convert a signed decimal word.	
!SL	Convert a signed decimal longword.	

Output String Formatting

!/	Insert new line (CR/LF).	None
!_	Insert a tab.	
!^	Insert a form feed.	
!!	Insert an exclamation mark.	
!%S	Insert S if most recently converted numeric value is not 1.	
!%T	Insert the system time.	Address of a quadword time value to be converted to ASCII. If 0 is specified, the current system time is used.
!%D	Insert the system date and time.	
!n< !>	Define output field width of n characters. Format all data and directives within delimiters,<and>, left-justified, and blank-filled within the field.	None

Table 8-1   Summary of FAO Directives (Cont)

!n*c	Repeat the character c in the output string n times.	None

Parameter Interpretation

!-	Reuse the last parameter in the list.	None
!+	Skip the next parameter in the list.	

*If a variable repeat count and/or a variable output field length is specified with a directive, parameters indicating the count and/or length must precede other parameters required by the directive.

Example 8-25 shows how the FAO directives should be built into the control string and how the print macros should be coordinated with the $DS_ERRxxx_x and $DS_CVTREG_x macros.

8.7.2.1 $DS_PRINTB_x Print Basic Error Information – Use this macro to print basic error information, for example:

```
explanatory message
expected data
received data
XOR data
```

Typically, one $DS_PRINTB_x macro is used to print one line. Therefore, four $DS_PRINTB_x macros would be used to print the explanatory message, the expected data, the received data, and the XOR data.

The message will be inhibited if either the IE1 or the IE2 flag is set.

$DS_PRINTB_x format, [arg], [arg], ...

format = the address of the control string. The control string consists of the fixed text of the output string and the conversion directives.

arg   = directive parameters contained in longwords. A parameter may be a value that is to be converted, the address of the string that is to be inserted, a length, or an argument count, depending on the directive. For each directive in the control string there may be corresponding parameters. Up to 16 arguments may be

8-33

Return Status Codes: None.

NOTE
Refer to Chapter 9, Paragraph 9.6 of
this manual and the VAX/VMS System
Services Reference Manual for more FAO
information.

8.7.2.2 $DS_PRINTX_x Print Extended Error Information – This
macro should be used to print information that supplements the
basic error information, such as:

    failing addresses
    device register contents
    channel register contents
    additional explanations

The service will not print the message if either IE1, IE2, or IE3
is set.

$DS_PRINTX_x format, [arg], [arg], ...

format = the address of the control string. The control string
          consists of the fixed text of the output string and the
          conversion directives.

arg   = directive parameters contained in longwords. A parameter
          may be a value that is to be converted, the address of
          the string that is to be inserted, a length, or an
          argument count, depending on the directive. For each
          directive in the control string there may be
          corresponding parameters. Up to 16 arguments may be
          supplied.

Return Status Codes: None.

NOTE
Refer to Chapter 9, Paragraph 9.6 of
this manual and the VAX/VMS System
Services Reference Manual for more FAO
information.

8.7.2.3 $DS_PRINTF_x Print a Forced Message – Use this macro when the inhibit error control flags (IE1, IE2, IE3) must be overridden. Messages which should be forced are those which alert the operator to significant events or notify the operator that some action on his part is required. This macro is normally used in main-line code.

$DS_PRINTF_x format, [arg], [arg], ...

format = the address of the control string. The control string consists of the fixed text of the output string and the conversion directives.

arg = directive parameters contained in longwords. A parameter may be a value that is to be converted, the address of the string that is to be inserted, a length, or an argument count, depending on the directive. For each directive in the control string there may be corresponding parameters. Up to 16 arguments may be supplied.

Return Status Codes: None.

> **NOTE**
> Refer to Chapter 9, Paragraph 9.6 of this manual and the VAX/VMS System Services Reference Manual for more FAO information.

8.7.2.4 $DS_PRINTS_x Print Summary Report – The $DS_PRINTS_x macro should be used only in the summary routine to print a summary message. The message will not be printed if the IES control flag is set.

$DS_PRINTF_x format, [arg], [arg], ...

format = the address of the control string. The control string consists of the fixed text of the output string and the conversion directives.

arg = directive parameters contained in longwords. A parameter may be a value that is to be converted, the address of the string that is to be inserted, a length, or an argument count, depending on the directive. For each directive in the control string there may be corresponding parameters. Up to 16 arguments may be supplied.

Return Status Codes: None.

> **NOTE**
> Refer to Chapter 9, Paragraph 9.6 of this manual and the VAX/VMS System Services Reference Manual for more FAO information.

8.7.3 $DS_CVTREG_x Convert Register

This macro calls a supervisor routine that converts the contents of a register to a counted ASCII string of mnemonics for inclusion in an error message. For every bit set in the register, the corresponding mnemonic is included in the ASCII string. Several bits in a register may make up a function. In this case, the corresponding mnemonic should be followed by "=n^R" or "=n@" in the device registers mnemonics list, where:

n = the number of bit positions that make up the field.

R = the radix in which the value of the function field should be printed.

The $DS_CVTREG_x macro can be used in a print routine in conjunction with one of the PRINT macros. Since the routine will be called through one of the $DS_ERRxxx_x macros, all information necessary to the routine must be specified with arguments to that $DS_ERRxxx_x macro.

$DS_CVTREG_x msb, data, mneadr, strbuf, maxlen, [V1--6]

msb = the most significant bit position.

data = the address of the register contents in memory.

mneadr = the address of a counted ASCII string of bit mnemonics.

strbuf = the address of a buffer where the counted ASCII string is to be returned.

maxlen = the length of the buffer.

V1 -- V6 = pointers to counted lists pointing to the ASCII strings defining function values specified with a mnemonic in the form xxxxxx=n@.

Return Status Codes
$DS_NORMAL: Service successfully completed.

$DS_PROGERR: This status code is returned to indicate any of the following conditions:

a.   the number of arguments is greater than 11,

b.   the MSB is greater than 32,

c.   the mnemonic string is exhausted and an = sign has been encountered with nothing to the right of it,

d.   a negative digit was encountered in the mnemonic string,

e.   a bad character was encountered in the mnemonic string,

f.   some character other than a comma has been used to separate mnemonics in the string,

g.   the ASCII format overflowed,

h.   the caller's buffer overflowed.

Notice that when the code DS$_PROGERR is returned, 16(AP), the output buffer address, is cleared. The zero in 16(AP) indicates that there is no output from this routine.

NOTES

1.   **The first mnemonic is associated with the bit position MSB.**

2.   **R1 contains the full length of the string plus the count byte. This is true even if the buffer size is too short or the string is longer than 255 bytes.**

Example 8-25 shows a print subroutine containing the $DS_CVTREG_x macro. The related global data section and the calling test routine are shown as well.

Global Data and buffers

```
        .SBTTL PROGRAM GLOBAL DATA SECTION.
; ++
; Functional Description:
;
;       All dynamically modified data should be placed in this
;       section. This is the only section which will normally be
;       write enabled.
; --
        LOG_UNIT$L::    .LONG                       ; LUN
        DZ$CSR::        .ADDRESS 0                  ; CSR address
        DEV_REG::       .ADDRESS 0                  ; register pointer
        CSR$W::         .WORD                       ; CSR buffer
        XOR$W::         .WORD                       ; XOR of exp and rec
                                                    ; data
        HDW_PTBLE$A::   .LONG                       ; address of pointer
                                                    ; to P-table
        CHAR_BUF::      .BLKB 132                   ; 132 character I/O
                                                    ; buffer

        CSRREG$C::      .ASCIC /TRDY,TIE,SA,SAE,NUM,/-
                        /BIT10,BIT9,BIT8,RDONE,RIE,/-
                        /MSE,CLR,MAINT,NU,NU,NU/
```

```
Print Subroutine

     .SAVE
     .PSECT PRINT, LONG
EXP_FMT$C::  .ASCIC \!/EXPECTED:!XW(X)\    ; Expected data is
                                           ; converted to hex.


RCV_FMT$C::  .ASCIC \!/RECEIVED:!XW(X)\    ; Received data is
                                           ; converted to hex.


XOR_FMT$C::  .ASCIC \!/XOR:!_!XW(X);!AC!/\
                                           ; XOR data is
                                           ; converted to hex.
                                           ; The string of
                                           ; bit mnemonics is
                                           ; added.
FMTCR::      .ASCIC \!/\                    ; line feed

REG_ERROR$A::
     $DS_BGNMESSAGE          REGMASK = <R2, R3, R4, R5, R6>
     MOVL ERR$_P1(AP),R2                    ; name of register
     MOVL ERR$_P2(AP),R3                    ; address of register
     MOVZWL ERR$_P3(AP),R4                  ; expected data
     MOVZWL ERR$_P4(AP),R5                  ; received data
     $DS_PRINTB_S -
         FORMAT = (R2)                      ; print message
     XORL3 R4, R5, R2                       ; exclusive OR
     $DS_CVTREG_S -
         MSB = #15, -                       ; 16-bit register
         DATA = R2, -                       ; bits in error
         MNEADR = CSR_REG_$C                ; address of counted
                                            ; ASCII
                                            ; string
         STRBUF = CHAR_BUF, -               ; address of string
                                            ; buffer
         MAXLEN - #132, -                   ; buffer length
                                            ; (CVTREG)

     $DS_PRINTX_S -
         FORMAT = EXP_FMT$C, -              ; print message
         P0 = R4                            ; expected data

     $DS_PRINTx_S -                         ; print message
         FORMAT = RCV_FMT$C, -
         P0 = R5                            ; received data

     $DS_PRINTX_S -
         FORMAT = XOR_FMT$C, -              ; print message
         P0 = R2, -                         ; XOR data
         P1 = #CHAR_BUF                     ; bits in error
```

```
Calling Test Routine
```
```
        MOVW #^X5068, @DZ$CSR                 ; Write to CSR.
        MOVL #^X5068, CSR$W                   ; expected data
        MOVW @DZ$CSR, CSR$W                   ; read CSR
        XORW3 #^X5068, CSR$W, XOR$W           ; check
        BEQL 10$
        $DS_ERRHARD_S -                       ; error message
            UNIT = LOG_UNIT$L, -
            MSGADR = REGERR, -
            PRLINK = REG_ERROR$A, -           ; extended print
                                              ; routine
            P1 = #MSG_CSRERR$C, -             ; extended message
            P2 = REGISTER$A, -                ; address of CSRREG$C
            P3 = #^X5068, -                   ; expected data
            P4 = CSR$W                        ; received data

10$:                                          ; (continue test)
```

Example 8-25   Use of the $DS_CVTREG_x Macro

In this example, the PRLINK parameter in the $DS_ERRHARD_S macro call in the calling test routine specifies the name of the print subroutine. P1 through P4 specify the extended message, the address containing bit mnemonics for the CSR, and expected and received data.

The $DS_CVTREG_S macro in the print subroutine places the mnemonics corresponding to the failing bits in the buffer CHAR_BUF. The third $DS_PRINTX_S macro call following the $DS_CVTREG_S macro causes the failing bit mnemonics to be printed.

Example 8-26 shows the message produced by the code shown in Example 8-25.

```
********  PROGNAME -- V1.0   ********
PASS 1  TEST 1  SUBTEST 1  ERROR 3  10-APR-1979  15:37:50.47
HARD ERROR WHILE TESTING TTA:   DEVICE REGISTER ERROR

CSR IN ERROR
EXPECTED: 5068(X)
RECEIVED: 0000(X)
XOR:      5068(X);TIE,SAE,RIE,MSE,MAINT
```

Example 8-26   A Sample Error Message Printout

8.8   PROGRAM-OPERATOR DIALOGUE SERVICE MACROS
Some diagnostic programs require operator-supplied information for
the execution of certain tests. These tests should be placed in a
separate section named Manual Intervention. The following service
call macros are useful in organizing the dialogue and adapting the
program flow to the operator response:

        $DS_ASKxxx_x macros
        $DS_PARSE_x
        $DS_CLI.

You can use any of the five $DS_ASKxxx_x macros to prompt the
operator with a message. The operator's response is returned in a
specified buffer.

The $DS_PARSE_x macro can then be used to parse a command line
with a syntax tree. At each node of the tree (created by a
$DS_CLI macro) control passes to a CASE instruction. Control then
passes to the code that executes the operator's command. Figure
8-2 shows the flow of a typical operator dialogue portion of a
program.
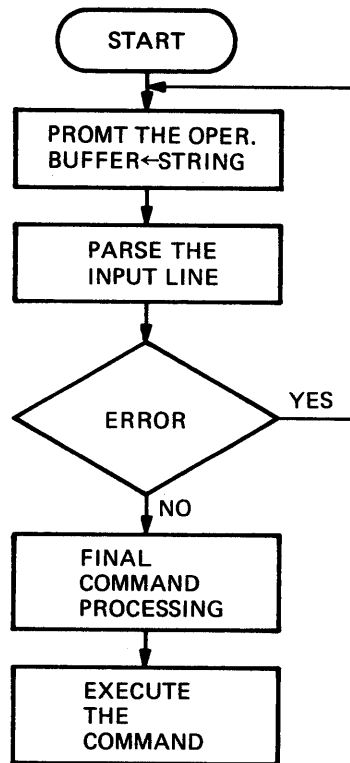
TK-1883

Figure 8-2   Operator Dialogue Flowchart

If the operator has typed in an improper command, the code should detect the error and prompt the operator for another command. If there is no error, the code should complete processing the command and then execute it.

8.8.1 $DS_ASKSTR_x Ask the Operator for a String
This macro calls a supervisor routine that prompts the operator for a string response. It accepts an ASCII string that it checks against a list of possible responses. If valid, the string is placed in a caller-specified buffer.

$DS_ASKSTR_x msgadr, bufadr, [maxlen], [valtab], [defadr]

msgadr = the address of a counted ASCII string used as a prompt.

bufadr = the address of a counted ASCII buffer.

maxlen = the maximum length of the response string (does not include count byte). Default = 72.

valtab = the address of a counted list of string pointers. Default = 0, meaning no validation table.

defadr = the address of a counted ASCII string to be used if the operator gives a null response. Default = 0, meaning that there is no default string.

Return Status Codes
DS$_NORMAL: Service successfully completed.

DS$_PROGERR: The number of arguments supplied is incorrect.

DS$_TRUNCATE: The string supplied by the operator has been truncated because it will not fit into the buffer supplied by the program.

NOTES
1.   Execution of this macro will cause a program abort if the Operator flag is clear.

2.   If the Prompt flag is set, ranges and default values will be printed with the prompt message.

3.   If VALTAB = 0, any string will be accepted without qualification.

4.   If VALTAB ≠ 0, R1 will return with an index value into the validation table. If DEFADR is specified and

the operator selects a carriage
return only, R1 will be returned
containing 0.

5.   Do not use FAO directives in the
prompt string. If you want to
ensure that the prompt message
always starts on a new line, place
CR and LF before the first message
delimiter. For example:
<13><10>\PLEASE TYPE A COMMAND\.

8.8.2 $DS_ASKDATA_x Ask the Operator for a Numeric Value

This macro calls a supervisor service routine that prompts the
operator for a numeric value and ensures that it is within an
acceptable range. The service converts the ASCII numeric string to
binary data, truncates it if necessary, and stores it in a
caller-specified field (mask).

$DS_ASKDATA_x msgadr, datadr, [radix], [mask], [lolim], [hilim],
[defalt], [unused], [exword]

msgadr =   the address of a counted ASCII string used as a prompt.

datadr =   the address of a longword that will receive the
response.

radix =   PAR$_BIN, PAR$_OCT, PAR$_HEX, or the default radix
PAR$_DEC.

mask   =   a bit mask indicating the field position and size.

lolim =   the minimum acceptable numeric response. Default value =
- maximum, $-(2^{31})$.

hilim =   the maximum acceptable numeric response. Default value =
+ maximum, $(2^{31}-1)$.

defalt =   the value to use if the operator gives a null response.

unused =   reserved for expansion.

exword =   the exception mask. PAR$_NODEF means that there is no
default. PAR$_ATDEF, PAR$_ATLO, and PAR$_ATHI cause the
parameters DEFALT, LOLIM, and HILIM to be interpreted as
containing the addresses where the corresponding values
may be found, instead of containing their literal
values.

Return Status Codes

DS$_NORMAL: Service successfully completed.

DS$_PROGERR: The number of arguments supplied is incorrect.

DS$_TRUNCATE: The value supplied by the operator is too large to fit in the specified buffer.

<div align="center">NOTES</div>

1. Execution of this macro will cause a program abort if the Operator flag is clear.

2. If the Prompt flag is set, ranges and default values will be printed with the prompt message.

3. Truncation of left-most bits occurs if a response is larger than the mask.

4. The radix and exception mask arguments are defined by $DS_PARDEF.

5. R1 contains the converted binary value, not truncated, on return.

6. Do not use FAO directives in the prompt string.

8.8.3 $DS_ASKVLD_x Ask the Operator for a Numeric Value

Like the $DS_ASKDATA macro, the $DS_ASKVLD macro calls a supervisor routine that prompts the operator for a numeric value. It accepts an ASCII numeric string as input, converts the string to binary format, and truncates the string if necessary. The supervisor routine then checks to determine whether the value is within an acceptable range and stores the value in a caller-specified variable field. This field is specified by position and size, rather then with a mask.

$DS_ASKVLD_x msgadr, datadr, [radix], [pos], [size], [lolim], [hilim], [defalt], [unused], [exword]

msgadr = the address of a counted ASCII string used as a prompt.

datadr = the address of a longword that will receive the response.

radix = PAR$_BIN, PAR$_OCT, PAR$_HEX, or the default radix PAR$_DEC.

pos   = the right-most bit of the field. Range = 0 through 31. Default = 0.

size   = the number of bits in the field.

lolim   =   the minimum acceptable numeric response. Default value =
            - maximum, $-(2^{31})$.

hilim   =   the maximum acceptable numeric response. Default value =
            + maximum, $(2^{31}-1)$.

defalt =   the value to use if the operator gives a null response.

unused =   reserved for expansion.

exword =   the exception mask. PAR$_NODEF means that there is no
            default. PAR$_ATDEF, PAR$_ATLO, and PAR$_ATHI cause the
            parameters DEFALT, LOLIM, and HILIM to be interpreted as
            containing the addresses where the corresponding values
            may be found, instead of containing their literal
            values.

Return Status Codes
DS$_NORMAL: Service successfully completed.

DS$_PROGERR: The number of arguments supplied is incorrect.

DS$_TRUNCATE: The value supplied by the operator is too large to
fit in the specified buffer.

NOTES
1. Execution of this macro will cause
    a program abort if the Operator
    flag is clear.

2. If the Prompt flag is set, ranges
    and default values will be printed
    with the prompt message.

3. The radix and exception mask
    arguments are defined by
    $DS_PARDEF.

4. Do not use FAO directives in the
    prompt string.

8.8.4 $DS_ASKLGCL_x Ask the Operator for a Logical Response
This macro calls a supervisor service routine that prompts the
operator for a logical response to a specified question. It
accepts an ASCII "yes" or "no" string and converts this to a
single bit (flag). The supervisor routine then stores the bit in a
caller-specified bit position within a caller-specified byte.

$DS_ASKLGCL_x msgadr, datadr, [pos], [yexfer], [noxfer], [defalt]

msgadr = the address of a counted ASCII string used as a prompt.

datadr = the address of a byte that will receive the response.

pos   = a bit position within DATADR. Range = 0 through 7.
           Default = 0.

yexfer = the branch address for a positive response. Default = 0,
           meaning no branch.

noxfer = branch address for negative response. Default = 0,
           meaning no branch.

defalt = PAR$_YES or PAR$_NO.

Return Status Codes
DS$_NORMAL: Service successfully completed.

DS$_PROGERR: The bit position argument (POS) specified by the
caller is too large a number or the number of arguments supplied
is incorrect.

> **NOTES**
> 1. Execution of this macro will cause
>   a program abort if the Operator
>   flag is clear.
>
> 2. If the Prompt flag is set, ranges
>   and default values will be printed
>   with the prompt message.
>
> 3. Do not use FAO directives in the
>   prompt string.

8.8.5 $DS_ASKADR_x Ask Operator for an Address
This macro calls a supervisor service routine that displays a
prompt message on the operator's terminal, asking the operator for
an address. The service accepts an ASCII numeric address string
and checks whether or not it is within an acceptable range. The
service then stores the address in a user-specified longword.

$DS_ASKADR x msgadr, datadr, [radix], [lolim], [hilim], [defalt], [unused], [exword]

msgadr = the address of a counted ASCII string used as a prompt.

datadr = the address of a longword that will receive the response.

radix = PAR$_BIN, PAR$_OCT, PAR$_DEC, or the default radix PAR$_HEX.

lolim = the minimum acceptable numeric response. Default value = 0.

hilim = the maximum acceptable numeric response. Default value = $(2^{32}-1)$.

defalt = the value to use if the operator gives a null response. Default value = 0.

unused = reserved for expansion.

exword = the exception mask. PAR$_NODEF means there is no default. PAR$_ATDEF, PAR$_ATLO, and PAR$_ATHI cause the parameters DEFALT, LOLIM, and HILIM to be interpreted as containing the addresses where the corresponding values may be found, instead of containing their literal values.

Return Status Codes
DS$_NORMAL: Service successfully completed.

DS$_PROGERR: The number of arguments supplied is incorrect.

NOTES
1. Execution of this macro will cause a program abort if the Operator flag is clear.

2. If the Prompt flag is set, ranges and default values will be printed with the prompt message.

3. The radix and exception mask arguments are defined by $DS_PARDEF.

4. Do not use FAO directives in the prompt string.

8.8.6 $DS_PARSE_x Parse

This macro calls a supervisor routine (DSX$PARSE) to parse an ASCII string in the buffer described, using a caller-supplied parse tree. When a match in the tree is found, the routine calls a caller-supplied action routine. Upon a mismatch, the supervisor routine branches to another point in the parse tree.

$DS_PARSE_x bufadr, tree, action

bufadr = the address of a quadword descriptor for the buffer.

tree   = the address of the tree structure to be used in parsing.

action = the address of the action routine.

Return Status Codes

DS$_NORMAL: Service successfully completed.

DS$_ERROR: Error match code encountered in the parse tree.

DFS$_OVERFLOW: Numeric input overflow in the quadword descriptor, BUFADR.

NOTES

1.   The tree structure is defined by repeated use of the macro $DS_CLI_x.

2.   The ASCII string that is parsed may be generated in a number of ways. The simplest way to generate the string is to use one of the $DS_ASKxxx_x macros.

Example 8-27 shows how the $DS_ASKSTR_x, $DS_PARSE_x, and $DS_CLI macros can be coordinated to solicit and act on information from the operator. The code in this example corresponds roughly to the flowchart shown in Figure 8-2.

```
; Header module

              NULL=0
              SUB1=1
              SUB2=2
              STOP=3
              INIT=4


CHAR_BUF::    .BLKB  132        ; 132 character I/O buffer
CMD_BUF::     .QUAD             ; quadword descriptor
TOKEN_BLOCK:: .ADDRESS 0        ; buffer for action routine
```

```
.PAGE
;++
; Command Interpreter Tree
;       This tree provides nodes for the interpretation of three
;       commands:
;               SUB1
;               SUB2
;               STOP
;       An ambiguous or invalid command will cause the operator to
;       be notified to that effect.
;
;
;--
FIRTREE::
10$:    $DS_CLI CLI$K_BR, INIT 20$ ; Clear token block.
20$:    $DS_CLI-                    ; Is the first character an S?
            CHAR=<^A"S">, -
            ACTION=NULL, -          ; not enough information
            MISS=110$
30$:    $DS_CLI-                    ; Are the next 3 characters TOP?
            CHAR=CLI$K_STRING, -
            ACTION=STOP, -
            MISS=50$, -
            ASCII=<\TOP\>
40$:    $DS_CLI-                    ; Operator has typed STOP.
            CHAR=CLI$K_EXIT
50$:    $DS_CLI-                    ; Is the next character U?
            CHAR=<^A"U">, -
            ACTION=NULL, -          ; still not enough information
            MISS=110$
60$:    $DS_CLI-                    ; Is the next character B?
            CHAR=<^A"B">, -
            ACTION=NULL, -          ; still not enough information
            MISS=110$
70$:    $DS_CLI-                    ; Is the next character 1?
            CHAR=<^A"1">, -
            ACTION=SUB1, -
            MISS=90$
80$:    $DS_CLI-                    ; Operator has typed SUB1.
            CHAR=CLI$K_EXIT
90$:    $DS_CLI-                    ; Is the next character 2?
            CHAR=<^A"2">, -
            ACTION=SUB2, -
            MISS=110$
100$:   $DS_CLI-                    ; Operator has typed SUB2.
            CHAR=CLI$K_EXIT
110$:   $DS_CLI-                    ; Operator has typed an
            CHAR=CLI$K_ERROR        ; improper command.
```

```
          .SBTTL PROGRAM SUBROUTINES
  •
  •
  •
ACT_ENTRY::
     CASEL   R0,#0,#4
10$:
     .WORD   ACT_NULL      -10$              ; If null
     .WORD   ACT_SUB1      -10$              ; If SUB1
     .WORD   ACT_SUB2      -10$              ; If SUB2
     .WORD   ACT_STOP      -10$              ; If STOP
     .WORD   ACT_INIT      -10$
20$: $DS_ERRSYS_S                            ; fatal error
     .PAGE
     $DS_ABORT       PROGRAM                 ; Abort program.
ACT_INIT:   CLRL    TOKEN_BLOCK
            RSB
ACT_SUB1:   MOVAL SUB1_ADR,TOKEN_BLOCK   ; Load command routine
                                         ; ADR.
            RSB
ACT_SUB2:   MOVAL SUB2_ADR,TOKEN_BLOCK   ; Load command routine
                                         ; ADR.
            RSB
ACT_STOP:   MOVAL STOP_ADR,TOKEN_BLOCK   ; Load command routine
                                         ; ADR.
ACT_NULL:   RSB
               •
               •
               •

GETLINE::
     .ASCIC <13><10>\PLEASE TYPE A COMMAND\
                                          ; FAO directives not
                                          ; permitted here.

AMBICMD::
     .ASCIC \!/AMBIGUOUS COMMAND\

INVCMD::
     .ASCIC \!/INVALID COMMAND\
               •
               •
               •
```

```
; Test Module
```

```
     $DS_BGNTEST      ALIGN=LONG
WHATNEXT::
10$: $DS_ASKSTR_S -                       ; Ask operator for a
                                          ; command.
          MSGADR=GETLINE, -               ; Prompt line.
          BUFADR=CHAR_BUF, -              ; command storage
          DEFADR=0                        ; There is no default.
```

```
        $DS_BNCOMPLETE 1Ø$              ; if at first you don't
                                        ; succeed...
        MOVZBL CHAR_BUF, CMDBUF         ; command string length
        MOVAL  CHAR_BUF+1, CMDBUF+4     ; Build quadword
                                        ; descriptor.
        $DS_PARSE_S -                   ; Parse the command.
             BUFADR=CMDBUF, -
             TREE=FIRTREE, -
             ACTION=ACT_ENTRY
        $DS_BCOMPLETE 2Ø$               ; The command has been
                                        ; parsed.
        $DS_PRINTF_S -
             FORMAT=INVCMD              ; "INVALID COMMAND"
        BRB 1Ø$                         ; Get another command.
2Ø$:    TSTL TOKEN_BLOCK                ; Is TOKEN_BLOCK empty?
        BNEQ 3Ø$                        ; No, a valid command was
                                        ; typed.
        $DS_PRINTF_S -
             FORMAT=AMBICMD             ; "AMBIGUOUS COMMAND"
        BRW 1Ø$                         ; Get another command.
3Ø$ JSB     @TOKEN_BLOCK               ; Execute command.
        BRW 1Ø$                         ; Last command executed,
SUB1_ADR::                              ; get another.
        .
        .
        .
        RSB
SUB2_ADR::
        .
        .
        .
        RSB
STOP_ADR::
        $DS_ENDTEST                     ; exit
```

Example 8-27 Prompting and Parsing a Command

The service called by $DS_ASKSTR_S prompts the operator with a message at GETLINE. When the operator responds, the service places the response string in the response buffer, CHAR_BUF. If the service does not complete these tasks successfully, the macro is called again.

On successful return from the ASKSTR service, control passes to the $DS_PARSE_S macro. This macro calls a service that parses the string in CHAR_BUF according to the tree created by the repetition of the $DS_CLI macro.

Figure 8-3 shows the configuration of the tree. The parse macro checks one letter at a time against the tree. At 2Ø$ the first character is checked. If the character is not an S, control passes to the label specified by the MISS argument, 11Ø$. A miss at this point shows that the operator has typed an improper command. If the first character is an S, control passes to the ACT_NULL label.
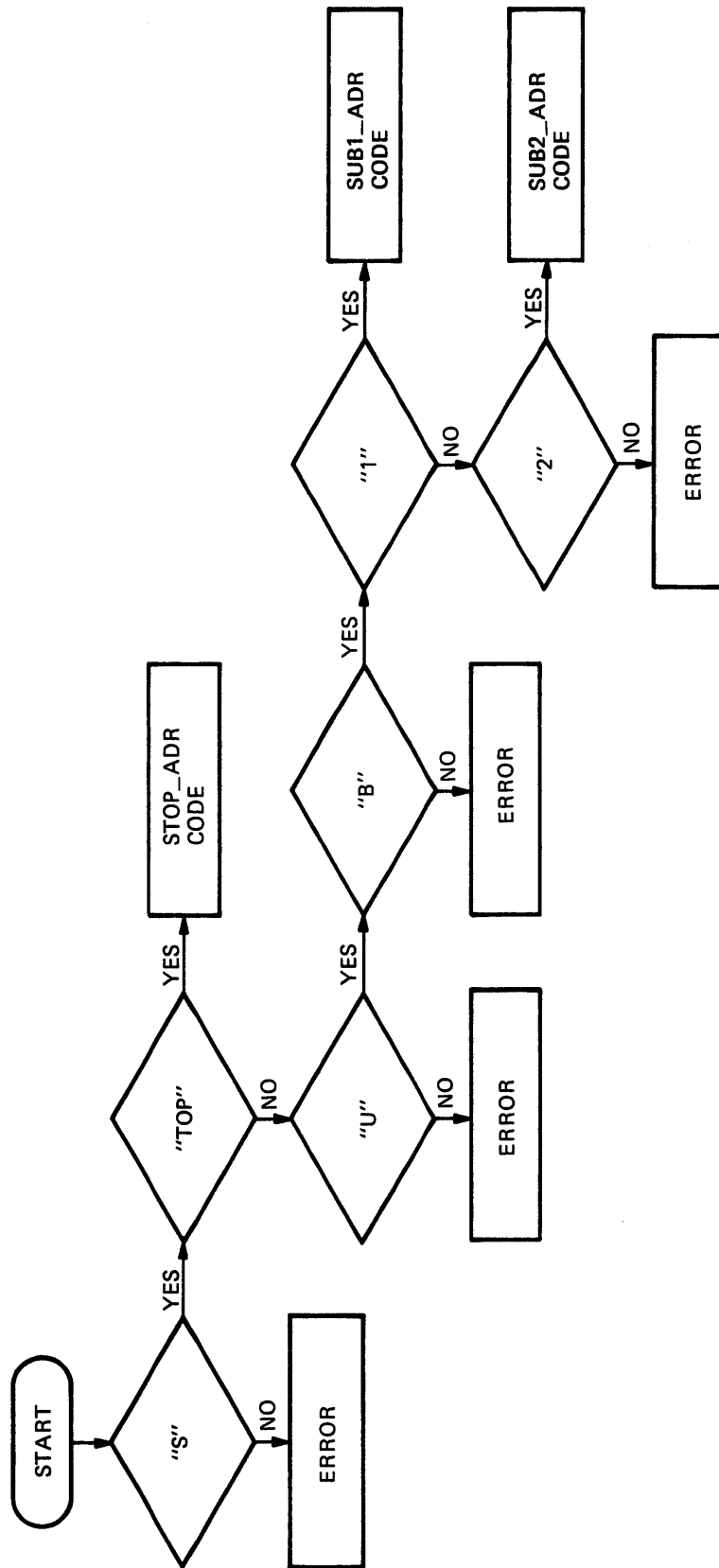
TK-1882

Figure 8-3   Command Interpreter Tree Structure

Control then returns to 30$, where the next three characters are
checked against TOP. T or TO or TOP will match. Any other
character combination is a miss. If there is a match, control
passes to 40$, an exit from the tree back to the parse service. If
there is a miss, the character checked may or may not be an error.
Control passes to 50$ which checks for a U, and so on.

The tree provides seven exits:

```
40$ for STOP
80$ for SUB1
100$ for SUB2
110$ for an improper command
20$, 50$, and 60$, for ambiguous commands
```

If the operator has typed STOP, SUB1, or SUB2, control passes to
the ACT_ENTRY label. The CASEL instruction selects ACT_STOP for
STOP, and so on. Each of these brief action routines places a
different address in TOKEN_BLOCK, after ACT_INIT has cleared it.
The JSB @TOKEN_BLOCK following the $DS_PARSE_S macro then passes
control to the appropriate test code:

```
SUB1_ADR
SUB2_ADR
or
STOP_ADR
```

If the operator has typed an ambiguous command, such as S, SU, or
SUB, the token block is left empty. The instruction at 20$ in the
test code detects this condition. The code then notifies the
operator of an ambiguous command and returns control to 10$ in
order to get another command.

8.9   SYSTEM CONTROL SERVICE MACROS
The programmer should not try to modify the system control block
(SCB) directly, since doing so would probably cause supervisor
errors. Therefore, four supervisor service macros are provided to
enable indirect control of the system control block by the
diagnostic program. They should be used in level 3 diagnostic
programs only.

8.9.1 $DS_SETIPL_x Set Interrupt Priority Level
This macro should be used to raise the IPL to block interrupts
from the device under test.

$DS_SETIPL_x level

level   =   interrupt priority level.

Return Status Codes
DS$_NORMAL: Service successfully completed.

8.9.2 $DS_SETVEC_x Set System Control Block Vector
This macro calls a supervisor routine (DSX$SETVEC) to load the
address of an exception or interrupt routine into the system
control block, setting a system control block vector for program
control.

$DS_SETVEC_x vector, isradr, [code]

vector = the absolute vector address.

isradr = the virtual address of a service routine.

code   = 0 or 1; 0 = kernel stack; 1 = interrupt stack.

Return Status Codes
DS$_NORMAL: Vector modified.
DS$_IVADDR: Service address bits <01:00> are not zero.
DS$_IVVECT: Invalid vector.
DS$_ICBUSY: Interval clock busy.

Use the $DS_SETVEC_x macro when you want your program to field
interrupts or exceptions in the most direct possible manner. When
set, the vector points directly to a service routine specified by
the programmer.

8.9.3 $DS_CLRVEC_x Clear a System Control Block Vector
This macro calls a supervisor routine that sets the system control
block vector for supervisor handling. The routine loads the vector
in the system control block (SCB) with the contents of the
corresponding vector in the SCB_IMAGE (a page that holds the
initial contents of each vector).

$DS_CLRVEC_x vector

vector = absolute vector address.

Return Status Codes
DS$_NORMAL: Service completed successfully.

The $DS_CLRVEC_x macro may be paired with the $DS_SETVEC macro to
keep the system control block in good order.

8.9.4 $DS_INITSCB_x Initialize System Control Block
This macro calls a supervisor routine (DSX$INITSCB) that sets the
vectors in the system control block for supervisor handling. The
routine copies a 512 byte image (SCB_IMAGE) into the system
control block.

$DS_INITSCB_x (no arguments)

Return Status Codes
DS$_NORMAL: Service successfully completed.

You should use the $DS_INITSCB_x macro to clear up the system control block if several elements of the system control block have been altered.

8.10   HARDWARE P-TABLE ACCESS

When the supervisor is running and a diagnostic program is started, the supervisor builds a hardware P-table for each device selected by the operator for testing. The format for the hardware P-table is shown in Example 6-5. The programmer can gain access to the hardware P-table for the device under test with the $DS_GPHARD_x macro. In this way, he can determine the base address of the device and calculate offsets to all of the device registers.

8.10.1 $DS_GPHARD_x Get Hardware P-Table Base Address

This macro calls a supervisor routine (RGPHARD) to obtain the address of the entry in the P-table associated with the given logical unit number.

$DS_GPHARD_x unit, retadr

unit   =   the logical unit number.

retadr =   the longword to receive the base address of the P-table entry.

Return Status Codes
DS$_NORMAL: Service successfully completed.

DS$_ERROR: The argument list does not contain exactly two elements, or the logical unit number specified is too large.

Example 8-28 shows a typical use of the $DS_GPHARD_x macro in the initialization code.

```
     $DS_BGNINIT

INITIALIZE:
           .WORD      ^M<>                    ; entry mask
     $DS_BPASS0        10$
     $DS_ENDPASS_G
10$: CLRL   LOG_UNIT$L                        ; logical unit 0
     CLRW   CSR$W                             ; Clear CSR Buffer
     $DS_GPHARD_S -                           ; Get hardware P-table
                                              ; address.
           DEVNUM = LOG_UNIT$L,-              ; logical unit

           ADRLOC = HDW_PTPBLE$A
     $DS_BERROR
     MOVL   HDW_PTBLE$A, R2                   ; P-table offset to R2
     MOVL   HP$A_DEVICE(R2), DZ$CSR           ; CSR address to DZ$CSR
```

Example 8-28   Use of the $DS_GPHARD_x Macro

Notice that the P-table base address is returned in HDW_PTBLE$A.
This address is then moved to R2. After this, the contents of the
P-table are easily accessible.

A small subset of VMS services are available to level 2 and 2R diagnostic programs. Some of these services are also available to level 3 diagnostic programs. These services always return status codes. Furthermore, they return control to the location following the instruction that called the VMS service. They do not generally change the flow of the program. Seven types of VMS service macros are available.

    I/O service macros
    Event flag service macros
    Timer service macros
    Formatted ASCII output service macros
    Memory management service macros
    Hibernate and Wake service macros
    Unwind service macro

9.1   Coding VMS Service Macros

The VMS system service macros take the form $xxxx_x. The prefix, $, signifies a VMS system service. The _x suffix shows that the macro may end in any of four ways:

DEF   - generate symbols and offsets

blank - generate an argument list (this corresponds to _L in the supervisor service macros)

_S   -. call the service with CALLS

_G   - call the service with CALLG

9

Consider the ASSIGN (Assign Channel) service. The four following macros are related to this service:

    $ASSIGNDEF
    $ASSIGN
    $ASSIGN_G
    $ASSIGN_S

This chapter and the following chapter provide explanations of the available VMS services.   Paragraph 9.3.1 gives the following format for the ASSIGN service.

    $ASSIGN_x devnam,chan,[acmode],[mbxnam]

The suffix _x following $ASSIGN shows that the ASSIGN service may be called with _G or _S suffixes. The arguments that follow it show the positional dependence and keyword names of each argument.

The arguments enclosed in square brackets are optional and may be omitted. When you do not specify an optional argument, the macro supplies a default value. The argument list supplied to any VMS service must have a format in memory like that shown in Figure 9-1.

```
┌───────────────────────┐
│                  │ 4   │
│                  │     │
├───────────────────────┤
│        DEVNAM         │
├───────────────────────┤
│         CHAN          │
├───────────────────────┤
│        ACMODE         │
├───────────────────────┤
│        MBXNAM         │
└───────────────────────┘
```
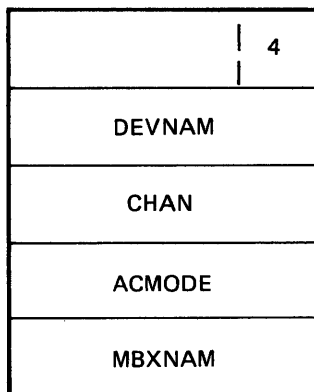
TK-1897

Figure 9-1   Memory Format for the ASSIGN VMS
Service Macro Arguments

All arguments are longwords. However, you should code them as symbols. The symbols must specify addresses or data. When the macro is assembled, the first longword will always contain, in its low-order byte, the number of arguments in the remainder of the list.

9.1.1 $name_G Macro Call Format

Use the $name_G macro call format when you wish to call a VMS service repeatedly with the same or nearly the same argument list. This macro generates a CALLG instruction. It requires the programmer to construct an argument list elsewhere in the program. You should specify the address of the list as an argument to the $name_G macro call, as follows:

        $name_G label

Use the $name macro format to generate the list of arguments referenced by the label. Again, the $name macro format for VMS services corresponds to the $DS_name_L macro format for supervisor services (refer to Chapter 8, Paragraph 8.1.1). In general, you should use the $name macro format to define argument lists in the data section of the program in the first module, as shown in Example 9-1.

```
LIST::
     $ASSIGN -                              ; arglist for assign
        DEVNAM=DISK, -                      ; device name
                                            ; string descriptor
        CHAN=CHANNEL_NO                     ; address for
                                            ; channel number
```

Example 9-1   Use of the $name Macro Format
to Build an Argument List

You can then call the VMS service associated with that argument
list with a $name_G macro format, as shown in Example 9-2.

```
     $ASSIGN_G        LIST                  ; Call ASSIGN VMS
                                            ; service with the
                                            ; arguments specified in
                                            ; LIST.
```

Example 9-2   Calling a VMS Service with the $name_G Macro Format

The argument, LIST, enables the macro to pass the items following
the LIST label in the data section of the program to the VMS
service being called.

Sometimes you may want to alter one or more of the arguments in
the list to be passed, leaving the rest unchanged. For example,
you may need to use the $ASSIGN_G macro again with a DEVNAM
argument of TAPE. Several steps are required. First, use the
$nameDEF macro format in the data section of the test module that
contains the VMS service call. This macro defines a set of
symbols that describe offsets from the base address of the
argument list. The $ASSIGNDEF macro creates the following symbols
and offsets.

```
          ASSIGN$_NARGS   =  4
          ASSIGN$_DEVNAM  =  4
          ASSIGN$_CHAN    =  8
          ASSIGN$_ACMODE  = 12
          ASSIGN$_MBXNAM  = 16
```

Second, you must replace the old value in the argument list with a
new value, as shown in Example 9-3.

```
     MOVAL       LIST,R2                    ; base address of
                                            ; arglist
     MOVL        TAPE,ASSIGN$_DEVNAM(R2)    ; Replace the device
                                            ; name in list
                                            ; with tape.
```

Example 9-3   Modification of an Argument List

Third, you can then call the VMS service with the $name_G macro
format. The macro will pass the argument list. Example 9-4 shows
the whole process.

9-3

```
; Header Module
```

```
LIST::
      $ASSIGN -                      ; arglist for ASSIGN
                DEVNAM=DISK, -       ; device name string descriptor
                CHAN=CHANNEL_NO      ; address for channel number
```

```
; Test Module
```

```
      $ASSIGNDEF                     ; Create offset symbols
      .                              ; for ASSIGN argument list.
      .
      $ASSIGN_G LIST                 ; CALL ASSIGN VMS service with
                                     ; unmodified argument
                                     ; list.
      MOVAL LIST,R2                  ; base address of argument list
      .                              ; for ASSIGN
      .
      .

      MOVL TAPE, ASSIGN$_DEVNAM(R2)  ; Replace the device name
      .                              ; in list with tape.
      .
      .

      $ASSIGN_G LIST                 ; Call ASSIGN VMS service
                                     ; with modified argument
                                     ; list.
```

Example 9-4   Uses of $name_G, $name, and $nameDEF Macro Formats

9.1.2 $name_S Macro Call Format

The $name_S macro call format is useful when you wish to call a
VMS service infrequently. It is also useful when you wish to call
a VMS service repeatedly, but with a different argument list each
time. This format generates a CALLS instruction. It requires the
programmer to specify the argument list as a part of the macro
call. The assembler (at assembly time) expands the macro to
create code that pushes the argument list on the stack during
program execution.

You can specify arguments for the $name_S macro format in either
of two ways:

   1.   Keywords
   2.   Position.

The $name_S macro call format functions just like the $DS_name_S
macro call format.   Refer to Chapter 8, Paragraph 8.1.2 for
further details. Be sure to omit the $DS_ prefix when coding VMS
service macros.

9.2   RETURN STATUS CODES

Like the return status codes for the supervisor services, the VMS service return status codes provide useful information. For example, the CLREF (Clear Event flag) service may return either of two codes after successful completion:

> SS$_WASCLR = the event flag was previously 0,
> SS$_WASSET = the event flag was previously 1.

Warning returns, and some error returns, indicate that the VMS service may have performed some, but not all, of the requested functions.

In general, the programmer should check the low-order bit of R0 following the completion of a VMS service call. If bit 0 is set, it indicates success. The programmer can cause a branch to an error checking routine if bit 0 is clear as shown in Example 8-5.

```
BLBC R0, LABEL              ; error if low bit clear
```

Example 9-5   Checking the Return Status Code
for an Error Condition

The error checking routine may check for specific codes or values. For example, the instruction in Example 9-6 checks for an illegal event flag number error condition following a call to the CLREF service.

```
CMPW SS$_ILLEFC, R0       ; Is the event
                          ; flag number
                          ; illegal?
```

Example 9-6   Checking the Return Status Code to Determine the
Nature of an Error

Notice that the instruction checks the low-order word in R0. This is possible because the high-order word of the longword returned in R0 is the same for all status codes.

9.3   I/O SERVICE MACROS

Direct access to peripheral device registers is unavailable to level 2 and 2R diagnostic programs. Furthermore, some of the supervisor services, including the channel services, cannot be called from these programs. The VMS I/O services that are available to the diagnostic system handle all peripheral device access and all channel control for level 2 and 2R diagnostic programs.

9.3.1 $ASSIGN_x Assign I/O Channel

A level 2 or 2R diagnostic program must assign a channel to a peripheral device before the program can perform any input or output operation on the device. The $ASSIGN_x macro calls a VMS service that assigns a channel and a channel number to a device.

This channel provides a path between the program and the device. In addition, you can use the $ASSIGN_x macro to establish a logical link with a remote node in a network in level 2R programs.

$ASSIGN_ devnam, chan, [acmode], [mbxnam]

devnam = the address of a character string descriptor that points to the device name string. The string may be either a physical device name or a logical name. If the first character in the string is an underscore (_), the service treats the name as a physical device name. Otherwise, the service performs a single level of logical name translation and uses the equivalence name, if there is any.

chan   = the address of a word to receive the channel number that the service assigns to the device.

acmode = the access mode to be associated with the channel. The service maximizes the specified access mode with the access mode of the caller. I/O operations on the channel can be performed only from equal and more privileged access modes.

          Kernel mode = 0
          Executive mode = 1
          Supervisor = 2
          User mode = 3

mbxnam = the address of the character string descriptor pointing to the logical name string for the mailbox to be associated with the device, if there is a mailbox. The mailbox receives status information from the device driver.

Return Status Codes
SS$_NORMAL: Service successfully completed.

SS$_REMOTE: Service successfully completed. A logical link was established with the target on a remote mode.

SS$_ACCVIO: The device or mailbox name string or string descriptor cannot be read, or the channel number cannot be written, by the caller (access violation).

SS$_DEVALLOC: Warning. The device is allocated to another process.

SS$_DEVNOTMBX: A mailbox name has been specified for a device that is not a mailbox.

SS$_EXQUOTA: The target of the assignment is on a remote node and the process has insufficient buffer quota to allocate a network control block.

SS$_INSFMEM: The target of the assignment is on a remote node and there is insufficient system dynamic memory to complete the request.

SS$_IVDEVNAM: No device name was specified, or the device or mailbox name string contains invalid characters. Or, the Network Connect Block has an invalid format.

SS$_IVLOGNAM: The device or mailbox name string has a length of 0 or has more than 63 characters.

SS$_NOIOCHAN: No I/O channel is available for assignment.

SS$_NOPRIV: The process does not have the privilege to perform network operations.

SS$_NOSUCHDEV: The specified device or mailbox does not exist.

Additional return status codes for network operations:

SS$_NOLINKS: No logical network links are available.

SS$_NOSUCHNODE: The specified network node is nonexistent or unavailable.

SS$_REJECT: The network connect was rejected by NSP (network services protocol) or the partner on the remote node; or the target image exited before the connect confirm could be issued.

<center>NOTES</center>

1. Channels can be assigned to devices on remote systems. For details on how to use Assign in conjunction with network operations, see the VAX-11 DECnet User's Guide.

2. Only the owner of a device can associate a mailbox with the device (the owner is the process that has allocated the device, either implicitly or explicitly). Then the device driver can send messages containing status information to the mailbox, as in the following cases:

   ● If the device is a terminal, a message may indicate dialup, hangup, or the reception of unsolicited input.

- If the target is on a network, the message may indicate the network connect or initiate. Or it may indicate whether the line is down.

- If the device is a line printer, the message may indicate that the printer is off-line.

   For details on the message format and the information returned, see the **VAX/VMS I/O User's Guide**.

3. Channels remain assigned until they are explicitly deassigned with the Deassign I/O Channel (DASSGN) system service, or until the image that assigned the channel exits.

4. The ASSIGN service establishes a path to a device, but does not check whether the caller can actually perform I/O operations to the device. Privilege and protection restrictions may be applied to the device drivers. For details on how the system controls access to devices, see the **VAX/VMS I/O User's Guide**.

The Assign Channel system service is basic to communication between a program and a device, because the QIO system service will function only on an assigned channel. The access mode of the process calling the QIO system service must be equal to or more privileged than the access mode from which the original channel assignment was made.

Example 9-7 shows the assignment of an I/O channel to the device TTA2. The Assign Channel service returns the channel number in the word at TTCHAN.

```
TTNAME:


      .ASCID                              ; terminal name string
                                          ; descriptor
TTCHAN:
      .BLKW   1                           ; terminal channel
      .                                   ; number
      .
      .
$ASSIGN_S-                                ; Assign channel.

           DEVNAM=TTNAME,-                ; terminal name
                                          ; string desciptor
           CHAN=TTCHAN                    ; terminal channel
                                          ; number
```

Example 9-7   Use of the $ASSIGN_x Macro

9.3.2 $DASSGN_x Deassign I/O Channel

This macro calls a VMS service that releases an I/O channel acquired for I/O operations with the Assign Channel system service.

$DASSGN_x chan

chan = the number of the I/O channel to be deassigned.

Return Status Codes

SS$_NORMAL: Service successfully completed.

SS$_IVCHAN: An invalid channel number was specified; that is, a channel number of 0 or a number larger than the number of channels available.

SS$_NOPRIV: The specified channel is not assigned, or was assigned from a more privileged access mode.

Privilege Restrictions

An I/O channel can be deassigned only from an access mode equal to or more privileged than the access mode from which the original channel assignment was made.

NOTES

1. When a channel is deassigned, any outstanding I/O requests on the channel are canceled. If a file has been opened on the specified channel, the file is closed.

2. If a mailbox was associated with the device when the channel was assigned, and there are not more channels assigned to the mailbox, the linkage to the mailbox is cleared.

3. If the I/O channel was assigned for a network operation, the network link is disconnected. For more information on channel assignment and desassignment for network operations, see the VAX-11 DECnet User's Guide.

4. If the specified channel is the last channel assigned to a device that has been marked for dismounting, the device is dismounted.

5. I/O channels are automatically deassigned at image exit.

You should use the Deassign Channel service in a diagnostic program when the program no longer needs access to the device. The service is normally used in the cleanup code and the initialization code, as shown in Example 9-8.

```
CLEANUP::
    •
    •
    •
        $DASSGN_S                ; Release TTCHAN.

        CHAN=TTCHAN
```

Example 9-8   Use of the $DASSGN_x Macro

9.3.3 $QIO_x Queue I/O Request

This macro calls a VMS service that begins an input or output operation. The service queues a request to a channel associated with a specific device. Control returns immediately to the calling program. The program can synchronize I/O completion in any of three ways.

1. Specify the address of an AST routine that is to execute when the I/O is completed.

2. Wait for a specified event flag to be set.

3. Poll the specified I/O status block for a completion status.

The service clears the event flag and the I/O status block, if they are specified, before it queues the I/O request.

$QIO_x efn, chan, func, [iosb], [astadr], [astprm], [p1], [p2], [p3], [p4], [p5], [p6]

efn   =   the number of the event flag that is to be set at request completion. If the event flag number is not specified, the default value of 0 will cause an error in the supervisor.

chan   =   the number of the I/O channel to which the request is directed. Use the ASSIGN service to obtain this number.

func   =   the function code and modifier bits that specify the operation to be performed. The code is expressed symbolically. For reference purposes, the function codes are listed in the notes below. Details on valid I/O function codes and parameters required by each are documented in the VAX/VMS I/O User's Guide.

iosb   =   the address of the quadword I/O status block that is to receive final completion status.

astadr = the entry point address of the AST routine to be executed when the I/O function is completed. If specified, the AST routine executes at the access mode from which the QIO service was requested.

astprm = the AST parameter to be passed to the AST service routine.

p1 to p6 = optional device- and function-specific I/O request parameters.

The first parameter may be specified as P1 or P1V, depending on whether the function code requires an address or a value, respectively. If the keyword is not used, P1 is the default, that is, the first argument is considered an address.

P2 through P6 are always interpreted as values. If they are to be used as addresses, preface the number with #.

Return Status Codes

SS$_NORMAL: Service successfully completed. The I/O request packet was successfully queued.

SS$_ACCVIO: The I/O status block cannot be written by the caller. This status code may also be returned if parameters for device-dependent function codes are incorrectly specified (access violation).

SS$_EXQUOTA: The process has exceeded its buffered I/O quota or direct I/O quota and has disabled resource wait mode with the Set Resource Wait Mode (SETRWM) system service. Or, the process has exceeded its AST limit quota or buffer space quota.

SS$_ILLEFC: An illegal event flag number was specified.

SS$_INSFMEM: Insufficient system dynamic memory is available to complete the service, and the process has disabled resource wait mode with the Set Resource Wait Mode (SETRWM) system service.

SS$_IVCHAN: An invalid channel number was specified. That is, a channel number of 0 or a number larger than the number of channels available.

SS$_NOPRIV: The specified channel does not exist, or was assigned from a more privileged access mode.

SS$_UNASEFC: The process is not associated with the cluster containing the specified event flag.

SS$_ABORT: A network logical link was broken.

Privilege Restrictions
The QIO Request system service can be performed only on assigned
I/O channels and only from access modes that are equal to or more
privileged than the access mode from which the original channel
assignment was made.

Resources Required/Returned
1. Queued I/O requests use three quotas:

    ● the process's quota for buffered I/O or direct I/O
    ● the process's quota for buffer space
    ● the process's AST limit quota, if an AST service
      routine is specified.

2. System dynamic memory is required to construct a data
    base to queue the I/O request. Additional memory may be
    required on a device-dependent basis.

                              NOTES
    1. The specified event flag is set even
        if the service terminates without
        queuing an I/O request.

    2. See Paragraph 9.3.3.4 for
        information on the I/O status block
        format.

    3. Many services return character
        string data, and write the length of
        the data returned in a word provided
        by the caller. Function codes for
        the QIO system service require
        length specifications in longwords.
        If lengths returned by other
        services are to be used as input
        parameters for QIO requests, a
        longword should be reserved to
        ensure that no error occurs when QIO
        reads the length.

    4. For information on performing input
        and output operations on a network,
        see the **VAX-11 DECnet User's Guide**.

    5. The QIO service provides special
        features for diagnostic functions.

9.3.3.1 QIO Service Diagnostic Functions – Diagnostic operations
are performed via physical I/O functions that specify a diagnostic
buffer. The diagnostic buffer must be large enough to receive the
final device register contents at the end of the operation.

A diagnostic buffer is specified by parameter 6 for all physical I/O functions. If this parameter is nonzero, then a diagnostic buffer is specified and the issuing process must have the diagnostic privilege.

Specification of a diagnostic buffer address causes the QIO system service to allocate a buffer and store the address of the buffer in the I/O packet (IRP$L_DIAGBUF). The virtual address of the requester's buffer bit is set in the I/O packet status word. When the I/O operation is completed, the final device register contents are stored in the buffer. The I/O packet is submitted to the I/O posting routine. The I/O posting routine determines that the diagnostic buffer bit is set and transfers the information to the requester's buffer. The allocated buffer is then returned to the dynamic storage pool. The information transferred to the requester's buffer has the format shown in Figure 9-2.

```
31                                          00
+------------------------------------------+
|                                          |
|         OPERATION START TIME             |
|           IN 64 BIT FORMAT               |
|                                          |
+------------------------------------------+
|                                          |
|       OPERATION COMPLETION TIME          |
|           IN 64 BIT FORMAT               |
|                                          |
+------------------------------------------+
|                                          |
|      FINAL ERROR COUNTER CONTENTS        |
+------------------------------------------+
|                                          |
|      NUMBER OF DEVICE REGISTERS          |
+------------------------------------------+
|                                          |
|         DEVICE REGISTERS,                |
|         ONE PER LONGWORD                 |
|                                          |
|                  .                       |
|                  .                       |
|                                          |
+------------------------------------------+
```
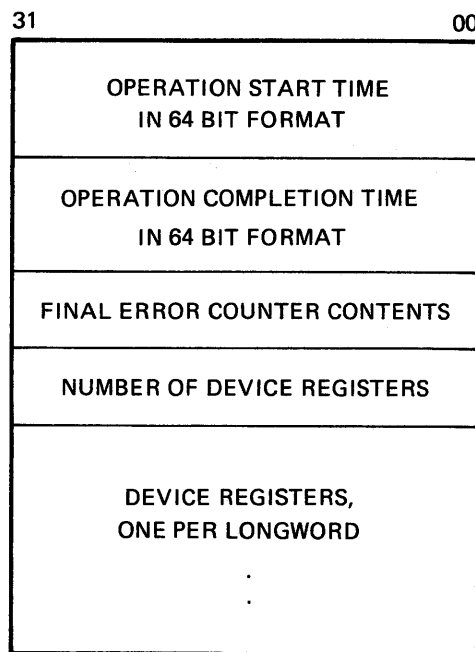
TK-3002

Figure 9-2   Queue I/O Diagnostic Buffer Format

9.3.3.2   I/O Function Encoding – I/O functions fall into three groups that correspond to the three I/O transfer modes (physical, logical, and virtual).

Depending on the device to which it is directed, an I/O function can be expressed in one, two, or all three transfer modes.

I/O functions are described by 16-bit values that are symbolically expressed. They specify the particular I/O operation to be performed and any optional function modifiers. Figure 9-3 shows the format of the 16-bit function value.
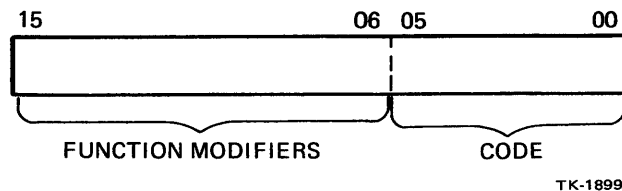
Figure 9-3   I/O Function Format

Symbolic names are described by the $IODEF macro. See the <u>VAX/VMS System Services Reference Manual</u>.

Function Codes – The low-order six bits of the function value make up a code that specifies the particular operation to be performed. For example, the code for read logical block is expressed as IO$_READLBLK. Table 9-1 lists the symbolic values for read and write I/O functions in the three transfer modes.

Table 9-1   Read and Write I/O Functions

Physical I/O	Logical I/O	Virtual I/O
IO$_READPBLK IO$_WRITEPBLK	IO$_READLBLK IO$_WRITELBLK	IO$_READVBLK IO$_WRITEVBLK

The set mode I/O function has a code of IO$_SETMODE.

Function codes are defined for all supported devices. Although some of the function codes (for example, IO$_READVBLK and IO$_WRITEVBLK) are used with several types of devices, most are device-dependent. That is, they perform functions specific to particular types of devices. For example, IO$_CREATE is a device-dependent function code. It is used only with file-structured devices such as disks and magnetic tapes.

Function Modifiers - The high-order 1Ø bits of the function value are function modifiers. These are individual bits that alter the basic operation to be performed. For example, the function modifier IO$M_NOECHO can be specified with the function IO$_READPBLK to a terminal. When used together, the two values are written as IO$_READPBLK!IO$M_NOECHO. This means that data typed at the terminal keyboard is entered in the user buffer but not echoed (displayed) at the terminal. Figure 9-4 shows the format of function modifiers.
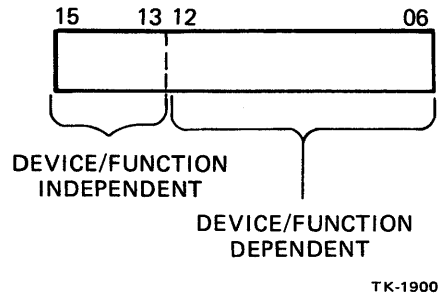
```
15    13 12               06
 ┌──────┬─────────────────┐
 │      ┊                 │
 └──────┴─────────────────┘
  ╰──╮╯ ╰────────╮────────╯
DEVICE/FUNCTION         │
  INDEPENDENT           │
                        │
              DEVICE/FUNCTION
                DEPENDENT

              TK-1900
```

Figure 9-4   Function Modifier Format

As shown, bits 13 through 15 are device/function independent bits, and bits 6 through 12 are device/function dependent bits. However, device/function dependent bits have the same function, whenever possible, for different device classes. For example, the function modifier IO$M_ACCESS is used with both disk and magnetic tape devices to cause a file to be accessed during a create operation. Device/function dependent bits always have the same function within the same device class. Most function modifiers are device/function dependent.

There are two device/function independent modifier bits: IO$M_INHRETRY and IO$M_DATACHECK. IO$M_INHRETRY is used to inhibit all error recovery. If any error occurs, and this modifier bit is specified, the operation is immediately terminated and a failure status is returned in the I/O status block. IO$M_DATACHECK is used to compare the data in memory with that on a disk or magnetic tape.

A list of function codes and function modifiers follows in Tables 9-2 through 9-9. The arguments (P1-P6) are also listed. These codes and modifiers are grouped according to device types. Note that the arguments required for the QIO service depend on the function specified. See the VAX/VMS I/O User's Guide for further details.

Table 9-2   Terminal I/O Driver Functions

Function	Parameters	Modifiers
IO$_READVBLK IO$_READLBLK IO$_READPROMPT IO$_READPBLK	P1=buffer address P2=buffer size P3=time-out P4=read terminator block address P5=prompt string buffer address[1] P6=prompt string buffer size[1]	IO$M_CVTLO IO$M_DISABLMBX IO$M_NOECHO IO$M_NOFILTR IO$M_PURGE IO$M_TIMED IO$M_TRMNOECHO
IO$_WRITEVBLK IO$_WRITELBLK IO$_WRITEPBLK	P1=buffer address P2=buffer size P3=(ignored) P4=carriage control specifier[2]	IO$M_CANCTRLO IO$M_ENABLMBX IO$M_NOFORMAT
IO$_SETCHAR IO$_SETMODE	P1=address of characteristics buffer P2=(ignored) P3=speed specifier P4=fill specifier P5=parity flags	

IO$_SETMODE!IO$M_HANGUP (none)
IO$_SETCHAR!IO$M_HANGUP
IO$_SETMODE!IO$M_CTRLCAST
               P1=AST service routine address
IO$_SETMODE!IO$M_CTRLYAST
               P2=AST parameter
IO$_SETCHAR!IO$M_CTRLYAST
               P3=access mode to deliver AST

[1]Only for IO$_READPROMPT.
[2]Only for IO$_WRITEBLK and IO$_WRITEVBLK.

Table 9-3   Disk I/O Driver Functions

Function	Parameters	Modifiers
IO$_READVBLK IO$_READLBLK IO$_READPBLK IO$_WRITEVBLK IO$_WRITELBLK IO$_WRITEPBLK	P1=buffer address P2=byte count P3=disk address	IO$M_DATACHECK IO$M_INHRETRY IO$M_INHSEEK[1]
IO$_SETMODE IO$_SETCHAR	P1=characteristic buffer address	IO$_INHRETRY
IO$_CREATE IO$_ACCESS IO$_DEACCESS IO$_MODIFY IO$_DELETE	P1=FIB descriptor address P2=file name string address P3=result string length address P4=result string descriptor address P5=attribute list address	IO$M_CREATE[2] IO$M_ACCESS[2] IO$M_DELETE[3]

[1]Only for IO$_READPBLK and IO$_WRITEPBLK.
[2]Only for IO$_CREATE and IO$_ACCESS.
[3]Only for IO$_CREATE and IO$_DELETE.

Table 9-4 Magnetic Tape I/O Driver Functions

Function	Parameters	Modifiers
IO$_READVBLK IO$_READLBLK IO$_READPBLK IO$_WRITEVBLK IO$_WRITELBLK IO$_WRITEPBLK	P1=buffer address P2=byte count	IO$M_DATACHECK IO$M_INHRETRY IO$M_REVERSE[1] IO$M_INHEXTGAP[2]
IO$_SETMODE IO$_SETCHAR	P1=characteristics buffer address	IO$M_INHRETRY IO$M_INHEXTGAP

IO$_CREATE
IO$_ACCESS
IO$_DEACCESS
IO$_MODIFY
IO$_ACPCONTROL
	P1=FIB descriptor address
	P2=file name string
	    address
	P3=result string length
	    address
	P4=result string descriptor
	    address
	P5=attribute list address
		IO$M_CREATE[3]
		IO$M_ACCESS[3]
		IO$M_DMOUNT[4]

IO$_SKIPFILE	P1=skip n tape marks	IO$M_INHRETRY
IO$_SKIPRECORD	P1=skip n records	IO$M_INHRETRY
IO$_MOUNT	(none)	
IO$_REWIND IO$_REWINDOFF	(none)	IO$M_INHRETRY IO$M_NOWAIT
IO$_WRITEOF	(none)	IO$M_INHEXTGAP IO$M_INHRETRY
IO$_SENSEMODE	(none	IO$_INHRETRY

[1] Only for read functions.
[2] Only for write functions.
[3] Only for IO$_CREATE and IO$_ACCESS.
[4] Only for IO$_ACPCONTROL.

Table 9-5   Line Printer I/O Driver Functions

Function	Parameters	Modifiers
IO$_WRITEVBLK IO$_WRITELBLK IO$_WRITEPBLK	P1=buffer address P2=buffer size P3=ignored P4=carriage control specifier[1]	(none)
IO$_SETMODE IO$_SETCHAR	P1=characteristics buffer address	(none)

[1]Only for IO$_WRITEVBLK and IO$_WRITELBLK

Table 9-6   Card Reader I/O Driver Functions

Functions	Parameters	Modifiers
IO$_READLBLK IO$_READVBLK IO$_READPBLK	P1=buffer address P2=byte count	IO$M_BINARY IO$M_PACKED
IO$_SETMODE IO$_SETCHAR	P1=characteristics buffer address	(none)
IO$_SENSEMODE	(none)	

Table 9-7   Mailbox I/O Driver Functions

Function	Parameters	Modifiers
IO$_READVBLK IO$_READLBLK IO$_READPBLK	P1=buffer address P2=buffer size	IO$M_NOW
IO$_WRITEVBLK IO$_WRITELBLK IO$_WRITEPBLK		
IO$_WRITEOF	none	

IO$_SETMODE!IO$M_READATTN
        P1=AST address
IO$_SETMODE!IO$M_WRTATTN
        P1=AST parameter		

Table 9-8   DMC-11 I/O Driver Functions

Function	Parameters	Modifiers
IO$_READLBLK IO$_READPBLK IO$_READVBLK IO$_WRITELBLK IO$_WRITEPBLK IO$_WRITEVBLK	P1=buffer size P2=message size P6=optional diagnostic buffer[1]	IO$M_DISABLMBX[2] IO$M_NOW[2] IO$M_ENABLMBX[3]
IO$_SETMODE IO$_SETCHAR	P1=characteristics buffer address	

IO$_SETMODE!IO$M_ATTNAST
            P1=AST service routine address
IO$_SETCHAR!IO$M_ATTNAST
            P2=(ignored)
            P3=AST access mode		

IO$_SETMODE!IO$M_SHUTDOWN P1=characteristics block address IO$_SETCHAR!IO$M_SHUTDOWN		

IO$_SETMODE!IO$M_STARTUP
            P1=characteristics block address
IO$_SETCHAR!IO$M_STARTUP
            P2=(ignored)
            P3=receive message blocks		

[1]Only for IO$_READPBLK and IO$_WRITEPBLK.
[2]Only for IO$_READLBLK and IO$_READPBLK.
[3]Only for IO$_WRITELBLK and IO$_WRITEPBLK.

Table 9-9   ACP Interface Driver Functions

Function	Parameters	Modifiers

IO$_CREATE
IO$_ACCESS
IO$_DEACCESS
IO$_MODIFY
IO$_DELETE
IO$_ACPCONTROL
	P1=FIB descriptor address
	P2=file name string
	   address
	P3=result string length
	   address
	P4=result string descriptor
	   address
	P5=address of attribute list
		IO$M_CREATE[1]
		IO$M_ACCESS[1]
		IO$M_DELETE[2]
		
		IO$M_DMOUNT[3]

IO$_MOUNT	(none)	

[1]Used only with IO$_ACCESS and IO$_CREATE.
[2]Used only with IO$_CREATE and IO$_DELETE.
[3]Used with IO$_ACPCONTROL.

9.3.3.3   Synchronizing I/O Completion - The QIO system service
returns control to the calling program as soon as the I/O request
is queued.   The status code returned in RØ indicates whether or
not the request was queued successfully. The program must perform
the following two steps to ensure proper synchronization of the
I/O operation with respect to the program.

   1.   Test whether the I/O operation has been queued
        successfully.

   2.   Test whether the I/O operation itself has been completed
        successfully.

Optional arguments to the QIO service provide techniques for
synchronizing I/O completion. There are three methods you can use
to test for the completion of an I/O request:

   a.   Specify the number of an event flag to be set when the
        I/O operation is completed.

   b.   Specify the address of an AST routine to be executed when
        the I/O operation is completed.

   c.   Specify the address of an I/O status block to be posted
        when the I/O operation is completed.

Example 9-9 shows how you can use these three techniques in three
unrelated programs. The following notes are keyed to the example.

   1.   When you code an event flag number as an argument, QIO
        clears the event flag when it queues the I/O request.
        When the I/O operation is completed, the flag is set.

   2.   In this example, the program issues two I/O requests.   A
        different event flag is specified for each request.

   3.   The Wait for Logical AND of Event Flags (WFLAND) system
        service places the process in a wait state until both I/O
        operations are complete. The EFN argument indicates that
        the event flags are both in cluster 1; the MASK argument
        indicates the flags that the service is to wait for.

   4.   When you code the ASTADR argument to the QIO system
        service, the system interrupts the process when the I/O
        operation is completed and passes control to the
        specified AST service routine.

   5.   The QIO system service call specifies the address of the
        AST routine, TTAST, and a parameter to pass as an
        argument to the AST service routine. When QIO returns
        control, the process continues execution.

   6.   When the I/O operation is completed, the routine TTAST is
        called, and it responds to the I/O completion.

When this routine has finished executing, control returns to the process at the point at which it was interrupted.

For more information on ASTs and how to code an AST service routine, refer to Paragraph 3.2 of the <u>VAX/VMS System Services Reference Manual</u>.

7. An I/O status block is a quadword structure that the system uses to post the status of an I/O operation. The quadword area must be defined in your program.

8. TTIOSB defines the I/O status block for this I/O operation. The IOSB argument in the QIO system service refers to this quadword.

9. QIO clears the quadword when it queues the I/O request. When the request is successfully queued, the program continues execution.

10. The process polls the I/O status block. If the low-order word still contains 0, it indicates that the I/O operation has not yet been completed. In program C, the program loops until the request is complete.

Program A: Using Event Flags

```
(1)     $QIO_S  EFN=#32,...           ; Issue 1st I/O request.
        BLBC  R0,ERROR               ; Queued successfully?
(2)     $QIO_S EFN=#33,...           ; Issue 2nd I/O request.
        BLBC  R0,ERROR               ; Queued successfully?
(3)     $WFLAND_S EFN=#32,MASK=^B11  ; Wait till both are done.
          .
          .
          .
```

Program B: Using An AST Routine

```
(4)(5) $QIO_S   ...,ASTADR=TTAST,ASTPRM=#1,...
                                     ; I/O with AST
        BLBC  R0,ERROR               ; Queued successfully?
          .                          ; Continue.
          .
          .
(6)     RET
          .
          .
```

```
          .
TTAST:  .WORD 0                       ; AST service routine entry
          .                          ; mask
          .                          ; Handle I/O completion.
          .
        RET                          ; end of service routine
          .
```

```
                        •
                        •
  ⑦
  Program C: USING THE I/O STATUS BLOCK

  ⑧                     •
                        •
                        •
  TTIOSB: .BLKQ 1                          ; I/O status block
                        •
                        •
                        •
  ⑨         $QIO_S ...,IOSB=TTIOSB,...     ; Issue I/O request.
            BLBC  R0,ERROR                 ; Queued successfully?
                        •                  ; Continue.
  ⑩                     •
  10$:      TSTW   TTIOSB                   ; Is I/O done yet?
            BEQL   10$                      ; No, loop till done.
            CMPW   TTIOSB,#SS$_NORMAL       ; I/O successful?
            BNEQ   ERROR                    ; No, error.
                        •                  ; Yes, handle it.
                        •
                        •
```

Example 9-9 Synchronizing I/O Completion, Three Methods

9.3.3.4   I/O Completion Status – When an I/O operation is completed, the system posts the completion status in the I/O status block, if one is specified. The completion status indicates whether or not the operation was actually completed successfully, the number of bytes that were transferred, and additional device-dependent return information.

Figure 9-5 shows the format for the I/O status block.

```
 31                      16 15                    00
 ┌──────────────────────┬─────────────────────────┐
 │     BYTE COUNT       ┆        STATUS           │
 ├──────────────────────┴─────────────────────────┤
 │      DEVICE-DEPENDENT INFORMATION               │
 └─────────────────────────────────────────────────┘
```

                                         TK-1901

Figure 9-5 I/O Status Block Format

The first word contains a system status code indicating the success or failure of the operation. The status codes used are the same as for all returns from system services; for example, SS$_NORMAL indicates successful completion.

The second word contains the number of bytes that were actually transferred in the I/O operation.

The second longword contains device-dependent return information.

To ensure successful I/O completion and the integrity of data transfers, the IOSB should be checked following I/O requests, particularly for device-dependent I/O functions. For complete details on how to use the I/O status block, refer to the VAX/VMS I/O User's Guide.

9.3.4 $QIOW_x Queue I/O Request and Wait for Event Flag

This macro calls a VMS service that combines the Queue I/O Request and the Wait for Event Flag system services. You should use this macro when your program must wait for I/O completion before proceeding. It takes the same arguments as $QIO_x.

$QIOW_x efn, chan, func, [iosb], [astadr], [astprm], [p1], [p2], [p3], [p4], [p5], [p6]

efn   =   the number of the event flag that is to be set at I/O completion. You must specify the event flag number. The default value of 0 will cause supervisor errors.

chan   =   the number of the I/O channel to which the request is directed.

func   =   the function code and modifier bits that specify the operation to be performed. The code is expressed symbolically.

iosb   =   the address of the quadword I/O status block that is to receive final completion status.

astadr   =   the entry point address of the AST routine to be executed when the I/O is completed. If specified, the AST routine executes at the access mode from which the QIOW service was requested.

astprm   =   the AST parameter to be passed to the AST completion routine.

p1 - p6   =   optional device-specific and function-specific I/O request parameters.

For return status codes, privilege restrictions, resources required and returned, and notes, refer to the description of the QIO system service in Paragraph 9.3.3.

9.3.5 $GETCHN Get I/O Channel Information

The Get I/O Channel Information system service returns information about a device to which an I/O channel has been assigned. Two sets of information may be requested:

1.   The primary device characteristics,
2.   The secondary device characteristics.

In most cases, the two sets of characteristic information are identical. However, the two sets provide different information in the following cases:

1. If the device has an associated mailbox, the primary characteristics are those of the assigned device and the secondary characteristics are those of the associated mailbox.

2. If the device is a spooled device, the primary characteristics are those of the intermediate device and the secondary characteristics are those of the spooled device.

3. If the device represents a logical link on the network, the secondary characteristics contain information about the link.

$GETCHN chan ,[prilen] ,[pribuf] ,[seclen] ,[secbuf]

chan   =   the channel number. The channel number of a device is returned by the Assign I/O Channel (ASSIGN) system service.

prilen =   the address of the word to receive the length of the primary characteristics.

pribuf =   the address of the character string descriptor pointing to the buffer that is to receive the primary device characteristics. An address of 0 (the default) implies that no buffer is specified.

seclen =   the address of the word to receive the length of the secondary characteristics.

secbuf =   the address of the character string descriptor pointing to the buffer that is to receive the secondary device characteristics. An address of 0 (the default) implies that no buffer is specified.

Return Status Codes
SS$_BUFFEROVF: Service successfully completed. The device information returned overflowed the buffers provided and has been truncated.

SS$_NORMAL: Service successfully completed.

SS$_ACCVIO: A buffer descriptor cannot be read, or a buffer or buffer length cannot be written, by the caller.

SS$_IVCHAN: An invalid channel number was specified. An invalid channel number is 0 or a number larger than the number of channels available.

SS$_NOPRIV: The specified channel is not assigned or was assigned from a more privileged access mode.

Privilege Restrictions
The Get I/O Channel information service can be performed only on assigned channels and from access modes that are equal to or more privileged than the access mode from which the original channel assignment was made.

Format of Device Information
The GETCHN system service returns information in a user-supplied 53-byte buffer. Symbolic names defined in the $DIBDEF macro represent offsets from the beginning of the buffer.

The field offset names, lengths, and contents are listed below.

Field Name	Length (bytes)	Contents
DIB$L_DEVCHAR	4	Device characteristics
DIB$B_DEVCLASS	1	Device class
DIB$B_TYPE	1	Device type
DIB$W_DEVBUFSIZ	2	Device buffer size
DIB$L_DEVDEPEND	4	Device dependent information
DIB$W_UNIT	2	Unit number
DIB$W_DEVNAMOFF	2	Offset to device name string
DIB$L_PID	4	Process identification of device owner
DIB$L_OWNUIC	4	UIC of device owner
DIB$W_VPROT	2	Volume protection mask
DIB$W_ERRCNT	2	Hard error count for device
DIB$L_OPCNT	4	Operation count
DIB$W_VOLNAMOFF	2	Offset to volume label string

The device name string and volume label string are returned in the buffer as counted ASCII strings and must be located by their offset values.

Figure 9-6 shows the buffer layout. The offsets are displacements from the front of the buffer. Offsets of 0 show missing fields. Both the device name and the volume label are stored as counted strings. You must locate them through the offset values. If the service returns both a volume label and a device name, the buffer length is 64 bytes.

9.3.6 $CANCEL_x Cancel I/O on Channel
The Cancel I/O On Channel system service cancels all pending I/O requests on a specific channel. In general, this includes all I/O requests that are queued, as well as the request currently in progress.

$CANCEL_x chan

chan = the number of the I/O channel assigned to the device for which I/O operation is to be cancelled.
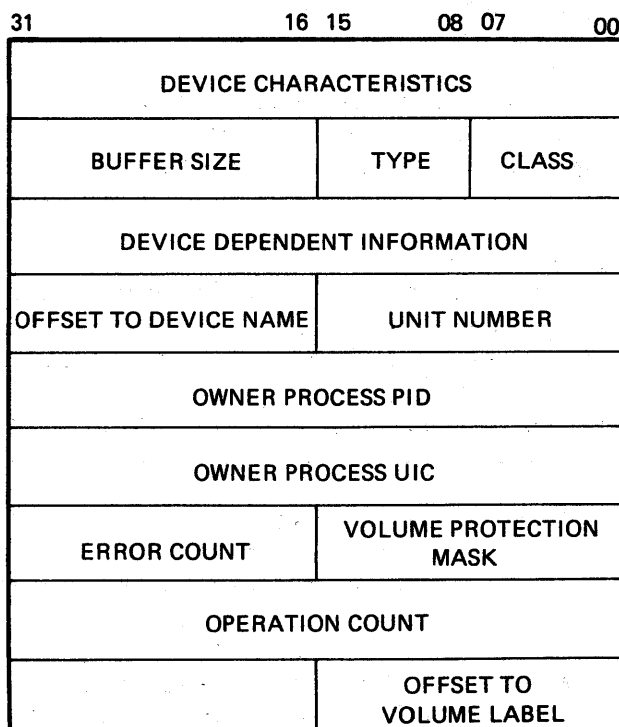
```
    31                      16 15      08 07        00
   ┌──────────────────────────────────────────────┐
   │            DEVICE CHARACTERISTICS              │
   ├──────────────────────┬──────────┬─────────────┤
   │      BUFFER SIZE      │   TYPE   │    CLASS    │
   ├──────────────────────┴──────────┴─────────────┤
   │         DEVICE DEPENDENT INFORMATION           │
   ├──────────────────────┬─────────────────────────┤
   │ OFFSET TO DEVICE NAME │      UNIT NUMBER         │
   ├──────────────────────┴─────────────────────────┤
   │             OWNER PROCESS PID                   │
   ├─────────────────────────────────────────────────┤
   │             OWNER PROCESS UIC                   │
   ├──────────────────────┬─────────────────────────┤
   │      ERROR COUNT      │   VOLUME PROTECTION      │
   │                       │        MASK             │
   ├──────────────────────┴─────────────────────────┤
   │              OPERATION COUNT                    │
   ├──────────────────────┬─────────────────────────┤
   │                       │      OFFSET TO           │
   │                       │    VOLUME LABEL          │
   └──────────────────────┴─────────────────────────┘
                                           TK-1902
```

Figure 9-6   Buffer Layout Supplied by the GETCHN System Service

Return Status Codes

SS$_NORMAL: Service successfully completed.

SS$_EXQUOTA: The process has exceeded its quota for direct I/O. In this context, direct I/O refers to use of the direct data path on the Unibus interface.

SS$_INSFMEM: Insufficient system dynamic memory is available to cancel the I/O, and the process has disabled resource wait mode with the Set Resource Wait Mode (SETRWM) system service.

SS$_IVCHAN: An invalid channel was specified, that is, a channel number of 0 or a number larger than the number of channels available.

SS$_NOPRIV: The specified channel is not assigned, or was assigned from a more privileged access mode.

Privilege Restrictions
I/O can be canceled only from an access mode equal to or more privileged than the access mode from which the original channel assignment was made.

NOTES

1. When a request currently in progress is canceled, the driver is notified immediately. Actual cancellation may or may not occur immediately depending on the logical state of the driver. When cancellation does occur, the same action as that taken for queued requests is performed:

    a. The specified event flag is set.

    b. The first word of the I/O status block, if specified, is set to SS$_CANCEL.

    c. The AST, if specified, is queued.

    Proper synchronization between this service and the actual canceling of I/O requests requires the issuing process to wait for I/O completion in the normal manner and to then note that the I/O has been canceled.

2. If the I/O operation is a virtual I/O operation involving a disk or tape ancillary control process, the I/O operation cannot be canceled. In the case of a magnetic tape, however, cancellation may occur if the device driver is hung.

3. Outstanding I/O requests are automatically canceled at image exit.

9.4   EVENT FLAG SERVICE MACROS

Event flags are status posting bits maintained by VAX/VMS. You can require some system services to set specific event flags in order to indicate the completion or occurrence of an event. The program that calls the service can test these flags. For example, the QIO system service sets an event flag when the requested input or output operation has been completed. You can use the event flag services to perform any of the following functions.

- Set or clear specific flags.

- Test the current status of flags.

- Place a process in a wait state pending the setting of a specific flag or group of flags.

The supervisor provides two event flag clusters, 0-31 and 32-63. However, event flags 1-23 are restricted to communication between the program and the operator, and event flags 24-31 are restricted for use by VAX/VMS. This leaves flags 32-63 for exclusive use by the diagnostic program developer. Do not use event flag 0. This is reserved for the supervisor.

9.4.1 $SETEF_x Set Event Flag

This macro calls a service that sets the event flag specified. Any processes waiting for the event flag are made executable.

$SETEF efn

<u>efn</u> = the number of the event flag to be set. Do not use event flag 0.

Return Status Codes

SS$_WASCLR: Service successfully completed. The specified event flag was previously 0.

SS$_WASSET: Service successfully completed. The specified event flag was previously 1.

SS$_ILLEFC: An illegal event flag number was specified.

SS$_UNASEFC: The process is not associated with the cluster containing the specified event flag.

9.4.2 $CLREF_x Clear Event Flag
This macro calls a service that clears an event flag.

$CLREF_x efn

efn = the number of the event flag to be cleared.

Return Status Codes
SS$_WASCLR: Service successfully completed. The specified event flag was previously 0.

SS$_WASSET: Service successfully completed. The specified event flag was previously 1.

SS$_ILLEFC: An illegal event flag number was specified.

SS$_UNASEFC: The process is not associated with the cluster containing the specified event flag.

9.4.3 $READEF_x Read Event Flags
This macro calls a service that returns the current status of all 32 event flags in an event flag cluster.

$READEF_X efn, state

efn   = the number of any event flag within the cluster to be read. A flag number of 0 through 31 specifies cluster 0; and a flag number of 32 through 63 specifies cluster 1.

state   = the address of a longword to receive the current status of all event flags in the cluster.

Return Status Codes
SS$_WASCLR: Service successfully completed. The specified event flag is clear.

SS$_WASSET: Service successfully completed. The specified event flag is set.

SS$_ACCVIO: The longword that is to receive the current state of all event flags in the cluster cannot be written by the caller.

SS$_ILLEFC: An illegal event flag number was specified.

SS$_UNASEFC: The process is not associated with the cluster containing the specified event flag.

9.4.4 $WAITFR_x Wait for Single Event Flag
This macro calls a service that tests a specific event flag. If the flag is set, control returns to the calling program immediately. If the flag is cleared, the process is placed in a wait state until the event flag is set.

$WAITFR_x efn

efn = the number of the event flag for which to wait.

Return Status Codes
SS$_NORMAL: Service successfully completed.

SS$_ILLEFC: An illegal event flag number was specified.

SS$_ UNASEFC: The process is not associated with the cluster containing the specified event flag.

> **NOTE**
> The wait state caused by this service can be interrupted by an asynchronous system trap (AST), if (1) the access mode at which the AST executes is less than or equal to the access mode from which the wait was issued and (2) the process is enabled for ASTs at that access mode.
>
> When the AST service routine completes execution, the system repeats the WAITFR request. If the event flag has been set, the process resumes execution.

9.4.5 $WFLAND_x Wait for Logical AND of Event Flags

This macro calls a service that allows the program to specify a mask of event flags for which it wishes to wait. If all of the flags indicated by the mask are set, control returns immediately to the calling program. Otherwise, the program is placed in a wait state until the flags are all set.

$WFLAND_x efn, mask

efn = the number of any event flag within the cluster being used.

mask = the 32-bit mask in which bits set to 1 indicate the event flags of interest.

Return Status Codes

SS$_NORMAL: Service successfully completed.

SS$_ILLEFC: An illegal event flag number was specified.

SS$_UNASEFC: The process is not associated with the cluster containing the specified event flag.

NOTE

The wait state caused by this service can be interrupted by an AST, if (1) the access mode at which the AST executes is less than or equal to the access mode from which the wait was issued and (2) the process is enabled for ASTs at that access mode.

When the AST service routine completes execution, the system repeats the WAITFR request. If the event flag has been set, the process resumes execution.

9.4.6 $WFLOR_x Wait for Logical OR of Event Flags

This macro calls a service that tests the event flags specified by a mask within a specified cluster. The service returns control to the calling program immediately if any of the flags are set. Otherwise, the service places the program in a wait state until one of the selected event flags is set.

$WFLOR_x efn, mask

efn = the number of any event flag within the cluster being used.

mask = a 32-bit mask in which bits set to 1 show the event flags of interest.

Return Status Codes

SS$_NORMAL: Service successfully completed.

SS$_ILLEFC: An illegal event flag number was specified.

SS$_UNASEFC: The process is not associated with the cluster containing the specified event flag.

> **NOTE**
> The wait state caused by this service can be interrupted by an AST, if (1) the access mode at which the AST executes is less than or equal to the access mode from which the wait was issued and (2) the process is enabled for ASTs at that access mode.
>
> When the AST service routine completes execution, the system repeats the WAITFR request. If the event flag has been set, the process resumes execution.

9.5   TIMER SERVICE MACROS

The timer and time conversion services enable the program to schedule program activity according to clock time. They are available to level 2, 2R, and 3 diagnostic programs. You may schedule events according to the time of day or the passage of a time interval. The timer services enable you to schedule the setting of an event flag or the queueing of an asynchronous system trap for the program. The timer services also enable you to cancel a pending request that has not yet been honored. These services require you to specify the time in a 64 bit system format.

You can use the time conversion services to perform two functions.

1.   Obtain the time in the system format.
2.   Convert an ASCII string into the system format.

VAX/VMS maintains the current date and time (using a 24 hour clock) in 64 bit format. The time value is a binary number representing 100 ns offsets from the system base date and time. This is 00:00 o'clock, November 17, 1858. All time values passed to the timer services must also be expressed in the 64 bit system format. A time value may be expressed in three ways.

1.   An absolute time, which is a specific date and time of day. Absolute times are always positive values. The standalone supervisor does not support absolute times.

2.   A delta time, which is a future offset (number of days, hours, minutes, seconds, and so on) from the current time. Delta times are always expressed as negative values.

3. A 0, which indicates that the system service should use
    the current date and time. The standalone supervisor
    does not support this function.

9.5.1 $GETTIM_x Get Time

This macro calls a service that furnishes the current system time
in the 64 bit system format suitable for input to the Set Timer
(SETIMR) system service.

$GETTIM_x timadr

timadr = the address of a quadword that is to receive the current
          time in 64 bit format.

Return Status Codes
SS$_NORMAL: Service successfully completed.

SS$_ACCVIO: The quadword to receive the time cannot be written by
the caller.

9.5.2 $BINTIM_x Convert ASCII String to Binary Time

This macro calls a service that converts an ASCII string to an
absolute or delta time value in the 64 bit system format suitable
for input to the Set Timer (SETIMR) service.

$BINTIM_x timbuf, timadr

timbuf = the address of the character string descriptor pointing
          to the absolute or delta time value to be converted.
          The ASCII string value must be in one of the formats
          shown in Note 1, which follows.

timadr = the address of a quadword that is to receive the
          converted time in 64 bit format.

Return Status Codes
SS$_NORMAL: Service successfully completed.

SS$_IVTIME: The syntax of the specified ASCII string is invalid,
or the time component is out of range.

NOTES
1. The required ASCII input strings
    have the following formats:

    Absolute Time:
    dd-mmm-yyyy hh:mm:ss.cc

    Delta Time:
    dddd hh:mm:ss.cc

    Table 9-10 shows the function of
    each string in the format.

2. The following syntax rules apply to the coding of the ASCII input string:

- Any of the fields of the date and time can be omitted.

  For absolute time values, the BINTIM service supplies the current system date and time for nonspecified fields. Trailing fields can be truncated. If leading fields are omitted, the punctuation (hyphens, blanks, colons, periods) must be specified. For example, the string

  --12:00:00.00

  results in an absolute time of 12:00 on the current day.

  For delta time values, the BINTIM service defaults nonspecified fields to 0. Trailing fields can be truncated. If leading fields are omitted from the time value, the punctuation (blanks, colons, periods) must be specified. For example, the string

  0::10

  results in a delta time of 10 seconds.

- For both absolute and delta time values, there can be any number of leading blanks, and any number of blanks between fields normally delimited by blanks. However, there may be no embedded blanks within either the date or time fields.

**Table 9-10   Field Function for ASCII Absolute
or Delta Time Values**

Field	Length (Bytes)	Contents	Range of Values
dd	2	day of month	1 - 31
-	1	hyphen	Required syntax
mmm	3	month	JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC
-	1	hyphen	Required syntax
yyyy	4	year	0 -
blank	n	blank	Required syntax (one or more blanks)
hh	2	hour	0 -23
:	1	colon	Required syntax
mm	2	minutes	0 -59
:	1	colon	Required syntax
ss	2	seconds	0 - 59
.	1	period	Required syntax
cc	2	hundredths of seconds	0 - 99
dddd	4	number of days (in 24-hour units)	0 - 9999

Example 9-10 shows how you can translate an ASCII absolute time value to the 64-bit system format.

```
ANOON:   DESCRIPTOR <-- 12:00:00.00>      ; ASCII 12 noon
BNOON:   .BLKQ   1                         ; buffer for binary 12
                                           ; noon
           .
           .
           .
         $BINTIM_S TIMBUF=ANOON,TIMADR=BNOON ; Convert time.
```

Example 9-10   Use of $BINTIM_x to Convert an Absolute Time Value
to System Format

When the program calls the BINTIM service, the service returns a 64 bit value representing noon today in the quadword at BNOON. The value in BNOON may then serve as an input to the SETIMR service.

Example 9-11 shows how to use the BINTIM service to translate a time interval (delta time) given in ASCII format to the 64 bit system format.

```
ATENMIN:   DESCRIPTOR <0 00:10:00.00>        ; ASCII ten minutes
BTENMIN:
           .BLKQ    1                        ; buffer for binary ten
                                             ; minutes

                 .
                 .
                 .
   $BINTIM_S TIMBUF=ATENMIN,TIMADR=BTENMIN   ; Convert time.
```

Example 9-11   Use of $BINTIM_x to Convert a Delta Time
Value to System Format

This code returns the delta time of ten minutes to BTENMIN in 64
bit system format.

9.5.3   Specifying Delta Time Values at Assembly Time

You can also specify delta time values at assembly time (instead
of run time) using a MACRO .LONG directive. You can code an
arithmetic expression to represent a time value in terms of 100
ns units. The arithmetic is based on the following formula.

                1 second = 10 million *100 ns

For example, the following statement defines a delta time value of
5 seconds in the 64-bit system format:

FIVESEC: .LONG -10*1000*1000*5,-1 ; five seconds

The value 10 million is expressed as 10*1000*1000 for readability.
Note that the delta time value is negative. The -1 in the second
longword, extends the sign bit.

If you use this notation, however, you are limited to the maximum
number of 100 ns units that can be expressed in a longword. This
is somewhat more than 7 minutes.

9.5.4 $SETIMR_x Set Timer

This macro calls a service that allows a program to schedule the
setting of an event flag and/or the queueing of an asynchronous
system trap (AST) at some future time. You can specify the time
for the event as an absolute time or as a delta time.

$SETIMR_x efn, daytim, [astadr], [reqidt]

efn   =   the number of the event flag to set when the time
            interval expires. You must specify the EFN, since the
            default value is 0 and will cause supervisor errors.

daytim =   the address of the quadword containing the expiration
            time value. A positive time value indicates an absolute
            time at which the timer is to expire. A negative time
            value indicates an offset (delta time) from the current
            time. The time value should be coded in the 64-bit
            system format.

astadr =   the address of the entry mask for an AST service routine
to be called when the time interval expires. If this
argument is not specified, the default value of 0 is
supplied. 0 shows that no AST is to be queued.

reqidt =   a request identification number. The default value is
0. You can specify a unique request identification
number in each Set Timer request. Or, you can give the
same request identification number to related Set Timer
requests. You can use the request identification number
later to cancel Set Timer requests. If an ASTADR
argument is also supplied, the request identification
number is passed to the AST routine.

Return Status Codes

SS$_NORMAL: Service successfully completed.

SS$_ACCVIO: The expiration time cannot be read by the caller.

SS$_EXQUOTA: The process quota for timer entries has been
exceeded, and the process has disabled resource wait mode with the
Set Resource Wait Mode (SETRWM) system service.

SS$_ILLEFC: An illegal event flag number was specified.

SS$_IVTIME: An absolute expiration time that was specified has
already passed, or the time was specified as 0.

SS$_INSFMEM: Insufficient dynamic memory is available to allocate
a timer queue entry, and the process has disabled resource wait
mode with the Set Resource Wait Mode (SETRWM) system service.

SS$_UNASEFC: The process is not associated with the cluster
containing the specified event flag.

Resources Required/Returned

1.   The Set Timer system service requires dynamic memory.

2.   The Set Timer system service uses the process's quota for
timer queue entries.

NOTES
1.   The access mode of the caller is the
access mode of the request and of
the AST.

2.   Use the Convert ASCII String to
Binary Time (BINTIM) system service
to convert a specified ASCII string
to the quadword time format required
as input to the SETIMR service.

9.5.5 $CANTIM_x Cancel Timer Request

The Cancel Timer Request system service cancels all, or a selected subset, of the Set Timer requests previously issued by the current image executing in a process. Cancellation is based on the request identification specified in the Set Timer (SETIMR) system service. If more than one timer request has been given the same request identification, they are all canceled.

Macro Format

$CANTIM_x [reqidt], [acmode]

reqidt = the request identification number of the timer request(s) to be canceled. A value of 0 (the default) indicates that all timer requests are to be canceled.

acmode = the access mode of the request(s) to be canceled. The access mode is maximized with the access mode of the caller. Only those timer requests issued from an access mode equal to or less privileged than the resultant access mode are canceled.

Return Status Codes
SS$_NORMAL: Service successfully completed.

Privilege Restrictions
Timer requests can be canceled only from access modes equal to or more privileged than the access mode from which the requests were issued.

Resources Required/Returned
Canceled timer requests are restored to the process's quota for timer queue entries.

                         NOTE
        Outstanding   timer   requests   are
        automatically canceled at image exit.

9.5.6   Use of the Timer Services

Timer requests made with the Set Timer service are queued. That is, they are ordered for processing according to their expiration times. The number of entries a process can have pending in this timer queue is controlled by a quota.

Example 9-12 shows how a 30 second delay can be coded.

```
WAITIME:
          .LONG    -10*1000*1000*30,-1           ; 30 second wait time
             .
             .
             .
          $SETIMR_S EFN=#36,DAYTIM=WAITIME    ; Set timer.
          BLBC     R0,ERROR                    ; Branch if error.
          $WAITFR_S EFN=#36                    ; Wait 30 seconds.
          BLBC     R0,ERROR                    ; Branch if error.
             .
             .
             .
```

Example 9-12   Use of the SETIMR Service to Create
a 30 Second Delay

Notice that the delta time is given through the MACRO .LONG directive.

The $SETIMR service sets event flag number 36 after 30 seconds. The $WAITFR service delays the program until the flag is set. Control then returns to the program.

Absolute time is used in Example 9-13.

```
ANOON:    DESCRIPTOR  <-- 12:00:00.00>         ; ASCII noon
BNOON:    .BLKQ    1                           ; binary noon
             .
             .
             .
          $BINTIM_S TIMBUF=ANOON,TIMADR=BNOON ; Convert time.
          BLBC     R0,ERROR                    ; Branch if error.
          $SETIMR_S DAYTIM=BNOON,ASTADR=ASTSERV,REQIDT=#12
          BLBC     R0,ERROR                    ; Branch if error.
             .
             .
             .
ASTSERV:
          .WORD    0                           ; entry mask
          CMPL     #12,4(AP)                   ; Check AST parameter.
          BEQL     10$                         ; Go to noon routine.
             .
             .
             .
```

9-41

```
10$:                                        ; Service noon
                                            ; request.
        •
        •
     RET
```

Example 9-13   Use of the SETIMR Service to Call
an AST at an Absolute Time

In this example, the call to BINTIM converts the ASCII string representing 12:00 noon to system format. The value returned in BNOON is used as input to the SETIMR system service.

The AST routine specified in the SETIMR request will be called when the timer expires, that is, at 12:00 noon. The REQIDT argument identifies the timer request. The process continues execution. When the timer expires, it is interrupted by the delivery of the AST. Note that if the current time of day is past noon, the timer expires immediately.

This AST service routine checks the parameter passed by the REQIDT argument and determines, in this example, that it must service the 12:00 noon timer request. When the AST service routine is completed, the process continues execution at the point of interruption.

You could cancel the request shown in Example 9-13 with the following code.

```
$CANTIM_S REQIDT=#12
```

9.6   FORMATTED ASCII OUTPUT SERVICE MACROS
When you wish to send a message to the operator from the diagnostic program, some of the information in the message may be variable strings or numerics. The formatted ASCII output system services enable level 2, 2R, and 3 diagnostic programs to assemble the complete message in the test code and place the text in a buffer, before calling one of the print macros. This procedure may be easier, in some cases, than assembling the message with the print macro in the print routine. See the VAX/VMS System Services Reference Manual for examples.

9.6.1 $FAO_x Formatted ASCII Output
The Formatted ASCII Output system service converts binary values into ASCII characters and returns the converted characters in an output string. It can be used to:

- Insert variable character string data (filename, for example) into an output string.

- Convert binary values into the ASCII representations of their decimal, hexadecimal, or octal equivalents and substitute the results into an output string.

$FAO_xctrstr, [outlen], outbuf, [pl], [p2]..., [pn]

ctrstr = the address of character string descriptor pointing to the control string. The control string consists of the fixed text of the output string and FAO directives.

outlen = the address of a word to receive the actual length of the output string returned.

outbuf = the address of a character string descriptor pointing to the output buffer. The fully formatted output string is returned in this buffer.

pl -- pn = the directive parameters contained in longwords. Depending on the directive, a parameter may be a value that is to be inserted, or a length, or an argument count. Each directive in the control string may require a corresponding parameter or parameters.

Return Status Codes
SS$_NORMAL: Service successfully completed.

SS$_BUFFEROVF: Service successfully completed. The formatted output string overflowed the output buffer, and has been truncated.

SS$_BADPARAM: An invalid directive was specified in the FAO control string.

<div align="center">NOTES</div>

1. The $FAO_S macro form uses a PUSHL instruction for all parameters (P1 through Pn) coded in the macro instruction. If a literal is specified, it must be preceded with a number sign (#) character or loaded into a register.

2. A maximum of 20 parameters can be specified in the $FAO_x macro instruction. If more than 20 parameters are required, use the $FAOL_x macro.

3. The FAO system service executes at the access mode of the caller and does not check whether address arguments are accessible before it executes. Therefore, an access violation causing an exception condition occurs if an input field cannot be read or, in some cases, if an invalid length is specified.

9.6.2 $FAOL_x Formatted ASCII Output with List Parameter
The Formatted ASCII Output with List Parameter macro provides an alternate way to specify input parameters for a call to the FAO system service.

$FAOL_x ctrstr, [outlen], outbuf, prmlst

ctrstr =   the address of character string descriptor pointing to the control string. The control string consists of the fixed text of the output string and conversion directives.

outlen =   the address of a word to receive the actual length of the output string returned.

outbuf =   the address of a character string descriptor pointing to the output buffer. The fully formatted output string is returned in this buffer.

prmlst =   the address of the parameter list of longwords to be used as P1 through Pn.

The parameter list may be a data structure that already exists in a program and from which certain values are to be extracted.

Return Status
Same as for the FAO system service.

9.6.3   FAO Directives
An FAO directive has the format:

!DD

!   (exclamation mark) indicates that the following character or characters are to be interpreted as an FAO directive.

DD   is a 1 character or 2 character code indicating the action that FAO is to perform. Each directive may require one or more input parameters on the call to FAO. Directives must be specified using uppercase letters.

Optionally, a directive may include:

    A repeat count
    An output field length

A repeat count is coded as follows:

    !n(DD)

where n is a decimal value indicating that FAO is to repeat the directive for the specified number of parameters.

An output field length is specified as follows:

    !lengthDD

where length is a decimal value instructing FAO to place the output resulting from a directive into a field of length characters in the output string.

A directive may contain both a repeat count and an output length, as shown below:

    !n(lengthDD)

Repeat counts and output field lengths may be specified as variables, by using a # (number sign) in place of an absolute numeric value. If a # is specified for a repeat count, the next parameter passed to FAO must contain the count. If a # is specified for an output field length, the next parameter must contain the length value.

If a variable output field length is specified with a repeat count, only one length parameter is required. Each output string will have the specified length.

9.6.4   FAO Control String and Parameter Processing

An FAO control string may be any length and may contain any number of FAO directives. The only restriction is on the use of the ! character (ASCII code X'21') in the control string. If a literal ! is required in the output string, the directive !! provides it.

When FAO processes a control string, each character that is not part of a directive is written into the output buffer. When a directive is encountered, it is validated. If it is not a valid directive, FAO terminates and returns an error status code. If the directive is valid, and if it requires one or more parameters, the next consecutive parameters specified are analyzed and processed.

FAO reads parameters from the argument list. It does not check the number of arguments it has been passed. If there are not enough parameters coded in the argument list, FAO will continue reading past the end of the list. It is your responsibility, when coding a call to FAO, to ensure that there are enough parameters to satisfy the requirements of all the directives in the control string.

Table 9-11 summarizes the FAO directives and lists the parameters required by each directive.

Table 9-11 Summary of FAO Directives

Character String Substitution

Directive	Function	Parameters*
!AC	Insert a counted ASCII string.	Address of the string; the first byte must contain the length.
!AD	Insert an ASCII string.	1. Length of string 2. Address of string
!AF	Insert an ASCII string. Replace all nonprintable ASCII codes with periods (.).	1. Length of string 2. Address of string
!AS	Insert an ASCII string.	Address of quadword character string descriptor pointing to the string.

Numeric Conversion (zero-filled)

!OB	Convert a byte to octal.	Value to be converted to ASCII representation.
!OW	Convert a word to octal.	

Directive	Function	Parameters*
!OL	Convert a longword to octal.	
!XB	Convert a byte to hexadecimal.	For byte or word conversion, FAO uses only the low-order byte or word of the longword parameter.
!XW	Convert a word to hexadecimal.	
!XL	Convert a longword to hexadecimal.	
!ZB	Convert an unsigned decimal byte.	
!ZW	Convert an unsigned decimal word.	
!ZL	Convert an unsigned decimal longword.	

Numeric Conversion (blank-filled)

!UB	Convert an unsigned decimal byte.	Value to be converted to ASCII representation.
!UW	Convert an unsigned decimal word.	
!UL	Convert an unsigned decimal longword.	

Table 9-11 Summary of FAO Directives (Cont)

Directive	Function	Parameters*
!SB !SW !SL	Convert a signed decimal byte. Convert a signed decimal word. Convert a signed decimal longword.	For byte or word conversion, FAO uses only the low-order byte or word of the longword parameter.

Output String Formatting

!/ !_ !⊤ !! !%S	Insert new line (CR/LF) Insert a tab. Insert a form feed. Insert an exclamation mark. Insert 'S' if most recently converted numeric value is not 1.	None
!%T !%D	Insert the system time. Insert the system date and time.	Address of a quadword value to be converted to ASCII. If 0 is specified, the current system time is used.

Directive	Function	Parameters*
!n< !>	Define output field width of "n" characters. Format all data and directives within delimiters, <and>, left-justified and blank-filled within the field.	None
!n*c	Repeat the character c in the output string n times	None

Parameter Interpretation

!-	Reuse the last parameter in the list.	None
!+	Skip the next parameter in the list.	None

* If a variable repeat count and/or a variable output field length is specified with a directive, parameters indicating the count and/or length must precede other parameters required by the directive.

9.7 MEMORY MANAGEMENT SERVICE MACROS

The Set Protection on Pages system service allows an image running in a process to change the protection on a page or range of pages.

$SETPRT_x inadr, [retadr], [acmode], prot, [prvprt]

inadr = the address of a 2-longword array containing the starting and ending virtual addresses of the pages on which protection is to be changed. If the starting and ending virtual addresses are the same, a single page is changed. Only the virtual page number portion of the virtual address is used. The low-order 9 bits are ignored.

retadr = the address of a 2-longword array to receive the starting and ending virtual addresses of the pages that have had their protection changed.

acmode = the access mode on behalf of which the request is being made. The specified access mode is maximized with the access mode of the caller. The resultant access mode must be equal to or more privileged than the access mode of the owner of each page in order to change the protection.

prot = the new protection specified in bits 0 through 3 in the format of the hardware page protection. The high-order 28 bits are ignored. Symbolic names defining the protection codes are listed in Note 2. If specified as 0, the default access of kernel read-only is used.

prvprt = the address of a byte to receive the protection previously assigned to the last page whose protection was changed. This argument is useful only when protection for a single page is being changed.

Return Status Codes

SS$_NORMAL: Service successfully completed.

SS$_ACCVIO: 1. The input address array cannot be read by the caller. 2. The output address array or the byte to receive the previous protection cannot be written by the caller. 3. An attempt was made to change the protection of a nonexistent page.

SS$_EXQUOTA: The process exceeded its paging file quota while changing a page in a read-only private section to a read/write page.

SS$_IVPROTECT: The specified protection code has a numeric value of 1 or is greater than 15 (decimal).

SS$_LENVIO: A page in the specified range is beyond the end of the program or control region.

SS$_NOPRIVE: A page in the specified range is in the system address space.

SS$_PAGOWNVIO: Page owner violation. An attempt was made to change the protection on a page owned by a more privileged access mode.

Privilege Restrictions
For pages in global sections, the new protection can alter only the accessibility of the page for modes less privileged than the owner of the page.

<div align="center">NOTES</div>

1. If an error occurs while changing page protection, the return array, if requested, indicates the pages that were successfully changed before the error occurred. If no pages have been affected, both longwords in the return address array contain a -1.

2. Hardware protection code symbols:

Symbol	Meaning
PRT$C_NA	No access
PRT$C_KR	Kernel read only
PRT$C_KW	Kernel write
PRT$C_ER	Executive read only
PRT$C_EW	Executive write
PRT$C_SR	Supervisor read only
PRT$C_SW	Supervisor write
PRT$C_UR	User read only
PRT$C_UW	User write
PRT$C_ERKW	Executive read; kernel write
PRT$C_SRKW	Supervisor read; kernel write
PRT$C_SREW	Supervisor read; executive write
PRT$C_URKW	User read; kernel write
PRT$C_UREW	User read; executive write
PRT$C_URSW	User read; supervisor write

These symbols are defined by $PRTDEF.

9.8   HIBERNATE AND WAKE SERVICE MACROS
A diagnostic program can place itself in a wait state with the Hibernate system service. This is a useful feature for diagnostic programs that test the magnetic media on several devices simultaneously (in parallel). After the transfers have all been started, the program can hibernate.

The wait state can be interrupted for delivery of an AST (asynchronous system trap). When the AST routine completes execution, the program continues hibernation. The program may, however, wake itself in the AST service routine by calling the Wake process system service. Then, after the AST service routine is completed, the program continues execution.

9.8.1. $HIBER_x Hibernate

The Hibernate system service allows a process to make itself inactive but to remain known to the system so that it can be interrupted, for example, to receive ASTs. A hibernate request is a wait-for-wake-event request. When a wake is issued for a hibernating process with the Wake system service, the process continues execution at the instruction following the Hibernate call.

$HIBER_s (no arguments)

Return Status Codes
SS$_NORMAL: Service successfully completed.

<div align="center">NOTES</div>

1. A hibernating process can be swapped out of the balance set if it is not locked into the balance set.

2. The wait state caused by this system service can be interrupted by an AST if (1) the access mode at which the AST is to execute is equal to or more privileged than the access mode from which the hibernate request was issued and (2) the process is enabled for ASTs at that access mode.

    When the AST service routine completes execution, the system reexecutes the HIBER system service on the process's behalf. If a wakeup request has been issued for the process during the execution of the AST service routine (either by itself or another process), the process resumes execution. Otherwise, it continues to hibernate.

3. If one or more wakeup requests are issued for the process while it is not hibernating, the next hibernate call returns immediately. That is, the process does not hibernate. No count is maintained of outstanding wakeup requests.

> 4. Only the _S macro form is provided for the Hibernate system service.

9.8.2. $WAKE_x Wake

The Wake system service activates a process that has placed itself in a state of hibernation with the Hibernate (HIBER) system service.

$WAKE_x [pidadr], [prcnam]

pidadr = the address of a longword containing the process identification of the process to be awakened.

prcnam = the address of a character string descriptor pointing to the process name string. The name is implicitly qualified by the group number of the process issuing the wake.

If neither a process identification nor a process name is specified, the wake request is for the caller.

Return Status Codes

SS$_NORMAL: Service successfully completed.

SS$_ACCVIO: The process name string or string descriptor cannot be read, or the process identification cannot be written, by the caller.

SS$_IVLOGNAM: The specified process name string has a length of 0 or has more than 15 characters.

SS$_NONEXPR: Warning. The specified process does not exist, or an invalid process identification was specified.

SS$_NOPRIV: The process does not have the privilege to wake the specified process.

Privilege Restrictions

User privileges are required to wake:

● Other processes in the same group (GROUP privilege)
● Any other process in the system (WORLD privilege).

NOTE
If one or more wake requests are issued for a process that is not currently hibernating, a subsequent hibernate request will be completed immediately. That is, the process will not hibernate. No count is maintained of outstanding wakeup requests.

This chapter lists all of the macros available to level 2, 2R, and 3 diagnostic programs, in alphabetical order. However, not all of the macros can be used at all program levels. Those macros that are available to one or two program levels only are identified in the descriptions.

For example, most of the VMS system service macros are available only to level 2 and 2R diagnostic programs, and the supervisor channel service macros are available only to level 3 diagnostic programs.

The supervisor service macros take the form $DS_xxxx_x. The $DS_ prefix signifies the diagnostic supervisor. The suffix, _x, shows that the macro may end in any of four ways, depending on the function required.

>   _L -- generate an argument list
>   _DEF -- generate symbols and offsets
>   _S -- call the service with CALLS
>   _G -- call the service with CALLG

The utility macros take the form $DS_xxxx. As in the case of the supervisor service macros, the $DS_ prefix signifies the diagnostic supervisor. The lack of a suffix indicates that the macro is a utility macro. Some of the utility macros call routines in the supervisor, while others merely generate in-line code. Status codes, however, are never returned.

The VMS system service macros take the form $xxxx_x. The prefix, $, signifies a VMS system service. The _x suffix shows that the macro may end in any of four ways.

>   blank -   generate an argument list (this corresponds to _L in
>   the supervisor service macros)
>
>   _DEF -   generate symbols and offsets
>
>   _S -   call the service with CALLS
>
>   _G -   call the service with CALLG

Most of the supervisor services and the VMS system services return status codes in RØ. The program should check the contents of RØ following one of these macros calls, in order to determine whether the service was completed successfully.

Arguments enclosed in square brackets are optional.

For detailed background information on the macros and for explanations of how to code the macros, refer to Chapters 7, 8, and 9 of this manual.

10.1 $ASSIGN_x Assign I/O Channel

$ASSIGN_x is a VMS system service macro available only to level 2 and 2R diagnostic programs. A level 2 or 2R program must assign a channel to a peripheral device before the program can perform any input or output operation on the device. The $ASSIGN_x macro calls a VMS service that assigns a channel and a channel number to a device. This channel provides a path between the program and the device. In addition, you can use the $ASSIGN_x macro to establish a logical link with a remote node in a network in level 2R programs.

$ASSIGN_x devnam, chan, [acmode], [mbxnam]

devnam =   the address of a character string descriptor that points to the device name string. The string may be either a physical device name or a logical name. If the first character in the string is an underscore (_), the service treats the name as a physical device name. Otherwise, the service performs a single level of logical name translation and uses the equivalence name, if there is any.

chan   =   the address of a word to receive the channel number that the service assigns to the device.

acmode =   the access mode to be associated with the channel. The service maximizes the specified access mode with the access mode of the caller. I/O operations on the channel can be performed only from equal and more privileged access modes.

Kernel mode = 0
Executive mode = 1
Supervisor mode = 2
User mode = 3

mbxnam =   the address of the character string descriptor pointing to the logical name string for the mailbox to be associated with the device, if there is a mailbox. The mailbox receives status information from the device driver.

Return Status Codes
SS$_NORMAL: Service successfully completed.

SS$_REMOTE: Service successfully completed. A logical link was established with the target on a remote node.

SS$_ACCVIO: The device or mailbox name string or string descriptor cannot be read, or the channel number cannot be written, by the caller (access violation).

SS$_DEVALLOC: Warning: the device is allocated to another process.

SS$_DEVNOTMBX: A mailbox name has been specified for a device that is not a mailbox.

SS$_EXQUOTA: The target of the assignment is on a remote node and the process has insufficient buffer quota to allocate a network control block.

SS$_INSFMEM: The target of the assignment is on a remote node, and there is insufficient system dynamic memory to complete the request.

SS$_IVDEVNAM: No device name was specified, or the device or mailbox name string contains invalid characters. Or, the Network Connect Block has an invalid format.

SS$_IVLOGNAM: The device or mailbox name string has a length of 0 or has more than 63 characters.

SS$_NOIOCHAN: No I/O channel is available for assignment.

SS$_NOPRIV: The process does not have the privilege to perform network operations.

SS$_NOSUCHDEV: The specified device or mailbox does not exist.

SS$_NOLINKS: No logical network links are available.

SS$_NOSUCHNODE: The specified network node is nonexistent or unavailable.

SS$_REJECT: The network connect was rejected by NSP (Network Services Protocol) or the partner on the remote node. Or, the target image exited before the connect confirm could be issued.

NOTES

1. Channels can be assigned to devices on remote systems. For details on how to use ASSIGN in conjunction with network operations, see the VAX-11 DECnet User's Guide.

2. Only the owner of a device can associate a mailbox with the device (the owner is the process that has allocated the device, either implicitly or explicitly). Then the device driver can send messages containing status information to the mailbox, as in the following cases:

   ● If the device is a terminal, a message may indicate dialup,

hangup, or the reception of unsolicited input.

● If the target is on a network, the message may indicate the network connect or initiate, or whether the line is down.

● If the device is a line printer, the message may indicate that the printer is off-line.

For details on the message format and the information returned, refer to the VAX/VMS I/O User's Guide.

3. Channels remain assigned until they are explicitly deassigned with the deassign I/O channel (DASSGN) system service, or until the image that assigned the channel exits.

4. The ASSIGN service establishes a path to a device, but does not check whether the caller can actually perform I/O operations to the device. Privilege and protection restrictions may be applied by the device drivers. For details on how the system controls access to devices, refer to the VAX/VMS I/O User's Guide.

10.2 $BINTIM_x Convert an ASCII String to Binary Time

$BINTIM_x is a VMS system service macro. It calls a service that converts an ASCII string to an absolute time value or a delta time value in 64-bit system format. This format is suitable for input to the Set Timer (SETTIMR) system service.

$BINTIM_x timbuf, timadr

timbuf = the address of the character string descriptor pointing to the absolute or delta time value to be converted. The ASCII string value must be in one of the formats shown in the following note.

timadr = the address of a quadword that is to receive the converted time in 64-bit format.

Return Status Codes

SS$_NORMAL: Service successfully completed.

SS$_IVTIME: The syntax of the specified ASCII string is invalid, or the time component is out of range.

NOTE
The required ASCII input strings have
the following formats: Absolute Time:
dd-mm-yyyy hh:mm:ss.cc, Delta Time:
dddd hh:mm:ss.cc.

10.3 $CANCEL_x Cancel I/O on Channel

$CANCEL_x is a VMS system service macro available only to level 2 and 2R diagnostic programs. It calls a VMS service routine that cancels all pending I/O requests on a specific channel. In general, this includes all I/O requests that are queued, as well as those that are currently in progress.

$CANCEL_x chan

chan = the number of the I/O channel assigned to the device for which I/O is to be canceled.

Return Status Codes
SS$_NORMAL: Service successfully completed.

SS$_EXQUOTA: The process has exceeded its quota for direct I/O. In this context, direct I/O refers to use of the direct data path on the Unibus channel adapter.

SS$_INSFMEM: Insufficient system dynamic memory is available to cancel the I/O, and the process has disabled resource wait mode with the Set Resource Wait Mode (SETRWM) system service.

SS$_IVCHAN: An invalid channel was specified, that is, a channel number of 0 or a number larger than the number of channels available.

SS$_NOPRIV: The specified channel is not assigned, or was assigned from a more privileged access mode.

Privilege Restrictions
I/O can be canceled only from an access mode equal to or more privileged than the access mode from which the original channel assignment was made.

NOTES
1.   When a request currently in progress
     is canceled, the driver is notified
     immediately. Actual cancellation may
     or may not occur immediately,
     depending on the logical state of
     the driver. When cancellation does
     occur, the same action as that taken
     for queued requests is performed.

     a. The specified event flag is set.

> b. The first word of the I/O status
>   block, if specified, is set to
>   SS$_CANCEL.
>
> c. The AST, if specified, is queued.
>
> Proper synchronization between this
> service and the actual canceling of
> I/O requests requires the issuing
> process to wait for I/O completion
> in the normal manner and then to
> note that the I/O has been canceled.
>
> 2. If the I/O operation is a virtual
>   I/O operation involving a disk or
>   tape ancillary control process, the
>   I/O cannot be canceled. In the case
>   of a magnetic tape, however,
>   cancellation may occur if the device
>   driver is hung.
>
> 3. Outstanding I/O requests are
>   automatically canceled at image
>   exit.

10.4 $CANTIM_x Cancel Timer Request

$CANTIM_x is a VMS system service macro. It calls a VMS service that cancels all or a selected subset of the Set Timer requests previously issued by the current image executing in a process. Cancellation is based on the request identification specified in the Set Timer (SETIMR) system service. If more than one timer request has been given the same request identification, they are all canceled.

$CANTIM_x [reqidt], [acmode]

reqidt = the request identification of the timer request(s) to be canceled. A value of 0 (the default) indicates that all timer requests are to be canceled.

acmode = the access mode of the request(s) to be canceled. The access mode is maximized with the access mode of the caller. Only those timer requests issued from an access mode equal to or less privileged than the resultant access mode are canceled.

Return Status Codes
SS$_NORMAL: Service successfully completed.

Privilege Restrictions
Timer requests can be canceled only from access modes equal to or more privileged than the access mode from which the requests were issued.

Resources Required/Returned
Canceled timer requests are restored to the process's quota for
timer queue entries.

NOTE
Outstanding timer requests are
automatically canceled at image exit.

10.5 $CLREF_x Clear Event Flag
$CLREF_x is a VMS system service macro. It calls a VMS service
that clears an event flag.

$CLREF_x efn

efn = the number of the event flag to be cleared.

Return Status Codes
SS$_WASCLR: Service successfully completed. The specified event
flag was previously 0.

SS$_WASSET: Service successfully completed. The specified event
flag was previously 1.

SS$_ILLEFC: An illegal event flag number was specified.

SS$_UNASEFC: The process is not associated with the cluster
containing the specified event flag.

10.6 $DASSGN_x Deassign I/O Channel

$DASSGN_x is a VMS system service macro available only to level 2 and 2R diagnostic programs. It calls a VMS service that releases an I/O channel acquired for I/O with the Assign Channel system service.

$DASSGN_x chan

chan = the number of the I/O channel to be deassigned.

Return Status Codes

SS$_NORMAL: Service successfully completed.

SS$_IVCHAN: An invalid channel number was specified; that is, a channel number of 0 or a number larger than the number of channels available.

SS$_NOPRIV: The specified channel is not assigned, or was assigned from a more privileged access mode.

Privilege Restrictions

An I/O channel can be deassigned only from an access mode equal to or more privileged than the access mode from which the original channel assignment was made.

<div align="center">NOTES</div>

1. When a channel is deassigned, any outstanding I/O requests on the channel are canceled. If a file has been opened on the specified channel, the file is closed.

2. If a mailbox was associated with the device when the channel was assigned, and there are no more channels assigned to the mailbox, the linkage to the mailbox is cleared.

3. If the I/O channel was assigned for a network operation, the network link is disconnected. For more information on channel assignment and deassignment for network operations, refer to the **VAX-11 DECnet User's Guide.**

4. If the specified channel is the last channel assigned to a device that has been marked for dismounting, the device is dismounted.

5. I/O channels are automatically deassigned at image exit.

10.7 $DEF Define Some Field Within a Data Structure
$DEF sym, alloc, siz

sym   =   the name of the symbol to be defined.

alloc   =   an assembler directive indicating the block size to
             be used:
             .BLKB
             .BLKW
             .BLKL

siz   =   the number of blocks to be allocated.

Return Status Codes: Not Applicable.

Use this structure definition utility macro to generate an offset
for the symbols given in a P-table data structure.

10.8 $DEFEND Finish Definitions
$DEFEND struc, gbl
struc =   the structure name (e.g., the name of the P-table).

gbl   =   GLOBAL or LOCAL

Return Status Codes: Not Applicable.

Use this structure definition utility macro to clean up the
P-table data stucture definition process after all of the symbols
for the structure have been defined.

10.9 $DEFINI Start a Data Structure
$DEFINI struc, gbl, dot

struc =   the structure name (e.g., the name of the P-table).

gbl   =   GLOBAL or LOCAL.

dot   =   the value of the first symbol to be generated.

Return Status Codes: Not Applicable.
Use this structure definition utility macro to start the
definition of a P-table data structure.

10.10 $DS_ABORT Abort Program or Test
$DS_ABORT is a utility macro. The function of the macro depends on
the argument supplied by the programmer. If the argument is
PROGRAM, the macro calls a routine in the supervisor that prints
an abort message on the operator's terminal, executes the
program's cleanup code, and then transfers control to the BEGIN
routine in the supervisor.

If the argument is TEST, the macro merely clears R0 and then does
an RET, to call the dispatcher and start the next test.

$DS_ABORT [arg]

arg   = the level of abort (PROGRAM or TEST).

PROGRAM   = Execute clean up code and return to command mode.

TEST   = Terminate current test and proceed to the next test.

NOTE
Default arg = PROGRAM.

Return Status Codes: Not Applicable.

10.11 $DS_ASKADR_x Ask Operator for an Address
$DS_ASKADR_x is a supervisor service macro. It calls a supervisor routine that displays a prompt message on the operator's terminal, asking the operator for an address. The service accepts an ASCII numeric address string, checks whether or not it is within an acceptable range, and stores the address in a user-specified longword.

$DS_ASKADR_x msgadr, datadr, [radix], [lolim], [hilim], [defalt], [unused], [exword]

msgadr =   the address of a counted ASCII string used as a prompt.

datadr =   the address of a longword that will receive the response.

radix =   PAR$_BIN, PAR$_OCT, PAR$_DEC, or the default radix, PAR$_HEX.

lolim =   the minimum acceptable numeric response. Default value = 0.

hilim =   the maximum acceptable numeric response. Default value = +maximum, $+(2^{31} - 1)$.

defalt =   the value to use if the operator gives a null response. Default value = 0.

unused =   reserved for expansion.

exword =   the exception mask. PAR$_NODEF means that there is no default. PAR$_ATDEF, PAR$_ALTO, and PAR$_ATHI cause the parameters DEFALT, LOLIM, and HILIM to be interpreted as containing the addresses where the corresponding values may be found, instead of containing their literal values.

Return Status Codes
DS$_NORMAL: Service successfully completed.

DS$_PROGERR: The number of arguments supplied is incorrect.

NOTES
1. Execution of this macro will cause a program abort if the Operator flag is clear.

2. If the Prompt flag is set, ranges and default values will be printed with the prompt message.

3. The radix and exception mask arguments are defined by $DS_PARDEF.

4. Do not use FAO directives in the prompt string.

10.12 $DS_ASKDATA_x Ask the Operator for a Numeric Value
$DS_ASKDATA_x is a supervisor service macro. It calls a supervisor service routine that prompts the operator for a numeric value and ensures that the value is within an acceptable range. The ASCII numeric string that the operator returns is converted, truncated if necessary, and stored in a caller-specified variable field (mask).

$DS_ASKDATA_x msgadr, datadr, [radix], [mask], [lolim], [hilim], [defalt], [unused], [exword]

msgadr = the address of a counted ASCII string used as a prompt.

datadr = the address of a longword that will receive the response.

radix = PAR$_BIN, PAR$_OCT, PAR$_DEC, or the default radix, PAR$_HEX.

mask = the bit mask indicating field position and size.

lolim = the minimum acceptable numeric response. Default value = $-$maximum, $-(2^{31})$.

hilim = the maximum acceptable numeric response. Default value = $+$maximum, $+(2^{31}-1)$.

defalt = the value to use if the operator gives a null response. Default value = 0.

unused = reserved for expansion.

exword = the exception mask. PAR$_NODEF means that there is no
default. PAR$_ATDEF, PAR$_ALTO, and PAR$_ATHI cause the
parameters DEFALT, LOLIM, and HILIM to be interpreted as
containing the addresses where the corresponding values
may be found, instead of containing their literal
values.

Return Status Codes
DS$_NORMAL: Service successfully completed.

DS$_PROGERR: The value supplied by the operator is invalid.

DS$_TRUNCATE: The value supplied by the operator is too large to
fit in the specified buffer.

NOTES

1. Execution of this macro will cause a
program abort if the Operator flag
is clear.

2. If the Prompt flag is set, ranges
and default values will be printed
with the prompt message.

3. Truncation of left-most bits occurs
if a response is larger than the
mask.

4. The radix and exception mask
arguments are defined by $DS_PARDEF.

5. General register R1 contains the
converted binary value, not
truncated, on return.

6. Do not use FAO directives in the
prompt string.

10.13 $DS_ASKLGCL_x Ask the Operator for a Logical Response

$DS_ASKLGCL_x is a supervisor service macro. It calls a supervisor
service routine that prompts the operator for a logical response
to a specified question. The service accepts an ASCII "yes" or
"no" string, converts this to a single bit (flag), and stores the
information in a caller-specified bit within a caller-specified
byte.

$DS_ASKLGCL_x msgadr, datadr, [pos], [yexfer], [noxfer], [defalt]

msgadr = the address of a counted ASCII string used as a prompt.

datadr = the address of a byte that will receive the response.

pos   =   the bit position within DATADR. Range = 0 through 7. Default = 0.

yexfer = the branch address for positive response. Default = 0, meaning no branch.

noxfer = the branch address for negative response. Default = 0, meaning no branch.

defalt = PAR$_YES or PAR$_NO.

Return Status Codes

DS$_NORMAL: Service successfully completed.

DS$_PROGERR: The bit position argument, POS, specified by the caller, is too large a number, or the number of arguments supplied is incorrect.

NOTES

1. Execution of this macro will cause a program abort if the Operator flag is clear.

2. If the Prompt flag is set, ranges and default values will be printed with the prompt message.

3. Do not use FAO directives in the prompt string.

10.14 $DS_ASKSTR_x Ask the Operator for a String

$DS_ASKSTR_x is a supervisor service macro. It calls a supervisor service routine that prompts the operator for a string response. It accepts an ASCII string, which is checked against a list of strings. If valid, the string is placed in a caller-specified buffer.

$DS_ASKSTR_x msgadr, bufadr, [maxlen], [valtab], [defadr]

msgadr = the address of a counted ASCII string used as a prompt.

bufadr = the address of a counted ASCII buffer.

maxlen = the maximum length of response string (does not include the count byte). Default = 72.

valtab = the address of a counted list of string pointers. Default = 0, meaning that there is no validation table.

<u>defadr</u> =   the address of a counted ASCII string to be used if the
operator gives a null response. Default = Ø, meaning
that there is no default string.

Return Status Codes
DS$_NORMAL: Service successfully completed.

DS$_PROGERR: The number of arguments supplied is incorrect.

DS$_TRUNCATE: The string supplied by the operator has been
truncated because it will not fit into the buffer supplied by the
program.

<div align="center">NOTES</div>

1.   Execution of this macro will cause a
     program abort if the Operator flag
     is clear.

2.   If the Prompt flag is set, ranges
     and default values will be printed
     with the prompt message.

3.   If VALTAB = Ø, any string will be
     accepted without qualification.

4.   If VALTAB not = Ø, R1 will return
     with an index value into the
     validation table. Warning: If the
     program uses default and the
     operator selects <CR> only (a null
     response), R1 will contain Ø.

5.   Do not use FAO directives in the
     prompt string.

10.15 $DS_ASKVLD_x Ask the Operator for a Numeric Value
$DS_ASKVLD_x is a supervisor service macro. It calls a supervisor
service routine that prompts the operator for a numeric value. The
routine accepts an ASCII numeric string as input, converts the
string, truncates the string if necessary, checks to determine
whether the value is within an acceptable range, and stores the
value in a caller-specified variable field (position and size).

$DS_ASKVLD_x msgadr, datadr, [radix], [pos], [size], [lolim],
[hilim], [defalt], [unused], [exword]

<u>msgadr</u> =   the address of a counted ASCII string used as a prompt.

<u>datadr</u> =   the address of a longword that will receive the
response.

<u>radix</u> =   PAR$_BIN, PAR$_OCT, PAR$_HEX, or the default radix,
PAR$_DEC.

<div align="center">

</div>

<u>pos</u>   = the right-most bit of the field, range = 0 through 31. Default = 0.

<u>size</u> = the number of bits in the field.

<u>lolim</u> = the minimum acceptable numeric response. Default value = -maximum, $-(2^{31})$.

<u>hilim</u> = the maximum acceptable numeric response. Default value = +maximum, $+(2^{31}-1)$.

<u>defalt</u> = the value to use if the operator gives a null response.

<u>unused</u> = reserved for expansion.

<u>exword</u> = the exception mask. PAR$_NODEF means that there is no default. PAR$_ATDEF, PAR$_ALTO, and PAR$_ATHI cause the parameters DEFALT, LOLIM, and HILIM to be interpreted as containing the addresses where the corresponding values may be found, instead of containing their literal values.

Return Status Codes
DS$_NORMAL: Service successfully completed.

DS$_PROGERR: An invalid value was supplied by the operator.

DS$_TRUNCATE: The string supplied by the operator has been truncated because it will not fit into the buffer supplied by the program.

<div align="center">NOTES</div>

1. Execution of this macro will cause a program abort if the Operator flag is clear.

2. If the Prompt flag is set, ranges and default values will be printed with the prompt message.

3. The radix and exception mask arguments are defined by $DS_PARDEF.

4. Do not use FAO directives in the prompt string.

10.16 $DS_BCOMPLETE Branch on Complete
$DS_BCOMPLETE is a utility macro. It generates code that tests R0. The program will branch to the address specified by LABEL, if the low bit of R0 is set, indicating successful completion of the previous supervisor call.

$DS_BCOMPLETE label

label = the address for transfer of program control.

Return Status Codes: Not Applicable.

10.17 $DS_BERROR Branch on Error
$DS_BERROR is a utility macro. It generates code that causes a branch to the address specified by LABEL, if the low bit of R0 is clear, indicating an error condition in a preceding supervisor call.

$DS_BERROR label

label = the address for transfer of program control.

Return Status Codes: Not Applicable.

10.18 $DS_BGNCLEAN Begin Cleanup Code Section
$DS_BGNCLEAN is a utility macro. This macro should be the first statement in the cleanup code section of a diagnostic program. It does not call a supervisor routine. It simply provides an entry point to the cleanup code section.

$DS_BGNCLEAN [regmask], [psect]

regmask = the entry point register save mask.

psect   = any legal arguments for the .PSECT directive.

Return Status Codes: Not Applicable.

NOTE
Default PSECT arguments = <CLEANUP, LONG>.

10.19 $DS_BGNDATA Begin Data Section
$DS_BGNDATA is a utility macro. Use this macro at the beginning of a test data section to provide a label. The macro does not call a supervisor routine.

$DS_BGNDATA [align]

align = the psect alignment.

Return Status Codes: Not Applicable.

NOTE
The default ALIGN argument is PAGE.

10.20 $DS_BGNINIT Begin Initialize Code Section
$DS_BGNINIT is a utility macro. Use this macro at the beginning of the initialization routine in the diagnostic program to provide an entry point to the routine. The macro does not call a supervisor routine.

$DS_BGNINIT [regmask], [psect]

regmask = the entry point register save mask.

psect   = any legal arguments for the .PSECT directive.

Return Status Codes: Not Applicable.

> **NOTE**
> Default PSECT arguments = <INITIALIZE,
> LONG>.

10.21 $DS_BGNMESSAGE Begin a Message
$DS_BGNMESSAGE is a utility macro. Use this macro at the beginning of a global error message print routine. The macro creates an entry mask. It does not call a supervisor routine.

$DS_BGNMESSAGE [regmask]

regmask = the entry point register save mask.

Return Status Codes: Not Applicable.

10.22 $DS_BGNMOD Begin a Program Module
$DS_BGNMOD is a utility macro. Use this macro at the beginning of each source module in a diagnostic program. The macro defines the beginning of a source module. It does not call a supervisor routine.

$DS_BGNMOD env, [test], [subtest]

env   = CEP_FUNCTIONAL, CEP_REPAIR, SEP_FUNCTIONAL or
           SEP_REPAIR.

test   = the number of the first test in this module.

subtest = the number of the first subtest in this module.

Return Status Codes: Not Applicable.

> **NOTES**
> 1. $DS_BGNMOD is absolutely required in
>   every module.
>
> 2. Default TEST = 1.   Default SUBTEST
>   argument = 1.

10.23 $DS_BGNREG Begin a Device Register Storage Area
$DS_BGNREG is a utility macro. You should use the macro to define
an area in memory where information from device registers may be
stored and kept up to date. The macro provides a label, DEV_REG,
which identifies the first location in the area. It also defines
this label for the $DS_HEADER macro. $DS_BGNREG does not call a
supervisor routine.

$DS_BGNREG (no arguments)

Return Status Codes: Not Applicable.

10.24 $DS_BGNSERV Begin Interrupt Service Routine
$DS_BGNSERV is a utility macro. Use this macro to define an entry
point for an interrupt service routine. The macro also ensures
longword alignment, and pushes registers R0 and R1 on the stack.
It does not call a supervisor routine.

$DS_BGNSERV label

label = an identifying label for this routine.

Return Status Codes: Not Applicable.

10.25 $DS_BGNSTAT Begin Statistics Section
$DS_BGNSTAT is a utility macro. Use this macro to define an area
in memory for each logical unit (LUN) being tested. The hardware
and software errors may be tabulated in this area. The macro does
not call a supervisor routine.

$DS_BGNSTAT (no arguments)

Return Status Codes: Not Applicable.

10.26 $DS_BGNSUB Begin a Subtest
$DS_BGNSUB is a utility macro. It calls a supervisor routine that
marks the beginning of a subtest. The routine ensures that the
diagnostic program is sequencing through its subtests in numerical
order. If the routine detects a sequencing error, it notifies the
operator and then returns control to the command mode.

$DS_BGNSUB (no arguments)

Return Status Codes: Not Applicable.

10.27 $DS_BGNSUMMARY Begin Summary Report Section
$DS_BGNSUMMARY is a utility macro. Use this macro at the beginning
of the summary code section of a program to produce an entry mask
and a label (SUMMARY). The summary report routine can be called to
print a report based on information in the data and statistics
areas of the program. This macro does not call a supervisor
routine.

$DS_BGNSUMMARY [regmask], [psect]

regmask =   the entry point register save mask.

psect   =   any legal arguments for the .PSECT directive.

Return Status Codes: Not Applicable.

NOTE
Default PSECT arguments = <SUMMARY, LONG>.

10.28 $DS_BGNTEST Begin a Test

$DS_BGNTEST is a utility macro. Use this macro at the beginning of a test in a diagnostic program. It produces an entry mask and notifies the supervisor of the beginning of a test routine. The macro does not call a supervisor routine.

$DS_BGNTEST [section], [regmask], [align]

section =   the test section name(s). This is the name of the program section that includes the test. If a test belongs to two or more sections, place the section names in single brackets, < >, and separate the section names with commas.

regmask =   the entry point register save mask.

align   =   the boundary alignment.

Return Status Codes: Not Applicable.

NOTES
1. The default SECTION argument is 0.

2. The default ALIGN argument is PAGE.

3. This macro also generates an entry in the dispatch table.

10.29 $DS_BITDEF Define Bit Value Mnemonics

$DS_BITDEF is a utility macro. It defines the symbols BIT0 through BIT31 and the masks corresponding to each bit.

$DS_BITDEF [gbl]

gbl = GLOBAL or LOCAL.

Return Status Codes: Not Applicable.

10.30 $DS_BNCOMPLETE Branch on not Complete
$DS_BNCOMPLETE is a utility macro. It generates code to check the low-order bit of R0, in order to test for a failure in a previous call to the supervisor. If bit 0 is clear, a branch occurs. The macro does not call a supervisor routine.

$DS_BNCOMPLETE label

label = the address for transfer of program control.

Return Status Codes: Not Applicable.

10.31 $DS_BNERROR Branch on not in Error
$DS_BNERROR is a utility macro. It generates code to check the low-order bit of R0, in order to test for the success of a previous call to the supervisor. If bit 0 is set, a branch occurs. The macro does not call a supervisor routine.

$DS_BNERROR label

label = the address for transfer of program control.

Return Status Codes: Not Applicable.

10.32 $DS_BNOPER Branch if no Operator Present
$DS_BNOPER is a utility macro. It generates code that checks the status of the Operator control flag. If no operator is present, the macro causes a branch to the address specified by the programmer. No supervisor routine is called by this macro.

$DS_BNOPER label

label = the address for transfer of program control.

Return Status Codes: Not Applicable.

10.33 $DS_BNPASS0 Branch if not in Pass Zero
$DS_BNPASS0 is a utility macro. It generates code that tests the status of a one-time switch used for program initialization in the initialization section. If pass 0 has already been completed, the macro causes a branch to the address specified by the programmer. No supervisor routine is called by the macro.

$DS_BNPASS0 label

label = the address for transfer of program control.

Return Status Codes: Not Applicable.

NOTE
This macro is valid only in the initialization section.

10.34 $DS_BNQUICK Branch if not in Quick Mode
$DS_BNQUICK is a utility macro. The macro generates code that checks the status of the Quick control flag. If the flag is not set, control passes to the address specified by the programmer. No supervisor routine is called by the macro.

$DS_BNQUICK label

label = the address for transfer of program control.

Return Status Codes: Not Applicable.

10.35 $DS_BOPER Branch if Operator Present
$DS_BOPER is a utility macro. It generates code that checks the status of the Operator control flag. If the flag is set, control passes to the address specified by the operator. This macro does not call a supervisor routine.

$DS_BOPER label

label = the address for transfer of program control.

Return Status Codes: Not Applicable.

10.36 $DS_BPASS0 Branch if in Pass Zero
$DS_BPASS0 is a utility macro. It generates code that checks the status of a one-time switch used for program initialization in the initialization section. If pass 0 has not been completed, the macro causes a branch to the address specified by the programmer. No supervisor routine is called by the macro.

$DS_BPASS0 label

label = the address for transfer of program control.

Return Status Codes: Not Applicable.

NOTE
This macro is valid only in the initialization section.

10.37 $DS_BQUICK Branch if in Quick Mode
$DS_BQUICK is a utility macro. The macro generates code that checks the status of the Quick control flag. If the flag is set, the macro causes a branch to the address specified by the programmer. No supervisor routine is called by the macro.

$DS_BQUICK label

label = the address for transfer of program control.

Return Status Codes: Not Applicable.

10.38 $DS_BREAK Break in Diagnostic Program
$DS_BREAK is a utility macro. It calls the DS$BREAK routine in the supervisor. The DS$BREAK routine, in turn, checks the status of the Control C flag. If the operator has typed Control C, setting the flag, the routine inserts a breakpoint and passes control to the command mode in the CLI.

$DS_BREAK is used primarily as a no-op call for tight, high-priority loops.

$DS_BREAK (no arguments)

Return Status Codes: Not Applicable.

10.39 $DS_CANWAIT_x Cancel Wait
$DS_CANWAIT_x is a supervisor service macro. It calls a routine in the supervisor that cancels the effect of a previous $DS_WAITMS_x or $DS_WAITUS_x macro by invoking the $WAKE_x system service macro.

$DS_CANWAIT_x (no arguments)

Return Status Code
DS$_NORMAL: Service successfully completed.

NOTE
In general, you should use this macro in
interrupt service routines.

10.40 $DS_CFDEF Call Frame Definitions
$DS_CFDEF is a utility macro. It provides symbolic definitions for call frame offsets from the frame pointer. It does not call a routine in the supervisor.

$DS_CFDEF [gbl]

gbl = GLOBAL or LOCAL.

Return Status Codes: Not Applicable.

NOTE
The defined symbols are:

CF$L_ONCOND	Condition Handler
CF$W_PSW	Processor Status Word
CF$W_MASK	Register Mask
CF$L_AP	Old Argument Pointer
CF$L_FP	Old Frame Pointer
CF$L_PC	Return PC
CF$L_REG	Saved Registers

10.41 $DS_CHANNEL_x Channel Service
$DS_CHANNEL_x is a supervisor service macro available to level 3 diagnostic programs only. It calls a supervisor routine that

provides a channel adapter interface service. The service enables a diagnostic program to exercise general control over the hardware status of the channel adapter.

$DS_CHANNEL_x unit, func, [vecadr], [stsadr]

unit   =   the logical unit number.

func   =   the function code specifying the operation to be performed. The code is expressed symbolically. The function argument should be preceded by a number sign (#).

vecadr =   the entry point address for interrupt service when an interrupt occurs. Required for the CHC$_ENINT function.

stsadr =   the address of a quadword to store adapter status for the CHC$_STATUS function or the CHC$_ENINT function. This argument is required for the CHC$_STATUS and CHC$_ENINT functions.

$DS_CHANNEL Functions

CHC$_INITA   Initialize Channel Adapter
    Return Status Code:
    DS$_NORMAL: Service successfully completed.

CHC$_INITB   Initialize device bus (UBA only)
    Return Status Code:
    DS$_NORMAL: Service successfully completed.

CHC$_ABORT   Adapter abort (MBA only)
    Return Status Codes:
    DS$_NORMAL: Service successfully completed.

    DS$_LOGIC: Bit set/clear failure. DTABT did not set (MBA). ABORT did not clear (MBA).

CHC$_PURGE   Purge data path (UBA only)
    Return Status Code:
    DS$_NORMAL: Service successfully completed. This function purges the data path specified by the last $DS_SETMAP_X macro call.

CHC$_ENINT   Enable interrupts
    Return Status Codes:
    DS$_NORMAL: Service successfully completed.

    DS$_IHWE: Initial hardware error. Adapter error condition was found before the function was performed.

    DS$_LOGIC: Bit set/clear failure. Enable bit failed to set.

    DS$_IVVECT: Invalid vector was found by $DS_SETVEC_x.

    DS$_IVADDR: Invalid address was found by $DS_SETVEC_x.

CHC$_DSINT   Disable interrupts
    Return Status Codes:
    DS$_NORMAL: Service successfully completed.

    DS$_IHWE: Initial hardware error. Adapter error condition was found before the function was performed.

    DS$_LOGIC: Bit set/clear failure. Interrupt enable bit failed to clear.

    DS$_IVVECT: Invalid vector was found by $DS_CLRVEC_x.

CHC$_CLEAR   Clear adapter status
    Return Status Codes:
    DS$_NORMAL: Service successfully completed.

    DS$_LOGIC: Bit set/clear failure. One of the status bits failed to clear.

CHC$_STATUS   Request adapter status
    Return Status Code:
    DS$_NORMAL: Service successfully completed.

CHC$_SETDFT   Set defeat parity (UBA only)
    Return Status Code:
    DS$_NORMAL: Service successfully completed.

CHC$_CLRDFT   Clear defeat parity (UBA only)
    Return Status Code:
    DS$_NORMAL: Service successfully completed.

NOTES

1. The interrupt enable function (CHC$_ENINT) will enable adapter interrupts and, in the case of the UBA, device interrupts. After Unibus initialization or UBA initialization, perform a clear function before enabling interrupts.

2. Adapter status, returned in response to the CHC$_STATUS function, is stored in location specified by the argument STSADR, as shown in Note 3.

3. Returned status description: Two longwords of status are returned for each request status function (CHC$_STATUS) of the CHANNEL service call. This status, along with certain specific interrupt information, is also supplied on all interrupts that are to be passed to

a program which has enabled interrupt processing. The two longwords returned have the following format.

status 1	
rvr	status 2

STATUS 1 = adapter status as defined in Note 4.

STATUS 2 = one word of interrupt status as defined in Note 5.

RVR   = receive vector register for those devices that interrupt through the UBA.

4. Adapter Status Definitions (STATUS 1)

Symbol	Definition
CHS$M_SYSERR	System error (category)
CHS$M_CHNERR	Channel error (category)
CHS$M_DEVERR	Device error (category)
CHS$M_PGMERR	Program error (category)
CHS$M_PGMHDE	Program error (hardware detected)
CHS$M_DEVBUS	Unibus/Massbus error
CHS$M_DEVTO	Device timeout
CHS$M_CHNDPE	Channel data parity error
CHS$M_CHNMPE	Channel memory parity error
CHS$M_CHPFOT	Channel power fail/overtemp
CHS$M_SYSMEM	System memory error
CHS$M_SYSSBI	System SBI error
CHS$M_MBAEXC	Massbus exception
CHS$M_MBANED	MBA nonexistent drive
CHS$M_MBADTB	MBA data transfer busy
CHS$M_MBADTC	MBA data transfer complete
CHS$M_MBACPE	MBA control parity error
CHS$M_MBAWCK	MBA write check
CHS$M_BUSIC	Bus init clear (deassertion)
CHS$M_BUSINIT	Bus init (assertion)
CHS$M_BUSPDN	Bus power down
CHS$M_ERRANY	Any error category (CHSM_SYSERR, CHSM_CHNERR, CHSM_DEVERR, CHSM_PGMERR)

5. Interrupt Status Definitions (STATUS 2)

CHI$M_CHNINT	Channel interrupt
CHI$M_DEVINT	Device interrupt
CHI$M_IPL	Interrupt priority level

6. When CHC$_ENINT is used, $DS_CHIDEF should be used to define the status codes returned at STSADR.

A complete list of return status codes for $DS_CHANNEL_x follows.

DS$_NORMAL: Service successfully completed.

DS$_PROGERR: Program error encountered.

DS$_IHWE: Initial hardware error encountered. The adapter hardware status was found to be in error before the performance of the requested function. The function will not be performed.

DS$_FHWE: Final hardware error status encountered. The adapter hardware status was found to be in error after the performance of the requested function. The capability of the adapter to operate correctly is in question.

DS$_LOGIC: An adapter function that sets or clears an adapter status bit has failed. The capability of the adapter to operate correctly is in question.

DS$_IVVECT: An invalid vector has been given as an argument.

DS$_ERROR: An error has been found trying to associate a hardware P-table with the logical unit argument.

10.42 $DS_CHCDEF Channel Function Definitions
$DS_CHCDEF is a utility macro available to level 3 diagnostic programs only. It provides symbolic definitions for the functions used in the channel call macro, $DS_CHANNEL_x. The macro does not call a supervisor routine.

$DS_CHCDEF [gbl]

gbl = GLOBAL or LOCAL.

Return Status Codes: Not Applicable.

10.43 $DS_CHDEF Channel Symbol Definitions
$DS_CHDEF is a utility macro available to level 3 diagnostic programs only. It invokes four other macros ($DS_CHSDEF, $DS_CHIDEF, $DS_CHMDEF, and $DS_CHSDEF) to define all of the channel symbols used by the channel service calls. It does not call a supervisor routine.

$DS_CHDEF [gbl]

gbl = GLOBAL or LOCAL.

Return Status Codes: Not Applicable.

10.44 $DS_CHIDEF Interrupt Status Definitions
$DS_CHIDEF is a utility macro available to level 3 diagnostic programs only. It defines the codes that are used by the channel service, $DS_CHANNEL_x, to indicate adapter status following an

interrupt. You should use the $DS_CHIDEF macro in conjunction with the CHC$_ENINT function of the $DS_CHANNEL_x macro.

$DS_CHIDEF [gbl]

gbl = GLOBAL or LOCAL.

Return Status Codes: Not Applicable.

10.45 $DS_CHMDEF Channel Mapping Function Definitions
$DS_CHMDEF is a utility macro available to level 3 diagnostic programs only. It provides symbolic definitions for the $DS_SETMAP_x functions.

The macro does not call a supervisor routine.

$DS_CHMDEF [gbl]

gbl = GLOBAL or LOCAL.

Return Status Codes: Not Applicable.

10.46 $DS_CHSDEF Channel Adapter Status Definitions
$DS_CHSDEF is a utility macro available to level 3 diagnostic programs only. It provides symbolic definitions for the CHC$_STATUS function of the $DS_CHANNEL_x macro.

The macro does not call a supervisor routine.

$DS_CHSDEF [gbl]

gbl = GLOBAL or LOCAL.

Return Status Codes: Not Applicable.

10.47 $DS_CKLOOP Check Loop
$DS_CKLOOP is a utility macro. It calls a supervisor routine that controls the subtest looping mechanism. If an error has been detected and the Loop flag is set, the supervisor routine sets up a scope loop by causing a branch back to the label specified.

If a new error occurs, the $DS_CKLOOP macro may modify the range of the loop the next time around.

$DS_CKLOOP label

label = the address for transfer of program control.

Return Status Codes: Not Applicable.

10.48 $DS_CLI Command Line Interpreter Tree
$DS_CLI is a utility macro. The macro is used to build parsing trees. Each call generates one node in the tree. The parser goes down the tree until a mismatch or branch directive is encountered. The macro does not call a supervisor routine.

$DS_CLI char, [action], [miss], [ascii]

char   =   comparison character. See Note 1.

action =   a code to be passed to the action routine. See Note 2.

miss   =   a mismatch or branch displacement in the tree.

ascii   =   an ASCII string to use for comparison.

Return Status Codes: Not Applicable.

NOTES
1.   Special character codes defined by $DS_CLIDEF to be optionally used as the CHAR argument:

Symbol	Function
CLI$K_ERROR	Action/Parser return (bit 0 of R0 = 0).
CLI$K_EXIT	Action/Parser return (bit 0 of R0 = 1).
CLI$K_BR	Unconditional branch within the tree using MISS.
CLI$K_BIF	Branch if. Checks bit 0 of R0. Bit 0 = 0, fall through to next node. Bit 0 = 1, branch using MISS.
CLI$K_SPACE	Traverse spaces and/or tabs and call ACTION if any were found (R8 points to next non-space CHAR).
CLI$K_NUM	Traverse numeric fields and call ACTION with numeric value in R10. This function uses the default radix. It branches using MISS if no numeric data is found.
CLI$K_ALPHA	Traverse alphabetic fields. (Uppercase A-Z).
CLI$K_ALNUM	Traverse alphanumeric fields. (Uppercase A-Z or 0-9).
CLI$K_OCT	Same as CLI$K_NUM except that the octal radix is forced.
CLI$K_DEC	Same as CLI$K_NUM except that the decimal radix is forced.
CLI$K_HEX	Same as CLI$K_NUM except that the hex radix is forced.
CLI$K_STRING	ASCII argument used for match. (Note: Only the first character of the string need match).

2. Upon entry to the action routine the registers contain:

Register	Content
R0	Action code parameter from tree.
R7	Parse tree pointer.
R8	Input string pointer.
R9	Input string count remaining.
R10	Numeric data buffer.

10.49 $DS_CLIDEF Command Line Interpreter Definitions
$DS_CLIDEF is a utility macro. It generates special character code definitions for the CHAR parameter of the $DS_CLI macro. A list of these symbols follows:

        CLI$K_ERROR
        CLI$K_EXIT
        CLI$K_BR
        CLI$K_BIT
        CLI$K_SPACE
        CLI$K_NUM
        CLI$K_ALPHA
        CLI$K_ALNUM
        CLI$K_OCT
        CLI$K_DEC
        CLI$K_HEX
        CLI$K_STRING

The macro does not call a supervisor routine.

$DS_CLIDEF [gbl]

gbl = GLOBAL or LOCAL.

Return Status Codes: Not Applicable.

10.50 $DS_CLRVEC_x Clear a System Control Block Vector
$DS_CLRVEC_x is a supervisor service macro available to level 3 diagnostic programs. The macro calls a supervisor routine that sets the system control block vector for supervisor handling. The routine loads the vector in the system control block (SCB) with the contents of the corresponding vector in the SCB_IMAGE (a page that holds the initial contents of each vector).

$DS_CLRVEC_x vector

vector = the absolute vector address.

Return Status Codes
DS$_NORMAL: Service successfully completed.

10.51 $DS_CNTRLC_x Enable Control C Interception
$DS_CNTRLC_x is a supervisor service macro. It calls a supervisor service routine that enables the diagnostic program to intercept the next Control C typed by the operator. It provides the address of a routine for the supervisor to call on Control C. The routine sets an Enable flag. This flag remains set until a Control C is typed or until the flag is canceled by a call with a routine address of zero. If a routine has been specified when the next Control C is typed, the Enable flag is cleared and the specified routine is called.

$DS_CNTRLC_x label

<u>label</u> = the address for transfer of program control.

Return Status Codes
SS$_NORMAL: Service successfully completed.

10.52 $DS_CVTREG_x Convert Register Bits to Mnemonics
$DS_CVTREG_x is a supervisor service macro. It calls a supervisor service routine that converts the contents of a register to a counted ASCII string of mnemonics for inclusion in an error message. For every bit set in the register, the corresponding mnemonic is included in the ASCII string. If several bits in the register make up a function, the corresponding mnemonic, in the names of device registers mnemonics list, should be followed by "=n^R" or "=n@", where,

n = the number of bit positions that make up the field.

R = the radix in which the function field should be printed.

$DS_CVTREG_x msb, data, mneadr, strbuf, maxlen, [V1-6]

<u>msb</u>   = the most significant bit position.

<u>data</u>   = the address of the register contents in memory.

<u>mneadr</u>   = the address of a counted ASCII string of bit mnemonics.

<u>strbuf</u>   = the address of the buffer where the counted ASCII string is returned.

<u>maxlen</u>   = the length of the buffer.

<u>V1-V6</u>   = pointers to counted lists of ASCII strings defining function values specified with a mnemonic in the form XXXXX=n@.

Return Status Codes
DS$_NORMAL: Service successfully completed.

DS$_PROGERR: This status code is returned to indicate any of the following conditions:

- The number of arguments is not equal to 11.

- The MSB is greater than 32.

- The mnemonic string is exhausted and an = sign has been encountered with nothing to the right of it.

- A negative digit was encountered in the mnemonic string.

- A bad character was encountered in the mnemonic string.

- Some character other than a comma has been used to separate mnemonics in the string.

- The ASCII format overflowed.

- The caller's buffer overflowed.

Notice that when the DS$_PROGERR code is returned, 16(AP), the output buffer address, is cleared. The zero in 16(AP) indicates that there is no output from this routine.

NOTES
1. **The first mnemonic is associated with the bit position MSB.**

2. **R1 contains the full length of the string plus the count byte. This is true even if the buffer size is too short or the string is longer than 255 bytes.**

10.53 $DS_DEFDEL Delete Macro Definitions
$DS_DEFDEL is a utility macro. It serves as a conservation mechanism for the assembly process. You should use the macro to delete the code generated by the symbol definition macros after the symbols have been defined. The macro does not call a supervisor routine.

$DS_DEFDEL (no arguments)

Return Status Codes: Not Applicable.

10.54 $DS_DEVTYP Device Types
$DS_DEVTYP is a utility macro. It generates a string of device type names and a string of P-table descriptors known to the program. The device type names, listed in the note, are the type names recognized in the ATTACH command (refer to Chapter 5, Paragragh 5.3.1.). No supervisor routine is called by this macro.

$DS_DEVTYP name, name,...

name = generic device type.

Return Status Codes: Not Applicable.

NOTE

Device Type Names

KA780	RK06
MS780	TM03
RH780	TE16
DW780	TU45
RP07	TU77
RP06	DZ11
RP05	DMC11
RP04	LP11
RM03	CR11
RK611	DR11B
RK07	PCL11

Return Status Codes: Not Applicable.

10.55 $DS_DISPATCH Initialize Dispatch Table
$DS_DISPATCH is a utility macro. It generates the psect directive, beginning tag, address label, and ending tag for the dispatch table. The actual entries in the dispatch table are generated (at link time) by use of the $DS_BGNTEST macro at the beginning of each test. No supervisor routine is called by this macro.

$DS_DISPATCH (no arguments)

Return Status Codes: Not Applicable.

10.56 $DS_DSADEF Define Supervisor, APT Command, and Mailbox Areas
$DS_DSADEF is a utility macro. It provides symbols that define CLI flags, command areas, and APT mailbox areas. The macro does not call a supervisor routine. DSAAL_APTMAIL is the base address of the APT mailbox.

$DS_DSADEF [gbl]

gbl = GLOBAL or LOCAL.

Symbol	Description
DSA$GL_FLAGS	Longword containing the following flag bits
DSA$M_HALTD	Halt on error detection
DSA$M_HALTI	Halt on error isolation
DSA$M_LOOP	Loop on error flag
DSA$M_BELL	Bell on error

Symbol	Description
DSA$M_IE1	Inhibit all error reports
DSA$M_IE2	Inhibit basic error reports
DSA$M_IE3	Inhibit extended error reports
DSA$M_IES	Inhibit summary reports
DSA$M_QUICK	Quick verify
DSA$M_TRACE	Trace tests
DSA$M_LOCK	Lock in physical memory
DSA$M_OPER	Operator present
DSA$M_PROMPT	Display long dialogue
DSA$M_NORPT	Suppress all output to the terminal
DSA$M_USER	User environment
DSA$M_PASS0	Pass 0 flag
DSA$M_APT	APT mode

Command Area Definitions

Symbol	Description
DSA$GL_APTCOM	APT command
DSA$GL_PASSES	Passes to run
DSA$GL_UNITS	Units to be tested
DSA$GL_SECTNO	Section number
DSA$GL_CPUTYP	VAX CPU type code

APT Mailbox Area Definitions

Symbol	Description
DSA$GL_MSGTYP	Message type
DSA$GL_ERRNO	Error number
DSA$GL_EVENT	Event counter
DSA$GL_SUBTNO	Subtest number
DSA$GL_TESTNO	Test number
DSA$GL_PASSNO	Pass number
DSA$GL_DEVLEN	Device descriptor length
DSA$GL_DEVNAM	Device descriptor
DSA$GL_MSGPTR	Message descriptor

Return Status Codes: Not Applicable.

10.57 $DS_DSDEF Define Supervisor Status and Condition Codes
$DS_DSDEF is a utility macro. It generates symbols that denote error conditions. Many supervisor service routines return one of the codes represented by these symbols in R0 to indicate the return status. The macro does not call a supervisor routine. The following symbols are generated:

Symbol	Description
DS$_WARNING	warning
DS$_NORMAL	normal
DS$_ERROR	error condition
DS$_SEVERE	severe error condition
DS$_OVERFLOW	overflow

Symbol	Description
DS$_NULLSTR	null string
DS$_PROGERR	program error
DS$_TRUNCATE	data truncation
DS$_NOTDON	not done
DS$_IVVECT	invalid vector
DS$_IVADDR	invalid address
DS$_VASFUL	virtual address space full
DS$_INSFMEM	insufficient memory
DS$_MMOFF	memory management off
DS$_IHWE	initial hardware error
DS$_FHWE	final hardware error
DS$_LOGIC	interface error
DS$_ILLPAGCNT	illegal page count
DS$_FRAGBUF	buffer was fragmented when released
DS$_MCHK	machine check
DS$_KRNLSTK	kernel stack not valid
DS$_POWER	power fail
DS$_TRANSL	translation not valid
DS$_CHME	change mode error
DS$_NOTIMP	not implemented
DS$_IPL2HI	IPL is too high
DS$_ICERR	interval clock error
DS$_ICBUSY	interval clock busy
DS$_ARITH	arithmetic trap

Return Status Codes: Not Applicable.

10.58 $DS_DSSDEF Define Supervisor Service Entry Points

$DS_DSSDEF is a utility macro. It generates symbols that define the supervisor service entry points for the diagnostic program. When a supervisor service macro is expanded, it generates a call to one of these entry points in the supervisor entry module. The $DS_DSSDEF macro generates the following symbols:

DS$ENDPASS	DS$GPHARD
DS$ABORT	DS$SUMMARY
DS$BGNSUB	DS$ENDSUB
DS$CKLOOP	DS$INLOOP
DS$ESCAPE	DS$BREAK
DS$WAITMS	DS$WAITUS
DS$CANWAIT	DS$CNTRLC
DS$ASKDATA	DS$ASKVLD
DS$ASKADR	DS$ASKLGCL
DS$ASKSTR	DS$CVTREG
DS$PARSE	DS$ERRSYS
DS$ERRDEV	DS$ERRHARD
DS$ERRSOFT	DS$PRINTB
DS$PRINTX	DS$PRINTF
DS$PRINTS	DS$ELOGON
DS$ELOGOFF	DS$GETBUF
DS$RELBUF	DS$GETMEM
DS$RELMEM	DS$MOVVRT

DS$MOVPHY	DS$MMON
DS$MMOFF	DS$SETVEC
DS$CLRVEC	DS$INITSCB
DS$SETIPL	DS$CHANNEL
DS$SETMAP	DS$SHOCHAN
SYS$QIOW	SYS$ALLOC
SYS$ASSIGN	SYS$BINTIM
SYS$CANCEL	SYS$CANTIM
SYS$CLREF	SYS$DALLOC
SYS$DASSGN	SYS$GETTIM
SYS$QIO	SYS$READEF
SYS$SETEF	SYS$SETIMR
SYS$SETPRT	SYS$WAITFR
SYS$WFLAND	SYS$WFLOR
SYS$GETCHN	

The macro does not call a supervisor service routine.

$DS_DSSDEF [gbl]

gbl = GLOBAL or LOCAL.

Return Status Codes: Not Applicable.

10.59 $DS_ENDCLEAN End of Cleanup Code Section
$DS_ENDCLEAN is a utility macro. Use the macro at the end of the cleanup code section of a diagnostic program. It produces an exit label, checks for Control C, and then produces a return instruction.

$DS_ENDCLEAN (no arguments)

Return Status Codes: Not Applicable.

10.60 $DS_ENDDATA End of Data Section
$DS_ENDDATA is a utility macro. Use this macro following the last item in the data section of a test routine. It provides a long word containing zeros at the end of the data section. It does not call a supervisor routine.

$DS_ENDDATA (no arguments)

Return Status Codes: Not Applicable.

10.61 $DS_ENDINIT End of Initialize Code Section
$DS_ENDINIT is a utility macro. Use this macro to end the initialization section of a diagnostic program. It produces an exit label, checks for Control C, and then produces a return instruction.

$DS_ENDINIT (no arguments)

Return Status Codes: Not Applicable.

10.62 $DS_ENDMESSAGE End of a Global Error Report Sequence
$DS_ENDMESSAGE is a utility macro. Use this macro at the end of a global error message print routine. It produces a return instruction. No supervisor routine is called by the macro.

$DS_ENDMESSAGE (no arguments)

Return Status Codes: Not Applicable.

10.63 $DS_ENDMOD End of a Program Module
$DS_ENDMOD is a utility macro. Use this macro to terminate each program module. The macro generates directives to the assembler. No supervisor routine is called.

$DS_ENDMOD (no arguments)

Return Status Codes: Not Applicable.

10.64 $DS_ENDPASS_x Indicate to the Supervisor that a Logical Pass is Completed
$DS_ENDPASS_x is a supervisor service macro. Use this macro in the initialization section of the program. The macro calls a service routine in the supervisor to mark the end of a pass of the diagnostic program. If the number of passes selected by the operator have been run, the routine causes execution of the user summary and cleanup codes.

$DS_ENDPASS_x (no arguments)

Return Status Codes: None.

10.65 $DS_ENDREG End a Device Register Storage Area
$DS_ENDREG is a utility macro. Use this macro to mark the end of a
device register storage area. The macro produces no executable
code and does not call a supervisor routine.

$DS_ENDREG (no arguments)

Return Status Codes: Not Applicable.

10.66 $DS_ENDSERV End of Interrupt Service Routine
$DS_ENDSERV is a utility macro. Use this macro to mark the end of
an interrupt service routine. When executed, it restores registers
R0 and R1 and performs a return from interrupt instruction.

$DS_ENDSERV (no arguments)

Return Status Codes: Not Applicable.

10.67 $DS_ENDSTAT End of Statistics Section
$DS_ENDSTAT is a utility macro. You should use it to mark the end
of the statistics section of a diagnostic program. The macro
produces no executable code and no calls to supervisor routines.

$DS_ENDSTAT (no arguments)

Return Status Codes: Not Applicable.

10.68 $DS_ENDSUB End of a Subtest
$DS_ENDSUB is a utility macro. Use the macro at the end of each
subtest within a diagnostic program. The macro produces an error
message in the listing at assembly time if no corresponding
$DS_BGNSUB macro is found. In addition, the macro calls the
RENDSUB service routine in the supervisor to check both the test
numbers and the subtest numbers within each test for correct
sequence. If the routine does detect a sequencing error, it
notifies the operator and returns control to the command mode. The
routine also checks for Control C.

$DS_ENDSUB (no arguments)

Return Status Codes: Not Applicable.

10.69 $DS_ENDSUMMARY End of Summary Report Section
$DS_ENDSUMMARY is a utility macro. Use this macro to mark the end
of the summary report section of a diagnostic program. It checks
for Control C and generates a return instruction to return control
to the calling routine.

$DS_ENDSUMMARY (no arguments)

Return Status Codes: Not Applicable.

10.70 $DS_ENDTEST End of a Test
$DS_ENDTEST is a utility macro. Use this macro to mark the end of
each test in a diagnostic program. The macro moves a 1 into R0 to
indicate normal test completion. Then it checks for Control C, and
executes an RET instruction to return to the test sequencer
(dispatch routine in the supervisor).

$DS_ENDTEST (no arguments)

Return Status Codes: 1 in R0.

10.71 $DS_ENVDEF Define Environment Codes
$DS_ENVDEF is a utility macro. It defines symbols that can be used
for the ENV argument of the $DS_BGNMOD macro. The $DS_BGNMOD macro
invokes the $DS_ENVDEF macro. The following symbols are defined.

Symbol	Definition
CEP_FUNCTIONAL	Cluster Environment Functional
CEP_REPAIR	Cluster Environment Repair
SEP_FUNCTIONAL	System Environment Functional
SEP_REPAIR	System Environment Repair

No supervisor routine is called by the macro.

$DS_ENVDEF (no arguments)

Return Status Codes: Not Applicable.

10.72 $DS_ERRDEF Define Error Call Arglist Offsets
$DS_ERRDEF is a utility macro. It defines symbolic arguments for
the $DS_ERRDEV, $DS_ERRHARD, $DS_ERRSOFT, and $DS_ERRSYS macros.
The following symbols are defined:

Symbol	Definition
ERR$_NUM	error number
ERR$_UNIT	logical unit number
ERR$_MSGADR	header message address
ERR$_POINTER	extended error print routine
ERR$_P1	additional data
ERR$_P2	additional data
ERR$_P3	additional data
ERR$_P4	additional data
ERR$_P5	additional data
ERR$_P6	additional data

No supervisor routine is called by this macro.

$DS_ERRDEF (no arguments)

Return Status Codes: Not Applicable.

10.73 $DS_ERRDEV_x Print Device Fatal Error Header Information
$DS_ERRDEV_x is a supervisor service macro. It calls a supervisor
routine to generate a 3-line error message for the operator. The
message indicates the program title and version number, pass
number, test and subtest numbers, and time stamp.

$DS_ERRDEV_x [num], [unit], [msgadr], [prlink], [pl-6]

num   =   a unique error number within the current subtest. NUM
            is initialized by the $DS_BGNTEST and $DS_BGNSUB macros
            and automatically sequenced when not specified. The
            automatic sequencing of this parameter also includes
            all its default uses in the macros $DS_ERRHARD_x,
            $DS_ERRSOFT_x, and $DS_ERRSYS_x.

unit   =   the logical unit number of the unit under test.

msgadr =   the address of a counted ASCII string. This message is
            included in the third line of the error header message
            and should be a brief description of the error or a
            module call out message.

prlink =   the address of the error reporting routine. This is
            the address of a closed routine to print supplemental
            information about the error that has occurred. The
            error reporting code at this address must be surrounded
            by $DS_BGNMESSAGE and $DS_ENDMESSAGE. Execution of this
            section of code is not contingent on the Halt On Error
            flag.

pl-6   =   one to six optional parameters that may be passed to
            the error print routine. If specified, they must be
            used in sequence.

Return Status Codes: None.

                            NOTES
        1.   R2 through R11 are preserved within
             this service by the supervisor and
             are intact when the call to PRLINK
             is executed.

        2. $DS_ERRDEV_x may not be used between
             subtests.

10.74 $DS_ERRHARD_x Print Hardware Error Header Information
$DS_ERRHARD_x is a supervisor service macro. It calls a supervisor
routine to generate a 3-line error message for the operator. The
message indicates the program title and version number, pass
number, test and subtest numbers, and time stamp.

$DS_ERRHARD_x [num], [unit], [msgadr], [prlink], [pl-6]

num   = a unique error number within the current subtest. NUM
            is initialized by the $DS_BGNTEST and $DS_BGNSUB macros
            and automatically sequenced when not specified. The
            automatic sequencing of this parameter also includes
            all its default uses in the macros $DS_ERRDEV_x,
            $DS_ERRSOFT_x, and $DS_ERRSYS_x.

unit   = the logical unit number of the unit under test.

msgadr   = the address of a counted ASCII string. This message is
            included in the third line of the error header message
            and should be a brief description of the error or a
            module call out message.

prlink   = the address of the error reporting routine. This is
            the address of a closed routine to print supplemental
            information about the error that has occurred. The
            error reporting code at this address must be surrounded
            by $DS_BGNMESSAGE and $DS_ENDMESSAGE. Execution of this
            section of code is not contingent on the Halt On Error
            flag.

pl-6   = one to six optional parameters that may be passed to
            the error print routine. If specified, they must be
            used in sequence.

Return Status Codes: None.

                         NOTES
            1. R2 through R11 are preserved within
                this service by the supervisor and
                are intact when the call to PRLINK
                is executed.

            2. $DS_ERRHARD_x may not be used
                between subtests.

10.75 $DS_ERRNUM Insert Numeric Error Header Information
$DS_ERRNUM is a utility macro. It inserts an error number into an
argument list at the label given. If an error number is not given,
the next sequential error number will be used. The macro does not
call a supervisor routine.

$DS_ERRNUM label, [num]

label = label attached to argument list.

num   = error number to insert.

Return Status Codes: Not Applicable.

NOTE
$DS_ERRNUM generates executable code
that uses (destroys) R0. R0 is left
pointing to LABEL.

10.76 $DS_ERRSOFT_x Print Software Error Header Information
$DS_ERRSOFT_x is a supervisor service macro. It calls a supervisor
routine to generate a 3-line error message for the operator. The
message indicates the program title and version number, pass
number, test and subtest numbers, and time stamp.

$DS_ERRSOFT_x [num], [unit], [msgadr], [prlink], [p1-6]

num   = a unique error number within the current subtest. NUM
              is initialized by the $DS_BGNTEST and $DS_BGNSUB
              macros and automatically sequenced when not specified.
              The automatic sequencing of this parameter also
              includes all its default uses in the macros
              $DS_ERRHARD_x, $DS_ERRDEV_x, and $DS_ERRSYS_x.

unit   = the logical unit number of the unit under test.

msgadr   = the address of a counted ASCII string. This message is
              included in the third line of the error header message
              and should be a brief description of the error or a
              module call out message.

prlink   = the address of the error reporting routine. This is
              the address of a closed routine to print supplemental
              information about the error that has occurred. The
              error reporting code at this address must be
              surrounded by $DS_BGNMESSAGE and $DS_ENDMESSAGE.
              Execution of this section of code is not contingent on
              the Halt On Error flag.

p1-6   = one to six optional parameters that may be passed to
              the error print routine. If specified, they must be
              used in sequence.

Return Status Codes: None.

NOTES
   1. R2 through R11 are preserved within
       this service by the supervisor and
       are intact when the call to PRLINK
       is executed.

   2. $DS_ERRSOFT_x may not be used
       between subtests.

10.77 $DS_ERRSYS_x Print System Fatal Error Header Information
$DS_ERRSYS_x is a supervisor service macro. It calls a supervisor
routine to generate a 3-line error message for the operator. The
message indicates the program title and version number, pass
number, test and subtest numbers, and time stamp.

$DS_ERRSYS_x [num], [unit], [msgadr], [prlink], [pl-6]

num   = a unique error number within the current subtest. NUM
             is initialized by the $DS_BGNTEST and $DS_BGNSUB macros
             and automatically sequenced when not specified. The
             automatic sequencing of this parameter also includes
             all its default uses in the macros $DS_ERRHARD_x,
             $DS_ERRSOFT_x, and $DS_ERRDEV_x.

unit   = the logical unit number of the unit under test.

msgadr   = the address of a counted ASCII string. This message is
             included in the third line of the error header message
             and should be a brief description of the error or a
             module call out message.

prlink   = the address of the error the reporting code. This is
             the address of a closed routine to print supplemental
             information about the error that has occurred. The
             error reporting code at this address must be surrounded
             by $DS_BGNMESSAGE and $DS_ENDMESSAGE. Execution of this
             section of code is not contingent on the Halt On Error
             flag.

pl-6   = one to six optional parameters that may be passed to
             the error print routine. If specified they must be used
             in sequence.

Return Status Codes: None.

NOTES

1. R2 through R11 are preserved within
    this service by the supervisor and
    are intact when the call to PRLINK
    is executed.

2. $DS_ERRSYS_x may not be used between
    subtests.

10.78 $DS_ESCAPE Escape Program Sequence

$DS_ESCAPE is a utility macro. It provides a conditional exit from a diagnostic program test. The macro calls a supervisor routine to check the status of the Error flag. If the Error flag is set, control passes to the end of the subtest or test, depending on the argument supplied. The $DS_ESCAPE macro enables the programmer to eliminate execution of certain portions of a program if prior testing indicates that those portions would be redundant and bound to fail.

The escape sequence preserves the contents of R0.

$DS_ESCAPE arg

arg = TEST or SUB.

Return Status Codes: Not Applicable.

10.79 $DS_EXIT Unconditional Exit

$DS_EXIT is a utility macro. It causes an unconditional branch to the last statement in the current test or subtest (depending on ARG). The macro does not call a supervisor routine.

$DS_EXIT arg

arg = TEST, SUB, INIT, CLEAN, SERV, MESSAGE, or SUMMARY.

Return Status Codes: Not Applicable.

10.80 $DS_GETBUF_x Get Virtual Memory Space

$DS_GETBUF_x is a supervisor service macro. It calls a supervisor routine (DSX$GETBUF) to obtain memory space for buffer areas. The memory space is allocated at the logical end of the program. In the standalone environment, the physical memory allocation is contiguous. The routine also checks for Control C.

$DS_GETBUF_x pagcnt, [retadr], [phyadr], [region]

pagcnt = the number of pages of memory desired.

retadr = the address of a two-longword array to receive the virtual buffer limits.

phyadr = the address of a two-longword array to receive the physical start and end addresses.

region = allocate to : 0=(default) P0, 1=pl, 2=system space.

Return Status Codes
SS$_NORMAL: Service successfully completed.

SS$_ILLPAGCNT: Page count is less than 1.

SS$_VASFULL: Virtual address space full.

NOTE
If memory management is off, space may
be allocated only to P0.

10.81 $DS_GPHARD_x Get Hardware P-table Base Address
$DS_GPHARD_x is a supervisor service macro. It calls a supervisor
routine (RGPHARD) to obtain the address of the entry in the
P-table associated with the given logical unit number.

$DS_GPHARD_x unit, retadr

unit   = the logical unit number.

retadr = the longword to receive the base address of the P-table
           entry.

Return Status Codes
DS$_NORMAL: Service successfully completed.

DS$_ERROR: The argument list does not contain exactly two
elements, or the logical unit number specified is too large.

NOTE
P-table entries are stacked in memory in
reverse order.

10.82 $DS_HDRDEF Define Header Section Area
$DS_HDRDEF is a utility macro. It defines absolute, addressable
symbols for the header section area (the area defined by the
$DS_HEADER macro). The macro defines the following symbols:

Symbol	Definition
L$L_HEADLENGTH	length of the header data block
L$L_ENVIRON	program environment
L$A_NAME	program name text address
L$L_REV	program revision level
L$L_UPDATE	diagnostic engineering patch order
L$L_LASTAD	first free location after program
L$L_DTP	test dispatch table pointer
L$L_DEVP	device type list pointer
L$L_UNIT	number of units that can be tested
L$A_DREG	device register contents table pointer
L$A_ICP	address initialize
L$A_CCP	cleanup code pointer
L$A_REPP	summary report code pointer
L$A_STATAB	statistics table pointer
L$L_ERRTYP	number of types of $ERRSOFT and $ERRHARD
L$A_SECNAM	list of section name addresses
L$A_TSTCNT	pointer to number of tests

The macro does not call a routine in the supervisor.

$DS_HDRDEF [gbl]

gbl = GLOBAL or LOCAL.

Return Status Codes: Not Applicable.

10.83 $DS_HEADER Generate Program Header Section
$DS_HEADER is a utility macro. Use this macro at the beginning of
a diagnostic program to generate a table that defines the program
structure to the supervisor. The macro does not call a supervisor
routine.

$DS_HEADER pname, rev, [update], [nunit], [errtyp], [stat]

pname   = the program name string.

rev   = the program release level number.

update   = the DEPO patch level number.

nunit   = the maximum number of units that can be tested
            concurrently.

errtyp   = the number of types of error that can occur on the
            unit under test.

stat   = a pointer to statistics table.

Return Status Codes: Not Applicable.

NOTES
1. **REV and UPDATE are used as major and
    minor revision numbers and will be
    changed to MAJREV and MINREV in the
    future.**

2. **If STAT is specified it must be
    STATISTIC; refer to $DS_BGNSTAT.**

10.84 $DS_HPODEF Define Hardware P-table Entry Offsets
$DS_HPODEF is a utility macro. It defines offsets for items in the
hardware P-table, allowing the programmer to access the items with
their symbolic offsets. The macro does not call a routine in the
supervisor.

$DS_HPODEF (no arguments)

Return Status Codes: Not Applicable.

NOTES
1. **Only device-independent offsets are
    defined.**

2. Symbol	Description
HP$Q_DEVICE	quadword descriptor of device name
HP$W_SIZE	total size of P-table
HP$B_DRIVE	unit number
HP$T_DEVICE	ASCII device name with leading " _ ", max length = 11 characters
HP$A_DEVICE	device address for this UUT
HP$A_DVA	address used to directly address another UUT through this device
HP$A_LINK	address of P-table for device linking this to the CPU
HP$W_VECTOR	primary interrupt vector for device
HP$T_TYPE	ASCIC hardware, type, max length = 11

1Ø.85 $DS_INITSCB_x Initialize System Control Block

$DS_INITSCB_x is a supervisor service macro. It calls a supervisor routine (DSX$INITSCB), which sets the vectors in the system control block for supervisor handling. The routine copies a 512 byte image(SCB_IMAGE) into the system control block.

$DS_INITSCB_x (no arguments)

Return Status Codes
$DS_NORMAL: Service successfully completed.

1Ø.86 $DS_INLOOP_x Check for a Loop

$DS_INLOOP_x is a supervisor service macro. It calls a supervisor routine to test whether or not the program is looping on an error. The routine sets the low bit of RØ if the program is in a loop. It clears the low bit of RØ if the program is not in a loop.

$DS_INLOOP_x (no arguments)

Return Status Codes
DS$_NORMAL: The program is in a loop.

DS$_ERROR: The program is not in a loop.

1Ø.87 $DS_MMOFF_x Turn Memory Management Off

$DS_MMOFF_x is a supervisor service macro available to level 3 diagnostic programs only. It calls a supervisor service routine (DSX$MMOFF) that turns off memory management.

$DS_MMOFF (no arguments)

Return Status Codes: None.

10.88 $DS_MMON_x Turn Memory Management On
$DS_MMON_x is a supervisor service macro available to level 3 diagnostic programs only. It calls a supervisor routine (DSX$MMON) that turns memory management on.

$DS_MMON_x (no arguments)

Return Status Codes: None.

10.89 $DS_PAGE Assembler Page Control
$DS_PAGE is a utility macro. It causes the .SBTTL assembler directive to appear as the first output line of the next assembler listing page when you use the $DS_SBTTL macro. The $DS_PAGE macro inhibits the listing of the $DS_SBTTL macro call. If this were not done, the .SBTTL expansion would appear on the second line of the listing page. The assembler, therefore, would not recognize it for the heading on that page. The optional argument will cause the $DS_PAGE macro to include a .PAGE directive if the NUM argument is 1. The macro does not call a supervisor routine.

$DS_PAGE [num]

<u>num</u> = 0 or 1.
        1 indicates that a .PAGE directive should be generated.
        0, the default, indicates that a .PAGE directive should not be generated.

Return Status Codes: Not Applicable.

10.90 $DS_PARDEF Parameter Definitions
$DS_PARDEF is a utility macro. It defines the radix and exception mask arguments for the macros of the form $DS_ASKxxxx_x.

        PAR$_BIN
        PAR$_OCT
        PAR$_DEC
        PAR$_HEX
        PAR$_NO
        PAR$_YES
        PAR$_NODEF
        PAR$_ATLO
        PAR$_ATHI
        PAR$_ATDEF

No supervisor routine is called.

$DS_PARDEF [gbl]

<u>gbl</u> = GLOBAL or LOCAL. LOCAL is the default.

Return Status Codes: Not Applicable.

10.91 $DS_PARSE_x Parse

$DS_PARSE_x is a supervisor service macro. It calls a supervisor routine (DSX$PARSE) to parse an ASCII string in the buffer described, using a caller-supplied parse tree. When a match in the tree is found, the routine calls a caller-supplied action routine. Upon a mismatch, the supervisor routine branches to another point in the parse tree.

$DS_PARSE_x bufadr, tree, action

bufadr = the address of a quadword descriptor for the buffer.

tree   = the address of the tree structure to be used in parsing.

action = the address of the action routine.

Return Status Codes
DS$_NORMAL: Service successfully completed.

DS$_ERROR: Error match code was encountered in the parse tree.

DS$_OVERFLOW: Numeric input overflowed a quadword.

NOTE
You should define the tree structure with successive use of the $DS_CLI macro.

10.92 $DS_PRINTB_x Print Basic Error Information

$DS_PRINTB_x is a supervisor service macro. Use this macro in print subroutines called through one of the $DS_ERRxxxx_x macros to print basic error information. The $DS_PRINTB_x macro calls the extended print routine in the supervisor. The supervisor routine uses formatted ASCII output (FAO) to compile a string and then print it. Either of two supervisor control flags, IE1 or IE2, when set, will inhibit the printing of the message. Typically, one $DS_PRINTB_x macro is used to print one line. Therefore, four $DS_PRINTB_x macros would be used to print the explanatory message, the expected data, the received data, and the XOR data.

$DS_PRINTB_x fmtadr, [arg], [arg],...

fmtadr =   the address of a character string descriptor pointing to
            the control string. The control string consists of the
            fixed text of the output string and the conversion
            directives.

arg   =   the directive parameters contained in longwords. A
            parameter may be a value that is to be converted, the
            address of the string that is to be inserted, a length,
            or an argument count, depending on the directive. For
            each directive in the control string there may be
            corresponding parameters. Up to 16 arguments may be
            supplied.

Return Status Codes: None.

NOTE
Refer to Chapter 9, Paragraph 9.6, of
this manual and the VAX/VMS System
Services Manual for FAO information.

10.93 $DS_PRINTF_x Print a Forced Message

$DS_PRINTF_x is a supervisor service macro. Use this macro in print subroutines called through one of the $DS_ERRxxx_x macros to force the printing of error information. The $DS_PRINTF_x macro calls the extended print routine in the supervisor. The supervisor routine uses formatted ASCII output (FAO) to compile a string and then print it. The message will be printed regardless of the condition of the Inhibit Error flags in the supervisor.

$DS_PRINTF_x fmtadr, [arg], [arg],...

fmtadr = the address of a character string descriptor pointing to the control string. The control string consists of the fixed text of the output string and the conversion directives.

arg = the directive parameters contained in longwords. A parameter may be a value that is to be converted, the address of the string that is to be inserted, a length, or an argument count, depending on the directive. For each directive in the control string, there may be corresponding parameters. Up to 16 arguments may be supplied.

Return Status Codes: None.

NOTE
Refer to Chapter 10, Paragraph 10.6, of this manual and the VAX/VMS System Services Reference Manual for FAO information.

10.94 $DS_PRINTS_x Print Summary Report

$DS_PRINTS_x is a supervisor service macro. Use this macro in the summary routine to print a summary report. The macro calls the extended print routine in the supervisor. The supervisor routine uses formatted ASCII output (FAO) to compile a string and then print it. If the IES control flag (inhibit summary) in the supervisor is set, the summary report will not be printed.

$DS_PRINTS_x fmtadr, [arg], [arg],...

fmtadr = the address of a character string descriptor pointing to the control string. The control string consists of the fixed text of the output string and the conversion directives.

arg = the directive parameters contained in longwords. A parameter may be a value that is to be converted, the address of the string that is to be inserted, a length, or an argument count, depending on the directive. For each directive in the control string, there may be corresponding parameters. Up to 16 arguments may be supplied.

Return Status Codes: None.

> **NOTE**
> Refer to Chapter 9, Paragraph 9.6, of this manual and the <u>VAX/VMS System Services Reference Manual</u> for FAO information.

10.95 $DS_PRINTX_x Print Extended Error Information
$DS_PRINTX_x is a supervisor service macro. Use this macro in a print subroutine called by one of the $DS_ERRxxxx_x macros. The macro should be used to print information that supplements the basic error message, such as:

        failing addresses
        device register contents
        channel register contents
        additional explanations.

If any of the supervisor control flags IE1, IE2, or IE3 is set, the service will not print the message.

$DS_PRINT_x fmtadr, [arg], [arg],...

<u>fmtadr</u> = the address of a character string descriptor pointing to the control string. The control string consists of the fixed text of the output string and the conversion directives.

<u>arg</u>   = the directive parameters contained in longwords. A parameter may be a value that is to be converted, the address of the string that is to be inserted, a length, or an argument count, depending on the directive. For each directive in the control string there may be corresponding parameters. Up to 16 arguments may be supplied.

Return Status Codes: None.

> **NOTE**
> Refer to Chapter 9, Paragraph 9.6 of this manual and the <u>VAX/VMS System Services Reference Manual</u> for FAO information.

10.96 $DS_PSLDEF Processor Status Longword Definitions
$DS_PSLDEF is a utility macro. It defines the symbols for the bit mask positions and the mode bits in the processor status longword. No supervisor routine is called.

$DS_PSLDEF [gbl]

<u>gbl</u> = GLOBAL or LOCAL.

The following symbols are defined:

PSL$M_CBIT
PSL$M_VBIT
PSL$M_ZBIT
PSL$M_NBIT

PSL$K_KERNEL
PSL$K_EXEC
PSL$K_SUPER
PSL$K_USER.

Return Status Codes: Not Applicable.

10.97 $DS_RELBUF_x Release Buffer Space
$DS_RELBUF_x is a supervisor service macro. It calls a supervisor routine to release a buffer area in memory from program control.

$DS_RELBUF_x pagcnt, [retadr], [region]

pagcnt = the number of pages.

retadr = the address of a two-longword array to receive deallocated buffer limits.

region = release from: 0 = (default) P0, 1 = P1, 2 = system space.

Return Status Codes
SS$_NORMAL: Service successfully completed.

DS$_FRAGBUF: Buffer was not contiguous.

SS$_ILLPAGCNT: Page count is less than 1.

SS$_PAGOWNVIO: Page is owned by more privileged access mode.

10.98 $DS_SBTTL Specify Test or Subtest Subtitle
$DS_SBTTL is a utility macro. It generates an assembler directive to provide the given subtitle, with the diagnostic test or subtest name included, for the assembly listing. No supervisor routine is called. Use this macro only at the beginning of each test and subtest in a program. Use the normal assembler subtitle directive (.SBTTL) in all other cases.

$DS_SBTTL ascii, [align]

ascii = a subtitle string; maximum length = 50.
align = the psect alignment of the test (default = PAGE).

Return Status Codes: Not Applicable.

10.99 $DS_SCBDEF System Control Block Definitions
$DS_SCBDEF is a utility macro. It defines symbols for the system control block vector offsets. No supervisor routine is called.

$DS_SCBDEF [gbl]

gbl = GLOBAL or LOCAL (default = LOCAL).

The following symbols are defined:

SCB$L_ZERO	SCB$L_MACHCHK	SCB$L_KNLSTK
SCB$L_POWER	SCB$L_OPCDEC	SCB$L_OPCCUS
SCB$L_ROPRAND	SCB$L_RADRMOD	SCB$L_ACCESS
SCB$L_TRANSL	SCB$L_TBIT	SCB$L_BREAK
SCB$L_COMPAT	SCB$L_ARITH	SCB$L_CHMK
SCB$L_CHME	SCB$L_CHMS	SCB$L_CHMU
SCB$L_SFTLVL1	SCB$L_SFTLVL2	SCB$L_SFTLVL3
SCB$L_SFTLVL4	SCB$L_SFTLVL5	SCB$L_SFTLVL6
SCB$L_SFTLVL7	SCB$L_SFTLVL8	SCB$L_SFTLVL9
SCB$L_SFTLVL10	SCB$L_SFTLVL11	SCB$L_SFTLVL12
SCB$L_SFTLVL13	SCB$L_SFTLVL14	SCB$L_SFTLVL15
SCB$L_TIMER	SCB$L_RXDB	SCB$L_TXDB

Return Status Codes: Not Applicable.

10.100 $DS_SECDEF Section Definitions
$DS_SECDEF is a utility macro. Use the macro to define the test
sections in each program source module. Use this macro in all
program modules except the header module. No supervisor routine
is called.

$DS_SECDEF arg, arg, arg,...

arg = section name.

Return Status Codes: Not Applicable.

                        NOTE
        The order of arguments here must be the
        same as the order of arguments in the
        $DS_SECTION macro in the header module.

10.101 $DS_SECTION Section Definitions Table
$DS_SECTION is a utility macro. It generates a table of ASCII
section names used to define the test sections for the supervisor.
These section names should be used with the $DS_BGNTEST macro to
show which section the test belongs to. This macro should be used
in the program text section of the header module. No supervisor
routine is called.

$DS_SECTION arg, arg, arg,...

arg = section name.

Return Status Codes: Not Applicable.

                        NOTE
        The order of arguments here must be the
        same as the order of arguments in the
        $DS_SECDEF macro.

10.102 $DS_SETIPL_x Set Interrupt Priority Level
$DS_SETIPL_x is a supervisor service macro. Use this macro to
raise the IPL to block interrupts from the device under test.

$DS_SETIPL_x level

level = interrupt priority level.

Return Status Codes
DS$_NORMAL: Service successfully completed.

10.103 $DS_SETMAP_x Set Channel Adapter Mapping
$DS_SETMAP_x is a supervisor service macro available to level 3
diagnostic programs only. It enables you to set channel adapter
mapping for I/O transfers.

$DS_SETMAP_x unit, func, phyadr, [mapbas], [bytcnt], [datpth]

<u>unit</u>   =   the logical unit number.

<u>func</u>   =   the function code symbol specifying the type of mapping to be performed. See the symbols listed below.

<u>phyadr</u>   =   the address of a two-longword array describing the physical buffer start and end addresses. Normally these will be the start and end addresses returned by a $DS_GETBUF_x macro call.

<u>mapbas</u>   =   the adapter map register address. For the UBA (DW780) this parameter corresponds to the upper nine bits of the Unibus address (bits 17:09) to be used in the base address (BA) register of the desired device. The number supplied for the MAPBAS parameter to the UBA should be in the range of 0 to 495 (decimal).

For the MBA (RH780) the MAPBAS parameter selects the current map register through bits 16:09 of the virtual map register. The default for MAPBAS is zero.

<u>bytcnt</u>   =   the positive byte count (used for the MBA only). This field is ignored when the UBA is used. However, the field is checked for validity regardless of the adapter type to be used.

<u>datpth</u>   =   the UBA data path number (0-15). This field is ignored when the MBA is used.

Function argument symbols:

Symbol	Function
CHM$_FORWARD	Prime the adapter for a forward operation
CHM$_REVERSE	Prime the adapter for a reverse operation
CHM$_INVALIDATE	Invalidate all map entries
CHM$_MAP	Set requested mapping
CHM$_OFFSET	Mapping with byte offset
CHM$_MFWDV	Invalidate all map entries Set requested mapping Prime the adapter for a forward operation
CHM$_MFWDN	Do not invalidate any map entries Set requested mapping Prime the adapter for a forward operation
CHM$_NFWDN	Do not invalidate any map entries Do not set mapping Prime the adapter for a forward operation

CHM$_MREVV   Invalidate all map entries
                              Set requested mapping
                              Prime the adapter for a reverse operation

CHM$_MREVN   Do not invalidate any map entries
                              Set requested mapping
                              Prime the adapter for a reverse operation

CHM$_NREVN   Do not invalidate any map entries
                              Do not set mapping
                              Prime the adapter for a reverse operation

CHM$_MFWDVO   Invalidate all map entries
                              Set requested mapping with byte offset   (UBA
                              only)
                              Prime the adapter for a forward operation

CHM$_MFWDNO   Do not invalidate any map entries
                              Set requested mapping with byte offset   (UBA
                              only)
                              Prime the adapter for a forward operation

CHM$_MREVVO   Invalidate all map entries
                              Set requested mapping with byte offset   (UBA
                              only)
                              Prime adapter for reverse operation

CHM$_MREVNO   Do not invalidate any map entries
                              Set requested mapping with byte offset   (UBA
                              only)
                              Prime adapter for a reverse operation

Return Status Codes
DS$_NORMAL: Service successfully completed.

DS$_PROGERR: Program error. The buffer address is out of range, the base address is out of range (0 through 225 MBA, 0 through 495 UBA), or the specified byte count is too large.

DS$_IHWE: Adapter hardware error status was encountered before the mapping was performed. The error conditions must be cleared before the mapping will be performed.

DS$_ERROR: An error has been found while trying to associate a hardware P-table with the logical unit argument.

10.104 $DS_SETVEC_x Set System Control Block
$DS_SETVEC_x is a supervisor service macro available only to level 3 diagnostic programs. It calls a supervisor routine (DSX$SETVEC) to load the address of an exception or interrupt routine into the system control block, setting a system control block vector for program control.

$DS_SETVEC_x vector, isradr, [code]

vector = the absolute vector address.

isradr = the virtual address of the service routine.

code = 0 or 1; 0 = kernel stack; 1 = interrupt stack.

Return Status Codes
DS$_NORMAL: Vector modified.

DS$_IVADDR: Service address bits <1:0> are not zero.

DS$_IVVECT: Invalid vector.

DS$_ICBUSY: Interval clock busy.

10.105 $DS_SHOWCHAN_x Show Channel Registers
$DS_SHOWCHAN_x is a supervisor service macro available only to
level 3 diagnostic programs. It calls a supervisor routine
(DSX$SHOWCHAN) that displays on the operator's terminal the
configuration register and the status register for the channel
adapter in use. If the status indicates errors that require the
display of additional registers, those registers will also be
displayed. For example, if the status indicates an invalid map,
the relevant map register will be displayed. The display for each
register contains the register mnemonic, the physical address of
the register, the register contents, and a mnemonic description of
the register contents.

$DS_SHOWCHAN_x unit

unit = a logical unit number.

Return Status Codes
DS$_NORMAL: Service successfully completed.

DS$_PROGERR: An error has been found while trying to associate a
hardware P-table with the logical unit argument.

10.106 $DS_STRING Generate ASCII String
$DS_STRING is a utility macro. It generates a quadword descriptor,
providing an ASCII string count, and an ASCII string terminated by
a zero byte. No supervisor routine is called.

$DS_STRING text, [locsyml], [locsym2]

text   = ASCII string to be generated.
locsyml = address of an ASCII string.
locsym2 = address of an ASCIZ string.

Return Status Codes: None.

10.107 $DS_SUMMARY_x Execute the Summary Report Section
$DS_SUMMARY_x is a supervisor service macro. It calls a supervisor routine which, in turn, calls a user summary report routine to print out a summary message. The summary report routine is normally called at the completion of program execution.

$DS_SUMMARY_x (no arguments)

Return Status Codes: None.

10.108 $DS_WAITMS_x Millisecond Delay
$DS_WAITMS_x is a supervisor service macro. It calls a supervisor service routine that suspends program execution for a number of milliseconds equal to ten times the number specified.

$DS_WAITMS_x time, [rettim]

time   = the number of 10 millisecond units.

rettim = the longword address to receive unused time in 10 millisecond units.

Return Status Codes
DS$_NORMAL: Service successfully completed.

DS$_ICERR: Interval clock error.

DS$_PROGERR: Negative time interval specified.

SS$_EXQUOTA: Multiple time function error.

NOTES
1. Interrupts can occur.

2. **The wait function can be terminated prematurely.**

3. **Wait functions may not be nested.**

10.109 $DS_WAITUS_x Microsecond Delay
$DS_WAITUS_x is a supervisor service macro. It calls a supervisor routine that suspends program execution for a number of microseconds equal to ten times the number specified.

$DS_WAITUS_x time, [rettim]

time   = the number of 10 microsecond units.

rettim = the longword address to receive unused time in 10 microsecond units.

Return Status Codes
DS$_NORMAL: Service successfully completed.

DS$_ICERR: Interval clock error.

DS$_PROGERR: Negative (or less than overhead) time interval specified.

SS$_QUOTA: Multiple time function error.

NOTES
1. Interrupts can occur.

2. The wait function can be terminated prematurely.

3. Wait functions may not be nested.

10.110 $DS_$DECIMAL Retrieve and Range Check a Decimal Number

$DS_$DECIMAL prompt, low, high

prompt = the ASCIC name of the field used as a prompt for the operator, if needed.

low   = the low limit for the parameter supplied by the operator.

high   = the high limit for the parameter supplied by the operator.

Use this P-table descriptor macro to prompt the operator for a decimal value. After the range check, the value becomes the current VALUE.

10.111 $DS_$END   Finish Processing P-table

$DS_$END (no arguments)
Use this P-table descriptor macro to mark the end of the P-table descriptor.

10.112 $DS_$FETCH   Extract a Field from the P-table

$DS_$FETCH offset, bit, size

offset = a byte offset from the base of the P-table. 0 < OFFSET < 65536

bit   = a bit displacement from the beginning of the OFFSET parameter. 0 < BIT < 255.

size   = the size of the field (in bits) to be extracted from the P-table. 0 < SIZE < 32

Use this P-table descriptor macro to extract a field from the P-table. The parameters OFFSET, BIT, and SIZE specify the field in the P-table.

10.113 $DS_$HEXADECIMAL Retrieve and Range Check a Hexadecimal Number
$DS_$HEXADECIMAL prompt, low, high

prompt   =   the ASCIC name of the field used as a prompt for the operator, if needed.

low   =   the low limit for the parameter supplied by the operator.

high   =   the high limit for the parameter supplied by the operator.

Use this P-table descriptor macro to prompt the operator for a hexadecimal value.   After the range check, the value becomes the current VALUE.

10.114 $DS_$INITIALIZE Start P-table Processing
$DS_INITIALIZE device, length, max, driver

device   =   the hardware name for the device, e.g., RP06 or DW780.

length   =   the total length of the associated P-table.

max   =   the maximum allowable unit number in the device name. This number should be zero if a unit number is not allowed (as on a TM03).

driver   =   the two-character QIO device driver name.   This argument should be null if it is not applicable.

Use this P-table descriptor macro as the first of the P-table descriptor macros.

10.115 $DS_$LITERAL   Define a Constant Value
$DS_$LITERAL value

value   =   a constant.

Use this P-table descriptor macro to create a constant value for VALUE from the program, if there is no need to retrieve a value from the operator.

10.116 $DS_$OCTAL Retrieve and Range Check an Octal Number
$DS_$OCTAL prompt, low, high
prompt   =   the ASCIC name of the field used as a prompt for the operator, if needed.

low   =   the low limit for the parameter supplied by the operator.

high   =   the high limit for the parameter supplied by the operator.

Use this P-table descriptor macro to prompt the operator for an octal value. After the range check, the value becomes the current VALUE.

10.117 $DS_$STORE Insert a Value into the P-table

$DS_$STORE offset, bit, size

offset = a byte offset from the base of the P-table. 0 < OFFSET < 65536.

bit = a bit displacement from the beginning of the OFFSET parameter. 0 < BIT < 255.

size = the size of the field (in bits) be inserted into the P-table. 0 < SIZE < 32.

10.118 $DS_$STRING Retrieve and Verify an ASCII String

$DS_$STRING prompt, strings

prompt = the ASCIC name of the field used as a prompt for the operator, if needed.

strings = a list of valid strings.

Use this P-table descriptor macro to prompt the operator for an ASCII string. The macro scans the input stream for a matching string.

10.119 $FAO_x Formatted ASCII Output

$FAO_x is a VMS system service. It converts binary values into ASCII characters and returns the converted characters in an output string. It can be used to:

- Insert variable character string data (filename, for example) into an output string.

- Convert binary values into the ASCII representations of their decimal, hexadecimal, or octal equivalents and substitute the results into an output string.

$FAO_x ctrstr, [outlen], outbuf, [p1], [p2]..., [pn]

ctrstr = the address of a character string descriptor pointing to the control string. The control string consists of the fixed text of the output string and FAO directives.

outlen = the address of a word to receive the actual length of the output string returned.

outbuf   =   the address of a character string descriptor pointing
               to the output buffer. The fully formatted output
               string is returned in this buffer.

pl -- pn =   the directive parameters contained in longwords.
               Depending on the directive, a parameter may be a value
               that is to be inserted, a length, or an argument
               count. Each directive in the control string may
               require a corresponding parameter or parameters.

Return Status Codes
SS$_NORMAL: Service successfully completed.

SS$_BUFFEROVF: Service successfully completed. The formatted
output string overflowed the output buffer, and has been
truncated.

SS$_BADPARAM: An invalid directive was specified in the FAO
control string.

<div align="center">NOTES</div>

1. The $FAO_S macro form uses a PUSHL
    instruction for all parameters (Pl
    through Pn) coded in the macro
    instruction. If a literal is
    specified, it must be preceded with
    a number sign (#) character or
    loaded into a register.

2. A maximum of 20 parameters can be
    specified in the $FAO_x macro
    instruction. If more than 20
    parameters are required, use the
    $FAOL_x macro.

3. The FAO system service executes at
    the access mode of the caller and
    does not check whether address
    arguments are accessible before it
    executes. Therefore, an access
    violation causing an exception
    condition occurs if an input field
    cannot be read or, in some cases, if
    an invalid length is specified.

10.119.1 FAO Directives

An FAO directive has the format:

    !DD

!   (exclamation mark) indicates that the following character or
     characters are to be interpreted as an FAO directive.

DD   is a 1 or 2 character code indicating the action that FAO is to perform. Each directive may require one or more input parameters on the call to FAO. Directives must be specified using uppercase letters.

Optionally, a directive may include:

>   A repeat count
>   An output field length

A repeat count is coded as follows:

>   !n(DD)

where n is a decimal value indicating that FAO is to repeat the directive for the specified number of parameters.

An output field length is specified as follows:

>   !lengthDD

where length is a decimal value instructing FAO to place the output resulting from a directive into a field of length characters in the output string.

A directive may contain both a repeat count and an output length, as shown below:

>   !n(lengthDD)

Repeat counts and output field lengths may be specified as variables by using a # (number sign) in place of an absolute numeric value. If a # is specified for a repeat count, the next parameter passed to FAO must contain the count. If a # is specified for an output field length, the next parameter must contain the length value.

If a variable output field length is specified with a repeat count, only one length parameter is required. Each output string will have the specified length.

10.119.2 FAO Control String and Parameter Processing
An FAO control string may be any length and may contain any number of FAO directives. The only restriction is on the use of the ! character (ASCII code X'21') in the control string. If a literal ! is required in the output string, the directive !! provides it.

When FAO processes a control string, each character that is not part of a directive is written into the output buffer. When a directive is encountered, it is validated. If it is not a valid directive, FAO terminates and returns an error status code. If the directive is valid, and if it requires one or more parameters, the next consecutive parameters specified are analyzed and processed.

FAO reads parameters from the argument list. It does not check the number of arguments it has been passed. If there are not enough parameters coded in the argument list, FAO will continue reading past the end of the list. It is your responsibility, when coding a call to FAO, to ensure that there are enough parameters to satisfy the requirements of all the directives in the control string.

Table 10-1 summarizes the FAO directives and lists the parameters required by each directive.

Table 10-1   Summary of FAO Directives

Character String Substitution

Directive	Function	Parameters*
!AC	Insert a counted ASCII string.	Address of the string; the first byte must contain the length.
!AD	Insert an ASCII string.	1. Length of string 2. Address of string
!AF	Insert an ASCII string. Replace all nonprintable ASCII codes with periods (.).	1. Length of string 2. Address of string
!AS	Insert an ASCII string.	Address of quadword character string descriptor pointing to the string.

Numeric Conversion (zero-filled)

Directive	Function	Parameters*
!OB !OW	Convert a byte to octal. Convert a word to octal.	Value to be converted to ASCII representation.

Table 10-1 Summary of FAO Directives (Cont)

Directive	Function	Parameters*
!OL	Convert a longword to octal.	For byte or word conversion, FAO uses only the low-order byte or word of the longword parameter.
!XB	Convert a byte to hexadecimal.	
!XW	Convert a word to hexadecimal.	
!XL	Convert a longword to hexadecimal.	
!ZB	Convert an unsigned decimal byte.	
!ZW	Convert an unsigned decimal word.	
!ZL	Convert an unsigned decimal longword.	

Numeric Conversion (blank-filled)

Directive	Function	Parameters*
!UB	Convert an unsigned decimal byte.	Value to be converted to ASCII representation.
!UW	Convert an unsigned decimal word.	
!UL	Convert an unsigned decimal longword.	

Directive	Function	Parameters*
!SB	Convert a signed decimal byte.	For byte or word conversion, FAO uses only the low-order byte or word of the longword parameter.
!SW	Convert a signed decimal word.	
!SL	Convert a signed decimal longword.	

Output String Formatting

Directive	Function	Parameters*
!/	Insert new line (CR/LF).	None
!_	Insert a tab.	
!^	Insert a form feed.	
!!	Insert an exclamation mark.	
!%S	Insert 'S' if most recently converted numeric value is not 1.	
!%T	Insert the system time.	Address of a quadword value to be converted to ASCII. If 0 is specified, the current system time is used.
!%D	Insert the system date and time.	

Table 10-1   Summary of FAO Directives (Cont)

Directive	Function	Parameters*
!n< !>	Define output field width of "n" characters. Format all data and directives within delimiters, < and > left-justified, and blank-filled within the field.	None
!n*c	Repeat the character C in the output string n times.	None

Parameter Interpretation

!-	Reuse the last parameter in the list.	None
!+	Skip the next parameter in the list.	None

* If a variable repeat count and/or a variable output field length is specified with a directive, parameters indicating the count and/or length must precede other parameters required by the directive.

10.120 $FAOL_x Formatted ASCII Output with List Parameter
The Formatted ASCII Output with List Parameter macro provides an alternate way to specify input parameters for a call to the FAC system service.

$FAOL_x ctrstr, [outlen], outbuf, prmlst

ctrstr =   the address of a character string descriptor pointing to the control string. The control string consists of the fixed text of the output string and conversion directives.

outlen =   the address of a word to receive the actual length of the output string returned.

outbuf =   the address of a character string descriptor pointing to the output buffer. The fully formatted output string is returned in this buffer.

prmlst =   the address of the parameter list of longwords to be used as P1 through Pn.

The parameter list may be a data structure that already exists in a program and from which certain values are to be extracted.

Return Status
Same as for FAO system service.

10.121 $GETCHN_x Get I/O Channel Information
$GETCHN_x is a VMS system service macro available only to level 2 and 2R diagnostic programs. It calls a VMS service that returns information about a device to which an I/O channel has been assigned. Two sets of information may be requested.

1.   The primary device characteristics.
2.   The secondary device characteristics.

In most cases, the two sets of characteristic information are identical. However, the two sets provide different information in the following cases:

1.   If the device has an associated mailbox, the primary characteristics are those of the assigned device and the secondary characteristics are those of the associated mailbox.

2.   If the device is a spooled device, the primary characteristics are those of the intermediate device and the secondary characteristics are those of the spooled device.

3.   If the device represents a logical link on the network, the secondary characteristics contain information about the link.

$GETCHN_x chan, [prilen], [pribuf], [seclen], [secbuf]

chan   =   the channel number. The channel number of a device is returned by the Assign I/O Channel (ASSIGN) system service.

prilen =   the address of a word to receive the length of the primary characteristics.

pribuf =   the address of a character string descriptor pointing to the buffer that is to receive the primary device characteristics. An address of 0 (the default) implies that no buffer is specified.

seclen =   the address of a word to receive the length of the secondary characteristics.

secbuf =   the address of a character string descriptor pointing to the buffer that is to receive to secondary device characteristics. An address of 0 (the default) implies that no buffer is specified.

Return Status

SS$_BUFFEROVF: Service successfully completed. The device information returned has overflowed the buffers provided and has been truncated.

SS$_NORMAL: Service successfully completed.

SS$_ACCVIO: A buffer descriptor cannot be read, or a buffer or buffer length cannot be written, by the caller.

SS$_IVCHAN: An invalid channel number was specified. That is, a channel number of 0 or a number larger than the number of channels available.

SS$_NOPRIV: The specified channel is not assigned, or was assigned from a more privileged access mode.

Privilege Restrictions

The Get I/O Channel Information service can be performed only on assigned channels and from access modes that are equal to or more privileged than the access mode from which the original channel assignment was made.

                                    NOTE
                The Get I/O Device Information (GETDEV)
                system   service   returns   the   same
                information as the Get I/O Channel
                Information system service.

Format of Device Information

The GETCHN and GETDEV system services return information in a user-supplied 53-byte buffer. Symbolic names defined in the $DIBDEF macro represent offsets from the beginning of the buffer.

The field offset names, lengths, and contents are listed below.

Field Name	Length (bytes)	Contents
DIB$L_DEVCHAR	4	Device characteristics
DIB$B_DEVCLASS	1	Device class
DIB$B_TYPE	1	Device type
DIB$W_DEVBUFSIZ	2	Device buffer size
DIB$L_DEVDEPEND	4	Device dependent information
DIB$W_UNIT	2	Unit number
DIB$W_DEVNAMOFF	2	Offset to device name string
DIB$L_PID	4	Process identification of device owner
DIB$L_OWNUIC	4	User identification code
DIB$W_VPROT	2	Volume protection mask
DIB$W_ERRCNT	2	Hard error count for device
DIB$L_OPCNT	4	Operation count
DIB$W_VOLNAMOFF	2	Offset to volume label string

The device name string and volume label string are returned in the buffer as counted ASCII strings and must be located by their offset values.

Any fields inapplicable to a particular device are returned as zeros.

For further details on the contents of this buffer, and on device-dependent information returned, refer to the VAX/VMS I/O User's Guide.

10.122 $GETTIM_x Get Time

$GETTIM_x is a VMS system service macro. It calls a VMS service that gives the current system time in the 64-bit system format suitable for input to the Set Timer (SETIMR) system service.

$GETTIM_x timadr

timadr =   the address of a quadword that is to receive the current time in 64-bit format.

Return Status Codes
SS$_NORMAL: Service successfully completed.

SS$_ACCVIO: The quadword to receive the time cannot be written by the caller.

10.123 $HIBER_S Hibernate
The Hibernate macro calls a VMS system service that allows a
process to make itself inactive but to remain known to the system,
so that it can be interrupted, for example, to receive ASTs. A
Hibernate request is a wait-for-wake-event request. When a wake
is issued for a hibernating process with the Wake system service,
the process continues execution at the instruction following the
Hibernate call.

$HIBER_S (no arguments)

Return Status Codes
SS$_NORMAL: Service successfully completed.
                          NOTES
     1.   A hibernating process can be
           swapped out of the balance set if
           it is not locked into the balance
           set.

     2.   The wait state caused by this
           system service can be interrupted
           by an asynchronous system trap
           (AST) if (1) the access mode at
           which the AST is to execute is
           equal to or more privileged than
           the access mode from which the
           hibernate request was issued and
           (2) the process is enabled for ASTs
           at that access mode.

     3.   If one or more wakeup requests are
           issued for the process while it is
           not hibernating, the next Hibernate
           call returns immediately. That is,
           the process does not hibernate. No
           count is maintained of outstanding
           wakeup requests.

     4.   Only the _S macro form is provided
           for the Hibernate system service.

10.124 $QIO_x Queue I/O Request
$QIO_x is a VMS system service macro available only to level 2 and
2R diagnostic programs. It calls a VMS service that begins an
input or output operation. The service queues a request to a
channel associated with a specified device. Control returns
immediately to the calling program. The program can synchronize
I/O completion in any of three ways.

     1.   Specify the address of an AST routine that is to execute
           when the I/O operation is completed.

     2.   Wait for a specified event flag to be set.

3. Poll the specified I/O status block for a completion status.

The service clears the event flag and the I/O status block, if they are specified, before it queues the I/O request.

$QIO_x efn, chan, func, [iosb], [astadr], [astprm], [pl], [p2], [p3], [p4], [p5], [p6]

efn   = the number of the event flag that is to be set at request completion. If the event flag number is not specified, the default value of 0 will cause errors in the supervisor.

chan   = the number of the I/O channel to which the request is directed. Use the ASSIGN service to obtain this number.

func   = the function code and modifier bits that specify the operation to be performed. The code is expressed symbolically. For reference purposes, the function codes are listed in the following notes. Details on valid I/O function codes and the parameters required by each are documented in the VAX/VMS I/O User's Guide.

iosb   = the address of the quadword I/O status block that is to receive final completion status.

astadr = the entry point address of the AST routine to be executed when the I/O operation is completed. If specified, the AST routine executes at the access mode from which the QIO service was requested.

astprm = the AST parameter to be passed to the AST service routine.

pl to p6 = optional device-specific and function-specific I/O request parameters.

         The first parameter may be specified as Pl or PlV, depending on whether the function code requires an address or a value, respectively. If the keyword is not used, Pl is the default. That is, the argument is considered an address.

         P2 through P6 are always interpreted as values. If they are to be used as addresses, preface the numbers with #.

Return Status Codes
SS$_NORMAL: Service successfully completed. The I/O request packet was successfully queued.

SS$_ACCVIO: The I/O status block cannot be written by the caller.

This status code may also be returned if parameters for device-dependent function codes are specified incorrectly.

SS$_EXQUOTA: The process has exceeded its buffered I/O quota or direct I/O quota and has disabled resource wait mode with the Set Resource Wait Mode (SETRWM) system service. Or, the process has exceeded its AST limit quota or buffer space quota.

SS$_ILLEFC: An illegal event flag number was specified.

SS$_INSFMEM: Insufficient system dynamic memory is available to complete the service, and the process has disabled resource wait mode with the Set Resource Wait Mode (SETRWM) system service.

SS$_IVCHAN: An invalid channel number was specified. That is, a channel number of 0 or a number larger than the number of channels available.

SS$_NOPRIV: The specified channel does not exist, or was assigned from a more privileged access mode.

SS$_UNASEFC: The process is not associated with the cluster containing the specified event flag.

SS$_ABORT: A network logical link was broken.

Privilege Restrictions
The Queue I/O Request system service can be performed only on assigned I/O channels and only from access modes that are equal to or more privileged than the access mode from which the original channel assignment was made.

Resources Required/Returned
1.   Queued I/O requests use three quotas:

   ●   the process's quota for buffered I/O or direct I/O
   ●   the process's quota for buffer space
   ●   the process's AST limit quota, if an AST service routine is specified.

2.   System dynamic memory is required to construct a data base to queue the I/O request. Additional memory may be required on a device-dependent basis.

NOTES
1.   The specified event flag is set even if the service terminates without queuing an I/O request.

2. Refer to Chapter 9, Paragraph
9.3.3.4 for information on the I/O
status block format.

3. Many services return character
string data, and write the length of
the data returned in a word provided
by the caller. Function codes for
the QIO system service require
length specifications in longwords.
If lengths returned by other
services are to be used as input
parameters for QIO requests, a
longword should be reserved to
ensure that no error occurs when
QIO reads the length.

4. For information on performing input
and output operations on a network,
see the VAX-11 DECnet User's Guide.

5. The queue I/O service provides
special features for diagnostic
functions.

Diagnostic operations are performed via physical I/O functions
that specify a diagnostic buffer. The diagostic buffer must be
large enough to receive the final device register contents at the
end of the operation.

A diagnostic buffer is specified by parameter 6 (P6) of all
physical I/O functions. If this parameter is nonzero, then a
diagnostic buffer is specified and the issuing process must have
the diagnostic privilege.

Specification of a diagnostic buffer address causes the Queue I/O
system service to allocate a buffer and store the address of the
buffer in the I/O packet (IRP$L_DIAGBUF). The virtual address of
the requester's buffer is stored in the allocated buffer, and the
diagnostic bit is set in the I/O packet status word. When the I/O
operation is completed, the final device register contents are
stored in the buffer. The I/O packet is submitted to the I/O
posting routine. The I/O posting routine determines that the
diagnostic buffer bit is set and transfers the information to the
requester's buffer. The allocated buffer is then returned to the
dynamic storage pool. The information transferred to the
requester's buffer has the format shown in Figure 10-1. Refer to
Chapter 9, Paragraph 9.3.3 for more details.

```
31                                          00
┌─────────────────────────────────────────┐
│                                           │
│          OPERATION START TIME             │
│            IN 64 BIT FORMAT               │
│                                           │
├─────────────────────────────────────────┤
│                                           │
│        OPERATION COMPLETION TIME          │
│            IN 64 BIT FORMAT               │
│                                           │
├─────────────────────────────────────────┤
│                                           │
│       FINAL ERROR COUNTER CONTENTS        │
│                                           │
├─────────────────────────────────────────┤
│                                           │
│        NUMBER OF DEVICE REGISTERS         │
│                                           │
├─────────────────────────────────────────┤
│                                           │
│                                           │
│            DEVICE REGISTERS,              │
│            ONE PER LONGWORD               │
│                                           │
│                   .                       │
│                   .                       │
│                   .                       │
│                                           │
└─────────────────────────────────────────┘
```
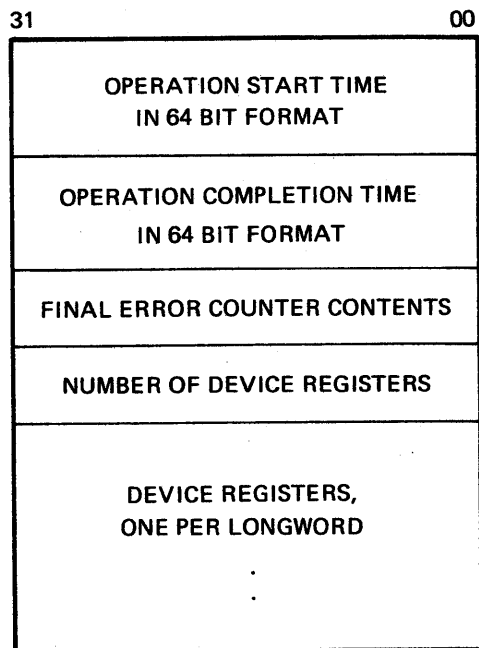
TK-3002

Figure 10-1 Diagnostic Buffer Format

10.125 $QIOW_x Queue I/O Request and Wait for Event Flag

$QIOW_x is a VMS system service macro available only to level 2 and 2R diagnostic programs. It calls a VMS service that combines the Queue I/O Request and Wait for Event flag system services. Use this macro where the program must wait for I/O completion before proceeding. The macro takes the same arguments as $QIO_x.

$QIOW_x efn, chan, func, [iosb], [astadr], [astprm], [p1], [p2], [p3], [p4], [p5], [p6]

efn   =   the number of the event flag that is to be set at request completion. If the event flag number is not specified, the default value of 0 will cause errors in the supervisor.

chan   =   the number of the I/O channel to which the request is directed. Use the ASSIGN service to obtain this number.

func   =   the function code and modifier bits that specify the operation to be performed. The code is expressed symbolically.

> iosb = th. address of the quadword I/O status block that is to receive final completion status.

> astadr = the entry point address of the AST routine to be executed when the I/O is completed. If specified, the AST routine executes at the access mode from which the QIOW service was requested.

> astprm = the AST parameter to be passed to the AST service routine.

> pl to p6 = optional device-specific and function-specific I/O request parameters.

For return status, privilege restrictions, and resources required/returned, refer to the description of the QIO system service in Paragraph 9.3.3 of Chapter 9.

10.126 $READEF_x Read Event Flags

$READEF_x is a VMS system service macro. It calls a VMS service that returns the status of all 32 event flags in an event flag cluster.

$READEF_x efn, state

> efn = the number of any event flag within the cluster to be read. A flag number of 0 through 31 specifies cluster 0. A flag number of 32 through 63 specifies cluster 1.

> state = the address of a longword to receive the current status of all event flags in the cluster.

Return Status Codes

SS$_WASCLR: Service successfully completed. The specified event flag is clear.

SS$_WASSET: Service successfully completed. The specified event flag is set.

SS$_ACCVIO: The longword that is to receive the current state of all event flags in the cluster cannot be written by the caller.

SS$_ILLEFC: An illegal event flag number was specified.

SS$_UNASEFC: The process is not associated with the cluster containing the specified event flag.

10.127 $SETEF_x Set Event Flag

$SETEF_x is a VMS system service macro. It calls a VMS service that sets the event flag specified. Any processes waiting for the event flag are made executable.

$SETEF_x efn

efn = the number of event flag to be set.

Return Status Codes
SS$_WASCLR: Service successfully completed. The specified event flag is clear.

SS$_WASSET: Service successfully completed. The specified event flag is set.

SS$_ILLEFC: An illegal event flag number was specified.

SS$_UNASEFC: The process is not associated with the cluster containing the specified event flag.

10.128 $SETIMR_x Set Timer
$SETIMR_x is a VMS system service macro. It calls a VMS service that allows a program to schedule the setting of an event flag and/or the queuing of an AST at some future time. You can specify the time for the event as an absolute time or as a delta time. However, the standalone supervisor does not support absolute times.

$SETIMR_x efn, daytim, [astadr], [reqidt]

efn   = the number of the event flag to set when the time
             interval expires. You must specify the EFN, since the
             default value is 0, and will cause supervisor errors.

daytim   = the address of the quadword containing the expiration
             time value.

             A positive time value indicates an absolute time at
             which the timer is to expire.

             A negative time value indicates an offset (delta time)
             from the current time.

astadr   = the address of the entry mask for an AST service
             routine to be called when the time interval expires.
             if this argument is not specified, the default value
             of 0 is supplied. 0 shows that no AST is to be queued.

reqidt   = a request identification number. The default value is
             0. You can specify a unique request identificaton
             number in each Set Timer request. Or, you can give the
             same request identification number to related Set
             Timer requests. You can use the request identification
             number later to cancel Set Timer requests. If an
             ASTADR argument is also supplied, the request
             identification number is passed to the AST routine.

Return Status Codes
SS$_NORMAL: Service successfully completed.

SS$_ACCVIO: The expiration time cannot be read by the caller.

SS$_EXQUOTA: The process quota for timer entries has been exceeded, and the process has disabled resource wait mode with the Set Resource Mode (SETRWM) system service.

SS$_ILLEFC: An illegal event flag number was specified.

SS$_IVTIME: An absolute expiration time that was specified has already passed, or the time was specified as 0.

SS$_INSFMEM: Insufficient dynamic memory is available to allocate a timer queue entry, and the process has disabled resource wait mode with the Set Resource Wait Mode (SETRWM) system service.

SS$_UNASEFC: The process is not associated with the cluster containing the specified event flag.

Resources Required/Returned
1. The Set Timer system service requires dynamic memory.

2. The Set Timer system service uses the process's quota for timer queue entries.

NOTES
1. The access mode of the caller is the access mode of the request and of the AST.

2. The Convert ASCII String to Binary Time (BINTIM) system service can be used to convert a specified ASCII string to the quadword time format required as input to the SETIMR service.

10.129 $SETPRT_x Set Protection on Pages

$SETPRT_x is a VMS system service macro available only to level 2 and 2R diagnostic programs. It calls a VMS service that enables an image running in a process to change the protection on a page or range of pages.

$SETPRT_x inadr, [retadr], [acmode], prot, [prvprt]

inadr   =   the address of a two-longword array containing the starting and ending virtual addresses of the pages on which protection is to be changed. If the starting and ending virtual addresses are the same, a single page is changed. Only the virtual page number portion of the virtual address is used. The low-order 9 bits are ignored.

retadr =   the address of a two-longword array to receive the starting and ending virtual addresses of the pages that have had their protection changed.

acmode =   the access mode on behalf of which the request is being made. The specified access mode is maximized with the access mode of the caller. The resultant access mode must be equal to or more privileged than the access mode of the owner of each page in order to change the protection.

prot   =   the new protection specified in bits 0 through 3 in the format of the hardware page protection. The high-order 28 bits are ignored. Symbolic names defining the protection codes are listed in Note 2. If specified as 0, the default access of kernel read-only is used.

prvprt =   the address of a byte to receive the protection previously assigned to the last page whose protection was changed. This argument is useful only when protection for a single page is being changed.

Return Status Codes

SS$_NORMAL: Service successfully completed.

SS$_ACCVIO: 1. The input address array cannot be read by the caller. 2. The output address array or the byte to receive the previous protection cannot be written by the caller. 3. An attempt was made to change the protection of a nonexistent page.

SS$_EXQUOTA: The process exceeded its paging file quota while changing a page in a read-only private section to a read/write page.

SS$_IVPROTECT: The specified protection code has a numeric value of 1 or is greater than 15 (decimal).

SS$_LENVIO: A page in the specified range is beyond the end of the program or control region.

SS$_NOPRIV: A page in the specified range is in the system address space.

SS$_PAGOWNVIO: Page owner violation. An attempt was made to change the protection on a page owned by a more privileged access mode.

Privilege Restrictions
For pages in global sections, the new protection can alter only the accessibility of the page for modes less privileged than the the owner of the page.

NOTES
1. If an error occurs while changing page protection, the return array, if requested, indicates the pages that were successfully changed before the error occurred. If no pages have been affected, both longwords in the return address array contain -1.

2. Hardware protection code symbols:

Symbol	Meaning
PRT$C_NA	No access
PRT$C_KR	Kernel read only
PRT$C_KW	Kernel write
PRT$C_ER	Executive read only
PRT$C_EW	Executive write
PRT$C_SR	Supervisor read only
PRT$C_SW	Supervisor write
PRT$C_UR	User read only
PRT$C_UW	User write
PRT$C_ERKW	Executive read; kernel write
PRT$C_SRKW	Supervisor read; kernel write
PRT$C_SREW	Supervisor read; executive write
PRT$C_URKW	User read; kernel write
PRT$C_UREW	User read; executive write
PRT$C_URSW	User read; supervisor write

These symbols are defined by $PRTDEF.

10.130 $UNWIND_x Unwind Call Stack
$UNWIND_x is a VMS service macro. It calls a system service that allows a condition handling routine to unwind the procedure call stack to a specified depth. A new return address can be specified to alter the flow of execution when the topmost call frame has been unwound.

$UNWIND_x [depadr], [newpc]

depadr   =   the address of a longword indicating the depth to which the stack is to be unwound. A depth of 0 indicates the call frame that was active when the condition occurred, 1 indicates the caller of that frame, 2 indicates the caller of the caller of the frame, and so on. If depth is specified as 0 or less, no unwind occurs. If no address is specified, the unwind is performed to the caller of the frame that established the condition handler.

newpc   =   the address to be given control when the unwind is complete.

Return Status Codes
SS$_NORMAL: Service successfully completed.

SS$_ACCVIO: The call stack is not accessible to ther caller. This condition is detected when the call stack is scanned to modify the return address.

SS$_INSFRAME: There are insufficient call frames to unwind to the specified depth.

SS$_NOSIGNAL: No signal is currently active for an exception condition.

SS$_UNWINDING: An unwind is already in progress.

NOTE
The actual unwind is not performed immediately. Rather, the return addresses in the call stack are modified so that when the condition handler returns, the unwind procedure is called from each frame that is being unwound.

10.131 $WAITFR_x Wait for Single Event Flag
$WAITFR_x is a VMS system service macro. It calls a service that tests a specific event flag. If the flag is set, control returns to the calling program immediately. If the flag is cleared, the process is placed in a wait state until the event flag is set.

$WAITFR_x efn

efn = the number of the event flag for which to wait.

Return Status Codes
SS$_NORMAL: Service successfully completed.

SS$_ILLEFC: An illegal event flag number was specified.

SS$_UNASEFC: The process is not associated with the cluster containing the specified event flag.

> **NOTE**
> The wait state caused by this service
> can be interrupted by an asynchronous
> system trap (AST), if (1) the access
> mode at which the AST executes is less
> than or equal to the access mode from
> which the wait was issued, and (2) the
> process is enabled for ASTs at that
> access mode.
>
> When the AST service routine completes
> execution, the system repeats the
> WAITFR request. If the event flag has
> been set, the process resumes execution.

10.132 $WAKE_x Wake
The Wake system service activates a process that has placed itself in a state of hibernation with the Hibernate (HIBER) system service.

$WAKE_x [pidadr], [prcnam]

pidadr = the address of a longword containing the process identification of the process to be awakened.

prcnam = the address of a character string descriptor pointing to the process name string. The name is implicitly qualified by the group number of the process issuing the wake.

Return Status Codes

SS$_NORMAL: Service successfully completed.

SS$_ACCVIO: The specified process name string or string descriptor cannot be read, or the process identification cannot be written, by the caller.

SS$_IVLOGNAM: The specified process name string has a length of 0 or has more than 15 characters.

SS$_NONEXPR: Warning. The specified process does not exist, or an invalid process identification was specified.

SS$_NOPRIV: The calling process does not have the privilege to wake the specified process.

Privilege Restrictions
User privileges are required to wake:

- Other processes in the same group (GROUP privilege)
- Any other process in the system (WORLD privilege).

> **NOTE**
> If one or more Wake requests are issued for a process that is not currently hibernating, a subsequent Hibernate request will be completed immediately. That is, the process does not hibernate. No count is maintained of outstanding Wake requests.

10.133 $WFLAND_x Wait for Logical AND of Event Flags
$WFLAND_x is a VMS system service macro. It calls a VMS service that allows the program to specify a mask of event flags for which it wishes to wait. If all of the flags indicated by the mask are set, control returns immediately to the calling program. Otherwise, the program is placed in a wait state until the flags are all set.

$WFLAND_x efn, mask

<u>efn</u> = the number of any event flag within the cluster being used.

<u>mask</u> = the 32-bit mask in which bits set to 1 show the event flags of interest.

Return Status Codes
SS$_NORMAL: Service successfully completed.

SS$_ILLEFC: An illegal event flag number was specified.

SS$_UNASEFC: The process is not associated with the cluster containing the specified event flag.

NOTE
The wait state caused by this service
can be interrupted by an asynchronous
system trap (AST), if (1) the access
mode at which the AST executes is less
than or equal to the access mode from
which the wait was issued, and (2) the
process is enabled for ASTs at that
access mode.

When the AST service routine completes
execution, the system repeats the
WAITFR request. If the event flag has
been set, the process resumes execution.

10.134 $WFLOR_x Wait for Logical OR of Event Flags
$WFLOR_x is a VMS system service macro. It calls a service that
tests the event flags specified by a mask within a specified
cluster. The service returns control to the calling program
immediately if any of the flags is set. Otherwise, the service
places the program in a wait state until one of the selected event
flags is set.

$WFLOR_x efn, mask

efn = the number of any event flag within the cluster being
        used.

mask = a 32-bit mask in which bits set to 1 show the event flags
        of interest.

Return Status Codes
SS$_NORMAL: Service successfully completed.

SS$_ILLEFC: An illegal event flag number was specified.

SS$_UNASEFC: The process is not associated with the cluster
containing the specified event flag.

NOTE
The wait state caused by this service
can be interrupted by an asynchronous
system trap (AST), if (1) the access
mode at which the AST executes is less
than or equal to the access mode from
which the wait was issued, and (2) the
process is enabled for ASTs at that
access mode.

When the AST service routine completes
execution, the system repeats the
WAITFR request. If the event flag has
been set, the process resumes execution.

Good documentation is important in all diagnostic programs. A program without it is incomplete and of little use, since code by itself is generally obscure.

Programs require several levels of documentation. 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. Here are three reasons for documenting your program carefully.

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 program maintainer, whether or not the maintainer is the individual who developed the program. Since the documentation tells what the code is intended to do, the maintainer can fix the code if, in fact, the code does not perform as intended. Or, if the maintainer wishes to alter the function of the code, the documentation will help him to determine what he must change.

Document your program in each phase of its development. Do not leave it until the end, when the program strategy and details may be hard to recall.

11.1 DOCUMENTATION FILE

Make the documentation file the first item in the listing when you release a diagnostic program. Direct this file at the program user, and include in it all information necessary to running and using the program. The documentation file identifies the name and function of the program. In addition, it gives program operating instructions, run-time requirements, and a functional description of each test in the program.

Organize the documentation file under several headings. The following headings are standard:

    Documentation cover sheet
    Program abstract
    Hardware requirements
    Software requirements
    Prerequisites

Operating instructions
Program functional description

11.1.1   Documentation Cover Sheet
The documentation cover sheet is the first page of the documentation file. Use the following format:

A   PRODUCT CODE: Component part number assigned to the document. The format is ZZ-XXXXX-V.U-E. All VAX diagnostic programs start with ZZ. XXXXX uniquely identifies the program and is assigned by B.S.D.E. V.U-E indicates the version number (Paragraph 11.2.4).

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 Diagnostic Engineering, is sufficient.

E   DISCLAIMER: The disclaimer statement should appear as shown in Example 11-1.

F   COPYRIGHT STATEMENT: DIGITAL engineers use the format shown in Example 11-1, 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.

Additional information, such as AUTHOR, REVISED BY, or REPLACES, is optional. Example 11-1 is typical.

```
                         IDENTIFICATION

     A    PRODUCT CODE:       ZZ-ESDAA-2.1-7
     B    PRODUCT NAME:       ESDAA21 DZ11 8 LINE ASYNC MUX TEST
     C    PRODUCT DATE:       20 FEBRUARY 1979
     D    MAINTAINER:         BASE SYSTEMS DIAGNOSTIC ENGINEERING

     E    THIS SOFTWARE IS FURNISHED UNDER A LICENSE FOR USE ONLY
          ON A SINGLE COMPUTER SYSTEM AND MAY BE COPIED ONLY 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
          EXCEPT FOR USE ON SUCH SYSTEM AND TO ONE WHO AGREES TO
          THESE LICENSE TERMS. TITLE TO AND OWNERSHIP OF THE
          SOFTWARE SHALL AT ALL TIMES REMAIN IN DEC.
```

Example 11-1   Documentation Cover Sheet

11.1.2   Program Abstract

The program abstract is a short statement (from 3 to 20 lines) that describes the major functions and features of the program. The following categories are representative.

> Diagnostic program level (1, 2, 2R, 3, or 4)
> Device tested
> Program purpose and functions
> Error resolution: chip, module, or function callout
> Operator selectable program sections
> Unique program features
> Program transportability

Example 11-2 shows the abstract for the DZ11 diagnostic program, ESDAA.

ABSTRACT

This level 3 diagnostic program checks the functionality of from one to eight DZ11 units attached to any VAX-11 processor. The program provides error messages which identify failing modules and functions and aid in the repair of the device. The program uses the internal loopback mode on the DZ11 to check most of the circuitry on the device. In addition, the program provides two operator selectable sections which test optional hardware configurations. The H327 section checks the modem control feature of the M7819 (EIA) module. The H3190 section enables the operator to use the H3190 turn around connector for testing the M7814 (20 ma) module.

Example 11-2   Program Abstract

11.1.3   Hardware Requirements

Under this heading list the minimum hardware configuration necessary for program execution. List optional hardware and specific diagnostic hardware as appropriate, as shown in Example 11-3.

VAX Diagnostic Design Guide

```
┌─────────────────────────────────────────────────────────────┐
│                   HARDWARE REQUIREMENTS:                      │
│                                                               │
│ VAX-11 processor with minimum configuration                   │
│                                                               │
│ DW78Ø                                                         │
│                                                               │
│ DZ11, M7819, or M7814                                         │
│                                                               │
│                   OPTIONAL REQUIREMENTS:                      │
│                                                               │
│ H327 turn around connector for use with the M7819 module      │
│                                                               │
│ H319Ø turn around connector for use with the M7814 module     │
└─────────────────────────────────────────────────────────────┘
```

Example 11-3   Hardware Requirements Documentation

11.1.4   Software Requirements
Under this heading you should list the software environment or
environments in which the program will run, as shown in Example
11-4.

```
┌─────────────────────────────────────────────────────────────┐
│                   SOFTWARE REQUIREMENTS:                      │
│                                                               │
│                  VAX Diagnostic Supervisor                    │
└─────────────────────────────────────────────────────────────┘
```

Example 11-4   Software Requirements Documentation

11.1.5   Prerequisites
This category includes all requirements that must be satisfied
before the diagnostic program will yield valid results. For ex-
ample, a level 3 Unibus device diagnostic program will provide a
useful test of that device only if the operator has verified the
correct operation of the central processor and the Unibus channel
adapter. The information shown in Example 11-5 is brief but
sufficient.

```
┌─────────────────────────────────────────────────────────────┐
│                      PREREQUISITES:                           │
│                                                               │
│           Functional VAX-11 Central Processor and Memory      │
│                                                               │
│           Functional Unibus Channel Adapter                   │
└─────────────────────────────────────────────────────────────┘
```

Example 11-5   Prerequisities

11.1.6   Operating Instructions
Include in this category all instructions for loading and execut-
ing the diagnostic program. If the program can be loaded and run
on-line (under VMS) as well as off-line (standalone), show both
methods. Example 11-6 shows the operating instructions for the
disk reliability program.

11-4

```
^P                               ! Type Control P.
>>>BOOT SX0                      ! Load and start
                                 ! the diagnostic supervisor
                                 ! from the floppy
                                 ! disk drive.
DS> ATTACH DW780 SBI DW0 3 4
                                 ! Attach the UBA, link.
DS> ATTACH RK611 DW0 DMA 777440 210 5
                                 ! Attach the RK611.
DS> SELECT DMA                   ! Select the RK611.
DS> RUN ESRAA                    ! Run the disk
                                 ! reliability program.
```

NOTE
Operator input is underlined.

Example 11-6   Operating Instructions.

11.1.7 Program Functional Description

This section of the documentation file should include the following information categories.

```
    Program overview
        program purpose and strategy
        transportability
    Program size
        .EXE file size
    Program run timer
        quick verify
        default
        other modes
    Run-time dynamics
        e.g. memory allocation requirements
        other restrictions
    Event flags used
    Fault detection
        error resolution
        error message formats
        percentage of possible faults detected
    Performance during hardware failures
        unsuspected traps
        power failure
    Sequence of testing on multiple units
    Program applications
        field service
        manufacturing
        customers
        engineering
    How to set up and run with APT and APT-RD
    Program test (and subtest) descriptions
```

For each test (and subtest, if appropriate) explain the functions tested, possible failures, and actions that the operator should take on error detection. Example 11-7 shows the description of the first test in the RK611-RKØ6/RKØ7 drive functional diagnostic program (ESREF).

```
TEST - 1

TEST DESCRIPTION:

This test will read the RKCS1 register and then test the UBA
status to verify that an error did not result from the read. If an
error is detected, the diagnostic will be aborted. This test's
primary function is to verify that the RK611 registers can be
accessed, without detection of an error on the UBA, before further
testing.

TEST STEPS:

1. Issue adapter init.
2. Report error and abort if init failed.
3. Clear UBA status.
4. Test for stuck UBA status error - abort if error.
5. Read RKCS1.
6. Wait (stall execution) for approx. 7Ø microseconds.
7. Get UBA status.
8. Test for UBA error - abort if error.

DEBUG:

If this test fails, the user should run the appropriate RK611 or
UBA diagnostic.
```

Example 11-7   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.

11.2   MODULE PREFACE

The module preface is a documenting comment that should be placed at the beginning of the source file of a module. Each line in the preface (except for linker and assembler directives) should, therefore, begin with a comment delimiter, ";" or "!", in the leftmost character position. The module preface provides information for the program maintainer. It contains certain control items (without a comment delimiter) that the linker and assembler need, and it contains the standard DIGITAL copyright statement needed for the protection of DIGITAL'S legal ownership rights. While each module in a diagnostic program must have a preface, the preface for the first module is the most important.

It contains some information that describes the program as a whole, as well as information specific to the first module. Five sections make up a module preface. Build the module prefaces from the diagnostic program templates as shown in Example 6-1 in Chapter 6.

       Linker and Assembler Directives
       Copyright statement
       Environment statement
       History
       Module functional description

11.2.1   Linker and Assembler Directives

This section of the module preface contains four items as shown in Example 11-8. Refer to Paragraph 6.2.1 of Chapter 6 for more details.

```
.SBTTL   DZ11 8 LINE ASYNC MUX TEST
.TITLE   DZ11 8 LINE ASYNC MUX TEST
.IDENT   /V2.1-7/
.PSECT   HEADER, PAGE, NOWRT
```

Example 11-8   Linker and Assembler Directives
in the Module Preface

11.2.2   Copyright Statement

The copyright statement should be identical to the copyright statement furnished in the documentation file.

11.2.3   Environment Statement

The environment statement lists any special environmental assumptions that a module may make.

For level 2, 2R, and 3 diagnostic programs, for example, the run-time environment includes the diagnostic supervisor. A module in a level 2R diagnostic program also assumes that it runs only on-line (under VMS). Or the module might assume that asynchronous system traps (ASTs) are disabled. In general, you should document anything out of the ordinary that the module assumes about its environment.

The first module in a program should also explain how to assemble (or compile) and link the modules into an executable program, given the source modules.

Example 11-9 shows the commands required to assemble and link a sample diagnostic program written in the VAX-11 Macro assembly language.

```
$ MACRO/LIS TESTM1                          ! Assemble module 1 (header)
                                            ! creating an OBJ file
                                            ! and a LIS file.
$ MACRO/LIS TESTM2                          ! Assemble module 2.
  .
  .
  .
$ MACRO/LIS TESTMN                          ! Assemble the last module.

$  LINK/EXE:SAMPLE/CONTIGUOUS/MAP/FULL/SYSTEM:200  TESTM1,  TESTM2,
...TESTMN
                                            ! Link the assembled
                                            ! modules creating an
                                            ! EXE file called SAMPLE in
                                            ! contiguous memory locations
                                            ! starting at address 200.
```

Example 11-9   Assemble and Link Commands Shown
in the Environment Statement

11.2.4   Program and Module Version Numbers

The VAX-11 standard version number provides a unique
identification for all DIGITAL in-house software. Whenever you
make a change to a completed program, the version number of the
header module and of each affected module should reflect the
change.

The version number is a compound string constructed as follows:

<support><version>.<update>-<edit>

<support> is a single capital letter (or null) identifying the
support level of the program.

S   special customer version

T   field test version

V   released or frozen version

X   unsupported experimental version

Normally you can omit this letter from the module identification,
since the letter reflects the program as a whole.

• <version> refers to major changes. Increment this number
  if you change a function or capability of the program.
  <version> is a decimal leading zero-suppressed number. It
  starts with 0 and progresses one increment at a time.
  Never skip a number. Use 0 before the first release. 1
  designates the first release, and so on.

- <update> refers to minor changes. If present, it is a period followed by a single decimal digit. Never skip a digit. Null designates a major change, because <update> is cleared when <version> is changed. 1 designates the first update, and so on.

- <edit> identifies any alteration of the source code and is never reset. <edit> is a hyphen sign followed by a decimal maintenance number, starting with 1. Numbers may be skipped but may never be lower than a previous edit number.

There may be several edits in one release. For example, if three software problem reports (SPRs) are handled, the edit number should be increased by 3. The three changes should then be identified in the maintenance histories of the affected modules. Use these edit numbers in the module maintenance history.

NOTE
The version number given in the module preface is also the program identification number to be used in the .IDENT statement as shown in Example 11-8.

11.2.5 Module Maintenance History

When you modify an existing program module, assign an edit number to each problem addressed (each logical unit of modification). After a release you may bump the edit number to a round number, but this number should never be reset. Add a maintenance comment, derived from the edit number, to each line of source code that is affected. There are two good reasons for using edit comments.

1. The modifications may well be distributed all over the module. The maintenance comment enables you to find all the places where a correction of a single functional problem was made. This is especially useful if the correction has to be further corrected by someone other than the original modifier and/or if it has to be understood by the field service engineer.

2. All too often it happens that as we correct bug B, we innocently modify an instruction that was the correction for a previous bug, A. Bug B is fixed at the expense of the reappearance of bug A (or one of its relatives). If modification of a program leads you to the modification of a line that already has a maintenance comment, then find out (from the detailed current history) who the modifier was, consult that person, and exercise extreme caution in effecting your modification.

In many cases the edit numbers may be assigned consistently across all modules in a program. In this case, the module defining the program's version number should have a full maintenance history and the others should include only module specific changes.

Periodically, old, detailed, current history log entries may be deleted, together with their corresponding documenting comments (and lines marked for deletion). Do not make the deletion until the program has proven itself in the field.

Keep a history of the changes to each module in the module preface. the header module preface must reflect the history of the entire program. Example 11-10 shows how a program history might look as documented in the preface to the header module.

```
; V1.0       First release, 21-FEB-79, Ted Bear

; V1.1-2     First minor change, 22-MAR-79, Ted Bear

;            -1 fixed Unibus timeout in Test 5
;            -2 added 2 bytes to CHAR_BUF
;
; V2.0-3     Major change, 31-DEC-83, Jim Skunk
;            -3 added conversation mode
;
; V2.1-4     Minor change, 12-JAN-84, Jim Skunk
;            -4 changed Test 46 Subtest 5 Error print out from
;            "CSR" to "TCR"
```

Example 11-10   Module History

11.2.6   Module Functional Description

Group the routines and data structures that make up a diagnostic program in modules according to their functions. For example, the header module should contain global data, program/supervisor interface structures such as the header macro, and the initialization, cleanup, and summary routines. Global subroutines, if they are large or many, should make up one or more separate modules. The test routines should be grouped in modules according to common elements.

Write the module functional description so that it describes the common elements and the general purpose of the module. For example, if module 3 of a program contains tests that check the interface between the Massbus and a tape formatter, you should explain this fact in the module functional description.

11.3   ROUTINE PREFACE

Like the module preface, the routine preface is a documenting comment included in a module source file. Begin the routine preface with a begin sentinel, of the form ";++". Use an end sentinel, of the form ";--" as the last line. Begin each line in the text of the routine preface with a comment delimiter in the leftmost character position.

Write a routine preface for each routine in each module in a diag-
nostic program. For example, the initialization routine, the glob-
al subroutines, and the test routines should each be preceded with
a routine preface. Example 11-11 shows a sample routine preface.
Paragraphs 11.3.1 and 11.3.2 provide a detailed discussion of the
different elements that make up the routine preface. Notice that
each section of the routine preface should be labeled with a
keyword, whether or not it contains any text. If there is no text,
leave the keyword and write "; ** None **" on the line that
follows.

```
;++
; Functional Description:
; This routine issues a read header and data command to the drive
; whose logical unit number is passed in the calling sequence.
; Note that no byte count is specified; the number of blocks
; determines the byte count.  If the drive under test is an RPØX,
; then 528 bytes are transferred for each block number.  If the
; drive under test is an RKØ6, then 12 bytes per block are
; transferred.  If the drive under test is an RKØ6 then only the
; headers in each block are read.
; Calling Sequence:
;     PUSHL      NO_BLOCKS              : number of sectors to read
;     MOVW       CYLINDER,-(SP)         : cylinder number to use
;     MOVB       TRACK,-(SP)            : track value
;     MOVB       SECTORS,-(SP)          : sector value
;     PUSHL      LUN                    : logical unit number
;     PUSHL      BUFFER_ADR             : address which is to receive
;                                       : header and
;                                       : data information
;     CALLS      #4,READ_HEADER         : call routine
;
; Input Parameters:
; The global symbol QIOLIST is the base address of a block of QIO
; argument lists.  The LUN creates an implied association with one
; of those blocks.
;
; Implicit Inputs:
; ** None **
;
; Output Parameters:
; ** None **
;
; Implicit Outputs:
; ** None **
;
; Completion Codes:
; RØ=1 if no errors are detected.
; RØ=Ø if any errors are detected.
```

```
;
; Side Effects:
; ** None **
;
; Registers Used:
; RØ=byte count
; R3=logical block number
; R4=DSKDC address
; R8=sector
; R9=track
; R1Ø=cylinder
; Debug:
; If this routine fails, run the appropriate level 3 diagnostic
; program.
; --
```

Example 11-11   Sample Routine Preface

11.3.1   Routine Functional Description

The routine functional description should explain the purpose of the routine, necessary run-time conditions, and debug or test procedures.

Explain the purpose of the routine according to what it does, not how it does it. Describe the routine as if it were a large scale instruction. Make the explanation clear and logical, so that the casual reader can get a fairly accurate idea of what the routine accomplishes (Example 11-11).

If there are run-time conditions necessary to proper operation of the routine, they will often involve assumptions that you should state explicitly. Diagnostic programs should be designed to operate in a hostile environment. Therefore, the programmer must explain how the routine will behave if the required conditions are not met, and he must provide debug or test instructions in the routine preface. These instructions will tell the operator or program maintainer how to verify the correct behavior of the routine under a variety of possible circumstances.

For example, a test routine may require that a loopback cable be installed on the device under test. The test routine should check for the presence of this cable and abort if it is missing. This point should be explained in the routine preface, together with instructions for verifying whether or not the routine does abort when the cable is not connected.

Of course, the debug instructions in the routine preface should also explain how to verify correct operation of the routine when run-time conditions are normal.

11.3.2 Routine Interface

Global subroutines are generally called from within tests to perform generalized functions. Unlike the test routines, which are always called by the dispatch routine in the supervisor, the global subroutines can be called in different ways and for a variety of reasons. The routine preface for a global subroutine, therefore, should specify the following items:

> Calling sequence
> Input parameters
> Implicit inputs
> Output parameters
> Implicit outputs
> Completion codes
> Side effects
> Registers used.

11.3.2.1 Calling Sequence - If the routine follows the standard call procedure, then the routine can be called with CALLS or CALLG, as shown in Example 11-12.

```
; Calling sequence:

;        CALLS ENTRY_NAME (formal parameters)

;        or

;        CALLG ENTRY_NAME (formal parameter list name)
```

Example 11-12 Standard Calling Sequence

11.3.2.2 Input and Output Parameters - The input parameters that the routine expects may be items in a table (a list) or items on the stack. The input parameters statement should declare what the parameters are. The argument pointer should point to the longword preceding the first parameter. Example 11-13 is taken from a print routine preface.

```
; Input parameters:

; 4(AP)= address of ASCII string of the extended error message.

; 8(AP)= address of ASCII name of the register to be converted.

; 12(AP)= expected data.

; 16(AP)= received data.
```

Example 11-13 Input Parameters

If the called routine returns values, addresses, or strings to the calling routine, you should document these items as output parameters. For example, if the called routine returns a string in a buffer, you should state the address (label) and size of the

buffer. Then it should be clear that the calling routine should access the buffer after the call is completed. Notice that the general registers (R1-R11) may be used for output parameters, as shown in Example 11-14.

```
; Output parameters:

; R2= Binary parameter retrieved from operator, if successful.
```

Example 11-14   Output Parameters

11.3.2.3 Implicit Input and Output Parameters - In these sections of the routine preface you should include all locations in global or own (local) storage that are accessed by the routine. Implicit inputs are locations that the routine reads. Implicit outputs are locations that the routine writes. Be sure not to confuse these items with input and output parameters.

For example:

```
; Implicit inputs:

; DS$GA_PBASE:   P-table base address,
;                0=direct addressing
```

Example 11-15   Implicit Inputs

11.3.2.4 Completion Codes - Your routine preface should specify all of the completion codes that the routine may return in R0 to the calling program. Opposite each code indicate the code's meaning, as shown in Example 11-16.

```
; Completion codes:

;        R0=0 if no errors are detected.

;        R0=1 if any errors are detected.
```

Example 11-16   Completion Codes

11.3.2.5 Side Effects - Document here anything out of the ordinary that the routine does to its environment. For example, the routine may leave the hardware in a strange situation under some circumstances, jeopardizing the integrity of the operating system.

11.3.2.6 Register Usage - Under this heading, list the registers used by the routine and the function to which each is applied.

11.4   COMMENTS
Use comments to make your program understandable to any reader. The reader should be able to read the comments alone and get a good understanding of what the program does.

In a sense there are two programs to be written: one consisting of code, and one consisting of comments. Write the comment program to describe the intent and algorithm of the code. That is, comments are not simply rewordings of the code. They are explanations of the overall logical meaning of the code.

A comment is any text embedded between a comment delimiter on the left and the end of the source line on the right. In addition to the module and routine prefaces, your program should provide three types of comments: block comments, group comments, and line comments.

11.4.1   Block Comments

Use a block comment to introduce and describe the function and strategy of each logically distinct grouping of code. It should allow the reader to understand the code that follows without having to read the actual code. Notice that the code may be assembler directives as well as executable code. The following rules apply to block comments:

- The block comment is a paragraph consisting of a number of page wide comment lines. The comment delimiter (; or !) is entered, left aligned, in the line's first character position.

- The first line of the block comment is a begin sentinel, of the form ";+" or ";++".

- The last line of the comment block is a matching end sentinel of the form ";-" or ";--".

- The body of the block comment consists of text describing the block. Separate the text from the comment delimiter by a tab.

- Follow the block comment with a blank line. The code that the block comment describes should follow the blank line.

```
<skip>

; ++

; This is a block comment.

; --

<skip>

<CODE>
```

Example 11-17   Block Comment

11.4.2   Group Comments

Use a group comment whenever the attention of the reader should be called to a particular sequence of code as shown in Example 11-18.

```
   1.   When several paths join, note the conditions which cause
        the flow to reach this point.

        ; All exceptions converge at
        ; this point with:
        ; ...<register and stack status>

   2.   At the top of a loop.

        ;
        ; looking for a handler to call
        ;

   3.   When some data base has been built, such as a complex
        sequence on the stack.

        ;
        ; At this point the stack has
        ; the following format:
        ;     00(SP) = saved R2
        ;     04(SP) = number of bytes...
        ;     ...
        ;
```

Example 11-18   Group Comments

The following rules apply to group comments:

- The group comment consists of a number of page wide comment lines: the comment delimiter is entered left aligned, in the line's first character position.

- The first and last lines of the group comment are comment delimiters and are set off from surrounding code by a blank line before and after the group. Both the blank comment lines and the blank lines are mandatory and help the reader to distinguish the comments and code visually.

- The body of the group comment consists of descriptive text, separated from the comment delimiter by a space.

- Tabular information is separated from the comment delimiter by a tab.

11.4.3   Line Comments

Use line comments to explain the meaning and function of the lines being commented. For example, the instruction

MOVAL A,B

should be commented "; Initialize pointer to first buffer in free area." or such, not "; Move the address of A into B." As a rule of thumb, symbols should not appear in a comment. Instead, the comment should say what each symbol is or means.

Line comments can be connected so that they show the function of a section of several lines of code. When they are used in this way, the line comments help the reader to follow the flow of the program.

Several lines of comment may be attached to one line of code. Or one or more lines of comments may be attached to several successive lines of code, in which case you can indicate the connection by tagging follow-on lines with comments of the form ";
<space>...". The following rules apply to line comments:

- The comment is placed on the right-hand side of a statement.

- All assembly language comments are aligned with the comment delimiter in column 41 of the text (five tabs from left margin).

- If the statement would normally overflow into the comment field, then it can be broken and continued on the next line. Place the comment on the second line of the statement.

- If the comment is too long to be contained on a single line, or if the statement is too long to be commented on the same line, then the comment may be placed (or continued) on the following line. Place the comment delimiter in column 41, as before.

- Leave a space between the comment delimiter and the text of the comment.

Example 11-19 shows some uses of the line comment.

```
STATEMENT                           ; Compute multiple-line function
STATEMENT                           ; ...
STATEMENT                           ; ...

STATEMENT                           ; Here we do something new
                                    ; and extend the comment to the
                                    ; next two lines.

A SOMEWHAT LONG STATEMENT           ; And its comment
A SOMEWHAT LONGER STATEMENT         ; And its long comment
                                    ; which continues on
                                    ; additional line(s).
A VERY VERY VERY VERY VERY VERY VERY LONG STATEMENT
                                    ; And its comment on next line
A FRAGMENTED      -                 ;
        STATEMENT                   ; the statement's comment
```

Example 11-19   Line Comments

Diagnostic engineers should organize all diagnostic programs in a modular fashion. The main-line code in the test routines should proceed in functionally distinct steps. Many of the functions can be broken out from the main routine and coded as subroutines. Proper use of subroutines should reduce the size of the object code. And, if you make all subroutines global, they can be assembled in a separate module. They can then be made available to other modules within the program and to other programs. Asynchronous system trap (AST) routines, condition handlers, and interrupt service routines should also be global.

In addition, proper organization of the program into functionally distinct steps requires that error reports be made from the highest level (the main-line code). Error reports should not be made from subroutines, interrupt service routines, AST routines, or condition handlers. Two important reasons for restricting error reports to the highest level follow.

1. If all error reports come from the main-line code, the user will find the failing test, subtest, and error number in the code easily and directly from the error message. However, if a subroutine has detected an error and delivered the error message, the user may in fact never find the code that tests the failing function.

2. A routine that performs one distinct function only is more globally useful than a routine that performs several functions.

Subroutines should deal with errors as the system and supervisor services do, by returning appropriate status codes. The main-line code should check return status codes following all calls to the supervisor and VMS services and to routines within the program.

12.1 GUIDELINES FOR EXECUTABLE CODE

Arrange each segment of executable code so that the procedure flows from the top down. All branches, and jumps when they are necessary, should go down the page, except in the case of loops. Use upward branches to implement loops.

Each segment of executable code should have one entry point and one exit point, ideally. This simplifies program debugging. It also increases the diagnostic value of a program for the user, by making it easy for him to retrace the steps that have led to a failure.

12

Use the general registers for local storage, pointers, and base addresses, when possible, making sure that your comments are adequate. However, before you use a register, make certain that you are not destroying important data. Use a register save mask at the beginning of each routine to save each register that the code uses.

Make your code efficient, so long as efficiency does not destroy the readability of the program. For example, a BSBW instruction uses memory more economically than a JSB instruction, and word-PC-relative addressing is more efficient than longword-PC-relative addressing.

12.2   GLOBAL SUBROUTINE GUIDELINES

Each subroutine should function independently, like a high level instruction. Design each subroutine so that it is totally closed, making its interface to the main-line code clean and precise. Programmers should be aware of the trade-offs between branches, jumps, and calls. The call procedure is the standard interface between the main-line code and subroutines. However, if no parameters (or few parameters) are to be passed and the routine is short, branches may be more efficient than calls, because branches require less overhead. On the other hand, branches may be cumbersome if you use them with more than a few parameters.

When you use standard call procedures (CALLS or CALLG), data passed should consist entirely of received values, returned values, and returned status codes. Make the status codes as accurate as necessary to describe all possible conditions.

In most cases you should not mix functions within subroutines. For example, a subroutine that handles I/O to the device under test should handle only I/O. In all cases, be sure to avoid nesting print routines.

12.3   ERROR MESSAGE PRINT ROUTINES

Call error message print routines through the error message header macros (for example, $DS_ERRHARD_x) from the main-line code, as shown in Example 8-24. Coordination of the error message header macros, the print macros (e.g., $DS_PRINTB_x), and the formatted ASCII output (FAO) directives should enable you to print any information, fixed and variable, as appropriate for each detectable error. Figure 12-1 shows the steps involved.
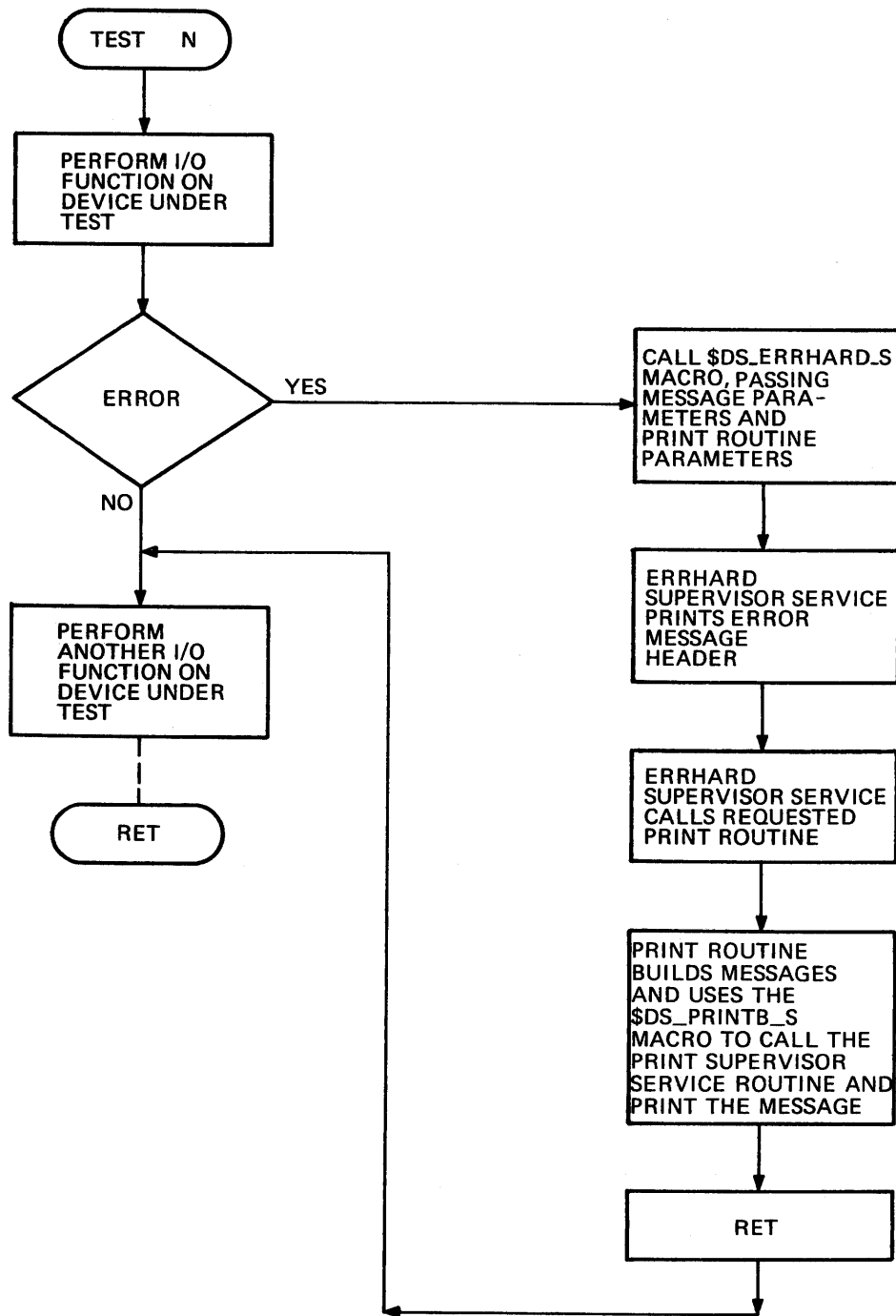
TK-3005

Figure 12-1   Printing an Error Message, Program Flow

12.4   HANDLING INTERRUPTS

Interrupt service routines enable level 3 diagnostic programs to field device interrupts directly, as they occur. Use interrupt service routines only toward the end of the program, after all hardware logic on the unit under test has been verified. Through interrupt service routines diagnostic programs can field interrupts, capture the status of the unit and the channel, if necessary, and return control to the main-line code. In some cases, the interrupt service routine may initiate other operations. Figure 12-2 shows the flow of control and interaction between main-line code (test n) and an interrupt service routine in a typical level 3 diagnostic program. The circled numbers in the paragraphs that follow are keyed to the flowchart.

The channel services macro ($DS_CHANNEL_x) is used in the test routine

(1)   To clear the channel.

(2)   Then to enable channel interrupts. In addition, the test routine clears the unit and enables device interrupts.

(3)   The test routine then sets up buffers for the I/O transfer and starts a watchdog timer. The timer should allow more than enough time for completion of the I/O transfer required (worst case).

(4)   Then the test code starts a transfer on the unit. When the unit finishes its transfer or detects an error, it sets an interrupt bit. The central processor fields the interrupt and calls the interrupt service routine

(5)   With the vector supplied to the channel service in step 1. The interrupt service routine analyzes the vector

(6)   To determine whether it can handle the interrupt. If it cannot, it prints an error message (note that an error message is permissible in the case of a fatal error) and terminates the program. If the interrupt service routine can handle the interrupt (the vector is valid), it takes appropriate action

(7)   Sets a done flag, and

(8)   Returns control to the main-line test code.

(9)   The test code is at this point in a loop checking the done flag and checking the timer. The program executes the loop until the unit has finished or the timer expires. If transfer has been completed, the code cancels the timer, and
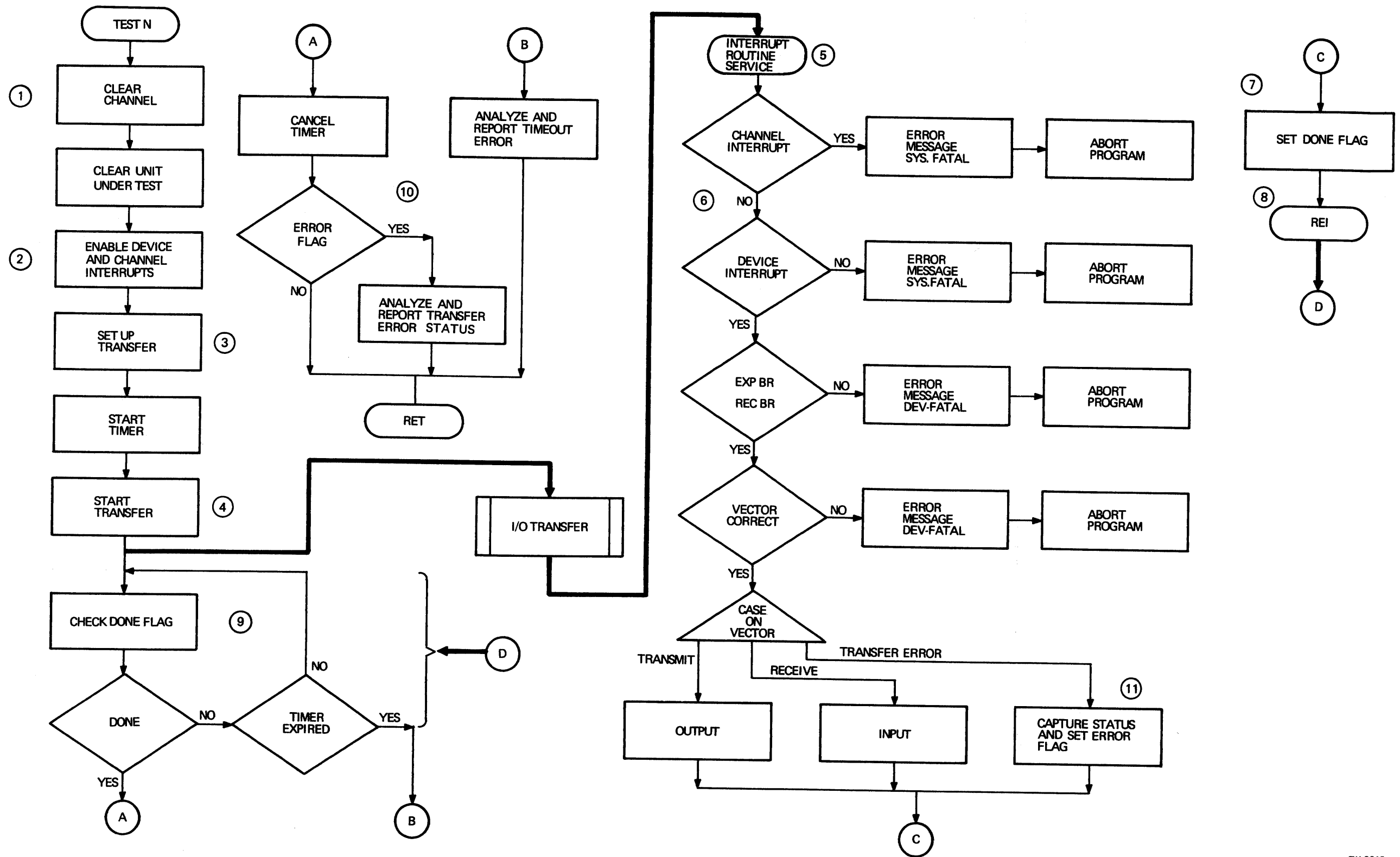
TK-3013

Figure 12-2
Handling Interrupts

(10) Checks the error flag. If the interrupt service routine has detected a transfer error, the test routine analyzes and reports the error. If the timer expires, one or more errors have occurred, and they must be analyzed and reported.

(11) Notice that the interrupt service routine captures the status of the device when it detects a transfer error vector. In this way error reporting is left to the main-line code.

12.5 ASYNCHRONOUS SYSTEM TRAPS

If a level 2 or 2R diagnostic program must test several devices in parallel, as in reliability testing, you should coordinate the various I/O transfers with asynchronous system trap routines (ASTs). The approach shown in Figure 12-3 requires three components of code

        a main-line test routine
        an I/O routine
        an AST routine.

The circled numbers in the following paragraphs are keyed to Figure 12-3.

(1) The main-line code (test n) calls the I/O routine once for each unit to be tested.

(2) The I/O routine reads the command buffer for a given unit. It then sets up the parameters for the queue I/O system service, queues the request, and returns control to the main-line code.

(3) The I/O transfers are started by the device driver code in VMS (or the supervisor) asynchronously.

(4) When queue I/O requests for all the units have been made, the main-line code hibernates.

(5) The driver routine calls the AST routine at the completion of each I/O transfer. For example, when the first transfer has been completed on the first unit, the AST routine identifies the device that has completed the transfer and checks for errors.

(6) If the AST routine finds that an error has occurred, it captures the status of the device and the diagnostic buffer in an error buffer for the failing unit. It then sets an error flag to signal the main-line code that an error has occurred. The AST routine does not report errors to the operator directly.

⑦   If the required number of transfers for the unit have not
    been completed,

⑧   The I/O routine is called again.

⑨   If all transfers for the given unit have been made, the
    AST sets a unit flag,

⑩   Wakes the main-line code, and returns. When the main-line
    code wakes up, it checks the error flag for each unit,
    and

⑪   Prints a message for each error on each failing unit. The
    test then checks to see whether all of the I/O transfers
    have been done on all units. .

⑫   If so, the test is finished. If not, the program
    hibernates until the AST routine wakes it, after all
    transfers on another unit have been completed.

Buffers and flags are necessary to support the AST as shown in
Figure 12-4.

AST routines are necessary in all diagnostic programs that
hibernate. A timer can be set to call an AST routine after the
expiration of a delta time or at a specified absolute time. The
AST routine can, in turn, wake the hibernating program, as it does
in Figure 12-3.

Notice that the AST mechanism, together with the Hibernate and
Wake system services, enables you to make level 2 and 2R
diagnostic programs highly efficient. When the program hibernates,
it frees the system resources for use by other processes.
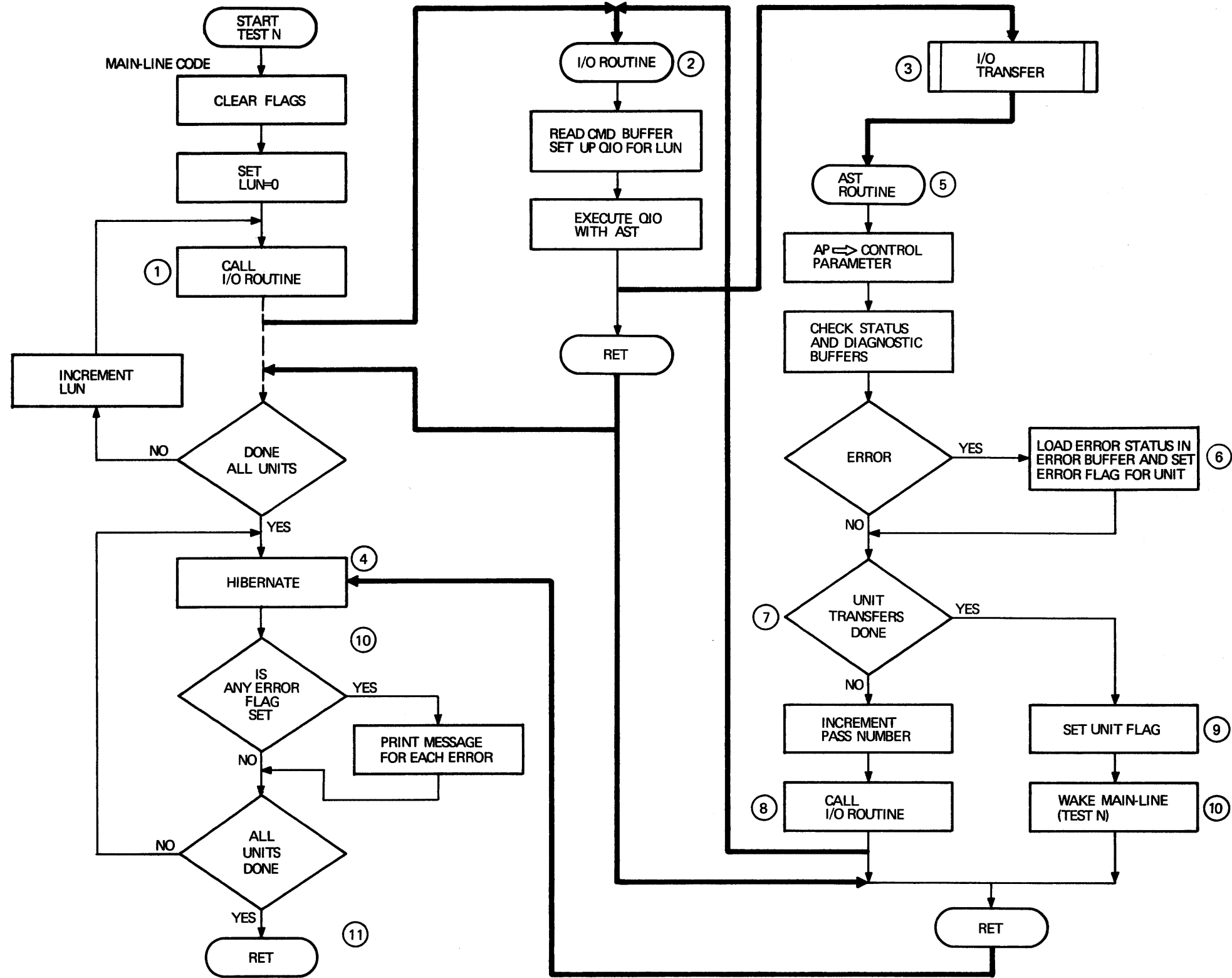
Figure 12-3 Coordination of
I/O Transfers with an AST
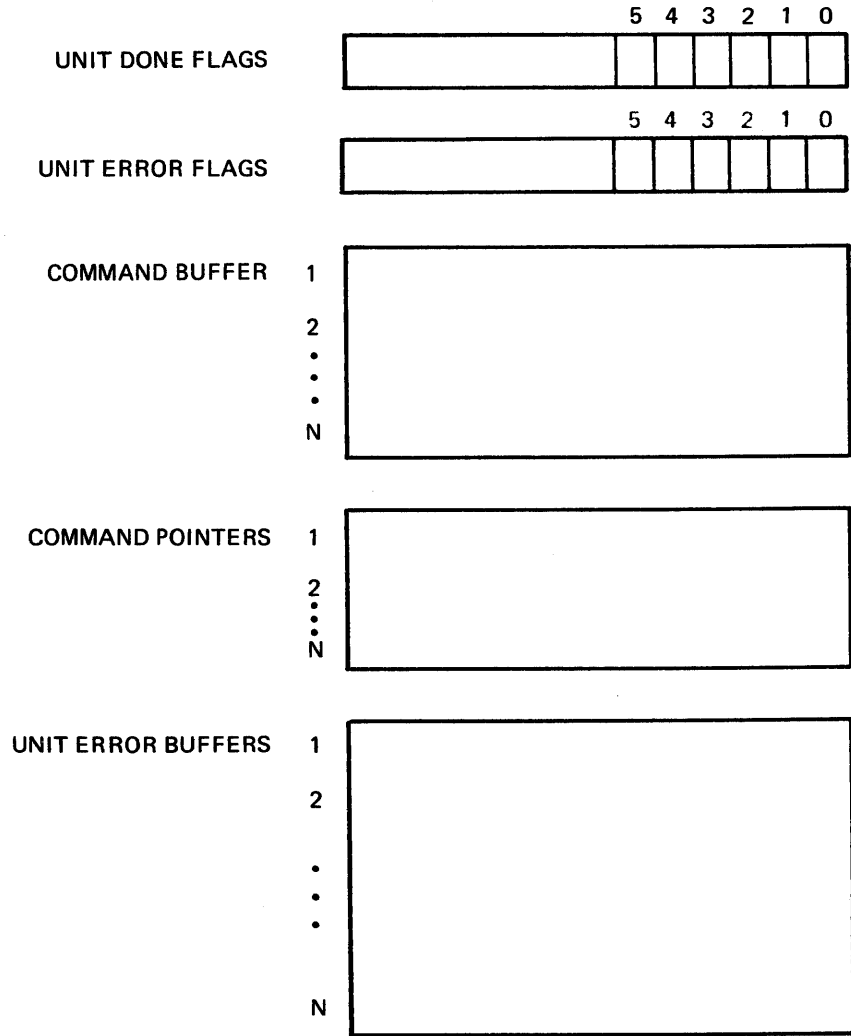
TK-3011

Figure 12-4 Data Structures that Support the AST

12.6 EXCEPTION HANDLERS AND CONDITION HANDLERS

Use of a handler routine increases the ability of a diagnostic program to deal with a hostile environment. It enables the program to detect, and in some cases recover from, exceptions (including machine checks) that would otherwise cause the program to abort. If your program may generate exceptions under some conditions, you should design a handler with those specific exceptions in mind. All conditions not handled by the program are reported by the supervisor.

There are two ways to deal with exceptions in the VAX diagnostic system: exception handlers and condition handlers.

First, level 3 programs that test processor specific functions should use the Set Vector supervisor service ($DS_SETVEC_x) to implement exception handlers. The Set Vector service loads the address of the exception handler into the system control block (SCB). When the exception condition occurs, the exception handler is entered directly. Use exception handlers primarily to test the system control block mechanism.

Level 2 and 2R diagnostic programs, and level 3 programs that are not processor specific, should use condition handlers to deal with exceptions. Those routines in a program which may generate exceptions should begin with instructions that declare the condition handler mechanism. You should move the address of the condition handler routine to the stack location that the frame pointer points to, as shown in Example 12-1.

```
MOVAL    COND_HANDLER, (FP)
```

Example 12-1   Declaration of a Condition Handler

This declaration enables the exception dispatcher routine in VMS or the supervisor to search the stack until it finds this address. The dispatcher routine then calls the condition handler. The condition handler is treated as a called procedure. As such it must end with an RET instruction.

Figure 12-5 is a flowchart showing the organization of a typical condition handler. It could be used in conjunction with the test routine shown in Figure 12-2. In that case, exceptions might be expected while the I/O transfers are in progress and the test code is looping at step 9.
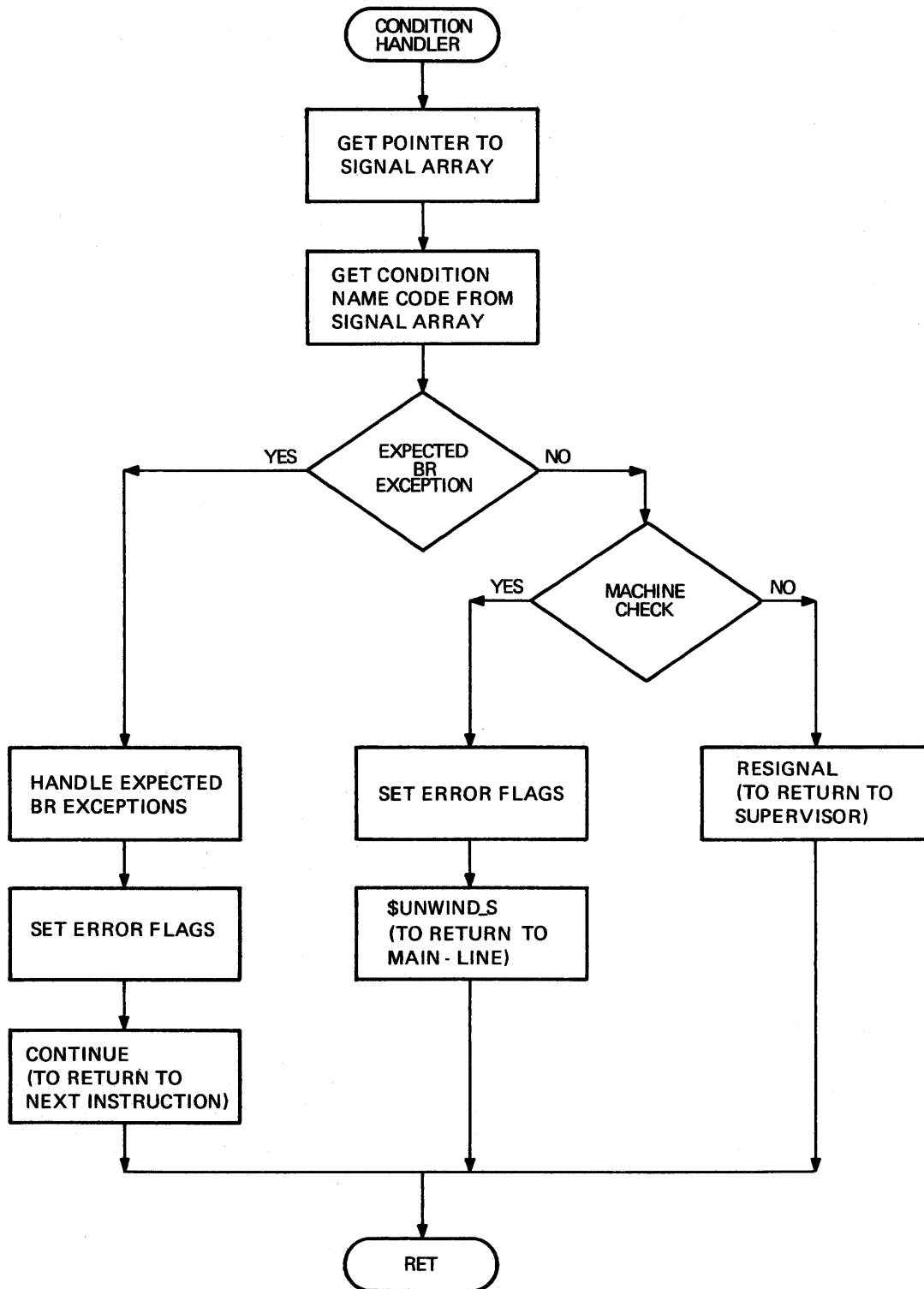
Figure 12-5   Condition Handler Flowchart

TK-3006

When the fault causing the exception occurs, an exception
dispatcher routine in VMS or the supervisor gains control. The
dispatcher examines the stack and vectors for the access mode in
which the violation occurred, in order to locate a condition
handler. The dispatcher follows the saved frame pointer (FP)
register images backward through the stack. It checks the first
longword in each frame to determine whether it is non-zero.

When the dispatcher locates the address of the condition handler
loaded previously, it constructs an argument list and calls the
handler. The argument list consists of two addresses that point to
longword arrays, as shown in Figure 12-6.
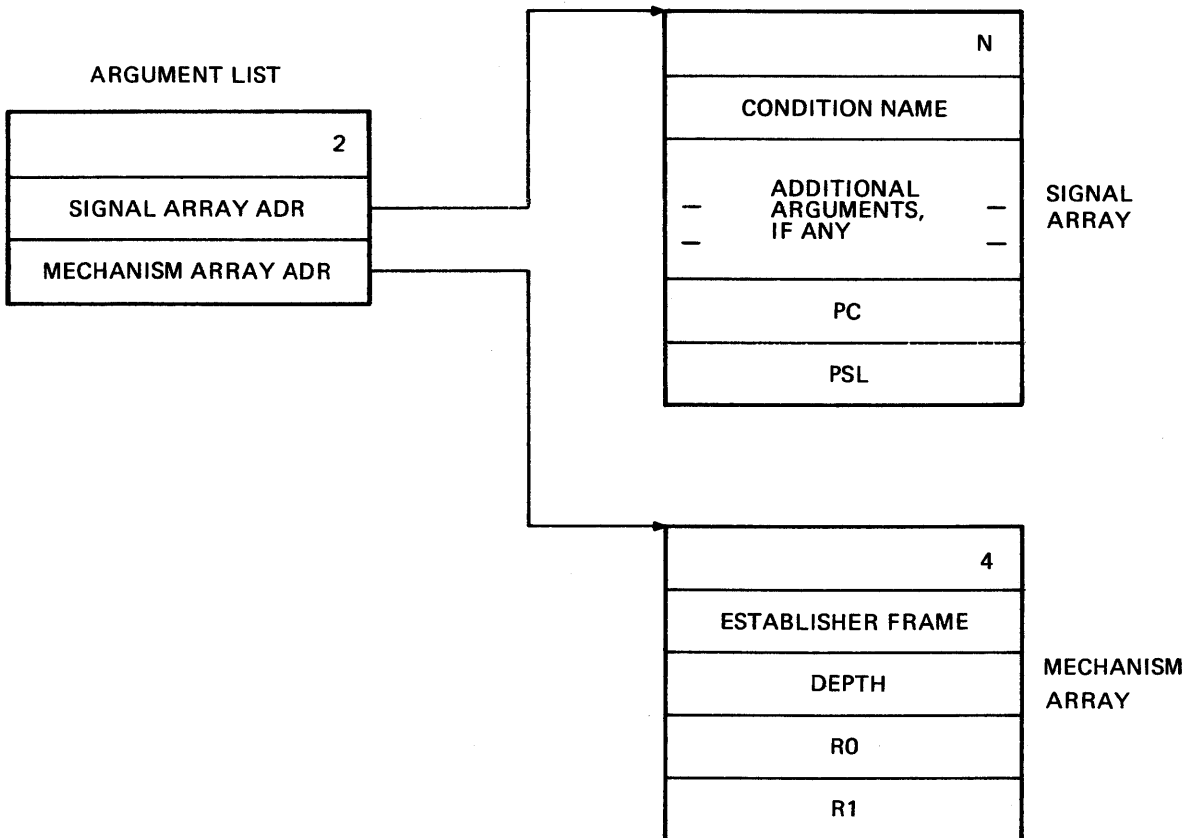
Figure 12-6 Condition Handler Argument List
and Associated Arrays

The first address in the argument list points to the signal array. The second points to an array containing the mechanism arguments. The first longword of each array shows the number of longword arguments in the array.

The condition handler routine should get the pointer to the signal array (Figure 12-5) and then find the condition name code. The handler can then check the condition code name and take appropriate action.

The condition handler can handle exceptions in any of three ways, as shown in Figure 12-5.

        Continue
        Unwind
        Resignal

If the condition handler can deal with the exception, it performs the required function, moves the continue code to R0, as shown in Example 12-2, and executes the RET instruction.

```
MOVZWL    #SS$_CONTINUE, R0
RET
```

Example 12-2 Continue from a Condition Handler

The continue code is a signal to the exception dispatcher to return control to the instruction that was executing when the exception occurred. Since the same exception will probably occur again unless the condition causing the exception has been fixed, care should be used with the continue function. For example, if the program is polling devices to determine what devices are on the system, or if the program is sizing memory, access violations will probably result, causing exceptions. The condition handler may have to change the address to be accessed next before returning control to the program.

When the handler cannot deal with a given condition, it can either resignal or unwind, depending on the condition.

If the program must abort, giving control to the supervisor, the handler should resignal, as shown in Example 12-3.

```
MOVZWL    #SS$_RESIGNAL, R0
RET
```

Example 12-3 Resignal and Return from a Condition Handler

If a program is executing a subroutine when the exception occurs, the exception may have occurred because of an error such as an AST routine failure. Such an exception, while not catastrophic, may require return from the subroutine to the main-line code before the problem can be corrected. If the subroutine has declared the

VAX Diagnostic Design Guide

condition handler, the handler should call the Unwind system
service ($UNWIND_x), generally without arguments. The Unwind
system service is available to level 2, 2R, and 3 diagnostic
programs. Refer to the VAX/VMS System Services Reference Manual
for details on UNWIND. However, the unwind operation is complex,
and should be used with care. The stack is then unwound to the
next higher level and control is returned to the return PC for
that procedure (the location to which control would have returned
following normal completion of the routine that caused the
exception), as shown in Figure 12-5. The handler should pass error
information to the test code via flags and status buffers. The
test code should, of course, check these flags and print error
messages as appropriate.

12.7   USER-DEFINED MACROS
Macros defined by the user (programmer) make an important
contribution to any diagnostic program. A program that uses
user-defined macros should be more readable and easier to code
than it would be otherwise. Most user-defined macros fall into two
categories:

        macros that build data structures
        macros that build executable code

12.7.1   Data Structure Macros
Most data structures are repetitive. A pattern developed for one
item is repeated continuously to form a table. If the table is
long, building it and making changes to it can be tedious.
Consider the table of addresses and labeled strings in Example
12-4.

```
            STRSET     <STAR,SUN,MOON,ROCK>
                       .LONG $$ T1          ; Insert count of strings.
                       .ADDRESS    T_STAR   ; address of string
                       .ADDRESS    T_SUN    ; address of string
                       .ADDRESS    T_MOON   ; address of string
                       .ADDRESS    T_ROCK   ; address of string
T_STAR:                .ASCIC /STAR/        ; Generate ASCIC string.
T_SUN:                 .ASCIC /SUN/         ; Generate ASCIC string.
T_MOON:                .ASCIC /MOON/        ; Generate ASCIC string.
T_ROCK:                .ASCIC /ROCK/        ; Generate ASCIC string.
```

Example 12-4 A Table of Addresses and Strings

The user-defined macro STRSET builds this table, given the
arguments STAR, SUN, MOON, and ROCK. Example 12-5 shows the macro
definition for STRSET.

```
        .MACRO     STRSET   STRINGS
                   $$T1=0                 ; counter
              .IRP          X,<STRINGS>
        (1)   $$T1=$$T1+1                 ; Count the number of
```

```
                                        ; strings.

                 .ENDR
                      .LONG    $$T1          ; Insert count of strings.
                 .IRP         X,<STRINGS>
         (2)     .ADDRESS     T_'X'          ; address of string
                 .ENDR
                 .IRP         X,<STRINGS>
T_'X:    (3)     .ASCIC       /X/            ; Generate ASCIC string.
                 .ENDR
         .ENDM
```

Example 12-5 A Data Structure Macro Definition

Three indefinite repeat macros (IRP) make up most of this macro
definition. After the counter $$T1 is initialized, the first
indefinite repeat macro (1) counts the number of strings to be set
up. The second indefinite repeat macro (2) sets up an address
pointer for each ASCIC string to be generated. It adds a "T_"
prefix to each string, making labels and pointers to the labels.
The third indefinite repeat macro (3) generates an ASCIC string
for each string given. The format of the table produced by this
macro may not be useful in every program. However, with macros
that you define for your program, you can produce data structures
with formats that are appropriate.

12.7.2 Macros that Build Executable Code

You can deal in two ways with functions that must be repeated
often: building macros or building subroutines. Consider the
trade-offs before choosing either. Macros require more memory
space than subroutines but execute more efficiently. Subroutines
save on memory space but require a greater overhead in execution
time. In general, you should use macros for repeated short
segments of code. Subroutines are useful for implementing longer
or more complex coding sequences. Be sure that user-defined
macros are expanded in the listing. Use the listing directives,
.LIST and .NLIST, to control the listing of the macro expansion,
as shown in Example 12-6 (refer to the VAX-11 Macro Language
Reference Manual for more details).

This example shows a user-defined macro that builds an argument
list from test data and calls a subroutine (CK_VERIFY) to check
for an error.

```
.MACRO  VERIFY DEVADR,FUNC,TEST,CMP
.LIST   MEB                          ; Expand macro.
        PUSHL CMP                    ; Save expected pattern.
        PUSHL TEST                   ; Save test pattern.
        PUSHL FUNC                   ; Save function.
        PUSHL DEVADR                 ; Save address of device
                                     ; register.

        CALLS #4, CK_VERIFY
.NLIST  MEB                          ; Disable macro expansion.
.ENDM
```
Example 12-6 A Macro that Generates Executable Code

12-15

12.7.3   Macro Libraries

Build a separate library file for user-defined macros for each diagnostic program.

In this way, the macros are available globally and available to other programs as well. Create a library from the file with a DCL command, as shown in Example 12-7.

```
$ LIB<name_of_library>/CREATE:::MAC/-SZ=<source-name>
```

Example 12-7 Creating a Library

List all of the required macro libraries (user-defined and VAX family libraries) in the include files section of each program module, as shown in Example 12-8.

```
        .SBTTL DECLARATIONS
;
;  Include Files:
;

.LIBRARY        \LB:DIAG.MLB\        ; VAX family diagnostic library
.LIBRARY        \ESDAA.MLB\          ; DZ unique macro library
```

Example 12-8   Include Files

12.8   SYMBOL NAMING CONVENTIONS

Two types of global symbols are used in the VAX diagnostic system: public symbols and private symbols.

12.8.1. Public Symbols

The diagnostic supervisor, the VMS operating system, and other DIGITAL software facilities use public symbols throughout. All DIGITAL public symbols contain a currency sign ($). Customers and diagnostic engineers are advised to use symbols without currency signs in order to avoid future conflicts.

12.8.2 Private Symbols

Diagnostic engineers should apply the following conventions to produce useful private symbols that convey as much information as possible about the entities they name.

1. A private symbol is an alphanumeric string of up to 15 characters in length. It consists of letters a through z and A through Z, digits 0 through 9, and the special characters underline (_), and dot (.).

    ● The assembler does not distinguish between uppercase and lowercase alphabetic characters constituting a symbol. Thus "symbol", "SYMBOL", "SyMbOl", etc., are all interpreted as equivalent.

To minimize reader confusion, never use lowercase in symbols. Lowercase should be used only in comments and in text strings.

- The underline character (_) is used to separate the parts of a compound (or qualified) name. Freely use the underline when constructing names to improve readability and comprehension.

- Make sure that your symbols are unique.

2. In general, follow the rules for DIGITAL public names, placing an underscore (_) where the currency sign ($) would go.

3. User-defined macros simply use a descriptive and unique name.

4. Global entry point names that have nonstandard calls are of the form:

   entryname_Rn

   where register R0 to Rn are not preserved. Note that the caller of such an entry point must include at least registers R2 through Rn in its own entry mask.

5. Global variable names are of the form:

   Gt_variablename

   The letter G stands for global variable and the t is a letter representing the type of the variable as defined in Paragraph 12.8.3.

6. Addressable global arrays use the letter A (instead of the letter G) and are of the form:

   At_arrayname

   The letter A stands for global array and t is one of the letters representing the type of the array element according to the list in Paragraph 12.8.3.

7. Structure offset names are of the form:

   structure_t_fieldname

   The t is a letter representing the data type of the field as defined in the next section. The value of the symbol is the byte offset to the start of the datum in the structure.

8. Structure bit field offset and single bit names are of the form:

    structure_V_fieldname

    The value of the symbol is the bit offset from the start of the containing field (not from the start of the control block).

9. Structure bit field size names are of the form:

    structure_S_fieldname

    The value of the symbol is the number of bits in the field.

10. Structure mask names are of the form:

    structure_M_fieldname

    The value of the symbol is a mask with bits set for each bit in the field. This mask is not right justified. Rather, it has structure_V_fieldname zero bits on the right.

11. Structure constant value names are of the form:

    structure_K_constantname

12.8.3 Object Data Types

Use the letters listed in Table 12-1 to represent the various data types and functions indicated.

Table 12-1   Object Data Types

Letter	Data Type or Usage
A	address
B	byte integer
C	single character
D	double-precision floating
E	reserved to DIGITAL
F	single-precision floating
G	general value
H	integer value for counters
I	reserved for integer extensions
J	reserved to customers for escape to other codes
K	constant
L	longword integer
M	field mask
N	numeric string (all byte forms)
O	reserved to DEC as an escape to other codes

Table 12-1   Object Data Types (Cont)

Letter	Data Type Or Usage
P	packed string
Q	quadword integer
R	reserved for records (structure)
S	field size
T	text (character) string
U	smallest unit of addressable storage
V	field position (assembler); field reference (BLISS)
W	word integer
X	context dependent (generic)
Y	context dependent (generic)
Z	unspecified or nonstandard

N, P, and T strings are typically variable length. Frequently in structures or I/O records they contain a byte-sized digit or character count preceding the string. If so, the location or offset points to the count. Counted strings cannot be passed in CALLs. Instead, a string descriptor is generated.

12.9   ASSEMBLY AND LINK PROCEDURES
Assemble each module in a program separately, creating an object file (.OBJ) and a listing file (.LIS). Then link the object files for all the modules together to produce an executable file (.EXE) and a map (.MAP). Use the CONTIGUOUS and SYSTEM switches with the Link command. Example 12-9 shows the assembly and linking of three modules.

```
$MACRO/LIS        HEADER
$MACRO/LIS        TESTA
$MACRO/LIS        TESTB
$LINK/EXE:SAMPLE/MAP/FULL/CONTIGUOUS/SYSTEM:200 HEADER,TESTA,TESTB
```

Example 12-9   Assembly and Link Commands

Diagnostic programs should be reliable, simple to set up, easy to run, and they should run under a variety of circumstances. A program that breaks down easily under automated product test (APT) or comes to a stop in a script file is limited in value. Eight considerations of this type are especially important:

> Program set up
> APT constraints
> Scripting constraints
> Run-time consideration
> Parallel versus serial testing
> Looping constraints
> Volume verification
> Long silences

13.1 PROGRAM SET UP

In general, diagnostic programs should not require special modifications to the hardware. However, on some devices abnormal jumper configurations or loop-back cables are necessary in order to enable the program to test specific logic or functions.

Keep set up requirements of this sort to a minimum. Place all tests that require special set up in a separate program section. In this way, the operator may run a quick verification check on the device without making changes to the hardware. Then, if he suspects a problem in the area left untested, he can make the special hardware set up and select the appropriate program section for execution.

13.2 AUTOMATED PRODUCT TEST (APT) CONSTRAINTS

All VAX diagnostic programs should be executable in the APT environment. APT imposes three constraints.

> First, the program must be able to yield control of the processor within 3 seconds at any time, following the typing of Control C by the operator. This means that any potentially long loops or operations must contain the $DS_BREAK macro at appropriate points.

> Second, the APT environment will not accept program image overlays, since it cannot open files. This means that if a program must load microcode into a microprogrammable device, the microcode must be built into the diagnostic program file at assembly time.

> Third, any tests that prompt the operator for a response must be placed in a separate, manual intervention section that will not be executed under APT. The program will fail if it executes an instruction that prompts the operator under APT.

13

13.3 SCRIPTING CONSTRAINTS

When diagnostic programs run under the control of a script file, no operator is present to answer questions or perform other tasks. Therefore, all functions that require operator action should be placed in a manual intervention section that is not executed unless explicitly selected. $DS_ASKxxx_x macros will normally solicit responses from the script, not the operator, if the program is running under a script (see Example 13-1 for the prompt message). However, if the programmer is using the $DS_ASKxxx_x macros to prompt for volume verification on a disk or tape diagnostic program, he should code the prompt string with a null as the first character. Example 13-2 shows the prompt message to be used with a $DS_ASKxxx_x macro to prompt the operator for a response, even when the program is running under a script file.

```
.ASCIC \ DO YOU WANT TO CONTINUE?\
```

Example 13-1   A Normal Prompt Message for the $DS_ASKxxx_x Macro

```
.ASCIC (0) \ DO YOU WANT TO CONTINUE?\
```

Example 13-2   A Special Prompt Message that Causes
Rejection of Scripted Responses

For further details on volume verification, refer to Paragraph 13.7.

13.4 RUN-TIME CONSIDERATIONS

In general, diagnostic programs should execute in as little time as possible. Programmers should be especially careful to optimize execution time on long programs. Three considerations are particularly important.

> First, where exhaustive tests are essential to the program design, make a quick run optional. For example, if a given test repeats a function with a large number of data patterns, choose basic patterns for the quick run and make the extra patterns optional.

> Second, make your error messages efficient. One large message tends to be more efficient, in terms of execution time and information conveyed, than several small messages. And you can improve the value of your messages if you dump only pertinent registers. Unnecessary information is worse than nothing.

> Third, make the scope loops for level 3 programs as short as possible. A long scope loop is generally hard to use.

13.5 PARALLEL VERSUS SERIAL TESTING

Most programs are designed to test several units of a particular device type.

Serial testing is useful for basic programs that test logic and provide scope loops. Therefore, all level 3 programs should test units serially. In serial testing, the initialization routine normally determines which unit is to be tested next. The entire program is run on the unit selected. Then, control returns to the initialization routine, and the next unit is selected. The program completes one pass only when it has tested all units. Alternatively, each test in a program may test all units serially before passing control to the next test.

Parallel testing is useful for level 2 and 2R programs that verify magnetic media and devices that perform direct memory access. For example, a program can initiate long transfers on several units and then hibernate. All of the units under test may perform the required functions simultaneously and independently without processor intervention. Parallel testing, therefore, is more efficient than serial testing for media diagnostic programs.

13.6 LOOPING CONSTRAINTS

Avoid making tight loops that do not include a $DS_BREAK macro or some supervisor service call that checks for Control C. If the program hangs in a loop that does not check for Control C, the operator will be unable to regain control of the system.

13.7 VOLUME VERIFICATION

Diagnostic programs that write on magnetic media (disk or tape) must use a fail-safe approach that prevents them from accidentally destroying the media of a customer. These programs should test the volume identification on the media under test for three possible cases:

    scratch ID
    unknown ID
    unrecognizable ID

If the program identifies the volume as "scratch" it should proceed. Otherwise, it should prompt the operator before continuing.

If the operator does not respond (time-out) or types a carriage return, the program should abort testing on the unit in question. The program should make no assumptions about the volume identification.

13.8 LONG SILENCES

Diagnostic programs with long execution times should avoid long silences. In order to assure the operator that the program is running properly, the program should type a message (a summary, for example) at least once every five minutes. You can set up an AST routine to perform this function.

If you fail to design and code your diagnostic program carefully, the debugging phase may require more effort than the development phase. However, even in the most carefully constructed program, a wide variety of errors may develop. Many of the coding errors will appear during initial program assembly, linking, and execution. Some types of design errors, however, may go undetected until you make a thorough quality assurance (QA) check.

The debug features of the supervisor provide you with the necessary tools at this point. In general, when you suspect that an error is connected with a specific area of the program, you can set a breakpoint to get there. Then proceed with Examine, Deposit, and Next commands to locate the problem and try solutions.

14.1   COMMON CODING ERRORS AND THEIR SYMPTOMS

Some coding errors are common to a wide variety of diagnostic programs. Others are unique to specific applications or methods. And while some programming errors are catastrophic and obvious, others may be difficult to identify or unpredictable. The list which follows explains some of the most common errors and their symptoms.

14.1.1   Endless Loops

Sometime after the program has started, the computer appears to sleep (silent death). It may even be unresponsive to the Control C command.

An endless loop will produce symptoms of this type. If the program checks a bit in a status register, for example, and then loops until the bit is set, an endless loop will result if the bit is never set.

You can avoid this kind of error by setting up a watchdog timer to force the program to branch out of the loop at the end of a given time (perhaps 5 seconds). The code that follows should report the condition to the operator.

14.1.2   Forgetting Initialization

If the program behaves in an erratic way, it may be because of an initialization or cleanup failure.

For example, when the program performs a sequence of tests on a device, test B may initialize conditions on the device that are necessary both to test B and to test C. Unless test C initializes the device as well, an error may result if the operator causes the program to loop on test C.

Make sure that each test within a program performs the initialization and cleanup functions necessary to enable it to stand alone.

14

14.1.3   Forgetting Return Status

If interrupts occur when they are unexpected, or the program indicates a hardware failure where none exists, the problem may be related to return status codes.

In a test that involves sequential steps, it may be that successful execution of step B depends on prior completion of step A. If step A involves a supervisor service or a call to a program subroutine, the code should check the return status before proceeding to step B. If the return status indicates that step A was not executed successfully, the program can report the error and take other appropriate steps. Consider a case in which step A disables interrupts. Step B may encounter unexpected interrupts and loose control to the supervisor if, in fact, interrupts have not been disabled.

14.1.4   Neglecting to Save Registers

Erratic program behavior may result from inadvertent destruction of the contents of a register (clobbering).

Make sure that your program saves the previous contents of the general registers when control passes from the main-line code to a subroutine. You should save the contents of all registers (RØ through R11) which the subroutine uses, whether or not you think that they all contain useful data. Failure to do this may introduce bugs that are very difficult to isolate.

Use of the register save mask procedure is the most efficient way to preserve the register contents.

14.1.5   Forgetting the Context or Properties of an Instruction

Your program may clobber a register or memory location, even though it makes no direct reference to that register or location, if you forget the context or properties of an instruction.

For example, a MOVC3 (Move Character) instruction destroys the contexts of RØ through R5.

14.1.6   Improper Context for I/O References

Portions of a program may be slow or unreliable if they make I/O references incorrectly. For example, an EXTV (Extract Field) instruction is too slow to operate efficiently on an I/O device register, and it may cause problems. In the same way, the BBS (Branch on Bit Set) instruction is unreliable in the I/O context. Instead, you should use the BIT (Bit Test) instruction and then branch conditionally.

In addition, be sure to use the correct data types for I/O references. Unibus device registers require word (16-bit) references. Massbus device registers, RH78Ø registers, and DW78Ø registers require longword (32-bit) references.

14.1.7   Forgetting the Number Sign (#)
Do not forget to put the number sign (#) in front of literals in argument lists for macro calls and literal operands in assembly language statements when appropriate or required.

14.1.8   Stack Underflow and Overflow
The stack may overflow or underflow, wiping out portions of the supervisor or memory buffers, if you use the push and pop instructions incorrectly when calling subroutines. The CALLS and CALLG instructions make the passing of parameters simple, and they keep the stack straight automatically. Use CALLS or CALLG, therefore, where there is a potential for stack problems.

14.2   QUALITY ASSURANCE PROCEDURES
The quality assurance process forms the last stage in the development of a diagnostic program. This stage is a vital part of the program development process, and it leads to formal diagnostic program release. Five distinct check procedures make up the standard quality assurance process.

  specification check
  conventions check
  fault detection and reporting check
  operational check
  user mode check

No diagnostic program is complete until these checks have been made.

14.2.1   Specification Check
After you have completed coding and documenting a diagnostic program, you must ensure that you have completely implemented the design and functional specifications (refer to Chapter 4 of this manual). Make this check with regard to test content and operational requirements, in a formal review. Include people from all concerned groups (e.g., engineering, field service, manufacturing, and training). Treat the review as the initial phase of training in use of the diagnostic program.

14.2.2 Conventions Check
The diagnostic program must follow the conventions set forth in this manual (refer to Chapters 6, 11, 12 and 13 in particular) and in the VAX-11 Software Engineering Manual. Positive answers to the six questions listed below are particularly important.

1.   Is the program documentation sufficient to convey understanding of each test without requiring the reader to analyze code?

2.   Do all bit and register mnemonics follow the names from the relevant hardware specification?

3.   Does the program correctly use features of the diagnostic supervisor (e.g. channel services)?

4.   Do the error formats agree with the standards set for header and basic error messages? See Example 8-23 in Chapter 8 of this manual.

5.   Are all error messages except system fatal messages reported from test or subtest bodies (main-line code)?

6.   Is error reporting from the cleanup code avoided?

14.2.3   Load and Execution Check

Make sure that the diagnostic program executes the load and sequence commands in the diagnostic supervisor without errors. Check the ability of the program to perform the following functions.

1.   An error free pass beginning with a normal load and start procedure.

2.   Multiple error free passes.

3.   A trace of the program with the Trace flag set.

4.   A loop on test check for each test, with no errors.

5.   Multiple loop on test.

6.   Infinite loop on test.

7.   Error free execution of all program sections not included in the default section.

14.2.4   Fault Detection and Reporting Check

Verify the fault detection and reporting capability of the diagnostic program. Introduce faults by setting breakpoints and inserting incorrect data. This will produce data comparison failures. Use this procedure to check each unique test case in the program for the following features.

1.   All error reports function properly.

2.   Loop-on-error and halt-on-error facilities function properly for each detectable error.

3. The four execution control flags that inhibit reports (IE1, IE2, IE3, IES) should function properly for each unique error report. Refer to Chapter 5, Paragraph 5.3.3. of this manual.

4. Error reports accurately reflect the faults that they report. You must verify module callout error reports with physical fault insertion.

14.2.5 Operational Checks

Make sure that the program functions properly under adverse conditions.

1. When the unit under test is powered off, the program gives a meaningful message to the operator, explaining the problem, and then returns control to the supervisor command line interpreter.

2. For devices capable of write protection which are write protected, the program reports the problem with a meaningful message (without bad side effects). The program then returns control to the supervisor command line interpreter.

3. For devices capable of being placed off-line, when the unit under test is placed off-line, the program types out an appropriate message and returns control to the supervisor command line interpreter.

4. The program runs under APT-VAX control.

5. The program runs in the minimum system configuration stated in the functional specification.

6. The program runs in the maximum system configuration stated in the functional specification.

7. The program runs with each appropriate module extended on a module extension board. (Perform this check one module at a time, making one complete pass per module.)

8. If the program is transportable, it satisfies the requirements of Paragraphs 14.2.1 through 14.2.4 on all VAX Family computer types.

14.2.6 User Mode Checks

Make sure that level 2 and level 2R diagnostic programs run in the user mode with the latest version of VMS. The program should run multiple passes without encountering software problems. When program execution is aborted, the unit under test should be left in the state in which the program found it.

14.3   DEBUG AND UTILITY COMMANDS IN THE DIAGNOSTIC SUPERVISOR
This group of commands provides the operator with the ability to isolate errors and to alter diagnostic program code. The supervisor allows up to 15 simultaneous breakpoints within the program. The operator can also examine and/or modify the program image in memory.

14.3.1   Set Base Command
Syntax: SET BASE <address><CR>

This command loads the address specified into a software register. This number is then used as a base to which the address specified in the Set Breakpoint, Clear Breakpoint, Examine, and Deposit commands is added. The Set Base command is useful when referencing code in the diagnostic program listings. The base should be set to the base address (see the program link map) of the program section referenced. Then the PC numbers provided in the listings can be used directly in referencing locations in the program sections.

For example:

```
DS> SET BASE E00             ! Set the base
                             ! address to the
                             ! beginning of the psect of
                             ! the routine under
                             ! examination.
DS>
```

Example 14-1   Set Base Command

NOTE
Virtual address = physical address (normally) when memory management is turned off.

See Example 14-7 for further clarification.

14.3.2   Set Breakpoint Command
Syntax: SET BREAKPOINT <address><CR>

This command causes control to pass to the supervisor when program execution encounters the <address> previously specified by this command. A maximum of 15 simultaneous breakpoints can be set within the diagnostic program.

For example:

```
DS> SET BREAKPOINT    30        ! Set a breakpoint
                                 ! at an offset of
                                 ! 30 from the
                                 ! base address.
```

Example 14-2 Set Breakpoint Command

14.3.3   Clear Breakpoint Command
Syntax: CLEAR BREAKPOINT <address> ! ALL<CR>

This command clears the previously set breakpoint at the memory
location specified by <address>. If no breakpoint existed at the
specified address, no error message is given. An optional
argument of ALL clears all previously defined breakpoints.

For example:

```
DS> CLEAR BREAKPOINT    30      ! Clear the breakpoint
                                 ! at the location which
                                 ! is offset 30 from
                                 ! the base address.
DS>
```

Example 14-3 Clear Breakpoint Command

14.3.4   Show Breakpoints Command
Syntax: SHOW BREAKPOINTS<CR>

This command displays all currently defined breakpoints.

For example:

```
DS> SHOW BREAKPOINTS            ! Display breakpoints
                                 ! currently set.
CURRENT BREAKPOINTS:
      00000E30 (x)
DS>
```

Example 14-4 Show Breakpoints Command

14.3.5   Set Default Command
Syntax: SET DEFAULT <argument-list><CR>

This command causes setting of default qualifiers for the Examine
and Deposit commands. The <argument-list> argument consists of
data length default and/or radix default qualifiers. If both
qualifiers are present, they are separated by a comma. If only
one default qualifier is specified, the other one is not affected.
Initial defaults are HEX and LONG. Default qualifiers are:

```
         Data Length:  Byte, Word, Long
         Radix:  Hexadecimal, Decimal, Octal
```

For example:

```
DS> SET DEFAULT BYTE, DECIMAL      ! Set the default data
                                   ! length qualifier as
                                   ! byte and the default
                                   ! radix qualifier as
                                   ! decimal.
DS>
```

Example 14-5 Set Default Command

14.3.6   Examine Command

Syntax: EXAMINE [<qualifiers>] [<address>] <CR>

The Examine command displays the contents of memory in the format described by the qualifiers. If no qualifiers are specified, the default qualifiers set by a previous set Default command are used. The applicable qualifiers are described in Table 14-1.

Table 14-1   Examine Command Qualifier Descriptions

Qualifier	Description
/B	Address points to a byte
/W	Address points to a word
/L	Address points to a longword
/H	Display in hexadecimal radix
/D	Display in decimal radix
/O	Display in octal radix
/A	Display in ASCII bytes

When specified, the <address> argument is accepted in hexadecimal format unless some other radix has been set with the set default command. Optionally, <address> may be specified in decimal, octal, or hexadecimal, by immediately preceding the address argument with %D, %O, or %X, respectively. <Address> may also be one of the following: R0-R11, AP, FP, SP, PC, PSL.

For example:

```
DS> EXAMINE 30                     ! Display the contents
                                   ! of the longword which
                                   ! is offset 30 from
                                   ! the base address.
00000E30:   D0513D01
DS>
```

Example 14-6 Examine Command

14.3.7   Deposit Command
Syntax: DEPOSIT [<qualifiers>] <address> <data><CR>

This command accepts data and writes it into the memory location specified by <address> in the format described by the qualifiers. If no qualifiers are specified, the default qualifiers are used. The applicable qualifiers are identical to those of the Examine command and described in Table 14-1.

The <address> argument is accepted in hexadecimal format unless some other radix has been set with the Set Default command. Optionally, <address> may be specified as decimal, octal, or hexadecimal by immediately preceding <address> with %D, %O, or %X, respectively.

For example:

```
DS> DEPOSIT/W/H 30 0001        ! Deposit 0001 (hex)
                               ! in the word
                               ! offset 30 from
                               ! the base address.
00000E30:    0001
DS>
```

Example 14-7 Deposit Command

See Example 14-1, preceding.

14.3.8   Next Command
Syntax: NEXT [number-of-instructions]<CR>

This command causes the supervisor to execute one machine language instruction. If you specify a number (decimal) after NEXT, the supervisor will execute that number of machine language instructions. The supervisor displays the PC of the next instruction and the contents of the next four bytes, after execution of each instruction.

Use this command to step through an area of a program where you suspect a problem. Do not use the Next command unless you have stopped the program at a breakpoint.

For example:

```
DS> NEXT    ! Execute the next instruction.
00000E31:    D0513D01
DS>
```

Example 14-8 Next Command

A SAMPLE DIAGNOSTIC PROGRAM

A

```
VAX/VMS  STAPLES      EVPRG      8-AUG-1979 16:15    LPA0:  8-AUG-1979 16:15    DB0:[STAPLES]EVPRG.MAP;3   VAX/VMS
VAX/VMS  STAPLES      EVPRG      8-AUG-1979 16:15    LPA0:  8-AUG-1979 16:15    DB0:[STAPLES]EVPRG.MAP;3   VAX/VMS
VAX/VMS  STAPLES      EVPRG      8-AUG-1979 16:15    LPA0:  8-AUG-1979 16:15    DB0:[STAPLES]EVPRG.MAP;3   VAX/VMS


VAX/VMS  STAPLES      EVPRG      8-AUG-1979 16:15    LPA0:  8-AUG-1979 16:15    DB0:[STAPLES]EVPRG.MAP;3   VAX/VMS
VAX/VMS  STAPLES      EVPRG      8-AUG-1979 16:15    LPA0:  8-AUG-1979 16:15    DB0:[STAPLES]EVPRG.MAP;3   VAX/VMS
VAX/VMS  STAPLES      EVPRG      8-AUG-1979 16:15    LPA0:  8-AUG-1979 16:15    DB0:[STAPLES]EVPRG.MAP;3   VAX/VMS
```

```
EEEEEEEEEE  VV      VV  PPPPPPP   RRRRRRR       GGGGGGGG
EEEEEEEEEE  VV      VV  PPPPPPP   RRRRRRR       GGGGGGGG
EE          VV      VV  PP    PP  RR      RR  GG
EE          VV      VV  PP    PP  RR      RR  GG
EE          VV      VV  PP    PP  RR      RR  GG
EE          VV      VV  PP    PP  RR      RR  GG
EEEEEEE     VV      VV  PPPPPPP   RRRRRRR     GG
EEEEEEE     VV      VV  PPPPPPP   RRRRRRR     GG
EE          VV      VV  PP        RR  RR      GG  GGGGGG
EE          VV      VV  PP        RR  RR      GG  GGGGGG
EE             VV  VV   PP        RR    RR    GG      GG    ....
EE             VV  VV   PP        RR    RR    GG      GG    ....
EEEEEEEEEE      VV      PP        RR      RR    GGGGGG      ....
EEEEEEEEEE      VV      PP        RR      RR    GGGGGG


MM      MM  AAAAAA    PPPPPPP    ;;;;   333333
MM      MM  AAAAAA    PPPPPPP    ;;;;   333333
MMMM  MMMM  AA    AA  PP    PP   ;;;;  33       33
MMMM  MMMM  AA    AA  PP    PP   ;;;;  33       33
MM  MM  MM  AA    AA  PP    PP                  33
MM  MM  MM  AA    AA  PP    PP                  33
MM      MM  AA    AA  PPPPPPP    ;;;;        33
MM      MM  AA    AA  PPPPPPP    ;;;;        33
MM      MM  AAAAAAAAAA  PP       ;;;;           33
MM      MM  AAAAAAAAAA  PP       ;;;;           33
MM      MM  AA    AA  PP          ;;   33       33
MM      MM  AA    AA  PP          ;;   33       33
MM      MM  AA    AA  PP          ;;   333333
MM      MM  AA    AA  PP          ;;   333333
```

```
VAX/VMS  STAPLES      EVPRG      8-AUG-1979 16:15    LPA0:  8-AUG-1979 16:15    DB0:[STAPLES]EVPRG.MAP;3   VAX/VMS
VAX/VMS  STAPLES      EVPRG      8-AUG-1979 16:15    LPA0:  8-AUG-1979 16:15    DB0:[STAPLES]EVPRG.MAP;3   VAX/VMS
VAX/VMS  STAPLES      EVPRG      8-AUG-1979 16:15    LPA0:  8-AUG-1979 16:15    DB0:[STAPLES]EVPRG.MAP;3   VAX/VMS
```

```
EVPRG                           8-AUG-1979 16:13        LINKER X01.20                    PAGE  1

                   +--------------------------------+
                   | OBJECT MODULE SYNOPSIS |
                   +--------------------------------+

MODULE NAME  IDENT  BYTES  FILE                        CREATION DATE       CREATOR
-----------  -----  -----  ----                        -------------       -------
EVPRG        V1.0   1410   DB0:[STAPLES]EVPRG.OBJ;3     8-AUG-1979 16:09    VAX-11 Macro V02.30
EVPRG1       V1.0    467   DB0:[STAPLES]EVPRG1.OBJ;3    8-AUG-1979 16:12    VAX-11 Macro V02.30
```

```
DB0:[STAPLES]EVPRG.EXE;3                8-AUG-1979 16:13        LINKER X01.20                    PAGE  2

                              +----------------------------+
                              | IMAGE SECTION SYNOPSIS     |
                              +----------------------------+

CLUSTER         TYPE PAGES   BASE ADDR DISK VBN PFC PROTECTION AND PAGING   GBL. SEC. NAME    MATCH   MAJORID  MINORID
-------         ---- -----   --------- -------- --- --------------------    --------------    -----   -------  -------

DEFAULT_CLUSTER  0    4      00009280     1      0  READ ONLY
```

8-AUG-1979 16:13   LINKER X01.20

DB0:[STAPLES]EVPRG.EXE;3

```
+------------------------+
| PROGRAM SECTION SYNOPSIS |
+------------------------+
```

P-SECT NAME	MODULE(S)	BASE	END	LENGTH	ALIGN	ATTRIBUTES
$HEADER	EVPRG	00000200 00000200	0000026A 0000026A	0000006B (107.)	PAGE 0 PAGE 0	NOPIC,USR,CON,REL,LCL,NOSHR,NOEXE, RD,NOWRT
$TSTCNT	EVPRG EVPRG1	0000026C 0000026C 0000026C	0000026F 0000026C 0000026F	00000004 (4.) 00000000 (0.) 00000004 (4.)	LONG 2 LONG 2 LONG 2	NOPIC,USR,OVR,REL,LCL,NOSHR,NOEXE, RD,NOWRT
. BLANK .	EVPRG EVPRG1	00000270 00000270 00000270	00000270 00000270 00000270	00000000 (0.) 00000000 (0.) 00000000 (0.)	BYTE 0 BYTE 0 BYTE 0	NOPIC,USR,CON,REL,LCL,NOSHR, EXE, RD, WRT
ARGLIST	EVPRG1	00000270 00000270	00000293 00000293	00000024 (36.) 00000024 (36.)	LONG 2 LONG 2	NOPIC,USR,CON,REL,LCL,NOSHR, EXE, RD,NOWRT
CLEANUP	EVPRG	00000294 00000294	000002AA 000002AA	00000017 (23.) 00000017 (23.)	LONG 2 LONG 2	NOPIC,USR,CON,REL,LCL,NOSHR, EXE, RD, WRT
DISPATCH	EVPRG EVPRG1	000002AC 000002AC 000002AC	000002C3 000002AC 000002C3	00000018 (24.) 00000000 (0.) 00000018 (24.)	LONG 2 LONG 2 LONG 2	NOPIC,USR,CON,REL,LCL,NOSHR,NOEXE, RD,NOWRT
DISPATCH_X	EVPRG	000002C4 000002C4	000002DB 000002DB	00000018 (24.) 00000018 (24.)	LONG 2 LONG 2	NOPIC,USR,CON,REL,LCL,NOSHR,NOEXE, RD,NOWRT
GBL_DATA	EVPRG	000002DC 000002DC	000003AF 000003AF	000000D4 (212.) 000000D4 (212.)	LONG 2 LONG 2	NOPIC,USR,CON,REL,LCL,NOSHR,NOEXE, RD, WRT
GBL_TEXT	EVPRG	00000380 00000380	00000694 00000694	000002E5 (741.) 000002E5 (741.)	LONG 2 LONG 2	NOPIC,USR,CON,REL,LCL,NOSHR,NOEXE, RD,NOWRT
INITIALIZE	EVPRG	00000698 00000698	000006E0 000006E0	00000049 (73.) 00000049 (73.)	LONG 2 LONG 2	NOPIC,USR,CON,REL,LCL,NOSHR, EXE, RD, WRT
SUBROUTINE	EVPRG	000006E4 000006E4	000007B2 000007B2	000000CF (207.) 000000CF (207.)	LONG 2 LONG 2	NOPIC,USR,CON,REL,LCL,NOSHR, EXE, RD,NOWRT
SUMMARY	EVPRG	000007B4 000007B4	000007CA 000007CA	00000017 (23.) 00000017 (23.)	LONG 2 LONG 2	NOPIC,USR,CON,REL,LCL,NOSHR, EXE, RD, WRT
TEST_001	EVPRG1	00000800 00000800	00000992 00000992	00000193 (403.) 00000193 (403.)	PAGE 0 PAGE 0	NOPIC,USR,CON,REL,LCL,NOSHR, EXE, RD,NOWRT
.LAST	EVPRG	00000A00 00000A00	00000A00 00000A00	00000000 (0.) 00000000 (0.)	PAGE 0 PAGE 0	NOPIC,USR,CON,REL,LCL,NOSHR, EXE, RD, WRT

DB0:[STAPLES]EVPRG.EXE;3   8-AUG-1979 16:13

```
                                    +=====================+
                                    | SYMBOL CROSS REFERENCE |
                                    +=====================+

SYMBOL          VALUE         DEFINED BY        REFERENCED BY ...
======          =====         =========         =============
$ENV            00000003      EVPRG
$MO             00000001      EVPRG
ACT_ENTRY       000006E4-R    EVPRG             EVPRG1
DEFAULT         00000000      EVPRG
GD_A_CSR        000002E8-R    EVPRG
GD_A_PTBL       000002E0-R    EVPRG
GD_A_XOR        000002EC-R    EVPRG
GD_K_CMDSIZ     00000008      EVPRG
GD_K_STRSIZ     00000004      EVPRG
GD_L_LUN        000002DC-R    EVPRG             EVPRG1
GD_L_TOKEN      000002F0-R    EVPRG             EVPRG1
GD_T_CMDBUF     000002F4-R    EVPRG             EVPRG1
GD_T_COMMAND    00000380-R    EVPRG             EVPRG1
GD_T_STRBUF     000002FC-R    EVPRG             EVPRG1
GD_W_CSR        000002E4-R    EVPRG             EVPRG1
GD_W_XOR        000002E6-R    EVPRG             EVPRG1
GT_T_AMBCMD     0000055A-R    EVPRG             EVPRG1
GT_T_CSRERR     000005CA-R    EVPRG
GT_T_CSRFUNC    0000051D-R    EVPRG
GT_T_CVT16      000004C8-R    EVPRG
GT_T_CVT32      00000415-R    EVPRG
GT_T_CVTCSR     000003F1-R    EVPRG
GT_T_FMTCR      00000622-R    EVPRG
GT_T_FMTEXP     000005DA-R    EVPRG
GT_T_FMTRCV     000005EF-R    EVPRG
GT_T_FMTXOR     00000604-R    EVPRG
GT_T_GETLINE    00000546-R    EVPRG             EVPRG1
GT_T_INVCMD     00000567-R    EVPRG             EVPRG1
GT_T_NOERR      000005B1-R    EVPRG             EVPRG1
GT_T_REGERR     0000059B-R    EVPRG             EVPRG1
GT_T_TIS2       0000057C-R    EVPRG             EVPRG1
G_K_INIT        00000004      EVPRG
G_K_NOP         00000000      EVPRG
G_K_STOP        00000003      EVPRG             EVPRG1
G_K_SUB1        00000001      EVPRG             EVPRG1
G_K_SUB2        00000002      EVPRG             EVPRG1
MSG_REGERR      00000727-R    EVPRG             EVPRG1
T1_S1           00000895-R    EVPRG1
T1_S2           00000926-R    EVPRG1
T1_S3           00000960-R    EVPRG1
TEST_001        00000818-R    EVPRG1
TEST_001_X      0000098B-R    EVPRG1
```

LINKER X01.20

DB0:[STAPLES]EVPRG.EXE;3   8-AUG-1979 16:13

```
+===================+
| SYMBOLS BY VALUE  |
+===================+
```

```
SYMBOLS...
----------
G_K_NOP
G_K_SUB1

G_K_STOP
```

VALUE	
00000000	DEFAULT
00000001	$MO
00000002	G_K_SUB2
00000003	$ENV
00000004	G_K_INIT
00000008	GD_K_CMDSIZ
00000084	GD_K_STRSIZ
000002DC	R-GD_L_LUN
000002E0	R-GD_A_PTBL
000002E4	R-GD_W_CSR
000002E6	R-GD_W_XOR
000002E8	R-GD_A_CSR
000002EC	R-GD_A_XOR
000002F0	R-GD_L_TOKEN
000002F4	R-GD_L_CMDBUF
000002FC	R-GD_L_STRBUF
00000300	R-GD_L_COMMAND
000003F1	R-GT_L_CVTCSR
00000415	R-GT_L_CVT32
000004C8	R-GT_L_CVT16
0000051D	R-GT_L_CSRFUNC
00000546	R-GT_L_GETLINE
00000550	R-GT_L_AMBCMD
00000567	R-GT_L_INVCMD
0000057C	R-GT_L_T1$2
00000598	R-GT_L_NOERR
000005CA	R-GT_L_CSRERR
000005DA	R-GT_L_FMTEXP
000005EF	R-GT_L_FMTRCV
00000604	R-GT_L_FMTXOR
00000622	R-GT_L_FMTCR
000006E4	R-ACT_ENTRY
00000727	R-MSG_REGERR
00000810	R-TEST_001
00000895	R-T1_$1
00000926	R-T1_$2
00000960	R-T1_$3
0000098B	R-TEST_001_X

```
KEY FOR SPECIAL CHARACTERS ABOVE:
+------------------+
| * = UNDEFINED    |
| U = UNIVERSAL    |
| R = RELOCATABLE  |
| WK = WEAK        |
+------------------+
```

```
DB0:[STAPLES]EVPRG.EXE;3                          8-AUG-1979 16:13        LINKER X01.20

                                        +------------------+
                                        ! IMAGE SYNOPSIS   !
                                        +------------------+

VIRTUAL MEMORY ALLOCATED:     000000200 000009FF 00000800 (2048. BYTES, 4. PAGES)
STACK SIZE:                   2. PAGES
IMAGE BINARY VIRTUAL BLOCK LIMITS:      1.      4. (   4. BLOCKS)
IMAGE NAME AND IDENTIFICATION:  EVPRG .EXE;3
NUMBER OF FILES:              2.
NUMBER OF MODULES:            2.
NUMBER OF PROGRAM SECTIONS:   16.
NUMBER OF GLOBAL SYMBOLS:     42.
NUMBER OF CROSS REFERENCES:   61.
NUMBER OF IMAGE SECTIONS:     1.
IMAGE TYPE:                   SYSTEM.
MAP FORMAT:                   FULL WITH CROSS REFERENCE IN FILE "DB0:[STAPLES]EVPRG.MAP;3"
ESTIMATED MAP LENGTH:         46. BLOCKS

                                        +---------------------+
                                        ! LINK RUN STATISTICS !
                                        +---------------------+

PERFORMANCE INDICATORS       PAGE FAULTS    CPU TIME       ELAPSED TIME
---------- ----------        ---- ------    --- ----       ------- ----
   COMMAND PROCESSING:-           16       00:00:00.06     00:00:00.32
   PASS 1:-                       48       00:00:00.21     00:00:01.32
   ALLOCATION/RELOCATION:-        21       00:00:00.08     00:00:00.36
   PASS 2:-                       18       00:00:00.18     00:00:01.18
   MAP DATA AFTER OBJECT MODULE SYNOPSIS:-  25  00:00:00.35  00:00:00.53
   SYMBOL TABLE OUTPUT:-           1       00:00:00.01     00:00:00.16
TOTAL RUN VALUES:-                129      00:00:00.89     00:00:03.91

USING A WORKING SET LIMITED TO 150 PAGES AND 22 PAGES OF DATA STORAGE (EXCLUDING IMAGE)

TOTAL NUMBER OBJECT RECORDS READ (BOTH PASSES):  120
   OF WHICH 0 WERE IN LIBRARIES AND 4 WERE DEBUG DATA RECORDS CONTAINING 431 BYTES

THERE WERE 0 LIBRARY BLOCK READ OPERATIONS
   WHICH ENCOMPASSED A TOTAL OF 0 BLOCKS
   USING A WINDOW OF 10 BLOCKS

NUMBER OF MODULES EXTRACTED EXPLICITLY      = 0
   WITH 0 EXTRACTED TO RESOLVE UNDEFINED SYMBOLS

0 LIBRARY SEARCHES WERE FOR SYMBOLS NOT IN THE LIBRARY SEARCHED

A TOTAL OF 0 GLOBAL SYMBOL TABLE RECORDS WAS WRITTEN
```

Page 8

8-AUG-1979 16:09:35   VAX-11 Macro V02.30

EVPRG   Diagnostic Program Example
Table of contents

EVPRG
V1.0

Diagnostic Program Example

```
                              8-AUG-1979 16:09:35   VAX-11 Macro V02.30
                              8-AUG-1979 16:00:34   DB0:[STAPLES]EVPRG.MAR;4

                       .TITLE  EVPRG   Diagnostic Program Example
                       .IDENT  /V1.0/

 1   0000
 2   0000
 3   0000
 4   0000
 5   0000    ;
 6   0000    ; Copyright (C) 1979
 7   0000    ; Digital Equipment Corporation, Maynard, Massachusetts 01754
 8   0000    ;
 9   0000    ; This software is furnished under a license for use only on a single
10   0000    ; computer system and may be copied only with the inclusion of the
11   0000    ; above copyright notice. This software, or any other copies thereof,
12   0000    ; may not be provided or otherwise made available to any other person
13   0000    ; except for use on such system and to one who agrees to these license
14   0000    ; terms. Title to and ownership of the software shall at all times
15   0000    ; remain in DEC.
16   0000    ;
17   0000    ; The information in this software is subject to change without notice
18   0000    ; and should not be construed as a commitment by Digital Equipment
19   0000    ; Corporation.
20   0000    ;
21   0000    ; DEC assumes no responsibility for the use or reliability of its
22   0000    ; software on equipment which is not supplied by DEC.
23   0000    ;
24   0000    ;
25   0000    ;++
26   0000    ; Facility:    VAX Diagnostic System
27   0000    ;
28   0000    ; Abstract:    The program consists of two subtests in one test.
29   0000    ;              The subtests are executed via commands by the user.
30   0000    ;              The main test routine uses the command parser in
31   0000    ;              conjunction with a command syntax tree.
32   0000    ;
33   0000    ; Environment: VAX Diagnostic Supervisor
34   0000    ;
35   0000    ; Author:      TED BEAR      8-Aug-79        Version V1.0
36   0000    ; Modified by:
37   0000    ;
38   0000    ;--
```

EVPRG
V1.0

Diagnostic Program Example
Declarations

A=AUG-1979 16:09:35
B=AUG-1979 16:00:34

VAX-11 Macro V02.30
DB0:[STAPLE8]EVPRG.MAR;4

Page 2 (2)

```
40              .SBTTL  Declarations
41      ;+
42      ; Include files:
43      ;-
44
45              .LIBRARY  \SYS$LIBRARY:DIAG\     ; VAX Diagnostic Macro Library
46
47      ;+
48      ; Macros:
49      ;-
50
51      ;+
52      ; Equated symbols:
53      ;-
54
55      ; Create symbols for the program/supervisor interface.
56
57      ;
58
59              $DS_BGNMOD  <SEP_REPAIR>         ; Level 3, system environment
        DIAGNOSTIC MACRO LIBRARY V5.0+ "DIAG.MLB (574)"
60
61              $DS_BITDEF                       ; mnemonic bit definition
62              $DS_CHDEF                        ; channel service symbols
63              $DS_CLIDEF                       ; syntax tree symbols & literals
64              $DS_DSADEF                       ; offsets in supervisor/APT mailbox
65              $DS_DSSDEF                       ; supervisor service entry vectors
66              $DS_ERRDEF                       ; error call parameter offsets
67              $DS_PARDEF                       ; parameter code symbols
68              $DS_PRINTX_DEF                   ; print call parameter offsets
69
70      ;
71      ; Define action codes for the command interpreter tree.
72      ;
73
00000000 74     G_K_NOP  == 0                    ; no action
00000001 75     G_K_SUB1 == 1                    ; select subroutine #1
00000002 76     G_K_SUB2 == 2                    ; select subroutine #2
00000003 77     G_K_STOP == 3                    ; exit from conversion mode
00000004 78     G_K_INIT == 4                    ; token block initialization
```

EVPRG
V1.0
Diagnostic Program Example
Program Header Data Block

```
              .SBTTL  Program Header Data Block
;++
; Functional description:
;
; The program header data block contains the parameters which
; allow the diagnostic supervisor to control the program.
; The diagnostic supervisor looks for the header information
; beginning at virtual address 200(X).
;--

        $DS_HEADER  <Diagnostic Example>, -
            REV    = 1, -          ; major revision
            UPDATE = 0, -          ; minor revision
            NUNIT  = 0             ; max # of selectable units
            .SAVE
            .PSECT  $HEADER, PAGE, NOEXE, NOWRT
SL_HEADLENGTH:  .LONG  A_HEADEND-.           ; LENGTH OF THE HEADER DATA BLOCK
SL_ENVIRON:     .LONG  <ENVS_SUPER@ENVSV_SUPER>+$ENV  ; PROGRAM ENVIRONMENT
SA_NAME:        .ADDRESS  T_NAME             ; PROGRAM NAME TEXT ADDRESS
SL_REV:         .LONG  1                     ; PROGRAM REVISION LEVEL
SL_UPDATE:      .LONG  0                     ; DIAGNOSTIC ENGINEERING PATCH ORDER
SA_LASTAD:      .ADDRESS  LASTAD             ; FIRST FREE LOCATION AFTER PROGRAM
SA_DTP:         .ADDRESS  DISPATCH           ; TEST DISPATCH TABLE POINTER
RA_DEVP:        .ADDRESS  AL_DEVTYP          ; DEVICE TYPE LIST POINTER
SL_UNIT:        .LONG  0                     ; NUMBER OF UNITS THAT CAN BE TESTED
SA_DREG:        .ADDRESS  DEV_REG            ; DEVICE REGISTER CONTENTS TABLE POINTER
                .LONG  28                    ; Ptable length for 4.05 compatability
                .LONG  0[4]                  ; UNUSED OFFSETS
SA_ICP:         .ADDRESS  INITIALIZE         ; INITIALIZATION CODE POINTER
SA_CCP:         .ADDRESS  CLEAN_UP           ; CLEAN-UP CODE POINTER
SA_REPP:        .ADDRESS  SUMMARY            ; SUMMARY REPORT CODE POINTER
SA_STATAB:      .ADDRESS  0                  ; STATISTIC TABLE POINTER
SL_ERRTYP:      .LONG  0                     ; # OF TYPES OF SERRSOFT & SERRHARD
SA_SECNAM:      .ADDRESS  SECTION            ; LIST OF SECTION NAME ADDRESSES
SA_TSTCNT:      .ADDRESS  L_TSTCNT           ; POINTER TO NUMBER OF TESTS
A_HEADEND:
T_NAME:         .ASCIC  \Diagnostic Example\

                .PSECT  _LAST, PAGE
LASTAD:
                .PSECT  STSTCNT, NOEXE, NOWRT, OVR, LONG
L_TSTCNT:
                .RESTORE
```

```
EVPRG       Diagnostic Program Example              8-AUG-1979 16:49:35    VAX-11 Macro V02.30                Page  4
V1.0        Dispatch Table                          8-AUG-1979 16:00:34    DB0:[STAPLES]EVPRG.MAR;4               (4)

                                95                  .SBTTL  Dispatch Table
                                96          ;++
                                97          ; Functional description:
                                98          ;
                                99          ; The dispatch table is a collection of addresses generated at the
                               100          ; beginning of each test and grouped together into a contiguous
                               101          ; list by the linker. This is done by defining a psect called
                               102          ; dispatch.
                               103          ;--
                               104                  $DS_DISPATCH
                               105                      .SAVE
        00000000                                      .PSECT  DISPATCH, LONG, NOWRT, NOEXE

               00000000        DISPATCH:
               00000002                                .PSECT  DISPATCH_X, LONG, NOWRT, NOEXE
00000000'00000000'00000000'00000000'00000000'  0010            .LONG   0[6]
00000000'00000000'00000000                      0018

                                                        .RESTORE
```

```
EVPRG                           8-AUG-1979 16:09:35   VAX-11 Macro V02.30        Page 5 (5)
V1.0                            8-AUG-1979 16:00:34   DB0:[STAPLES]EVPRG.MAR;4

Diagnostic Program Example
Global Data Section

0000        107            .SBTTL  Global Data Section
0000        108    ;+
0000        109    ; Functional description:
0000        110    ;
0000        111    ;       The global data section defines all permanent and working storage
0000        112    ;       elements referenced by more than one module.
000         113    ;-
00000000    114
            115            .PSECT  GBL_DATA, LONG, NOEXE
0000        116
0000        117    ;+
00000       118    ; Program control locations
0000        119    ;-
0000        120
00000000    121    GD_L_LUN::
00000000    122            .BLKL   1               ; number of logical unit under test
00000004    123    GD_A_PTBL::
00000008    124            .BLKL   1               ; address of pointer to P-table
0000        125
0000        126    DEV_REG:
0000        127    GD_W_CSR::
0000000A    128            .BLKW   1               ; buffer for CSR contents
0000000A    129    GD_W_XOR::
0000000A    130            .BLKW   1               ; xor of exp and rec data
0000000C    131    GD_A_CSR::
00000010    132            .BLKL   1               ; pointer to CSR contents
00000010    133    GD_A_XOR::
00000014    134            .BLKL   1               ; pointer to xor data
0000        135
0000        136    GD_L_TOKEN::
00000018    137            .BLKL   1               ; buffer for action routine address
00000       138
00000018    139    GD_T_CMDBUF::
00000020    140            .BLKB   8               ; conversation mode command buffer
00000028    141    GD_K_CMDSIZ ==  . - GD_T_CMDBUF  ; size of command buffer
00000       142
00000028    143    GD_T_STRBUF::
000000A4    144            .BLKB   132             ; string I/O buffer
000000A4    145    GD_K_STRSIZ ==  . - GD_T_STRBUF  ; size of string buffer
000A4       146
000A4       147    ;+
000A4       148    ; Statistics table (not implemented).
000A4       149    ;-
0000        150
000A4       151    STATISTIC:      $DS_BGNSTAT
000A4       152                    $DS_ENDSTAT
```

EVPRG
V1.0

Diagnostic Program Example
Global Data Section

8-AUG-1979 16:09:35   VAX-11 Macro V02.30
8-AUG-1979 16:00:34   DB0:[STAPLES]EVPRG.MAR;4

```
                                      154                          .PSECT  GBL_DATA, LONG, NOEXE
                                      155  ;+
                                      156  ; Command Interpreter Tree
                                      157  ;
                                      158  ; This tree provides nodes for the interpretation of three commands:
                                      159  ;
                                      160  ;       SUB1
                                      161  ;       SUB2
                                      162  ;       STOP
                                      163  ;-
                                      164
                                      165  GD_T_COMMAND::
                                      166  ;
                                      171  ; First, clear token block.  If it remains clear, the command is ambiguous.
                                      172  ;
                                      174          $DS_CLI CLISK_BR,       G_K_INIT,    10$
                                      175
                                      176  ; Is the first character an "S"?
                                      177  ;
                                      180  10$:    $DS_CLI <"A"S">,        G_K_NOP,     40$
                                      181
                                      182  ; Followed by "TOP"?  If yes, operator typed "STOP".
                                      183  ;
                                      186          $DS_CLI CLISK_STRING,   G_K_STOP,    20$,    <\TOP\>
                                      187          $DS_CLI CLISK_EXIT
                                      188
                                      189  ; "S" followed by "UB"?
                                      192  20$:    $DS_CLI <"A"U">,        G_K_NOP,     40$
                                      193          $DS_CLI <"A"B">,        G_K_NOP,     40$
                                      195
                                      196  ; Followed by "1"?  If yes, operator typed "SUB1".
                                      199          $DS_CLI <"A"1">,        G_K_SUB1,    30$
                                      201          $DS_CLI CLISK_EXIT
                                      202
                                      203  ; "SUB" followed by a "2"?  If yes, operator typed "SUB2".
                                      207  30$:    $DS_CLI <"A"2">,        G_K_SUB2,    40$
                                      208          $DS_CLI CLISK_EXIT
```

```
Diagnostic Program Example                  8-AUG-1979 16:09:35   VAX-11 Macro V02.30
Global Data Section                         8-AUG-1979 16:30:34   DB0:[STAPLES]EVPRG.MAR;4

4000                 210 ;
4000                 211 ; Operator has typed an invalid command.
4000                 212 ;
8000                 213
8000                 214  435:    $DS_CLI CLI$K_ERROR
```

EVPRG
V1.0

EVPRG
V1.0

Diagnostic Program Example
Global Text Section

8-AUG-1979 16:09:35   VAX-11 Macro V02.30
8-AUG-1979 16:00:34   DB0:[STAPLES]EVPRG.MAR;4

```
219                    .SBTTL  Global Text Section
220           ;++
221  ;  Functional description:
222  ;
223  ;        This section contains all the common character string type
224  ;        data entries.
225  ;--
226
227          .PSECT  GHL_TEXT, LONG, NOEXE, NOWRT
228
229   ;++
230   ; Program section names.
231   ;--
232
233          $DS_SECTION
     S_DEFAULT:      .ASCIC  \DEFAULT\

     SECTION:        .LONG   1
                     .LONG   S_DEFAULT
```

EVPRG
V1.0

Diagnostic Program Example
Global Text Section

8-AUG-1979 16:09:35   VAX-11 Macro V02.30
8-AUG-1979 16:08:34   DB0:[STAPLES]EVPRG.MAR;4

```
00000010                              235          .PSECT  G$L_TEXT, LONG, NOEXE, NOWRT
   0010                               236     ;+
   0015                               237     ; Device mnemonics list.
   0015                               238     ;-
   001A                               239
   001A                               240     $DS_DEVTYP   .ASCIC   <RP06,RM03,RK07,TE16,TU77>

36 50 50 52 00' 001A                         T_RP06:      .ASCIC   \RP06\
      04   001F
33 30 4D 52 00' 001F                         T_RM03:      .ASCIC   \RM03\
      04   0024
37 30 4B 52 00' 0024                         T_RK07:      .ASCIC   \RK07\
      04   0029
36 31 45 54 00' 0029                         T_TE16:      .ASCIC   \TE16\
      04   002E
37 37 55 54 00' 002E                         T_TU77:      .ASCIC   \TU77\
      04   0031
00000005  0031                               AL_DEVTYP:   .LONG    5$$
00000010' 0035                                            .LONG    T_RP06
00000015' 0039                                            .LONG    T_RM03
0000001A' 003D                                            .LONG    T_RK07
0000001F'                                                 .LONG    T_TE16
00000024'                                                 .LONG    T_TU77
```

```
EVPRG          Diagnostic Program Example        8-AUG-1979 16:09:35   VAX-11 Macro V02.30
V1.0           Global Text Section               8-AUG-1979 16:04:34   DB0:[STAPLES]EVPRG.MAR;4

242                    .PSECT  GBL_TEXT, LONG, NOEXE, NOWRT
243            ;+
244            ; Device registers mnemonics.
245            ;-
246
247  GT_T_CVTCSR::
248            .ASCIC  \DONE,IE,,,,,,,UNIT=^03,FUNC=^2,GO\

249
250  GT_T_CVT32::
251            .ASCIC  \BIT31,BIT30,BIT29,BIT28,BIT27,BIT26,BIT25,BIT24\ -

252                    \BIT23,BIT22,BIT21,BIT20,BIT19,BIT18,BIT17,BIT16\ -
253                    \BIT15,BIT14,BIT13,BIT12,BIT11,BIT10,BIT9,BIT8\ -

254                    \BIT7,BIT6,BIT5,BIT4,BIT3,BIT2,BIT1,BIT0\

255  GT_T_CVT16::
256            .ASCIC  \BIT15,BIT14,BIT13,BIT12,BIT11,BIT10,BIT9,BIT8\ -
257

258                    \BIT7,BIT6,BIT5,BIT4,BIT3,BIT2,BIT1,BIT0\

259  GT_T_CSRFUNC::
260            .LONG   4, T_NOP, T_READ, T_WRT, T_RST
261
262
263  T_NOP:   .ASCIC  \NOP\
264  T_READ:  .ASCIC  \READ\
265  T_WRT:   .ASCIC  \WRITE\
266  T_RST:   .ASCIC  \RESET\
```

```
EVPRG                Diagnostic Program Example                      8-AUG-1979 16:09:35    VAX-11 Macro V02.30
V1.0                 Global Text Section                             4-AUG-1979 16:00:34    DB0:[STAPLES]EVPRG.MAR;4

       00000196                                        268                    .PSECT  GBL_TEXT, LONG, NOEXE, NOWRT
            0196                                        269  ;+
            0196                                        270  ; Formatted ascii output statements.
            0196                                        271  ;   These format statements serve the command interpreter.
            0196                                        272  ;-
            0196                                        273
            0196                                        274  GT_T_GETLINE::
            0196                                        275       .ASCII  <13><10>\EVPRG> \
 20 3E 47 52 50 45 0A 0D 00'  0196
                           09  01A0
                               01A0                     276  GT_T_AMBCMD::
                                                        277       .ASCII  \I/?? Ambiguous command\
 75 67 69 62 6D 41 20 3F 3F 2F 21 00'  01A0
 64 6E 61 6D 6F 63 20 73 75      6F  01AC
                                 16  01A3
                                     01A7
                                     01A7            278  GT_T_INVCMD::
                                                     279       .ASCII  \I/?? Invalid command\
 69 6C 61 76 6E 49 20 3F 3F 2F 21 00'  01A7
 64 6E 61 6D 6D 6F 63 20  01C3
                      14  01H7
                          01CC
                          01CC            280  GT_T_T1S?::
                                          281       .ASCII  \Test 1, Subtest 2 is executing\
 62 75 53 20 2C 31 20 74 73 65 54 00'  01CC
 76 65 20 73 69 20 32 20 74 73 65 74 74  01D8
 67 6E 69 74 75 63 65  01E4
                   1E  01E4
                       01EB
                       01EB
                       01EB            282
                                       283  ;+
                                       284  ; Error report statements.
                                       285  ;   These output statements provide error messages to the operator.
                                       286  ;-
                                       287
                                       01EB            288  GT_T_REGERR::
                                       01EB            289       .ASCII  \Device register error\
 69 67 65 72 20 65 63 69 76 65 44 00'  01EB
 72 6F 72 72 65 20 72 65 74 73  01F7
                            15  01F8
                                01F8            290  GT_T_NOERR::
                                                291       .ASCII  \No device register error\
 72 20 65 63 69 76 65 64 20 6F 4E 00'  0201
 6F 72 72 65 20 72 65 74 73 69 67 65  0240
 72                                     0219
 1A                                     0201
                                        021A            292  GT_T_CSRERR::
                                        021A            293       .ASCII  \I/ CSR in error\
 65 20 6E 69 20 52 53 43 20 2F 21 00'  021A
          72 6F 72 72 65  0226
                      0F  021A
```

A-20

```
EVPRG      Diagnostic Program Example                8-AUG-1979 16:09:35    VAX-11 Macro V02.30
V1.0       Global Text Section                       8-AUG-1979 16:00:34    DB0:[STAPLES]EVPRG.MAR;4

00000022A                                     295          .PSECT GBL_TEXT, LONG, NOEXE, NOWRT
    022A                                      296   ;+
    022A                                      297   ; Formatted ascii output statements.
    022A                                      298   ;    These format statements serve the error message subroutine.
    022A                                      299   ;-
    022A                                      300
    022A                                      301   GT_T_FMTEXP::
65 74 63 65 70 78 45 20 2D 20 2F 21 00*022A   302          .ASCIC  \I/ Expected: !XW(X)\
      29 58 20 28 57 21 20 3A 64 022A
                                  14 022A

    023F                                      303   GT_T_FMTRCV::
65 76 69 65 63 65 52 20 2D 20 2F 21 00*023F   304          .ASCIC  \I/ Received: !XW(X)\
      29 58 20 28 57 21 20 3A 64 023F
                                  14 023F

    0254                                      305   GT_T_FMTXOR::
6F 58 20 20 20 20 20 20 2F 21 00*0254         306          .ASCIC  \I/    Xor: !XW(X)  ; !ACI\
3B 20 20 20 29 58 57 21 20 3A 72 026D
                        2F 21 43 41 026C
                                    1D 0254

    0272                                      307   GT_T_FMTCR::
                  2F 21 00*0272               308          .ASCIC  \I\
                        02 0272

    0272                                      309   ;+
    0275                                      310   ; Formatted ascii output statements.
    0275                                      311   ;    These format statements are used to trace major code sections.
    0275                                      312   ;-
    0275                                      313
    0275                                      314
    0275                                      315   GT_T_INITEXE::
61 69 74 69 6E 20 2E 2E 2F 21 00*0275         316          .ASCIC  \I/.. Initialize routine executed.!\
65 6E 69 74 75 6F 72 20 65 6C 6C 6C 0281
2F 21 2E 64 65 74 75 63 65 78 20 23 028D
                                    0275

    0299                                      317   GT_T_CLEANEXE::
75 6E 61 65 6C 43 20 2E 2E 2F 21 00*0299      318          .ASCIC  \I/.. Cleanup routine executed.!\
70 65 20 65 6E 69 74 75 6F 72 20 70 02A5
2F 21 2E 64 65 74 75 63 65 65 20 65 02B1
                                    0299

    02BA                                      319   GT_T_SUMMARYEXE::
72 61 6D 6D 75 53 20 2E 2E 2F 21 00*02BA      320          .ASCIC  \I/.. Summary reporting routine executed.!\
20 67 6E 69 74 72 6F 70 65 72 20 79 02C6
63 65 78 65 20 65 6E 69 74 75 6F 72 02D2
                  2F 21 2E 64 65 74 75 02DE
                                    2A 02EA
```

```
Diagnostic Program Example                8-AUG-1979 16:19:35      VAX-11 Macro V02.30        Page 13 (13)
Initialization Code                       8-AUG-1979 16:34:34      DB0:[STAPLES]EVPRG.MAR;4

322                            .SBTTL  Initialization Code
323      02E5      ;++
324      02E5      ; Functional description:
325      02E5      ;
326      02E5      ;       This routine is executed at the beginning of each test sequence.
327      02E5      ;
328      02E5      ; Calling sequence:
329      02E5      ;
330      02E5      ;       The diagnostic supervisor calls this routine with a CALL instruction.
331      02E5      ;
332      02E5      ; Input parameters:
333      02E5      ;
334      02E5      ;       None
335      02E5      ;
336      02E5      ; Implicit inputs:
337      02E5      ;
338      02E5      ;       None
339      02E5      ;
340      02E5      ; Output parameters:
341      02E5      ;
342      02E5      ;       None
343      02E5      ;
344      02E5      ; Implicit outputs:
345      02E5      ;
346      02E5      ;       None
347      02E5      ;
348      02E5      ; Completion codes:
349      02E5      ;
350      02E5      ;       None
351      02E5      ;
352      02E5      ; Side effects:
353      02E5      ;
354      02E5      ;       None
355      02E5      ;--
```

```
EVPRG      Diagnostic Program Example          8-AUG-1979 16:09:35   VAX-11 Macro V02.30
V1.0       Initialization Code                 8-AUG-1979 16:00:34   DB0:[STAPLES]EVPRG.MAR;4

                                          357        $DS_BGNINIT
                    02E5                                  .SAVE
                    02E5                                  .PSECT  INITIALIZE,LONG
                 00000000                      INITIALIZE:
           0000     0000                   358              .WORD   ^M<>           ; ENTRY MASK
                    0002                   359                                     ; trace routine execution
00010050 9F         0002
           9F       0008                   358      $DS_PRINTB_S GT_T_INITEXE
           FB       0008                            PUSHAB  GT_T_INITEXE
                    000F                   359              CALLS   #$SN, @#DS$PRINTB
           DE       000F
                    0016                   360      MOVAL   GD_L_LUN, R1           ; setup for frequent reference
                    0016                   361
                    0016                   362      $DS_BPASS0 10$                 ; if this is the very first time
           E0       001A                   363                                    ; thru init, go do lun 0, otherwise
                                           364                                    ; BR IF PASS 0
           1D       001E                            BBS     #DSASV_PASS0,-         ; 10$
           9F       0022                                    @#DSASGL_FLAGS,
           61       0027                   365      INCL    (R1)                   ; bump up to next selected unit
           61       0027                   366      CMPL    (R1), DSASGL_UNITS     ; check against max. no. of units
           19       0029                   367      BLSS    20$                    ; if too big, reset
                    0029                   368
                    0029                   369      $DS_ENDPASS_G
           FA       0029                            CALLG   (SP), @#DS$ENDPASS     ; increment pass count
           04       002A                   370
           61       0032              10$:  CLRL    (R1)                           ; set lun to 0
                    0032              20$:  $DS_GPHARD_S
                    0032                   371      DEVNUM = (R1),                 ; ; get hardware p-table address
           EF       0032                   372      ADRLOC = GD_A_PTBL             ; ; for current logical unit
           DF       0032                   373      PUSHAL  GD_A_PTBL              ; ; and store for test routines
           DD       0038                   374      PUSHL   (R1)
           FB       0041                   375      CALLS   #2, @#DS$GPHARD
                    0041                   376
                    0041              $DS_ENDINIT
           6E       0041                   377      INITIALIZE_X:
                                                    CALLG   (SP), @#DS$BREAK       ; RETURN TO DIAGNOSTIC SUPERVISOR
           FA       0041                            RET
           04       0048                            .RESTORE
                 000002E5
```

A-23

EVPRG   Diagnostic Program Example   8-AUG-1979 16:09:35   VAX-11 Macro V02.30   Page 15
V1.0   Clean-up Code   8-AUG-1979 16:00:34   DB0:[STAPLES]EVPRG.MAR;4   (15)

```
02E5                        379            .SBTTL   Clean-up Code
02E5                        380    ;++
02E5                        381    ; Functional description:
02E5                        382    ;
02E5                        383    ;   This routine is executed at the completion of the last pass
02E5                        384    ;   or when the program is aborted.
02E5                        385    ;
02E5                        386    ; Calling sequence:
02E5                        387    ;
02E5                        388    ;   The diagnostic supervisor calls this routine with a CALL instruction.
02E5                        389    ;
02E5                        390    ; Input parameters:
02E5                        391    ;
02E5                        392    ;   None
02E5                        393    ;
02E5                        394    ; Implicit inputs:
02E5                        395    ;
02E5                        396    ;   None
02E5                        397    ;
02E5                        398    ; Output parameters:
02E5                        399    ;
02E5                        400    ;   None
02E5                        401    ;
02E5                        402    ; Implicit outputs:
02E5                        403    ;
02E5                        404    ;   None
02E5                        405    ;
02E5                        406    ; Completion codes:
02E5                        407    ;
02E5                        408    ;   None
02E5                        409    ;
02E5                        410    ; Side effects:
02E5                        411    ;
02E5                        412    ;   None
02E5                        413    ;
02E5                        414    ;--
02E5                        415
02E5                        416            $DS_BGNCLEAN
00000000                                  .SAVE
0000                                      .PSECT   CLEANUP,LONG
0000                        417    CLEAN_UP:
        9F   0000           418            .WORD    ^M<>              ; ENTRY MASK
        EF   0002
        01   0002                          $DS_PRINTB_S  GT_T_CLEANEXE    ; trace routine execution
             0008                          PUSHAB   GT_T_CLEANEXE
             000F                          CALLS    #$$N, @#DSSPRINTB
00010000 9F  000F           419            $DS_ENDCLEAN
             000F           420    CLEAN_UP_X:
        FA   000F                          CALLG    (SP), @#DS$BREAK  ; RETURN TO DIAGNOSTIC SUPERVISOR
00010058 9F  04   0016                     RET
             000002E5                       .RESTORE
```

EVPRG   Diagnostic Program Example   8-AUG-1979 16:09:35   VAX-11 Macro V02.30   Page 16 (16)
V1.0   Summary Report Code   8-AUG-1979 16:08:34   DB0:[STAPLES]EVPRG.MAR;4

```
02E5                          422           .SBTTL  Summary Report Code
02E5                          423  ;++
02E5                          424  ; Functional description:
02E5                          425  ;
02E5                          426  ;    This routine issues a summary report upon request from the operator
02E5                          427  ;    or when a $DS_SUMMARY call is made from with the program.
02E5                          428  ;
02E5                          429  ; Calling sequence:
02E5                          430  ;
02E5                          431  ;    The diagnostic supervisor calls this routine with a CALL instruction.
02E5                          432  ;
02E5                          433  ; Input parameters:
02E5                          434  ;
02E5                          435  ;    None
02E5                          436  ;
02E5                          437  ; Implicit inputs:
02E5                          438  ;
02E5                          439  ;    None
02E5                          440  ;
02E5                          441  ; Output parameters:
02E5                          442  ;
02E5                          443  ;    None
02E5                          444  ;
02E5                          445  ; Implicit outputs:
02E5                          446  ;
02E5                          447  ;    None
02E5                          448  ;
02E5                          449  ; Completion codes:
02E5                          450  ;
02E5                          451  ;    None
02E5                          452  ;
02E5                          453  ; Side effects:
02E5                          454  ;
02E5                          455  ;    None
02E5                          456  ;
02E5                          457  ;--
02E5                          458          $DS_BGNSUMMARY
00000000                           .SAVE
00000                              .PSECT  SUMMARY,LONG
0000                      459  SUMMARY:  .WORD  ^M<>          ; ENTRY MASK
0002                      460          $DS_PRINTB_S  GT_T_SUMMARYEXE  ; trace routine execution
000002BA'EF  0002           9F          PUSHAB  GT_T_SUMMARYEXE
             0008  31                   CALLS   #$$N, @#DSSPRINTB
             000F                  461  SUMMARY_X:  $DS_ENDSUMMARY
             000F             6E
00010058  9F 000F           FA   462          CALLG  (SP), @#DSSBREAK
             0016         04        RET                   ; RETURN TO COMMAND MODE
000002E5                           .RESTORE
```

A-25

VAX Diagnostic Design Guide

Diagnostic Program Example
Program Subroutines

8-AUG-1979 16:09:35   VAX-11 Macro V02.30   Page 17 (17)
8-AUG-1979 16:00:34   DB0:[STAPLES]EVPRG.MAR;4

EVPRG
V1.0

```
02E5    464                .SBTTL  Program Subroutines
02E5    465        ;++
02E5    466        ; Functional description:
02E5    467        ;
02E5    468        ;   This subroutine performs actions at the direction of codes from
02E5    469        ;   the command syntax tree.
02E5    470        ;
02E5    471        ; Calling sequence:
02E5    472        ;
02E5    473        ;   The diagnostic supervisor calls this routine with a JSB instruction
02E5    474        ;   from within the string parsing service.
02E5    475        ;
02E5    476        ; Input parameters:
02E5    477        ;
02E5    478        ;   None
02E5    479        ;
02E5    480        ; Implicit inputs:
02E5    481        ;
02E5    482        ;   None
02E5    483        ;
02E5    484        ; Output parameters:
02E5    485        ;
02E5    486        ;   None
02E5    487        ;
02E5    488        ; Implicit outputs:
02E5    489        ;
02E5    490        ;   None
02E5    491        ;
02E5    492        ; Completion codes:
02E5    493        ;
02E5    494        ;   None
02E5    495        ;
02E5    496        ; Side effects:
02E5    497        ;
02E5    498        ;   None
02E5    499        ;--
```

EVPRG   Diagnostic Program Example   8-AUG-1979 16:09:35   VAX-11 Macro V02.30
V1.0   Program Subroutines   8-AUG-1979 16:00:34   DB0:[STAPLES]EVPRG.MAR;4

```
                 00000000              501          .PSECT  SUBROUTINE, LONG, NOWRT
                 0000                  502
                 0000                  503  ACT_ENTRY::
        00       0000                  504
     50 CF       0003                  505          CASEL   R0, #0, #4
     54
        003E'    0004                  506  10$:     .WORD   ACT_NULL - 10$    ; if NULL
        0020'    0006                  507          .WORD   ACT_SUB1 - 10$    ; if SUB1
        0028'    0008                  508          .WORD   ACT_SUB2 - 10$    ; if SUB2
        0030'    000A                  509          .WORD   ACT_STOP - 10$    ; if STOP
        0038'    000C                  510          .WORD   ACT_INIT - 10$    ; if INIT
                 000E                  511
                 000E                  512          $DS_ERRSYS_S
     00 DD       000E                               PUSHL   #0
     00 DD       0010                               PUSHL   #0
     00 DD       0012                               PUSHL   #0
     01 DD       0014                               PUSHL   #$ER     ; fatal error

                                            ;;; GLOBAL ERROR 1
     04 FB       0016                  513          $DS_ABORT PROGRAM   CALLS   #$$M, @#DSSERRSYS
 00010000 9F
     6C FA       001D                               CALLG   (AP), @#DSSABORT  ; abort program
 00010020 9F
                 0024                  514
                 0024                  515
                 0024                  516  ;+
                 0024                  517  ; Action subroutines for command line parsing
                 0024                  518  ;-
                 0024                  519
                 0024                  520  ACT_SUB1:
     01 D0       0024                  521          MOVL    #G_K_SUB1, GD_L_TOKEN  ; load command routine adr
 00000014'EF
        05       002A                  522          RSB
                 002C                  523
                 002C                  524  ACT_SUB2:
     02 D0       002C                  525          MOVL    #G_K_SUB2, GD_L_TOKEN  ; load command routine adr
 00000014'EF
        05       0033                  526          RSB
                 0034                  527
                 0034                  528  ACT_STOP:
     03 D0       0034                  529          MOVL    #G_K_STOP, GD_L_TOKEN  ; load command routine adr
 00000014'EF
        05       003B                  530          RSB
                 003C                  531
                 003C                  532  ACT_INIT:
     04 D4       003C                  533          CLRL    GD_L_TOKEN            ; initialize token block
 00000014'EF
        05       0042                  534  ACT_NULL:
                 0042                  535          RSB
```

A-27

EVPRG
V1.0

Diagnostic Program Example
Program Subroutines

8-AUG-1979 16:09:35   VAX-11 Macro V02.30
8-AUG-1979 16:00:34   DB0:[STAPLES]EVPRG.MAR;4

```
537  ;++
538  ; Functional description:
539  ;
540  ;     This routine prints expected data, received data, and xor data.
541  ;     It also identifies the register bits in error.
542  ;
543  ; Calling sequence:
544  ;
545  ;     Called from within a $DS_ERRxxx call when PRLINK is specified.
546  ;
547  ; Input parameters:
548  ;
549  ;     P1 = address of error message
550  ;     P2 = address of register name string
551  ;     P3 = expected data
552  ;     P4 = received data
553  ;
554  ; Implicit inputs:
555  ;
556  ;     None
557  ;
558  ; Output parameters:
559  ;
560  ;     None
561  ;
562  ; Implicit outputs:
563  ;
564  ;     None
565  ;
566  ; Completion codes:
567  ;
568  ;     None
569  ;
570  ; Side effects:
571  ;
572  ;     None
573  ;--
574
575          .PSECT  SUBROUTINE, LONG, NOWRT
576
577  MSG_REGERR::
578          $DS_BGNMESSAGE  REGMASK=<R2,R3,R4,R5,R6>
             .WORD   ^M<R2,R3,R4,R5,R6>        ; ENTRY MASK
579          MOVL    ERRS_P1(AP), R2           ; name of register
580          MOVL    ERRS_P2(AP), R3           ; address of register
581          MOVZWL  ERRS_P3(AP), R4           ; expected data
582          MOVZWL  ERRS_P4(AP), R5           ; received data
583          $DS_PRINTB_S (R2)                 ; print message
584          PUSHAB  (R2)
             CALLS   #$$N, #DSSPRINTB
585          XORL3   R4, R5, R2                ; exclusive or
```

```
                        0262                587        $DS_CVTREG_S
                        0262                588          MSB = #15,          = ; 16 bit register
                        0262                589          DATA = R2,          = ; bits in error
                        0262                590          MNEADR = GT_T_CVT16, = ; string of bits mnemonics
                        0262                591          STRBUF = GD_T_STRBUF, = ; address of string buffer
                        0262                592          MAXLEN = #132,      = ; buffer length
                        0262                593
                  00    0262                               PUSHL  #0
                  00    0264                               PUSHL  #0
                  00    0266                               PUSHL  #0
                  00    0268                               PUSHL  #0
                  00    026A                               PUSHL  #0
                  00    026C                               PUSHL  #132
        00000084'8F 9F  026E                               PUSHAB GD_T_STRBUF
        00000020'EF 9F  0274                               PUSHAB GT_T_CVT16
        00000118'EF     027A                               PUSHL  R2
                  52 DD  0282                               PUSHL  #15
                  0F DD  0282                               CALLS  #11, @#DS$CVTREG
                  0B FB  0284        594        $DS_PRINTX_S
                        028B          FORMAT = GT_T_FMTEXP, = ; print message
                        028B          P0 = R4               = ; expected data
                  54    028B          595                    PUSHL  R4
        0000022A'EF 9F  028D          596                    PUSHAB GT_T_FMTEXP
                  02 FB 0293                               CALLS  #SN, @#DS$PRINTX
                  55 DD 029A          597        $DS_PRINTX_S
        0000023F'EF 9F  029A          598          FORMAT = GT_T_FMTRCV, = ; print message
                  02 FB 029A          599          P0 = R5               = ; received data
                  029C                               PUSHL  R5
                  02A2                               PUSHAB GT_T_FMTRCV
                  02A9          600        $DS_PRINTX_S
                  02A9          601          FORMAT = GT_T_FMTXOR, = ; print message
                  02A9          602          P0 = R2,            = ; xor data
                  02A9          603          P1 = #GD_T_STRBUF   = ; mnemonic bits in error
                  8F DD 02AF                               PUSHL  #GD_T_STRBUF
                  52 DD 02AF                               PUSHL  R2
        00000254'EF 9F  02B1                               PUSHAB GT_T_FMTXOR
                  03 FB 02B7          604                    CALLS  #SN, @#DS$PRINTX
                  02BE          625        $DS_PRINTX_S
        00000272'EF 9F  02BE          FORMAT = GT_T_FMTCR, = ; print <cr>/<lf>
                        02C4          PUSHAB GT_T_FMTCR
                  01 FB 02C9          CALLS  #SN, @#DS$PRINTX
                  A1 D0  02CB    MOVL  #1, R0               ; success return code
                        02CE        $DS_ENDMESSAGE
                  04    02CE          RET                    ; RETURN TO DIAGNOSTIC SUPERVISOR

00010CB0 9F
00010CE8 9F
00010CE8 9F
00010CE8 9F
00010CE8 9F
00010CE8 9F 50
```

Diagnostic Program Example   8-AUG-1979 16:09:35   VAX-11 Macro V02.30   Page 21
Program Subroutines   8-AUG-1979 16:00:34   DB0:[STAPLES]EVPRG.MAR;4   (21)

```
02CF    610        $DS_ENDMOD
02CF    611        .END
```

EVPRG
V1.0

EVPRG
Symbol table   Diagnostic Program Example   8-AUG-1979 16:09:35   VAX-11 Macro V02.30
                                              8-AUG-1979 16:00:34   DB0:[STAPLES]EVPRG.MAR;4

Symbol	Value	Flag
$$ARGS	00000011	
$$E	00000001	
$$M	00000004	
$$N	00000001	
$$S	FFFFFFFF	
$$T1	00000048	
$ENV	00000003	
$ER	00000002	G
$MO	00000001	
$ST	00000000	
$TN	00000000	
ACT_ENTRY	0000003C	RG
ACT_INIT	00000042	R
ACT_NULL	00000034	R
ACT_STOP	00000001	R
ACT_SUB1	00000002	R
ACT_SUB2	0000002C	R
AL_DEVTYP	00000029	R
A_HEADEND	00000058	R
BIT...	00000000	
BIT0	00000001	
BIT1	00000002	
BIT10	00000400	
BIT11	00000800	
BIT12	00001000	
BIT13	00002000	
BIT14	00004000	
BIT15	00008000	
BIT16	00010000	
BIT17	00020000	
BIT18	00040000	
BIT19	00080000	
BIT2	00000004	
BIT20	00100000	
BIT21	00200000	
BIT22	00400000	
BIT23	00800000	
BIT24	01000000	
BIT25	02000000	
BIT26	04000000	
BIT27	08000000	
BIT28	10000000	
BIT29	20000000	
BIT3	00000008	
BIT30	40000000	
BIT31	80000000	
BIT4	00000010	
BIT5	00000020	
BIT6	00000040	
BIT7	00000080	
BIT8	00000100	
BIT9	00000200	
CEP_FUNCTIONAL	00000000	
CEP_REPAIR	00000001	
CHCS_ABORT	00000002	
CHCS_CLEAR	00000006	
CHCS_CLRDFT	00000009	

Symbol	Value	Flag
CHC$_DSINT	00000005	
CHC$_ENINT	00000004	
CHC$_INITA	00000000	
CHC$_INITR	00000001	
CHC$_PURGE	00000003	
CHC$_SETDFT	0000000A	
CHC$_STATUS	00000007	
CHI$M_CHNINT	00000001	
CHI$M_DEVINT	00000002	
CHI$M_IPL	0000007C	
CHI$S_IPL	00000005	
CHI$V_CHNINT	00000000	
CHI$V_DEVINT	00000001	
CHI$V_IPL	00000002	
CHM$_FORWARD	00000001	
CHM$_INVALIDATE	00000001	
CHM$_MAP	00000002	
CHM$F_MFWDN	00000002	
CHM$S_MFWDNO	0000000A	
CHM$S_MFWDV	00000003	
CHM$F_MFWDVO	0000000B	
CHM$S_MREVN	00000006	
CHM$S_MREVNO	0000000E	
CHM$S_MREVV	00000007	
CHM$S_MREVVO	0000000F	
CHM$F_NFWDN	00000004	
CHM$F_NREVN	00000004	
CHM$_OFFSET	00000008	
CHM$F_REVERSE	00000004	
CHS$M_BUSIC	00000000	
CHS$M_BUSINIT	00000004	
CHS$M_BUSPDN	00000000	
CHS$M_CHNDPE	00000002	
CHS$M_CHNERR	00000010	
CHS$M_CHNMPE	00000000	
CHS$M_CHPFOT	00000100	
CHS$M_DEVBUS	00000010	
CHS$M_DEVERR	00000004	
CHS$M_DEVTQ	00000020	
CHS$M_ERRANY	0000000F	
CHS$M_MBAATN	00100000	
CHS$M_MBACPE	00200000	
CHS$M_MBADTB	00040000	
CHS$M_MBADTC	00080000	
CHS$M_MBAEXC	00010000	
CHS$M_MBANED	00020000	
CHS$M_PGMERR	00000008	
CHS$M_PGMHDE	00000001	
CHS$M_SYSERR	00000200	
CHS$M_SYSMEM	00000400	
CHS$M_SYSSBI	00000400	
CHS$V_BUSIC	00000017	
CHS$V_BUSINIT	00000016	
CHS$V_BUSPDN	00000018	
CHS$V_CHNDPE	00000006	
CHS$V_CHNERR	00000001	
CHS$V_CHNMPE	00000007	

Symbol	Value	Flag
CHS$V_CHPFOT	00000008	
CHS$V_DEVBUS	00000004	
CHS$V_DEVERR	00000002	
CHS$V_DEVTQ	00000005	
CHS$V_MBAATN	00000014	
CHS$V_MBACPE	00000015	
CHS$V_MBADTB	00000012	
CHS$V_MBADTC	00000013	
CHS$V_MBAEXC	00000010	
CHS$V_MBANED	00000011	
CHS$V_PGMERR	00000003	
CHS$V_PGMHDE	00000000	
CHS$V_SYSERR	00000009	
CHS$V_SYSMEM	0000000A	
CHS$V_SYSSBI	0000000A	R 08
CLEAN_UP	0000000F	R 08
CLEAN_UP_X	00000007	
CLISK_ALNUM	00000086	
CLISK_ALPHA	00000085	
CLISK_BIF	00000082	
CLISK_BR	0000008A	
CLISK_DEC	0000008A	
CLISK_ERROR	0000008E	
CLISK_EXIT	00000081	
CLISK_HEX	00000089	
CLISK_KEYWORD	0000008C	
CLISK_NUM	00000085	
CLISK_OCT	00000088	
CLISK_SPACE	00000084	
CLISK_STRING	00000088	
CPUSV_COMET	00000002	
CPUSV_HS	00000000	G 08
CPUSV_STAR	00000001	R 06
DEFAULT	00000008	
DEV_REG	00000000	R
DISPATCH	00010020	
DSSABORT	00010020	
DSSASKADR	00010090	
DSSASKDATA	00013060	
DSSASKLGCL	00013098	
DSSASKSTR	000100A0	
DSSASKYLD	00010088	
DSSBGNSUB	00010030	
DSSBREAK	00010058	
DSSCANWAIT	00010070	
DSSCHANNEL	00010180	
DSSCKLOOP	00010040	
DSSCLRVEC	00010168	
DSSCNTRLC	00010078	
DSSCVTREG	00010080	
DSSENDPASS	00010010	
DSSENDSUB	00010038	
DSSERRDEV	000100C0	
DSSERRHARD	00010180	
DSSERRSOFT	000100D0	
DSSERRSYS	000100C0	
DSSESCAPE	00010050	

EVPRG Symbol table   Diagnostic Program Example   8-AUG-1979 16:09:35   9-AUG-1979 16:08:34   VAX-11 Macro V02.30   DB0:[STAPLES]EVPRG.MAR;4

Symbol	Value		Symbol	Value		Symbol	Value
DSSGETBUF	00010120		DSASM_QUICK	00000100		GD_I_CMDBUF	00000010 RG 08
DSSGETMEM	00010130		DSASM_SPOOL	00000200		GD_I_COMMAND	00000044 RG 08
DSSGPHARD	00010018		DSASM_TRACE	00000033		GD_I_STRBUF	00000020 RG 08
DSSINITSCB	00010170		DSASM_USER	10000033		GD_W_CSR	0000000B RG 08
DSSINLOOP	00010048		DSASV_APT	0000001F		GD_A_XOR	0000000A RG 09
DSSLOAD	00010198		DSASV_BELL	00000003		GT_I_AMBCMD	000001A0 R 09
DSSHMOFF	0001015A		DSASV_COMPAT	0000001A		GT_I_CLEANEXE	00000299 R 09
DSSHMON	0001015A		DSASV_ENSPEC	0000001E		GT_T_CSRERR	0000021A RG 09
DSSMOVPHY	00010148		DSASV_HALTD	00000020		GT_T_CSRFUNC	0000016D RG 09
DSSMOVVRT	00010140		DSASV_HALTI	00000021		GT_I_CVT16	00000118 RG 09
DSSPARSE	00010088		DSASV_IE1	00000024		GT_I_CVT32	00000065 RG 09
DSSPRINTB	000100E0		DSASV_IE2	00000005		GT_T_CVTCSR	00000041 RG 09
DSSPRINTF	000100F0		DSASV_IE3	00000006		GT_I_FMTCR	00000272 RG 09
DSSPRINTS	000100F8		DSASV_IES	00000007		GT_I_FMTEXP	0000022A RG 09
DSSPRINTX	000100E8		DSASV_LOCK	0000000B		GT_I_FMTRCV	0000023F RG 09
DSSRELBUF	00010128		DSASV_LOOP	00000002		GT_I_FMTXOR	00000254 RG 09
DSSRELMEM	00010138		DSASV_NORPT	00000010		GT_T_GETLINE	00000196 RG 09
DSSSETPL	00010178		DSASV_OPER	0000000C		GT_I_INITEXE	00000275 R 09
DSSSETMAP	00010188		DSASV_PASSO	0000000D		GT_I_INVCMD	00000187 RG 09
DSSSETVEC	0001016A		DSASV_PROMPT	0000000F		GT_I_NOERR	00000201 RG 09
DSSSHOCHAN	00010190		DSASV_QUICK	00000008		GT_I_REGERR	000001EB RG 09
DSSSUMMARY	00010028		DSASV_SPOOL	00000009		GT_I_SUMMARYEXE	000002BA R 09
DSSWAITMS	00010060		DSASV_TRACE	0000000A		GT_T_T1S2	000001CC RG 09
DSSWAITUS	00010068		DSASV_USER	0000001C		G_K_INIT	00000004
DSASAL_APTMAIL	0000FE00		ENVSM_DOMAIN	00000002		G_K_NOP	00000003
DSASAT_APTTXT	0000FA00		ENVBM_LEVEL	00000001		G_K_STOP	00000001
DSASGL_APTCOM	0000FE04		ENVBM_SUPER	000003FC		G_K_SUB1	00000002
DSASGL_DEVLEN	0000FE58		ENVS_DOMAIN	00000001		G_K_SUB2	00000000
DSASGL_ERRNO	0000FE48		ENVS_LEVEL	00000008		INITIALIZE	00000041 R
DSASGL_EVENT	0000FE00		ENVS_SUPER	00000001		INITIALIZE_X	0000001C R
DSASGL_FLAGS	0000FE40		ENVV_DOMAIN	00000000		LSA_CCP	00000040 R
DSASGL_MSGTYP	0000FE40		ENVV_LEVEL	00000001		LSA_DEVP	0000001C R
DSASGL_PASSES	0000FE4C		ENVV_SUPER	00000002		LSA_DREG	00000024 R
DSASGL_PASSNO	0000FE54		ENVS_CPU	00000000		LSA_DTP	00000018 R
DSASGL_SECTNO	0000FE10		ENVF_FUNCTIONAL	00000000		LSA_ICP	0000003C R
DSASGL_SID	0000FE14		ENVF_REPAIR	00000001		LSA_LASTAD	00000014 R
DSASGL_SUBTNO	0000FE4C		ENVS_SUBPER	00000001		LSA_NAME	00000044 R
DSASGL_TESTNO	0000FE50		ENVS_SYSTEM	00000001		LSA_REPP	00000050 R
DSASGL_UNITS	0000FE0C		ERRS_MSGADR	0000000C		LSA_SECNAM	00000048 R
DSASGT_MSGPTR	0000FE68		ERRS_NARGS	0000000A		LSA_STATAB	00000054 R
DSASGT_DEVNAM	0000FE5C		ERRS_NUM	00000000		LSA_TSTCNT	00000004 R
DSASM_APT	80000000		ERRS_P1	00000014		LSL_ENVIRON	0000004C R
DSASM_BELL	04000000		ERRS_P2	00000018		LSL_ERRTYP	00000010 R
DSASM_COMPAT	40000000		ERRS_P3	0000001C		LSL_HEADLENGTH	0000000C R
DSASM_ENSPEC	00000001		ERRS_P4	00000020		LSL_REV	00000020 R
DSASM_HALTD	00000002		ERRS_P5	00000024		LSL_UNIT	00000010 R
DSASM_HALTI	00000020		ERRS_P6	00000028		LSL_UPDATE	00000008 R
DSASM_IE1	00000040		ERRS_POINTER	00000008		LASTAD	00000002 R
DSASM_IE2	00000080		ERRS_UNIT	00000004		L_TSTCNT	00000000 R
DSASM_IE3	00000000		GBL...			MSG_REGERR	00000043 RG 0D
DSASM_IES	00000000		GD_A_CSR	0000000C RG 08		PARSM_ATDEF	00000010
DSASM_LOCK	00000200		GD_A_PTBL	0000000C RG 08		PARSM_ATHI	00000008
DSASM_LOOP	00000004		GD_A_XOR	00000010 RG 08		PARSM_ATLO	00000004
DSASM_NORPT	00000100		GD_K_CMDSIZ	00000008 G		PARSM_NODEF	00000002
DSASM_OPER	00000010		GD_K_STRSIZ	00000384 G		PARSV_ATDEF	00000004
DSASM_PASSO	20000000		GD_L_LUN	00000000 RG 08		PARSV_ATHI	00000003
DSASM_PROMPT	00002000		GD_L_TOKEN	00000014 RG		PARSV_ATLO	00000002

EVPRG   Diagnostic Program Example   8-AUG-1979 16:09:35   VAX-11 Macro V02.30
Symbol table   8-AUG-1979 16:00:34   DB0:[STAPLES]EVPRG.MAR;4

Symbol	Value				Symbol	Value		
PARSV_NODEF	00000001	=			SYS$WFLOW	0001A490		R
PARS_BIN	00000002	=			S_DEFAULT	00000002	09	
PARS_DEC	0000000A	=			TE16	00000004		
PARS_HEX	00000010	=			TU77	00000005		
PARS_NO	00000000	=			T_NAME	00000058	03	R
PARS_OCT	00000008	=			T_NOP	00000181	09	R
PARS_YES	00000001	=			T_READ	00000185	09	R
PRINTX$_FORMAT	00000004	=			T_RK07	0000001A	09	R
PRINTX$_NARGS	00000011	=			T_RM03	00000015	09	R
PRINTX$_P0	00000008	=			T_RP06	00000210	09	R
PRINTX$_P1	0000000C	=			T_RST	00000190	09	R
PRINTX$_P2	00000010	=			T_TE16	0000001F	09	R
PRINTX$_P3	00000014	=			T_TU77	00000724	09	R
PRINTX$_P4	00000018	=			T_WRT	0000018A	09	R
PRINTX$_P5	0000001C	=						
PRINTX$_P6	00000020	=						
PRINTX$_P7	00000024	=						
PRINTX$_P8	00000028	=						
PRINTX$_P9	0000002C	=						
PRINTX$_PA	00000030	=						
PRINTX$_PB	00000034	=						
PRINTX$_PC	00000038	=						
PRINTX$_PD	0000003C	=						
PRINTX$_PE	00000040	=						
PRINTX$_PF	00000044	=						
RK07	00000003	=						
RM03	00000002	=						
RP06	00000001	=						
SECTION	00000000	09	R					
SEP_FUNCTIONAL	00000002	=						
SEP_REPAIR	00000003	98	R					
STATISTIC	00000044	0C	R					
SUMMARY	0000000F	0C	R					
SUMMARY_X	0000000F							
SYS$ALLOC	00010236							
SYS$ASCTIM	00010248							
SYS$ASSIGN	00010250							
SYS$BINTIM	00010258							
SYS$CANCEL	00010260							
SYS$CANTIM	00010268							
SYS$CLREF	00010298							
SYS$DALLOC	000102D8							
SYS$DASSGN	000102E0							
SYS$FAO	00010350							
SYS$FAOL	00010358							
SYS$GETCHN	000104C8							
SYS$GETTIM	00010378							
SYS$NUMTIM	000103B8							
SYS$QIO	000103C8							
SYS$QIOW	00010200							
SYS$READEF	000103D0							
SYS$SETEF	00010400							
SYS$SETIMR	00010420							
SYS$SETPRT	0001043A							
SYS$UNWIND	00010478							
SYS$WAITFR	00010478							
SYS$WFLAND	00010488							

VAX Diagnostic Design Guide

```
EVPRG    Diagnostic Program Example        8-AUG-1979 16:09:35   VAX-11 Macro V02.30
Psect synopsis                             8-AUG-1979 16:04:34   DB0:[STAPLES]EVPRG.MAR;4
```

```
                          +-------------------+
                          |  Psect synopsis   |
                          +-------------------+
```

PSECT name	Allocation		PSECT No.	Attributes
. ABS .	00000000	(0.)	00	NOPIC USR CON ABS LCL NOSHR NOEXE NORD NOWRT BYTE
. BLANK .	00000000	(0.)	01	NOPIC USR CON REL LCL NOSHR EXE RD WRT BYTE
ABS	000104D0	(66768.)	02	NOPIC USR CON ABS LCL NOSHR EXE RD WRT BYTE
$HEADER	0000006B	(107.)	03	NOPIC USR CON REL LCL NOSHR NOEXE RD NOWRT PAGE
$LAST	00000000	(0.)	04	NOPIC USR OVR REL LCL NOSHR EXE RD WRT PAGE
$TSTCNT	00000002	(2.)	05	NOPIC USR CON REL LCL NOSHR NOEXE RD NOWRT LONG
DISPATCH	00000018	(24.)	06	NOPIC USR CON REL LCL NOSHR NOEXE RD NOWRT LONG
DISPATCH_X	000002D4	(212.)	07	NOPIC USR CON REL LCL NOSHR NOEXE RD WRT LONG
GBL_DATA	000002E5	(741.)	08	NOPIC USR CON REL LCL NOSHR NOEXE RD NOWRT LONG
GBL_TEXT	00000049	(73.)	09	NOPIC USR CON REL LCL NOSHR EXE RD WRT LONG
INITIALIZE	00000017	(23.)	0A	NOPIC USR CON REL LCL NOSHR EXE RD WRT LONG
CLEANUP	00000017	(23.)	0B	NOPIC USR CON REL LCL NOSHR EXE RD WRT LONG
SUMMARY	000000CF	(207.)	0C	NOPIC USR CON REL LCL NOSHR EXE RD WRT LONG
SUBROUTINE		(207.)	0D	NOPIC USR CON REL LCL NOSHR EXE RD NOWRT LONG

EVPRG   Diagnostic Program Example   8-AUG-1979 16:09:35   VAX-11 Macro V02.30   Page 26
Cross reference   8-AUG-1979 16:00:34   DB0:[STAPLES]EVPRG.MAR;4   (21)

```
                            +===============================+
                            | Symbol Cross Reference        |
                            +===============================+

SYMBOL        VALUE         DEFINITION    REFERENCES...

$SARGS        =00000011     68    (2)     66    (2)    68    (2)
                                          377   (14)   420   (15)
$SE           =00000001     416   (15)    512   (18)
$SM           =00000004     512   (1A)    233   (8)    240   (14)   #-418  (15)   #-460  (16)
$SN           =00000001     605   (2M)    512   (18)   599   (20)   603    (20)
                                          512   (20)
                                          #-605

$S8           =FFFFFFFF     59    (2)     66    (2)
$ST1          =00000348     68    (2)
$SENV         =00000403     59    (2)     93    (3)
$SM0          =00000002     512   (18)    357   (14)   416   (15)   #-359  (14)   #-512  (18)
$SST          =00000001     610   (21)    610   (21)
$STN          =00000000     420   (15)    377   (14)   420   (15)   512    (18)   512    (16)
                            59    (2)     357   (14)   416   (15)             458  (16)
ACT_ENTRY     =00000000-R   503   (14)    510   (18)
ACT_INIT      =0000003C-R   532   (18)    506   (18)
ACT_NULL      =00000042-R   534   (18)    509   (18)
ACT_STOP      =00000034-R   528   (18)    507   (18)
ACT_SUB1      =00000024-R   520   (18)    508   (18)
ACT_SUB2      =0000002C-R   524   (18)
AL_DEVTYP     =00000029-R   240   (9)     93    (3)
A_HEADEND     =00000058-R   93    (3)     93    (3)
BIT...        =00000005     67    (2)     59    (2)     62     63     64     67    (2)
BIT0          =00000001     61    (2)
BIT1          =00000002     61    (2)
BIT10         =00000400     61    (2)
BIT11         =00000800     61    (2)
BIT12         =00001000     61    (2)
BIT13         =00002000     61    (2)
BIT14         =00004000     61    (2)
BIT15         =00008000     61    (2)
BIT16         =00010000     61    (2)
BIT17         =00020000     61    (2)
BIT18         =00040000     61    (2)
BIT19         =00080000     61    (2)
BIT2          =00000004     61    (2)
BIT20         =00100000     61    (2)
BIT21         =00200000     61    (2)
BIT22         =00400000     61    (2)
BIT23         =00800000     61    (2)
BIT24         =01000000     61    (2)
BIT25         =02000000     61    (2)
BIT26         =04000000     61    (2)
BIT27         =08000000     61    (2)
BIT28         =10000000     61    (2)
BIT29         =20000000     61    (2)
BIT3          =00000008     61    (2)
BIT30         =40000000     61    (2)
BIT31         =80000000     61    (2)
BIT4          =00000010     61    (2)
BIT5          =00000020     61    (2)
```

A-35

EVPRG   Diagnostic Program Example
Cross reference

Symbol	Value	Ref	Ref	Ref
BIT6	=00000040	61	(2)	
BIT7	=00000080	61	(2)	
BIT8	=00000100	61	(2)	
BIT9	=00000200	61	(2)	
CEP_FUNCTIONAL	=00000030	59	(2)	
CEP_REPAIR	=00000001	59	(2)	
CHCS_ABORT	=00000002	62	(2)	
CHCS_CLEAR	=00000006	62	(2)	
CHCS_CLRDFT	=00000009	62	(2)	
CHCS_DSINT	=00000005	62	(2)	
CHCS_ENINT	=00000004	62	(2)	
CHCS_INITA	=00000000	62	(2)	
CHCS_INITB	=00000001	62	(2)	
CHCS_PURGE	=00000003	62	(2)	
CHCS_SETDFT	=00000008	62	(2)	
CHCS_STATUS	=00000007	62	(2)	
CHISM_CHNINT	=00000001	62	(2)	
CHISM_DEVINT	=00000002	62	(2)	
CHISM_IPL	=0000007C	62	(2)	
CHISS_IPL	=00000035	62	(2)	
CHISV_CHNINT	=00000000	62	(2)	62 (2)
CHISV_DEVINT	=00000001	62	(2)	62 (2)
CHISV_IPL	=00000002	62	(2)	62 (2)
CHMS_FORWARD	=00000000	62	(2)	
CHMS_INVALIDATE	=00000001	62	(2)	
CHMS_MAP	=00000002	62	(2)	
CHMS_MFWDN	=0000000A	62	(2)	
CHMS_MFWDNO	=0000000A	62	(2)	
CHMS_MFWDV	=00000003	62	(2)	
CHMS_MFWDVO	=0000000B	62	(2)	
CHMS_MREVN	=00000006	62	(2)	62 (2)
CHMS_MREVNO	=0000000E	62	(2)	62 (2)
CHMS_MREVV	=00000007	62	(2)	
CHMS_MREVVO	=0000000F	62	(2)	
CHMS_NFWDN	=00000000	62	(2)	62 (2)
CHMS_NREVN	=00000008	62	(2)	62 (2)
CHMS_OFFSET	=00000000	62	(2)	
CHMS_REVERSE	=00000004	62	(2)	
CHSSM_BUSIC	=00000000	62	(2)	
CHSSM_BUSINIT	=01000000	62	(2)	
CHSSM_BUSPDN	=00000040	62	(2)	
CHSSM_CHNDPE	=00010002	62	(2)	
CHSSM_CHNERR	=00000002	62	(1)	
CHSSM_CHNMPE	=00000080	62	(2)	
CHSSM_CHPFOT	=00000100	62	(2)	
CHSSM_DEVBUS	=00000010	62	(2)	
CHSSM_DEVERR	=00000004	62	(2)	
CHSSM_DEVTO	=00000020	62	(2)	
CHSSM_ERRANY	=0000000F	62	(2)	
CHSSM_MBAATN	=00100000	62	(2)	
CHSSM_MBACPE	=00200000	62	(2)	
CHSSM_MBADTB	=00040000	62	(2)	
CHSSM_MBADTC	=00080000	62	(2)	
CHSSM_MBAEYC	=00010000	62	(2)	
CHSSM_MBANED	=00020000	62	(2)	
CHSSM_PGMERR	=00000008	62	(2)	
CHSSM_PGMHDE	=00000400	62	(2)	

A-36

EVPRG   Diagnostic Program Example   VAX-11 Macro V02.30
Cross reference   DB0:[STAPLES]EVPRG.MAR;4

8-AUG-1979 16:09:35
8-AUG-1979 16:00:34

Symbol	Value	References
CHSSM_SYSERR	=00000001	62 (2)
CHSSM_SYSMEM	=00000200	62 (2)
CHSSM_SYSSBI	=00000400	62 (2)
CHSSV_BUSIC	=00000F17	62 (2)
CHSSV_BUSINIT	=00000016	62 (2)
CHSSV_BUSPDN	=00000018	62 (2)
CHSSV_CHNDPE	=00000006	62 (2)
CHSSV_CHNERR	=00000001	62 (2)
CHSSV_CHNMPE	=00000007	62 (2)
CHSSV_CHPFOT	=0000000A	62 (2)
CHSSV_DEVBUS	=00000004	62 (2)
CHSSV_DEVERR	=00000002	62 (2)
CHSSV_DEVTO	=00000005	62 (2)
CHSSV_MBAATN	=00000014	62 (2)
CHSSV_MBACPE	=00000015	62 (2)
CHSSV_MBADTB	=00000012	62 (2)
CHSSV_MBADTC	=00000013	62 (2)
CHSSV_MBAEXC	=00000010	62 (2)
CHSSV_MBANED	=00000011	62 (2)
CHSSV_PGMERR	=00000003	62 (2)
CHSSV_PGMHDE	=0000000B	62 (2)
CHSSV_SYSERR	=00000009	62 (2)
CHSSV_SYSMEM	=00000008	62 (2)
CHSSV_SYSSBI	=0000000A	62 (2)
CLEAN_UP	=0000000F=R	416 (15)
CLEAN_UP_X	=0000000F=R	427 (15)
CLISK_ALNUM	=00000087	63 (2)
CLISK_ALPHA	=00000086	63 (2)
CLISK_BIF	=00000083	63 (2)
CLISK_BR	=00000082	63 (2) 174 (6)
CLISK_DEC	=0000008A	63 (2) 214 (7)
CLISK_ERROR	=00000080	63 (2) 187 (6) 201 ( ) 208 (6) 201 (6) 207 (6) 208 (6)
CLISK_EXIT	=00000081	63 (2) 174 (6)
CLISK_HEX	=00000089	63 (2) 194 (6)
CLISK_KEYWORD	=0000008C	63 (2) 214 (7)
CLISK_NUM	=0000008S	63 (2) 174 (6) 186 (6) 187 207 193 208
CLISK_OCT	=0000008B	63 (2) 194 (6) 201 (6)
CLISK_SPACE	=00000084	63 (2) 214 (7)
CLISK_STRING	=00000088	63 (2)
CPUSV_COMET	=00000202	64 (2)
CPUSV_HS	=00000200	64 (2)
CPUSV_STAR	=00000001	64 (2)
DEFAULT	=00000000	233 (4)
DEV_REG	=0000000A=R	126 (5)
DISPATCH	=00010020	105 (4)
DSSABORT	=00010020	513 (18)
DSSASKADR	=00010090	65 (2)
DSSASKDATA	=0001008F	65 (2)
DSSASKLGCL	=00010098	65 (2)
DSSASKSTR	=00010042	65 (2)
DSSASKVLD	=0001003C	65 (2)
DSSBGNSUB	=0001003C	65 (2)
DSSBREAK	=00010058	65 (2) 377 (14) 420 (15) 462 (16)

8-AUG-1979 16:09:35   VAX-11 Macro V02.30
8-AUG-1979 16:30:34   DB0:[STAPLES]EVPRG.MAR;4

EVPRG Cross reference   Diagnostic Program Example

Symbol	Address								
DSSCANWAIT	00010170	65	(2)						
DSSCHANNEL	0001018P	65	(2)						
DSSCKLOOP	C001014C	65	(2)						
DSSCLRVEC	00010168	65	(2)						
DSSCNTRLC	00010178	65	(2)						
DSSCVTREG	0001018B0	65	(2)	593 (22)					
DSSENDPASS	0001001P	65	(2)	369 (14)					
DSSENDSUB	0001013A	65	(2)						
DSSERRDEV	0031010C8	65	(2)						
DSSERRHARD	000101D0	65	(2)						
DSSERRSOFT	000120D8	65	(2)	512 (18)					
DSSERRSYS	000100C0	65	(2)						
DSSESCAPE	00010A50	65	(2)						
DSSGETBUF	00010120	65	(2)						
DSSGETMEM	0001011B	65	(2)	375 (14)					
DSSGPHARD	0001021B	65	(2)						
DSSINITSCB	00010171	65	(2)						
DSSINLOOP	00010048	65	(1)						
DSSLOAD	00010198	65	(2)						
DSSMMOFF	0001215A	65	(2)						
DSSMMON	0001015A	65	(2)						
DSSMOVPHY	00010148	65	(2)						
DSSMOVVRT	0001014C	65	(2)						
DSSPARSE	000102B8	65	(2)						
DSSPRINTB	000100EV	65	(2)	357 (14)	418	460 (16)	584 (19)		
DSSPRINTF	000100F0	65	(2)						
DSSPRINTS	00010AFB	65	(2)	596 (20)	599	603 (20)	605 (20)		
DSSPRINTX	000103E8	65	(2)						
DSSRELBUF	0001012A	65	(2)						
DSSRELMEM	0001013A	65	(2)						
DSSSETIPL	00010178	65	(2)						
DSSSETMAP	00010188	65	(2)						
DSSSETVEC	00013160	65	(2)						
DSSSHOCHAN	00010190	65	(2)						
DSSSUMMARY	00010028	65	(2)						
DSSWAITMS	0001006A	65	(2)						
DSSWAITUS	00010168	65	(2)						
DSASAL_APTMAIL	0000FE80	64	(2)						
DSASAT_APTTXT	0000FAA0	64	(2)						
DSASGL_APTCOM	0000FE44	64	(2)						
DSASGL_DEVLEN	0000FE5A	64	(2)						
DSASGL_ERRNO	0000FE44	64	(2)						
DSASGL_EVENT	0000FE48	64	(2)	364 (14)					
DSASGL_FLAGS	0000FE00	64	(2)						
DSASGL_MSGTYP	0000FE40	64	(2)						
DSASGL_PASSES	0000FE28	64	(2)						
DSASGL_PASSNO	0000FE5A	64	(2)						
DSASGL_SECTNO	0000FE1C	64	(2)						
DSASGL_SID	0000FE14	64	(2)						
DSASGL_SUBTNO	0000FE4C	64	(2)						
DSASGL_TESTNO	0000FE50	64	(2)	g-366 (14)					
DSASGL_UNITS	0000FE0C	64	(2)						
DSASGQ_MSGPTR	0000FE68	64	(2)						
DSASGT_DEVNAM	0000FE5C	64	(2)						
DSASM_APT	=80000000	64	(2)						
DSASM_BELL	=00000008	64	(2)						
DSASM_COMPAT	=04000000	64	(2)						

EVPRG   Diagnostic Program Example
Cross reference

Symbol	Value	Ref	Count			
DSASM_ENSPEC	=40000000	64	(2)			
DSASM_HALTD	=00000001	64	(2)			
DSASM_HALTI	=00000002	64	(2)			
DSASM_IE1	=00000010	64	(2)			
DSASM_IE2	=00000020	64	(2)			
DSASM_IE3	=00000040	64	(7)			
DSASM_IE5	=00000080	64	(2)			
DSASM_LOCK	=00000800	64	(2)			
DSASM_LOOP	=00000004	64	(2)			
DSASM_NORPT	=08000000	64	(2)			
DSASM_OPER	=00001000	64	(2)			
DSASM_PASS0	=20000000	64	(2)			
DSASM_PROMPT	=00000200	64	(2)			
DSASM_QUICK	=00000100	64	(2)			
DSASM_SPOOL	=00000400	64	(2)			
DSASM_TRACE	=00000040	64	(2)			
DSASM_USER	=10000000	64	(2)			
DSASV_APT	=0000001F	64	(2)			
DSASV_BELL	=00000003	64	(2)			
DSASV_COMPAT	=0000001A	64	(2)			
DSASV_ENSPEC	=0000001E	64	(2)			
DSASV_HALTD	=00000000	64	(2)			
DSASV_HALTI	=00000001	64	(2)			
DSASV_IE1	=00000004	64	(2)			
DSASV_IE2	=00000005	64	(2)			
DSASV_IE3	=00000006	64	(2)			
DSASV_IE5	=00000007	64	(2)			
DSASV_LOCK	=0000000B	64	(2)			
DSASV_LOOP	=00000002	64	(2)			
DSASV_NORPT	=0000001B	64	(2)			
DSASV_OPER	=0000000C	64	(2)	#-364	(14)	
DSASV_PASS0	=0000001D	64	(2)			
DSASV_PROMPT	=00000009	64	(2)			
DSASV_QUICK	=00000008	64	(2)			
DSASV_SPOOL	=0000000A	64	(2)			
DSASV_TRACE	=0000000A	64	(2)			
DSASV_USER	=0000001C	64	(2)			
ENVSM_DOMAIN	=00000002	59	(2)			
ENVSM_LEVEL	=00000001	59	(2)			
ENVSM_SUPER	=000003FC	59	(2)			
ENVSS_DOMAIN	=00000001	59	(2)			
ENVSS_LEVEL	=00000001	59	(2)			
ENVSS_SUPER	=00000008	59	(2)			
ENVSV_DOMAIN	=00000001	59	(2)	59	(2)	
ENVSV_LEVEL	=00000002	59	(2)	59	(3)	
ENVSV_SUPER	=00000002	59	(2)	93	(2)	
ENVS_CPU	=00000007	59	(2)	59	(2)	
ENVS_FUNCTIONAL	=00000007	59	(2)	59	(3)	
ENVS_REPAIR	=00000001	59	(2)	59	(2)	
ENVS_SUPER	=00000001	59	(2)			
ENVS_SYSTEM	=00000001	59	(2)			
ERRS_MSGADR	=0000000C	66	(2)			
ERRS_NARGS	=0000000A	66	(2)			
ERRS_NUM	=00000004	66	(2)			
ERRS_P1	=00000014	66	(2)	#-580	(19)	
ERRS_P2	=00000018	66	(2)	#-581	(19)	
ERRS_P3	=0000001C	66	(2)	#-582	(19)	

VAX-11 Macro V02.30
DB0:[STAPLES]EVPRG.MAR;4

8-AUG-1979 16:39:35
8-AUG-1979 16:39:34

EVPRG
Cross reference

Diagnostic Program Example

Symbol	Value	References
ERRS_P4	=00000020	66 (2) #-583 (19)
ERRS_P5	=00000024	66 (2)
ERRS_P6	=00000028	66 (2)
ERRS_POINTER	=00000010	66 (2)
ERRS_UNIT	=0000000B	66 (2)
GBL...	=0000000B	67 (2)
GD_A_CSR	0000000C-R	131 (5) 59 (2) 61 (2) 62 (2) 63 (2) 64 (2)
GD_A_PTBL	00000004-R	123 (5) 05 (2)
GD_A_XOR	00000004-R	133 (5)
GD_K_CMDSIZ	=00000008	141 (5)
GD_K_STRSIZ	=00000084	145 (5)
GD_L_LUN	00000000-R	121 (5) 375 (14)
GD_L_TOKEN	00000010-R	136 (5) #-561 (14) #-525 (18) #-529 (18) #-533 (18)
GD_T_CMDBUF	00000018-R	139 (5) #-521 (18)
GD_T_COMMAND	00000044-R	165 (6) 141 (5)
GD_T_STRBUF	00000020-R	143 (5)
GD_W_CSR	00000038-R	127 (5) 145 (5) 593 (5)
GD_W_XOR	0000003A-R	129 (5)
GT_T_AMBCMD	00000142-R	276 (11)
GT_T_CLEANEXE	00000299-R	317 (12) 418 (15)
GT_T_CSRERR	0000021A-R	292 (12)
GT_T_CSRFUNC	0000016D-R	262 (11)
GT_T_CVT16	00000118-R	256 (11) 593 (20) 593 (5)
GT_T_CVT32	00000065-R	250 (11)
GT_T_CVTCSR	00000041-R	247 (11)
GT_T_FMTCR	00000272-R	307 (12) 645 (20)
GT_T_FMTEXP	0000022A-R	301 (12) 596 (20)
GT_T_FMTRCV	0000023F-R	303 (12) 599 (20)
GT_T_FMTXOR	00000254-R	305 (12) 603 (20)
GT_T_GETLINE	00000196-R	274 (11)
GT_T_INTEXE	00000275-R	315 (12) 359 (14)
GT_T_INVCMD	000001B7-R	278 (11)
GT_T_NOERR	00000201-R	290 (11)
GT_T_REGERR	000001EB-R	288 (11) 463 (16)
GT_T_SUMMARYEXE	0000022A-R	319 (12)
GT_T_TI92	000001CC-R	280 (11)
G_K_INIT	=00000004	78 (2)
G_K_NOP	=00000000	74 (2) 174 (6) 193 (6) 194 (6)
G_K_STOP	=00000003	77 (2) 182 (6) #-529 (18)
G_K_SUB1	=00000001	75 (2) 186 (6) #-521 (18)
G_K_SUB2	=00000002	76 (2) 236 (6) #-525 (18)
INITIALIZE	00000100-R	357 (14) 257 (6)
INITIALIZE_X		377 (14) 93 (3)
LSA_CCP	0000001C-R	93 (3)
LSA_DEVP	00000024-R	93 (3)
LSA_DREG	00000018-R	93 (3)
LSA_DTP	0000003C-R	93 (3)
LSA_ICP	00000014-R	93 (3)
LSA_LASTAD	00000008-R	93 (3)
LSA_NAME	00000044-R	93 (3)
LSA_REPP	00000050-R	93 (3)
LSA_SECNAM	00000048-R	93 (3)
LSA_STATAB	00000004-R	93 (3)
LSA_TSTCNT	00000054-R	93 (3)
LSL_ENVIRON	00000044-R	93 (3)
LSL_ERRTYP	0000004C-R	93 (3)

A-40

8-AUG-1979 16:49:35   VAX-11 Macro V02.30
8-AUG-1979 16:00:34   DB0:[STAPLES]EVPRG.MAR;4

EVPRG   Diagnostic Program Example
Cross reference

Symbol	Value	Ref	Ref	Ref
LSL_HEADLENGTH	00000000C-R	93	(3)	
LSL_REV	00000000C-R	93	(3)	
LSL_UNIT	00000020-R	93	(3)	
LSL_UPDATE	00000010-R	93	(3)	
LASTAD	00000000-R	93	(3)	
L_TSTCNT	00000043-R	93	(19)	
MSG_REGERR	=00000010	577	(2)	
PAR$M_ATDEF	=00000008	67	(2)	
PARSM_ATHI	=00000004	67	(2)	
PARSM_ATLO	=00000002	67	(2)	
PARSM_NODEF	=00000001	67	(2)	
PARSV_ATDEF	=00000003	67	(2)	
PARSV_ATHI	=00000002	67	(2)	
PARSV_ATLO	=00000001	67	(2)	
PARSV_NODEF	=00000000	67	(2)	
PARS_BIN	=00000002	67	(2)	
PARS_DEC	=0000000A	67	(2)	
PARS_HEX	=00000010	67	(2)	
PARS_NO	=00000000	67	(2)	
PARS_OCT	=00000008	67	(2)	
PARS_YES	=00000001	67	(2)	
PRINTXS_FORMAT	=00000004	68	(2)	
PRINTXS_NARGS	=00000011	68	(2)	
PRINTXS_P0	=00000008	68	(2)	
PRINTXS_P1	=0000000C	68	(2)	
PRINTXS_P2	=00000010	68	(2)	
PRINTXS_P3	=00000014	68	(2)	
PRINTXS_P4	=00000018	68	(2)	
PRINTXS_P5	=0000001C	68	(2)	
PRINTXS_P6	=00000020	68	(2)	
PRINTXS_P7	=00000024	68	(2)	
PRINTXS_P8	=00000028	68	(2)	
PRINTXS_P9	=0000002C	68	(2)	
PRINTXS_PA	=00000030	68	(2)	
PRINTXS_PB	=00000034	68	(2)	
PRINTXS_PC	=00000038	68	(2)	
PRINTXS_PD	=0000003C	68	(2)	
PRINTXS_PE	=00000040	68	(2)	
PRINTXS_PF	=00000044	68	(2)	
RK07	=00000003	240	(6)	93 (3)
RM03	=00000002	240	(9)	
RP06	=00000001	240	(9)	
SECTION	0000000A-R	233	(8)	
SEP_FUNCTIONAL	=00000002	59	(2)	93 (3)
SEP_REPAIR	=00000003	59	(5)	59 (2)
STATISTIC	00000044-R	151	(5)	93 (3)
SUMMARY_X	00000000F-R	458	(16)	
SYSSALLOC	0000000F-R	462	(16)	
SYS$ASCTIM	00010234R	65	(2)	
SYS$ASSIGN	00010244R	65	(2)	
SYS$BINTIM	0001025AR	65	(2)	
SYS$CANCEL	0001025AR	65	(2)	
SYS$CANTIM	00010268R	65	(2)	
SYS$CLREF	0001029AR	65	(2)	
SYS$DALLOC	000102DAR	65	(2)	
SYS$DASSGN	000102EAR	65	(2)	

EVPRG   Diagnostic Program Example
Cross reference

Symbol	Value	Ref		Ref	
SYS$FAO	00010350	65	(2)		
SYS$FAOL	00010358	65	(2)		
SYS$GETCHN	000104C8	65	(2)		
SYS$GETTIM	00010378	65	(2)		
SYS$NUMTIM	000103B8	65	(2)		
SYS$QIO	000103C8	65	(2)		
SYS$QIOW	00010200	65	(2)		
SYS$READEF	000103D0	65	(2)		
SYS$SETEF	00010400	65	(2)		
SYS$SETIMR	00010420	65	(2)		
SYS$SETPRT	00010438	65	(2)		
SYS$UNWIND	00010470	65	(2)		
SYS$WAITFR	00010478	65	(2)		
SYS$WFLAND	00010488	65	(2)		
SYS$WFLOR	00010490	65	(2)		
S_DEFAULT	*0000000-R	233	(8)	233	(8)
TE16	*00000004	240	(9)		
TU77	*00000005	93	(3)	93	(3)
T_NAME	0000005A-R	93	(3)	261	(10)
T_NOP	00000181-R	263	(12)	261	(10)
T_READ	00000185-R	264	(11)	240	(9)
T_RK07	0000001A-R	240	(9)	240	(9)
T_RM03	00000015-R	240	(9)	261	(10)
T_RP06	00000010-R	240	(9)	240	(9)
T_RST	00000190-R	266	(12)	240	(9)
T_TE16	0000001F-R	240	(9)	261	(10)
T_TU77	00000024-R	240	(9)		
T_WRT	0000001A-R	265	(11)		

EVPRG   Diagnostic Program Example   8-AUG-1979 16:09:35   VAX-11 Macro V02.30   Page 34 (21)
Cross reference   8-AUG-1979 16:00:34   DB01:[STAPLES]EVPRG.MAR;4

```
+----------------------------+
| Macro Cross Reference      |
+----------------------------+
```

MACRO	SIZE	DEFINITION	REFERENCES...
SD1_BPROGRAM	1	513 (18)	513 (18)
SD1_ERR	1	512 (18)	512 (18)
SD1_ERR_S	2	512 (18)	512 (18)
SD1_PRINT_S	2	359 (14)	359 (14) 418 (15) 460 (16) 584 (19) 596 (20) 599 (20) 603 (20) 605 (20)
SD1_SEC	2	233 (8)	233 (8)
SDEF	1	64 (2)	64 (2)
SDEFEND	1	64 (2)	64 (2)
SDEFINI	1	64 (2)	64 (2)
SD8_ABORT	1	513 (18)	513 (18)
SD8_BGNCLEAN	1	416 (15)	416 (15)
SD8_BGNINIT	1	357 (14)	357 (14)
SD8_BGNMESSAGE	1	578 (19)	578 (19)
SD8_BGNMOD	1	59 (2)	59 (2)
SD8_BGNSTAT	1	151 (5)	151 (5)
SD8_BGNSUMMARY	1	458 (16)	458 (16)
SD8_BITDEF	2	61 (2)	61 (2)
SD8_BPASS0	1	364 (14)	364 (14)
SD8_BREAK	1	377 (14)	377 (14) 420 (15) 462 (16)
SD8_CHCDEF	1	62 (2)	62 (2)
SD8_CHDEF	1	62 (2)	62 (2)
SD8_CHIDEF	1	62 (2)	62 (2)
SD8_CHMDEF	2	62 (2)	62 (2)
SD8_CHSDEF	3	62 (2)	62 (2)
SD8_CLI	1	174 (6)	174 (6) 180 (6) 186 (6) 187 (6) 193 (6) 194 (6) 200 (6) 201 (6) 207 (6) 208 (6) 214 (7)
SD8_CLIDEF	1	63 (2)	63 (2)
SD8_CVTREG_S	2	588 (20)	588 (20)
SD8_DEVTYP	1	240 (9)	240 (9)
SD8_DISPATCH	1	105 (4)	105 (4)
SD8_DSADEF	1	64 (2)	64 (2)
SD8_DSBDEF	1	64 (2)	65 (2)
SD8_DSSDEF	1	65 (2)	65 (2)
SD8_ENDCLEAN	1	420 (15)	420 (15)
SD8_ENDINIT	1	377 (14)	377 (14)
SD8_ENDMESSAGE	1	608 (20)	608 (20)
SD8_ENDMOD	1	610 (21)	610 (21)
SD8_ENDPASS_G	1	369 (14)	369 (14)
SD8_ENDSTAT	1	152 (5)	152 (5)
SD8_ENDSUMMARY	1	462 (16)	462 (16)
SD8_ENVDEF	2	59 (2)	59 (2)
SD8_ERRDEF	1	66 (2)	66 (2)
SD8_ERRSYS_S	1	512 (18)	512 (18)
SD8_GPHARD_S	3	374 (14)	374 (14)
SD8_HEADER	1	90 (3)	90 (3)
SD8_PARDEF	1	67 (2)	67 (2)
SD8_PRINTB_S	2	359 (14)	359 (14) 418 (15) 460 (16) 584 (19) 605 (20)
SD8_PRINTX_DEF	1	68 (2)	68 (2)
SD8_PRINTX_S	2	595 (20)	595 (20) 598 (20) 601 (20)

```
EVPRG    Diagnostic Program Example          8-AUG-1979 16:09:35   VAX-11 Macro V02.30      Page 35 (21)
Cross reference                              8-AUG-1979 16:00:34   DB0:[STAPLES]EVPRG.MAR;4
```

Symbol									
SDS_SECTION	2	233 (8)	233 (8)	61 (2)	62	62 (2)	63	63 (2)	64	64 (2)
SEGU	1	59 (2)	59 (2)						
SEGULS1	1	59 (2)	59 (2)	62 (2)	63	63 (2)			
SEGULST	1	59 (2)	59 (2)	62 (2)	63	63 (2)			
SEGLIN1	1	59 (2)	59 (2)	61 (2)	62	62 (2)			
SOFFDEF	1	66 (2)	66 (2)	67 (2)	68				
SPUSHADR	1	512 (18)	512 (18)	593 (20)					
SVIELD	1	59 (2)	59 (2)	62 (2)	64	64 (2)	67 (2)		
SVIELD1	2	59 (2)	65 (2)	62 (2)	64	64 (2)	67 (2)		

```
+======================+
| Performance Indicators |
+======================+
```

Phase	Page faults	CPU Time	Elapsed Time
Initialization	6	00:00:00.02	00:00:00.99
Command processing	19	00:00:00.17	00:00:01.14
Pass 1	7129	00:00:22.21	00:01:39.73
Symbol table sort	21	00:00:00.54	00:00:02.18
Pass 2	823	00:00:24.30	00:00:26.25
Symbol table output	39	00:00:00.35	00:00:03.05
Psect synopsis output	8	00:00:00.06	00:00:00.28
Cross-reference output	392	00:00:01.74	00:00:10.91
Assembler run totals	8446	00:00:29.41	00:02:25.41

The working set limit was 150 pages.
184642 bytes (361 pages) of virtual memory were used to buffer the intermediate code.
There were 30 pages of symbol table space allocated to hold 413 non-local and 11 local symbols.
611 source lines were read in Pass 1, producing 37 object records in Pass 2.
64 pages of virtual memory were used to define 31 macros.

```
+======================+
| Macro library statistics |
+======================+
```

Macro library name	Index size (pages)	Macros defined
DRA2:[SYSLIB]DIAG.MLB;574	15	44
DRA2:[SYSLIB]STARLET.MLB;10	22	11
TOTALS (all libraries)	37	55

750 GETS were required to define 55 macros.

There were no errors or warnings.

/LIS/CROSS EVPRG

8-AUG-1979 16:12:06   VAX-11 Macro V02.30

EVPRG1   Diagnostic Program Example
Table of contents

EVPRG1
V1.0

Diagnostic Program Example

```
                                        8-AUG-1979 16:12:06   VAX-11 Macro V02.30
                                        8-AUG-1979 13:22:58   DB0:[STAPLES]EVPRG1.MAR;1

 1              .TITLE FVPRG1  Diagnostic Program Example
 2              .IDENT /V1.0/
 3
 4
 5      ;   Copyright (C) 1979
 6      ;   Digital Equipment Corporation, Maynard, Massachusetts 01754
 7      ;
 8      ;   This software is furnished under a license for use only on a single
 9      ;   computer system and may be copied only with the inclusion of the
10      ;   above copyright notice. This software, or any other copies thereof,
11      ;   may not be provided or otherwise made available to any other person
12      ;   except for use on such system and to one who agrees to these license
13      ;   terms. Title to and ownership of the software shall at all times
14      ;   remain in DEC.
15      ;
16      ;   The information in this software is subject to change without notice
17      ;   and should not be construed as a commitment by Digital Equipment
18      ;   Corporation.
19      ;
20      ;   DEC assumes no responsibility for the use or reliability of its
21      ;   software on equipment which is not supplied by DEC.
22      ;
23      ;
24      ;++
25      ;  Facility:    VAX Diagnostic System
26      ;
27      ;  Abstract:    The program consists of two subtests in one test.
28      ;               The subtests are executed via commands by the user.
29      ;               The main test routine uses the command parser in
30      ;               conjunction with a command syntax tree.
31      ;
32      ;  Environment: VAX Diagnostic Supervisor
33      ;
34      ;  Author:   TED BEAR      8-Aug-79        Version V1.0
35      ;
36      ;  Modified by:
37      ;--
```

EVPRG1   Diagnostic Program Example   8-AUG-1979 16:12:06   VAX-11 Macro V02.30   Page 2
V1.0   Declarations   8-AUG-1979 13:22:58   DB0:[STAPLES]EVPRG1.MAR;1   (2)

```
   39                    .SBTTL  Declarations
   40            ;+
   41            ; Include files:
   42            ;-
   43
   44                    .LIBRARY  \SYS$LIBRARY:DIAG\        ; VAX Diagnostic Macro Library
   45
   46            ;+
   47            ; Macros:
   48            ;-
   49
   50            ;+
   51            ; Equated symbols:
   52            ;-
   53
   54            ;+
   55            ; Create symbols for the program/supervisor interface.
   56            ;-
   57                    $DS_BGNMOD  <SEP_REPAIR>          ; Level 3, system environment
   58                    DIAGNOSTIC MACRO LIBRARY V5.0+ "DIAG.MLB (574)"
                 ;;
   59                    $DS_CHDEF                         ; channel service symbols
   60                    $DS_CLIDEF                        ; syntax tree symbols & literals
   61                    $DS_DSSDEF                        ; supervisor service entry vectors
   62
   63            ;+
   64            ; Own storage:
   65            ;-
   66
   67
   68                    $DS_PAGE  <1>
   69
```

EVPRG1
V1.0

Diagnostic Program Example
TEST 1: Conversation test

8-AUG-1979 16:12:06   VAX-11 Macro V02.30   Page 3 (3)
8-AUG-1979 13:22:58   DB0:[STAPLES]EVPRG1.MAR;1

```
                    .SBTTL  TEST 1:  Conversation test
                    .PSECT  TEST_001, PAGE, NOWRT

72      00000         ;++
73      00000060      ; Test description:
74      00014         ;
75      00014         ; This test consists of three subtests.  The subtests are conditionally
76      00014         ; executed via a conversational command interaction with the user.
77      00014         ;
78      00014         ; An ambiguous or invalid command will cause the user to be notified
79      00014         ; that his command was unacceptable and the prompt will be re-issued.
80      00014         ;
81      00014         ; Calling sequence:
82      00014         ;
83      00014         ; The diagnostic supervisor calls this routine with a CALL instruction.
84      00014         ;
85      00014         ; Input parameters:
86      00014         ;
87      00014         ;       None
88      00014         ;
89      00014         ; Implicit inputs:
90      00014         ;
91      00014         ;       None
92      00014         ;
93      00014         ; Output parameters:
94      00014         ;
95      00014         ;       None
96      00014         ;
97      00014         ; Implicit outputs:
98      00014         ;
99      00014         ;       None
100     00014         ;
101     00014         ; Completion codes:
102     00014         ;
103     00014         ;       None
104     00014         ;
105     00014         ; Side effects:
106     00014         ;
107     00014         ;       None
108     00014         ;--
```

A-48

```
EVPRG1                Diagnostic Program Example                     8-AUG-1979 16:12:06   VAX-11 Macro V02.30        Page  4
V1.0                  TEST 1: Conversation test                      8-AUG-1979 13:22:58   DB0:[STAPLES]EVPRG1.MAR;1        (4)

                                                110    DATA_001:
           00000000     0014  0014                   .LONG    0                     ; TEST ARGUMENT TABLE TERMINATOR
                                                111    TEST_001::
                00000   0018  0018                   .WORD    ^M<>                   ; ENTRY MASK
                                                112    T1_GETCMD:
                                                113    10$:   SDS_ASKSTR_S
                                                                MSGADR = GT_T_GETLINE,-   ; ask operator for a command
                                                                BUFADR = GD_T_STRBUF,-    ; prompt line
                                                                DEFADR = 0                ; command storage
              00 DD      001A  001A              114           PUSHL  #0               ; there is no default
              00 DD      001C  001E             115           PUSHL  #0
              00 DD      0020                   116           PUSHL  #0
              8F DD      0026                                 PUSHL  #72
  00000048'EF DF         002C                                 PUSHAL GD_T_STRBUF
  00000000'EF DF         0032                                 PUSHAL GT_T_GETLINE
           06 FB         0039                                 CALLS  #6, @#DSS_ASKSTR
00010A0 9F    DE         0039                   117    SDS_BNCOMPLETE 10$             ; if at first you don't succeed
           50 DE         003C                                 BLBC   R0, 10$
  00000000'EF 9A         003C                   118    MOVZBL GD_T_STRBUF,            ; command string length
  00000001'EF DE         0047                   119    MOVAL  GD_T_CMDBUF,
                         0047                   120           GD_T_STRBUF+1,          ; build quadword descriptor
                         0052                   121           GD_T_CMDBUF+4
                         0052                   122    SDS_PARSE_S                    ; parse the command
  00000000'EF DF         0052                   123           BUFADR = GD_T_CMDBUF,
  00000000'EF DF         0052                   124           TREE = GD_T_COMMAND,
  00000000'EF 7F         0058                                 ACTION = ACT_ENTRY
           03 FB         005E                   125           PUSHAL ACT_ENTRY
                         0064                   126           PUSHAL GD_T_COMMAND
                                                              PUSHAQ GD_T_CMDBUF
                                                              CALLS  #3, @#DSS_PARSE
00010B0 9F    50         006B                   127    SDS_BCOMPLETE 20$             ; the command has been parsed
           0F            006E                                 BLBS   R0, 20$
  00000000'EF           006E                   128    SDS_PRINTF_S
           01 FB         0074                   129           FORMAT = GT_T_INVCMD    ; invalid command
00010F0 9F    9D         0078                                 GT_T_INVCMD
                                                130           PUSHAB #SSN, @#DSSPRINTF
                                                              CALLS  10$
                                                131    BRB                           ; get another command
```

```
EVPRG1        Diagnostic Program Example              8-AUG-1979 16:12:06   VAX-11 Macro V02.30    Page  5
V1.0          TEST 1: Conversation test              8-AUG-1979 13:22:58   DB0:[STAPLES]EVPRG1.MAR;1   (5)

00000000'EF  DS   133  29$:  TSTL    GD_L_TOKEN          ; is token block empty?
         10  12   134        BNEQ    30$                 ; no, a valid command was typed
                  135
                  136        $DS_PRINTF_S
                  137          FORMAT = GT_T_AMBCMD   * ; ambiguous command
                                      PUSHAB  GT_T_AMBCMD
                                      CALLS   #$$N, e#DS$PRINTF

00000000'EF  9F   138
         01  FB   139  30$:  BRW     10$                 ; get another command
000100F0 9F  31
      FF85
```

```
EVPRG1          Diagnostic Program Example            8-AUG-1979 16:12:06    VAX-11 Macro V02.30
V1.0            TEST 1:  Conversation test            8-AUG-1979 13:22:58    DB0:[STAPLES]EVPRG1.MAR;1

                    0095    141   ;+
                    0095    142   ; Subtest description:
                    0095    143   ;
                    0095    144   ;   *** Brief description of what the subtest checks. ***
                    0095    145   ;
                    0095    146   ; Subtest steps:
                    0095    147   ;
                    0095    148   ;   *** Detailed flow of the testing sequence. ***
                    0095    149   ;
                    0095    150   ; Errors:
                    0095    151   ;
                    0095    152   ;   *** Brief description of each of the possible errors detected. ***
                    0095    153   ;
                    0095    154   ; Debug:
                    0095    155   ;
                    0095    156   ;   *** Helpful hints for tracking the hardware faults. ***
                    0095    157   ;-
                    0095    158
                    0095    159   T1_S1::       CALLG   $$$, #^DS$BGNSUB
00010030 9F    FA   0095          SDS_BGNSUB
00000000'EF    D1   00A0    160              CMPL    GD_L_TOKEN, #G_K_SUB1   ; check command code
               13   00A8    161              BEQL    30$                     ; proceed if correct
        0068    31   00AD    162              BRW     T1_S1_X                 ; otherwise, skip out of here
                    00B0    163
     5068 8F   80   00B0    164   30$:        MOVH    #^X5068, GD_W_CSR       ; expected data
00000000'EF    84   00B9    165              CLRW    GD_W_CSR                ; cause failure for now
     5068 8F   AD   00BF    166              XORW3   #^X5068, GD_W_CSR,      ; check
00000000'EF         00C8    167                      GD_W_XOR
               13   00CD    168              BEQL    40$
         35         00CD    169
                    00CF    170   SDS_ERRHARD_$
                    00CF    171              UNIT = GD_L_LUN,
                    00CF    172              MSGADR = GT_T_REGERR,       ; error message
                    00CF    173              PRLINK = MSG_REGERR,        ; extended print routine
                    00CF    174              P1 = #GT_T_CSRERR,          ; extended message
                    00CF    175              P2 = #GD_W_CSR,
                    00CF    176              P3 = #^X5068,               ; expected data
                    00CF    177              P4 = GD_W_CSR               ; received data
                    00CF    178
00000000'EF    DD   00CF              PUSHL   GD_W_CSR
0000505068 8F  DD   00D5              PUSHL   #^X5068
00000000'EF    DD   00DB              PUSHL   #GD_W_CSR
00000000'EF    DF   00E1              PUSHL   #GT_T_CSRERR
00000000'EF    DF   00E7              PUSHAL  MSG_REGERR
00000000'EF    DD   00ED              PUSHAL  GT_T_REGERR
         01    DD   00F3              PUSHL   GD_L_LUN
                    00F9              PUSHL   #SER
                    00FB   ;)) TEST 1, SUBTEST 1, ERROR 1
00010000 9F    08   00FB              CALLS   #SSM, #^DS$ERRHARD
               17   0102    179              BRB     T1_S1_X             ; skip out
```

```
EVPRG1           Diagnostic Program Example        8-AUG-1979 16:12:06   VAX-11 Macro V02.30
V1.0             TEST 1: Conversation test         8-AUG-1979 13:22:58   DB0:[STAPLES]EVPRG1.MAR;1

                              181  05B:   $DS_ERRHARD_S
                                          UNIT = GD_L_LUN,                 ; ; no error message
                                          MSGADR = GT_T_NOERR              ; ;
                          02  DD   0160    182  PUSHL   #0                 ; ;
              00000000'EF  DF   0166    183  PUSHAL  GT_T_NOERR
              00000000'EF  DD   016C         PUSHL   GD_L_LUN
                          02  DD   0112         PUSHL   #SER
                                                CALLS   #SSM, @#DSSERRHARD
                                          ; ;; TEST 1, SUBTEST 1, ERROR 2
00010000  9F   04  FB  0114    184          $DS_ENDSUB
                              0116    185  T1_S1_X:
                                  0119
00010038  9F  00000000'EF  FA  011A          CALLG   $SS, @#DSSENDSUB
```

```
EVPRG1        Diagnostic Program Example                    9-AUG-1979 16:12:06   VAX-11 Macro V02.30        Page 8
V1.0          TEST 1: Conversation test                     8-AUG-1979 13:22:58   DB0:[STAPLES]EVPRG1.MAR;1       (8)

                                          167  ;+
                                          168  ; Subtest description:
                                          180  ;
                                          190  ;   *** Brief description of what the subtest checks. ***
                                          191  ;
                                          192  ; Subtest steps:
                                          193  ;
                                          194  ;   *** Detailed flow of the testing sequence. ***
                                          195  ;
                                          196  ; Errors:
                                          197  ;
                                          198  ;   *** Brief description of each of the possible errors detected. ***
                                          199  ;
                                          200  ; Debug:
                                          201  ;
                                          202  ;   *** Helpful hints for tracking the hardware faults. ***
                                          203  ;
                                          204  ;-
             0126                         205  T1_S2::       CALLG   $SS, e#DS$BGNSUB
00010030 9F  0126  FA                     206
0000000'EF   0131  D1                     207            CMPL    GD_L_TOKEN, #G_K_SUB2    ; check command code
         17  0131  12                     208            BNEQ    T1_S2_X                  ; and skip out if not correct
             013C                         209
             013F                         210  $DS_ERRHARD_S       -                      ; report an error
         00  013E                         211            UNIT = GD_L_LUN,      -          ; lun
0000000'EF   013E  DD                     212            MSGADR = GT_I_T1S2               ; message pointer
0000000'EF   0146  DF                              PUSHL   #0
         01  014C  DD                              PUSHAL  GT_I_T1S2
             014C  DD                              PUSHL   GD_L_LUN
                                                   PUSHL   #$ER
00010000 9F  014E  04                              CALLS   #$$M, e#DS$ERRHARD
                                          ;;; TEST 1, SUBTEST 2, ERROR 1
             0155                         213  T1_S2_X:      $DS_ENDSUB
00010038 9F  0155  FA                     214                CALLG   $SS, e#DS$ENDSUB
             0155
```

A-53

```
EVPRG1       Diagnostic Program Example                      8-AUG-1979 16:12:06   VAX-11 Macro V02.30                    Page  9
V1.0         TEST 1: Conversation test                       8-AUG-1979 13:22:58   DB0:[STAPLES]EVPRG1.MAR;1                   (9)

                  216  0160             ;+
                  217  0160             ; Subtest description:
                  218  0160             ;
                  219  0160             ;   *** Brief description of what the subtest checks. ***
                  220  0160             ;
                  221  0160             ; Subtest steps:
                  222  0160             ;
                  223  0160             ;   *** Detailed flow of the testing sequence. ***
                  224  0160             ;
                  225  0160             ; Errors:
                  226  0160             ;
                  227  0160             ;   *** Brief description of each of the possible errors detected. ***
                  228  0160             ;
                  229  0160             ; Debug:
                  230  0160             ;
                  231  0160             ;   *** Helpful hints for tracking the hardware faults. ***
                  232  0160             ;
                  233  0160             ;-
                  234  0160             $DS_BGNSUB
                                        T1_S3::
00010030 9F  00010030'EF   FA  0160     235        CALLG   $SS, ##DS$BGNSUB
00000000'8F  00000000'EF   01  0168     236        CMPL    GD_L_TOKEN, #G_K_STOP    ; check command code
                 32        12  016E     237        BNEQ    T1_S3_X                  ; and skip out if not correct
                           11  0176     238
                           11  0178     239        BRB     TEST_001_X               ; user commanded an end to this
                               017A     240        $DS_ENDSUB
00010030 9F  00010030'EF   FA  017A     241  T1_S3_X:    CALLG   $SS, ##DS$ENDSUB
```

```
EVPRG1    Diagnostic Program Exercise                8-AUG-1979 16:12:06   VAX-11 Macro V02.30      Page 10 (10)
V1.0      TEST 1: Conversation test                  8-AUG-1979 13:22:58   DB0:[STAPLES]EVPRG1.MAR;1

            FE92 31  0185  243      BRW      T1_GETCMD         ; last command executed, get another
                     0188  244
                     0188  245      $DS_ENDTEST
            01       018A           TEST_001_X::
            50       018B           MOVL     #1, R0            ; NORMAL EXIT
            6E FA    0185  246      CALLG    (SP), @#DS$BREAK
               04    0192  247      RET                        ; RETURN TO TEST SEQUENCER
                     0193           $DS_ENDMOD
            00000000                .PSECT   $TSTCNT, NOEXE, NOWRT, OVR, LONG
00010058 9F 00000001                .LONG    $TN
                           248      .END
```

```
EVPRG1   Diagnostic Program Example        8-AUG-1979 16:12:06   VAX-11 Macro V02.38
Symbol table                                8-AUG-1979 13:22:58   DBB1[STAPLES]EVPRG1.MAR;1
```

Symbol names (left column):

SS$
SSE
SSM
SSN
SSU
SENV
SER
SMO
SSB
SST
STN
ACT_ENTRY
BASE_001
BIT...
CEP_FUNCTIONAL
CEP_REPAIR
CHCS_ABORT
CHCS_CLEAR
CHCS_CLRDFT
CHCS_DSINT
CHCS_ENINT
CHCS_INITA
CHCS_INITB
CHCS_PURGE
CHCS_SETDFT
CHCS_STATUS
CHISM_CHNINT
CHISM_DEVINT
CHISS_IPL
CHISS_IPL
CHISV_CHNINT
CHISV_DEVINT
CHISV_IPL
CHMS_FORWARD
CHMS_INVALIDATE
CHMS_MAP
CHMS_MFWDN
CHMS_MFWDNO
CHMS_MFWDV
CHMS_MFWDVO
CHMS_MREVN
CHMS_MREVNO
CHMS_MREVV
CHMS_MREVVO
CHMS_NFWDN
CHMS_NREVN
CHMS_OFFSET
CHMS_REVERSE
CHSM_BUSIC
CHSM_BUSINIT
CHSM_BUSPDN
CHSM_CHNDPE
CHSM_CHNERR
CHSM_CHNMPE
CHSM_CHPFOT
CHSM_DEVBUS
CHSSM_DEVERR

Symbol names (middle column):

CHSSM_DEVTO
CHS..ERRANY
CHSM_MBAATN
CHSM_MBACPE
CHSM_MBADTR
CHSM_MBADTC
CHSM_MBAEXC
CHSM_MBANED
CHSM_PGMERR
CHSM_PGMHDE
CHSM_SYSMEM
CHSM_SYSSBI
CHSV_BUSIC
CHSV_BUSINIT
CHSV_HUSPDN
CHSV_CHNDPE
CHSV_CHNERR
CHSV_CHNMPE
CHSV_CHPFOT
CHSV_DEVBUS
CHSV_DEVERR
CHSV_DEVTO
CHSV_MBAATN
CHSV_MBACPE
CHSV_MBADTR
CHSV_MBADTC
CHSV_MBAEXC
CHSV_MBANED
CHSV_PGMERR
CHSV_PGMHDE
CHSV_SYSERR
CHSV_SYSMEM
CHSV_SYSSBI
CLISK_ALNUM
CLISK_ALPHA
CLISK_BIF
CLISK_BR
CLISK_DEC
CLISK_ERROR
CLISK_EXIT
CLISK_HEX
CLISK_KEYWORD
CLISK_NUM
CLISK_OCT
CLISK_SPACE
CLISK_STRING
DATA_001
DSSAHORT
DSSASKADR
DSSASKDATA
DSSASKLGCL
DSSASKSTR
DSSA6KVLD
DSSHGNSUB
DSSBREAK
DSSCANWAIT

Symbol names (right column):

DSSCHANNEL
DSSCKLOOP
DSSCLQVEC
DSSCNTRLC
DSSCVTREG
DSSENDPASS
DSSENDSUB
DSSERRDEV
DSSERRHARD
DSSERRSOFT
DSSERRSYS
DSSESCAPE
DSSGETBUF
DSSGETMEM
DSSGPHARD
DSSINITSCB
DSSINLOOP
DSSLOAD
DSSMMOFF
DSSMMON
DSSMOVPHY
DSSMOVVRT
DSSPARSE
DSSPRINTB
DSSPRINTF
DSSPRINTS
DSSPRINTX
DSSRELBUF
DSSRELMEM
DSSSETIPL
DSSSFTMAP
DSSSETVEC
DSSSHOCHAN
DSSSUMMARY
DSSWAITMS
DSSWAITUS
ENVSM_DOMAIN
ENVSM_LEVEL
ENVSM_SUPER
ENVSS_DOMAIN
ENVSS_LEVEL
ENVSS_SUPER
ENVSV_DOMAIN
ENVSV_LEVEL
ENVSV_SUPER
ENVS_CPU
ENVS_FUNCTIONAL
ENVS_REPAIR
ENVS_SUPER
ENVS_SYSTEM
GBL...
GD_L_LUN
GD_L_TOKEN
GD_T_CMDBUF
GD_T_CMDBUF
GD_T_COMMAND
GD_T_STRBUF
GD_W_CSR

```
EVPRG1          Diagnostic Program Example      8-AUG-1979 16:12:06   VAX-11 Macro V02.30
Symbol table                                    8-AUG-1979 13:22:58   DB0:[STAPLES]EVPRG1.MAR;1

GD_W_XOR            ********  X   03
GT_T_AMBCMD        ********  X   03
GT_T_CSRERR        ********  X   03
GT_T_GETLINE       ********  X   03
GT_T_INVCMD        ********  X   03
GT_T_NOERR         ********  X   03
GT_T_REGERR        ********  X   03
GT_T_T1S2          ********  X   03
G_K_STOP           ********  X   03
G_K_SUB1           ********  X   03
G_K_SUB2           ********  X   03
MSG_001            00000002  R
MSG_REGERR         ********      R
SEP_FUNCTIONAL  =  00000002     X
SEP_REPAIR      =  00000003
SYSSALLOC          00010238
SYSSASCTIM         00010248
SYSSASSIGN         00010250
SYSSBINTIM         00010258
SYSSCANCEL         00010260
SYSSCANTIM         00010268
SYSSCLREF          00010290
SYSSDALLOC         000102D8
SYSSDASSGN         000102E0
SYSSFAO            00010357
SYSSFAOL           00010358
SYSSGETCHN         000104C8
SYSSGETTIM         00010378
SYSSNUMTIM         00010388
SYSSQIO            000103C8
SYSSQIOW           00010200
SYSSREADEF         000103D0
SYSSSETEF          00010400
SYSSSETIMR         00010420
SYSSSETPRT         00010430
SYSSUNWIND         00010470
SYSSWAITFR         00010478
SYSSWFLAND         00010488
SYSSWFLOR          00010490
T1_GETCMD          0000001A  R    03
T1_S1              00000005  RG   03
T1_S1_X            0000011B  R    03
T1_S2              00000126  RG   03
T1_S2_X            00000155  R    03
T1_S3              0000016A  RG   03
T1_S3_X            0000017A  R    03
TEST_001           00000018  RG   03
TEST_001_X         0000018A  RG   03
```

```
EVPRG1      Diagnostic Program Example                    8-AUG-1979 16:12:06    VAX-11 Macro V02.30
Psect synopsis                                            8-AUG-1979 13:22:58    DB01:[STAPLES]EVPRG1.MAR;1

                                           +-----------------+
                                           | Psect synopsis  |
                                           +-----------------+

PSECT name   Allocation         PSECT No.  Attributes
----------   ----------         ---------  ----------
. ABS .      00000000 (    0.)   00 ( 0.)  NOPIC USR CON ABS LCL NOSHR NOEXE NORD NOWRT BYTE
. BLANK .    00000000 (    0.)   01 ( 1.)  NOPIC USR CON REL LCL NOSHR EXE   RD   WRT   BYTE
$ABS$        000104D0 (66768.)   02 ( 2.)  NOPIC USR CON ABS LCL NOSHR EXE   RD   WRT   BYTE
TEST_001     00000193 (  403.)   03 ( 3.)  NOPIC USR CON REL LCL NOSHR EXE   RD   NOWRT PAGE
DISPATCH     00000018 (   24.)   04 ( 4.)  NOPIC USR CON REL LCL NOSHR NOEXE RD   NOWRT LONG
ARGLIST      00000024 (   36.)   05 ( 5.)  NOPIC USR CON REL LCL NOSHR EXE   RD   NOWRT LONG
STSTCNT      00000004 (    4.)   06 ( 6.)  NOPIC USR OVR REL LCL NOSHR NOEXE RD   NOWRT LONG
```

EVPRG1   Diagnostic Program Example   8-AUG-1979 16:12:06   VAX-11 Macro V02.30
Cross reference   8-AUG-1979 13:22:58   DB0:[STAPLES]EVPRG1.MAR;1

```
                                    +-------------------------+
                                    ! Symbol Cross Reference  !
                                    +-------------------------+
```

SYMBOL	VALUE	DEFINITION	REFERENCES...					
SSS	=000000018-R	234 (9)	159 (6)	205 (8)	234 (9)	212 (8)		
SSE	=00000001	234 (9)	185 (7)	214 (8)	241 (9)	183 (8)	212 (8)	
SSM	=00000004	212 (8)	110 (4)	178 (6)	183 (7)	234 (9)	71 (3)	
SSN	=00000000	212 (8)	#=130 (4)	#=137 (5)	178 (6)	205 (8)		
SSS	=FFFFFFFF	245 (12)	110 (4)	159 (6)	205 (8)	234 (9)		
SENV	=00000003	58 (2)						
SER	=FFFFFFFF	245 (14)	159 (6)	#=178 (6)	205 (7)	#=212 (8)		
SMO	=00000001	247 (14)	234 (9)	247 (10)	185 (7)	205 (8)	214 (8)	234 (9)
SSS	=000000018-R	234 (9)	159 (6)	185 (7)	205 (8)	214 (8)	234 (9)	
SST	=00000000	245 (12)	110 (4)	159 (6)	178 (6)	183 (7)	185 (7)	212 (8)
			205 (8)	212 (10)	214 (8)	241 (9)		
STN	=00000001	247 (10)	110 (4)	178 (6)	183 (7)	212 (8)	245 (10)	
			247 (10)	71 (2)	61 (2)			
ACT_ENTRY	00000000-XR		126 (4)					
BASE_001	00000000-R	71 (3)	110 (4)					
BIT...	=0000000D	61 (2)	58 (2)	60 (2)				
CEP_FUNCTIONAL	=00000000	58 (2)						
CEP_REPAIR	=00000001	58 (2)						
CHCS_ABORT	=00000002	60 (2)						
CHCS_CLEAR	=00000006	60 (2)						
CHCS_CLRDFT	=00000009	60 (2)						
CHCS_DSINT	=00000005	60 (2)						
CHCS_ENINT	=00000004	60 (2)						
CHCS_INITA	=00000000	60 (2)						
CHCS_INITB	=00000001	60 (2)						
CHCS_PURGE	=00000003	60 (2)						
CHCS_SETDFT	=00000008	60 (2)						
CHCS_STATUS	=00000007	60 (2)						
CHISM_CHNINT	=00000001	60 (2)						
CHISM_DEVINT	=00000000	60 (2)						
CHISM_IPL	=0000007C	60 (2)						
CHISS_IPL	=00000005	60 (2)						
CHISV_CHNINT	=00000000	60 (2)						
CHISV_DEVINT	=00000001	60 (2)						
CHISV_IPL	=00000002	60 (2)						
CHMS_FORWARD	=00000000	60 (2)						
CHMS_INVALIDATE	=00000001	60 (2)						
CHMS_MAP	=00000002	60 (2)						
CHMS_MFWDN	=0000000A	60 (2)						
CHMS_MFWDNO	=00000003	60 (2)						
CHMS_MFWDV	=0000000B	60 (2)						
CHMS_MFWDVO	=00000006	60 (2)						
CHMS_MREVN	=0000000E	60 (2)						
CHMS_MREVNO	=00000007	60 (2)						
CHMS_MREVV	=0000000F	60 (2)						
CHMS_MREVVO	=00000000	60 (2)						
CHMS_NFWDN	=00000000	60 (2)						

8-AUG-1979 16:12:06   VAX-11 Macro V02.30
8-AUG-1979 13:22:58   DB0:[STAPLES]EVPRG1.MAR;1

EVPRG1   Diagnostic Program Example
Cross reference

Symbol	Value	Refs		
CHMS_NREVN	=00000004	60	(2)	
CHMS_OFFSET	=00000008	60	(2)	60 (2)
CHMS_REVERSE	=00000004	60	(2)	60 (2)
CHMSM_BUSIC	=00800000	60	(2)	
CHMSM_BUSINIT	=01000000	60	(2)	
CHMSM_BUSPDN	=00000040	60	(2)	60 (2)
CHMSM_CHNDPE	=00000002	60	(2)	
CHMSM_CHNERR	=00000080	60	(2)	
CHMSM_CHNMPE	=00000100	60	(2)	
CHMSM_CHPFOT	=0000001A	60	(2)	60 (2)
CHMSM_DEVBUS	=00000010	60	(2)	
CHMSM_DEVERR	=00000004	60	(2)	
CHMSM_DEVTO	=00000020	60	(2)	
CHMSM_ERRANY	=0000000F	60	(2)	
CHMSM_MBAATN	=00100000	60	(2)	
CHMSM_MBACPE	=00200000	60	(2)	
CHMSM_MBADTB	=00400000	60	(2)	
CHMSM_MBADTC	=00800000	60	(2)	
CHMSM_MBAEXC	=00010000	60	(2)	
CHMSM_MBANED	=00020000	60	(2)	
CHMSM_PGMERR	=00080000	60	(2)	60 (2)
CHMSM_SYSERR	=00000001	60	(2)	
CHMSM_SYSMEM	=00000200	60	(2)	
CHMSM_SYSSBI	=00000400	60	(2)	60 (2)
CHMSV_BUSIC	=00000017	60	(2)	
CHMSV_BUSINIT	=00000016	60	(2)	
CHMSV_BUSPDN	=00000018	60	(2)	
CHMSV_CHNDPE	=00000006	60	(2)	
CHMSV_CHNERR	=00000001	60	(2)	
CHMSV_CHNMPE	=00000007	60	(2)	
CHMSV_CHPFOT	=00000008	60	(2)	
CHMSV_DEVBUS	=00000004	60	(2)	
CHMSV_DEVERR	=00000005	60	(2)	
CHMSV_DEVTO	=00000002	60	(2)	
CHMSV_MBAATN	=00000014	60	(2)	
CHMSV_MBACPE	=00000015	60	(2)	
CHMSV_MBADTB	=00000012	60	(2)	
CHMSV_MBADTC	=00000013	60	(2)	
CHMSV_MBAEXC	=00000010	60	(2)	
CHMSV_MBANED	=00000011	60	(2)	
CHMSV_PGMERR	=00000003	60	(2)	
CHMSV_PGMHDE	=0000000A	60	(2)	
CHMSV_SYSERR	=00000000	60	(2)	
CHMSV_SYSMEM	=00000009	60	(2)	
CHMSV_SYSSBI	=0000000A	61	(2)	
CLISK_ALNUM	=00000087	61	(2)	
CLISK_ALPHA	=00000086	61	(2)	
CLISK_BIF	=00000083	61	(2)	
CLISK_BR	=0000008A	61	(2)	
CLISK_DEC	=0000008C	61	(2)	
CLISK_ERROR	=00000080	61	(2)	
CLISK_EXIT	=00000081	61	(2)	
CLISK_HEX	=00000089	61	(2)	
CLISK_KEYWORD	=0000008C	61	(2)	
CLISK_NUM	=00000085	61	(2)	
CLISK_OCT	=00000088	61	(2)	

VAX-11 Macro V02.30
DB0:[STAPLES]EVPRG1.MAR;1

8-AUG-1979 16:12:06
8-AUG-1979 13:22:58

EVPRG1   Diagnostic Program Example
Cross Reference

Symbol	Value				
CLISK_SPACE	=02000084	61 (2)			
CLISK_STRING	=02000088	61 (2)			
DATA_001	00000014-R	11 (4)	113 (4)		
DSSABORT	00010020	62 (2)			
DSSASKADR	00010090	62 (2)			
DSSASKDATA	00010080	62 (2)			
DSSASKLGCL	00010098	62 (2)			
DSSASKSTR	00010040	62 (2)	116 (4)		
DSSASKVLD	00010088	62 (2)			
DSSBGNSUB	00010030	62 (2)	159 (6)	205 (8)	234 (9)
DSSBREAK	00010058	62 (2)	245 (10)		
DSSCANWAIT	00010070	62 (2)			
DSSCHANNEL	00010180	62 (2)			
DSSCKLOOP	00010040	62 (2)			
DSSCLRVEC	00010168	62 (2)			
DSSCNTRLC	00010078	62 (2)			
DSSCVTREG	000100B0	62 (2)			
DSSENDPASS	00010010	62 (2)			
DSSENDSUB	00010038	62 (2)	185 (7)	214 (8)	241 (9)
DSSERRDEV	000100C0	62 (2)	178 (6)	193 (7)	212 (8)
DSSERRHARD	000100D0	62 (2)			
DSSERRSOFT	000100D8	62 (2)			
DSSERRSYS	000100C8	62 (2)			
DSSESCAPE	00010050	62 (2)			
DSSGETBUF	00010120	62 (2)			
DSSGETMEM	00010130	62 (2)			
DSSGPHARD	00010018	62 (2)			
DSSINITSCB	00010170	62 (2)			
DSSINLOOP	00010048	62 (2)			
DSSLOAD	00010198	62 (2)			
DSSMMOFF	00010158	62 (2)			
DSSMMON	00010150	62 (2)			
DSSMOVPHY	00010148	62 (2)			
DSSMOVVRT	00010140	62 (2)			
DSSPARSE	00010088	62 (2)	126 (4)		
DSSPRINTB	000100E0	62 (2)	130 (4)	137 (5)	
DSSPRINTF	000100F0	62 (2)			
DSSPRINTS	000100F8	62 (2)			
DSSPRINTX	000100E8	62 (2)			
DSSRELBUF	00010128	62 (2)			
DSSRELMEM	00010138	62 (2)			
DSSSETIPL	00010178	62 (2)			
DSSSETMAP	00010188	62 (2)			
DSSSETVEC	00010160	62 (2)			
DSSSHOCHAN	00010190	62 (2)			
DSSSUMMARY	00010028	62 (2)			
DSSWAITMS	00010060	62 (2)			
DSSWAITUS	00010068	62 (2)			
ENVSM_DOMAIN	=00000002	58 (2)			
ENVSM_LEVEL	=00000001	58 (2)			
ENVSM_SUPER	=000003FC	58 (2)			
ENVSS_DOMAIN	=00000001	58 (2)			
ENVSS_LEVEL	=00000001	58 (2)			
ENVSS_SUPER	=00000008	58 (2)			
ENVSV_DOMAIN	=00000001	58 (2)	58 (2)		
ENVSV_LEVEL	=00000001	58 (2)	58 (2)		
ENVSV_SUPER	=00000002	58 (2)			

Page 17 (10)

VAX-11 Macro V02.30
DB0:[STAPLE8]EVPRG1.MAR;1

8-AUG-1979 16:12:06
8-AUG-1979 13:22:58

EVPRG1   Diagnostic Program Example
Cross reference

Symbol	Address				
ENVS_CPU	#00000400P	58 (2)			
ENVS_FUNCTIONAL	#00000A00	58 (2)			
ENVS_REPAIR	#00000001	58 (2)			
ENVS_SUPER	#00000001	58 (2)			
ENVS_SYSTEM	#00000001	62 (2)			
GBL...	#00000000	58 (2)	60	61 (2)	62 (2)
GD_L_LUN	00000000-XR	58 (2)	#-183 (2)	#-212 (8)	
GD_L_TOKEN	00000000-XR	#-178 (6)	#-161	#-207 (8)	#-236 (9)
GD_L_CMDBUF	00000000-XR	#-133 (5)	#-122 (6)	126 (4)	
GD_I_COMMAND	00000000-XR	#-120 (4)			
GD_I_STRBUF	00000000-XR	130 (4)			
GD_W_CSR	00000000-XR	116 (6)	#-119 (4)	121 (4)	62 (2)
GD_W_XOR	00000000-XR	#-165 (6)	#-166 (6)	#-167 (6)	#-178 (6)
GT_T_AMBCMD	00000000-XR	#-168 (5)			
GT_I_CSRERR	00000000-XR	137 (6)			
GT_I_GETLINE	00000000-XR	#-178 (6)			
GT_I_INVCMD	00000000-XR	116 (4)			
GT_I_NOERR	00000000-XR	183 (7)			
GT_T_REGERR	00000000-XR	178 (6)			
GT_T_TIS2	00000000-XR	212 (8)			
G_K_STOP	00000000-XR	#-236 (9)			
G_K_SUB1	00000000-XR	#-161 (6)			
G_K_SUB2	00000000-XR	#-207 (8)			
MSG_001	00000002-R	71 (3)	110 (4)		
MSG_REGERR	00000000-XR		178 (6)		
SEP_FUNCTIONAL	#00000002	58 (2)			
SEP_REPAIR	#00000003	58 (2)			
SYSSALLOC	00010238	62 (2)			
SYSSASCTIM	00010248	62 (2)			
SYSSASSIGN	00010250	62 (2)			
SYSSBINTIM	00010258	62 (2)			
SYSSCANCEL	00010260	62 (2)			
SYSSCANTIM	00010268	62 (2)			
SYSSCLREF	00010290	62 (2)			
SYSSDALLOC	000102D0	62 (2)			
SYSSDASSGN	000102E0	62 (2)			
SYSSFAO	00010350	62 (2)			
SYSSFAOL	000104C8	62 (2)			
SYSSGETCHN	00010378	62 (2)			
SYSSGETTIM	00010388	62 (2)			
SYSSNUMTIM	000103C8	62 (2)			
SYSSQIO	00010200	62 (2)			
SYSSQIOW	000103D0	62 (2)			
SYSSREADEF	00010400	62 (2)			
SYSSSETEF	00010420	62 (2)			
SYSSSETIMR	00010430	62 (2)			
SYSSSETPRT	00010470	62 (2)			
SYSSUNWIND	00010478	62 (2)			
SYSSWAITFR	00010488	62 (2)			
SYSSWFLAND	00010490	62 (2)			
SYSSWFLOR					
TI_GETCMD	00000001A-R	112 (4)	*-243 (14)		
TI_S1	00000095-R	159 (0)			
TI_S1_X	0000011B-R	185 (7)	+-163 (6)	#-179 (6)	
TI_S2	00000126-R	205 (8)			
TI_S2_X	00000155-R	214 (8)	#-228		
TI_S3	00000160-R	234 (9)			

EVPRG1   Diagnostic Program Example   8-AUG-1979 16:12:06   Page 18
Cross reference   8-AUG-1979 13:22:58   (18)

                                              VAX-11 Macro V02.30
                                              DB0:[STAPLES]EVPRG1.MAR;1

```
T1.S3.X       0000017A-R    241   (9)    u-237   (9)
TEST.001      000018-R      110   (4)     11?    (4)
TEST.001.X    0000188-R     245   (13)   u-239   (9)
```

EVPRG1   Diagnostic Program Example

Cross reference

8-AUG-1979 16:12:06   VAX-11 Macro V02.30
8-AUG-1979 13:22:58   DB0:[STAPLES]EVPRG1.MAR;1

```
+=========================+
| Macro Cross Reference   |
+=========================+
```

MACRO	SIZE	DEFINITION	REFERENCES...
SD1_BBUBT	1	159 (6)	159 (6) 205 (8) 234 (9)
SD1_BTEST	1	110 (4)	110 (4)
SD1_ERR	1	178 (6)	178 (6) 183 (7) 212 (8)
SD1_ERR_S	2	178 (6)	178 (6) 183 (7) 212 (8)
SD1_ESUBT	1	185 (7)	185 (7) 214 (8) 241 (9)
SD1_ETEST	2	245 (10)	245 (10)
SD1_PRINT_B	2	130 (4)	130 (4) 137 (5)
SD1_SBTTL	1	71 (3)	71 (3)
SD2_BDATA	1	110 (4)	110 (4)
SD2_BTEST	1	110 (4)	110 (4)
SD2_SBTTL	1	71 (3)	71 (3)
SDEF	1	62 (2)	62 (2)
SDEFEND	1	62 (2)	62 (2)
SDEFINI	1	62 (2)	62 (2)
SDS_ASKSTR_S	1	114 (4)	114 (4)
SDS_BCOMPLETE	1	127 (4)	127 (4)
SDS_BGNMOD	1	58 (2)	58 (2) 205 (8) 234 (9)
SDS_BGNSUB	1	159 (6)	159 (6)
SDS_BGNTEST	1	110 (4)	110 (4)
SDS_BNCOMPLETE	1	117 (4)	117 (4)
SDS_BREAK	1	245 (10)	245 (10)
SDS_CHCDEF	1	60 (2)	60 (2)
SDS_CHDEF	1	60 (2)	60 (2)
SDS_CHIDEF	1	60 (2)	60 (2)
SDS_CHMDEF	2	60 (2)	60 (2)
SDS_CHSDEF	3	60 (2)	60 (2)
SDS_CLIDEF	1	61 (2)	61 (2)
SDS_DSSDEF	1	62 (2)	62 (2)
SDS_ENDDATA	1	114 (4)	114 (4)
SDS_ENDMOD	1	247 (10)	247 (10) 241 (9)
SDS_ENDSUB	1	185 (7)	185 (7) 211 (8)
SDS_ENDTEST	1	245 (10)	245 (10)
SDS_ENVDEF	2	58 (2)	58 (2)
SDS_ERRHARD_S	2	172 (6)	172 (6) 182 (7)
SDS_PAGE	1	69 (2)	69 (2)
SDS_PARSE_S	2	124 (4)	124 (4) 137 (5)
SDS_PRINTF_S	1	130 (4)	130 (4)
SDS_SBTTL	1	71 (3)	71 (3)
SEQU	1	58 (2)	58 (2) 61 (2)
SEQULS1	1	58 (2)	58 (2) 61 (2)
SEQULST	1	58 (2)	58 (2) 61 (2)
SGBLINI	1	116 (4)	116 (4) 183 (7) 62 (2) 212 (8)
SPUSHADR	1	58 (2)	58 (2)
SVIELD	1	58 (2)	58 (2)
SVIELD1	2	58 (2)	58 (2)

```
EVPRG1      Diagnostic Program Example        8-AUG-1979 16:12:06   VAX-11 Macro V02.30
VAX-11 Macro Run Statistics                   8-AUG-1979 13:22:58   DB0:[STAPLES]EVPRG1.MAR;1

                                              +-----------------------+
                                              | Performance Indicators |
                                              +-----------------------+

Phase                       Page faults   CPU Time        Elapsed Time
-----                       -----------   --------        ------------
Initialization                    13      00:00:00.02     00:00:00.33
Command processing                20      00:00:00.18     00:00:01.05
Pass 1                          4141      00:00:12.72     00:01:47.29
Symbol table sort                 12      00:00:00.23     00:00:01.41
Pass 2                           455      00:00:01.94     00:00:08.05
Symbol table output               29      00:00:00.14     00:00:00.47
Pass synopsis output               6      00:00:00.03     00:00:00.30
Cross-reference output           205      00:00:00.78     00:00:02.53
Assembler run totals            4894      00:00:16.05     00:01:22.54

The working set limit was 150 pages.
107018 bytes (210 pages) of virtual memory were used to buffer the intermediate code.
There were 20 pages of symbol table space allocated to hold 219 non-local and 5 local symbols.
248 source lines were read in Pass 1, producing 21 object records in Pass 2.
52 pages of virtual memory were used to define 36 macros.

                                              +------------------------+
                                              | Macro library statistics |
                                              +------------------------+

Macro library name              Index size (pages)    Macros defined
------------------              ------------------    --------------
DRA2:[SYSLIB]DIAG.MLB;574              15                   35
DRA2:[SYSLIB]STARLET.MLB;10            22                   10
TOTALS (all libraries)                37                   45

500 GETS were required to define 45 macros.

There were no errors or warnings.

/LIS/CROSS EVPRG1
```

absolute (ABS) -- A program section (psect) attribute. An absolute psect contains only symbol definitions. It does not contribute binary code to the image. Therefore, it must have a zero-length memory allocation. The converse is relocatable (REL).

access mode -- Any of the four processor access modes in which software executes. Processor access modes are, in order, from most to least privileged and protected: kernel (mode Ø), executive (mode 1), supervisor (mode 2), and user (mode 3).

When the processor is in kernel mode, the executing software has complete control of, and responsibility for, the system. When the processor is in any other mode, the processor is inhibited from executing privileged instructions. The processor status longword contains the current access mode field. The operating system uses access modes to define protection levels for software executing in the context of a process. For example, the executive runs in kernel and executive modes and is most protected. The command interpreter is less protected and runs in supervisor mode. The debugger runs in user mode and is no more protected than normal user programs.

access type -- The way in which the processor accesses instruction operands. Access types are: read, write, modify, address, and branch.

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 (Ø--9).

ancillary control process (ACP) -- A process that acts as an interface between user software and an I/O driver. An ACP provides functions supplemental to those performed in the driver, such as file and directory management.

APT -- An automated product test application used throughout DIGITAL manufacturing. APT employs remote diagnostic scripting with down-line diagnostic load.

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).

argument pointer -- General register 12 (R12). By convention, AP contains the address of the base of the argument list for procedures initiated using the CALL instructions.

assign a channel -- To establish the necessary software linkage between a user process and a device unit before a user process can transfer any data to or from that device. A user process requests the system to assign a channel and the system returns a channel number.

assembler -- A program that translates source language code, whose operations correspond directly to machine op codes, into object language code.

asynchronous system trap (AST) -- A software-simulated interrupt to a user-defined service routine. ASTs enable a user process to be notified asynchronously, with respect to its execution, of the occurrence of a specific event. If a user process has defined an AST routine for an event, the system interrupts the process and executes the AST routine when that event occurs. When the AST routine exits, the system resumes the process at the point where it was interrupted.

attributes -- Various characteristics that can be assigned by the programmer to each psect in a module (e.g., ABS).

base register -- A general register used to contain the address of the entry in a list, table, array, or other data structure.

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).

breakpoint -- In diagnostics, an address assigned through the diagnostic supervisor. When the PC equals the value of the breakpoint, control returns to the diagnostic supervisor.

boot (bootstrap) -- A program that loads another (usually larger) program into memory from a peripheral device.

buffer -- A temporary data storage area.

call frame -- A standard data structure built on the stack during a procedure call, starting from the location addressed by the FP to lower addresses, and popped off during a return from procedure (also called stack frame).

channel -- A logical path connecting a user process to a physical device unit. A user process requests the operating system to assign a channel to a device so that the process can transfer data to or from that device.

command file -- A file containing command strings.

command interpreter -- Procedure-based code to receive, syntax check, and parse commands typed by the user at a terminal or submitted in a command file.

command line interpreter (CLI) -- The portion of the diagnostic supervisor that handles communication between the operator's terminal, the diagnostic supervisor, and the program to be run. The CLI interprets commands typed on the operator's terminal.

command parameter -- The positional operand of a command delimited by spaces, such as a file specification, option, or constant.

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.

concatenate (CON) -- A program section attribute. If a psect is concatenated, all psects of the same name yet from different modules are to be assigned contiguous addresses in the virtual address space. Each module can specify an independent alignment. The linker performs the necessary padding of zero bytes between contributions. The base alignment of the resulting concatenated psects is according to the greatest alignment granularity of all the contributions to the psect. For example, if the greatest alignment granularity of all contributors is a page, the psect is page-aligned; although, some contributors may be byte-aligned, others word-aligned, etc.

condition -- An exception condition detected and declared by software.

condition codes -- Four bits in the processor status word that indicate the results of the previously executed instruction.

condition handler -- A procedure that a process wants the system to execute when an exception condition occurs. When an exception condition does occur, the operating system searches for a condition handler. When it finds the condition handler, the operating system initiates the handler immediately. The condition handler may perform some act to change the situation that caused the exception condition and then continue execution of the process that incurred the exception condition. Condition handlers execute in the context of the process at the access mode of the code that incurred the exception condition.

console level -- console-based diagnostic program that can be run in the standalone mode only.

context switching -- Interrupting the activity in progress and switching to another activity. Context switching occurs as one process after another is scheduled for execution. The operating system saves the interrupted process's hardware context in its hardware PCB using the save process context instruction, loads another process's hardware PCB into the hardware context using the load process context instruction, and schedules that process for execution.

cylinder -- The tracks at the same radius on all recording surfaces of a disk pack.

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.

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 interrupt -- An interrupt received on interrupt priority levels 16 through 23. Device interrupts can be requested only by devices, controllers, and memories.

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). A colon (:) separates it from following fields.

diagnostic supervisor -- 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.

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 set of system code that handles physical I/O to a device.

entry mask -- A word (1) whose bits represent the registers to be saved or restored on a subroutine or procedure call using the call and return instructions, and (2) which includes trap enable bits.

entry point -- A location that can be specified as the object of a call. It contains an entry mask and exception enables known as the entry point mask.

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's ability to execute. Events can be synchronous with the process's execution (a wait request, or they can be asynchronous (I/O completion). Some examples of events: swapping, wake request, page fault.

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.

exception -- An event detected by the hardware (other than an interrupt or jump, branch, case, or call instruction) that changes the normal flow of instruction execution. An exception is always caused by the execution of an instruction or set of instructions, while an interrupt is caused by an activity in the system independent of the current instruction. There are three types of hardware exceptions: traps, faults, and aborts. Examples are: attempts to execute a privileged or reserved instruction; trace traps; compatibility mode faults; breakpoint instruction execution; and arithmetic traps such as overflow, underflow, and divide-by-zero.

exception condition -- A hardware- or software-detected event (other than an interrupt or jump, branch, case, or call instruction) that changes the normal flow of instruction execution.

exception dispatcher -- An operating system procedure that searches for an exception handler when an exception condition occurs. If no exception handler is found for an exception or condition, the image that incurred the exception is terminated.

executable (EXE) -- A program section attribute. The psect contains only instructions. This attribute provides the capability to separate instructions from read-only and read/write data. The linker uses this attribute in gathering psects and in the verification of the transfer address that must be present in an executable psect.

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.

field replaceable unit (FRU) -- A subassembly, one or a few modules, or one or a few integrated circuits 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.

frame pointer -- General register 13 (R13). By convention, FP contains the base address of the most recent call frame on the stack.

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.

granularity -- The alignment of a contribution to a psect on a boundary. The alignment granularity may be byte, word, longword, quadword, or page.

home block -- A block in the index file that contains the volume identification, such as volume label and protection.

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.

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 priority level (IPL) -- The interrupt level at which the processor executes when an interrupt is generated. There are 31 possible interrupt priority levels. IPL 1 is lowest, 31 highest. The levels arbitrate contention for processor service. 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 PSL. The interrupt stack is not context-switched.

I/O function code -- A 6-bit value specified in a Queue I/O request system service that describes the particular I/O operation to be performed (e.g., read, write, rewind).

level 1 -- Operating system (VMS) based diagnostic programs using logical or virtual QIO.

level 2 -- Diagnostic supervisor-based diagnostic programs that can be run either under VMS (on-line) or in the standalone mode, using physical QIO.

level 2R -- Diagnostic supervisor-based diagnostic programs that can be run only under VMS, using physical QIO.

level 3 -- Diagnostic supervisor-based diagnostic programs that can be run in standalone mode only, using direct I/O.

level 4 -- Standalone macrodiagnostic programs that run without the supervisor.

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.

link map -- A link map shows the virtual memory allocation of the total program image. The link map is found in a program listing in the program section allocation synopsis.

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.

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. They can, however, be passed through the linker to the symbolic debugger. Contrast with global symbol.

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 (normally 0-7) of a device under test.

macro -- A statement that requests a language processor to generate a predefined set of instructions.

memory management -- The system functions that include the hardware's page mapping and protection and the operating system's image activator and pager.

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.

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).

network service protocol (NSP) -- The logical link control layer of DECNET architecture.

object module -- The binary output of a language processor such as the assembler or a compiler, which is used as input to the linker.

on disk structure (ODS1, ODS2) -- A files-11 disk format used by VMS.

operand -- a value (address or data) that is operated on, or with, by an instruction.

overlay (OVR) -- A program section attribute. If a psect is overlayed, all contributions to the psect have the same base address. The length of the psect is the size of the largest contribution. All contributions to an overlayed psect must have the same alignment.

page -- A set of 512 contiguous byte locations used as the unit of memory mapping and protection. Also, the data between the beginning of a file and a page marker, between two markers, or between a marker and the end of a file.

page frame number (PFN) -- The address of the first byte of a page in physical memory. The high-order 21 bits of the physical address of the base of a page make up the PFN.

page table entry (PTE) -- The data structure that identifies the location and status of a page of virtual address space. When a virtual page is in memory, the PTE contains the page frame number needed to map the virtual page to a physical page. When it is not in memory, the PTE contains the information needed to locate the page on secondary storage (disk).

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 the components of a command into structural forms.

physical address -- The address used by hardware to identify a location in physical memory or on directly addressable secondary storage devices such as disks. A physical memory address consists of a page frame number and the number of a byte within the page. A physical disk block address consists of a cylinder or track and sector number.

physical block -- A block on a mass storage device referred to by its physical (device-oriented) address rather than a logical (volume-relative) or virtual (file-relative) address.

position independent code (PIC) -- A program section attribute. The contents of the psect do not depend on a specific location in virtual memory. The converse is nonposition independent code (NOPIC).

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.

program interface (PGI) -- The portion of the diagnostic supervisor that handles communication between the diagnostic program and the supervisor. The PGI implements services requested by diagnostic programs.

program section -- A portion of a module. The assembler creates a number of program sections (psect) within a module, according to directives by the program developer. In addition, any code that precedes the first defined program section is placed in the BLANK program section by the assembler.

Through program sectioning the program developer controls the virtual memory allocation of a program. Any program attributes established by the program section directive are passed on to the linker. Thus, program sections can be declared as read only, nonexecutable, etc. See the VAX-11 MACRO Language Reference Manual for an explanation of the various program section attribute functions.

In diagnostic programs, each test is given a separate program section.

prompt -- A program's typed out response to and/or request for operator action.

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.

queue I/O -- A mode of access to peripheral devices in which a program calls on driver routines provided by the VMS operating system or the diagnostic supervisor to transfer data.

radix -- The base of the number system currently in use.

readable (RD) -- A program section attribute. The contents of the psect can be read at the execute time. The converse is nonreadable (NORD).

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.

relocatable (REL) -- A program section attribute. The psect must be assigned a base address by the linker. This psect can contain code and/or data.

script file -- A line-oriented ASCII file that contains a list of commands.

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. On a VAX-11 system, each track on a disk is normally divided into 22 sectors.

semantics -- The interpretation of and relation between commands or command symbols.

sharable image -- An image that has all of its internal references resolved, but which must be linked with an object module(s) to produce an executable image. A sharable image cannot be executed. A sharable image file can be used to contain a library of routines. A sharable image can be installed as a global section by the system manager.

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 frame -- A standard data structure built on the stack during a procedure call, starting from the location addressed by the FP to lower addresses, and popped off during a return from procedure. Also called call frame.

stack pointer -- General register 14 (R14). SP contains the address of the top (lowest address) of the processor-defined stack. Reference to SP will access one of the five possible stack pointers: kernel, executive, supervisor, user, or interrupt, depending on the value in the current mode and interrupt stack bits in the Processor Status Longword (PSL).

standalone mode -- A diagnostic program environment in which the program and the diagnostic supervisor run without the VMS 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.

status code -- A longword value that indicates the success or failure of a specific function. For example, system services always return a status code in R0 upon completion.

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.

syntactic unit -- An item contained within a command statement (e.g., an argument, a qualifier).

system image -- The image that is read into memory from secondary storage when the system is started up.

switch -- A parameter that is passed from a command line to a program.

test -- A unit of a diagnostic program that checks a specific function or portion of the hardware.

time stamp -- A statement of the time of day at which a specific event occurred.

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.

unit record device -- A device such as a card reader or line printer.

unwind the call stack -- To remove call frames from the stack by tracing back through nested procedure calls using the current content of the FP register and FP register content stored on the stack for each call frame.

UUT (unit under test) -- The device or portion of the computer hardware being tested by a diagnostic program.

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.

writable (WRT) -- A program section attribute. The content of the psect can be modified at execute time. The converse is nonwritable (NOWRT).

INDEX

Your comments and suggestions will help us in our continuous effort to improve the quality and usefulness of our publications.

What is your general reaction to this manual? In your judgment is it complete, accurate, well organized, well written, etc.? Is it easy to use? _____

_____

_____

_____

What features are most useful? _____

_____

_____

_____

What faults or errors have you found in the manual? _____

_____

_____

_____

Does this manual satisfy the need you think it was intended to satisfy? _____

Does it satisfy *your* needs? _____ Why? _____

_____

_____

_____

☐ Please send me the current copy of the *Technical Documentation Catalog*, which contains information on the remainder of DIGITAL's technical documentation.

Name _____   Street _____
Title _____   City _____
Company _____   State/Country _____
Department _____ Zip _____

Additional copies of this document are available from:

Digital Equipment Corporation
444 Whitney Street
Northboro, Ma 01532
Attention: Communications Services (NR2/M15)
            Customer Services Section

Order No. _____ EK-1VAXD-TM-001 _____

---- Fold Here ----

---- Do Not Tear - Fold Here and Staple ----