EK-1VAXD-TM-003

VAX Diagnostic Design Guide

Prepared bv Educational Services
of
Digital Equipment Corporation

First Edition August 1979
Second Edition, March 1981
Third Edition, November 1983

The manuscript for this book was created using DIGITAL Standard Runoff. Book production was done by Educational Services Development and Publishing in Nashua, NH.

CONTENTS

FIGURES

TABLES

EXAMPLES

CHAPTER 1
WHAT IS A DIAGNOSTIC PROGRAM?

1.1 INTRODUCTION

This chapter presents an introduction to diagnostic program design. It discusses the uses and users of diagnostic programs, the testing goals any diagnostic program design should meet, and the various methods used to test hardware. This chapter discusses those characteristics that are common to all diagnostic programs, regardless of the hardware they are designed to execute in or test.

1.2 USES OF DIAGNOSTIC PROGRAMS

A diagnostic program is any program designed specifically to discover and identify hardware failures in a computer system. There are three main cases in which diagnostic programs are used.

1. During execution of applications or systems programs, when the system produces unexpected events or incorrect computation results.

   This indicates the possibility of malfunctioning hardware. A diagnostic program or set of programs will be executed to determine if there was a hardware malfunction and, if so, which part of the system failed.

2. During manufacturing.

   After a hardware device is built, it must be thoroughly tested before it is shipped to a customer. This testing generally is performed "bottom-up." First, logic modules making up the device are tested separately. Then, the modules are put together to create the whole device and the device itself is tested. Sometimes entire systems are put together in the manufacturing plant and tested before being shipped to a customer. Other times, systems are not put together until the individual parts (previously tested processors and peripherals) arrive at the customer site.

3. During the design of a new product.

   If the functionality of a product is accurately defined, and a diagnostic program for the product is correctly written, then when the diagnostic program is executed it should (if the product has no hardware malfunctions) indicate that the product is functioning correctly. If the diagnostic program indicates that errors have been detected, they could be the result of a faulty product design that needs to be corrected.

1.3 DIAGNOSTIC PROGRAM USERS

Because diagnostic programs are put to various uses, the users (operators) of these programs are also varied. When a diagnostic program is used to identify problems in a system at a customer site, the program may be run by a customer service representative or by the customer.

Diagnostic programs used to verify proper functioning of new devices or systems might be run by technicians at the manufacturing site. They might be loaded and run using an automated method requiring no operator. Also, customer service representatives must verify proper functioning of new systems when the systems arrive at a customer site.

A diagnostic program used for design verification would probably be run by a hardware design engineer.

Because of the variety of users for diagnostic programs, the program developer should be aware who the users of his or her program will be. Some programs may be intended for a specific audience, and the program can be tailored to its needs, background, and experience. Other programs will be intended for a wide range of users and must be written to be useful to all of them.

1.4 USER REQUIREMENTS

All diagnostic program users have specific requirements that the programs must meet. While some requirements are common to more than one user, some are not.

All users have in common the need for good fault detection, or "coverage" (the ability to find as many failures as possible). Every user expects that if an error exists on the device being tested, then some diagnostic program will detect that error.

Customers, or "end users," have three main requirements for diagnostic programs.

- Ease of use.

  The functions of diagnostic programs are technical and relate to internal system hardware. An end user may not have the training to understand what operations are taking place in the diagnostic program. Therefore, the human interface must be simple. For example, installing cables, setting switches on logic boards, requesting information such as CSR addresses or device priority levels are all inappropriate.

- Preservation of user data.

  Since device media may contain data needed by the user, diagnostic programs must provide safeguards against destruction of this data. This is generally accomplished by only writing on media designated for diagnostic use. Some disks provide specific sectors that are used only for diagnostic purposes.

- Nonexclusion of users.

  A large system at a customer site will usually be timeshared by many users. If the users cannot use the system while diagnostic programs are running, significant loss to the customer can occur. Therefore, diagnostic programs should operate under the user's operating system and not preempt other system users.

Customer service representatives have the following diagnostic program requirements.

- Quick fault detection.

  The faster a customer service representative arrives at a site, fixes the problem, and leaves, the happier the customer. Diagnostic programs should be able to find faults as quickly as possible.

- Identification of bad field-replacable units.

  The diagnostic program should be able to tell the customer service representative which FRU (see definition in Section 1.6) should be replaced.

- Good program documentation.

  To identify a failure, it is often necessary for the customer service representative to understand what functions a diagnostic program is performing. Therefore, the program should be well documented with detailed functional descriptions of each test.

Manufacturing requirements depend on which phase of the manufacturing process a diagnostic program is used in.

In the module test phase, quick error detection is valued, particularly in high volume settings. Good error identification is sometimes NOT necessary, because modules are sent to module repair stations that use their own special-purpose hardware and software to identify module failures. In other cases, module repair stations are not used and good error identification IS important.

During device testing, manufacturing technicians have the same requirements as customer service representatives. Quick detection is needed so the manufacturing process will not be slowed. Identification of an easily replaced constituent part of the hardware system is necessary so the part can be replaced and the device shipped while the bad part is repaired, instead of holding up shipment of the device. Good documentation is necessary because determining the bad part sometimes requires a thorough understanding of the diagnostic program's functionality.

The main requirement of design engineers is that the program give good fault detection. Since the engineer is using the diagnostic program to check out his or her design, any section of the logic that the program does not test could contain a design flaw that may not be caught until after the hardware is in production, necessitating an engineering change order (ECO).

It is important to note that user requirements often vary from product to product. A particular user's specific needs often depend on the type of product for which the diagnostic program is being designed, or the program's use. For example, program requirements specified by manufacturing personnel will depend on the manufacturing site's testing stategy for the product. This strategy is often not the same from one product to the next. The program developer must maintain close communication with the program's eventual users in order to tailor the program to the requirements of those users.

1.5 RUN-TIME ENVIRONMENTS

The variety of uses and users of diagnostic programs creates a variety of "run-time environments" in which diagnostic programs must be able to execute. A "run-time environment" is the control-level software, if any, under which the diagnostic program must run. Some diagnostic programs cannot function in all run-time environments. The environments a program is designed to run in are determined by the purpose the program is to serve.

In the "user mode" run-time environment, a timesharing operating system is executing on the system tested. There could be many users on the system at the time a diagnostic program is run, and the diagnostic program is just another user of the system. The diagnostic program should not affect any other user on the system. (The operating system will prohibit the diagnostic program from exceeding its bounds.) Often, the device tested is assigned exclusively to the diagnostic program, and the device's storage medium must be replaced with a "scratch" medium the diagnostic program can use to write test patterns. Some storage devices provide an area for the exclusive use of diagnostic programs, such as the "maintenance cylinders" on some disk media. In such cases, the diagostic program uses this reserved area and other users of the device are unaffected.

The opposite of the user mode run-time environment is the "standalone mode" environment. In standalone mode, the diagnostic program has exclusive use of the computer system. There is no high-level operating system to allow other users to run at the same time or to place execution boundaries on the diagnostic program. Thus the diagnostic program can run in privileged execution modes and use reserved registers and memory space. Sometimes in standalone mode a monitor or other type of control program provides services to and controls execution of the diagnostic program. However, this type of monitor will not place execution constraints on the diagnostic program.

The advantage of standalone mode over user mode is that the lack of execution boundaries sometimes offers a greater level of resolution in error identification. The disadvantage is that the computer's operating system must be brought down, costing the customer time and money. This disadvantage does not exist when these programs are used on new systems at the manufacturing site.

The description of user and standalone modes has implied that the computer system under test is not connected to another system via any type of network used for system diagnosis. There are, however, networks that are used to load and run diagnostic programs, increasing the number of run-time environments to be contended with.

Networks are commonly used at manufacturing sites, where it is necessary to test a large number of systems at once. Typically, a host processor will maintain up-to-date copies of all diagnostic programs. The system to be tested will be connected to the host, and the host will transmit the appropriate programs to the test system. The programs will be executed in the test system's processor, but the host will monitor the performance of the programs and note any errors that occur.

Networks can also be used to diagnose systems at customer sites. In this case, a centrally located host system can use phone lines to "call" a customer's system. The host can then monitor diagnostic programs executed on the system tested and provide customer service representatives with the results of the tests. This can greatly decrease the amount of time customer service personnel must spend at the customer's site. Since they will not go to the customer site until after the tests are executed, they will have a good idea of what the problem is before they arrive.

1.6 DEFINITIONS

The following are some commonly used terms.

- **System under test (SUT)** - The hardware system on which a diagnostic program is executed.

- **Unit under test (UUT)** - The device tested (part of the SUT). The UUT is defined by the diagnostic program, and can be one drive of a particular device type or an entire subsystem of the SUT, such as one of the remote nodes of a host system.

- **Hardcore** - The portion of the SUT's hardware that must operate properly for the diagnostic program to execute. Programs that test peripheral devices typically have a hardcore consisting of the processor, main memory, and a program load device. A program's hardcore should never include any portion of the UUT.

- **Field-replacable unit (FRU)** - Any portion of the UUT that can be easily and quickly replaced at a customer's site (for example, a logic board).

1.7 TESTING GOALS

All diagnostic programs have the same testing goals, regardless of what they test and what their execution environments or main users are. The first goal is to

- Clearly define the testing scope and required hardcore.

The "testing scope" is that portion of the hardware logic which the program tests. It should never extend beyond the boundaries of the unit under test. For example, consider a disk controller that can support several drives. A diagnostic program to test the controller should not detect faults on the drives, unless it cannot be avoided. Signals generated in the logic should be limited to those areas meant to be tested by the diagnostic program. (The fewer stray signals there are in the system, the easier it will be to identify the failure.)

The hardcore required by the diagnostic program should be as small as possible. Testing almost any peripheral device requires some correctly functioning logic that signals must pass through in order to get to and from the UUT. The smaller this hardcore, the more likely that a diagnosis of the UUT can be made without finding other errors within the the system but outside the scope of test, which could invalidate the diagnosis. For example, a program designer writing a diagnostic program for a disk might have the option of having memory management on or off while the program is running. Having memory management on will increase the hardcore for the diagnostic program, and the program will not be able to test the disk if there are errors in the memory management logic.

The next goal of a diagnostic program is to

- Detect any and all failures that could occur within the testing scope.

If any part of the unit under test could malfunction, the diagnostic program should be able to detect that malfunction. The diagnostic program does NOT need to be concerned with problems outside the scope of the unit it is intended to test. For example, a diagnostic to test a disk driver should not be expected to detect CPU problems (although it might detect them inadvertently).

This goal is clear-cut and simple -- if a malfunction occurs anywhere within the unit under test, the diagnostic program should detect and report it. Thus a diagnostic program designed to test a set of tape drive controllers and their attached drives should be able to detect any failure occurring in either the controllers or their associated drives. A system exerciser (designed to validate the overall functionality of a computer system, including the CPU, memory, and all peripheral devices) should be able to detect errors on any device attached to the system.

Once a failure has been detected, the diagnostic progam must

- Attempt to identify which part of the unit under test caused the malfunction.

It is not enough to recognize that an error has occurred. The diagnostic program should also be able to indicate which part (or parts) need(s) to be repaired or replaced.

This third goal is not as clear-cut as the last one, for it involves the concept of "degree of resolution." When attempting to identify a failing part, the diagnostic program designer must decide what the smallest part within the system is that should be considered. Each computer system is made up of hardware devices, which contain one or several logic boards, which in turn are made up of IC chips. A diagnostic program's degree of resolution is a relative measure of its ability to identify the smallest possible failing constituent part. For example, consider a tape subsystem consisting of several tape drives connected to one controller. A diagnostic program that could identify the failing logic board within the failing tape drive would have a higher degree of resolution than one that only identified the failing drive. ("Fault isolation" is another phrase often used to refer to the degree of error resolution.)

A particular program's proper degree of resolution depends on its intended function. For example, it would be impractical for a system exerciser (described in Section 1.8) to attempt to identify failures to the degree of the failing chip. More likely, it would determine which peripheral device was malfunctioning and, if the peripheral consisted of several drives attached to one controller, which drive was in error. On the other hand, a diagnostic program designed to test a specific peripheral device probably should attempt to identify the failing logic board within that device.

A diagnostic program's degree of resolution can also be affected by the program's user requirements. It is not always practical to achieve the highest possible degree of resolution, because increasing resolution can also cause increased program size and run-time, and may require a more highly skilled operator. In some cases it may be more important to keep these variables within bounds than to attain a high degree of resolution.

Unfortunately, achieving a high degree of error resolution is sometimes more an ideal than an attainable goal. Diagnostic programs used by customer service representatives should ideally be able to identify the smallest malfunctioning FRU. But for the program to identify an error as existing on one particular FRU, two requirements must be met. First, all the hardware logic used to execute the function that failed must reside on a single FRU. Second, the diagnostic program must be able to determine which FRU the logic resides on. Both these requirements can only be met through proper hardware design of the device. Close communication between the hardware designer and the diagnostic program designer are essential when a new product is in development, to guarantee proper logic partitioning along with visibility of all signals needed by the program to achieve high error resolution.

It is sometimes not possible for a diagnostic program to accurately identify a failure to the degree of resolution desired in a particular situation. In these cases a technician will have to determine the failing component by examining electrical signals on logic boards with an oscilloscope. The responsibility of the diagnostic program then is to provide the technician with aids to locate the failure quickly and accurately. These aids mainly consist of program loops that can be invoked if an error is detected, and whose purpose is to provide repetitive state transitions on small subsets of the hardware logic so that the techinician can easily observe these transitions and make sure they are taking place properly.

Thus we have one final design goal for diagnostic programs that cannot isolate all faults automatically (at the present time, this includes ALL diagnostic programs). The goal is

- To provide enough useful program loops that all possible errors can be quickly and easily detected by observing logic state transitions.

This goal is more relevant to logic tests than to function tests, both of which are discussed next.

1.8 LOGIC TESTS, FUNCTION TESTS, AND EXERCISERS

Not all diagnostic programs have the same functional goals. In general, diagnostic programs can be divided into three groups: "logic tests," "function tests," and "exercisers."

A logic test is usually used during the repair of a failing device. A logic test tests the device's combinational logic (verifies that a specific section of hardware logic within the device is functioning correctly). A logic test should provide the greatest degree of error resolution of the three types of tests. Logic tests are designed to run in a standalone environment.

A function test verifies the functionality of a device. For example, a function test for a disk drive would be used to verify that the "functions" provided by the disk, such as reading and writing blocks of data, are operating properly. Function tests may be used in the repair of failing devices or to detect the failure. These tests may not have as great a degree of error resolution as logic tests. Function tests can be designed to run in either a standalone or user mode environment.

For many products, both a logic test and a function test are developed. The function test is used to detect the hardware failure and the logic test to repair the failing part. For some products, the function test is used for repair. Some products have logic tests in microprograms (see Section 1.11). In short, the types of programs developed vary from product to product. Program users will specify the types of programs they desire for a particular product.

A third type of diagnostic program is an exerciser. Its purpose is to verify that a system's functionality can be sustained over a period of time. Exercisers are more likely to be designed for use on entire systems than on a single device. Typically, an exerciser will simultaneously perform repeated functional testing of every device composing the system, in an attempt to detect (1) failures that result from this simultaneous use of numerous devices, or (2) failures that only occur rarely.

1.9 SERIAL AND PARALLEL TESTING

Many diagnostic programs are designed to test all units of a specific type of device existing on a given system. There are two methods by which this testing of multiple units can be accomplished, "serial testing" and "parallel testing." Serial testing involves testing each unit of the device individually, one at a time. Parallel testing is the testing of all units at once. Serial testing is more likely to be found in a logic test, where it is desirable to keep the overall level of system activity to a minimum. Parallel testing, on the other hand, may be included in function tests to achieve higher levels of system activity.

1.1Ø BOTTOM-UP AND TOP-DOWN TESTING

Two testing techniques are used to test hardware systems. They are generally used in combination to produce a thorough test of the SUT.

"Bottom-up testing" involves testing a device or system by considering the UUT to be made up of a set of layers. The lowest layer is the simplest and most elementary. Successively higher layers depend on proper functioning of the layers underneath. All layers taken together make up the entire UUT. Layers are tested from lowest to highest. Once a layer is tested it is considered the hardcore for the next layer. This testing technique is based on a "guilty until proven innocent" assumption. That is, a section of hardware is not assumed to be functioning properly ("innocent" of causing errors) until its integrity is verified.

Bottom-up testing is a important in logic tests, where the logic must be tested in an order such that whenever a certain section of logic is being examined, all the logic that electrical signals must pass through before reaching the logic being tested should have itself been tested previously. Each section of logic is looike upon as another layer that depends on the previous sections or layers operating properly. Function tests also make use of bottom-up testing.

The bottom-up technique provides a thorough, systematic, step-by-step approach to hardware testing. However, using this method to validate an entire system can take a long time.

"Top-down testing" consists of first looking at the UUT as a whole, then gradually subdividing the UUT into its component parts until the failing part can be identified. This technique uses an assumption of "innocent until proven guilty." (The program assumes everything is operating properly unless errors are detected.) The problem with this approach is that a fault might exist in a portion of the hardware outside the testing scope of the diagnostic program. In this case the diagnostic program might not detect or might incorrectly diagnose the error, or might not be able to execute at all.

In practice, diagnosis of a hardware system suspected of containing faults uses a combination of top-down and bottom-up techniques. Often, bottom-up programs will be run in a top-down manner. Programs written to use the bottom-up technique are run in an order such that those that test the largest subsystems are executed first, followed by those that test devices tht previously executed programs point to as questionable.

1.11 MACROPROGRAMS AND MICROPROGRAMS

Many computer processors built today have two types of programming instructions. "Macro-instructions" make up the processor's machine language. These instructions are the "moves," "branches," arithmetic and boolean operators, and so on, that are used to manipulate data in specific memory locations. Programs that use these instructions, either directly through the use of an assembly language or indirectly by using a high-level language compiled down to an assembly language, are called "macroprograms." By far most programs written are macroprograms.

Beneath the macro-instructions is a set of "micro-instructions" used to implement the processor's machine language. Micro-instructions define the macro-level instructions, plus the registers defined by the machine language as existing "in the processor" (such as general purpose registers or a program counter). Micro-instructions do not execute in the system's main memory. Instead, they are loaded into and executed in a "writable control store" (WCS). (Micro-instructions also often exist in ROMs.) Since micro-instructions execute more rapidly than macro-instructions, it is sometimes useful for applications or systems programmers to use the micro-instruction set to create "microprograms."

Developers of diagnostic programs sometimes make use of microprogramming. Programs designed to test the processor will most likely use micro-instructions, executing them in a WCS. Some peripheral devices possess their own microprocessors. These devices usually also have ROMs in which diagnostic routines have been stored. In this case the diagnostic programmer writes a macrodiagnostic program that activates the microprograms residing in the ROM.

Parts of Chapter 2 discusses diagnostic microprograms further. However, most of this manual concerns diagnostic macroprograms.

CHAPTER 2
VAX DIAGNOSTIC PROGRAMS

2.1 INTRODUCTION

The discussion in Chapter 1 consisted of an overview of diagnostic programs. It did not deal with specific types of computer systems. This chapter introduces characteristics of diagnostic programs that are unique to VAX.

2.2 RUN-TIME ENVIRONMENTS FOR VAX DIAGNOSTIC PROGRAMS

VAX diagnostic programs are expected to operate in several run-time environments. These include user mode, standalone mode, and network environments. The user mode environment that supports execution of VAX diagnostic programs is the VAX/VMS operating system. For almost all devices supported by DIGITAL under VAX/VMS, a user mode diagnostic program must be developed. These programs are used extensively at customer sites so that diagnostic programs can be executed without bringing down VMS and thus locking other users out of the system under test.

Many VAX diagnostic programs are designed to execute in standalone mode. Manufacturing sites commonly use standalone programs, because if user mode programs were used it would be necessary to boot VMS just to run the diagnostic programs. Since standalone programs often provide better error detection than user mode programs, customer service personnel sometimes must use standalone programs at customer sites. Repair of failing device parts (after they have been identified and removed from the system under test) almost always involves the use of standalone diagnostic programs.

Networking environments have been developed for loading and executing diagnostic programs on VAX computer systems. One example is the Automated Product Test (APT) run-time environment, commonly used at manufacturing sites. Under this environment, a system under test is connected to a "mother" system that has copies of all diagnostic programs used. For each system to be tested, a "script" is built. A script is a file containing a list of diagnostic programs to be run, along with any run-time parameters that must be passed to the diagnostic program. The mother system reads this script and sends the appropriate diagnostic programs, one at a time, to the system under test. (This is referred to as "down-line loading.") Once a program has been sent to the system under test, it is started and monitored by the mother system, which will note any errors detected. When one program has completed execution, the next one listed in the script is sent down the line and started, until all programs in the script have been run. Programs executing on the system under test can only run in standalone mode.

Another example of a diagnostic network is APT/RD (for Remote Diagnosis), which provides a method of loading and monitoring diagnostic programs for diagnosing a system at a customer site. With APT/RD, a temporary communications link (via phone lines) is established between the system to be tested and a centrally located system belonging to DIGITAL and running the APT/RD software. Once the link is established, the central system can step through a script of diagnostic programs to attempt to diagnose the customer's system. Unlike the APT system used at manufacturing sites, though, the APT/RD system usually does not perform down-line loading of diagnostic programs. Instead, the programs must exist on some storage medium of the customer's system. They are loaded "locally" from that medium, on command from the central system. (Programs can be loaded down-line if necessary, for example when the diagnostic load medium of the system under test is malfunctioning.)

2.2.1 The VAX Diagnostic Supervisor

The previous chapter detailed the various uses and users a diagnostic program may encounter. The above section describes the run-time environments supported for VAX diagnostic programs. If a diagnostic program designer had to include proper interfaces for all users and environments in each program he or she developed, the task would become burdensome. For this reason the "VAX Diagnostic Supervisor" was developed for diagnostic macroprograms designed to run on VAX systems. The VAX Diagnostic Supervisor, or VDS, is a control program that will load, execute, and provide run-time services to diagnostic programs.

The VDS is divided into two major sections. One section is an interface between the VDS and the program user and is called the "human interface." The other is an interface between the VDS and the diagnostic program and is referred to as the "program interface."

The human interface consists of a command line interpreter (CLI) that receives and processes commands typed on a terminal by a user. Commands supported by the CLI include those for loading and running diagnostic programs, selecting which device units to test, displaying execution summaries, and controlling program looping.

The program interface consists of a set of service routines for service calls from the diagnostic program to the VDS, along with a mechanism for dispatching calls from the program to the proper routines in the VDS. These service routines provide the diagnostic program with convenient methods for performing device I/O, formatting error messages, controlling program loops, storing and retrieving system-specific device parameters, prompting the user for additional run-time parameters, and providing file management facilities.

The specific purposes of the VDS are to

1. Provide a common human interface for all diagnostic programs. With the large number of VAX diagnostic programs in existence, it is important that users not be required to spend time learning how to use each one. The VDS provides the user with a standard set of commands and functions that can be performed for all diagnostic programs.

2. Insulate the diagnostic program from the run-time environment. The VDS performs any communication that may be needed between the diagnostic program and the run-time environment, be that environment VMS (user mode), APT, APT/RD, or standalone.

3. Insulate the diagnostic program from processor-specific hardware differences. The VDS performs I/O initialization operations that are unique to the type of VAX processor being used. Thus the diagnostic program does not need to be concerned with knowing the type of VAX processor.

4. Make the programmer's job easier. Providing facilities for formatting error messages, controlling program looping, initiating I/O activity, manipulating files, and other services not only guarantees consistency among diagnostic programs from the user's standpoint but also greatly reduces the development effort necessary to produce a new program.

Later chapters of this manual discuss the VDS in detail. The VDS is introduced at this point in the manual because it plays a role in the VAX diagnostic strategy, discussed next.

The VDS is used by most, but not all, diagnostic macroprograms written for VAX systems, as will be shown in the following section.

2.3 INTRODUCTION TO THE VAX DIAGNOSTIC STRATEGY

In order to ensure a careful, comprehensive, step-by-step approach to diagnosing problems, a strategy for diagnosis of VAX systems has been developed. This strategy, generally referred to simply as the "VAX diagnostic strategy," has been to create a hierarchy of diagnostic programs based on hardcore requirements. Programs higher in the hierarchy require greater hardcore (they require a larger portion of the whole system to be operating).

Programs higher in the hierarchy are more likely to provide a versatile human interface and are less likely to require exclusive use of the system under test. On the other hand, programs lower in the hierarchy can test a device more thoroughly and thus provide a more accurate diagnosis. Hence it is best, when diagnosing a customer's system, to begin by using diagnostic programs of as high a level as possible and then drop down the hierarchy as necessary until a program is found that can detect the fault.

The diagnostic strategy has been implemented by creating various types, or "levels," of diagnostic programs. These levels were defined by:

1. Making use of the fact that the VAX hardware can be divided into various building blocks that, when connected together, create a whole system. These building blocks consist of

   - A system console
   - A CPU "cluster" consisting of processor, memory, and I/O channels
   - Peripheral devices

2. Remembering that some fault diagnosis can take place while a system's operating system is running.

3. Using the VAX diagnostic supervisor when appropriate.

By using these considerations, a set of five program levels has been defined. The diagnostic programs belonging to each level possess characteristics that differentiate them from programs belonging to the other levels. These characteristics are related to the program's run-time environment, hardware environment (see below), and method of performing I/O operations (see below).

Table 2-1 introduces each program level by listing its level name and the run-time environment associated with it.

Table 2-1 Program Levels and Run-Time Environments

Level	Run-Time Environment
1	Runs under VMS operating system.
2R	Runs under VDS in user mode only.
2	(Before 1982 only. No new programs are written for this level.) Runs under VDS in both user and standalone modes.
3	Runs under VDS in standalone mode only.
4	Runs in standalone without VDS.
5	Runs in WCS or system console, not in VAX main memory.

A program's "hardware environment" is the minimum hardware configuration on which the program will execute. (Do not confuse this with the program's hardcore, which is the minimum amount of hardware that must be functioning properly in order for the diagnostic program to execute. For example, the hardware environment of a program to test a disk controller would be the CPU cluster, buses connecting the controller to the cluster, and the controller itself, while the hardcore requirements in this case would be the CPU cluster and the buses.)

Three different hardware environments can be defined for VAX diagnostic programs. The hardware environments relate to the building blocks listed above. These environments are

1. Console environment. Consists of only the system console and the console load device.

2. CPU cluster environment. Consists of the system console, the VAX processor, main memory, and I/O channels.

3. System environment. Consists of the system console, the CPU cluster, and all attached peripherals. In other words, this is the whole system.

Figure 2-1 illustrates the hardware environments for a typical VAX hardware configuration.

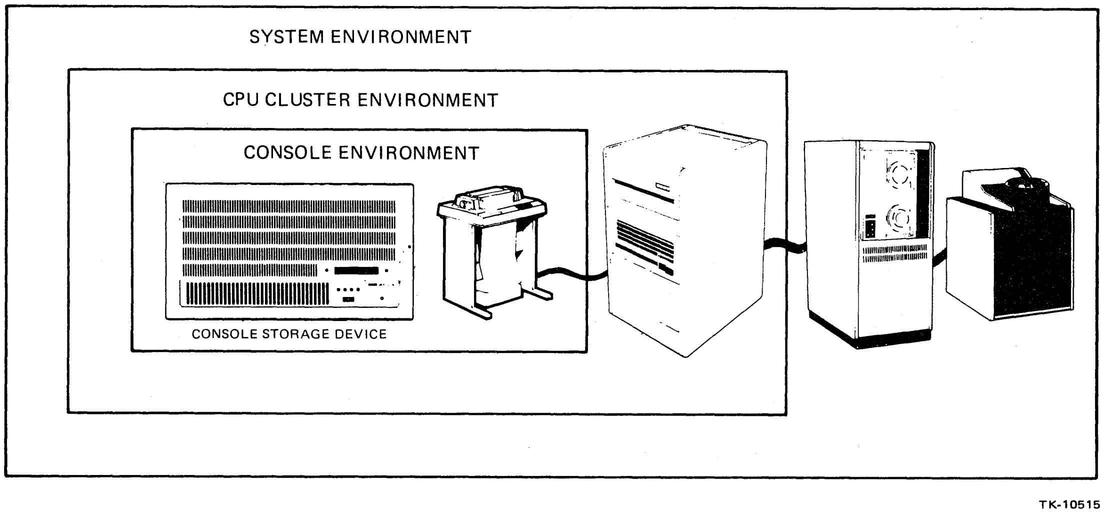

Figure 2-1 Hardware Environments for VAX Diagnostic Programs

The hardcore requirements and the hardware environments of the levels vary, with both increasing as the hierarchical level increases. Thus level 1 programs have the greatest hardcore requirements and largest hardware environments, while level 5 programs have the least and smallest.

The hardware environment and hardcore requirements of each program level are listed in Table 2-2.

Table 2-2 Hardware Environments and Hardcore Requirements

Level	Hardware Environment	Hardcore Requirements
1	System	Enough of system for VMS to execute
2R	Enough of system for VMS to execute, plus UUT	Enough of system for VMS to execute
2	Same as 2R in user mode. Same as 3 in standalone mode.	Same as 2R in user mode. Same as 3 in standalone mode.
3	CPU cluster, UUT, load device	CPU cluster, load device
4	CPU cluster	Console, subset of CPU cluster
5	Console, CPU cluster	Subset of console

2.4 METHODS OF PERFORMING I/O

Perhaps the most significant difference among the various program levels is the method of performing I/O operations. The various I/O methods are determined by the run-time environments existing for VAX diagnostic programs, since run-time environments generally put restrictions on I/O operations.

Before discussing the methods of performing I/O operations used by each level, it is necessary to define three types of I/O operations that are provided by the run-time environments.

- Physical I/O - In physical I/O operations, references can be made to the actual physically addressable units of the device or its storage medium, such as sectors on a disk, ignoring any block structuring or file structuring algorithms that may have been created for the device by software.

- Logical I/O - For logical I/O operations, a disk-type storage device may be referenced by addressing "logical" blocks on the device (blocks defined by software, such as the 512-byte blocks defined by VMS). Blocks are referenced relative to the beginning of the storage medium, and are numbered from Ø to n, where n is the last block. File structuring algorithms are ignored.

- Virtual I/O - With virtual I/O operations, software-defined blocks are referenced relative to the beginning of a file. They are numbered from 1 to n, where n is the last block in the file being referenced. This method of I/O takes full advantage of software-defined blocking and file structuring on the storage medium.

A more detailed discussion of the I/O types can be found in the VAX/VMS I/O User's Guide. That guide should be read before the development of a level 1 or 2R program is initiated.

In level 1 programs, I/O transfers are accomplished by issuing requests to the VMS operating system by using the $QIO system service call, or by using the Record Management Services (RMS) routines. Level 1 programs are expected to perform virtual, or sometimes logical, I/O operations, allowing them to execute without corrupting existing data on any storage media and thus not affecting the operation of any other processes executing concurrently.

For level 2R programs, I/O transfers are performed by issuing the $QIO service call, but in this case the VAX diagnostic supervisor fields the call. The VDS in turn passes the I/O request to VMS, where the I/O operation is actually performed. Level 2R programs are used for exercisers of devices or entire systems, and for functional testing of devices when it is desirable to not force other users off the system.

Physical I/O transfers are generally used in level 2R programs, since this type of transfer allows access to all areas of the device medium and thus provides maximum usage of the device's logic. It provides minimum device accesstime. Use of physical I/O implies that a "scratch" medium will have to be placed in the UUT in order to not corrupt valid user data, unless the device possesses special "maintenance cylinders" reserved for use by diagnostic programs. It also requires that the user of the program be granted special VMS "user privileges" (see the <u>VAX/VMS Command Language User's Guide</u>). While physical I/O is most often used, logical or even virtual I/O may be more appropriate in some cases.

Level 2 programs also perform I/O transfers using the $QIO service call, with the VDS fielding the call. In user mode, the VDS passes the request on to VMS. In standalone mode, the VDS itself services the request. It is not clear that one diagnostic program should be written to run in two different run-time environments, since the program is at best a compromise of the sometimes conflicting characteristics of the two environments (for example, ability to run with other users in user mode vs. ability to have unlimited system access in standalone mode). Also, the difficulty in maintaining this duplicity of functionality within the VDS is considerable. Therefore, LEVEL 2 DIAGNOSTIC PROGRAMS ARE NO LONGER BEING DEVELOPED. No new level 2 programs will be accepted for release.

Level 3 diagnostic programs perform their I/O operations directly. That is, they address the device's registers and field its interrupts. The VDS provides services for creating a "channel," or addressing path, to the device. This insulates the diagnostic program from the specific VAX processor type, enabling the programmer to create code that does not need to be concerned with I/O characteristics of particular processors. Since at this program level there are no software provisions for block formatting or file structuring, the only I/O type possible is physical. Logic tests (see Chapter 1) are written in level 3, since this level allows relatively comprehensive access to the device under test while also providing the VDS's common user and programming interfaces.

Level 4 programs are not used to test peripheral I/O devices and thus do not perform I/O operations. They should only be used to test those portions of the CPU cluster environment that are considered to be a part of the VAX Diagnostic Supervisor's hardcore.

Level 5 programs generally do not perform I/O operations, since they are generally microprograms used to test portions of the processor. However, some level 5 programs (specifically those diagnostic microprograms that test peripheral devices) may perform physical I/O operations.

Table 2-3 summarizes the I/O methods used in the various program levels. The table also indicates the types of diagnostic programs generally assigned to each level.

Table 2-3 I/O Methods and Program Types

Level	I/O Method	Types of Programs
1	Virtual or logical, using VMS QIO service.	System exercisers.
2R	Generally physical (but virtual or logical are allowed), using VMS QIO service.	Exercisers and function tests of peripheral devices.
2	Physical, using VMS/VDS QIO service.	Function tests of peripheral devices.
3	Physical, using program-defined I/O functions.	Function tests and logic tests of peripheral devices.
4	None.	Function and logic tests of CPU cluster.
5	None, or physical using program-defined functions.	Microprograms.

2.5 APPLYING THE VAX DIAGNOSTIC STRATEGY

Applying the VAX diagnostic strategy to a specific product usually implies developing a set of diagnostic programs to test the product.

2.5.1 Testing The CPU Cluster

The VAX CPU cluster is tested by a set of programs, existing at several program levels, as follows.

Level 5

- Console tests
- Processor tests
- Memory tests

Level 4

- VAX instruction set test (hardcore for VDS)
- Cache and translation buffer tests (VAX-11/750 only)

Level 3

- Memory tests (if no level 5 test possible)
- Channel adapter tests
- Cluster exerciser

This set of programs implements the VAX diagnostic strategy by providing a set of building blocks by which a system may be tested, starting with the level 5 basic processor tests and ending with the level 3 "cluster exerciser," which is a program meant to exercise all components of the cluster.

Level 5 programs may not exist for all VAX processors, since they are microprograms. Ideally (but not necessarily), microdiagnostic programs should be executed in a separate console processor ("front end"), making use of a writable control store (WCS). Low-cost VAX processors may not provide these features.

Most of the programs can be used on all types of VAX processors, so when a new processor is developed it is not necessary to produce a whole new set of programs for testing the new cluster. However, A new processor-specific module must be added to the cluster exerciser.

2.5.2 Testing Peripheral Devices

Thorough testing of a peripheral device requires the development of three different diagnostic programs. For each device type there will typically (but not necessarily) exist

1. A level 3 logic test
2. A level 3 function test
3. A level 2R function test

This group of programs implements the diagnostic strategy by providing a facility for producing very accurate and detailed identifications of fault conditions via the level 3 programs and by also providing a method by which the device may be tested without bringing down the customer's operating system via the level 2R program.

The level 3 logic test will provide the greatest detail of error resolution, indicating which section of logic is failing. This program will be used by technicians to repair bad logic boards, and will provide very high test coverage. Some devices contain ROM-resident microprograms ("self-tests") that perform logic testing, making a level 3 logic test unnecessary.

The level 3 function test will provide a comprehensive test of all of the device's functions. This program will be used to determine accurately whether or not a device is operating correctly. This is the definitive function test and provides very high test coverage. Level 3 function tests are usually required even if the device possesses self-testing capabilities, because self-tests generally aren't capable of complete detection of function failures.

The level 2R program will typically consist of a subset of the level 3 function test. It will test as much of the device's functionality as can be tested in the user (VMS) environment. The tests it contains are exact or approximate copies of tests existing in the level 3 program.

A typical sequence of use for these programs, when dealing with a system at a customer site, is as follows.

1. The customer (or field service) suspects a fault existing in the device.

2. The level 2R program is run to see if the error can be detected without stopping the operating system. If the error is found, go to step 4.

3. If the level 2R program cannot identify the fault, the operating system is brought down and the level 3 function test is run.

4. The fault is identified and the failing FRU is replaced. The operating system is then brought back up.

5. The failing FRU is brought back to DIGITAL, where the level 3 logic test, the level 3 function test, or perhaps a module test station is used to identify the failing logic on the FRU. The FRU is repaired.

2.6 GUIDELINES FOR WRITING VAX DIAGNOSTIC PROGRAMS

This sections contains general guidelines that should be followed when writing VAX diagnostic programs.

2.6.1 Level 1 Guidelines

Level 1 diagnostic programs are usually used as exercisers of the entire hardware system. Level 1 is used when it is necessary to cause various concurrent activities to take place, using numerous types of devices and other hardware and software resources provided by the system.

Since no standard human interface exists for level 1 programs, it is important for the program developer to design a convenient, "user-friendly" interface using such techniques as English-like commands, menus, and detailed "help" messages.

Error reporting will also be the responsibility of the program designer. However, much use can be made of the system software's error reporting facilities.

2.6.2 Level 2R Guidelines

Level 2R programs run under the VDS is user mode. They test device functionality and must test as many of a device's functions as can be performed under the constraints of the operating system.

I/O is performed by issuing QIO requests to the VDS. These requests are passed directly to VMS, which performs the indicated operations and returns an error status. Actual I/O activity is controlled by VMS device drivers. Full use should be made of the returned error information, which may include device register contents. All information made available should be displayed to the user via the VDS error reporting facilities.

The level 2R program should be written after the level 3 function test has been developed, since the level 2R program should be a subset of the level 3 program. Take the level 3 program, change the I/O method from the channel services of the level 3 (see below) to QIO calls, and remove any functions that the VMS operating system will not allow to be performed.

2.6.3 Level 2 Guidelines

DO NOT WRITE ANY NEW LEVEL 2 DIAGNOSTIC PROGRAMS.

2.6.4 Level 3 Function Tests Guidelines

Level 3 programs run under the VDS. There is no operating system software to limit the functionality or access rights of the diagnostic program. However, the program should use VDS channel services (discussed in the following chapters) for creating data paths to the device under test in order to eliminate the need for diagnostic programs to concern themselves with processor-specific details of bus adapter mapping.

I/O operations are initiated and interrupts are fielded by the diagnostic program. Since these programs have unlimited access to system hardware resources, detailed error messages can and should be created that contain dumps of pertinent registers.

Level 3 function tests should test every function the the device is capable of performing. Illegal orders and combinations should also be tried.

Not only should the data transfer functions be performed, but electromechanical functions should also be tested to assure that they operate within specified parameters and time intervals, as should the operater-related functions, such as setting the write-protect switch.

All timing operations must be performed by using the timing services provided by the VDS, since the VDS takes into account the type of VAX processor being used and corrects for timing differences between processor types.

2.6.5 Level 3 Logic Test Guidelines

Because logic tests are designed to help technicians repair malfunctioning logic boards, it is important that they provide good fragmentation of activity in the logic, causing as little overall activity as possible at a given point in execution time. Every effort should be made to concentrate electrical activity to one small section at a time. The extent to which this is possible depends on the particular hardware design, and it is often more of an ideal than an attainable goal.

The first section of logic to be tested should be that which is most likely to be depended on by other logic. Thus a general sequence of steps this type of program might contain would be as follows.

1. Test the interface between the device's controller and the I/O bus to which it is attached, including address decoding logic and logic used in referencing controller registers.

2. Test the controller's commands and the logic associated with each command, using the device's "maintenance mode" if applicable.

3. Test the data transfer functions of the device, again using maintenance mode.

In each step, invalid and borderline conditions should be checked. For example, purposely formatting data improperly, issuing illegal function codes, and making illegal references to device registers are techniques that can be used.

All timing operations must be performed by using the timing services provided by the VDS, since the VDS takes into account the type of VAX processor being used and corrects for timing differences between processor types.

2.6.6 Level 4 Guidelines

Level 4 programs are only used to test those parts of the system that belong to the VDS environment's hardcore, and that are not tested by level 5 programs. For example, level 4 programs are needed to test the VAX instruction set, the translation buffer, and cache of some (but not all) VAX processors.

If a new level 4 program needs to be developed, the following rules should be adhered to.

1. Use straight-line code (no subroutines). This makes it easier for the user to step through the program when necessary.

2. Use a minimum instruction set, at least at the beginning of the program.

3. Write the program in position-independent code, so that it may be loaded and executed in any section of memory in case there is a bad area of memory.

4. Create a section of code to handle unexpected interrupt conditions, such as machine checks or other traps.

5. Do not use any terminal I/O routines unless all the logic required to perform the I/O has been previously tested.

6. When an error is detected, execute the HALT instruction.

7. Use the general purpose registers (GPRs) to pass information to the user. For example, on a data comparison error, the expected and actual bit patterns can be placed in the GPRs.

8. Store the current test and subtest numbers in some location, such as address Ø, so the user can obtain them.

9. Provide very precise program documentation. Since no terminal displays can be provided, the user must be able to use the PC of a failure to find out exactly what type or error occurred and what was happening to cause the error. This information must be clearly indicated in the program listing.

2.6.7 Level 5 Guidelines

Level 5 programs are microprograms. Since the microcode and hardware design of each VAX processor type is different, there must be a separate set of level 5 programs for each processor type. Following are general rules that should be followed when developing diagnostic microprograms.

1. Diagnostic microprograms should always be designed to perform bottom-up testing.

2. Program loops should be as short as possible, in order to isolate electrical activity to as small an area of the logic as possible. Ideally, these loops should enable a technician to isolate a fault to the failing component.

3. Error reports should be precise enough for the technician to locate the code in a program listing. The listing should contain a clear description of what logic was being tested and which component(s) may be failing. Avoid referring to components by their "E-numbers," since these can change when ECOs are issued.

4. A level 5 program should be able to test every component except those requiring an external stimulus.

CHAPTER 3
THE STRUCTURE OF A VAX SUPERVISOR DIAGNOSTIC PROGRAM

3.1 INTRODUCTION

This chapter describes the composition of a diagnostic program designed to run under the VAX Diagnostic Supervisor (VDS). It discusses all of the functions that must be performed by the diagnostic program, such as device initialization and testing, error reporting, and input/output functions. It also provides an introduction to the macros detailed in Chapter 4 by indicating where within the diagnostic program the various macros should be used.

3.1.1 Overview Of The VAX Diagnostic Supervisor

The VDS is divided into three major segments, each segment performing a separate function. These segments are the command line interpreter, the dispatcher, and the system service routines.

- Command Line Interpreter

  The command line interpreter provides the human interface to the diagnostic program. It allows the diagnostic program user to select which programs to execute, which portions of that program to run, and which of the system's device units to test.

  The command line interpreter implements the commands described in the VAX Diagnostic Supervisor User's Guide.

- Dispatcher

  The dispatcher controls the operation of the diagnostic program. It is given control whenever the command line interpreter recognizes a START or RUN command. The dispatcher will call the various segments of the diagnostic program (such as the program's initialization code, tests, cleanup code, and summary routine, all of which are discussed in this chapter) at the appropriate times.

- System Service Routines

  The system service routines provide run-time services to the diagnostic program to facilitate many of the functions a diagnostic program must perform, such as I/O operations, error reporting, and event synchronization.

Figure 3-1 illustrates the VDS segments and their relationship to a diagnostic program.

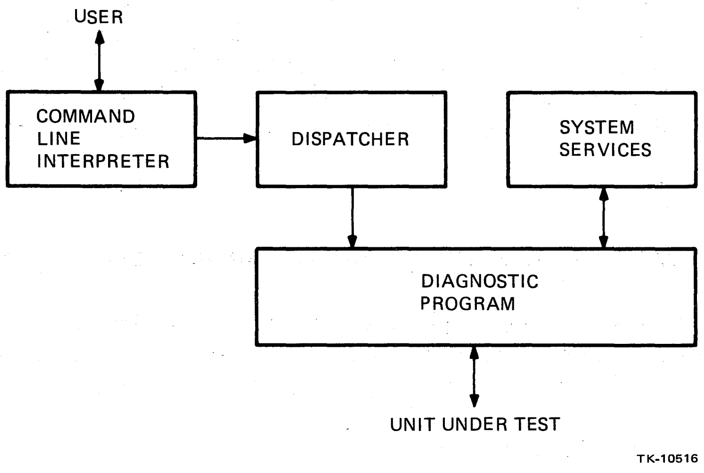

Figure 3-1 VDS Overview

3.1.2 Overview Of A VDS Diagnostic Program

Every diagnostic program must possess several major segments, as follows:

- Initialization Code

  This is code that is executed before a device unit is tested. It performs the operations necessary for creating a data link to the unit.

- Tests

  These are the actual device tests. They report any errors detected and provide the ability to create loops.

- Cleanup Code

  This code performs any operations that might be needed to leave the UUT in a state such that it is available to the next system user.

- Tables

  There are various tables residing in the diagnostic program for the purpose of enabling the VDS to control the diagnostic program's operation.

Additionally, a diagnostic program can possess other optional segments, such as

- A summary routine
- Error reporting routines
- Interrupt service routines
- Condition handling routines

Notice that the diagnostic program contains no dispatching mechanism. The program should be viewed simply as a set of low-level routines to be called by the VDS when needed.

Following are illustrations of program flow for both serial testing and parallel testing of devices. As will be seen as this chapter is read, these program flows are accomplished through interaction between the diagnostic program and the VDS.

Program Flow for Serial Testing:

```
Get RUN or START command.
Get passes_requested.
Passes_executed = Ø.
REPEAT
        Unit_number = Ø.
        REPEAT
                Call initialization code.
                Call selected tests.
                Call summary code.
                Unit_number = unit_number + 1.
        UNTIL unit_number = max_unit_number.
        Passes_executed = passes_executed + 1.
UNTIL passes_executed EQL passes_requested.
Call cleanup code.
```

Program Flow for Parallel Testing:

```
Get RUN or START command.
Get passes_requested.
Passes_executed = Ø.
REPEAT
        Unit_number = Ø.
        REPEAT
                Call initialization code.
                Unit_number = unit_number + 1.
        UNTIL unit_number = max_unit_number.
        Call selected tests.
        Call summary code.
        Passes_executed = passes_executed + 1.
UNTIL passes_executed EQL passes_requested.
Call cleanup code.
```

3.1.3 Memory Layout

Figure 3-2 shows the layout within memory of the various pieces of software existing when a VDS diagnostic program is executing. All addresses are virtual. In standalone mode, the virtual addresses are also the physical addresses, so the illustration represents a true picture of the actual program layout in memory. In user mode, memory management is in operation and thus the virtual addresses shown have no relation to the actual program layout in memory.

As can be seen in the figure, the base address of a diagnostic program is 2ØØ (hex). (When a diagnostic program is linked, a base address of 2ØØ (hex) must be explicitly specified.) The loadable image of a diagnostic program may not extend beyond virtual address F9FF (hex). Thus the maximum size for the loadable image of a diagnostic program is 63487 (decimal) bytes.

Addresses from FAØØ to FFFF are used by the VDS to communicate with APT. The VDS loadable image starts at virtual address 1ØØØØ (hex). At run time, the VDS occupies a contiguous portion of memory starting at 1ØØØØ (hex). The total size of this area depends on such parameters as the type of processor being used, memory size, and the number of attached devices.

Two areas of memory are used to allocate buffer space to diagnostic programs. The first area is any space that may exist between the top of the diagnostic program's loadable image and address FAØØ (hex). The second (and generally larger) area consists of addresses above the highest address used by the VDS. Allocation of this buffer space to a diagnostic program is discussed in Section 3.13.3, Memory Allocation.

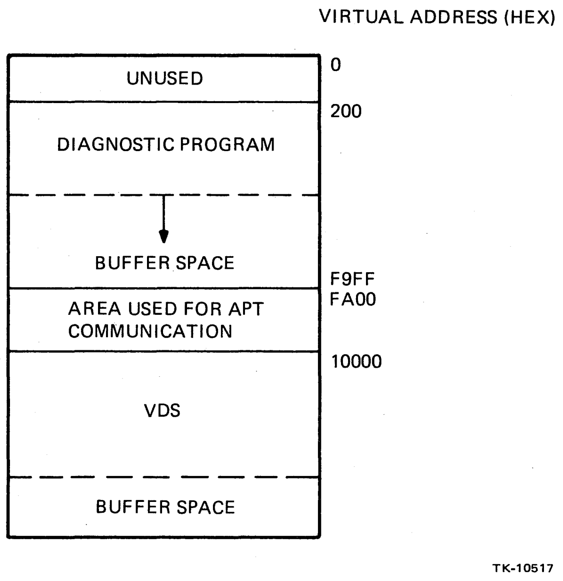

Figure 3-2 VDS Memory Layout

3.1.4 Introduction To The Macros

All linkages between the diagnostic program and the VDS are defined by a set of macros. These macros can be divided into four main groups.

- Program Structure Macros

  This group consists of those macros used to define the various sections, tables, and data structures making up the diagnostic program. For example, every test must be delimited by the $DS_BGNTEST and $DS_ENDTEST macros. Using the program structure macros enables the VDS dispatcher to locate and call the initialization code, tests, and cleanup code. Most of the macros in this group are required to exist in every diagnostic program.

- Program Control Macros

  These macros are used to affect the program's execution path and provide such facilities as looping and branch-on-error. For example, the $DS_CKLOOP macro can be used to define the upper bound of a program loop.

- System Service Macros

  This group is used to call service routines. An example service macro is $DS_WAITMS, which can be used to cause a program delay of a specified number of milliseconds.

- Symbol Definition Macros

  This is a set of macros that define global symbols used by the other macros, the VDS, and the diagnostic program. For example, the $DS_HDRDEF macro defines symbols for the locations within the diagnostic program's header (see Section 3.3.1).

This chapter will not give detailed descriptions of the macros, but it will indicate when and where each macro (except the symbol definition macros) should be employed. The macros are discussed in detail in Chapter 4.

3.2 P-TABLES

3.2.1 Introduction To P-Tables

In order to test a device, a diagnostic program must have access to the device's characteristics. Since some device characteristics are system-specific, it is impossible to define them permanently in the diagnostic program. Instead, it is necessary to provide a means by which these system-specific characteristics can be specified at run time. The VDS provides the "hardware parameter tables," or simply "p-tables," for this purpose.

A p-table is a data structure containing the information about a device that is needed in order for a diagnostic program to access the device. P-tables are constructed by the VDS when the program user types the ATTACH command (refer to the VAX Diagnostic Supervisor User's Guide). Each time the ATTACH command is used, a new p-table is created. Once the VDS has created the p-tables, the diagnostic program can reference the tables to obtain information necessary for testing a UUT. Thus the burden of determining device characteristics is removed from the diagnostic program itself.

When the user attaches a device, one of the parameters he or she must specify is the device's "link." The link is the piece of hardware to which the device is connected. The link must have been previously specified with another ATTACH command so that its p-table already exists. A set of ATTACH commands will result in a tree structure of device links. The root of this tree is a pseudo-device called HUB. This pseudo-device was created because the actual hardware interconnect existing depends on the type of processor (for example, the SBI on the VAX-11/78Ø). In general, processors and buses are linked to HUB, controllers are linked to buses, and device units are linked to controllers. Figure 3-3 illustrates the manner in which p-tables describe a hardware system.

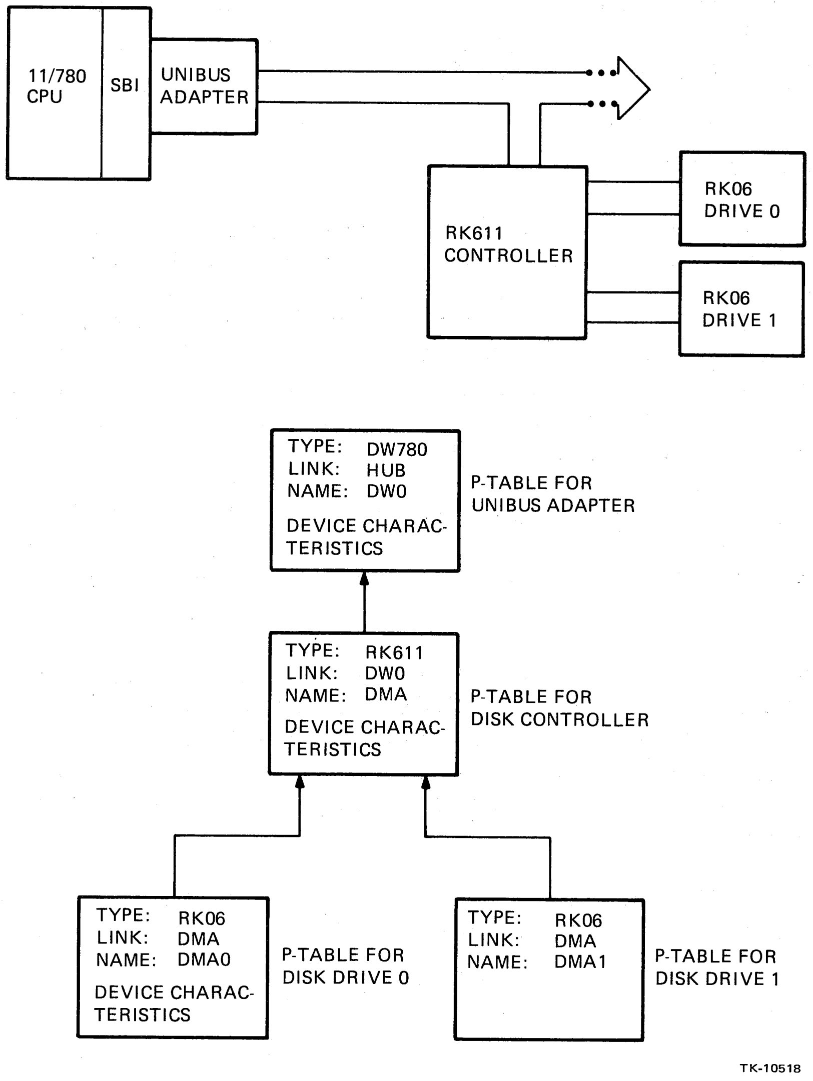

Figure 3-3 Sample Hardware Configuration and Associated P-Tables

The p-table for a particular device will contain the information provided by the ATTACH command arguments. Each p-table will contain the following standard information:

- Device type - This is the product name for the device, such as RK06 or TM03.
- Device's generic name - This is the name with which the device will be referred to, such as DRB1 or DMA0.
- Address of p-table for device's link
- Device characteristics - The types of information that must be included in a p-table to sufficiently describe a device depend on both the type of device and its link. For example, devices linked to a UNIBUS require the UNIBUS CSR address and bus request level, plus the device's interrupt vector address.

3.2.2 P-Table Format

P-tables have a standard format. Each p-table is divided into two sections. The first section contains device-independent fields. All p-tables for all devices contain these fields. Each device-independent field in the p-table has a mnemonic assigned to it which can be used by the diagnostic program when these fields are referenced. The second section of the p-table contains device-dependent information. This section is unique to the type of device being described.

Figure 3-4 shows the standard layout of all p-tables.

Following is a description of the device-independent p-table fields.

HP$Q_DEVICE - A VMS-type quadword descriptor of the device name string (see HP$T_DEVICE below). That is, the first word of the field contains the length (number of characters) in the device name string, the next word is unused, and the following longword contains the address of the string (the address of HP$T_DEVICE).

HP$W_SIZE - The size of the p-table in bytes. This includes both the device-independent and the device- dependent p-table fields.

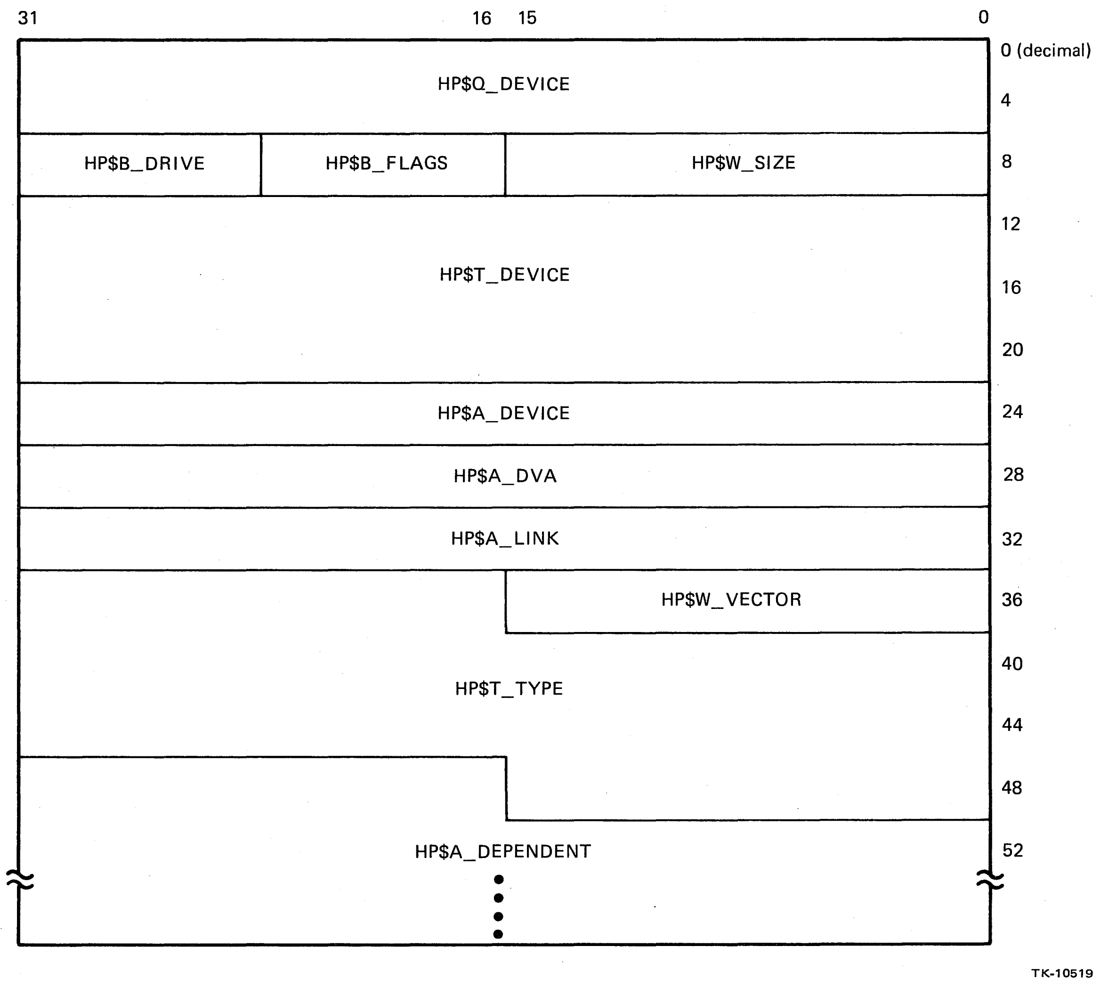

Figure 3-4 P-Table Layout

HP$B_FLAGS - Flags used by the VDS when the device is initialized. Flags are defined as follows.

- HP$M_ALLOC - (bit Ø) - If set, indicates that the VDS must request VMS to allocate ($ALLOCATE system service) the device before it can be tested in user mode.
- HP$M_WASALL - (bit 1) - Set by VDS if a device has been successfully allocated.
- (Bits 2-7) - Unused.

HP$B_DRIVE - The unit number of the device. This is the number appearing at the end of the device's generic name, such as '7' in '_TTA7'.

HP$T_DEVICE - An ASCII string representing the device's generic name. All device names begin with '_', as in '_RHØ'.

HP$A_DEVICE - The virtual address of the lowest-addressed device register. The type of register being pointed to depends on the device type. For example, it would be a CSR for a UNIBUS device, a configuration register for an SBI device, and so on.

The address must be virtual, in Pl space (bit 3Ø set). This is because when memory management is enabled in standalone mode, the VDS maps all physical I/O addresses through virtual Pl space.

HP$A_DVA - This is the base of the virtual address space assigned to this device. Devices linked to this device will have address assignments relative to this base address. When the VDS constructs a new p-table for a device linked to this one, it copies this field into the linked device's HP$A_DEVICE field. When the device address for the new device is fetched from the user, it can be added to the base address already in HP$A_DEVICE.

The address must be virtual, in Pl space (bit 3Ø set). This is because when memory management is enabled in standalone mode, the VDS maps all physical I/O addresses through virtual Pl space.

The HP$A_DVA field is not always relevant. An example of its use is the case of UNIBUS adapters. Each UNIBUS is assigned to a certain base address. The addresses of devices connected to a particular UNIBUS are added to the UNIBUS's base address to obtain the device's actual physical address. A UNIBUS's base address is stored in the HP$A_DVA field for a UNIBUS's p-table. When a controller is linked to the UNIBUS, its HP$A_DEVICE field will be initialized to the value contained in the UNIBUS's HP$A_DVA field. Subsequently, the user will be prompted for the controller's 18-bit address. This address can be stored in the low-order 18 bits of HP$A_DEVICE to result in a full physical address for the controller.

HP$A_LINK - The address of the p-table for the device to which this one is linked. If this device is linked to HUB, the field contains Ø.

HP$W_VECTOR - If relevant, contains the vector address through which the device will interrupt. This address is an offset into the System Contol Block (SCB).

HP$T_TYPE - Contains a counted ASCII string representing the device type, such as DW78Ø, RH78Ø, or RK611.

HP$A_DEPENDENT - The first location of the device-dependent section of the p-table.

The HPW_SIZE, HPQ_DEVICE, HPB_DRIVE, HPT_DEVICE, HP$A_LINK, and HP$T_TYPE fields are filled in automatically by the VDS. The other fields are loaded (if needed -- not all fields are relevant to all devices) in accordance to directions contained in the p-table descriptors (see below).

The fields within the device-dependent section also have mnemonics, but they are unique to the device (see below).

3.2.3 P-Table Descriptors

3.2.3.1 Introduction To P-Table Descriptors - The VDS builds a p-table by referring to a "p-table descriptor." This is a set of instructions that indicate the size and format of the device-dependent p-table fields. When a user types an ATTACH command, the VDS will refer to the p-table descriptor of the specified device type in order to determine how to construct the device-dependent fields of a particular p-table.

Following is a sample ATTACH command dialogue. Portions of the dialogue that are typed by the user the VDS are underlined.

```
DS> ATTACH

Device type? RK611

Device link? DWØ

Device name? DMA

CSR? 77744Ø

VECTOR? 21Ø

BR? 4
```

In the sample, the first three prompts fill in device-independent fields of the p-table. These prompts are generated by the VDS and will be displayed every time the ATTACH command is used. The last three prompts are device-specific. These prompts are defined by the p-table descriptor for the RK611.

Instructions within the p-table descriptor specify to the VDS the following types of information.

- The p-table's size
- The device type
- A prompting message for each device-dependent hardware parameter to be stored in the p-table
- The format in which user response to the device-dependent prompts is to be interpreted
- The p-table field in which the responses to the device-dependent prompts are to be stored

3.2.3.2 Location of P-Table Descriptors - P-table descriptors generally reside in the VDS. When a diagnostic program is written to test a device for which the VDS does not possess p-table descriptors, it is the reponsibility of the diagnostic program developer to also create a p-table descriptor for the device. This descriptor will then be incorporated into the VDS.

Note: It is important to work in cooperation with the VDS support group when developing a p-table descriptor.

P-table descriptors may also be included in the diagnostic program. When processing an ATTACH command, the VDS will first check the diagnostic program to see if a p-table descriptor exists for the specified device type. If none exists, the VDS will check its own p-table descriptors to locate the appropriate one. Thus, a descriptor residing in a diagnostic program will have precedence over a descriptor for the same device residing within the VDS.

Including the descriptors in a diagnostic program has several disadvantages.

- They can only be used by the diagnostic program in which they are defined.
- The devices they describe cannot be attached unless the diagnostic program has been loaded.

- These diagnostic programs will not be executable under APT. Other special environments, such as Customer-Runnable Diagnostics (CRD) may also place prohibitions on execution of programs containing their own p-tables.

- The autosizer program will only support devices for which the descriptors reside in the VDS.

When development of a program for a new device begins, the p-table descriptor should be first placed in the diagnostic program until the descriptor design, and indeed the design of the device hardware itself, has been solidified. Once the p-table's design is certain, it can be included in the VDS. Only in rare instances should it be necessary to release a diagnostic program that contains its own p-table descriptors.

3.2.3.3 Creating P-Table Descriptors - The following general guidelines should be followed when creating a p-table descriptor.

- Each user prompting message should provide a clear indication of what information the user must provide.

- Responses should be requested in a format that is relevant to the particular type of data being requested. For example, UNIBUS addresses should be formatted in octal instead of hexadecimal, since that is their normal format.

- Only include information that is needed for referencing a device. This information may include such items as the device's address, interrupt vector, BR or TR level, and so on. Do not include information that will only be used by one diagnostic program; remember that a p-table for a particular device will be used by all diagnostic programs that test that device. Information needed by a particular program or test should be obtained via the $DS_ASKxxxx macros (see Chapter 4).

There are two steps to creating a p-table descriptor. First, a "skeleton" for the p-table's device-dependent fields must be defined. This skeleton is a representation of the memory space required for the p-table. When the VDS builds a p-table in response to an ATTACH command, skeletons of both the device-independent and device-dependent fields are copied into a dynamic memory storage area, and the fields are filled in with the proper information. The MACRO-32 skeleton for the device-dependent fields is defined by using the $DEFINI, $DEF, and $DEFEND macros, which are defined in the VMS system library LIB.MLB. An example skeleton is as follows:

```
.MACRO $DS_RK611_DEF $GBL
        $DEFINI      RK611, $GBL, HP$A_DEPENDENT
        $DEF         HP$L_RK611_CSR, .BLKL, 1  ;18-bit CSR address
        $DEF         HP$B_RK611_BR, .BLKB, 1   ;UNIBUS BR level
        $DEF         HP$K_RK611_LEN
        $DEFEND      RK611, $GBL, DEF
.ENDM $DS_RK11_DEF
```

Note: The final $DEF statement in the example defines the length of the p-table.

The BLISS-32 version of this example is:

BLISS-32:

```
   $DS_RK611_DEF=
           SET
           HP$L_RK611_CSR = [HP$K_LENGTH+Ø,Ø,32,Ø],
           HP$B_RK611_BR  = [HP$K_LENGTH+4,Ø,8,]
           TES;
```

This skeleton represents the device-dependent fields for a p-table of an RK611 controller. Each field is assigned a mnemonic. There are two fields, named HP$L_RK611_CSR and HP$B_RK611_BR. (See Section 3.2.3.4 for field naming conventions.)

Notice that the MACRO-32 skeleton is defined as a macro. When the p-table descriptor is added to the VDS, this macro is made available to diagnostic programs. After the diagnostic program calls this macro it can reference the p-table fields by using the mnemonics. (See the MACRO-32 example in Section 3.2.4.)

Notice that the BLISS-32 skeletion is simply a field declaration statement. The BLISS-32 example in Section 3.2.4 indicates how the field declaration is used by a diagnostic program.

The second step in creating a p-table descriptor involves generating the instructions that the VDS will use when filling in the device-dependent fields. Also, instructions must be developed for filling in the following device-independent fields, if they are relevant to the device: HPA_DEVICE, HPA_DVA, HP$B_FLAGS, and HP$W_VECTOR.

These instructions are produced by using a set of macros. The macros make use of a temporary storage location referred to as the "value register." Certain macros cause information to be read from the ATTACH command line and placed into the value register. Other macros can manipulate the value register's contents, and still others can transfer those contents into fields of the p-table. The p-table descriptor macros are as follows:

- $DS_$INITIALIZE - This is the first macro in any p-table descriptor. It indicates the device type, the p-table size, the maximum number of units allowed, and the name of the device driver used for level 2 diagnostic programs (see Chapter 2).

- $DS_$NAME - Specifies a format to which the device unit's generic name must conform.

- $DS_$DECIMAL, $DS_$OCTAL, $DS_$HEX, $DS_$STRING, $DS_$LOGICAL - Each of these macros is used to obtain hardware parameters from the user when an ATTACH command is typed. The exact macro to use depends on the format in which the input string of the particular parameter is to be interpreted. For example, the $DS_$DECIMAL macro should be used if the user is to type a decimal number, and the $DS_$STRING macro is used if an alphabetic string is to be typed. For each of these macros, the programmer specifies a user prompting message. Information is read from the ATTACH command line and stored in the value register.

- $DS_$STORE, $DS_$ADD, $DS_$FETCH - These macros are used to manipulate data that was received from a $DS_$DECIMAL, $DS_$OCTAL, $DS_$HEX, $DS_$STRING, or $DS_$LOGICAL command and placed in the value register. $DS_$STORE will place the value register's contents into a field within the p-table. $DS_$ADD will add the value register's contents to the current contents of a field. $DS_$FETCH will retreive data from a field and place it, right-justified, in the value register.

- $DS_$COMPLEMENT, $DS_$CASE, $DS_$LITERAL - These macros are used to alter the contents of the value register.

- $DS_$END - The $DS_$END macro is used to indicate the end of a p-table descriptor.

Example 3-1 shows how these macros are used.

```
$DS_$INITIALIZE RK611, RK611$K_LEN, 0, DM
$DS_$NAME       PTD$M_CONTROLLER, DM
$DS_$OCTAL      CSR, 760000, 777776
$DS_$STORE      HP$L_RK611_CSR, 0, 32
$DS_$STORE      HP$A_DEVICE, 0, 18
$DS_$OCTAL      VECTOR, 2, 776
$DS_$STORE      HP$W_VECTOR, 0, 9
$DS_$DECIMAL    BR, 4, 7
$DS_$STORE      HP$B_RK611_BR, 0, 8
$DS_$END
```

Example 3-1 P-Table Descriptor for RK611 Disk Controller

This example will produce the dialogue illustrated in Section 3.2.3.1. Explanations of the macro arguments can be found in Chapter 4.

This example will:

1. Cause the VDS to request the user to type a CSR address.

2. Store the CSR address in HP$L_RK611_CSR, bits Ø through 31, and in HP$A_DEVICE, bits Ø through 17.

3. Cause the VDS to request the user to type a vector address.

4. Store the vector address in HP$W_VECTOR, bits Ø through 8.

5. Cause the VDS to request the user to type a BR level.

6. Store the BR level in HP$B_RK611_BR, bits Ø through 7.

Following is a more complex example -- the p-table descriptor for the RH78Ø (MASSBUS adapter for the VAX-11/78Ø). Example 3-2 contains the MACRO-32 and BLISS-32 skeletons.

```
MACRO-32:

.MACRO  $DS_RH780_DEF       $GBL
        $DEFINI    RH780,$GBL,HP$A_DEPENDENT
        $DEF       HP$B_RH780_TR,.BLKB,1    ; TR number of adapter
        $DEF       HP$B_RH780_BR,.BLKB,1    ; BR level of adapter
        $DEF       HP$K_RH780_LEN
        $DEFEND    RH780,$GBL,DEF
.ENDM   $DS_RH780_DEF


BLISS-32:

$DS_RH780_DEF=
        SET
        HP$B_RH780_TR = [HP$K_LENGTH+0,0,8,0],
        HP$B_RH780_BR = [HP$K_LENGTH+1,0,8,0]
        TES;
```

Example 3-2 P-Table Skeletons for RH780 MASSBUS Adapter

Example 3-3 presents the p-table descriptor for the RH780. This descriptor causes the following events to occur:

1. The VDS will request the user for an SBI transfer request (TR) level.

2. The TR level will be stored in HP$B_RH780_TR, bits 0 through 7.

3. The TR level is also stored in HP$A_DEVICE, bits 13 through 16.

4. The TR level is also stored in HP$W_VECTOR, bits 2 through 5.

5. The VDS will request the user for a BR level.

6. The BR level is stored in HP$B_RH780_BR, bits 0 through 7.

7. The BR level is also stored in HP$W_VECTOR, bits 6 through 7.

8. The value register is loaded with the value "6."

9. The "6" is placed in HP$A_DEVICE, bits 28 through 31. (This will create a virtual P1 space address for the physical address 20000000 (hex).)

10. The contents of HP$A_DEVICE is loaded into the value register.

11. This value is written into HP$A_DVA.

12. The value register is loaded with the value "1."

13. The "1" is placed in HP$A_DVA, bit 10.

14. The "1" is placed in HP$W_VECTOR, bit 8.

```
$DS_RH780_DEF
$DS_$INITIALIZE RH780,RH780$K_LEN,8
$DS_$NAME       PTD$M_UNIT, RH
$DS_$DECIMAL    TR,1,15
$DS_$STORE      HP$B_RH780_TR,0,8
$DS_$STORE      HP$A_DEVICE,13,4
$DS_$STORE      HP$W_VECTOR,2,4
$DS_$DECIMAL    BR,4,7
$DS_$STORE      HP$B_RH780_BR,0,8
$DS_$STORE      HP$W_VECTOR,6,2
$DS_$LITERAL    6
$DS_$STORE      HP$A_DEVICE,28,4
$DS_$FETCH      HP$A_DEVICE,0,32
$DS_$STORE      HP$A_DVA,0,32
$DS_$LITERAL    1
$DS_$STORE      HP$A_DVA,10,1
$DS_$STORE      HP$W_VECTOR,8,1
$DS_$END
```

Example 3-3 P-Table Descriptor for RH780 MASSBUS Adapter

Note that several fields of a p-table created from this descriptor require several steps. For instance, the HP$A_DEVICE field is constructed by:

- Setting the high order four bits to "6" (bit 30 indicates P1 space and bit 29 indicates VAX-11/780 I/O addresses). Note: This is an important step to remember. The VDS maps P1 addresses to I/O space when memory management is turned on. Therefore device addresses must be constructed as virtual addresses in P1 space.

- Using the TR level to set bits 13 through 16, which will select the address space for the indicated TR level.

- In this case the contents of HP$A_DEVICE are copied into HP$A_DVA, and bit 10 of HP$A_DVA is set.

(Note: When a device is attached to this RH780 adapter, the VDS will initialize the HP$A_DEVICE field of that device to the contents of the adapter's HP$A_DVA field. The p-table descriptor for the device must be careful not to overwrite bits in HP$A_DEVICE that were loaded in HP$A_DVA of the adapter. This example illustrates that it is important, when designing a p-table descriptor, to first obtain copies of the descriptors for all possible link devices. The design of the new p-table must be coordinated with p-table design for these link devices.)

3.2.3.4 Creating Names for Device-dependent Fields - For easy reference, all device-dependent fields of a p-table should be assigned mnemonics. These mnemonics can then be used by the p-table descriptor macros $DS_$STORE, $DS_$ADD, and $DS_$FETCH. Also, the diagnostic program can use the mnemonics when it references a p-table.

The field naming conventions for p-tables follow the VMS standard for data structure naming conditions. The field name begins with the name of the data structure (HP), followed by a dollar sign ($), followed by the data type specifier (L for longword, W for word, and so on, as listed in Table 5-1), followed by an underscore (_), followed by the field name. For example, the RK611 controller's p-table has a device-dependent field for storing the controller's CSR address. This field is named HP$L_RK611_CSR.

Note: Many p-table descriptors were developed before this standard was implemented. Previously, the standard was for field names to consist of the device name, dollar sign, data type, underscore, field name, as in 'RK611$L_CSR'. If the mnemonics for the device-dependent fields of a particular p-table do not match the current standard, then they will conform to this old standard.

3.2.4 Referencing P-Tables from a Diagnostic Program

A diagnostic program gains access to a p-table by using the $DS_GPHARD macro. The program indicates a unit number as an argument to the macro, and the VDS will pass to the diagnostic program the base address of the p-table for that unit. The program can then access fields within the p-table by using the base address and the predefined field mnemonic offsets (see above). The $DS_GPHARD macro is discussed further in the description of initialization code (see Section 3.5).

Example 3-4 provides an example of referencing a p-table in a MACRO-32 program. Notice that before the p-table field mnemonics can be referenced, the macros which define them must be called ($DS_HPODEF for the device-independent fields and, in this case, $DS_RK611_DEF for device-dependent fields).

```
                .
                .
                .
                $DS_HPODEF        ; Define device-independent p-table fields
                $DS_RK611_DEF     ; Define RK611 device-dependent fields
                .
                .
LOG_UNIT:       .BLKL    1                      ; Place to store log. unit no.
PTABLE:         .BLKL    1                      ; Place to store pointer
DEV_NAM:
                .ASCIC   \RK611\                ; Ascii name of desired device
                .
                .
                .
        INCL      LOG_UNIT
                $DS_GPHARD_S DEVNUM=LOG_UNIT, - ; Get Ptable for next log. unit
                             ADRLOC=PTABLE      ; .. address in PTABLE
                CMPL     R0, DS$_NORMAL         ; If all units done
                BNEQ     40$                    ; then branch to re-init.
10$:            MOVL     PTABLE, R2             ; Use R2 as structure pointer
                MOVAL    DEV_NAM, R0            ; Set up pointer to type
                CMPL     (R0), HP$T_TYPE(R2)    ; Check length and first 3
                BNEQ     20$                    ;   characters of type.
                CMPW     4(R0), HP$T_TYPE+4(R2) ; Check last 2 characters
                BEQL     30$                    ; If it matches, OK
20$:            $DS_ABORT ARG=TEST              ; If not RK611, abort test
30$:            MOVZBL   HP$B_RK611_BR(R2), R10 ; Set R10 to BR level
                MOVL     HP$A_DEVICE(R2), R11   ; Set R11 to CSR address
                .
                .
                .
40$:
```

Example 3-4 Referencing P-Tables in MACRO-32

(Note: This code is meant only to show an example of the use of p-table mnemonics. The function performed does not need to be included in a real diagnostic program.)

Example 3-5 is a BLISS-32 example of referencing p-tables. Notice that before p-table mnemonics can be referenced, a pointer must be declared (in this case called 'PTABLE') using the $DS_HPO_DECL macro and including the field declaration for the device type being tested (an RK611 in this case).

Notice that the 'HP$T' prefix fields expand only to addresses. To do data fetches from these fields, explicit field references must be made (as in the example for HP$T_TYPE).

```
          .
          .
          .
      BEGIN

      LOCAL
          LOG_UNIT,                         ! Place to store log. unit no.
          BR_LEVEL,                         ! Place to store BR level
          STATUS,                           ! Status return from service calls
          CSR : REF VECTOR [, LONG],     ! Device register access
          PTABLE : REF $DS_HPO_DECL ($DS_RK611_DEF); ! Address of Ptable

      BIND
          DEV_NAM = UPLIT BYTE (%ASCIC'RK611');       ! Ascii name of device
          .
          .
          .

      ! ++
      ! Get the address of the p-table for the next logical unit number.
      ! If the $DS_GPHARD call returns successfully, do the processing.
      ! --

      LOG_UNIT = .LOG_UNIT + 1;
      STATUS = $DS_GPHARD (UNIT=.LOG_UNIT,          ! Get Ptable
                        RETADR=PTABLE);

      IF .STATUS EQL DS$_NORMAL
      THEN
          BEGIN                                     ! $DS_GPHARD worked

          IF .(PTABLE [HP$T_TYPE]) NEQ .DEV_NAM    ! Validate type
              OR .(PTABLE [HP$T_TYPE] + 4)<0, 16> NEQ .(DEV_NAM + 4)<0, 16>
          THEN
              $DS_ABORT (ARG = TEST);               ! Abort test if wrong device

          BR_LEVEL = .PTABLE [HP$B_RK611_BR]; ! Get bus request level
          CSR = .PTABLE [HP$A_DEVICE];              ! Get CSR pointer
          .
          .
          .
          END
      ELSE
          BEGIN                             ! $DS_GPHARD returned error.
          .
          .
          .
          END
      END;
```

Example 3-5 Referencing P-Tables in BLISS-32

(Note: This code is meant only to show an example of the use of p-table mnemonics. The function performed does not need to be included in a real diagnostic program.)

3.2.5 Attaching From Within The Diagnostic Program

It may occasionally be necessary for a diagnostic program to explicitly attach a device instead of depending on the program user to issue an ATTACH command. For example, if the program is going to access a file (see Section 3.15, File Management), the device on which the file resides must be attached before it can be referenced. In this case, the diagnostic program can issue the $DS_ATTACH macro. This macro serves exactly the same function as the ATTACH command.

3.3 DIAGNOSTIC PROGRAM GLOBAL DATA STRUCTURES

The data structures described here are used to pass information about the diagnostic program to the VDS.

3.3.1 Diagnostic Program Header

The diagnostic program header is a data block containing various types of information needed by the VDS, such as the program's title and pointers to the various areas of the program that the VDS must call during program execution.

The header is allocated by using the $DS_HEADER macro. This macro will be at the beginning of the program. It is the first (lowest) area of memory allocated to the program. When the program is loaded by the VDS, the header's first address will be location 2ØØ (hex).

Some header entries must be initialized at assembly time using macro arguments. Other entries are filled in by the linker. The diagnostic program should not alter or reference any header entries during program execution.

3.3.2 Dispatch Table

The dispatch table is the means by which the VDS dispatches program control to the various tests in the diagnostic program. The table consists of a list of addresses of the tests.

The dispatch table is defined by the $DS_DISPATCH macro. The table's entries (test addresses) are generated when the diagnostic program is linked.

3.3.3 Program Sections Table

The program sections table contains character strings defining the names of the program sections (see Section 3.8.3), as well as pointers to the sections. The VDS uses this table when the user specifies a section name with a RUN or START command, in order to determine if the specified section exists and where it is located.

The program sections table is defined with the $DS_SECTION macro.

3.3.4 Device Mnemonics List

The device mnemonics list is the means by which the VDS determines what types of devices the diagnostic program is capable of testing. When a RUN or START command is issued by the user, the VDS compares the device types in the device mnemonics list against the types of the SELECTed devices (see the VAX Diagnostic Supervisor User's Guide) to determine if there are any SELECTed devices that the program can test. The list has two kinds of entries. Entries can either be addresses of counted ASCII strings or addresses of p-table descriptors.

For device types having p-table descriptors defined within the VDS, the device mnemonics list entry will be the address of an ASCIC string representing the device type (for example, RKØ6, TMØ3).

For device types having p-table descriptors defined within the diagnostic program, the device mnemonics list entry will be the address of the device's p-table descriptor.

The device mnemonics list is created and formatted by the $DS_DEVTYP macro.

3.4 PROGRAM PASSES AND SUBPASSES

Most diagnostic programs contain several tests (see Section 3.8.1). It is common for a system-under-test to have several units of the type of device being tested.

One complete execution of all selected tests on all selected units is one program "pass."

One complete execution of all selected tests on one selected unit is one "subpass."

For a diagnostic program employing serial testing (see Chapter 1), each pass will consist of one or more subpasses.

For a diagnostic program employing parallel testing (see Chapter 1), each pass will contain only one subpass, since all devices are tested concurrently.

3.5 INITIALIZATION CODE

Prior to the execution of a group of tests on a particular device, the diagnostic program generally must perform some initialization functions. These functions include obtaining the address and other needed characteristics of the next unit to be tested, creating a data path to the device, and initializing program buffers and counters. These functions are placed in a portion of the diagnostic program known as the "initialization code." This code is delimited by the macros $DS_BGNINIT and $DS_ENDINIT. The VDS will dispatch control to this code at the beginning of each program subpass, before calling any of the tests.

3.5.1 Format Of The Initialization Code

The format of the initialization code depends on whether the diagnostic program performs serial testing or parallel testing of the units (see Chapter 1). For serial testing, one unit will be initialized each time the initialization code is executed. The VDS will dispatch control to each selected test and then call the initialization code again so that the next unit may be initialized. For parallel testing, each execution of the initialization code should cause all units to be initialized. When the VDS calls the tests, all units will be tested at once. (Note that the VDS itself does not operate any differently when parallel testing is occurring instead of serial testing. The initialization code determines the type of testing to be performed by initializing only one device at a time for serial testing, or all devices at once for parallel testing.)

3.5.2 Services Used By The Initialization Code

The $DS_GPHARD service is very important in the initialization code. This macro will pass the address of a p-table to the diagnostic program. The program will then use the device parameters stored in the p-table to determine how to reference the device. (P-tables are discussed in Section 3.2).

For level 3 (standalone mode) programs, initializing a unit involves executing the $DS_GPHARD macro to get a unit's p-table address, and then executing the $DS_CHANNEL macro to initialize the appropriate bus adapter. The $DS_SETMAP macro may also be used in the initialization code. (Both the $DS_CHANNEL and $DS_SETMAP macros may also be used within the actual tests.)

For level 2R (user mode) programs, unit initialization will consist of executing the $DS_GPHARD macro to obtain the unit's p-table address, followed by issuing the $ASSIGN system service. Device allocation (using the $ALLOCATE system service) is requested by the VDS if the p-table descriptor for the device indicates that the device must be allocated (see Section 3.2.2).

3.5.3 Logical Units

The initialization code must be written to handle an unspecified number of units, since the number of units will vary from system to system. At run time, the VDS determines the number of units that can be tested by using the list of SELECTed units (see the <u>VAX Diagnostic Supervisor User's Guide</u>) and comparing it with the list of device types testable by the diagnostic program (as contained in the Device Mnemonics List - see Section 3.3.4). One of the arguments to the $DS_GPHARD macro is the "logical unit number." If this value is greater than the actual number of testable units, the VDS will return from the $DS_GPHARD service routine with an error status. Thus the initialization code can contain a REPEAT-UNTIL loop that executes the $DS_GPHARD macro and increments the logical unit number until the macro's return status value indicates the error.

It is important to note that the "logical unit number" argument to the $DS_GPHARD macro does not refer to the actual unit number of a hardware configuration. For example, consider a program that tests disks. Suppose this program is run on a system that has two controllers, each possessing one drive. Each of these drives could be unit Ø on its respective controller. The logical unit number associated with the unit would depend on the order in which the drives were attached. Once the $DS_GPHARD service has been executed, the p-table for the logical unit number can be examined (specifically, field HP$B_DRIVE) to determine which unit has been associated with the logical unit number.

3.5.4 Program Passes And The Initialization Code

When $DS_GPHARD returns an error status, indicating the highest numbered logical unit has been tested, the initialization code must signal the VDS that one program pass has been completed. The $DS_ENDPASS macro is used for this purpose. This macro will call a VDS service that will update the count of passes executed and check to see if the number of passes requested by the user has been executed. If so, the program's summary routine (see Section 3.7) and cleanup code (see Section 3.6) will be executed, and the VDS command line interpreter will be called. Otherwise program control is returned to the diagnostic program's initialization code, which can reset the logical unit number to zero so that a new program pass can begin.

Two other macros useful in the initialization code are $DS_BPASSØ and $DS_BNPASSØ. These macros are used to cause program branching depending on whether or not the first program pass is being executed. It is often necessary to perform special initialization the first time the initialization code is executed. For example, the location containing the number of the next logical unit to be tested must be initialized the first time through the code. Another example of a function that should only be performed the first time the initialization code is executed is "volume verification" (see Section 5.6.2). These macros are discussed in Section 3.11, Conditional and Unconditional Branching.

3.5.5 Initialization Code Examples

The following are examples of program steps needed in initialization code.

Initialization Code for Serial Testing:

```
IF PASS Ø
THEN
        BEGIN
        ! Program initialization
        ALLOCATE BUFFERS
        LOGICAL_UNIT_NUMBER=Ø
        END
ELSE
        INCREMENT LOGICAL_UNIT_NUMBER
        IF ALL UNITS DONE
        THEN
                BEGIN
                ! End of pass
                CALL $DS_ENDPASS
                LOGICAL_UNIT_NUMBER=Ø
                END
! Per-pass code
CALL $DS_GPHARD
ASSIGN CHANNEL
CLEAR BUFFERS
CLEAR COUNTERS
        :
        :
```

Initialization Code for Parallel Testing:

```
IF PASS Ø
THEN
        BEGIN
        ! Program initialization
        ALLOCATE BUFFERS
        END
ELSE
        BEGIN
        ! End of pass
        CALL $DS_ENDPASS
        END
LOGICAL_UNIT_NUMBER=Ø
REPEAT
        $DS_GPHARD
        ASSIGN CHANNEL
        INCREMENT LOGICAL_UNIT_NUMBER
UNTIL ALL UNITS DONE
CLEAR BUFFERS
CLEAR COUNTERS
        :
        :
```

3.6 CLEANUP CODE

When all testing of a device has been completed, there must be a means for guaranteeing that the device is left in a known, initialized, static state. The "cleanup code" is provided for this purpose. This code resides in the diagnostic program, delimited by the macros $DS_BGNCLEAN and $DS_ENDCLEAN.

The cleanup code will be executed under the following circumstances.

- The last program pass has been completed.
- The diagnostic program executes the $DS_ABORT macro. This macro should be used when a catastrophic failure is detected by the program.
- The user issues the VDS's ABORT command.
- An exception condition occurs and is handled by the VDS last chance condition handler (see Section 3.14.5, Condition Handling).
- The program is aborted because a $DS_ASKxxxx macro was executed with no user present and no default response (see Chapter 4).

Cleanup code should perform the following functions.

- Disable all device and adapter interrupts.
- Deassign channels, if in user mode.
- Deallocate memory buffers.
- Cancel timers.

3.7 SUMMARY ROUTINE

The "summary routine" is an optional portion of the diagnostic program. If included, it is used to display on the user's terminal a summary of the program's execution history. Summary routines are most likely to be included in programs that perform many repetitive functions and/or have long execution times, since these program are likely to compile large error counts. The summary routine will be called by the VDS at the end of the last program pass (unless the user has inhibited the display with the IES flag; see the VAX Diagnostic Supervisor User's Guide). Additionally, the routine will be executed when the user issues the SUMMARY command (see the User's Guide).

When the SUMMARY command is issued, the VDS provides a generalized summary message whether or not the diagnostic program includes a summary routine. This message indicates the program name and the number of errors that were reported (Section 3.9 discusses error reporting). An example of the message is as follows:

```
Summary of EVRAD - LEVEL 2 DISK FUNCTIONAL TEST, Rev 1.1:
 1 program detected error (1 Hard, 0 Soft, 0 System, 0 Device).
 0 Supervisor detected errors.
```

If a summary routine is included in the diagnostic program, the message generated by that routine is displayed with the above message.

The summary routine is delimited by the $DS_BGNSUMMARY and $DS_ENDSUMMARY macros. All messages displayed with the summary routine must be printed by using the $DS_PRINTS macro.

Typically, the routine will contain code to display such runtime statistics as the total numbers of read transfers, write transfers, read errors, and write errors that have been detected on each unit being tested. Any other information relevant to the type of device being tested may also be displayed. A separate set of totals must be kept for each unit. It is useful to store these sets of totals in one large data area within the program, delimited by the $DS_BGNSTAT and $DS_ENDSTAT macros.

3.8 TESTS, SUBTESTS, AND SECTIONS

3.8.1 Tests

All diagnostic programs contain one or more (usually several) "tests." A test consists of code that examines a portion of the UUT. If the diagnostic program is a logic test (see Chapter 1), each test should be designed to check a subset of the UUT's logic. If the program is a function test (see Chapter 1), then each test will check a subset of the total functionality of the device. Specific design, content, and number of tests are the program designer's decision of what is appropriate for a particular device.

Each test must be free-standing. That is, proper execution of a test must not depend on the previous execution of any other test. Thus, any group of tests must be executable in all possible combinations and sequences.

If several tests require a common segment of code, this common seqment may be made into a global routine called by each test. Global routines should be placed in a separate area of the diagnostic program, outside the boundaries of any particular test.

Each test is delimited by the $DS_BGNTEST and $DS_ENDTEST macros.

Sometimes it may be desirable to execute the same test repeatedly, but using a different set of input arguments each time. This may be accomplished by grouping the various sets of input arguments together and delimiting them with the $DS_BGNDATA and $DS_ENDDATA macros. When this is done, the VDS will automatically execute the code within the test once for every set of arguments specified before going on to the next test. From the user's point of view, this repeated execution of the code within the test will appear to be one execution of the test.

3.8.2 Subtests

Tests should be composed of one or more of "subtests." A subtest is a small section of code that performs one function. Each subtest must be delimited by the $DS_BGNSUB and $DS_ENDSUB macros. The $DS_BGNSUB macro automatically assigns a number to each subtest. Subtests are numbered from 1 to N for each test, where N is the total number of subtests within the test. Subtests cannot be nested. It is not legal to branch from one subtest to another using GOTO-type instructions. Subtests may be either executed sequentially or called from a higher-level routine. Figure 3-5 illustrates legal and illegal program flow using subtests.

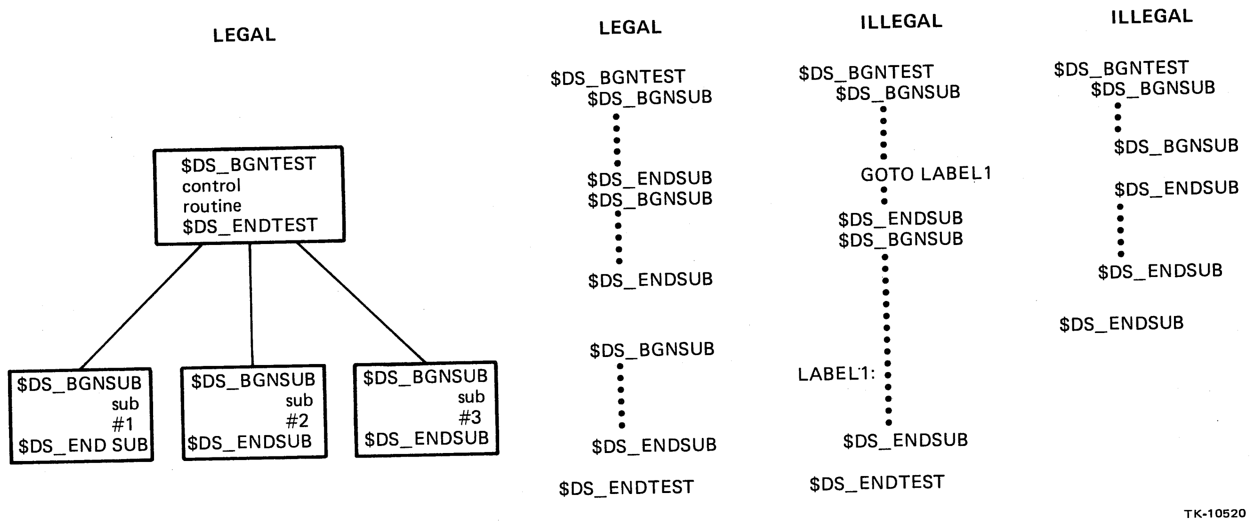

Figure 3-5 Legal and Illegal Usage of Subtests

If several tests require the use of the same subtest, the code within the subtest (NOT including the $DS_BGNSUB and $DS_ENDSUB macros) can be placed in a global subroutine placed in a separate area of the diagnostic program, outside any particular test. Then each subtest requiring the code can call the subroutine.

Subtests are useful for the following reasons:

- They define loop boundaries for the loop-on-error facility. Refer to Section 3.10, Looping, for a discussion of loop boundaries and looping on errors.

- They provide a means by which the program user can execute a small portion of a test. The user can use the VDS command language to cause the diagnostic program to be executed up to and including a particular subtest, with the option of looping on the subtest. Refer to the VAX Diagnostic Supervisor User's Guide.

3.8.3 Sections

A "section" is a group of tests. Sections are defined for the convenience of the program user. If the user specifies that a certain section of the program is to be executed, all the tests assigned to that section are automatically run. This frees the user of needing to specify a long string of test numbers manually.

The programmer should assign to a section groups of tests performing similar functions. The number, names, and purposes of a particular program's sections are the programmer's option, but the program should consider which groups of tests a user might wish to run as a set and create a section for that set. A test may belong to any number of sections.

Sections are defined by using the $DS_SECTION and $DS_SECDEF macros, and by including the section name(s) as arguments to the $DS_BGNTEST macro. These macros indicate to the VDS which tests should be associated with which sections.

Every program has a default section called DEFAULT. The contents of this section depend on the particular program application and are generally specified by the program's user community. However, no test within the default section can require any sort of manual intervention, such as altering switch positions, adding cables, and so on. The default section MAY ask for keyboard responses using the $DS_ASKxxxx macros (see Section 3.12.2.2, Prompting the User), but all $DS_ASKxxxx macros included in the default section MUST provide default responses. This will ensure that the default section will execute to completion if the VDS control flag OPERATOR is clear, indicating that no operator (user) is present.

If any tests in the diagnostic program require manual intervention, these tests must be grouped together in one section. This section should be called MANUAL. The manual section MUST test for the presence of an operator by using the $DS_BOPER or $DS_BNOPER macro (see Section 3.11, Conditional and Unconditional Branching). If an operator is not present, each test in this section must use the $DS_ABORT macro.

3.9 REPORTING ERRORS

The VDS provides extensive capabilities, via macro calls, for reporting detected error conditions. All error conditions MUST be reported by using the VDS macro calls. Error macros have the format $DS_ERRxxxx, as indicated later in this section.

3.9.1 Error Message Formats

The macros call VDS services that will cause error messages to be displayed on the user's terminal. Error messages are divided into three sections, or "levels." This is so users can use VDS control flags to select or inhibit the display of all or part of a message, as discussed in Section 3.9.2.

The first level is referred to as the "message header." Part of this header is generated automatically by the VDS and identifies the current test, subtest, unit, and error. The rest of the header consists of a message specified by the programmer as an argument to the $DS_ERRxxxx macro. This last part of the message is a short statement identifying the type of error.

The second level is provided by the programmer via the $DS_PRINTB macro. This level is used to provide a clear statement of what the error is. For example, if a particular register's contents are tested and found to be not as expected, this level would be used to display the expected and actual contents of the register.

The third level, also provided by the programmer (this time by using the $DS_PRINTX macro), can be a detailed error description, including such variable data as device register dumps and buffers of send vs. received data patterns. This level is used for dumping out large amounts of auxiliary information.

The $DS_PRINTB and $DS_PRINTX macros that are used to generate the second and third message levels are contained in a subroutine referred to as an "error reporting routine." When the address of an error reporting routine is passed to an error macro ($DS_ERRxxxx), the VDS will cause the routine to be executed after the message header (first level) has been displayed.

Details on specifying error messages are given in the description of the individual error macros ($DS_ERRxxxx) in Chapter 4.

Example 3-6 shows a typical error message. In this example, the first three lines comprise the message header. The second half of the third line was specified by the programmer; the rest of the header (plus the last line of the message) was generated by the VDS. The remaining portions of the message were generated by an error reporting routine. In this example, only the $DS_PRINTB macro would be used within the error reporting routine.

```
******* ECKAX - VAX 11/750-specific CPU Cluster Exerciser - 4.0 ********
 Pass 1, test 8, subtest 2, error 2, 4-MAR-1983 09:04:30.04
 Hard error while testing KA0: Attempting to initialize TU58 controller.

  Incorrect number of bytes received.

  EXPECTED: CONTINUE flag = 1
  Unrecognizable packet received.
  ACTUAL: 00000092(X) bytes beginning at 0000BA00

******** End of hard error number 2 *********
```

Example 3-6 Sample Error Message

Example 3-7 illustrates an error message in which both $DSPRINTB and $DS_PRINTX macros should be used. The first line following the three-line header should be displayed using $DS_PRINTB. The last part of the message displays the parameters of a $QIO service. Since this is a fairly long list of auxiliary information, it belongs to the third message level and hence should be displayed using $DS_PRINTX.

```
****** EVXBA - VAX Bus Interaction Program - 5.1 ******
 Pass 1, subtest 1, error 5, 9-MAY-83 14:55:29.16
 System fatal error while testing TTG1: ERROR ON QIO COMPLETION

ERROR ATTEMPTING TO WRITE TO TTG1:

QIO COMPLETION STATUS WAS: NOPRIV
_TTG1 QIO BLOCK PARAMETERS WERE:
QIO_EFN:     00000020(X)       ; EVENT FLAG #
QIO_CHAN:    00000050(X)       ; QIO CHANNEL #
QIO_FUNC:    0000000B(X)       ; IO$_WRITEPBLK FUNCTION
QIO_IOSB:    0004E888(X)       ; IOSB ADDRESS
QIO_ASTADR:          00001069(X)       ; ADDRESS OF AST
QIO_ASTPRM:          0004E800(X)       ; VALUE OF AST PARAMETER
QIO_P1:              00004C10(X)       ; P1 ARG VALUE
QIO_P2:              00000005(X)       ; P2 ARG VALUE
QIO_P3:              00000000(X)       ; P3 ARG VALUE
QIO_P4:              00000000(X)       ; P4 ARG VALUE
QIO_P5:              00000000(X)       ; P5 ARG VALUE
QIO_P6:              0004E940(X)       ; P6 ARG VALUE

****** End of device fatal error number 5 ******
```

Example 3-7 Sample Error Message

3.9.2 VDS Control Flags Associated With Error Reporting

Several VDS control flags are associated with error reporting. These flags are IE1, IE2, IE2, HALT, and LOOP. (See the VAX Diagnostic Supervisor User's Guide for a complete discussion of VDS control flags.)

The IE1, IE2, and IE3 flags control error message displays. If the user sets the IE3 flag, message level 3 is not displayed. If the IE2 flag is set, messages levels 2 and 3 are not displayed. Setting the IE3 flag will inhibit displaying of the entire error message.

If the user has set the VDS control flag HALT to activate halt-on-error, the VDS will stop execution of the diagnostic program after the error message has been printed. If the VDS control flag LOOP has been set, the VDS will begin executing a program loop after the error message has been executed (see Section 3.10, Looping).

3.9.3 Error Types

Error conditions are divided into five classes, depending on their severity. A macro is provided for each class. The five error classes are "preparation errors," "soft errors," "hard errors," "device-fatal errors," and "system-fatal errors."

3.9.3.1 Preparation Errors - Preparation errors are not hardware faults. They refer to the case in which the program user has not properly "prepared" the UUT for testing. For example, a particular diagnostic program may require that a disk drive be write-enabled by the user. If the program finds that the user has not write-enabled the drive, it can declare a preparation error. The program could then run only those tests that do not require writing to the UUT, or it could skip the unit altogether.

Preparation errors are declared by using the $DS_ERRPREP macro. This macro may be issued from any point within the diagnostic program except the cleanup code.

3.9.3.2 Soft Errors - A soft error is one that potentially can be recovered from. That is, it is an error which may go away if the operation that detected the error is repeated. In an operating system this type of error probably would not even be reported to the user, but in a diagnostic program it is important to flag all errors whether or not they can be recovered from so that the operation can be completed. An example of a soft error might be the occurrence of a write-check error when writing data to a medium. (It may be the medium that is bad, and not the device.) When a soft error is detected by the diagnostic program, the error should be reported and the operation reexecuted. However, there is generally a maximum number of retries that should be allowed. If the maximum is reached, a hard error (see below) should then be declared.

The macro to use when reporting a soft error is $DS_ERRSOFT. This macro can only be issued from within tests (see Section 3.8.1).

3.9.3.3 Hard Errors - A hard error is one that cannot be recovered from. That is, it is an error so serious that the operation being performed cannot be completed. Such an error might be a disk seek error. A hard error should also be declared if an operation detected a soft error and the operation was retried unsuccessfully several times. If, for example, a routine performing write operations on a disk detected several write-check errors (which are soft errors), then a hard error should be declared.

Hard errors are reported by using the $DS_ERRHARD macro. This macro can only be issued from within tests (see Section 3.8.1).

3.9.3.4 Device-Fatal Errors - Sometimes a diagnostic program detects so many hard errors on a UUT that it is pointless to continue testing the device. Perhaps there is something so seriously wrong with the device that it cannot be tested at all. Or maybe an attempt has been made to test a nonexistent unit. In any of these cases it is appropriate to declare a device-fatal error, which indicates to the user that the program intends to stop attempting to test the UUT in question. Whenever a device-fatal error is declared in a program performing serial testing, the program should leave the current test (by issuing the $DS_EXIT macro). Additionally, an internal flag could be set to indicate that a fatal error has been declared. Each test could check this flag and, if set, immediately issue the $DS_EXIT macro. That way no more testing would be performed on the unit (for this pass). The initialization code would reset the flag to allow testing of the next unit.

The macro for declaring device-fatal errors is $DS_ERRDEV. This macro may be issued from anywhere within a diagnostic program except the cleanup code.

3.9.3.5 System-Fatal Errors - A system-fatal error is one so serious that the diagnostic program cannot be executed at all. In user mode, for example, a system-fatal error should be declared if the user's process does not possess VMS privileges necessary to perform functions required by the diagnostic program (such as PHYSIO for a program that uses physical I/O -- refer to the VAX/VMS System Services Reference Manual.) Any time a system-fatal error is declared, the diagnostic program should subsequently execute the $DS_ABORT macro to abort program execution.

The macro for system-fatal errors is $DS_ERRSYS. This macro may be issued from anywhere within a diagnostic program except the cleanup code.

3.10 LOOPING

The VDS facility that is probably the most useful to repair technicians is program looping. Program loops, often called "scope loops," because they aid the technician in tracing signals with an oscilloscope, are enabled when the technician sets the VDS control flag LOOP (see the VAX Diagnostic Supervisor User's Guide). Once this flag has been set, a loop will begin executing any time an error macro ($DS_ERRxxxx) is issued.

3.10.1 Defining Loop Boundaries

Although actual execution of program loops is initiated automatically by the VDS, it is the responsibility of the programmer to define the boundaries of the loops.

Each loop will have a lower bound and an upper bound. Within these bounds will be at least one error macro. Whenever an error macro is serviced with the LOOP flag set, the VDS begins execution of the loop. Loop execution proceeds in the following sequence.

1. After servicing the error macro call, the VDS returns program control to the diagnostic program, to the point directly after the error call.

2. The diagnostic program continues execution until the loop's upper bound is reached.

3. From the upper bounf, the VDS causes program control to branch to the loop's lower bound.

4. Execution of the diagnostic program continues until the upper bound is again reached, whether or not the error macro is again issued.

5. The cycle is repeated.

Note that once the cycle is started, through the execution of an error macro, the macro may or may not be executed on subsequent passes through the loop. This means that the loop will continue to execute even if the error condition disappears. In fact, once a program loop has been initiated, it will continue to execute perpetually until a control-C is typed on the user's terminal.

Loop boundaries may be defined explicitly by the programmer. If they are not, then default values will be used. For a test that does not contain subtests, the default lower bound and upper bound for loops in that test are the $DS_BGNTEST and $DS_ENDTEST macros, respectively. For tests containing subtests, the default lower and upper bounds are, respectively, the $DS_BGNSUB and $DS_ENDSUB macros of the subtest containing the error macro that was executed to report the error condition.

The programmer can explicitly define loop boundaries by using the $DS_CKLOOP macro. This macro is placed after an error macro, but before the next $DS_ENDSUB or $DS_ENDTEST. If the the $DS_CKLOOP macro is contained within a test that consists of subtests, it must be placed within the bounds of a subtest. The macro takes as an argument the name of a program label. This label must be located before the error macro, but after the most recent $DS_BGNSUB or $DS_BGNTEST. The result is a loop whose lower bound is the label and whose upper bound is the $DS_CKLOOP macro itself.

Figure 3-6 illustrates the various loop boundaries.

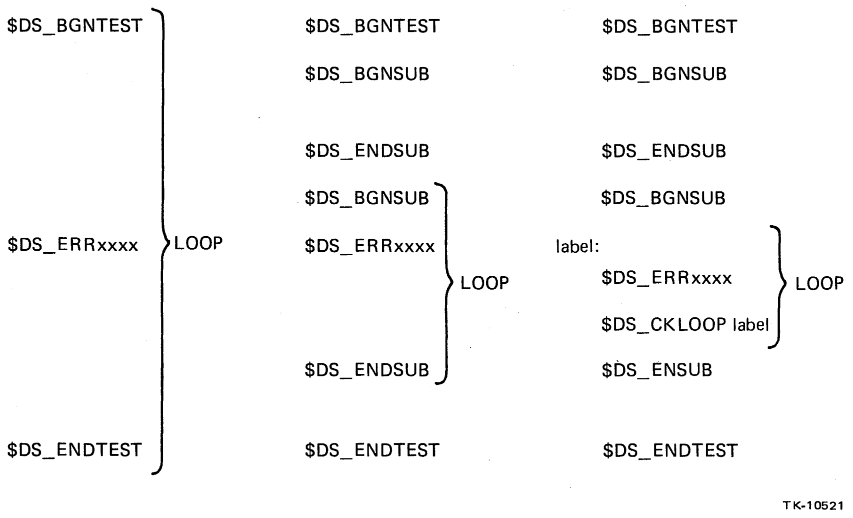

Figure 3-6 Examples of Loop Boundaries

3.10.2 Characteristics Of Loops

Loops should be small. Each loop should generate a minimum amount of electrical activity on the UUT. The less activity that is occurring, the easier it will be for the technician to trace relevant signals.

Loops must be made up of code that is repeatable. There is no point in creating a program loop unless the code within that loop can be executed repeatedly. The code must cause the same electrical activity to occur each time it is executed. For example, a loop that just sets a bit is useless, because the bit will be set the first time through the loop, and subsequent passes through the loop will cause no changes to take place. A loop that sets and then clears the bit would be appropriate.

In order to make a loop's code repeatable, it may occasionally be necessary to alter the program flow within the loop after the first pass through the loop. The $DS_INLOOP macro can be used to determine if a loop is being executed. Branching within the loop can be performed depending on the return status from this macro. This macro is useful in places where severe errors occur. Ordinarily the programmer may want to abort the program (using the $DS_ABORT macro) in such a case. However, if a loop is present, it may be desirable to branch around the $DS_ABORT macro to allow the loop to continue.

Caution should be practiced when branching within subtests containing $DS_CKLOOP macros. It is important not to branch past the $DS_CKLOOP macro, or the loop could be broken. For example, suppose a loop is being executed, with a $DS_CKLOOP macro as the loop's upper bound. Suppose now that a section of code within the loop tests for a hard error condition and then branches around a $DS_ERRHARD macro if the error does not exist. If the branch goes past the $DS_CKLOOP macro, the loop will be broken. Illustrations of proper and improper branching within loops are shown in Figure 3-7.

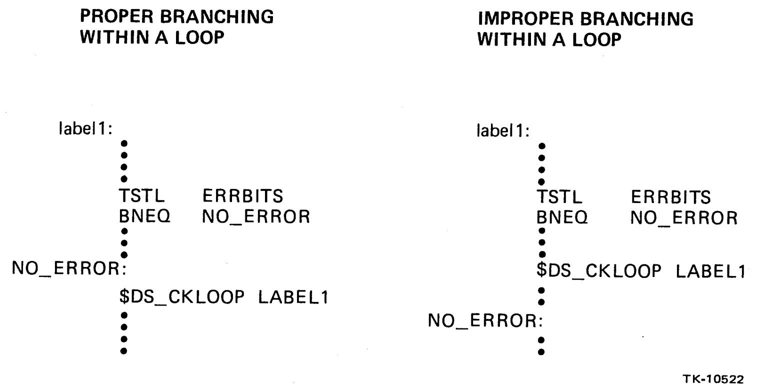

Figure 3-7 Proper and Improper Branching Within Loops

3.10.3 Nesting Loops

Loops whose boundaries are defined with the $DS_CKLOOP macro may be nested. Figure 3-17 illustrates nesting of loops. In Example A of Figure 3-8, loop 2 and loop 3 are contained in loop 1. In Example B, loop 3 is contained within loop 2, and loop 2 is contained within loop 1.

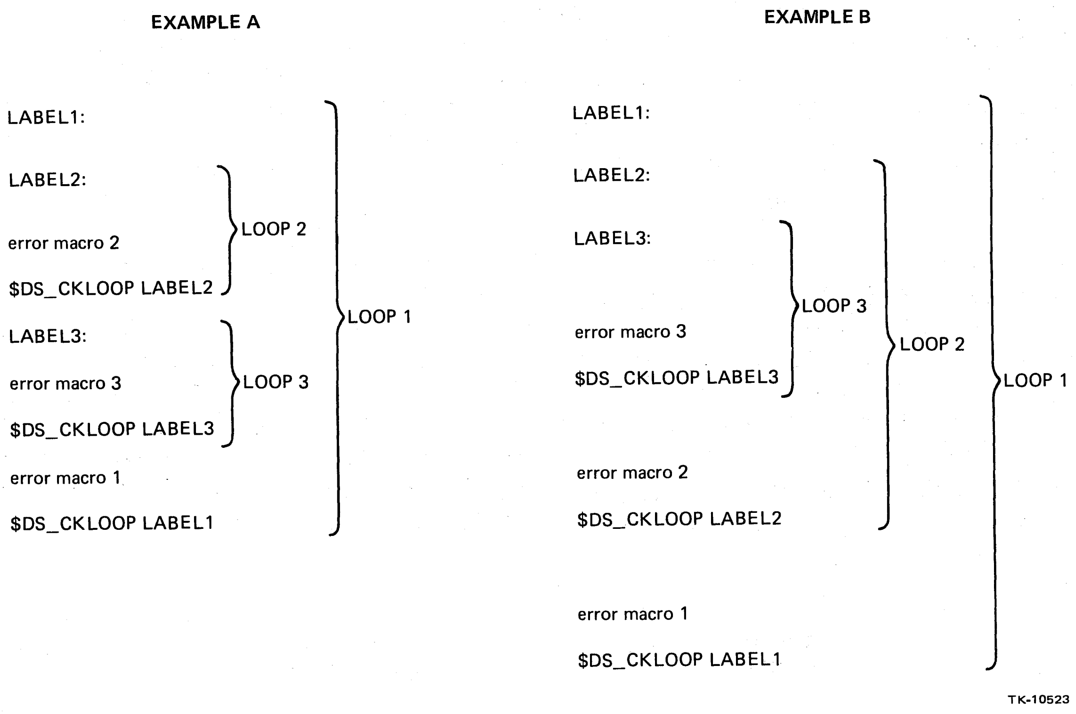

Figure 3-8 Nesting Loops

When loops are nested, the VDS always executes the smallest loop containing the issued error macro. If error macro 2 was issued in Example B, loop 2 would be executed.

The VDS will always execute the loop containing the most recently issued error macro. In Example A, suppose error macro 1 was issued. This would cause loop 1 to begin executing. Suppose at a later point in time that error macro 2 was executed for the first time (perhaps because of an intermittent hardware failure). Then loop 2 would begin execution and loop 1 would be forgotten.

3.10.4 User-Specified Looping

There is a method by which the user can request a loop to be executed even though an error macro has not been issued. The /SUBTEST qualifier on the RUN and START commands (see the VAX Diagnostic Supervisor User's Guide) can be used to specify a subtest on which the user wishes looping to occur. When the specified subtest is reached, looping begins on that subtest. The programmer should keep this feature in mind as subtests are designed.

3.11 CONDITIONAL AND UNCONDITIONAL BRANCHING

The VDS provides several macros to facilitate conditional branching within the diagnostic program.

$DS_BERROR, $DS_BNERROR

The "branch if error" and "branch if no error" macros can be used immediately after macros that call system services. The services will return a status indication (in RØ), and these macros cue on that status. The macros accept as an argument the address to which the program should branch.

$DS_BCOMPLETE, $DS_BNCOMPLETE

The "branch if complete" and "branch if incomplete" macros are also used immediately following macros that call system services. Their function is the inverse of that of the $DS_BERROR and $DS_BNERROR macros. That is, $DS_BCOMPLETE is equivalent to $DS_BNERROR and $DS_BNCOMPLETE is the same as $DS_BERROR. Choosing one set of macros over the other is simply a matter of "readability" in the source code. For some system services it makes more sense to branch if the service "completed successfully," while for others it is more appropriate to branch if there was "no error."

$DS_BOPER, $DS_BNOPER

The "branch if operator present" and "branch if operator not present" macros can be used anywhere in the diagnostic program. They cue on the setting of the OPERATOR flag (see the VAX Diagnostic Supervisor User's Guide). They make it possible to execute or skip certain segments of code, depending on whether a user is or is not present.

$DS_BQUICK, $DS_BNQUICK

The "branch if QUICK flag set" and "branch if QUICK flag not set" macros can be used anywhere in the diagnostic program. They cue on the setting of the QUICK flag (see the User's Guide). These macros allow you to create a "quick mode" in your program. This mode is selected optionally if the user sets the QUICK flag.

Quick mode provides a fast program pass that does not perform thorough testing and is used when the user is more interested in a fast run time than in careful, complete fault detection. The macros can be used to skip around segments of code in quick mode. Determination of what segments of code should be included or excluded in quick mode depends on specific program requirements.

$DS_BPASSØ, $DS_BNPASSØ

The "branch if pass Ø" and "branch if not pass Ø" macros can be used when it is necessary to cause program flow to change depending on whether or not the current program pass is the first one. The macros call a system service that returns a status indication (in RØ) of whether or not the current pass is the first one, then an appropriate branch is generated. These macros are only to be used in the program's initialization code.

$DS_ESCAPE

The $DS_ESCAPE macro is used to exit from a test or subtest if an error has been detected within that test or subtest. It is used when it is pointless to execute the rest of the code within the test or subtest after the error was detected. For example, there is no point in executing write tests on a disk if it has been discovered that the disk is write-protected and a user is not present.

If an error reporting macro ($DS_ERRxxxx) has been issued from within the current subtest or test, then issuing an $DS_ESCAPE macro will cause program control to pass to the end of the subtest or test.

$DS_EXIT

The $DS_EXIT macro provides for unconditional branching to the end of a test, a subtest, an interrupt service routine, or the summary routine. This macro is often used in conjunction with the conditional branching macros, as in the following example:

```
        $DS_BGNTEST
            :
            :
        $DS_BOPER 1Ø$
        $DS_EXIT TEST
1Ø$:        :
            :
            :
        $DS_ENDTEST
```

3.12 INPUT/OUTPUT

3.12.1 I/O With The Unit Under Test

3.12.1.1 I/O in User Mode - In user mode (level 2R programs), all input/output operations must be accomplished by using the VMS $QIO system service. The details of performing I/O operations with the $QIO service are described in the VAX/VMS I/O User's Guide, which MUST be read before development of a level 2R program is begun.

Initiating I/O activity in user mode is a process involving three steps, each of which requires use of a VMS system service.

- Assigning a channel to the device.

  A device cannot be referenced unless a channel linking the device to the program has been "assigned" to the user. A "channel" is a data path linking the device to the diagnostic program.

  Channel assignments are accomplished by using the $ASSIGN system service. This service request should be issued from the diagnostic program's initialization code.

  When the diagnostic program has finished using the device, its channel should be deassigned by using the $DASSGN system service. This service should be requested in the program's cleanup code.

- Allocating the device.

  If the diagnostic program will need exclusive use of the device to be tested (no other users allowed to reference the device while it is being tested), then the device must be "allocated" to the diagnostic program. Allocation is necessary if the program requires that a scratch medium be placed in the UUT. If the program can use the currently loaded (nonscratch) device medium in a nondesructive manner, device allocation is not necessary.

Device allocation is not performed directly by the diagnostic program. Instead, the allocation request is issued by the VDS (via the $ALLOCATE system service) when the user types the VDS SELECT command (see the _VAX Diagnostic Supervisor User's Guide_). The VDS determines whether or not to allocate the device by checking the HP$M_ALLOC bit in the device's p-table (see Section 3.2.1, P-Table Format). If this bit is set (by the program developer who created the p-table descriptor; see Section 3.2.2, P-Table Descriptors), then the $ALLOCATE service is requested. If the device cannot be allocated because it has already been allocated to someone else, the VDS informs the user.

An allocated device will be deallocated (by the VDS issuing a $DEALLOCATE service request) when the device is DESELECTed or when the VDS EXIT command is typed.

An instance when the diagnostic program might have to specifically allocate and deallocate a device is in the case of error logging. (We are not referring to VMS system error logging.) If a level 2R program writes error logging data to a device, it MAY be necessary to allocate the device. In this case the diagnostic program should use the $ALLOCATE service of VMS within the initialization code. The cleanup code will have to use the $DEALLOCATE service to deallocate the device. Refer to the _VAX/VMS System Services Reference Manual_.

- Queueing I/O requests.

  Actual input/output operations are requested by using the $QIO and $QIOW system services, which will place the request in an I/O queue. These services require that a set of parameters be passed to the service routine. These parameters specify the following types of information.

  - The channel number over which the data transfer is to take place. The channel number is obtained from the $ASSIGN service.

  - The type of transaction desired. This is indicated by a "function code" and is discussed below under "I/O Function Encoding."

  - The method by which the program is to be notified that the transaction has been completed. Three methods are available and are discussed below under "Synchronizing I/O Completion."

- The address of a buffer to receive diagnostic information. This buffer is discussed under "The $QIO Diagnostic Buffer."

1. I/O Function Encoding

I/O functions fall into three groups, corresponding to the three I/O methods (physical, logical, and virtual). The type of function to be used will depend on the type of device being tested and the type of diagnostic program being written (refer to Chapter 2).

The function that is to be performed by a $QIO service is indicated by passing to the service routine a 16-bit value having the format illustrated in Figure 3-9.

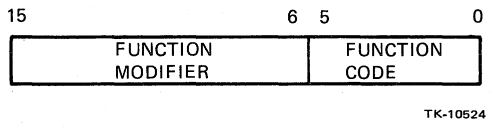

Figure 3-9 $QIO Function Code and Modifier Fields

The "function code" is a six-bit field indicating the type of I/O operation to be performed. Some function codes are device-independent, and others are device-dependent. Table 3-1 contains device-independent function codes for read and write functions in the three I/O transfer modes.

Table 3-1 Device-Independent Read and Write Functions

Physical I/O	Logical I/O	Virtual I/O
IO$_READPBLK	IO$_READLBLK	IO$_READVBLK
IO$_WRITEPBLK	IO$_WRITELBLK	IO$_WRITEVBLK

Refer to the VAX/VMS I/O User's Guide for discussions of the function codes available to individual devices.

The "function modifier" field is used to modify the operation specified by the function code. Bits within this field can be set in conjunction with the function code, and the $QIO service will alter the function to be performed accordingly. For example, the IO$_INHRETRY modifier can be used with an IO$_READVLBK function to inhibit retries when read errors are encountered.

Refer to the VAX/VMS I/O User's Guide for a more detailed dicussion of I/O function encoding, along with tables of all function codes and modifiers that are valid for each device supported by VMS.

2. Synchronizing I/O Completion

Three methods exist by which the diagnostic program can determine that an I/O request has been completed. The desired method of determination is indicated with the $QIO service call. The three methods available are

a. Waiting for an event flag.

It is possible to specify, as an argument to the $QIO or $QIOW macros, the number of an event flag (see Section 3.14.2) that system service is to set when I/O has completed. The diagnostic program can (by using a system service) wait for the specified flag to be set. (The $QIOW service is a combination of the $QIO and $WAITFR services.)

b. Testing an I/O status block.

The address of an "I/O status block" can be specified as an argument to the $QIO macro. When this is done, the $QIO service will cause the first word of this block to be loaded with a status code when the I/O operation has been completed. The program can test the contents of the block to determine the status of the I/O operation. The format of an I/O status block is shown in Figure 3-1Ø.

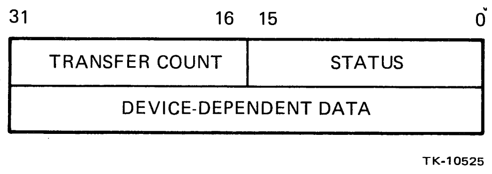

Figure 3-1Ø I/O Status Block Format

Refer to the VAX/VMS I/O User's Guide for more details about the contents of the I/O status block.

c. Execution of an AST routine.

It is possible to specify, as a $QIO argument, the address of an AST routine. (ASTs -- asynchronous system traps -- are discussed in Section 3.14.3.) If this is done, an AST will be delivered (and the AST routine called) when the I/O operation has been completed. This method of determining I/O completion provides for the most asynchronous (and most efficient, with regards to processor usage) I/O activity.

3. The $QIO Diagnostic Buffer

When a $QIO or $QIOW macro is issued, it is possible to request the system service routine to load a buffer with the contents of the device's registers. This "diagnostic buffer" will be loaded if two conditions are met:

a. The I/O transfer method is physical (see Chapter 2).

b. The process possesses the "diagnostic" VMS privilege (see the VAX/VMS Command Language User's Guide).

To request the system service to load the buffer, the programmer must:

a. Define a buffer area within the diagnostic program. This buffer must be large enough to contain the contents of all the device's registers.

b. Specify the address of this buffer as the "P6" argument to the $QIO or $QIOW macro (see Chapter 4).

When the I/O operation has completed, the buffer will contain the final contents of the device registers, plus additional information. Generally (but not always), the format of the buffer's contents will be as indicated in Figure 3-11.

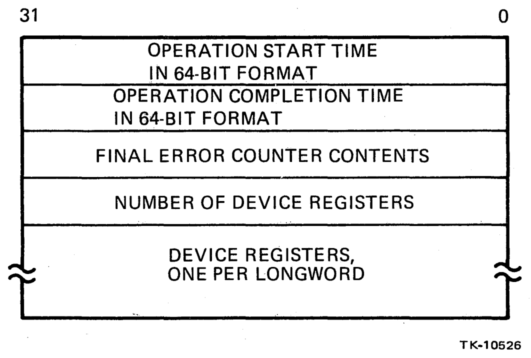
31
0
OPERATION START TIME IN 64-BIT FORMAT
OPERATION COMPLETION TIME IN 64-BIT FORMAT
FINAL ERROR COUNTER CONTENTS
NUMBER OF DEVICE REGISTERS
DEVICE REGISTERS, ONE PER LONGWORD
TK-10526

Figure 3-11 Typical $QIO Diagnostic Buffer Format

Two other VMS system services are useful to diagnostic programmers. The $GETCHN service will provide information about the device attached to a specific channel. This information consists of the "primary" and "secondary" device characteristics as described in the VAX/VMS I/O User's Guide. The $CANCEL system service will cancel all pending I/O requests on a specified channel, including those already in progress.

3.12.1.2 I/O in Standalone Mode - In standalone mode (level 3 programs), I/O is performed by direct reference of the device's registers. Thus routines to set up a device's control registers, service its interrupts, and check for error conditions must be contained within the diagnostic program.

The diagnostic program must set up the bus adapters so that a data channel can be created to transfer information across the buses. Because of the differences inherent in the bus adapters of the various VAX processor types, the VDS provides facilities for channel initialization that remove from the diagnostic programmer the burden of dealing with processor-specific details. This allows diagnostic programs to be automatically compatible with all VAX processor types.

The $DS_CHANNEL and $DS_SETMAP services of the VDS are used to create data channels in standalone mode. The $DS_CHANNEL service is used to initialize the MASSBUS and UNIBUS adapters. Depending on the parameters included with the $DS_CHANNEL macro, the service will

- Initialize the adapter
- Clear the adapter
- Enable or disable interrupts
- Provide current adapter status

Details are provided in the description of the $DS_CHANNEL macro in Chapter 4.

The $DS_SETMAP service will set up the adapter's mapping registers so that data transfers will reference the desired areas of main memory. Details are provided in the description of the $DS_SETMAP macro in Chapter 4.

The $DS_SHOCHAN service provides automatic display on the user's terminal of a bus adapter's internal registers. The configuration register and the status register are always displayed. If error conditions exist, additional registers will also be displayed. This macro should be used whenever the $DS_CHANNEL system service detects an error condition.

Interrupt service routines in a diagnostic program should be delimited by the $DS_BGNSERV and $DS_ENDSERV macros. The address of the interrupt service routine is passed to the $DS_CHANNEL service. The VDS has an interrupt preprocessor that initially receives control when an interrupt occurs, and then dispatches control to the specified interrupt service routine.

An interrupt service routine's function should be minimal, such as disabling further interrupts, making sure that the interrupt was expected (arrived through the proper vector), and saving device status. Error reporting should NOT be carried out in an interrupt service routine, with one exception; interrupt service routines should report unexpected interrupts.

Typical program flow when using an interrupt service routine is as follows.

Main-Line Code:

```
   Clear and initialize channel.
   Set up I/O transfer.
   Start watchdog timer.
   Enable interrupts.
   Clear done flag.
   REPEAT
       Test done flag.
   UNTIL done flag set OR watchdog timer finishes.
   IF done flag set
   THEN cancel watchdog timer; report I/O status
   ELSE report timeout error.
```

Interrupt Service Routine:

```
   Disable interrupts.
   IF unexpected interrupt (wrong vector)
   THEN set error status
   ELSE save device status.
   Set done flag.
   Return.
```

More information on interrupts can be found in the description of the $DS_CHANNEL service in the next chapter.

Other macros useful when performing I/O functions in standalone mode are:

$DS_SETVEC - Sets the contents of a specified interrupt or exception vector to a specified address. This macro is the ONLY method by which the vectors may be loaded (do not reference the SCB directly).

$DS_CLRVEC - Removes from a specified vector whatever contents had been placed in it by a $DS_SETVEC macro, and replaces it with the address of the VDS condition handler. This macro is the ONLY method by which the vectors may be reset (do not clear the SCB directly).

$DS_INITSCB - Reinitializes the system control block (SCB), which contains all of the interrupt and exception vectors, to their standard (VDS-defined) values. Useful if several $DS_SETVEC macros have been used.

$DS_PROBE - Attempts to access an address to determine whether or not hardware (either memory or an I/O device) is connected to it.

$DS_SETIPL - Sets the processor's interrupt priority level (IPL) to a specified value.

3.12.2 I/O With The User Terminal

All I/O between a diagnostic program and the user's terminal must be accomplished via VDS macros. Macros are provided for:

- Displaying messages consisting of simple ASCII strings or a combination of ASCII strings and variable data

- Prompting the user for a response, and then receiving and storing the response

- o Displaying the contents of a register and assigning a mnemonic to each bit

- o Determining the user's terminal type and characteristics

3.12.2.1 Displaying Message Strings

3.12.2.1 Displaying Message Strings - Message strings consisting of a combination of ASCII strings and data variables are displayed by means of the "print" macros. This set of macros has the general form $DS_PRINTx. Specifically, there are four print macros, known as $DS_PRINTB, $DS_PRINTX, $DS_PRINTF, and $DS_PRINTS. The $DS_PRINTB and $DS_PRINTX macros are only used to print error messages, and are used in conjunction with the error macros ($DS_ERRxxxx). The VDS control flags used to inhibit error messages (see the VAX Diagnostic Supervisor User's Guide) are keyed to the $DS_PRINTB and $DS_PRINTX macros. The $DS_PRINTF macro is used when it is necessary to provide the user with information unrelated to error reports. The $DS_PRINTS macro is used for summaries (see Section 3.7, Summary Routines).

The print macros are used to print simple ASCII strings, such as

```
DEVICE IS WRITE LOCKED.
```

They can also be used to display the contents of data words or to print a combination of ASCII strings and variable data, such as

```
EXPECTED:        1010101010101010 (B)
RECEIVED:        1011101010101010 (B)
XOR:             0001000000000000 (B)
```

Using a print macro involves specifying the address of a "format statement" and a list of variables. Format statements indicate the format in which the variables are to be printed. The method used by the print macros to format messages is the same as the $FAO system service provided by VMS. In fact, the $FAO service is also provided by the VDS. This service will format, but not print, a message. The print macros will both format and print the desired message. It is also possible (and occasionally desirable) to format a message with the $FAO service and then display it by using one of the print macros.

Another macro useful for displaying information to the user is $DS_CVTREG. With this macro, you specify the address of a register and the address of a string of mnemonics. The mnemonics are the names assigned to the bits within the register. The macro will read the register and produce a character string telling which bits of the register are set. This string can then be displayed using one of the print macros.

Details on the print macros are in Chapter 4. The $FAO service is discussed in Chapter 4 and in the VAX/VMS System Services Reference Manual.

It is sometimes useful to know the type and characteristics of the user terminal. For instance, you may want to format text displays differently on a video terminal than on a hardcopy terminal. The $DS_GETTERM service may be used to determine the user terminal's type and characteristics.

3.12.2.2 Prompting the User - There are occasionally instances in which it is necessary to solicit information from the user. A common example is the case in which the program must, at a certain point in its execution, ask the user to perform an action such as connecting a cable and to then type a response indicating that the action has taken place. Also, there may be circumstances under which it is necessary to obtain information about the UUT other than what is contained in the p-tables. (It is important, however, to TRY to place all device-specific information in the p-tables so that it can be specified in ATTACH commands BEFORE the diagnostic program is started.)

All solicitation of user responses during the diagnostic program's execution must be made through the use of the $DS_ASKxxxx macros. These macros allow the programmer to specify a prompting message, the format in which the user's response is to be interpreted, and a storage area into which the response should be placed.

Specifically, there are five $DS_ASKxxxx macros.

1. $DS_ASKADR - Prompt the user for an address within a specified range and store the result.

2. $DS_ASKDATA - Prompt the user for a numeric value within a specified range and store the result.

3. $DS_ASKVLD - Same as $DS_ASKDATA, except allows programmer to specify storage location of result as a field (using position and size) within a large bit structure.

4. $DS_ASKLGCL - Prompt the user for a "yes" or "no" response, and store the result as one bit, set or cleared.

5. $DS_ASKSTR - Prompt the user for a character string and store the result.

The macros also allow the programmer to specify a default value for the response. If there is no user present (indicated by the state of the VDS control flag OPERATOR, see the VAX Diagnostic Supervisor User's Guide), the default value will automatically be used. If no default value exists, the program will be aborted.

Sometimes diagnostic programs require the user to specify run-time options other than those that can be selected using the VDS command language. In such cases, the $DS_ASKxxxx macros can be used to prompt the user to indicate the required run-time parameters. One method of accomplishing this is to ask a set of questions that can be answered with "yes" or "no," such as

```
DO YOU WISH TO RUN OPTION X?
DO YOU WANT THE DEVICE TO RUN IN MODE Y?
```

The responses to these question could be converted to set or cleared bits that the diagnostic program could test. This method is fine if the total number of program options is small.

However, for a program with a large number of run-time options, the program user might have to answer a large list of questions every time the program is executed (assuming he or she did not want to use the default values for these questions). In such cases, the programmer might want to just prompt the user once and allow him or her to type a string of options, as

```
OPTIONS ARE OPTION_X, OPTION_Y, OPTION_Z
(DEFAULT IS OPTION_X)
TYPE OPTIONS:
```

Allowing the user to type a list of the options wanted is more convenient for the user, even though it is more difficult for the programmer to check the strings typed to see if they are valid.

For a program having a very large set of run-time options it might be beneficial for the programmer to create a command language. An example might be

```
Commands:
    OPTIONS - select options
    MODES   - select device modes
    BEGIN   - begin program execution
    RESUME  - continue after control-C
```

The user would type the VDS RUN or START command to start the diagnostic program's execution. In the program's initialization code or perhaps within a particular test, the program executes $DS_ASKxxxx macros to prompt the user for command strings. The program parses and executes each command. The BEGIN command (or something similar) simply allows the VDS to continue normal dispatching of the diagnostic program. The RESUME command would be useful if a control-C handler is defined within the diagnostic program (see Section 3.14.6, Handling Control-Cs). The number and types of commands defined would, of course, depend completely on the particular diagnostic program being designed.

The VDS provides two macros to facilitate command parsing. The $DS_CLI macro is used to define the desired command language. The $DS_PARSE macro compares an input stream (obtained from the user via a $DS_ASKxxxx macro) against the command language defined with a set of $DS_CLI macros and will either dispatch to the proper action routines or inform the user if an illegal command has been typed.

3.12.2.3 Displaying HELP Text - Chapter 5 discusses the creation of HELP files, which are supplemental files containing informational text that the user can read. It may sometimes be desirable for the diagnostic program to fetch and display sections of the HELP file. This can be accomplished by using the $DS_HELP macro. Read the section of Chapter 5 on HELP files, and then refer to Chapter 4 for a description of the $DS_HELP macro.

3.13 MEMORY MANAGEMENT AND ALLOCATION

Note: For a discussion of VAX memory management, see the VAX Architecture Handbook.

The memory management hardware may not be directly referenced by diagnostic programs running under the VDS.

3.13.1 Memory Management In User Mode

In user mode (level 2R programs), memory management hardware is under the control of VMS and it is always turned on. All of the VMS memory management system services are available for use by diagnostic programs. See the VAX/VMS System Services Reference Manual for the uses and restrictions applying to VMS memory management services. Allocation of new memory space should only be accomplished with the VDS $DS_GETBUF macro, as described in Section 3.13.3.

3.13.2 Memory Management In Standalone Mode

In standalone mode (level 3 programs), the memory management hardware may be turned on or off. Normally, it is off. Diagnostic programs can turn on memory management with the $DS_MMON macro. Once on, it can be turned back off with the $DS_MMOFF macro. All map register initialization is performed by the VDS, outside the control of the diagnostic program. Turning on memory management will not increase the diagnostic program's virtual address space, since the VDS loads the mapping registers so that there is a direct one to one correspondence between virtual and physical addresses in PØ memory space.

When memory management is enabled, the VDS sets the protection of all pages to"user write." It is possible to change the protection of any page or group of pages by using the $SETPRT macro.

In standalone mode, the memory management hardware can be turned on and off by the user, via the SET MM ON and SET MM OFF commands. These commands override the $DS_MMON and $DS_MMOFF macros contained within a dignostic program. Thus if a user has issued the SET MM ON command, the diagnostic program cannot shut off memory management with the $DS_MMOFF macro.

Some diagnostic programs may not be able to execute if the memory management hardware is enabled. If this is the case, the $DS_MMOFF macro must be issued at the beginning of the program. If the status return from this macro indicates that the operator has turned on memory management then the program must abort itself (with the $DS_ABORT macro), printing an appropriate error message before doing so.

3.13.3 Memory Allocation

Many diagnostic programs need extra memory space for I/O buffers or other uses. Additional memory space may be acquired by using the $DS_GETBUF macro. Both user mode and standalone mode programs should use this macro, since this method is the only way of assuring that there will be no memory allocation conflicts between the VDS and the diagnostic program.

The VDS keeps track of all free memory. The $DS_GETBUF macro is used to request the VDS to assign a certain number of pages to the diagnostic program. The VDS will return the starting address of the memory space it has assigned. (Space will be assigned as a group of contiguous physical pages in standalone mode, and as a group of contiguous virtual pages in user mode.) When a diagnostic program is executing on a system possessing 512K bytes of physical memory (the smallest memory size supported by the VDS), the program is guaranteed access to at least 96K bytes of buffer space.

Memory space is returned to the VDS's free memory pool by using the $DS_RELBUF macro.

3.14 SYNCHRONOUS AND ASYNCHRONOUS EVENTS

3.14.1 Introduction

Synchronous events are those that occur within the normal execution flow of a program. Waiting for a bit to become set or creating a time delay are both examples of synchronous events. Asynchronous events are those that happen outside the normal execution flow. VAX exceptions are asynchronous, because they cause the normal flow of a program to be changed (program control is passed to the condition handler). Refer to the VAX Architecture Handbook for a detailed discussion of VAX exceptions.

Diagnostic programs often need to handle occurrences of synchronous and asynchronous events. "Event flags" are useful for synchronous processing of events. AST routines and condition handlers are used for asynchronous processing. There are both synchronous and asynchronous methods available for handling time-critical situations.

3.14.2 Event Flags

Event flags are flags that can be used by diagnostic programs to indicate status information. Services are provided for setting, clearing, and reading the flags. Additional services allow the diagnostic program to wait for a flag or group of flags to be set before proceeding with program execution. The services are called via macros.

There are 64 event flags, numbered from Ø to 63. The flags are divided into two clusters, each containing 32 flags. Some event flag macros require that the cluster be indicated.

Event flag Ø is reserved for exclusive use by the VDS and is not available to diagnostic programs.

Flags 1 through 23 can be set or cleared by the user via the SET EVENT FLAGS and CLEAR EVENT FLAGS commands, which means they can be used to implement user selection of optional program features.

Flags 24 through 31 are used by VMS and hence cannot be used by level 2R diagnostic programs. They are available, however, to level 3 programs.

Flags 32 through 63 are available to all diagnostic programs. Users cannot modify these flags.

In user mode (level 2R programs), event flags are maintained by VMS. The event flag macros call service routines within VMS. Event flags Ø through 63 are referred to as "local event flags," since they can only be used internally by a single process. Another set of event flags, numbered from 64 through 127, are referred to as "common event flags" since they can be shared by cooperating processes. The VMS system service $ASCEFC must be used to associate common event flags with processes in order for these flags to be shared. See the VAX/VMS System Service Reference Manual for details.

In standalone mode (level 3), event flags are maintained by the VDS, and the event flag macros call service routines within the VDS.

The following macros are used in both level 2R and level 3 programs to reference event flags:

$SETEF - Sets specified event flags.

$CLREF - Clears specified event flags.

$READEF - Read the current status of specified event flags.

$WAITFR - Wait for a specified event flag to become set.

$WFLAND - Wait for a group of event flags to become set.

$WFLOR - Wait for one of a group of event flags to become set.

$QIOW - Queue an I/O request and wait for a specified event flag to become set (indicating I/O completion). Equivalent to $QIO followed by $WAITFR.

Additionally, the $SETIMR (see Section 3.14.4, Timing) and $QIO (see Section 3.12.1.1, I/O in User Mode) macros can optionally specify references to event flags.

3.14.3 Asynchronous System Traps (ASTs)

An asynchronous system trap (AST) is a method by which a routine can be entered asynchronously, outside the normal program flow, similar to a device interrupt. A routine that is entered via an AST is referred to as an AST routine. The process by which AST routines are dispatched is called AST delivery.

3.14.3.1 **AST Delivery** - Three macros, available to both level 2R and level 3 diagnostic programs, allow the use of ASTs. These macros are $SETIMR, $QIO, $QIOW, and $DS_CNTRLC. Each of these macros will accept as an argument the address of an AST routine. In the case of the $SETIMR macro, the AST routine will be entered when the specified amount of time has elapsed. For the $QIO and $QIOW macros, the AST routine will be executed when the requested I/O operation has completed. If the $DS_CNTRLC macro is used, it will cause an AST routine to be entered when the program user types a control-C.

ASTs may be enabled or disabled with the $SETAST macro. If ASTs are disabled, ASTs will not be delivered and thus the AST routines will not be executed.

If a diagnostic program is waiting for an event flag (see Section 3.14.2, Event Flags) or hibernating (see Section 3.14.4, Timing), ASTs will still be delivered to it. After the AST routine has been executed, the program will be returned to the state it was in before the AST was delivered (unless, of course, the AST routine itself set the desired flag or woke the program).

3.14.3.2 **AST Routines** - An AST routine is entered via a CALLG instruction. Thus the routine must have an entry mask and must return by using a RET instruction. It must save (by specifying them in the entry mask) any registers it uses, other than RØ or R1.

When an AST routine is entered, the AP points to an argument list in the format illustrated by Figure 3-12. The register values in the argument list are those saved when the main-line code was interrupted on delivery of the AST. The AST parameter is a value specified by the "AST parameter" argument of the $SETIMR, $QIO, or $QIOW macro used to request delivery of the AST. This argument can be used by the AST routine to determine who called it.

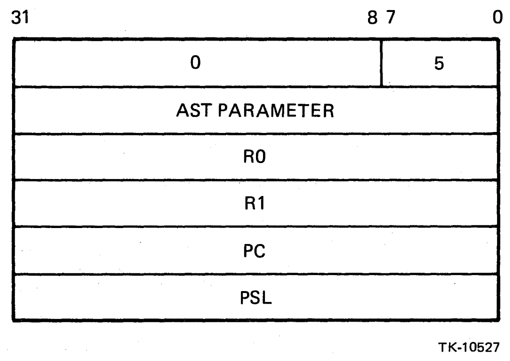

Figure 3-12 Argument List Passed to an AST Routine

3.14.4 Timing

Facilities are provided for creating timing delays within a diagnostic program. These facilities allow you to

- Specify a particular length of time you wish to wait before proceeding
- Cause the diagnostic program to stop executing until an expected event occurs
- Cause an asynchronous event to occur after a specified length of time has passed

The timing facilities provided by the VDS take into account speed differences among the various VAX process types. Therefore, all diagnostic programs containing time-dependent operations MUST use the VDS timing facilities when coding these operations, in order to guarantee program compatability with all current and future processor types.

The VDS timer services are accessed by macro calls. Some macros can be used for both level 2R (user mode) and level 3 (standalone) programs, while others may only be used for level 3 programs.

3.14.4.1 Timing Facilities Available in User Mode and Standalone Mode - The following is a list of macros that may be used by both level 2R and level 3 programs to control time-dependent functions.

$GETTIM - Gets the current system time.

$SETIMR - Allows you to cause an event to take place after a specified amount of time has passed.

$BINTIM - Converts an ASCII string that specifies a time into a numeric value meaningful to the $SETIMR macro.

$CANTIM - Cancels requests specified with the $SETIMR macro.

$HIBER - Causes processing to stop until an expected event occurs. Referred to as "hibernation."

$WAKE - Cancels a $HIBER request. Referred to as "waking" the program.

$DS_WAITMS - Delays sequential program execution for a specified number of milliseconds.

$DS_CANWAIT - Cancels time remaining from a $DS_WAITUS or $DS_WAITMS call

A typical use of these macros in standalone mode would be to issue a $SETIMR macro that will cause an AST routine (see Section 3.14.3) to be executed when the specified time has expired. Then a device's interrupts could be enabled. Some other processing could take place while waiting for the interrupt. When the interrupt occurs, the interrupt service routine could issue a $CANTIM macro to cancel the $SETIMR. If the interrupt does not occur before the time period ends, the AST routine would be entered. This routine could declare a timeout error. Program steps for this function would be as follows.

```
Main Program:              Interrupt Service           AST Routine:
                           Routine:


Issue $SETIMR macro.       Process interrupt.          Set error flag.
Enable interrupts.         Issue $DS_CANTIM macro.     Return.
Continue.                  Return from interrupt.
IF error flag set
THEN
  issue $DS_ERRxxxx macro
ELSE
  continue.
```

3.14.4.2 Timing Facilities Available in Standalone Mode Only -

The next macro may only be used by level 3 programs.

$DS_WAITUS - Delays sequential program execution for a specified number of microseconds.

A typical use of this service would be the enabling of a device's interrupts followed by a call to the $DS_WAITUS service to see if an interrupt occurred within a certain time frame. The interrupt service routine would set a flag to indicate that the interrupt occurred and would issue a $DS_CANWAIT to cancel any time remaining from the wait service. (Usually, the $DS_CANWAIT is optional and simply improves execution time by eliminating unnecessary time remaining in wait loops.) After the $DS_WAITUS call would be code to test the interrupt service flag. If the flag is set, the interrupt occurred. If not, the entire time delay was used up, indicating a time out condition. Program steps for this function would be as follows.

```
Main Program:                     Interrupt Service Routine:

Set up device for I/O.            Process interrupt.
Enable interrupts.                Set interrupt-occurred flag.
Issue $DS_WAITxx macro call.      Issue $DS_CANWAIT macro.
Test interrupt-occurred flag.     Return from interrupt.
If flag not set
THEN
   issue $DS_ERRxxxx macro
ELSE
   continue.
```

3.14.5 Condition Handling

Note: For additional information regarding condition handling, refer to the VAX Architecture Handbook and the VAX/VMS Software Handbook.

The VDS contains condition handling routines that will handle all exception conditions. It is therefore unnecessary under most circumstances for the diagnostic program to provide exception handling facilities. However, the VDS provides the ability for the diagnostic program to field exceptions when necessary.

The VDS searches for condition handlers in exactly the same manner as VMS. As detailed in VMS documentation, handlers are searched for in the following order:

1. If a primary handler exists, it is used.
2. If secondary handler exists, use it.
3. Search call frames for address of handler.
4. Use "last chance" handler.

If a handler is found it can

- Handle the condition and indicate a "success" (SS$_CONTINUE) return, or

- Not handle the condition and indicate a "resignal" (SS$_RESIGNAL) return, which causes the handler dispatcher to continue to search for a handler.

The VDS has a secondary condition handler, but it only services BPT and T-bit exceptions associated the the VDS's breakpoint and single-step facilities (see the <u>VAX Diagnostic Supervisor User's Guide</u>).

The main condition handling facility of the VDS is a "last chance" handler that is capable of dealing with all exception conditions (and will abort execution of the diagnostic program by causing the program's cleanup code to be executed).

In standalone mode, the VDS searches for a condition handler and if none is found, a call to the last chance handler is forced. This call to the last chance handler cannot be disabled by a diagnostic program.

Additionally, the address of the VDS last chance handler is placed on the highest call frame of the VDS. This means that in user mode, the VDS last chance handler will take precedence over the VMS last chance handler. It also means that, as in standalone mode, a diagnostic program cannot disable the VDS handler.

If a diagnostic program declares a handler in one of its call frames, that handler will take precedence over the VDS's last chance handler. In both user mode and standalone mode, a condition handler may be specified by loading the handler's address into the first address of the call frame (the address pointed to by the FP). In MACRO-32, this would be accomplished with the instruction

```
        MOVAB   CONDHNDLR,(FP)
```

To declare a condition handler in BLISS-32, refer to the <u>BLISS Language Guide</u>.

In user mode, diagnostic programs may also declare condition handlers by using the $SETEXP system service of VMS. Refer to the VAX/VMS System Services Reference Manual.

When a condition handler is given control, it is passed two arguments. The first argument is the address of a "signal array" and the second is the address of a "mechanism array." These arguments are passed in the manner indicated by Figure 3-13.

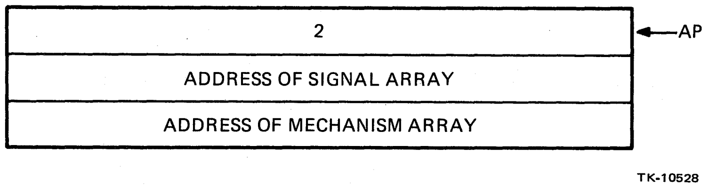

Figure 3-13 Argument List Passed to a Condition Handler

The signal array indicates the type of condition. Its format is shown in Figure 3-14. In the figure, N is the total number of longwords (excluding the one containing N) making up the array. Condition names are defined by the $SSDEF macro of LIB.MLB listed in the VAX/VMS System Services Reference Manual, or the $DS_DSDEF VDS macro. If the "condition name" parameter is DS$_UNEXPINT, then the next argument is the SCB vector offset.

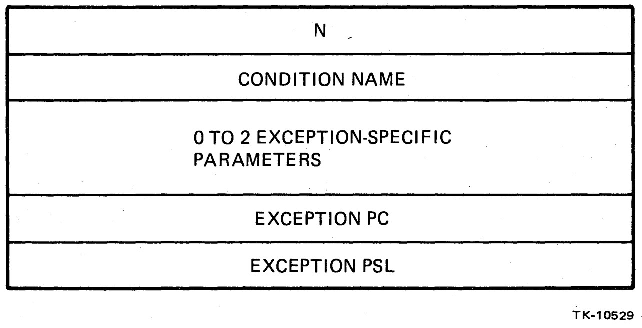

Figure 3-14 Format of Signal Array

The mechanism array is illustrated in Figure 3-15.

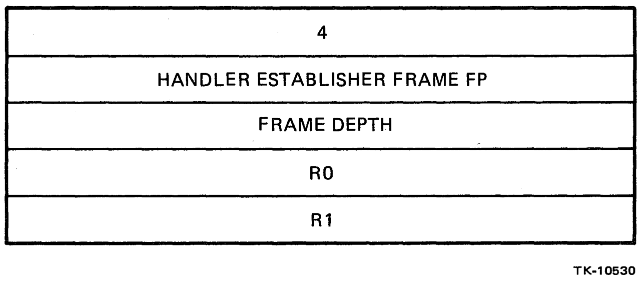

Figure 3-15 Format of Mechanism Array

A condition handler can either field the condition or return with a "resignal" status to indicate that another handler should be called. If the handler fields the condition, it must place the status code SS$_CONTINUE in RØ before returning. If the handler does not field the condition, the SS$_RESIGNAL status code must be placed in RØ. Condition handlers end with an RET MACRO-32 instruction. A condition handler may use the $UNWIND macro to "unwind" the call frame (alter program flow) if it cannot handle the condition. Unwinding is detailed in the discussion of the $UNWIND macro in Chapter 4.

The condition handler will receive control when ANY exception condition occurs. The handler must determine the type of exception (by examining the signal array) and decide whether or not to handle the particular condition. It is quite common to write a condition handler that will only process one or two types of exception conditions, and resignal all others so that another handler (such as the VDS last chance handler) can process them.

As an alternate method in standalone mode, the programmer may use the VDS macro $SETVEC to store the address of a condition handler in the system control block (SCB). This allows the diagnostic program to field specific exception conditions, instead of all of them. By using this method, the VDS handler dispatcher is bypassed and control passes directly to the handler pointed to by the exception vector. This handler MUST process the exception and cannot "resignal."

If the diagnostic program contains a condition handler, the $DS_PRINTSIG macro can by used to automatically format and print the contents of the signal array.

3.14.6 Handling Control-Cs

Normally when the user types control-C, program control passes to a VDS routine which aborts the current VDS function (such as executing a diagnostic program or building a p-table). It is possible to specify an alternate control-C handling mechanism within the diagnostic program by using the $DS_CNTRLC macro. The diagnostic program can use this macro to specify the address of a routine that is to be executed when a control-C is typed.

If the macro is used and a control-C is typed, the VDS will pass program control to the specified routine. This routine may perform any processing needed and then

a. Pass a return status code of zero (in RØ), which will cause the VDS to then execute its own control-C handler. This technique is useful in cases where it is desirable for the diagnostic program to perform some processing of its own whenever a control-C is typed, before the VDS takes over.

b. Pass a nonzero status code (in RØ), to indicate that the VDS should not execute its own control-C handler. In such a case, the VDS will continue performing the function it was performing before the control-C was typed.

c. Not return at all.

A possible use of options (b) and (c) would be the case where a special command language has been defined by the programmer (see Section 3.12.2.2, Prompting the User). In this case it might be desirable for the user to be brought back to the special command line interpreter when a control-C is typed. One of the special commands might have the same function as the VDS CONTINUE command (such as the RESUME used above), in which case option (b) would be used. If the RESUME command was not typed, the current function would be aborted and a new command would be fetched from the user, so option (c) would be selected.

The $DS_CNTRLC macro also allows the programmer to disable control-C servicing altogether. This makes it possible to ensure that certain portions of code will be executed without interruption, if necessary. Control-C servicing can be disabled temporarily while this uninterruptable code is executing, and then reenabled. If a control-C is typed while control-C servicing is disabled, the control-C is not lost. It will be serviced when the servicing is reenabled. It is important to note that CONTROL-C SERVICING MUST NOT BE DISABLED FOR LONGER THAN THREE SECONDS AT ONE TIME. Some run-time environments (APT in particular) cannot tolerate a longer control-C response delay, nor do users appreciate long periods of time when control-Cs are not serviced.

Because dispatching to the control-C handler is performed by the VDS, a control-C will not be acknowledged while the diagnostic program is executing. Whenever the diagnostic program calls a system service routine, the service routine will check to see if a control-C has been typed. But suppose that by some chance the program contains a large segment of code that does not call any system service routines for a long period of time. If a control-C is typed, it will not be acknowledged while this code is executing. In order to prevent this problem, any large section of code (or small section that loops for a long period of time) that does not call any system services must occasionally issue the $DS_BREAK macro. This macro will call a service that simply checks for a control-C and, if none has been received, just returns. A $DS_BREAK MACRO OR SOME OTHER SYSTEM SERVICE MUST BE ISSUED AT LEAST EVERY THREE SECONDS.

3.15 FILE MANAGEMENT

3.15.1 Introduction

It may occasionally be necessary for a diagnostic program to make reference to a separate, subsidiary file. In such a case, two facilities are available, namely:

- The $DS_LOAD system service
- Record management services (RMS)

The $DS_LOAD system service is useful for loading an entire file into a buffer area of memory.

If more involved manipulations of files is desired, such as referencing specific records or blocks, then the record management services should be used.

Level 2R (user mode) programs may use VAX-11 record management services (RMS) to manipulate files. The entire range of RMS services is available to the diagnostic program. Detailed information for VAX-11 RMS is available in the VAX-11 Record Management Services Reference Manual.

Level 3 (standalone mode) programs are provided with a subset of the VAX-11 RMS functionality. This functionality resides within the VDS. It emulates VAX-11 RMS and is referred to in this manual as VDS RMS. For those functions supported by VDS RMS, the program interface is exactly the same as that of VAX-11 RMS. That is, both level 2R and level 3 programs will use the same macros. In user mode the service calls are fielded by VMS, while in standalone mode they are handled by the VDS.

Table 3-2 lists the limitations of VDS RMS, as compared to VAX-11 RMS.

Table 3-2 Comparison of VAX-11 RMS and VDS RMS

VAX-11 RMS	VDS RMS
• Provides read and write access.	• Provides read access only.
• Supports sequential and relative files.	• Supports sequential files only.
• Supports sequential, random, and random-by-RFA file access.	• Supports sequential and random-by-RFA file access.
• Terminals can be accessed.	• Terminals cannot be accessed.
• Console device cannot be referenced.	• Console device can be referenced. (RT-11 format only.)
• FAB, RAB, XAB, and NAM control structures are defined.	• Only FAB, RAB, and FHC fields of XAB are defined.

Also, many of the option bits defined in the VAX-11 RMS control structures are not defined in VDS RMS.

When using RMS in a level 2R program, use the VAX-11 Record Management Services Reference Manual as a reference guide. When using RMS in a level 3 program, use this manual as the main reference guide. However, since this manual is not as detailed as the VAX-11 RMS reference manual, it may be necessary to refer to that manual also.

Whether the diagnostic program is level 3 or level 2R, the RMS macros are defined in LIB.MLB for MACRO-32 and LIB.L32 for BLISS-32. Note that these are VMS libraries and thus contain the VAX-11 RMS macro definitions. Therefore, inclusion of unsupported RMS functions in a level 3 program will not be detected until the program is actually executed.

In order for a diagnostic program to use RMS services on a file, the device on which the file resides must have been previously attached. (This is true for both level 2R and level 3 programs.) If the device is the one from which the VDS was loaded, the VDS will automatically issue a $DS_ATTACH macro for the device. If the device is not the VDS load device then the diagnostic program can issue an $DS_ATTACH macro, or the program can depend on the user to issue an ATTACH command.

3.15.2 VDS RMS Overview

VDS RMS provides facilities for easily gaining access to and reading sequential files on a disk or magnetic tape device, including the system's console device. The records within a file may be accessed sequentially, or they may be accessed randomly by a record's file address (RFA, discussed later).

VDS RMS consists of a set of routines that will service requests for reading files, along with a group of control structures that are used to pass information about the file back and forth between the diagnostic program and the VDS. VDS RMS supports three control structures: the file access block (FAB), an extended attribute block (XAB), and the record access block (RAB). When a program requests a file service, it usually must load fields within these control structures. The control structures contain information such as the name and type of file to be read along with codes indicating how the file is to be referenced.

3.15.3 The FAB, RAB, And XAB

The file access block (FAB) is a user control block that describes a particular file. An FAB is allocated by using the $FAB macro. One FAB must be defined for each file that is to be referenced.

The record access block (RAB) contains information about the file's records. There must be an RAB associated with each FAB. An RAB is allocated by using the $RAB macro.

An extended attribute block (XAB) is an optional control block that, if used, contains file attributes beyond those contained in a file's FAB. While VAX-11 RMS supports several different types of XABs, VDS RMS supports only the "file header characteristics XAB" (FHC XAB). The FHC XAB is allocated with the $XABFHC macro.

3.15.4 Accessing The VDS RMS Control Structures

The various fields of the FAB, RAB, and XAB can be initialized at program assembly time by using the predefined keywords that exist for each field. The fields can also be loaded at run time. The fields defined for each control block are named and described in the descriptions of the $FAB, $RAB, and $XABFHC macros in Chapter 4.

VDS RMS control structure fields are defined by a mnemonic of the general format

```
            structure$datatype_name
```

where "structure" is FAB, RAB, or XAB; "datatype" is a data type specifier (see Table 5-1); and name is the field name. Examples are FAB$L_FNA and RAB$V_BIO.

To access a structure field at run time, use the field name as an offset from the beginning of the structure. For example, suppose an FAB has been defined with the $FAB macro and has been labeled FAB_BLOCK. Accessing fields of the FAB in a MACRO-32 program would be done with instuctions such as

```
   MOVAB    FILE_NAME, FAB_BLOCK+FAB$L_FNA   ;Load filename addr.
or MOVB     RØ,FAB_BLOCK+FAB$B_FNS           ;Load filename size.
```

In BLISS-32, the same fields would be referenced with the statements

```
FAB_BLOCK [FAB$L_FNA] = FILE_NAME;          !Load filename addr.
FAB_BLOCK [FAB$B_FNS] = .FILE_NAME_SIZE;    !Load filename size.
```

For some fields, offsets have been defined. Mnemonics are defined for both the bit offsets and the mask values of these offsets. For example, the mnemonics FAB$V_BIO and FAB$M_BIO are defined for the bit offset and the mask value of BIO parameter in the FAC field of the FAB. Referencing this bit at run time in MACRO-32 could be accomplished with the following (unrelated) instructions.

```
   BISB     #FAB$M_BIO,FAB_BLOCK+FAB$B_FAC   ;Set BIO in FAB's FAC
or BBC      #FAB$V_BIO,FAB_BLOCK+FAB$B_FAC   ;Branch if BIO clear.
```

Similar BLISS-32 statements would be

```
   FAB_BLOCK [FAB$B_FAC] = .FAB_BLOCK [FAB$B_FAC] OR FAB$M_BIO;
   IF .FAB_BLOCK [FAB$B_FAC] <FAB$V_BIO,1> THEN ... ;
```

When a control block is declared (with the $FAB, $RAB, or $XABFHC macro), relevant fields may be initialized at compile time by using keyword representations of the fields. An example (in MACRO-32) is

```
$FAB    FAC = <BIO,GET>,-
        FOP = RWO,-
        XAB = FHCXAB
```

Similarly, fields can be loaded at run time with the $FAB_STORE, $RAB_STORE, and $XABFHC_STORE macros, as in this BLISS-32 example.

```
$RAB_STORE      (BKT = 10,
                 FAB = FAB_BLOCK,
                 RAC = SEQ,
                 FNA = .FILE_NAME [ADDRESS],
                 FNS = .FILE_NAME [SIZE]);
```

Using the STORE macros is a run-time alternate to directly referencing the fields, described above.

3.15.5 Reading Files

Two methods are available for reading files. These methods are "record processing" and "block processing." When a file is being referenced, the programmer may use whichever method is more appropriate to the file and operations being performed. It is also possible to combine the two methods.

3.15.6 Record Processing

When using record processing, reading a file involves accessing records within the file. The number, size, and contents of a file's records are immaterial to RMS and are determined by whatever utility created the file.

Two access methods are available for referencing records. The record access method is specified by loading the record access (RAC) field in the RAB. When specifying the RAC field, one of the following values may be chosen.

- SEQ - sequential access

  Records retrieved through sequential access are returned in the order in which they were stored. Once all the records have been retrieved, any further attempt to sequentially access records in the file will cause an end-of-file condition to be returned.

- RFA - record's file address access

  Whenever a record is read from a file, an internal representation of the record's location within the file is returned in the RFA field of the RAB. VDS RMS can save the value contained in the RFA field and use it to again retrieve that record by using a random-by-RFA request. (Note: In VDS RMS, random-by-RFA access is supported for both disks and magnetic tapes.)

Before the records of a file can be read, a "record stream" to the file must be established. A record stream is the association of an RAB to an FAB. After the file has been opened with the $OPEN macro, the record stream is established by placing the address of the FAB into the FAB field of the RAB. Then a $CONNECT macro is issued.

Once the record stream has been established, records in the file can be read using the $GET macro. The first $GET will cause the file's first record to be read, and each successive $GET will fetch the next record, if the RAB's RAC field is set to SEQ. If the RAC field is set to RFA, then each $GET will retrieve the record whose record file address (RFA) is stored in the RAB's RFA field.

To break the record stream after record processing has been completed, a $DISCONNECT macro is issued. Then the $CLOSE macro is used to terminate processing of the file.

Example 3-8 illustrates record processing of a file.

```
;
; This routine reads a sequential file into a buffer.
;

        .PSECT  DATA,WRT,NOEXE
BUFFER: .BLKB   1000               ; Allocate a 1000-byte buffer
BUFF_DESC:                         ; Descriptor for buffer
        .LONG   0                  ; Length will be set at run time
        .LONG   BUFFER             ; Address of buffer

MY_FAB: $FAB    FNM = <INFILE:>    ; File access block
MY_RAB: $RAB    FAB=MY_FAB,-       ; Record access block
                UBF=BUFFER,-
                USZ=100

        .PSECT  CODE,NOWRT,EXE
        .ENTRY  SIMPLE, ^M<>

        $OPEN   FAB=MY_FAB         ; Open the file.
        BLBC    R0,EXIT            ; Exit on error.
        $CONNECT RAB=MY_RAB        ; Connect for record operations.
        BLBC    R0,EXIT            ; Exit on error.

GET_RECORD:
        $GET    RAB=MY_RAB         ; Get a record
        BLBC    R0,CHECK_DONE      ; Branch on error.
        ADDL    MY_RAB+$W_RSZ,-    ; Advance buffer pointer
                MY_RAB+RAB$L_BUF
        BRB     GET_RECORD         ; Get another record

CHECK_DONE:
        CMPL    R0,#RMS$_EOF       ; Done?
        BNEQ    ERRORS             ; No -- error.
        $CLOSE  FAB=MY_FAB         ; Close the file.
        RET                        ; Return.

ERRORS:
        (Error handler.)
```

Example 3-8 Record Processing with RMS

3.15.7 Block Processing

Block processing makes it possible to directly read the blocks of a file, ignoring any sort of record structure that might exist for the file.

To indicate that block I/O will be performed on a file, the BIO bit in the FAC field of the FAB must be set before issuing the $OPEN macro. To perform block processing, the file must first be opened with the $OPEN macro. Then an RAB must be associated with the file's FAB by using the $CONNECT macro. Blocks can then be read from the file using the $READ macro. The first $READ will cause the first block of the file to be read. Each subsequent $READ will fetch the next sequential block of the file.

When file processing has been completed, issue the $DISCONNECT macro followed by the $CLOSE macro.

3.15.8 Mixing Block Processing And Record Processing

If the BRO bit in the FAC field of the FAB is set, then both block processing and record processing may be performed on the file. The BRO bit cannot be set after the $OPEN macro has been issued.

It is possible to initially allow both block processing and record processing, then to at some later time disable record processing and only allow block processing. This is accomplished by setting the BIO bit in the ROP field of the RAB (NOT the BIO bit in the FAC field of the FAB!). Once this bit is set, no further record processing will be allowed.

Mixing processing modes requires some caution. For example, when switching from block reads to record reads on a disk, RMS's next record pointer and its next block pointer are both undefined, so the first $GET after a $READ and the first $READ after a $GET must both use random-by-RFA access. For magnetic tape devices, the pointers will indicate the next block of the tape.

CHAPTER 4
VDS MACROS

4.1 INTRODUCTION

This chapter describes in detail the format and function of each macro used in VDS diagnostic programs. The macros are grouped in four sections. Within each section, macros are listed alphabetically by the macro name, ignoring the name's prefix ($DS_ or $).

The "program structure macros" are used to define the various sections, tables, and data structures making up a diagnostic program.

The "program control macros" are used to affect the program's execution path and provide facilities such as looping and branch-on-error.

The "system service macros" are used to call system services provided by the VDS.

The "symbol definition macros" are used to define global symbols used by the other macros, the VDS, and the diagnostic program. For programs written in MACRO-32, these macros must be issued before any of the symbols defined by the macros are referenced. In BLISS-32 programs, however, the symbols are not defined within symbol definition macros and thus may be referenced without first issuing symbol definition macros. For BLISS-32 programs, the documentation on symbol definition macros provided in this guide can be used simply as a list of available symbols.

4.2 CODING SYSTEM SERVICE MACRO CALLS

The VDS system services are invoked by issuing a macro call for the desired service and, if required, including an argument list to provide values for the macro's parameters. Before any system service macros can be called, the $DS_DSSDEF macro must be declared. This macro defines the system service entry points.

4.2.1 Fields Of The Macro Name

Macro names consist of three fields. These fields are:

- A prefix.

  This prefix may be '$DS_' or '$'. Macro names having the '$DS_' prefix are defined exclusively for use with the VAX Diagnostic Supervisor. Macro names having the '$' prefix are defined for use not only with the VAX Diagnostic Supervisor, but also for any program running under the VAX/VMS operating system.

  Diagnostic programmers should not assume that a macro name's prefix implies any restriction on the run-time environment in which the macro may be used. For instance, do NOT assume that macros with the '$' prefix may only be used for level 2R programs. Any run-time environment restrictions that may exist for a particular macro will be spelled out in the description of the macro.

  Because of the different prefixes, macro names have been alphabetized in this chapter by ignoring the prefix. Thus, for example, $BINTIM will be located after $DS_ABORT, and $DS_SETMAP will follow $READEF.

- A name.

  This name identifies the system service being invoked by the macro call.

- A suffix.

  For MACRO-32 programs this suffix may be '_S', '_G', '_L', or '_DEF'.

  The '_S' suffix indicates that the system service routine is to be called with a CALLS MACRO-32 instruction. If this suffix is used, the macro call must include an argument list to provide values for required parameters. (Specifying argument lists is detailed below.) Following is an example of the '_S' form of the macro call:

```
    $DS_ERRHARD_S -
            UNIT = LOG_UNIT, -
            MSGADR = HARD12_MSG, -
            PRLINK = HARD_MSGRTN, -
            P1 = SAVED_STATUS
```

If the '_G' suffix is used, the system service routine will be called with a CALLG MACRO-32 instruction. In this case, only one argument is specified with the macro call; that argument is the address of a list of arguments to the system service. Following is an example of the '_G' form of the macro call:

```
    $DS_ERRHARD_G HARD_ARGLIST
```

The '_L' suffix will not call the system service. It will generate an argument list. This argument list may later be passed to the system service when the service is called with a '_G' suffix, if the list's address is used as the macro call's argument. Following is an example of the '_L' form of the macro call:

```
HARD_ARGLIST:
    $DS_ERRHARD_L   UNIT = LOG_UNIT, -
                    MSGADR = HARD12_MSG, -
                    PRLINK = HARD_MSGRTN, -
                    P1 = SAVED_STATUS
```

The '_DEF' suffix simply generates symbolic names for the service's parameters. These symbolic names can be used to fill in fields of an argument list that was defined with a '_L' macro. Names will consist of the service name, a "$", an '_', and the parameter name. The symbolic names should be used as offsets from the beginning of the argument list. Following is an example of the '_DEF' form of the macro call:

```
    $DS_ERRHARD_DEF
            :
            :
            :
    MOVAL   HARD13_MSG, HARD_ARGLIST+ERRHARD$_MSGADR
```

For BLISS-32 programs, the suffix field of the macro call is always left blank. System services are always called with a CALLS MACRO-32 instruction, and the macro call must include an argument list. (Specifying argument lists in BLISS-32 is decribed in the next section.) Following is an example of invoking a system service in BLISS-32.

```
    $DS_ERRHARD
            (UNIT = .LOG_UNIT,
            MSGADR = HARD12_MSG,
            PRLINK = HARD_MSGRTN,
            P1 = .SAVED_STATUS);
```

4.2.2 Macro Arguments

Most system services possess a set of input parameters for which values must be provided when a service is invoked. Values are associated with input parameters via arguments to the service's macro call.

For MACRO-32 programs, macro arguments may be specified in either of two ways:

1. Arguments may be specified as a list with each argument except the last followed by a comma. The position of each argument is significant and thus arguments must be listed in the order specified in the macro's description. If a particular argument is optional and is to be omitted, a comma must be included to signify its omission. An example of a macro call using positional specification of arguments is:

   ```
   $DS_GETBUF_S     #3,,, #1
   ```

2. Arguments may be specified by "keywords." Keywords are symbolic names that are assigned to input parameters. A keyword is defined for every parameter of every macro, and that keyword is the name used to identify the parameter in the description of the macro's MACRO-32 format. For example, the $DS_GETBUF macro's MACRO-32 format is defined as:

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

   (brackets indicate optional arguments). Specifying this macro's arguments with keywords would appear as:

   ```
   $DS_GETBUF_S PAGCNT=#3, REGION=#1
   ```

   Notice that when using keywords, it is not necessary to include commas for unspecified arguments.

For BLISS-32 programs, macro arguments may also be specified positionally or by keyword, but the choice is NOT up to the programmer. For some macros, arguments must be specified with keywords. For others, arguments must be specified positionally. If the description of the macro's BLISS-32 format specifies keywords (capital letters followed by an equal sign), the keyword must be used. If the description does not indicate keywords, then positional specification is required.

4.2.3 Return Status Codes

All system services return an error status code in RØ. This status code should always be examined immediately after the diagnostic program regains program control from the service.

All status codes have symbolic names associated with them. Each of these names will have one of three possible prefixes. These prefixes are

- SS$_ - Most status codes begin with this prefix. For MACRO-32, these codes are defined by the $SSDEF macro.
- RMS$_ - Status codes associated with Record Management Services (RMS) begin with this prefix. For MACRO-32, these codes are defined by the $RMSDEF macro.
- DS$_ - A few status codes begin with this prefix. Such codes are defined for MACRO-32 by the $DS_DSDEF macro.

For status codes whose symbolic names begin with "SS$_" or "RMS$_", the low-order three bits indicate the severity of the error. Severity codes are as follows:

Value (Binary)	**Meaning**	**Symbolic Name**
ØØØ	Warning	STS$K_WARNING
ØØ1	Success	STS$K_SUCCESS
Ø1Ø	Error	STS$K_ERROR
Ø11	Informational	STS$K_INFO
1ØØ	Severe or fatal error	STS$K_SEVERR
1Ø1-111	Reserved	

Symbolic names are defined by VMS with the $STSDEF macro.

SS$_NORMAL vs. DS$_NORMAL - Most services return the "normal" status to indicate that the service was successfully completed. For some services, the correct prefix on the "normal" return code is "SS$_"; for other services, "DS$_" is the proper prefix. These two status codes are NOT interchangable. Care must be taken that a program's code uses the proper "normal" status code for the particular service being invoked. Each service's macro description will indicate which code is correct.

For all status codes that indicate an error condition, bit Ø of RØ will be cleared. For all other status codes, bit Ø of RØ will be set. Thus for MACRO-32 programs it is possible to determine that an error has occurred by simply using the BLBS or BLBC instruction. However, this method is NOT recommended. Program readablility is improved if status codes are always tested by using symbolic names, as in the example:

```
$QIO_G   QIO_ARGLIST      ;Enqueue I/O request.
CMPL     RØ, #SS$_NORMAL  ;If success, then continue.
BNEQ     QIO_ERROR        ;Else branch to the error handler.
                          ;Continue
```

4.3 CONVENTIONS USED IN THIS CHAPTER

In the macro descriptions that follow, certain conventions have been adhered to. These conventions are as follows:

1. For macros that accept arguments, those arguments that are optional have been indicated by enclosing the parameter name in brackets ([...]).

2. Macro parameters are listed in positional order. That is, if arguments are to be listed positionally, they must be listed in the order indicated in the macro format.

3. For MACRO-32, the parameter name indicates the keyword that must be used if arguments are to be specified with keywords.

4. For BLISS-32, keywords are indicated in capital letters. If a keyword is not supplied in the macro format, then the macro will not accept keyword arguments. In such a case arguments must be specified positionally.

5. The description of each macro parameter will indicate whether the argument supplied for that parameter must be a "value," an "address," or a "string."

   - Values as arguments - If a value is required, the argument will be interpreted as a value. Thus if a literal is specified for the argument, that literal will be interpreted as being the argument. If an address is specified, the CONTENTS of that address will be interpreted as being the argument.

   - Addresses as arguments - If an address is required, the argument will be interpreted as an address. No translation of the argument occurs.

   - Strings as arguments - If a string is required, the argument will be interpreted as a literal string. For MACRO-32, strings must be enclosed in angle brackets (<...>). For BLISS-32, strings must be enclosed in single quotation marks ('...'), and if the string itself is to contain the (') character it must be included twice, as in 'Debbie''s Program'.

6. Some services require that the address of a "quadword descriptor" or "character string descriptor" be passed. For our purposes, these terms are interchangeable and refer to a quadword that describes a string in the manner illustrated by Figure 4-1.

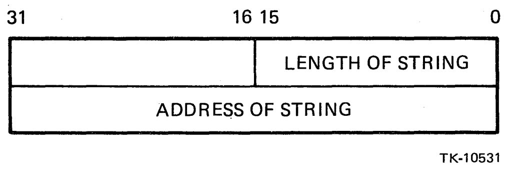

Figure 4-1 Quadword String Descriptor

String descriptors can be generated by using the .ASCID directive in MACRO-32, the %ASCID directive in BLISS-32, or the $DS_STRING macro.

4.4 PROGRAM STRUCTURE MACROS

$DEF

The $DEF macro is used to define a field in a data structure. It is defined in the VMS system library LIB.MLB. This macro can be used to define p-table desciptors, as discussed in Section 3.2.2.

MACRO-32 Format:

```
$DEF    sym, alloc, siz
```

BLISS-32 Format:

Not supported for BLISS-32.

sym

Symbolic name to be associated with the field.

alloc

Allocation unit. Use one of the MACRO-32 block storage directives for this parameter. MACRO-32 block storage directives are of the form ".BLKx," such as .BLKW or .BLKQ.

siz

Size of the field. This indicates the number of allocation units to assign.

Example:

```
$DEF   FIELD1, .BLKL, 1   ;Field named FIELD1 is 1 lonsword.
```

$DEFINI - $DEFEND

The $DEFINI and $DEFEND macros are used to define data structures. The fields within the data structure are defined by using the $DEF macro. These macros are defined in the VMS system library LIB.MLB and can be used to define p-table desciptors, as discussed in Section 3.2.2.

MACRO-32 Format:

```
$DEFINI struc, gbl, dot

(data structure field definitions)

$DEFEND struc, gbl
```

BLISS-32 Format:

Not supported for BLISS-32.

struc

Symbolic name that was assigned to the structure by the $DEFINI macro.

gbl

GLOBAL or LOCAL. Indicates whether the data structure's symbolic name ("struc") will be defined globally or locally.

dot

Address of the first field within the data structure. The symbol defined by the first $DEF macro will be assigned to this value. Subsequent fields are assigned to the next sequential memory addresses. The argument can be numeric (for example, 512), or symbolic (for example, BLOCK_ADDR). If symbolic, the symbol must be defined before the $DEFINI macro call.

Example:

```
$DEFINI TABLE1, GLOBAL, OFFSET
$DEF    FIELD1, .BLKL, 2
$DEF    FIELD2, .BLKB, 1
$DEFEND TABLE, GLOBAL
```

In this example, a global data structure named "TABLE1" has been defined to contain two fields, called FIELD1 and FIELD2. FIELD1 starts at location TABLE1+OFFSET and consists of 2 longwords. FIELD2 immediately follows FIELD1 and is one byte long.

$DS_$ADD

The $DS_$ADD p-table descriptor macro is used to add the contents of the "value register" (see Section 3.2.3.3) into a field of the p-table being built. The field is fetched, the addition is performed, and the result is placed back into the field.

Macro-32 Format:

```
$DS_$ADD (offset, pos, size)
```

Bliss-32 Format:

```
$DS_$ADD (OFFSET=offset, POS=pos, SIZ=size);
```

offset

The byte offset into the p-table of the field to which the contents of the value register are to be added.

pos

Bit position of the field, relative to the beginning of the byte specified by "offset." If the field starts on a byte boundary, this value will be Ø.

size

Number of bits making up the field. The size cannot be larger than 32.

Notes:

1. Bits added (or carried) beyond the field width are lost.
2. The contents of the value register are not changed.
3. Code generated by macro (shown in Macro-32; Bliss-32 is equivalent):

```
.BYTE   ^X8A             ; Beginning of ADD directive
.WORD   offset           ; Word data structure offset
.BYTE   pos              ; Bit position in word
.BYTE   size             ; Bit field size
```

Examples:

Macro-32 Examples:

```
     $DS_$ADD       OFFSET=HP$A_DEVICE, POS=0, SIZE=32

     $DS_$ADD       <^X40>, 0, 32
```

Bliss-32 Examples:

```
     $DS_$ADD (OFFSET=%FIELDEXPAND(HP$A_DEVICE,0),
                 POS=%FIELDEXPAND(HP$A_DEVICE,1),
                 SIZE=%FIELDEXPAND(HP$A_DEVICE,2));

     $DS_$ADD (OFFSET=%X'40', POS=0, SIZ=32);
```

$DS_$CASE

The $DS_$CASE p-table descriptor macro is used to test the current contents of the "value register" (see Section 3.2.3.3) and then load a new value into the register, depending on the old contents. The $DS_$CASE macro is used to specify pairs of values. The current value register contents are compared with the first value of each pair until a match is found; the second value of the pair is then loaded into the value register. There may be any number of pairs in the case list. If no pair matches the value register, then the value register is not altered.

Macro-32 Format:

```
$DS_$CASE <<case_pair>, [<case_pair>, ...]>
```

Bliss-32 Format:

```
$DS_$CASE ((case_pair), [(case_pair), ...]);
```

case_pair

A pair of values, separated by a comma. Each value will be stored in a longword.

Notes:

1. Code generated by macro (shown in Macro-32; Bliss-32 is equivalent):

```
.BYTE   ^X8C              ; Beginning of CASE
.BYTE   n                 ; Number of case pairs
.LONG   match1, value1    ; First case pair
        .
        .                 ; Other case pairs
        .
.LONG   match-n, value-n; Last (nth) case pair
```

Examples:

Macro-32 Example:

```
     $DS_$CASE < -
                    <1,2>, -
                    <2,3>, -
                    <3,4>>

     $DS_$CASE <<1,<^XFFFFF>>,<2,<^XFFFEFFFF>>>
```

Bliss-32 Example:

```
     $DS_$CASE (
                    (1,2),
                    (2,3),
                    (3,4));

     $DS_$CASE ((1,%X'FFFFF'),(2,%X'FFFEFFFF'));
```

$DS_$COMPLEMENT

This p-table descriptor macro complements the current contents of the value register.

Macro-32 Format:

```
$DS_$COMPLEMENT
```

Bliss-32 Format:

```
$DS_$COMPLEMENT;
```

Notes:

1. Code generated by macro (shown in Macro-32; Bliss-32 is equivalent):

   ```
   .BYTE ^X89 ;  Complement value register
   ```

$DS_$DECIMAL

This p-table descriptor macro reads a value from the ATTACH command line. If no more parameters are available on the command line, or if the next parameter is not a decimal value, it will prompt the operator with the prompt text value. The value that is read is stored in the "value register" (see Section 3.2.3.3) for use by a $DS_$COMPLEMENT, $DS_$STORE, or $DS_$CASE statement.

Macro-32 Format:

```
$DS_$DECIMAL <prompt>, low, high
```

Bliss-32 Format:

```
$DS_$DECIMAL (PROMPT='prompt', LOW=low, HIGH=high);
```

prompt
Character string that is to be printed as a prompt to the user. This prompt will be used if the ATTACH command line does not contain the required value.

low

The low limit for the value. If the value given is lower than this, an error message followed by the prompt message will be displayed. The default radix for this value is decimal.

high

The high limit for the value. If the value given is higher than this, an error message followed by the prompt message will be displayed. The default radix for this value is decimal.

Notes:

1. Code generated by macro (shown in Macro-32; Bliss-32 is equivalent):

```
.BYTE   ^X82              ; Beginning of DECIMAL prompt
.ASCIC  \prompt\          ; Prompt string
.LONG   low               ; Low limit
.LONG   high              ; High limit
```

Examples:

Macro-32 Example:

```
$DS_$DECIMAL TR, 1, 15

$DS_$DECIMAL PROMPT=<NUMBER OF ARRAY CARDS>, LOW=0, HIGH=15
```

Bliss-32 Example:

```
$DS_$DECIMAL (PROMPT='TR', LOW=1, HIGH=15);

$DS_$DECIMAL (PROMPT='NUMBER OF ARRAY CARDS', LOW=0, HIGH=15);
```

$DS_$END

The $DS_$END macro is used to terminate a p-table descriptor.

Macro-32 Format:

```
$DS_$END
```

Bliss-32 Format:

```
$DS_$END;
```

Notes:

1. Code generated by macro (shown in Macro-32; Bliss-32 is equivalent):

   ```
   .BYTE ^X81 ;  End of p-table descriptor
   ```

$DS_$FETCH

The $DS_$FETCH macro is used in p-table descriptors. It will extract the contents of a field within the p-table, and store the contents, right-justified, in the "value register" (see Section 3.2.3.3). It is possible to reference a device-dependent field that was filled with a previous $DS_$STORE macro, or device-independent field that was filled by the VDS. The macro can also be used to facilitate temporary storage, by storing a value in the p-table while the value register is needed for something else, then restoring the old value.

Macro-32 Format:

```
$DS_$FETCH offset, pos, size
```

Bliss-32 Format:

```
$DS_$FETCH (OFFSET=offset, POS=pos, SIZE=size);
```

offset

The byte offset into the p-table of the field from which the contents are to be fetched.

pos

Bit position of the field, relative to the beginning of the byte specified by "offset." If the field starts on a byte boundary, this value will be Ø.

size

Number of bits making up the field. The size cannot be larger than 32.

Notes:

1. Code generated by macro (shown in Macro-32; Bliss-32 is equivalent):

```
.BYTE   ^X87            ; Beginning of FETCH directive
.WORD   offset          ; Word data structure offset
.BYTE   pos             ; Bit position in word
.BYTE   size            ; Bit field size
```

Examples:

Macro-32 Example:

```
     $DS_$FETCH OFFSET=HP$A_DVA, POS=0, SIZE=32

     $DS_$FETCH <^x40>, 0, 32
```

Bliss-32 Example:

```
     $DS_$FETCH (OFFSET=%FIELDEXPAND(HP$A_DVA,0),
                 POS=%FIELDEXPAND(HP$A_DVA,1),
                 SIZE=%FIELDEXPAND(HP$A_DVA,2));

     $DS_$FETCH (OFFSET=%X'40', POS=0, SIZ=32);
```

$DS_$HEX

The $DS_$HEX p-table descriptor macro is used to read a value from the ATTACH command line. If no more parameters are available on the command line, or if the next parameter is not a hex value, the user will be prompted with the prompt text value. The value that is read is left in the "value register" (see Section 3.2.3.3) for use by a $DS_$COMPLEMENT, $DS_$STORE, or $DS_$CASE statement.

Macro-32 Format:

```
$DS_$HEX <prompt>, low, high
```

Bliss-32 Format:

```
$DS_$HEX (PROMPT='prompt', LOW=low, HIGH=high);
```

prompt

Character string that is to be printed as a prompt to the user. This prompt will be used if the ATTACH command line does not contain the required value.

low

The low limit for the value. If the value given is lower than this, an error message followed by the prompt message will be displayed. For MACRO-32, the default radix for this value is hexadecimal. For BLISS-32, the default radix is decimal.

high

The high limit for the value. If the value given is higher than this, an error message followed by the prompt message will be displayed. For MACRO-32, the default radix for this value is hexadecimal. For BLISS-32, the default radix is decimal.

Notes:

1. Code generated by macro (shown in Macro-32; Bliss-32 is equivalent):

```
.BYTE   ^X84           ; Beginning of HEX prompt
.ASCIC  \prompt\       ; Prompt string
.LONG   ^X<low>        ; Low limit
.LONG   ^X<high>       ; High limit
```

Examples:

Macro-32 Example:

```
$DS_$HEX <WCS Last address>,0,FFF0
```

Bliss-32 Example:

```
$DS_$HEX (PROMPT='WCS Last address', LOW=0, HIGH=%X'FFF0');
```

$DS_$INITIALIZE

The $DS_$INITIALIZE p-table descriptor macro must be the first macro in every p-table descriptor. It is used to indicate the device type, the p-table's total size, the maximum number of units allowed by the hardware, and the name of the device driver required for a level 2 diagnostic program to reference the device.

MACRO-32 Format:

```
$DS_$INITIALIZE device, length, max, driver
```

BLISS-32 Format:

```
$DS_$INITIALIZE (DEVICE=device, LENGTH=length, MAX=max,
DRIVER=driver);
```

device

Character string representing the device type of the hardware being described by the p-table, such as RK611, RK06, RM03, RH780, and so on. The string specified here will be the string that the user must type as the first argument to the ATTACH command, as in "ATTACH RK611".

length

The length (in bytes) of the p-table that is to be created. The length includes both the device-independent and the device-dependent fields. Generally, a symbolic name for this value is created with a $DEF macro during memory allocation specifications, as illustrated in Section 3.2.2.3.

max

The maximum number of units that can exist. This number is controlled by the hardware design. For example, the number would be 8 for an RK07, since that is the maximum number of RK07 drives that can exist on an RK711 controller.

Some devices, such as controllers and adapters, are not assigned a unit number. For these cases, "max" should be 0.

If this value is greater than Ø, and if the $DS_$NAME macro is not used, the device's generic name will be required to contain a unit number. If, on the other hand, the $DS_$NAME macro is used, then whether or not a unit number must be typed is controlled by the $DS_$NAME statement.

The default value for "max" is Ø.

driver

The name of the QIO driver (if any) needed by level 2 diagnostic programs in order to reference the device. The value must be a string of two characters. The string given, 'dn', determines the driver loaded as follows: the string is appended to the string 'EVQ' and followed by the file type '.EXE'. Thus the driver's filename is 'EVQdn.EXE'.

Notes:

1. Code generated by macro (shown in Macro-32; Bliss-32 is equivalent):

```
.ASCIC  \Device\          ; ASCIC device type
.BYTE   Length            ; Length of p-table
.BYTE   Max_Units         ; Maximum unit number
.WORD   ^A"Driver"        ; Driver suffix
.BYTE   ^X8Ø              ; End of initialization statement
```

Examples:

Macro-32 Examples:

```
$DS_$INITIALIZE       DEVICE=DMC11, -
               LENGTH=HP$K_DMC11_LEN, -
               DRIVER=<XM>

$DS_$INITIALIZE       DEVICE=DW780, -
               LENGTH=HP$K_DW780_LEN, -
               MAX=3

$DS_$INITIALIZE       RK611, HP$K_RK611_LEN, 0
```

Bliss-32 Examples:

```
    $DS_$INITIALIZE (DEVICE='DMC11',
                     LENGTH=HP$K_DMC11_LEN,
                     DRIVER='XM');

    $DS_$INITIALIZE (DEVICE='DW780',
                     LENGTH=HP$K_DW780_LEN,
                     MAX=3);

    $DS_$INITIALIZE (DEVICE='RK611', LENGTH=HP$K_RK611_LEN);
```

$DS_$LITERAL

This p-table descriptor macro is used to load a literal value into the value register. This value can then be manipulated by a $DS_$COMPLEMENT, $DS_$STORE, or $DS_$CASE statement.

Macro-32 Format:

```
$DS_$LITERAL lit
```

Bliss-32 Format:

```
$DS_$LITERAL (LIT=lit);
```

Value (longword) to be loaded into the "value register" (see Section 3.2.3.3).

Notes:

1. Code generated by macro (shown in Macro-32; Bliss-32 is equivalent):

```
.BYTE   ^X86             ; Beginning of LITERAL
.LONG   lit              ; Literal value
```

Examples:

Macro-32 Examples:

```
$DS_$LITERAL LIT=^XFF

$DS_$LITERAL ^O776
```

Bliss-32 Examples:

```
$DS_$LITERAL (LIT=%X'FF');

$DS_$LITERAL (LIT=%O'776');
```

$DS_$LOGICAL

This p-table descriptor macro is used to read a "yes" or "no" response from an ATTACH command line. The expected response is one of the strings 'YES' or 'NO'. They may be abbreviated, and may be upper or lower case. The value register will be loaded with a Ø if the response was "no," or with a 1 if the response was "yes."

Macro-32 Format:

```
$DS_$LOGICAL <prompt_>
```

Bliss-32 Format:

```
$DS_$LOGICAL (PROMPT='prompt');
```

prompt

A character string representing the prompting message to be displayed by the ATTACH command processing routine of the VDS.

Notes:

1. Code generated by macro (shown in Macro-32; Bliss-32 is equivalent):

```
.BYTE   ^X8B            ; Beginning of LOGICAL prompt
.ASCIC  \prompt\        ; Prompt string
```

Examples:

Macro-32 Examples:

```
$DS_$LOGICAL <Load WCS_>
```

Bliss-32 Examples:

```
$DS_$LOGICAL (PROMPT='Load WCS');
```

$DS_$NAME

The $DS_$NAME p-table descriptor macro is used if device name validation is desired. If used, the macro must immediately follow the $DS_$INITIALIZE macro. When this macro is present, the device generic name (the third argument to the ATTACH command) must conform to the naming conventions specified. (See note 1 for exceptions.)

All device names can be described by the general formula 'ggan'; where 'gg' is a generic device prefix (not necessarily only two characters), 'a' is a letter representing which controller or bus adapter the device is on, and 'n' represents the device's unit number on that controller or adapter. Both the 'a' and 'n' portions are optional, but every device must have a 'gg' portion. For most devices, 'gg' is fixed by the physical type of the device; or, it may be determined by its LINK device (the controller to which it is attached). The $DS_$NAME statement allows specification and enforcement of these rules.

Macro-32 Format:

```
$DS_$NAME flags, generic
```

Bliss-32 Format:

```
$DS_$NAME (FLAGS=flags, GENERIC=generic);
```

flags

Flag bits that control the format of the device name. Symbolic names for the flags are defined by the $DS_PTDDEF macro. The flag bits are:

- Bit Ø - PTD$M_UNIT - The 'n' portion of the generic name is required for this device. Its maximum value is specified by the "max" parameter of the $DS_$INITIALIZE macro.
- Bit 1 - PTD$M_CONTROLLER - The 'a' portion of the generic name is required for this device. If the bit PTD$M_INHERIT_CON is also set, the 'a' portion must match the 'a' portion of the controller to which this device is attached.

- Bit 2 - PTD$M_NAME - Only the 'gg' portion of the generic name is required. This is most common for network devices, which are known by their DECnet names (for example, YODA, STAR, GALAXY).

- Bit 3 - PTD$M_INHERIT_PRE - The 'gg' device name prefix is inherited from the controller to which the device is attached. This, for example, allows a VT1ØØ to require a name of the form 'TTan' when attached to a DZ11 ('TTa'), or 'TXan' when attached to a DMF32A ('TXa').

- Bit 4 - PTD$M_INHERIT_CON - The 'a' controller designator portion of the device name is inherited from the controller to which the device is attached. This, for example, allows a VT1ØØ to require a name of the form 'TTAn' when attached to DZ11 'TTA', or 'TTBn' when attached to DZ11 'TTB'.

- Bits 5 to 7 are reserved for future expansion and must not be set by any p-table descriptor.

Additionally, several special names are defined that combine common sets of these flag bits. They are:

- PTD$M_INHERIT - This combines the bits PTD$M_INHERIT_PRE and PTD$M_INHERIT_CON. This is the normal permutation of the two bits.

- PTD$M_DEVICE - This combines the bits PTD$M_CONTROLLER and PTD$M_UNIT. It would commonly be used for devices that are connected directly to a bus, rather than a controller, and therefore require both 'a' and 'n' portions but should not inherit them from their LINK device.

- PTD$M_ENDDEVICE - This combines the bits PTD$M_CONTROLLER, PTD$M_UNIT, and PTD$M_INHERIT. It would commonly be used for devices that have controllers, such as an RKØ7 that is attached to an RK711, and should inherit the controller's name prefix and controller letter.

The default is PTD$M_DEVICE.

generic

The 'gg' portion required for this device. If the flag PTD$M_INHERIT_PRE is set, this argument is used only if the device is linked to HUB.

Notes:

1. The naming conventions specified with the $DS_$NAME will be ignored if the VDS is running under APT, or if the VDS is executing a script file. This is to ensure compatability with APT scripts and VDS scripts that do not adhere to proper naming conventions.

2. Code generated by macro (shown in Macro-32; Bliss-32 is equivalent):

```
.BYTE   ^X8D            ; Start of NAME statment
.BYTE   flags           ; Generic name format flags
.ASCIC  "generic"       ; Enforced generic name
```

Examples:

Macro-32 Examples:

```
$DS_$NAME FLAGS=PTD$M_ENDDEVICE, GENERIC=DM

$DS_$NAME PTD$M_UNIT, DM
```

Bliss-32 Examples:

```
$DS_$NAME (FLAGS=(PTD$M_ENDDEVICE), GENERIC='DM');

$DS_$NAME (FLAGS=(PTD$M_UNIT), GENERIC='KA');
```

$DS_$OCTAL

The $DS_$OCTAL p-table macro is used to read a value from the ATTACH command line. If no more parameters are available on the command line, or if the next parameter is not an octal value, the prompting message will be displayed to the user. The value that is read is stored in the "value register" (see Section 3.2.3.3) for use by a $DS_$COMPLEMENT, $DS_$STORE, or $DS_$CASE statement.

Macro-32 Format:

```
$DS_$OCTAL   prompt, low, high
```

Bliss-32 Format:

```
$DS_$OCTAL (PROMPT=prompt, LOW=low, HIGH=high);
```

prompt

Character string that is to be printed as a prompt to the user. This prompt will be used if the ATTACH command line does not contain the required value.

low

The low limit for the value. If the value given is lower than this, an error message followed by the prompt message will be displayed. For MACRO-32, the default radix of this value is octal. For BLISS-32, the default radix is decimal.

high

The high limit for the value. If the value given is higher than this, an error message followed by the prompt message will be displayed. For MACRO-32, the default radix of this value is octal. For BLISS-32, the default radix is decimal.

Notes:

1. Code generated by macro (shown in Macro-32; Bliss-32 is equivalent):

```
.BYTE   ^X83          ; Beginning of OCTAL prompt
.ASCIC  \prompt\      ; Prompt string
.LONG   ^O<low>       ; Low limit
.LONG   ^O<high>      ; High limit
```

Examples:

Macro-32 Examples:

```
$DS_$OCTAL CSR,760000,777776

$DS_$OCTAL PROMPT=<VECTOR_>, LOW=2, HIGH=776
```

Bliss-32 Examples:

```
$DS_$OCTAL (PROMPT='CSR', LOW=%O'760000', HIGH=%O'777776');

$DS_$OCTAL (PROMPT='VECTOR', LOW=%O'2', HIGH=%O'776');
```

$DS_$STORE

The $DS_$STORE p-table descriptor macro is used to load the contents of the "value register" (see Section 3.2.3.3) into a field of the p-table being built. The macro can be used to store values read by the $DS_$DECIMAL, $DS_$OCTAL, $DS_$HEX, $DS_$STRING, or $DS_$LOGICAL statements, or generated by the $DS_$LITERAL, $DS_$FETCH, $DS_$COMPLEMENT, or $DS_$CASE statements. It can also be used to facilitate temporary storage. A value can be stored in the p-table temporarily while the value register is needed for something else, then later restored with the $DS_$FETCH statement. This macro does not change the contents of the value register.

Macro-32 Format:

```
$DS_$STORE   offset, pos, size
```

Bliss-32 Format:

```
$DS_$STORE (OFFSET=offset, POS=pos, SIZE=size);
```

offset

The byte offset into the p-table of the field into which the contents of the value register are to be placed.

pos

Bit position of the field, relative to the beginning of the byte specified by "offset." If the field starts on a byte boundary, this value will be Ø.

size

Number of bits making up the field. The size cannot be larger than 32.

Notes:

1. Code generated by macro (shown in Macro-32; Bliss-32 is equivalent):

```
.BYTE   ^X88            ; Beginning of STORE directive
.WORD   offset          ; Word data structure offset
.BYTE   pos             ; Bit position in word
.BYTE   size            ; Bit field size
```

Examples:

Macro-32 Examples:

```
  $DS_$STORE       OFFSET=HP$L_RK611_CSR, POS=0, SIZE=32

  $DS_$STORE       <^X40>, 0, 32
```

Bliss-32 Examples:

```
  $DS_$STORE (OFFSET=%FIELDEXPAND(HP$L_RK611_CSR,0),
              POS=%FIELDEXPAND(HP$L_RK611_CSR,1),
              SIZE=%FIELDEXPAND(HP$L_RK611_CSR,2));

  $DS_$STORE (OFFSET=%X'40', POS=0, SIZ=32);
```

$DS_$STRING

The $DS_$STRING p-table descriptor macro is used to read a string from an ATTACH command line. If the string exists on the ATTACH command line, it will be used; otherwise the prompting message will be displayed. The string read from the command line is compared against a list of valid strings, and the number of the match string (Ø, 1, 2, and so on, in the order given) is returned in the value register. This can be used, for example, to determine if a DZ-11 line card to be tested is 'EIA' or '2ØMA', by the statement "$DS_$STRING ('Line type','EIA','2ØMA')" which would return Ø if the response was EIA, or 1 if the response was 2ØMA.

Macro-32 Format:

```
$DS_$STRING  <prompt_>, <string, [string, ...]_>
```

Bliss-32 Format:

```
$DS_$STRING ('prompt', 'string', ['string', ...]);
```

prompt

Character string to be used as a prompting message.

string

A character string with which the input string is to be compared. The number of the first string that exactly matches the input will be returned.

Notes:

1. Code generated by macro (shown in Macro-32; Bliss-32 is equivalent):

```
.BYTE   ^X85               ; Beginning of STRING prompt
.ASCIC  \prompt\           ; Prompt string
.ASCIC  \string1\          ; ASCIC string 1
        .
        .                  ; ASCIC strings
        .
.ASCIC  \stringn\          ; ASCIC string n
.BYTE   Ø                  ; List terminator
```

Examples:

Macro-32 Examples:

```
    $DS$STRING <Module type>, <<EIA>, <20MA>>

    $DS$STRING PROMPT=<Node type>, STRINGS=<780,750,730>
```

Bliss-32 Examples:

```
    $DS_$STRING ('Module type', 'EIA', '20MA');

    $DS_$STRING ('Node type', '780', '750', '730');
```

$DS_BGNCLEAN - $DS_ENDCLEAN

The $DS_BGNCLEAN and $DS_ENDCLEAN macros are used to delimit the program's clean-up code. These macros create the connections which make it possible for the VDS to locate and execute the clean-up code.

MACRO-32 Format:

```
$DS_BGNCLEAN [<regmask>], [<psect>]

(clean-up code)

$DS_ENDCLEAN
```

BLISS-32 Format:

```
$DS_BGNCLEAN;

(clean-up code);

$DS_ENDCLEAN;
```

regmask

List of general purpose register names to be placed in the entry mask.

psect

Any argument string that is valid for a MACRO-32 .PSECT statement. If none is specified, the argument string "CLEANUP,LONG" will be used.

Notes:

1. In MACRO-32, the $DS_BGNCLEAN macro will generate the following code:

```
            .SAVE
            .PSECT psect
CLEAN_UP:
            .WORD ^M<regmask>
```

In MACRO-32, the $DS_ENDCLEAN macro will generate the following code:

```
CLEAN_UP_X:
            $DS_BREAK
            RET
            .RESTORE
```

2. In BLISS-32, the $DS_BGNCLEAN macro will generate the following code:

```
    %SBTTL 'CLEAN UP'
    PSECT CODE = CLEANUP(WRITE);
    GLOBAL ROUTINE CLEAN_UP:NOVALUE =
    BEGIN
```

In BLISS-32, the $DS_ENDCLEAN macro will generate the following code:

```
    END
```

Examples:

MACRO-32 Example:

```
    $DS_BGNCLEAN <R2,R3,R4,R5>, <CLEANSECT,LONG>
           :
           :
    $DS_ENDCLEAN
```

BLISS-32 Example:

```
    $DS_BGNCLEAN;
           :
           :
    $DS_ENDCLEAN;
```

$DS_BGNDATA - $DS_ENDDATA

The $DS_BGNDATA and $DS_ENDDATA macros are used to optionally provide lists of input arguments to a test. Each time the VDS executes a test for which argument lists have been specified, it will execute the code within the test once for each argument list. From the user's point of view, this repeated execution of the code within a test will appear to be one execution of the test.

The $DS_BGNDATA and $DS_ENDDATA macros must be located immediately before the $DS_BGNTEST macro of the test to which the parameter lists belong.

MACRO-32 Format:

```
$DS_BGNDATA [align]
argument-list
[argument-list]
        .
        .
        .
$DS_ENDDATA
```

BLISS-32 Format:

This macro is not supported for BLISS-32.

align

Desired alignment for the psect containing the argument lists. Possible values are BYTE, WORD, LONG, QUAD, PAGE, or an integer from Ø to 9. If an integer is specified, the psect will start at the next address that is a multiple of two raised to the power of the integer.

argument-list

A list of arguments to be used by the test. Each argument must occupy a longword. Each parameter list must be formatted as shown in Figure 4-2.

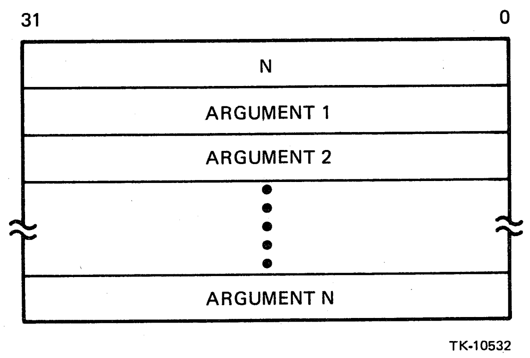

Figure 4-2 Argument List Format for $DS_BGNDATA - $DS_ENDDATA

The $DS_ENDDATA will provide termination for the set of lists by generating a longword of Ø.

Notes:

1. The VDS will call the test code with a CALLG instruction. Each time the test is called, the address of the next argument list will be used as the CALLG instruction's argument list parameter. Thus the arguments can be referenced within the test by offsets from the AP.

Example:

```
$DS_BGNDATA

.LONG   4, DATA_1, DATA_2, DATA_3, DATA_4
.LONG   4, DATA_5, DATA_6, DATA_7, DATA_8
.LONG   4, DATA_1, DATA_3, DATA_7, DATA_9

$DS_ENDDATA
```

$DS_BGNINIT - $DS_ENDINIT

The $DS_BGNINIT and $DS_ENDINIT macros are used to delimit the diagnostic program's initialization code. These macros create the connections that make it possible for the VDS to locate and execute the initialization code.

MACRO-32 Format:

```
$DS_BGNINIT [<regmask_>], [<psect_>]

(initialization code)

$DS_ENDINIT
```

BLISS-32 Format:

```
$DS_BGNINIT;

(initialization code);

$DS_ENDINIT;
```

regmask

List of general purpose register names to be placed in the entry mask.

psect

Any argument string that is valid for a MACRO-32 .PSECT statement. If none is specified, the argument string "INITIALIZE,LONG" will be used.

Notes:

1. In MACRO-32, the $DS_BGNINIT macro will generate the following code:

```
                .SAVE
                .PSECT psect
INITIALIZE:
                .WORD   ^M<regmask>
```

   In MACRO-32, the $DS_ENDINIT macro will generate the following code:

```
INITIALIZE_X:
            $DS_BREAK
            RET
            .RESTORE
```

2. In BLISS-32, the $DS_BGNINIT macro will generate the following code:

    ```
    %SBTTL 'INITIALIZE'
    PSECT CODE = INITIALIZE(WRITE);
    GLOBAL ROUTINE INITIALIZE : NOVALUE =
    BEGIN
    ```

    In BLISS-32, the $DS_ENDINIT macro will generate the following code:

    ```
    $DS_BREAK;
    END
    ```

Examples:

MACRO-32 Example:

```
$DS_BGNINIT <R2,R3,R4,R5>, <INITSECT,LONG>
        :
        :
$DS_ENDINIT
```

BLISS-32 Example:

```
$DS_BGNINIT;
        :
        :
$DS_ENDINIT;
```

$DS_BGNMESSAGE - $DS_ENDMESSAGE

The $DS_BGNMESSAGE and $DS_ENDMESSAGE macros should be used to delimit each error reporting routine used in conjunction with the error reporting macros ($DS_ERRxxxx).

MACRO-32 Format:

```
$DS_BGNMESSAGE [<regmask>]

(error reporting routine)

$DS_ENDMESSAGE
```

BLISS-32 Format:

```
$DS_BGNMESSAGE (ROUTINE_NAME=routine_name);

(error reporting routine);

$DS_ENDMESSAGE;
```

regmask

List of general purpose register names to be placed in the entry mask.

routine_name

Symbolic name to be associated with the error reporting routine.

Notes:

1. The error reporting routine must use $DS_PRINTB and $DS_PRINTX macros to print messages. The most important information should be printed first, using $DS_PRINTB macros. The most detailed information, such as dumps of device registers, should be printed last, using $DS_PRINTX macros. Refer to Section 3.9.1, Error Message Formats, for example error messages.

2. Further details on error reporting routines are listed in the description of the error macros ($DS_ERRxxxx).

3. In MACRO-32, the $DS_BGNMESSAGE macro generates an entry mask. The $DS_ENDMESSAGE macro generates a RET instruction.

4. In BLISS-32, THE $DS_BGNMESSAGE macro generates the following code:

```
GLOBAL ROUTINE %NAME(routine_name)(NUM, UNIT, MSGADR, PRLINK,
                     P1, P2, P3, P4, P5, P6) : NOVALUE =

BEGIN
```

   The $DS_ENDMESSAGE macro generates the following code:

```
   RETURN
   END
```

Examples:

Refer to the description of the $DS_ERRxxxx macros (later in this chapter) for examples of $DS_BGNMESSAGE and $DS_ENDMESSAGE.

$DS_BGNMOD - $DS_ENDMOD

The $DS_BGNMOD and $DS_ENDMOD macros must be included at the beginning and end, respectively, of every source module making up the diagnostic program.

MACRO-32 Format:

```
$DS_BGNMOD [env], [tn], [st]

(source module)

$DS_ENDMOD
```

BLISS-32 Format:

```
$DS_BGNMOD ([ENV=evn], [TEST=tn]);

(source module);

$DS_ENDMOD;
```

env

Used to indicate if the program is a level 2 program. If so, this value must be 2. Otherwise the value should be Ø (the default). NOTE: In the past, this parameter was assigned one of four predefined values: CEP_FUNCTIONAL, CEP_REPAIR, SEP_FUNCTIONAL, or SEP_REPAIR. These symbols are meaningless and should not be used. (SEP_FUNCTIONAL is equivalent to 2.)

tn

Value representing the number to be assigned to the first test in this module, if this module contains tests. Default value is 1.

st

Value representing the number to be assigned to the first subtest in this module, if this module contains subtests. Default value is 1.

Notes:

1. In BLISS-32, the $DS_BGNMOD and $DS_ENDMOD macros must be contained within the bounds of the MODULE and ELUDOM keywords, as follows.

```
MODULE modnam =
BEGIN
   :
   :
$DS_BGNMOD ();
   :
   :
   :
$DS_ENDMOD;
END
ELUDOM
```

$DS_BGNREG - $DS_ENDREG

The $DS_BGNREG and $DS_ENDREG macros may be used to delimit a storage area in which device register contents are placed.

MACRO-32 Format:

```
$DS_BGNREG

(register storage area)

$DS_ENDREG
```

BLISS-32 Format:

```
$DS_BGNREG;

(register storage area);

$DS_ENDREG;
```

Notes:

1. In MACRO-32, the $DS_BGNREG macro generates the label "DEVREG:".

   In BLISS-32, the $DS_BGNREG macro generates the statement

   ```
   OWN DEV_REG :  VECTOR [Ø];
   ```

2. The $DS_ENDREG does not generate any source code.

$DS_BGNSERV - $DS_ENDSERV

The $DS_BGNSERV and $DS_ENDSERV macros should be used to delimit interrupt service routines.

MACRO-32 Format:

```
$DS_BGNSERV addr

(interrupt service routine)

$DS_ENDSERV
```

BLISS-32 Format:

These macros are not supported for BLISS-32.

addr

Symbolic name to be associated with the interrupt service routine.

Notes:

1. The $DS_BGNSERV macro will generate the following code:

```
        .ALIGN   LONG, Ø        ; ALIGN ON LONGWORD BOUNDARY
ADDR:
        PUSHR    #^M<RØ,R1>     ; SAVE RØ AND R1
```

   The $DS_ENDSERV macro will generate the following code:

```
        POPR     #^M<RØ,R1>     ; RESTORE RØ AND R1
        REI                     ; RETURN FROM SERVICE
```

$DS_BGNSTAT - $DS_ENDSTAT

The $DS_BGNSTAT and $DS_ENDSTAT macros should be used to delimit the data storage area referenced by the summary routine (see Section 3.7, Summary Routines). This data area should contain a set of error counts for each unit under test. Thus there must be enough storage space allocated to handle the maximum number of device units the diagnostic program can test.

MACRO-32 Format:

```
$DS_BGNSTAT

(statistics tables)

$DS_ENDSTAT
```

BLISS-32 Format:

```
$DS_BGNSTAT;

(statistics tables);

$DS_ENDSTAT;
```

Notes:

1. In MACRO-32, the $DS_BGNSTAT macro simply generates the label 'STATISTIC:'. The $DS_ENDSTAT does not generate any code.

2. In BLISS-32, the $DS_BGNSTAT macro generates the following statement:

   ```
   GLOBAL STATISTIC :  VECTOR [0];
   ```

   The $DS_ENDSTAT macro does not generate any code.

$DS_BGNSUB - $DS_ENDSUB

The $DS_BGNSUB and $DS_ENDSUB macros are used to delimit each subtest existing in any particular test. Refer to Section 3.8, Tests, Subtests, and Sections, for a discussion of subtests.

MACRO-32 Format:

```
$DS_BGNSUB

(subtest)

$DS_ENDSUB
```

BLISS-32 Format:

```
$DS_BGNSUB;

(subtest);

$DS_ENDSUB;
```

Notes:

1. The macro automatically numbers each subtest. Subtests are numbered from 1 to N for each test, where N is the total number of subtests within the test.

2. The $DS_BGNSUB macro generates a call to a VDS routine that will record the numbers of the current test and subtest. The $DS_ENDSUB macro will generate a call to a VDS routine that will verify that the current test and subtest numbers are the same as those stored when the $DS_BGNSUB macro was issued. If the numbers do not match, the VDS will stop execution of the diagnostic program.

$DS_BGNSUMMARY - $DS_ENDSUMMARY

The $DS_BGNSUMMARY and $DS_ENDSUMMARY macros are used to delimit the summary routine. Summary routines are discussed in Section 3.7.

MACRO-32 Format:

```
$DS_BGNSUMMARY [<regmask>], [<psect>]

(summary routine)

$DS_ENDSUMMARY
```

BLISS-32 Format:

```
$DS_BGNSUMMARY;

(summary routine);

$DS_ENDSUMMARY;
```

regmask

List of general purpose register names to be placed in the entry mask.

psect

Any argument string that is valid for a MACRO-32 .PSECT statement. If none is specified, the argument string 'SUMMARY,LONG' will be used.

Notes:

1. In MACRO-32, the $DS_BGNSUMMARY macro will generate the following code:

```
                .SAVE
        .PSECT  psect
    SUMMARY:
        WORD ^M<regmask>                    ;ENTRY MASK
```

In MACRO-32, the $DS_ENDSUMMARY macro will generate the following code:

```
SUMMARY_X:
    $DS_BREAK
    RET
    .RESTORE
```

2. In BLISS-32, the $DS_BGNSUMMARY macro will generate the following code:

```
    PSECT CODE = SUMMARY (WRITE);
    GLOBAL ROUTINE SUMMARY : NOVALUE =
    BEGIN
```

In BLISS-32, the $DS_ENDSUMMARY macro will generate the following code:

```
    $DS_BREAK;
    END
```

$DS_BGNTEST - $DS_ENDTEST

The $DS_BGNTEST and $DS_ENDTEST macros are used to delimit each test existing in a diagnostic program. Tests are discussed in Section 5.8, Tests, Subtests, and Sections.

MACRO-32 Format:

```
$DS_BGNTEST [<section-name,section-name,...>],
[<regmask>], [align]

(test code)

$DS_ENDTEST
```

BLISS-32 Format:

```
$DS_BGNTEST ([SECTION=<section-name,section-name,...>],
[TEXT='test-name']);

(test code);

$DS_ENDTEST;
```

section-name

Name of a program section to which this test belongs. Refer to Section 3.8, Tests, Subtests, and Sections.

regmask

List of general purpose register names to be placed in the entry mask.

align

Desired alignment for the psect containing the argument lists. Possible values are BYTE, WORD, LONG, QUAD, PAGE, or an integer from 0 to 9. If an integer is specified, the psect will start at the next address that is a multiple of two raised to the power of the integer.

text

Text string identifying the test. This test will be displayed on the user terminal each time the test is executed, provided that the user has set the VDS control flag TRACE. If the (') character is to be included within the text string, it must be specified twice, as in:

```
TEXT='Fred''s test'
```

(In MACRO-32, the identifying message is defined by using the $DS_SUBTTL macro.)

Notes:

1. The $DS_BGNTEST macro will assign a test number to the test. The test number is incremented each time the $DS_BGNTEST macro is called within a source module. (The test number can be initialized when the $DS_BGNMOD macro is called at the beginning of the source module.)

2. In MACRO-32, the $DS_BGNTEST macro causes the following label to be generated:

```
TEST_xxx::        .WORD ^M< >
```

   where "xxx" is the current test number.

   In MACRO-32, the $DS_ENDTEST macro generates the following code:

```
     MOVL #1,R0  ;NORMAL EXIT
TEST_nnn_X::
     $DS_BREAK
     RET
```

3. In BLISS-32, the $DS_BGNTEST macro generates the following entry point:

```
.ENTRY   TEST_xxx,^M< >
```

   where "xxx" is the current test number.

   In BLISS-32, the $DS_ENDTEST macro generates the following code:

```
$DS_BREAK;
SS$_NORMAL
END;
```

$DS_CLI

The $DS_CLI program structure macro is used to create a parse tree. The tree can then be used to parse command strings containing commands defined by the diagnostic program (see Section 3.12.2.2, Prompting the User). Actual parsing of a command string can be performed by the $DS_PARSE system service. That service will traverse a parse tree previously constructed with the $DS_CLI macro.

A parse tree is created by using a set of $DS_CLI macros. Each time the macro is used, a node of the tree is created. Most nodes will possess the following:

- A character, string of characters, or special "traversal code" that will indicate what must be next in the input command string to constitute a legal command.
- An "action code" that will be passed to an "action routine" if there is a match between the tree node and the input command string. Action routines are detailed in the discussion of the $DS_PARSE macro.
- The address of a node to jump to if the current traversal path turns out to be the wrong one (a mismatch has been encountered).

Once the tree has been created, the $DS_PARSE system service can be used. That service will start at the root of the tree and traverse it, comparing an input command string with the characters or "traversal codes" contained in each node. Each time there is a match, the $DS_PARSE service will call the "action routine," passing to the routine the "action code" specified with the $DS_CLI macro. Then the next node in the current path will be checked. If, on the other hand, there is a mismatch, the system service will jump to the node specified as being the one to go to on a mismatch.

MACRO-32 Format:

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

BLISS-32 Format:

Not implemented for BLISS-32.

char

1. A character to be compared to the next character in the input string, or

2. A "traversal code," indicating which types of characters should be expected next in the input string. The traversal codes are defined by the $DS_CLIDEF macro. They are discussed in Note 1.

action

Code to be passed to the action routine. The action routine is called every time there is a match between the current node and the input string.

miss

Address of node to jump to if there is a mismatch at the current node.

ascii

ASCII string to be used as node content if CLI$K_STRING is used for "char" (see Note 1). See examples for proper format.

Notes:

1. The "char" parameter may either be a single ASCII character or it may be a traversal code. Its purpose is to indicate to the $DS_PARSE system service what character, characters, or types of characters should be expected next in the input string. The traversal codes are defined by the $DS_CLIDEF macro. The actions that the $DS_PARSE service will take for each traversal code are defined as follows:

- CLI$K_ALNUM - Continue reading input string as long as alphabetic or numeric characters are encountered.

- CLI$K_ALPHA - Continue reading input string as long as alphabetic characters are encountered.

- CLI$K_NUM - Continue reading input string as long as numeric characters are encountered. Numeric characters must be valid for the current default radix setting (refer to the SET DEFAULT command in the <u>VAX Diagnostic Supervisor User's Guide</u>.)

- CLI$K_SYMBOL - Continue reading input string as long as valid symbol characters are encountered. Valid symbol characters are A - Z, Ø - 9, $, and _.

- CLI$K_FILE - Continue reading input string as long as valid filename characters are encountered. (Filename characters are A-Z, Ø-9, plus the wildcard characters * and %.)

- CLI$K_SPACE - Continue reading input string as long as spaces are encountered. If no spaces exist at the current point in the input string, do not call the action routine; branch to "miss" instead.

- CLI$K_COMMA - Find next nonspace input character, and see if it is a comma. If so, find next nonspace input character, then call action routine. Otherwise branch to "miss."

- CLI$K_SLASH - Find next nonspace input character, and see if it is a slash (/). If so, find next nonspace input character, then call action routine. Otherwise branch to "miss."

- CLI$K_VALUE - Find next nonspace input character, and see if it is a : or an =. If so, find next nonspace input character, then call action routine. Otherwise branch to "miss."

- CLI$K_EOL - Find next nonspace input character, and see if it is a line terminator. If so, call action routine. Otherwise branch to "miss."

- CLI$K_DEC - Continue reading input string as long as valid decimal numeric characters are encountered.

- CLI$K_HEX - Continue reading input string as long as valid hexadecimal numeric characters are encountered.

- CLI$K_OCT - Continue reading input string as long as valid octal numeric characters are encountered.

- CLI$K_STRING - Continue reading input string as long as the input string matches the character string specified by the "ascii" parameter. The comparison is considered to be a match even if only the first character of the input string (starting at the current pointer position) matches the character string.

- CLI$K_BR - Call the action routine, then branch unconditionally to the address specified by "miss." No reading of the input string occurs.

- CLI$K_BIF - Call the action routine, then branch to address specified by "miss" if bit Ø of RØ is set. No reading of the input string occurs.

- CLI$K_CALL - Call action routine, then unconditionally branch to another parse tree. Address of tree is specified by "miss." Do not nest calls.

- CLI$K_RETURN - Call action routine, then return to original parse tree, to the $DS_CLI macro directly following the macro containing the CLI$K_CALL code. The action routine may set or clear bit Ø of rØ. The contents of RØ will then be saved for use by the CLI$K_BIFS macro.

- CLI$K_BIFS - Used after return from a subtree. Call action routine, then branch if the action routine had set bit Ø of RØ during processing of CLI$K_RETURN macro. (Contents of RØ will have already changed, but its value will have been saved during processing of CLI$K_RETURN.)

- CLI$K_EXIT - Call the action routine, then stop traversing the tree. The $DS_PARSE system service returns control to the caller, with RØ set to SS$_NORMAL. No reading of the input string occurs. This code is used to indicate that the input string has been successfully parsed.

- CLI$K_ERROR - Call the action routine, then stop traversing the tree. The $DS_PARSE system service returns control to the caller, with RØ set to DS$_ERROR. No reading of the input string occurs. This code is used to indicate an unsuccessful parse of the input string (an illegal command string was specified).

Examples:

Here is a simple but instructive example of a user-defined command language. Suppose we wanted to create a command language to represent some of the steps involved in baking a cake. Consider just the following steps:

1. Add sugar.
2. Add salt.
3. Add milk.
4. Beat ingredients.
5. Bake cake.

Figure 4-3 illustrates a parse tree for this command language.

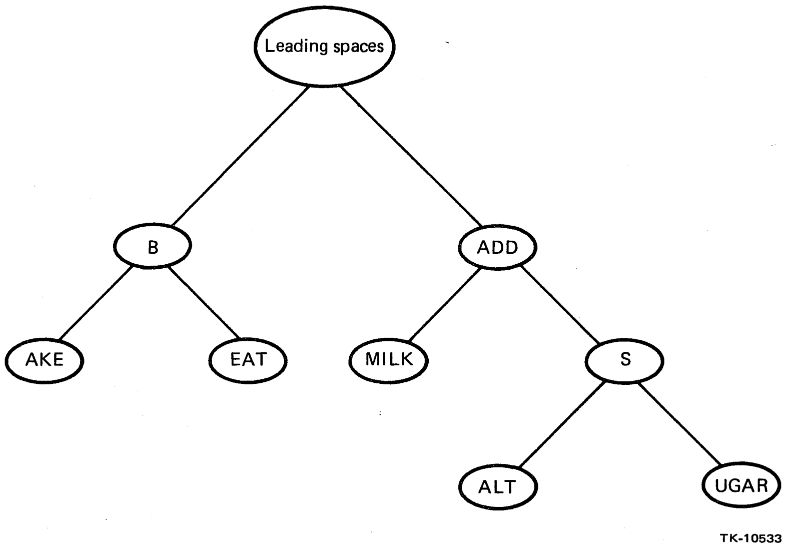

Figure 4-3 Sample Parse Tree

This tree would be described with $DS_CLI macros as follows:

```
NO_ACTION=Ø
ADD=1
BAKE=2
BEAT=3
MILK=4
SALT=5
SUGAR=6
ILLCMD=7
BADARG=8

TREE_ROOT::
        $DS_CLI CLI$K_SPACE, NO_ACTION, ADD_NODE        ;Leading spaces

ADD_NODE:
        $DS_CLI CLI$K_STRING, ADD, B_NODE, 'ADD'         ;ADD
        $DS_CLI CLI$K_SPACE, NO_ACTION, ILLCMD$          ;ADD<space>
        $DS_CLI CLI$K_STRING, MILK, S_NODE, 'MILK'       ;ADD<space>MILK
        $DS_CLI CLI$K_EOL, NO_ACTION, BADARG$            ;ADD<space>MILK<cr>
        $DS_CLI CLI$K_EXIT

B_NODE:
        $DS_CLI <^A'B'>, NO_ACTION, ILLCMD$              ;B
        $DS_CLI CLI$K_STRING, BAKE, EAT_NODE, 'AKE'      ;BAKE
        $DS_CLI CLI$K_EOL, NO_ACTION, ILLCMD$            ;BAKE<cr>
        $DS_CLI CLI$K_EXIT

EAT_NODE:
        $DS_CLI CLI$K_STRING, BEAT, ILLCMD$, 'EAT'       ;BEAT
        $DS_CLI CLI$K_EOL, NO_ACTION, ILLCMD$            ;BEAT<cr>
        $DS_CLI CLI$K_EXIT

S_NODE:
        $DS_CLI <^A'S'>, NO_ACTION, ILLCMD$              ;ADD<space>S
        $DS_CLI CLI$K_STRING, SALT, UGAR_NODE, 'ALT'     ;ADD<space>SALT
        $DS_CLI CLI$K_EOL, NO_ACTION, BADARG$            ;ADD<space>SALT<cr>
        $DS_CLI CLI$K_EXIT

UGAR_NODE:
        $DS_CLI CLI$K_STRING, SUGAR, BADARG$, 'UGAR'     ;ADD<space>SUGAR
        $DS_CLI CLI$K_EOL, NO_ACTION, BADARG$            ;ADD<space>SUGAR<cr>
        $DS_CLI CLI$K_EXIT

DONE:
        $DS_CLI CLI$K_EXIT

ILLCMD$:
        $DS_CLI CLI$K_ERROR, ILLCMD

BADARG$:
        $DS_CLI CLI$K_ERROR, BADARG
```

$DS_DEVTYP

The $DS_DEVTYP macro is used to indicate to the VDS which types of devices the diagnostic program is capable of testing.

MACRO-32 Format:

```
$DS_DEVTYP     <[string],[string],...>,
               <[address],[address],...>
```

BLISS-32 Format:

```
$DS_DEVTYP     ([STRINGS=<string,[string],...>],
               [ADDRESSES=<address,[address],...>]);
```

string

Character string representing a device type, such as 'RKØ6' or 'TMØ3'. This parameter is used to specify device types for which p-table descriptors exist in the VDS.

address

Address of a p-table descriptor defined within the diagnostic program. P-table desciptors must be defined within the diagnostic program if:

1. A p-table descriptor for the device does not exist in the VDS, or

2. The programmer wishes to override the VDS's p-table descriptor for a device. P-table descriptors are discussed in Section 3.2.2.

Examples:

MACRO-32 Examples:

```
    $DS_DEVTYP <RP04, RP05, RP06>

    $DS_DEVTYP <>,<DESCR1, DESCR2>
```

BLISS-32 Examples:

```
    $DS_DEVTYP (STRINGS=<RP04, RP05, RP06>);

    $DS_DEVTYP (ADDRESSES=<DESCR1, DESCR2>);
```

$FAB

The $FAB macro is used to allocate an RMS file access block (FAB) at program compilation time and, optionally, to load values into the various fields within the FAB. An FAB is a data structure that is required for performing file management operations using RMS. Refer to Section 3.15, File Management.

This description only discusses FAB fields supported by VDS RMS. For a discussion of VMS RMS-supported fields, refer to the VAX/VMS RMS Reference Manual.

Besides allocating the FAB, the $FAB macro also defines symbols for each FAB field. Symbols are of the form "FAB$datatype_fieldname," where "datatype" is a data type specifier listed in Table 5-1.

MACRO-32 Format:

```
$FAB    DNA=default-name-address,-
        DNM=<default-name-filespec>,-
        DNS=default-string-size,-
        FAC=fac-param,-
        FNA=filename-address,-
        FNM=<filename-filespec>,-
        FNS=filename-string-size,-
        FOP=RWO,-
        FSZ=header-size,-
        XAB=xab-addr
```

BLISS-32 Format:

```
$FAB    (DNA=default-name-address,
        DNM='default-name-filespec',
        DNS=default-string-size,
        FAC=fac-param,
        FNA=filename-address,
        FNM='filename-filespec',
        FNS=filename-string-size,
        FOP=RWO,
        FSZ=header-size,
        XAB=xab-addr);
```

Note: All parameters are optional. Refer to descriptions of the RMS run-time services to determine which fields are required for which services. Fields may be loaded at run-time with the $FAB_STORE macro, or by directly referencing FAB fields, as described in Section 3.15.4.

DNA = default-name-address

Address of a character string representing defaults to be used for the filename, if the actual filename specification is incomplete. The default string may contain all or some of the following fields:

- Node
- Device
- Device directory
- Filename
- Filename extension
- File version number

An example default string is

```
DEF_STRING:    .ASCII /.DAT/
```

The DNS field must be used in conjunction with the DNA field.

DNM = default-name-filespec

A character string representing defaults to be used for the filename, if the actual filename specification is incomplete. Using the DNM parameter is an alternative to using the DNA and DNS parameters.

A MACRO-32 example of this parameter is DNM=<.EXE;Ø>. A BLISS-32 example is DNM='.EXE;Ø'.

DNS = default-string-size

Size of the string pointed to by "default-name-address." Used only if the DNA parameter is also used.

FAC = fac-param

File access parameters. If the program is to perform $GET or $READ operations, the FAC field must be set up before the $OPEN operation is performed. Following are valid file access parameters:

- BIO - Block I/O operations ($READ) will be performed.
- BRO - Both Block I/O ($READ) and Record I/O ($GET) operations will be performed.
- GET - Record I/O operations ($GET) will be performed. This is the default.

FNA = filename-address

Address of character string representing the name of the file on which operations are to be performed. If any filename components are missing from the string, those components will be extracted from the default string specified by either the DNA or the DNM parameter. If components are still missing, they will be defaulted to the fields that would be exhibited if a SHOW LOAD user command were issued.

The FNS parameter must be used in conjunction with the FNA parameter.

FNM = filename-filespec

Character string representing the name of the file on which operations are to be performed. This parameter is an alternative to the FNA and FNS parameters, and would most likely be used in programs that always open the same file. An example in BLISS-32 would be FNM='EVABC.DAT'.

FNS = filename-string-size

Size of character string pointed to by "filename-address." The FNS parameter is used only if the FNA parameter is also used.

FOP = RWO

Rewind on open. Indicates that a magnetic tape should be rewound before a file on the tape is opened.

FSZ = header-size

Size of file's fixed control area. Used only for files containing fixed-length control records. Refer to the VAX/VMS RMS Reference Manual for details. It is unlikely that a diagnostic program will make use of this field.

XAB = xab-addr

Address of the FHC XAB, if used. (The FHC XAB is declared with the $XABFHC macro.)

Notes:

1. Read-Only FAB Fields

   The following FAB fields are not loaded by the programmer under VDS RMS. They are filled in by RMS services, and may be read after the service has completed. (Some of these fields are read/write in VMS RMS.)

   - BID - Block identifier field. Indicates to RMS that a block is an FAB.

   - BLN - Block length field. Defines the length of the FAB.

   - DEV - Device characteristics field. A bitmap indicating various characteristics of the device on which the file being referenced resides. Following is a list of bits supported by VDS RMS:

     - DIR - Directory-structured device.

     - FOD - File-oriented device (disk and magnetic tape).

     - RND - Random access device.

     - SDI - Single directory device (master file directory only).

     - SQD - Sequential block-oriented device (magnetic tape).

- IFI - Internal file identifier field. Used to associate the FAB with an internal access block.

- MRS - Maximum record size.

- ORG - File organization. Valid values for this field are:

  - REL - Relative file organization.

  - IDX - Indexed file organization.

  - SEQ - Sequential file organization.

  Note: VDS RMS only supports operations on files having sequential organization.

- RAT - Record attributes. Indicates that special control information has been attached to the records of a file. Refer to the VAX/VMS RMS Reference Manual for a discussion of record attributes. It is unlikely that a diagnostic program will make use of this field.

- RFM - Record format. Indicates the format of the records in the file. Possible values for this field are:

  - FIX - (FAB$C_FIX) Fixed length record format.

  - VFC - (FAB$C_VFC) Variable length with fixed length control record format.

  - VAR - (FAB$C_VAR) Variable length record format.

  - UDF - (FAB$C_UDF) Undefined record format.

  - If the file is on the console medium (RT-11 format), the RFM code returned by the $OPEN service will be 4. There is no symbolic repesentation for this value.

- STS - Completion status code field. RMS services load this field with a success or failure completion status before returning to the caller of the service. The completion status code is also passed to the caller in RØ.
- STV - Status value field. Sometimes used to pass additional status information from a service to the caller.

2. Table 4-1 lists all of the FAB fields.

Table 4-1 FAB Fields

Field and Keyword Name	Field Size	Description	Offset
ALQ	Longword	Allocation quantity	FAB$L_ALQ
BID	Byte	Block identifier	FAB$B_BID
BKS	Byte	Bucket size	FAB$B_BKS
BLN	Byte	Block length	FAB$B_BLN
BLS	Word	Block size	FAB$W_BLS
CTX	Longword	Context	FAB$L_CTX
DEQ	Word	Default file extension quantity	FAB$W_DEQ
DEV	Longword	Device characteristics	FAB$L_DEV
DNA	Longword	Default file specification string address	FAB$L_DNA
DNS	Byte	Default file specification string size	FAB$B_DNS
FAC	Byte	File access	FAB$B_FAC
FNA	Longword	File specification string addr.	FAB$L_FNA
FNS	Byte	File specification string size	FAB$B_FNS
FOP	Longword	File-processing options	FAB$L_FOP
FSZ	Byte	Fixed control area size	FAB$B_FSZ
IFI	Word	Internal file identifier	FAB$W_IFI
MRN	Longword	Name block address	FAB$L_MRN
MRS	Word	Maximum record size	FAB$W_MRS
NAM	Longword	Name block address	FAB$L_NAM
ORG	Byte	File organization	FAB$B_ORG
RAT	Byte	Record attributes	FAB$B_RAT
RFM	Byte	Record format	FAB$B_RFM
RTV	Byte	Retrieval window size	FAB$B_RTV
SDC	Longword	Spooling device characteristics	FAB$L_SDC
SHR	Byte	File sharing	FAB$B_SHR
STS	Longword	Completion status code	FAB$L_STS
STV	Longword	Status values	FAB$L_STV
XAB	Longword	Extend attribute block address	FAB$L_XAB

Examples:

MACRO-32 Example:

```
FAB_BLOCK:        $FAB     DNM=<.EXE>, -
                           FAC=BIO, -
                           FNA=FILE_NAME, -
                           FNS=FILE_NAME_SIZE
```

BLISS-32 Example:

```
OWN

    FAB_BLOCK : $FAB (FAC=GET, -
                     FNM='EVXYZ.DAT');
```

$FAB_INIT

$FAB_STORE

The $FAB_STORE and $FAB_INIT macros can be used to load FAB fields at run time. The $FAB_STORE macro is used for MACRO-32 programs. The $FAB_INIT macro is used in BLISS-32 programs. Refer to the discussion of the $FAB macro for a description of FAB fields.

MACRO-32 Format:

```
$FAB_STORE      DNA=default-name-address,-
                DNM=<default-name-filespec>,-
                DNS=default-string-size,-
                FAC=fac-param,-
                FNA=filename-address,-
                FNM=<filename-filespec>,-
                FNS=filename-string-size,-
                FOP=RWO,-
                FSZ=header-size,-
                XAB=xab-addr
```

BLISS-32 Format:

```
$FAB_INIT       (DNA=default-name-address,
                DNM='default-name-filespec',
                DNS=default-string-size,
                FAC=fac-param,
                FNA=filename-address,
                FNM='filename-filespec',
                FNS=filename-string-size,
                FOP=RWO,
                FSZ=header-size,
                XAB=xab-addr);
```

Refer to the discussion of the $FAB macro for descriptions of input parameters. All parameters are optional.

Examples:

MACRO-32 Example:

```
$FAB_STORE     FNM=<FILE1.DAT>, -
               XAB=XABFHC_ADDR
```

BLISS-32 Example:

```
$FAB_INIT      (FNM='FILE1.DAT',
                FOP=RWO);
```

$DS_DISPATCH

The $DS_DISPATCH macro generates the diagnostic program "dispatch table." This table contains the starting addresses of all the tests. (These addresses are placed in the table by the linker.) The VDS uses the table when dispatching control to the tests.

MACRO-32 Format:

```
$DS_DISPATCH
```

BLISS-32 Format:

```
$DS_DISPATCH;
```

Notes:

1. In BLISS-32 programs, the $DS_DISPATCH macro must be placed before the $DS_HEADER macro. (Refer to the template in Appendix A.)

Examples:

MACRO-32 Example:

```
$DS_DISPATCH
```

BLISS-32 Example:

```
$DS_DISPATCH;
```

$DS_HEADER

The $DS_HEADER macro generates the diagnostic program header. The header must be situated so that its starting address is virtual 512 (200 hexadecimal). (The diagnostic program may not use address space below the header.)

MACRO-32 Format:

```
$DS_HEADER <pname>, rev, [update], [nunit]
```

BLISS-32 Format:

```
$DS_HEADER (PNAME='pname', REV=rev, [UPDATE=update],
[NUNIT=nunit]);
```

pname

Character string representing the program's name. This string is displayed on the user's terminal when the program is started. Note: In BLISS-32, if a (') character is to be included in the string, it must be included twice, as in PNAME='MARY''S PROGRAM'.

The string should contain the following information:

- The program's name (EVKAC, EVRAD, and so on)
- The program's level (2, 2R, or 3)
- The type of program (logic test, function test, or exerciser; see Chapter 1)
- The types of devices that the program can test

Refer to the examples below.

rev

Numeric value representing the program revision level.

update

Numeric value representing the program patch level. The default is 0.

nunit

Numeric value representing the maximum number of device units that can be tested by the program. The default is Ø.

Notes:

1. Refer to the templates in Appendix A to determine the exact location of the $DS_HEADER macro in relation to other macros appearing in the program. The arrangement of macros depends on whether the program is written in MACRO-32 or BLISS-32.

Examples:

MACRO-32 Example:

```
$DS_HEADER -
    PNAME = <EVXYZ - LEVEL 3 LOGIC TEST FOR XXYY DISK CONTROLLER>, -
    REV = 1, -
    NUNIT = 8
```

BLISS-32 Example:

```
$DS_HEADER
    (PNAME = 'EVZYX - LEVEL 2R FUNCTION TEST FOR YYZZ TAPE DRIVE',
     REV = 1, -
     NUMINT = 16);
```

$DS_PAGE

The $DS_PAGE macro is used in conjunction with the $DS_SBTTL macro. If the $DS_PAGE macro with a nonzero argument is placed immediately before the $DS_SBTTL macro, the following actions will take place:

1. Printing of the $DS_SBTTL call in the assembly listing will be suppressed, but the expansion of the $DS_SBTTL macro will be printed.

2. The subtitle will appear at the top of a new page.

The result of these actions is that the .SBTTL statement accompanying text generated by the $DS_SBTTL macro will appear at the top of the next page in the assembly listing.

MACRO-32 Format:

```
$DS_SBTTL num
```

BLISS-32 Format:

Not supported for BLISS-32.

num
: Flag indicating whether or not the subtitle generated by the $DS_SBTTL macro should appear on a new page. If this value is Ø, the subtitle will appear on the current page, and printing of the $DS_SBTTL macro call will be suppressed. If the value is nonzero, a new page will be started. The subtitle will appear at the top of the new page, and printing of the $DS_SBTTL macro call will be suppressed.

Example:

```
$DS_PAGE 1
$DS_SBTTL <READ/WRITE TESTS>
```

$RAB

The $RAB macro is used to allocate an RMS record access block (RAB) at program compilation time and, optionally, to load values into the various fields within the RAB. An RAB is a data structure that is required for performing file management operations using RMS. Refer to Section 3.15, File Management.

This description only discusses RAB fields supported by VDS RMS. For a discussion of VMS RMS-supported fields, refer to the VAX/VMS RMS Reference Manual.

Besides allocating the RAB, the $RAB macro also defines symbols for each RAB field. Symbols are of the form "RAB$datatype_fieldname," where "datatype" is a data type specifier listed in Table 5-1.

MACRO-32 Format:

```
$RAB     BKT=bkt-code,-
         FAB=fab-address,-
         RAC=rac-param,-
         RHB=header-buffer-address,-
         ROP=BIO,-
         UBF=user-buffer-address,-
         USZ=user-buffer-size
```

BLISS-32 Format:

```
$RAB     (BKT=bkt-code,
         FAB=fab-address,
         RAC=rac-param,
         RHB=header-buffer-address,
         ROP=BIO,
         UBF=user-buffer-address,
         USZ=user-buffer-size);
```

BKT = bkt-code

Bucket code. Used only with block I/O. Should be loaded with the number of the first virtual block that is to be read by the $READ service. If 0 is specified, reading will begin at block 0 for the first $READ, or at the block pointed to by the internal "next block pointer" for subsequent $READs.

FAB = fab-address

Address of the FAB describing the file to be accessed.

RAC = rac-param

Record access mode. Indicates the type of access to be used in retrieving records from the file. Valid values are

1. SEQ - Sequential record access. This is the default.

2. RFA - Random access by record's file address (RFA).

Refer to Section 5.15.6, Record Processing, and Note 2 below.

RHB = header-buffer-address

Address of buffer to store record header buffer. Used only for files consisting of variable records with fixed-length control. The $GET service will load the record's header into the specified buffer. The size of this buffer must match the size specified by the FSZ field of the FAB.

ROP = BIO

Block I/O. Only meaningful if BRO was set in the FOP field of the FAB before $OPEN was issued. If so, then setting the BIO record processing option will enable record processing and block processing to be mixed.

UBF = user-buffer-address

Address of a buffer to receive record fetched by $GET or block fetched by $READ. Buffer size is specified with USZ.

USZ = user-buffer-size

Size (number of bytes) of buffer pointed to by UBF field.

Notes:

1. Read-Only RAB Fields

   The following RAB fields are not loaded by the programmer under VDS RMS. They are filled in by RMS services, and may be read after the service has completed. (Some of these fields are read/write in VMS RMS.)

   - BID - Block identifier field. Identifies the block as a RAB.

   - BLN - Block length field. Contains the length of the RAB.

- ISI - Internal stream identifier. Associates the RAB with an FAB.
- RBF - Contains the address of the last record read.
- RFA - Record's file address. File address of last record read. See Note 2.
- RSZ - Length, in bytes, of the last record read.
- STS - Completion status code field. RMS services load this field with a success or failure completion status before returning to the caller of the service. The completion status code is also passed to the caller in RØ.
- STV - Status value field. Sometimes used to pass additional status information from a service to the caller.

2. Record's File Address (RFA)

   After a successful $GET operation, the file address of the record read into memory is stored in the RFA field. The program can extract this field and store it elsewhere in memory. Then if it is later necessary to re-read the record, the program returns the extracted address to the RFA, sets the record access mode to random-by-RFA (by setting RFA in RAC), and issues another $GET.

   The RFA field is six bytes long. There are two ways to reference the field:

   1. RAB$W_RFA is the field's offset into the RAB. RAB$S_RFA is the field's size. Thus the field may be copied as follows:

```
MOVAL   RABBLK, RØ
MOVC3   #RAB$S_RFA, RAB$W_RFA(RØ), SAVE_RFA
```

   2. RAB$LRFAØ is the offset of the first longword of the six-byte field. RAB$WRFA4 is the offset of the last word of the field. Thus the field may be copied as follows:

```
MOVAL   RABBLK, RØ
MOVL    RAB$L_RFAØ(RØ), SAVE_RFA
MOVW    RAB$W_RFA4(RØ), SAVE_RFA+4
```

3. Table 4-2 lists all of the RAB fields.

Table 4-2 RAB Fields

Field and Keyword Name	Field Size	Description	Offset
BID	Byte	Block identifier	RAB$B_BID
BKT	Longword	Bucket code	RAB$L_BKT
BLN	Byte	Block length	RAB$B_BLN
CTX	Longword	Context	RAB$L_CTX
FAB	Longword	File access block address	RAB$L_FAB
ISI	Word	Internal stream identifier	RAB$W_ISI
KBF	Longword	Key buffer address	RAB$L_KBF
KRF	Byte	Key of reference	RAB$B_KRF
MBC	Byte	Multiblock count	RAB$B_MBC
MBF	Byte	Multibuffer count	RAB$B_MBF
PBF	Longword	Prompt buffer address	RAB$L_PBF
PSZ	Byte	Prompt buffer size	RAB$B_PSZ
RAC	Byte	Record access mode	RAB$B_RAC
RBF	Longword	Record address	RAB$L_RBF
RFA	3 words	Record's file address	RAB$W_RFA
RHB	Longword	Record header buffer	RAB$L_RHB
ROP	Longword	Record-processing options	RAB$L_ROP
RSZ	Word	Record size	RAB$W_RSZ
STS	Longword	Completion status code	RAB$L_STS
STV	Longword	Status value	RAB$L_STV
STVØ	Word	Low-order word of status value	RAB$W_STVØ
STV2	Word	High-order word of status value	RAB$W_STV2
TMO	Byte	Timeout period	RAB$B_TMO
UBF	Longword	User record area address	RAB$L_UBF
USZ	Word	User record area size	RAB$W_USZ

Examples:

MACRO-32 Example:

```
BUFFER: .BLKB 50
BUF_SIZE = . - BUFFER
FAB_BLOCK:
  $FAB  FNM=<INFILE.DAT>
RAB_BLOCK:
  $RAB  FAB=FAB_BLOCK, -
        RAC=SEQ, -
        UBF=BUFFER, -
        USZ=BUF_SIZE
```

BLISS-32 Example:

```
LITERAL
  BUF_SIZE = 50;

OWN
  BUFFER    : VECTOR [BUF_SIZE, BYTE],
  FAB_BLOCK : $FAB      (FNM='FILE1.DAT'),
  RAB_BLOCK : $RAB      (FAB=FAB_BLOCK,
                         RAC=SEQ,
                         UBF=BUFFER,
                         USZ=BUF_SIZE);
```

$RAB_INIT

$RAB_STORE

The $RAB_STORE and $RAB_INIT macros can be used to load RAB fields at run time. The $RAB_STORE macro is used for MACRO-32 programs. The $RAB_INIT macro is used in BLISS-32 programs. Refer to the discussion of the $RAB macro for a description of RAB fields.

MACRO-32 Format:

```
$RAB_STORE     BKT=bkt-code,-
               FAB=fab-address,-
               RAC=rac-param,-
               RHB=header-buffer-address,-
               ROP=BIO,-
               UBF=user-buffer-address,-
               USZ=user-buffer-size
```

BLISS-32 Format:

```
$RAB_INIT      (BKT=bkt-code,
               FAB=fab-address,
               RAC=rac-param,
               RHB=header-buffer-address,
               ROP=BIO,
               UBF=user-buffer-address,
               USZ=user-buffer-size);
```

Refer to the discussion of the $RAB macro for descriptions of input parameters. All parameters are optional.

Examples:

MACRO-32 Example:

```
                     BUF_SIZE = 50
  IN_BUF: .BLKB BUF_SIZE

          $RAB_STORE      UBF=IN_BUF, -
                          UBZ=#BUF_SIZE
```

BLISS-32 Example:

```
  LOCAL
          INBUF : VECTOR [50, BYTE];

          $RAB_INIT       (UBF=INBUF, UBZ=BUF_SIZE);
```

$DS_SBTTL

The $DS_SBTTL macro should be used at the beginning of each test and subtest. It will perform the following functions:

- It will generate text containing the test and subtest numbers, along with the contents of a programmer-specified character string. This text will be included in a .SBTTL MACRO-32 statememt, and will also be displayed on the user terminal when the test or subtest is entered and the VDS Control Flag TRACE is set.
- If the macro is at the beginning of a test, a new program section (.PSECT) is assigned to the test. (A subtest will be included in the PSECT of the test to which it belongs.)
- The code of the test or subtest will be aligned as specified by the programmer.

MACRO-32 Format:

```
$DS_SBTTL ascii, [align]
```

BLISS-32 Format:

Not supported for BLISS-32.

ascii

Character string representing text to be used as program subtitle and to be displayed when VDS TRACE flag is set.

align

Desired program section alignment for the test or subtest. Possible values are BYTE, WORD, LONG, QUAD, PAGE, or an integer from Ø to 9. If an integer is specified, the psect will start at the next address that is a multiple of two raised to the power of the integer.

Notes:

1. The $DS_SBTTL macro should be used in conjunction with the $DS_PAGE macro.

Example:

```
$DS_SBTTL -
        ALIGN = BYTE, -
        ASCII = <READ/WRITE SWAP DATA TEST>
```

$DS_SECDEF

The $DS_SECDEF macro is used to declare all of the names of the test sections (see Section 3.8.3) of the diagnostic program. This macro must appear in every source module that contains tests. The macro is used in conjunction with the $DS_SECTION macro.

MACRO-32 Format:

```
$DS_SECDEF a, [b, c, d, e, f, g, h, i, j, k, l, m, n, o, p]
```

BLISS-32 Format:

```
$DS_SECDEF (a, [b, c, d, e, f, g, h, i, j, k, l, m, n, o, p]);
```

a, b, ..., o, p

List of from 1 to 16 test section names. This list must be identical to the list included with the $DS_SECTION macro, even if the module in which the $DS_SECDEF macro is being placed does not include tests belonging to every listed section.

Notes:

1. The macro automatically includes the section name DEFAULT at the beginning of the section name list.

Examples:

MACRO-32 Example:

```
$DS_SECDEF READTESTS, WRITETESTS, SEEKTESTS
```

BLISS-32 Example:

```
$DS_SECDEF (READTESTS, WRITETESTS, SEEKTESTS);
```

$DS_SECTION

The $DS_SECTION macro is used to declare all of the names of the test sections (see Section 3.8.3) of the diagnostic program. This macro must appear in the source module that contains the $DS_HEADER macro. The $DS_SECTION macro is used in conjunction with the $DS_SECDEF macro.

MACRO-32 Format:

```
$DS_SECTION a, [b, c, d, e, f, g, h, i, j, k, l,
m, n, o, p]
```

BLISS-32 Format:

```
$DS_SECTION (a, [b, c, d, e, f, g, h, i, j, k, l,
m, n, o, p]);
```

a, b,...,o, p

List of from 1 to 16 test section names. This list must be identical to the list included with the $DS_SECDEF macro.

Notes:

1. The macro automatically includes the section name DEFAULT at the beginning of the section name list.

Examples:

MACRO-32 Example:

```
$DS_SECTION READTESTS, WRITETESTS, SEEKTESTS
```

BLISS-32 Example:

```
$DS_SECTION (READTESTS, WRITETESTS, SEEKTESTS);
```

$DS_STRING

The $DS_STRING macro can be used to generate a quadword descriptor (see section 4.3) for a given character string. In MACRO-32 programs, .ASCIC and .ASCIZ formats for the string may also be generated. This enables the programmer to reference the same string in any of the three formats.

MACRO-32 Format:

```
$DS_STRING       <text>, [label1], [label2]
```

BLISS-32 Format:

```
$DS_STRING      ('text');
```

text

Character string for which a quadword descriptor is to be constructed.

label1

Label to be placed at the .ASCIC construction of the character string. (This parameter may not be referenced by keyword.)

label2

Label to be placed at the .ASCIZ construction of the character string. (This parameter may not be referenced by keyword.)

Notes:

1. The quadword descriptor will be contructed at the current PC. It may be accessed by placing a label at the macro call, as illustrated in the example.

Examples:

MACRO-32 Example:

```
MSG_LABEL:
   $DS_STRING -                     ;Create descriptor for string.
           <THIS IS A MESSAGE.>, -  ;
           MSG_LABEL1, -               ;Include label for .ASCIC
           MSG_LABEL2                  ;Include label for .ASCIZ
```

BLISS-32 Example:

```
BIND
        MSG_LABEL = $DS_STRING (THIS IS A MESSAGE.);
```

$XABFHC

The $XABFHC will allocate the File Header Characteristics Extended Attribute Block (FHC XAB), which is an optional data structure used by RMS. If the $XABFHC macro is used, and if a pointer to the FHC XAB is specified in the FAB, then the $OPEN operation will load the FHC XAB with file header characteristics obtained from the header of the file that was opened.

Besides allocating the XAB, the $XAB macro also defines symbols for each XAB field. Symbols are of the form XAB$datatype_fieldname, where "datatype" is a data type specifier listed in Table 5-1.

MACRO-32 Format:

```
$XABFHC
```

BLISS-32 Format:

```
$XABFHC;
```

Notes:

1. FHC XAB Fields

   Following are the FHC XAB fields filled in by VDS RMS. Refer to the <u>VAX/VMS RMS Reference Manual</u> for fields filled in by VMS RMS.

   - ATR - Record attributes. Same as RAT field of FAB.
   - BLN - Length of the XAB.
   - COD - Type of XAB. (Only FHC XAB type is allowed.)
   - EBK - Virtual block number of end-of-file.
   - FFB - First free byte in end-of-file block.
   - HSZ - Fixed length control header size. Same as FSZ field of FAB.
   - LRL - Longest record length.

- MRZ - Maximum record size. Same as MRS field of FAB.
- RFO - File organization and record format. Combines ORG and RFM fields of FAB.
- SBN - Starting block number of the file if it is contiguous; otherwise field is Ø.

Examples:

MACRO-32 Example:

```
XAB_BLOCK:      $XABFHC
```

BLISS-32 Example:

```
LOCAL
    XAB_BLOCK :  $XABFHC;
```

$DS_ERRNUM

The $DS_ERRNUM macro is used in conjunction with the $DS_ERRxxxx_L macros. It generates executable code that will dynamically load the "num" argument of the argument list created by the $DS_ERRxxxx_L macro.

MACRO-32 Format:

```
$DS_ERRNUM      label, [num]
```

BLISS-32 Format:

Not supported for BLISS-32.

label

Address of the argument list generated by the $DS_ERxxxx_L macro.

num

Error number. If a value is specified, the value will be used as the "num" parameter in the argument list. If a value is not specified, the current assembly-time error number is used. Refer to the description of the $DS_ERRxxxx system services for an explanation of the assignment of error numbers at assembly time.

Notes:

1. Using the $DS_ERRxxxx_L macro to create an argument list, dynamically altering the error number with the $DS_ERRNUM macro, then calling the error service with a $DS_ERRxxxx_G call has a disadvantage. It is difficult to relate a specific error message, displayed at run-time, to a specific point in the program listing because the error number is not explicitly specified as a macro argument. This may or may not be a problem, depending on the program's use and users.

Example:

```
ARG_LIST:
        $DS_ERRHARD_L -                 ;Declare hard error ars. list
                UNIT = LOG_UNIT, -
                MSGADR = HARD_MSG1, -
                PRLINK = HARD_RTN1, -
                P1 = CSR_REG
            .
            .
            .
        $DS_ERRNUM ARG_LIST             ;Put error number in ars. list
```

4.5 PROGRAM CONTROL MACROS

$DS_BCOMPLETE - $DS_BNCOMPLETE

The $DS_BCOMPLETE and $DS_BNCOMPLETE program control macros can be used to test the return status of a system service (or any routine which returns a status code in RØ) and branch if the service's operation was "complete" or "incomplete."

MACRO-32 Format:

```
$DS_BCOMPLETE adr
$DS_BNCOMPLETE adr
```

BLISS-32 Format:

Not supported for BLISS-32, since testing RØ is implicit in the language. See the example below.

adr

Address to which to branch if tested condition is satisfied.

Notes:

1. For all error status codes, bit Ø is clear. Therefore, these macros simply generate the following code:

```
$DS_BCOMPLETE -     BLBS RØ,adr
$DS_BNCOMPLETE -    BLBC RØ,adr
```

2. If an error status code is detected, the contents of RØ should be compared with all error codes that could possibly be returned from the service (or other) routine to determine the exact nature of the error.

Examples:

MACRO-32 Example:

```
$DS_GETBUF       #2, RETADDR, PHYSADDR
$DS_BNCOMPLETE   BAD_BUF
```

BLISS-32 Example:

```
IF $DS_GETBUF (PAGCNT=2) THEN ...
```

$DS_BERROR - $DS_BNERROR

The $DS_BERROR and $DS_BNERROR program control macros can be used to test the return status of a system service (or any routine which returns a status code in RØ) and branch if the service's operation was in error or was error-free.

MACRO-32 Format:

```
$DS_BERROR adr

$DS_BNERROR adr
```

BLISS-32 Format:

Not supported for BLISS-32, since testing RØ is implicit in the language. See the example below.

adr

Address to which to branch if tested condition is satisfied.

Notes:

1. For all error status codes, bit Ø is clear. Therefore, these macros simply generate the following code:

   ```
   $DS_BERROR -     BLBC RØ,adr

   $DS_BNERROR -    BLBS RØ,adr
   ```

2. If an error status code is detected, the contents of RØ should be compared with all error codes that could possibly be returned from the service (or other) routine to determine the exact nature of the error.

Examples:

MACRO-32 Example:

```
$DS_GPHARD      LOG_UNIT, ADDR1

$DS_BNERROR    10$
```

BLISS-32 Example:

```
IF NOT $DS_GPHARD (UNIT=.LOG_UNIT, RETADR=ADDR1) THEN ...
```

$DS_BOPER - $DS_BNOPER

The $DS_BOPER and $DS_BNOPER macros can be used to determine the presence of an operator (user) during program execution. (The presence of a user is indicated by the condition of the VDS control flag OPERATOR.) These macros can be used to control whether certain portions of the program are executed only if a user is present. $DS_BOPER will cause a branch if the OPERATOR flag is set, and $DS_BNOPER will cause a branch if the flag is clear.

MACRO-32 Format:

```
$DS_BOPER adr

$DS_BNOPER adr
```

BLISS-32 Format:

Not implemented for BLISS-32. Direct reference of the corresponding VDS control flag, as illustrated in the example below, is recommended.

adr

Address to which to branch if the tested condition is satisfied.

Examples:

MACRO-32 Example:

```
$DS_BNOPER 100$
```

BLISS-32 Example:

```
IF .DSA$V_OPER THEN BEGIN ... END;
```

$DS_BPASS0 - $DS_BNPASS0

The $DS_BPASSØ and $DS_BNPASSØ program control macros can be used within the initialization code to determine if the current pass through the initialization code is the first one. It is often necessary to perform certain operations the first time the initialization code is executed that should not be repeated on subsequent passes through the initialization code, such as initialization of run-time variables. (It is helpful to think of "pass Ø" as the execution that takes place before the first pass through the tests occurs.)

$DS_BPASSØ will cause a branch if the current pass through the initialization code is the first one. $DS_BNPASSØ will cause a branch if the current pass through the initialization code is not the first one. These macros may only be used in the initialization code.

MACRO-32 Format:

```
$DS_BPASSØ adr

$DS_BNPASSØ adr
```

BLISS-32 Format:

Not implemented for BLISS-32. Direct reference of the corresponding VDS control flag, as illustrated in the example below, is recommended.

adr

Address to which to branch if the tested condition is satisfied.

Examples:

MACRO-32 Examples:

```
$DS_BNPASS0    50$
$DS_BPASS0    PASS1
```

BLISS-32 Example:

```
IF .DSA$V_PASS0 THEN BEGIN ... END;
```

$DS_BQUICK - $DS_BNQUICK

The $DS_BQUICK and $DS_BNQUICK program control macros can be used to determine if the VDS control flag QUICK has been set by the program user. The $DS_BQUICK will cause a branch if the QUICK flag is set, and the $DS_BNQUICK will cause a branch if the flag is clear. If the flag has been set, the diagnostic program should execute only the portions of code deemed appropriate to the "quick" mode of operation.

MACRO-32 Format:

```
$DS_BQUICK adr

$DS_BNQUICK adr
```

BLISS-32 Format:

Not implemented for BLISS-32. Direct reference of the corresponding VDS control flag, as illustrated in the example below, is recommended.

adr

Address to which to branch if the tested condition is satisfied.

Examples:

MACRO-32 Examples:

```
$DS_BQUICK TAG1

$DS_BNQUICK 100$
```

BLISS-32 Example:

```
IF .DSA$V_QUICK THEN BEGIN ... END;
```

$DS_CKLOOP

The $DS_CKLOOP program control macro is used to explicitly specify the upper bound of a program loop. It is used when the implicit upper bound provided by a $DS_ENDSUB or $DS_ENDTEST macro creates a loop that is too large to be useful. A detailed discussion of program looping, including the use of the $DS_CKLOOP macro, is provided in Section 3.10, Looping.

MACRO-32 Format:

```
$DS_CKLOOP label
```

BLISS-32 Format:

Not supported for BLISS-32. See note 2.

label

Address of loop's lower bound. After the $DS_CKLOOP is executed, program flow branches to this address. The address must be lower than the location of the $DS_CKLOOP macro, but higher than the most recent $DS_BGNTEST or $DS_BGNSUB macro.

Notes:

1. If $DS_CKLOOP macros are used in a test that does not contain subtests, the $DS_CKLOOP macros may be placed anywhere within the test. For tests that contain subtests, the $DS_CKLOOP macros must be placed within the subtests.

2. The $DS_CKLOOP has not been implemented for BLISS-32. However, programs written in BLISS-32 (and MACRO-32, for that matter) can define sufficiently small program loops with judicious use of $DS_BGNSUB and $DS_ENDSUB macros.

3. The $DS_INLOOP system service may be used inside the bounds of a loop to determine whether or not the loop is actually being executed.

Example:

```
        $DS_BGNSUB
             .
             .
   LOOP_BGN:
             .
        $DS_ERRHARD       UNIT=LOG_UNIT, MSGADR=HRD1, PRLINK=HRDRTN1
             .
             .
        $DS_CKLOOP        LOOP_BGN
             .
             .
        $DS_ENDSUB
```

$DS_ESCAPE

The $DS_ESCAPE program control macro can be used to exit from a test or subtest if a hardware failure has been detected from within the test or subtest. If the failure is reported using one of the error reporting macros ($DS_ERRxxxx), and if $DS_ESCAPE is executed before the next $DS_ENDSUB or $DS_ENDTEST macro is encountered, then program control will branch to the next $DS_ENDSUB or $DS_ENDTEST (whichever is specified).

MACRO-32 Format:

```
$DS_ESCAPE arg
```

BLISS-32 Format:

Not supported for BLISS-32. See Note 1.

arg

Indicates whether program control should branch to nearest $DS_ENDSUB or nearest $DS_ENDTEST. The argument may be either SUB or TEST.

Notes:

1. For programs written in BLISS-32, similar functionalilty can be obtained by following the $DS_ERRxxx macro with a LEAVE statement, as shown in the example below.

Examples:

MACRO-32 Example:

```
    $DS_BGNSUB
          :
          :
    $DS_ERRHARD      UNIT=LOG_UNIT, MSGADR=HRDMSG3, PRLINK=HRDRTN3
    $DS_ESCAPE       SUB
          :
          :
    $DS_ENDSUB
```

BLISS-32 Example:

```
    $DS_BGNSUB;
SUB3:       BEGIN
          .
          .
    $DS_ERRHARD_S (UNIT=.LOG_UNIT, MSGADR=HRDMSG3, PRLINK=HRDRTN3);
    LEAVE SUB3;
          .
          .
    END;
    $DS_ENDSUB;
```

$DS_EXIT

The $DS_EXIT program control macro is used to unconditionally branch to the end of the currently executing program segment. Exits can be made from any of the following:

1. A test
2. A subtest
3. An interrupt service routine
4. The summary routine

MACRO-32 Format:

```
$DS_EXIT arg
```

BLISS-32 Format:

Not supported for BLISS-32. See note 1.

arg

Indicates program segment type. Valid arguments are TEST, SUB, SERV, and SUMMARY.

Notes:

1. For programs written in BLISS-32, similar functionalilty can be obtained by using the LEAVE statement, as shown in the example below.

Examples:

MACRO-32 Example:

```
$DS_BGNSERV   SERV_RTN
     :
     :
     :
$DS_EXIT      SERV
     :
     :
 $DS_ENDSERV
```

BLISS-32 Example:

```
            $DS_BGNTEST;
    T2_BLK1:
            BEGIN
               :
               :
               :
            LEAVE T2_BLK1;
               :
               :
            END;
            $DS_ENDTEST;
```

4.6 SYSTEM SERVICE MACROS

$DS_ABORT

The Abort Program or Test service can be used to stop execution of either the whole diagnostic program or just the current test. If the program is aborted, a system service is called. This service will execute the program's cleanup code and return control to the VDS command line interpreter. If only the current test is aborted, the test is exited (with an RET instruction) and the next selected test is called.

MACRO-32 Format:

```
$DS_ABORT arg
```

(No suffix.)

BLISS-32 Format:

```
$DS_ABORT (ARG=arg);
```

arg

'PROGRAM' or 'TEST'. If 'PROGRAM' is specified, then the program will be aborted. If 'TEST' is specified, the current test will be exited (with an RET instruction) and the next selected test will be called.

Return Status:

No status is returned, because $DS_ABORT (TEST) does not generate a service call and $DS_ABORT (PROGRAM) does not allow program control to return to the diagnostic program.

Examples:

MACRO-32 Example:

```
$DS_ABORT (PROGRAM)

$DS_ABORT (TEST)
```

BLISS-32 Example:

```
$DS_ABORT (ARG=PROGRAM);

$DS_ABORT (ARG=TEST);
```

$ASCTIM

The Convert Binary Time to ASCII String system service converts the contents of a quadword from 64-bit time format into an ASCII string. This is the converse of the function performed by the $BINTIM service.

MACRO-32 Format:

```
$ASCTIM_x [timlen], timbuf, [timadr], [cvtflg]
```

BLISS-32 Format:

```
$ASCTIM ([TIMLEN=timlen], TIMBUF=timbuf, [TIMADR=timadr],
[CVTFLG=cvtflg]);
```

timlen

Address of a word to receive length of output string.

timbuf

Address of a character string descriptor (see Section 4.3) pointing to buffer to receive converted string.

timadr

Address of the 64-bit time value to be converted. A value of Ø (the default) results in the current system time being converted. A positive value represents an absolute time. A negative value represents a relative time (offset from the current time).

cvtflg

Conversion indicator. A value of 1 causes only the hour, minute, second, and hundredth of second fields to be returned, while a value of Ø causes the full date and time to be returned.

Return Status:

SS$_NORMAL

Service successfully completed.

SS$_IVTIME

The specified relative time is equal to or greater than 10,000 days.

Notes:

1. The ASCII string returned by the service will be in the format specified in the notes to the $BINTIM service.

2. To receive full absolute date and time, the "timbuf" buffer length must be 23 bytes. To receive the full relative day and time, the buffer length must be 16 bytes. Specifying a shorter buffer length will cause the returned string to be truncated to the buffer size. This may be useful if, say, only the absolute date is required, and not the time. It is only necessary to provide a buffer that can hold the amount of the returned string the caller wishes to receive.

Examples:

MACRO-32 Example:

```
$ASCTIM_S    STR_LENGTH, BUFPTR, TIME, #1
```

BLISS-32 Example:

```
$ASCTIM (TIMLEN=STR_LENGTH, TIMBUF=BUFPTR, TIMADR=TIME,
CVTFLG=1);
```

$DS_ASKADR

$DS_ASKDATA

$DS_ASKLGCL

$DS_ASKSTR

$DS_ASKVLD

The "ask" system services are used to obtain information from the program user at run time. With these services, the diagnostic program can

- Prompt the user with a message specified by the programmer
- Obtain keyboard input from the user
- Interpret and validate the input string
- Store the value specified by the input string

The Ask for Address ($DS_ASKADR) system service is used when the information requested from the user is an address.

The Ask for Data ($DS_ASKDATA) system service is used when the information requested from the user is a numeric value other than an address.

The Ask for Logical Response ($DS_ASKLGCL) system service is used to ask the user a question that can be answered with a "yes" or "no" response. Optionally, the caller can specify addresses of routines that will automatically be branched to on a "yes" or "no" response.

The Ask for Character String ($DS_ASKSTR) system service is used to obtain an alphabetic character string from the user. Optionally, the caller can also provide a set of valid response strings. The system service will compare the input string to the valid responses and indicate to the caller which response was provided.

The Ask for Data Field ($DS_ASKVLD) system service is used to obtain a numeric value from the user and insert the value into a data field indicated by a position and size. This service is useful for loading fields in large data structures (greater than 32 bits).

MACRO-32 Format:

```
$DS_ASKADR_x msgadr, datadr, [radix], [lolim], [hilim],
[defalt], [unused], [exword]

$DS_ASKDATA_x msgadr, datadr, [radix], [mask], [lolim],
[hilim], [defalt], [unused], [exword]

$DS_ASKLGCL_x msgadr, datadr, [pos], [yexfer], [noxfer],
[defalt]

$DS_ASKSTR_x msgadr, bufadr, [maxlen], [valtab], [defadr]

$DS_ASKVLD_x msgadr, datadr, [radix], [pos], [size], [lolim],
[hilim], [defalt], [unused], [exword]
```

BLISS-32 Format:

```
$DS_ASKADR (MSGADR=msgadr,
                DATADR=datadr,
                [RADIX=radix],
                [LOLIM=lolim],
                [HILIM=hilim],
                [DEFALT=defalt],
                [EXWORD=exword]);

$DS_ASKDATA (MSGADR=msgadr,
                DATADR=datadr,
                [RADIX=radix],
                [MASK=mask],
                [LOLIM=lolim],
                [HILIM=hilim],
                [DEFALT=defalt],
                [EXWORD=exword]);

$DS_ASKLGCL (MSGADR=msgadr,
                DATADR=datadr,
                [POS=pos],
                [YEXFER=yexfer],
                [NOXFER=noxfer],
                [DEFALT=defalt]);

$DS_ASKSTR (MSGADR=msgadr,
                BUFADR=bufadr,
                [MAXLEN=maxlen],
                [VALTAB=valtab],
                [DEFADR=defadr]);
```

```
$DS_ASKVLD (MSGADR=msgadr,
            DATADR=datadr,
            [RADIX=radix],
            [POS=pos],
            [SIZE=size],
            [LOLIM=lolim],
            [HILIM=hilim],
            [DEFALT=defalt],
            [EXWORD=exword]);
```

msgadr

Address of counted ASCII string to be used as user prompting message.

datadr

Address of longword to receive interpreted user response value.

For $DS_ASKDATA, value is placed in bit position indicated by "mask."

For $DS_ASKVLD, value is placed in field indicated by "pos" and "siz," where "pos" is bit offset from "datadr."

For $DS_ASKLGCL, value will be placed in one bit, indicated by "pos." The bit can be compared with PAR$NO and PAR$YES, defined in $DS_PARDEF. (No = Ø, yes = 1).

bufadr ($DS_ASKSTR only)

Address of buffer that will receive counted ASCII input string.

maxlen ($DS_ASKSTR only)

Size of the buffer specified in "bufadr." The default value is 72.

valtab ($DS_ASKSTR only)

Address of table containing list of string pointers. See Note 4 for table format. Each table entry points to an ASCII string (uncounted) that represents a valid user response. The system service will compare actual user input to the valid responses. If a match is found, the number of the table entry pointing to the matched string will be returned in R1. If a match is not found, the system service will inform the user that an invalid response has been issued and will then reissue the prompt message. See Note 5 for a description of the string comparison algorithm.

If this parameter is Ø (the default), no validation will take place.

defadr ($DS_ASKSTR only)

Address of counted ASCII string to be used as a default user response. The default value for this parameter is Ø, which means there is no default user response.

radix

Radix in which the user response is to be interpreted. Legal values for this parameter are defined by the macro $DS_PARDEF, and consist of PAR$_BIN, PAR$_OCT, PAR$_DEC, and PAR$_HEX. The default radix is decimal, except in the case of $DS_ASKADR, for which the default is hexadecimal.

mask ($DS_ASKDATA only)

Mask indicating the bit positions within "datadr" in which the interpreted user response should be stored. The default value is FFFFFFFF (hexadecimal), indicating 32 bits starting at bit Ø.

pos ($DS_ASKVLD and $DS_ASKLGCL only)

Bit offset from "datadr," indicating where interpreted user response is to be stored. See Note 6 for legal values. Default is Ø, indicating value should be stored starting at bit Ø of "datadr."

size ($DS_ASKVLD only)

Number of bits in "datadr" in which interpreted user response is to be stored. Range is 1 through 32.

lolim

Minimum acceptable value for numeric user reponse. Default is minus 2 to the 31st power, except in the case of $DS_ASKADR, for which the default is (unsigned) Ø.

hilim

Maximum acceptable value for numeric user response. Default is 2 to the 31st power minus 1, except in the case of $DS_ASKADR, for which the default is (unsigned) FFFFFFFF (hexadecimal).

defalt

The value to be used if the user does not provide a response (user just types return key). The default value for "defalt" is Ø (which, for $DS_ASKLGCL, is equivalent to a "no" response). If no default is to be used, then NODEF must be set in the "exword" parameter.

For the $DS_ASKLGCL macro, default values may be specified by the symbols PAR$NO and PAR$YES, defined by the $DS_PARDEF macro.

yexfer ($DS_ASKLGCL only)

Address to branch to if user response is "yes." Default is Ø, meaning no branch will take place.

noxfer ($DS_ASKLGCL only)

Address to branch to if user response is "no." Default is Ø, meaning no branch will take place.

unused

Reserved for expansion.

exword

The "exception mask." This is a longword containing "exception flags." These flags are used to modify the interpretations of some of the other parameters. Symbols for the exception flags are defined by the $DS_PARDEF macro. Refer to the description of that macro for the complete symbol names. The flags are:

- NODEF - There is to be no default value for the user response. In other words, the "defalt" parameter is to be ignored.

- ATDEF - The argument specified for the "defalt" parameter is the address of a location containing the default value.

- ATLO - The argument specified for the "lolim" parameter is the address of a location containing the low limit value.

- ATHI - The argument specified for the "hilim" parameter is the address of a location containing the high limit value.

By default, all flags are cleared.

Return Status:

SS$_NORMAL

Service successfully completed.

DS$_PROGERR

An incorrect number of arguments was supplied with the macro.

DS$_TRUNCATE

For $DS_ASKSTR, the string supplied by the user was too long to fit into the buffer pointed to by "bufadr." The string was truncated in order to fit into the buffer.

For $DS_ASKDATA and $DS_ASKVLD, the value specified by the user was too large to fit into the bit field specified by the caller. The value was truncated in order to fit into the specified field.

Notes:

1. If the VDS control flag OPERATOR is clear, and if no default value has been specified for the prompting message, then the diagnostic program will be aborted. Thus if the diagnostic program is intended to be executed in an automated run-time environment (such as APT), these macros cannot be used unless default values are provided.

   It is also required that if these macros are used in the DEFAULT program section (see Section 3.8.3), default values must be provided.

2. If the VDS control flag PROMPT is set, the ranges and default values for user responses will be displayed along with the prompting message.

3. To ensure that prompting messages are left-justified, precede each prompting message with a CR and LF.

4. Figure 4-4 illustrates the format of the "valtab" table.

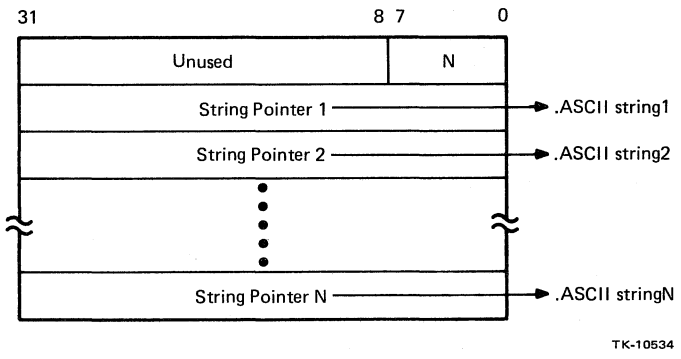

Figure 4-4 "Valtab" Table Format

5. When the $DS_ASKSTR system service compares the user response string with the set of valid responses optionally specified by "valtab," it will go through the table of valid string pointers and, for each valid string, it will compare the characters of the user response with the characters of the valid string until the end of the user response is reached. If all the characters of the user response match all the characters of the current valid string, up to the end of the user reponse, a match is declared. This means the user can abbreviate input strings. For example, if a valid string is EXECUTE, the user can type EXEC, EX, or even just E. However, suppose two valid strings are START and STOP. If the user typed ST, then the service routine would declare a match on whichever of the valid strings was defined first in "valtab."

6. When the "pos" parameter is used with the $DS_ASKLGCL macro, its legal range is Ø through 7.

   When the "pos" parameter is used with the $DS_ASKVLD macro, its legal range normally is Ø through the largest value that can be stored in a longword. However, if a register is specified for "datadr," then the legal range for "pos" is Ø through 31.

Examples:

MACRO-32 Example:

```
PROMPT:         .ASCIC  /DEVICE ADDRESS:/
RESPONSE:       .LONG 0

   $DS__ASKADR_S -
           MSGADR = PROMPT, -
           DATADR = RESPONSE, -
           RADIX  = #PAR$_OCT, -
           LOLIM  = #760000, -
           HILIM  = #777777, -
           DEFALT = #764000
```

BLISS-32 Example:

```
BIND
   PROMPT = UPLIT (%ASCIC 'IS THE DRIVE WRITE-ENABLED?);

LOCAL
   RESPONSE;
          :
          :
          :
   $DS_ASKLGCL (MSGADR=PROMPT, DATADR=RESPONSE);
```

$ASSIGN

The Assign I/O Channel system service of VMS is used to provide an I/O channel that can be used by the caller to communicate with a peripheral device in user mode. Level 2R programs must issue the $ASSIGN macro before the $QIO macro can be used. Refer to Section 3.12.1.1 for details of I/O operations in user mode.

This service can also be used to create a logical link with a remote node on a network. Refer to the DECnet-VAX User's Guide for details.

MACRO-32 Format:

```
$ASSIGN_x devnam, chan, [acmode], [mbxnam]
```

BLISS-32 Format:

```
$ASSIGN (DEVNAM=devnam, CHAN=chan, [ACMODE=acmode],
[MBXNAM=mbxnam]);
```

devnam

Address of a character string descriptor (see Section 4.3) pointing to the device name string. The string may be either a physical device name or a logical name. If the first character of the string is an underscore (_), then the name is a physical name. Otherwise one level of logical name translation is performed and the equivalence name, if any, is used.

If the device name contains a double colon (::), VMS assigns a channel to the device NETØ: and performs an access function on the network.

chan

Address of a longword to receive the channel number assigned.

acmode

Access mode to be associated with the channel. The specified access mode is maximized with the access mode of the caller. I/O operations on the channel can only be performed from equal and more privileged access modes. Legal values are Ø for Kernel, 1 for Executive, 2 for Supervisor, and 3 for User.

mxbnam

Address of a character string descriptor (see section 4.3) pointing to the logical name string for the mailbox to be associated with the device, if any. The mailbox receives status information from the device driver. An address of Ø implies no mailbox. This is the default value.

Return Status:

SS$_NORMAL

Service successfully completed.

SS$_REMOTE

Service successfully completed. A logical link is established with the target on a remote node.

SS$_ABORT

A physical line went down during a network correct operation.

SS$_ACCVIO

A device or mailbox name string or string descriptor cannot be read by the caller, or the channel number cannot be written by the caller.

SS$_DEVACTIVE

A mailbox name has been specified, but a mailbox is already associated with the device.

SS$_DEVALLOC

Warning. The device is allocated to another process.

SS$_DEVNOTMBX

A logical name has been specified for the associated mailbox, but the logical name refers to 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 dynamic system memory to complete the request.

SS$_IVDEVNAM

No device name was specified, or the device or mailbox name string contains invalid characters. If the device name is a target on a remote node, this status code indicates that the Network Control Block has an invalid format.

SS$_IVLOGNAM

The device or mailbox name string has a length of Ø or has more than 63 characters.

SS$_NOIOCHAN

No I/O channel is available for assignment.

SS$_NOLINKS

No logical network links are available.

SS$_NOPRIV

The process does not have the privilege to perform network operations.

SS$_NOSUCHDEV

Warning. The specified device or mailbox does not exist.

SS$_NOSUCHNODE

The specified network node is nonexistent or unavailable.

SS$_REJECT

The network connect was rejected by the network software or by the partner at the remote node; or the target image exited before the connect confirm could be issued.

Notes:

Refer to the VAX/VMS System Services Reference Manual for notes on the $ASSIGN system service. That manual should be read before attempting I/O operations in user mode.

Examples:

MACRO-32 Example:

```
TTNAME:    .ASCID     /TTA2:/     ;TERMINAL DESCRIPTOR
TTCHAN:    .BLKL      1           ;TERMINAL CHANNEL NUMBER
            :
            :
            :
$ASSIGN_S                      DEVNAM=TTNAME, CHAN=TTCHAN
```

BLISS-32 Example:

```
BIND
  TTNAME = UPLIT (%ASCID 'TTA2:');

OWN
  TTCHAN : VECTOR;
           :
           :
           :
  $ASSIGN (DEVNAM=,TTNAME, CHAN=TTCHAN);
```

$DS_ATTACH

The Attach Device system service can be used to "attach" a device automatically from within the diagnostic program, instead of requiring the program user to issue the ATTACH command. Attaching devices is discussed in Section 3.2. An example of when it might be desirable to use the $DS_ATTACH macro is the case in which record management services (RMS) are to be used to reference a file on a device other than the VDS default load device.

MACRO-32 Format:

```
$DS_ATTACH_x cmd, [pmt]
```

BLISS-32 Format:

```
$DS_ATTACH (CMD=cmd, [PMT=pmt]);
```

cmd

Address of a quadword descriptor that points to a valid ATTACH command argument string. If the argument string does not contain every necessary response to each ATTACH prompt, the "pmt" parameter must also be specified. (The argument string should not include prompting strings).

pmt

Address of a quadword descriptor pointing to a buffer that will receive error messages and prompting messages if the command string pointed to by "cmd" is incomplete or in error. This parameter is optional only if the programmer is absolutely sure that the specified command string will always be correct for any hardware configuration. Using the contents of this buffer is discussed in Note 1.

Return Status:

SS$_NORMAL

Service successfully completed.

DS$_BADTYPE

An invalid device type was specified in the argument string.

DS$_BADLINK

The device link specified in the argument string is not attached.

DS$_ILLUNIT

The device unit number was required and not given, or was too large.

DS$_DEVNAME

The device name specified in the argument string is invalid.

SS$_BADPARAM

A numeric argument was specified in an invalid radix or was out of range.

SS$_INSFARG

The argument string was incomplete.

Notes:

1. If an argument in the argument string is invalid, or if the argument string is incomplete, the following will occur:

   a. One of the error status codes will be returned.

   b. The length field of the quadword descriptor pointed to by "cmd" will be altered to reflect the length of the valid portion of the argument string.

   c. The buffer described by "pmt" will contain a VDS-generated error message and the user prompt for the invalid or missing argument.

   The contents of the "pmt" buffer can be used as the prompting string ("msgadr" parameter) of a $DS_ASKSTRING macro. The user's response could then be added to the argument string, after the last valid argument. The argument string's size would then be readjusted and the $DS_ATTACH macro would be reissued. (Note that a p-table is not actually built until all arguments are valid, so this process can be repeated until the user has supplied a complete argument string.)

This service will not display any information on the user's terminal. Thus if an error occurs, simply using $DS_ASKSTRING macro to display the error message and prompt is insufficient, since the user will have no idea what device is being attached! It will be necessary for the program to display an explanatory message indicating (1) that an attach was being attempted and (2) which device was being attached.

Examples:

MACRO-32 Example:

```
CMDLINE:  .ASCID   /RH780 SBI RHO 8 5/
          :
          :
          :
          $DS_ATTACH_S CMDLINE;
```

BLISS-32 Example:

```
BIND
  CMDLINE = UPLIT (%ASCID 'RH780 SBI RHO 8 5');
          :
          :
          :
  $DS_ATTACH (CMD=.CMDLINE);
```

$BINTIM

The Convert ASCII String to Binary Time system service converts an ASCII string to an absolute or offset time value in the system 64-bit time format suitable for input to the $SETIMR service.

MACRO-32 Format:

```
$BINTIM_x timbuf, timadr
```

BLISS-32 Format:

```
$BINTIM (TIMBUF=timbuf, TIMADR=timadr);
```

timbuf

Address of a character string descriptor (see Section 4.3) pointing to the buffer containing the absolute or offset time to be converted. See notes for input string format.

The maximum offset time that may be specified is 10,000 days.

timadr

Address of a quadword to receive the converted time in 64-bit format.

Return Status:

SS$_NORMAL

Service successfully completed.

SS$_IVTIME

Syntax of the input string is invalid, or the specified time is out of range.

Notes:

1. For absolute time, the input string must be formatted as

   dd-mmm-yyyy hh:mm:ss.cc

   For absolute time, any of the fields may be omitted, but all punctuation must be included. The system will fill in the current values for all unspecified fields.

   Examples are:

   a. 5-DEC-1983 5:16:14.98 (16 minutes, 14.98 seconds after 5 A.M. on 5-DEC-1983)

   b. -- 14:ØØ:ØØ.ØØ (2 P.M. today)

   c. -- ::Ø5 (5 seconds past the current time)

2. For relative time (time offset from the current time), the input string format is

   dddd hh:mm:ss.cc

   For relative time, any of the fields may be omitted, but all punctuation must be included. The system will default all unspecified fields to Ø.

   Examples are:

   a. 4 12:46:14.56 (4 days, 12 hours, 46 minutes, 14.56 seconds from now)

   b. Ø 5:12 (5 minutes and 12 seconds from now)

   c. Ø ::1Ø (1Ø seconds from now)

Examples:

MACRO-32 Example:

```
ONE_MIN:          .ASCID  /0 00:01:00.00/ ;DESCRIPTOR FOR 1 MINUTE.
BIN_TIM:          .QUAD 1                 ;QUADWORD TO HOLD BINARY TIME.
                            :
                            :
                  $BINTIM_S ONE_MIN, BIN_TIM
```

BLISS-32 Example:

```
BIND
  ONE_MIN =
          UPLIT (%ASCID '0 00:01:00.00'); ! DESCRIPTOR FOR 1 MINUTE.

LOCAL
  BIN_TIM : VECTOR [2];      ! QUADWORD TO HOLD BINARY TIME.
                :
                :
          $BINTIM (TIMBUF=ONE_MIN, TIMADR=BIN_TIM);
```

$DS_BREAK

The Break system service causes a temporary return to the VDS to take place. The main purpose of this return is to see if any asynchronous events (including receipt of a control-C character from the user terminal) have occurred and are waiting to be processed.

All diagnostic programs must return to the VDS at least once every three seconds. Issuing any system service macro or program control macro, plus some program structure macros (such as $DS_ENDSUB and $DS_ENDTEST) is considered to be a return to the VDS, so the $DS_BREAK service only needs to be called if none of those macros has been issued in a particular three-second interval. Be particularly careful that all potential program loops (see Section 3.1Ø) adhere to this constraint.

MACRO-32 Format:

```
$DS_BREAK
```

(No suffix.)

BLISS-32 Format:

```
$DS_BREAK;
```

Return Status:

None.

Examples:

MACRO-32 Example:

```
$DS_BREAK
```

BLISS-32 Example:

```
$DS_BREAK;
```

$CANCEL

The Cancel I/O on Channel system service can be used to cancel I/O requests that were created with the $QIO and $QIOW system services. The caller specifies the number of the channel for which I/O requests are to be canceled, and the service will cancel all current and pending I/O operations directed to the channel.

Level 3 programs may not use this service.

MACRO-32 Format:

```
$CANCEL_x chan
```

BLISS-32 Format:

```
$CANCEL (CHAN=chan);
```

chan

Number of the I/O channel on which I/O is to be canceled.

Return Status:

SS$_NORMAL

Service successfully completed.

SS$_EXQUOTA (user mode only)

The process has exceeded its direct I/O quota.

SS$_INSFMEM

Insufficient memory space is available to perform the Cancel I/O service.

SS$_IVCHAN

An invalid channel number was specified, that is, a channel number of Ø or a number larger than the number of channels available.

SS$_NOPRIV (user mode only)

The specified channel was not assigned, or was assigned from a more privileged access mode.

Notes:

1. See the VAX/VMS System Services Reference Manual for discussions of privilege restrictions, resource requirements, and other notes relating to the $CANCEL service.

Examples:

MACRO-32 Example:

```
$CANCEL_S CHANNUM
```

BLISS-32 Example:

```
$CANCEL (CHAN=.CHANNUM);
```

$CANTIM

The Cancel Timer Request system service can be used to cancel timer requests previously made with the $SETIMR macro. See Section 3.14.4, Timing.

MACRO-32 Format:

```
$CANTIM_x [reqidt], [acmode]
```

BLISS-32 Format:

```
$CANTIM ([REQIDT=reqidt], [ACMODE=acmode]);
```

reqidt

The request identification number of the timer request(s) to be canceled. A request id number is associated with each timer request when the $SETIMR macro is used. The $CANTIM service will only cancel the requests having the specified id number. The default value is Ø, which means that all timer requests should be canceled, regardless of their id numbers.

acmode (user mode only)

Access mode of the requests to be canceled. In user mode, 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:

SS$_NORMAL

Service successfully completed.

Examples:

MACRO-32 Example:

```
$CANTIM_S #2    ;Cancel timer request(s) with ID of 2.
```

BLISS-32 Example:

```
$CANTIM ();     !Cancel all timer requests.
```

$DS_CANWAIT

The Cancel Wait system service is used to cancel a program wait state that was created by using the $DS_WAITMS or $DS_WAITUS macro. See Section 3.14.4, Timing.

MACRO-32 Format:

```
$DS_CANWAIT_x
```

BLISS-32 Format:

```
$DS_CANWAIT;
```

Return Status:

SS$_NORMAL

Service successfully completed.

Notes:

1. The $DS_CANWAIT macro is only useful if it is included in an AST routine or interrupt service routine that was entered while a $DS_WAITMS or $DS_WAITUS service was being executed. See Section 3.14.4.

Examples:

MACRO-32 Example:

```
$DS_CANWAIT_S
```

BLISS-32 Example:

```
$DS_CANWAIT;
```

$DS_CHANNEL

The Channel Adapter system service of the VDS is used to control functions that are initiated by referencing internal registers in the bus adapters. This service takes into account all processor-specific differences in the adapters and thus insulates the diagnostic program from those differences.

The Channel Adapter service enables the program to:

- Initialize a MASSBUS adapter or a UNIBUS adapter
- Initialize a UNIBUS
- Enable and disable interrupts from a MASSBUS adapter or a UNIBUS adapter
- Abort data transfers on a MASSBUS adapter
- Purge a UNIBUS data path
- Set or clear UNIBUS defeat parity
- Request or clear adapter status

For descriptions of the design and operation of the various bus adapters for VAX processors, refer to the VAX Hardware Handbook.

The Channel Adapter system service may only be used by level 3 diagnostic programs.

MACRO-32 Format:

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

BLISS-32 Format:

```
$DS_CHANNEL (UNIT=unit, FUNC=func, [VECADR=vecadr],
[STSADR=stsadr]);
```

unit

Logical unit number of the device unit to be tested. The function specified by "func" will be performed on the adapter to which this device unit is attached.

func

Function code indicating the function to be performed by the $DS_CHANNEL service. Must be a literal value. In MACRO-32, function codes are defined by the $DS_CHCDEF macro. The function codes are described in Note 1.

vecadr

Address of interrupt service routine to receive control when an interrupt occurs. The interrupt may come from the device specified by "unit" or from the adapter to which the device is attached. This parameter is only used with the CHC$_ENINT function code, in which case it is required.

stsadr

Address of a quadword to receive adapter status. Used only with the CHC$_ENINT and CHC$_STATUS function codes, in which cases it is required. The adapter status is discussed in Note 2.

Return Status:

$DS_NORMAL

Service successfully completed.

$DS_ERROR

The specified logical unit number is too large.

$DS_IHWE

Initial hardware error. An error condition was detected in the adapter before the specified function was initiated. The function will not be performed. Note: To determine exact hardware error after this status is returned, issue a CHC$_STATUS function.

$DS_IVVECT

The p-table for the device unit indicated with the "unit" parameter contains an invalid vector address.

$DS_LOGIC

An attempt to set or clear a bit within an adapter register has failed. Indicates a hardware failure.

$DS_NOSUPPORT

The specified function is not supported on the processor type being used. This is not an error condition. See Note 4.

$DS_PROGERR

An invalid function code was specified.

A required argument was not included with the macro call.

Notes:

1. Following is a list of the valid function codes along with their functions and return status codes.

   - CHC$_INITA - Initialize the MASSBUS or UNIBUS adapter to which the device unit specified by "unit" is attached.

     Return status codes: DS$_NORMAL, DS$_ERROR, DS$_LOGIC, DS$_NOSUPPORT

   - CHC$_INITB - Initialize the UNIBUS to which the device unit specified by "unit" is attached.

     Return status codes: DS$_NORMAL, DS$_ERROR, DS$_LOGIC, DS$_NOSUPPORT

   - CHC$_ENINT - Enable interrupts for the MASSBUS or UNIBUS adapter to which the device unit specified by "unit" is attached. Refer to Note 3 for details.

     Return status codes: DS$_NORMAL, DS$_ERROR, DS$_IHWE, DS$_NORMAL, DS$_LOGIC, DS$_PROGERR

   - CHC$_DSINT - Disable interrupts for the MASSBUS or UNIBUS adapter to which the device unit specified by "unit" is attached.

     Return status codes: DS$_NORMAL, DS$_ERROR, DS$_IHWE, DS$_IVVECT, DS$_LOGIC

   - CHC$_ABORT - Abort data transfers on the MASSBUS adapter to which the device unit specified by "unit" is attached.

Return status codes: DS$_NORMAL, DS$_ERROR, DS$_NOSUPPORT

- CHC$_PURGE - Purge a buffered data path on a UNIBUS. The buffered data path that is purged is the one specified by the last DS$_SETMAP macro call. The UNIBUS will be the one to which the device unit specified by "unit" is attached.

  Return status codes: DS$_NORMAL, DS$_ERROR, DS$_NOSUPPORT

- CHC$_CLEAR - Clear status bits. Clears error bits in the status registers of the adapter to which the device unit specified by "unit" is attached. This function should be requested before interrupts are enabled.

  Return status codes: DS$_NORMAL, DS$_ERROR, DS$_LOGIC, DS$_NOSUPPORT

- CHC$_STATUS - Fetch status for the adapter to which the device unit specified by "unit" is attached. The current status of the adapter will be returned in the quadword specified by "stsadr." See Note 1 for details.

  Return status codes: DS$_NORMAL, DS$_ERROR

- CHC$_SETDFT - Sets the Defeat Data Path Parity bit for the adapter to which the device unit specified by "unit" is attached.

  Return status codes: DS$_NORMAL, DS$_ERROR, DS$_NOSUPPORT

- CHC$_CLRDFT - Clears the Defeat Data Path Parity bit for the adapter to which the device unit specified by "unit" is attached.

  Return status codes: DS$_NORMAL, DS$_ERROR, DS$_NOSUPPORT

2. Adapter Status

   Adapter status will be returned to the caller in two instances:

   1. The CHC$_STATUS function is requested.

   2. An interrupt has occurred.

In the latter case, the interrupt service routine (whose address was specified with the "vecadr" parameter) can (and should) examine the status quadword to see if errors have occurred.

The returned status quadword will have the format indicated in Figure 4-5.

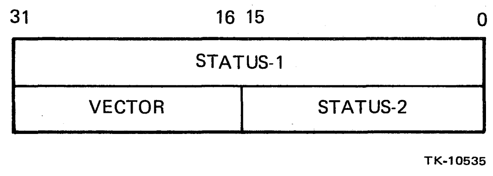

Figure 4-5 Adapter Status Format

Note: Both longwords are filled when an interrupt occurs. If the CHC$_STATUS function is requested, however, only the first longword is filled in -- the second longword is cleared.

"Status-1" is a bitmap, each bit representing an error condition. Each bit has a symbolic name associated with it, in the form CHS$V_xxxxxx. A longword mask for each bit is also defined, with the form CHS$M_xxxxxx. In MACRO-32, these symbols are defined by the $DS_CHSDEF macro. Status-1 bits are defined as follows:

Bit Ø - CHS$V_SYSERR - System error. Set if either of bits 9 or 1Ø is set.

Bit 1 - CHS$V_CHNERR - Channel error. Set if any of bits 6, 7, 8, 25, 26, and 27 are set.

Bit 2 - CHS$V_DEVERR - Device error. Set if either of bits 4 or 5 is set.

Bit 3 - CHS$V_PGMERR - Program error. Set if bit 11 is set.

Bits Ø, 1, 2, and 3 - CHS$M_ERRANY (defined only as longword mask) - Can be used to test if any error conditions of types SYSERR, CHNERR, DEVERR, OR PGMERR exist.

Bit 4 - CHS$V_DEVBUS - Bus error. Some type of error has occurred on the bus.

Bit 5 - CHS$V_DEVTO - Device timeout. The referenced device did not respond.

Bit 6 - CHS$V_CHNDPE - Data path parity error.

Bit 7 - CHS$V_CHNMPE - Map parity error. A MASSBUS page frame map parity error or a UNIBUS map register parity failure was detected.

Bit 8 - CHS$V_CHPFOT - Power failure/Overtemp. A power failure or overtemperature condition was detected.

Bit 9 - CHS$V_SYSMEM - System memory error. Set if any of a number of error conditions relating to data transfers was detected.

Bit 1Ø - CHS$V_SYSSBI - SBI error. For processors having an SBI, this bit is set if an SBI error condition is detected.

Bit 11 - CHS$V_PGMHDE - Hardware-detected program error. The mapping registers were not set up correctly by the software, or the software attempted to initiate a MASSBUS data transfer while one was already in progress.

Bits 12 through 15 - Unused.

Bit 16 - CHS$V_MBAEXC - MASSBUS exception.

Bit 17 - CHS$V_MBANED - Nonexistent MASSBUS device. The referenced MASSBUS device did not respond. Equivalent to bit 5.

Bit 18 - CHS$V_MBADTB - MASSBUS DTBUSY. Set if MASSBUS DTBUSY is set (not an error bit).

Bit 19 - CHS$V_MBADTC - MASSBUS data transfer completed. Set if MASSBUS DT CMP is set.

Bit 2Ø - CHS$V_MBAATN - MASSBUS attention. Set if MASSBUS ATTN is set.

Bit 21 - CHS$V_MBACPE - MASSBUS control parity error. Set if MASSBUS MCPE is set.

Bit 22 - CHS$V_BUSINIT - UNIBUS INIT asserted. Set if UB INIT is set.

Bit 23 - CHS$V_BUSIC - UNIBUS initialization completed. Set if UBIC is set.

Bit 24 - CHS$V_BUSPDN - UNIBUS power down. Set if UB PDN is set.

Bit 25 - CHS$V_MBAWCKLWR - MASSBUS write check lower error. Set if MASSBUS WCK LWR ERR is set.

Bit 26 - CHS$V_WBAWCKUPR - MASSBUS write check upper error. Set if MASSBUS WCK UP ERR is set.

Bit 27 - CHS$V_BUSNXM - UNIBUS nonexistent memory or device. The referenced address does not respond.

Note: Whenever "status-1" indicates error conditions, the program should call the $DS_SHOWCHAN service so that the bus adapter's internal registers will be displayed on the user's terminal. This will enable the user to determine the exact cause of the error condition.

If examined in an interrupt service routine, "status-2" will contain the following:

1. A bit called CHI$V_CHNINT which, if set, indicates that the interrupt was issued from the adapter.

2. A bit called CHI$V_DEVINT which, if set, indicates that the interrupt was issued from a device.

3. A five-bit field, starting at bit position CHI$V_IPL and having a length defined by CHI$S_IPL, which contains the IPL of the interrupt.

For MACRO-32, the fields of "status-2" are defined by the $DS_CHIDEF macro. Note that CHI$V_CHNINT and CHI$V_DEVINT are not mutually exclusive, that is, both a device interrupt and an adapter interrupt can be received at the same time.

The "vector" field will contain the vector of the device which caused the interrupt. The field is only relevant for interrupts from devices attached to a UNIBUS.

Interrupts

The CHC$_ENINT function enables interrupts for the adapter (if the adapter is capable of generating interrupts). Device interrupts must be explicitly enabled by the diagnostic program. The CHC$_ENINT function also loads the appropriate vector addresses. Thus this function MUST be used, even if the adapter itself cannot generate interrupts.

Device vector addresses are loaded with the address of an interrupt preprocessor within the VDS. When an interrupt occurs, program control is vectored to the interrupt preprocessor.

The preprocessor will first raise the processor's IPL to 17 (hex). Next it will check for errors incurred by the bus adapter and then construct the status quadword. It will then determine the type of interrupt: adapter, device, or "passive release." If the interrupt was from an adapter or device, the appropriate bit in "status-2" is set and control is passed to the user's interrupt service routine ("vecadr") with a JMP instruction. If a "passive release" has occurred, an REI instruction is executed without calling the user's interrupt service routine.

The user's interrupt service routine should check the vector address passed in the status quadword to ensure that the interrupt received was from the expected device. This can be done by comparing the vector in the status quadword with the vector in HP$W_VECTOR of the interrupting device's p-table.

It is not wise to request the CHC$_INITA or CHC$_INITB function while interrupts are enabled.

Processor-Specific Considerations

For some processors, some functions are not relevant. However, requesting such functions will not cause an error. The $DS_NOSUPPORT status will be returned, but the program need not necessarily test for this code. For example, the CHC$_INITB is not relevant on a VAX-11/730 but, in order to allow a diagnostic program to be compatible with all processor types, the VDS will not reject the function -- it will just return the $DS_NOSUPPORT status code.

Examples:

Following is an example in MACRO-32 and BLISS-32 of code that will initialize a MASSBUS, enable bus interrupts, then issue a SEARCH function on an RP06 disk drive.

MACRO-32 Example:

```
                :
                :
        $DS_CHANNEL_S -                          ; Initialize MASSBUS
                DRIVE, #CHC$_INITA               ;
        $DS_SETIPL_S #0                          ; Lower IPL
        MOVL    NEXT_ADDR,RPDA(R2)               ; Next disk address to access.
        MOVL    CYLINDER,RPDC(R2)                ; Desired cylinder.
        $DS_CHANNEL_S -                          ; Enable interrupts.
                DRIVE, #CHC$_ENINT, SERVICE_RTN, CH_STATUS
        CLRQ    CH_STATUS                        ; Clear status quadword.
        MOVL    #SEARCH!GO,(R2)                  ; SEARCH function.
20$:            BBC     #CHS$V_MBAATN, -         ; Wait for SEARCH to finish.
                CH_STATUS, 20$                   ;
        BITL    #ERR,RPDS(R2)                    ; Check for drive errors.
                :
                :
```

BLISS-32 Example:

```
        :
        :
    $DS_CHANNEL                                  ! Initialize MASSBUS
        (UNIT = .DRIVE,                          !
        FUNC = CHC$_INITA);                      !
    $DS_SETIPL (0);                              ! Lower IPL
    .(RP_BASE + RPDA) = .NEXT_ADDR;              ! Next disk address to access.
    .(RP_BASE + RPDC) = .CYLINDER;               ! Desired cylinder.
    $DS_CHANNEL                                  ! Enable interrupts.
            UNIT = .DRIVE,                       !
            FUNC = CHC$_ENIT,                    !
            VECADR = SERVICE_RTN,                !
            STSADR = CH_STATUS;                  !
    CH_STATUS = 0;                               ! Clear status quadword.
    .(RP_BASE + RPCS) = SEARCH OR GO;            ! SEARCH function.
    REPEAT                                       ! Wait for SEARCH to finish.
        1                                        !
    UNTIL .CH_STATUS <CHS$V_MBAATN,1>;           !
    IF .(RP_BASE + RPDS) <ERR,1>                 ! If drive errors occurred
    THEN ...                                     ! then ...
    ELSE ... ;                                   ! else ...
        :
        :
```

$CLOSE

The Close File service of RMS is used to close a file after all processing of the file has been completed. The $CLOSE service will also perform a $DISCONNECT operation.

MACRO-32 Format:

```
$CLOSE fab, [err], [suc]
```

BLISS-32 Format:

```
$CLOSE (FAB=fab, [ERR=err], [SUC=suc]);
```

rab

Address of the RAB to be associated with the FAB describing the file to which connection is to be made. (The address of the FAB is in the RAB.)

err (user mode only)

Address of a routine to be executed on error return from the service.

suc (user mode only)

Address of a routine to be executed on successful return from the service.

Return Status:

Note: For further details on return status values, refer to the VAX-11 RMS Reference Manual.

RMS$_NORMAL

Service successfully completed.

RMS$_CCF

Cannot close file. (Status value will be placed in STV of FAB.)

Notes:

1. Table 4-3 lists the FAB fields used by the $CLOSE service IN STANDALONE MODE. For user mode, refer to the VAX-11 RMS Reference Manual.

Table 4-3 FAB Fields Used by $CLOSE (Standalone Mode)

Field Mnemonic	Field Name
Input:	
IFI	Internal file identifier.
XAB	Extended attribute block address.
Output:	
IFI	Internal file identifier (zeroed).
STS	Completion status code (also returned in RØ).
STV	Status value.

Examples:

MACRO-32 Example:

```
$CLOSE FAB_ADDR
```

BLISS-32 Example:

```
$CLOSE (FAB=FAB_ADDR);
```

$CLREF

The $CLREF macro is used to clear event flags. (Event flags are discussed in Section 3.14.2).

MACRO-32 Format:

```
$CLREF_x efn
```

BLISS-32 Format:

```
$CLREF (EFN=efn);
```

efn

Number of the event flag to be cleared. In user mode, the number may be from 1 through 23 or from 32 through 127. In standalone mode, flags 1 through 64 may be used.

Return Status:

SS$_WASCLR

Service successfully completed. The specified flag was previously Ø.

SS$_WASSET

Service successfully completed. The specified flag was previously 1.

SS$_ILLEFC

An illegal event flag number was specified.

SS$_UNASEFC

In user mode, indicates that the specified common event flag (see Section 3.14.2) has not been associated with the process issuing the CLREF macro.

In standalone mode, indicates that an event flag from 64 through 127 was specified. These flags are not valid in standalone mode.

Examples:

MACRO-32 Example:

```
   $CLREF #5              ;Clear event flag 5.
```

BLISS-32 Example:

```
   $CLREF (EFN=5);        !Clear event flag 5.
```

$DS_CLRVEC

The Clear Exception or Interrupt Vector system service is used to load an exception or interrupt vector with the address of the standard VDS condition handler for the specified vector. The macro's purpose is to restore the standard VDS vector contents after the vector has been modified with the $DS_SETVEC service.

Only level 3 diagnostic programs may use the $DS_CLRVEC macro.

MACRO-32 Format:

```
$DS_CLRVEC_x vector
```

BLISS-32 Format:

```
$DS_CLRVEC (VECTOR=vector);
```

vector

The vector address, relative to the base of the System Control Block (SCB).

Return Status:

DS$_NORMAL

Service successfully completed.

DS$_IVVECT

Address specified for "vector" is not a valid vector address.

Examples:

MACRO-32 Example:

```
$DS_CLRVEC_S    #^X60      ;Restore VDS handler address for
                           ; memory write timeout vector
```

BLISS-32 Example:

```
$DS_CLRVEC (%X'60');        !Restore VDS handler address for
                            ! memory write timeout vector
```

$DS_CNTRLC

The Declare Control-C Handler system service has two purposes. It can be used to:

- Declare a control-C handler that will receive control when the program user types a control-C
- Enable and disable delivery of control-Cs

Refer to Section 3.14.6, Handling Control-Cs, for a details on control-C handlers and disabling delivery of control-Cs.

If the $DS_CNTRLC service is not used, the VDS control-C handler will be invoked.

MACRO-32 Format:

```
$DS_CNTRLC_x [astadr], [disabl]
```

BLISS-32 Format:

```
$DS_CNTRLC ([ASTADR=astadr], [DISABL=disable]);
```

astadr

Address of the control-C handler. Default value is Ø, which causes VDS control-C handler to be declared.

disable

Value used to indicate if control-C delivery should be disabled or enabled. If disable is set to 1, control-C delivery will be disabled. If the value is Ø (the default), control-C delivery is enabled, and control-Cs will be delivered to whichever control-C handler has been selected.

Return Status:

SS$_WASSET

Service successfully completed. Control-C delivery was previously disabled (the disable flag was previously set).

SS$_WASCLR

Service successfully completed. Control-C delivery was previously enabled (the disable flag was previously clear).

Examples:

MACRO-32 Examples:

```
    $DS_CNTRLC_S   CNTRLC_HDLR      ;I want to handle control-Cs.

    $DS_CNTRLC_S                    ;Let VDS handle control-Cs.

    $DS_CNTRLC_S   DISABL=#1        ;Disable control-Cs.
```

BLISS-32 Examples:

```
    $DS_CNTRLC (ASTADR=CNTRLC_HDLR);!I want to handle control-Cs.

    $DS_CNTRLC ();                   !Let VDS handle control-Cs.

    $DS_CNTRLC (DISABLE=1);          !Disable control-Cs.
```

$CONNECT

The Connect RAB to FAB service of RMS is used to associate an RAB to an FAB after the file described in the FAB has been opened with the $OPEN service. The file cannot be read until after it has been connected.

MACRO-32 Format:

```
$CONNECT rab, [err], [suc]
```

BLISS-32 Format:

```
$CONNECT (RAB=rab, [ERR=err], [SUC=suc]);
```

rab

Address of the RAB to be associated with the FAB describing the file to which connection is to be made. (The address of the FAB is in the RAB.)

err (user mode only)

Address of a routine to be executed on error return from the service.

suc (user mode only)

Address of a routine to be executed on successful return from the service.

Return Status:

Note: For further details on return status values, refer to the VAX-11 RMS Reference Manual.

RMS$_NORMAL

Service successfully completed.

RMS$_CCR

An RAB is already associated with the specified FAB.

RMS$_FAB

The FAB block is invalid.

RMS$_IFI

The FAB's IFI field is invalid.

RMS$_RAB

The RAB block is invalid.

RMS$_RAC

Invalid record access mode. In standalone mode, only sequential and RFA access modes are allowed.

Notes:

1. Table 4-4 lists the RAB fields used by the $CONNECT service IN STANDALONE MODE. For user mode, refer to the VAX-11 RMS Reference Manual.

Table 4-4 RAB Fields Used by $CONNECT (Standalone Mode)

Field Mnemonic	Field Name
Input:	
FAB	Address of FAB.
ROP	Record-processing options. (Only BIO is allowed.)
Output:	
STS	Completion status code. (Also returned in RØ.)

Examples:

MACRO-32 Example:

```
$CONNECT RAB_ADDR
```

BLISS-32 Example:

```
$CONNECT (RAB=RAB_ADDR);
```

$DS_CVTREG

The Convert Register Contents to Character String system service can be used to produce an ASCIC character string that associates each field in a register (or any longword) with a mnemonic and indicates the current value of each field. When the string is constructed, the following algorithm is used:

- For fields consisting of only one bit, the field mnemonic is placed into the output string only if the bit is set.
- For fields greater than one bit in length, two options are available:
  - A mnemonic can be associated with the field, in which case the mnemonic and the field's numeric value (in the specified radix) are placed into the output string.
  - Instead of associating a mnemonic with the field, the field's VALUE can have a mnemonic assigned to it. In this case, only the mnemonic is placed into the output string.

The string can be displayed on the user terminal by using one of the $DS_PRINTx services.

MACRO-32 Format:

```
$DS_CVTREG_x msb, data, mneadr, strbuf, maxlen, [vl], [v2],
[v3], [v4], [v5], [v6]
```

BLISS-32 Format:

```
$DS_CVTREG (MSB=msb, DATA=data, MNEADR=mneadr, STRBUF=strbuf,
MAXLEN=maxlen, [V1=vl], [V2=v2], [V3=v3], [V4=v4], [V5=v5],
[V6=v6]);
```

msb

Most significant bit. Reading of the specified location's fields progresses from left to right, so this parameter indicates the first bit that is to be read. Maximum value is 31.

data

Contents to be converted. (Note that this is not the address of the contents, but the contents themselves.)

mneadr

Address of a string of mnemonics and field specifiers.

A mnemonic may be a string of any length, containing any character except '=', ',', or '@'.

Fields are specified in the following manner:

1. For one-bit fields, simply include the mnemonic and follow it by a comma, such as ...,MNEM1,MNEM2,MNEM3,...

2. For multiple-bit fields, two formats are used:

   - If a mnemonic is to be associated with the field, the format is "mnemonic=size^radix", where "size" is the size of the field and "radix" is the radix in which the field contents is to be displayed. Valid values for "radix" are "X" (hexadecimal), "O" (octal), and "D" (decimal). An example is IPL=5^X.

   - If a mnemonic is to be associated with the field's VALUE, then the format is "mnemonic=size@", where "size" is the size of the field. The value's mnemonic is specified using the "v1" through "v6" parameters.

3. If a bit is not to be included in any field, simply include a comma in the mnemonics list; for example, ...,BIT1Ø,BIT9,,,BIT6,...

4. The first mnemonic in the list will be associated with the bit indicated by the "msb" parameter. Mnemonics will be assigned from left to right until the mnemonics list has been exhausted, or until bit Ø has been reached, whichever happens first.

strbuf

Address of a buffer to receive the character string.

maxlen

Length of the buffer pointed to by "strbuf." The buffer may not be greater than 255 bytes. Caution: If the character string overruns the specified length, the buffer will not contain a valid string.

v1 through v6

Each of these, if used, is the address of a counted table of value mnemonics. Each table will contain pointers to lists of mnemonics that are to be associated with the possible values for a particular field. One of these tables will be referenced each time a field specifier with the format "mnemonic=size@" is encountered in the mnemonic string (pointed to by "mneadr"). The first time that format is used, the table pointed to by "v1" will be referenced; the second time the format is used, the table pointed to by "v2" will be referenced, and so on.

Each entry in a table will be the address of a mnemonic that is to be associated with the field's value. The value contained in the field will be used as an offset into the table. If the field's value is Ø, the first table entry will be fetched; if the field's value is 1, the table's second entry will be used, and so on. The mnemonic pointed to by the table entry must be defined by an ASCIC string. The mnemonic will be placed into the output string. Figure 4-6 illustrates the linkages involved in this mechanism.

Return Status:

DS$_NORMAL

Service successfully completed.

DS$_PROGERR

The output string was too large to fit into the buffer provided, or was larger than 255 characters.

The string of mnemonics and field descriptors contains an invalid field descriptor.

The value specified for "msb" was greater than 31.

The total number of macro arguments was greater than 11.

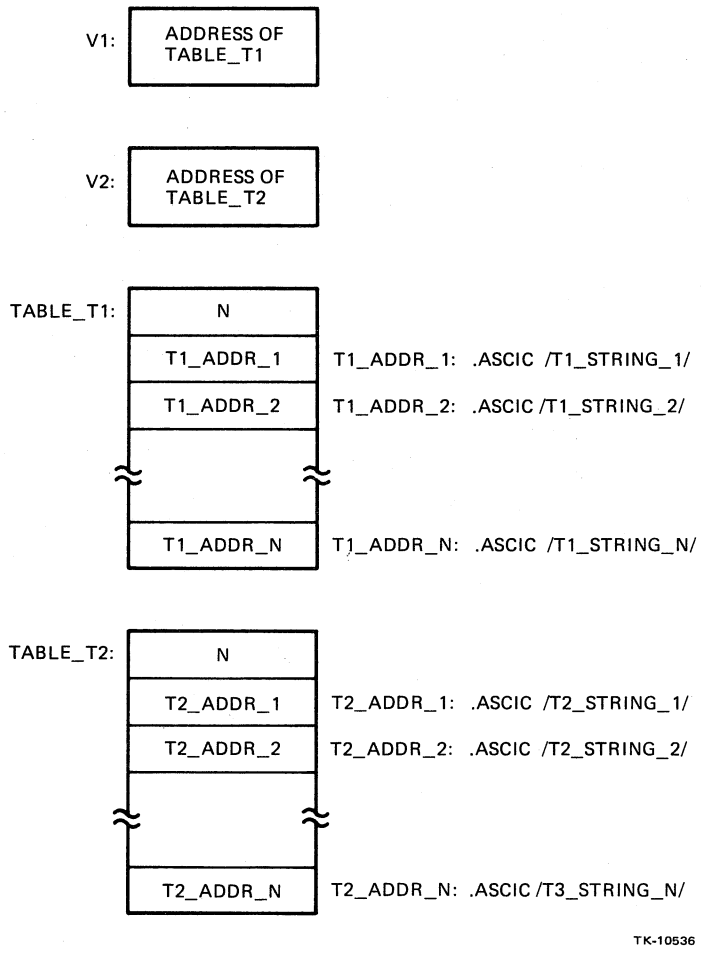
V1:
ADDRESS OF TABLE_T1
V2:
ADDRESS OF TABLE_T2
TABLE_T1:
N
T1_ADDR_1
T1_ADDR_2
T1_ADDR_N
T1_ADDR_1: .ASCIC /T1_STRING_1/
T1_ADDR_2: .ASCIC /T1_STRING_2/
T1_ADDR_N: .ASCIC /T1_STRING_N/
TABLE_T2:
N
T2_ADDR_1
T2_ADDR_2
T2_ADDR_N
T2_ADDR_1: .ASCIC /T2_STRING_1/
T2_ADDR_2: .ASCIC /T2_STRING_2/
T2_ADDR_N: .ASCIC /T3_STRING_N/
TK-10536

Figure 4-6 $DS_CVTREG Value Mnemonics Table Usage

Notes:

1. On return from the service, R1 will contain the total length of the output string, even if the string overflowed.

2. A good convention to follow is to not leave any fields unlabeled. Fields that must be zero (MBZ), are not used, or consist of "don't care" bits should be identified as such. This will cause the fields to be read and displayed, and the program user will know if, for example, an MBZ bit actually is Ø.

Examples:

The following examples illustrate, in both MACRO-32 and BLISS-32, a method of displaying the processor's PSL.

MACRO-32 Example:

```
PSL_NME:    .ASCIC  /CM,TP,MBZ=2^X,FPD,IS,CUR=2@,PRV=2@,MBZ,/ -
                    /IPL=5^X,MBZ=8^X,DV,FU,IV,T,N,Z,V,C/

MODE_LIST:  .LONG   4
            .ADDRESS        KERNEL
            .ADDRESS        EXEC
            .ADDRESS        SUPER
            .ADDRESS        USER

KERNEL:             .ASCIC  /KERNEL/
EXEC:               .ASCIC  /EXECUTIVE/
SUPER:              .ASCIC  /SUPERVISOR/
USER:               .ASCIC  /USER/

OUT_BUF:    .BLKB   255

            :
            :
   MOVPSL   R0                          ;Fetch PSL contents.
   $DS_CVTREG -                         ;Create string.
            MSB     = #31, -            ;Read all 32 bits.
            DATA    = R0, -             ;PSL contents.
            MNEADR  = PSL_MNE, -        ;Mnemonics string.
            STRBUF  = OUT_BUF, -        ;Output buffer.
            MAXLEN  = #255, -           ;Maximum length.
            V1      = MODE_LIST, -      ;1st table.
            V2      = MODE_LIST         ;2nd table (use 1st one again).
            :
            :
```

BLISS-32 Example:

```
BIND
   PSL_MNE =
   UPLIT
   (%ASCIC
   'CM,TP,MBZ=2^X,FPD,IS,CUR=2@,PRV=2@,MBZ,IPL=5^X,MBZ=8^X,
   DV,FU,IV,T,N,Z,V,C');

BIND
   KERNEL   = UPLIT (%ASCIC 'KERNEL'),
   EXEC     = UPLIT (%ASCIC 'EXECUTIVE'),
   SUPER    = UPLIT (%ASCIC 'SUPERVISOR'),
   USER     = UPLIT (%ASCIC 'USER');

OWN
   MODE_LIST : VECTOR [5] INITIAL (4, KERNEL, EXEC, SUPER, USER);

OWN
   OUT_BUF : VECTOR [255, BYTE];

BUILTIN
   MOVPSL;

LOCAL
   PSL_STORE;
        :
        :
        :
   MOVPSL  (PSL_STORE);             !Fetch PSL contents.
   $DS_CVTREG                       !Create string.
        (MSB     = 31,              !Read all 32 bits.
        DATA     = .PSL_STORE,      !PSL contents.
        MNEADR   = PSL_MNE,         !Mnemonics string.
        STRBUF   = OUT_BUF,         !Output buffer.
        MAXLEN   = 255,             !Maxlength.
        V1       = MODE_LIST,       !1st table.
        V2       = MODE_LIST);      !2nd table (use 1st one again).
        :
        :
        :
```

$DASSGN

The Deassign I/O Channel system service of VMS is used to release an I/O channel that was previously assigned with the $ASSIGN service. Level 2R diagnostic programs should use this macro when all I/O operations on a device have been completed. See Section 3.12.1.1 for details of I/O in user mode.

MACRO-32 Format:

```
$DASSGN_x chan
```

BLISS-32 Format:

```
$DASSGN (CHAN=chan);
```

Return Status:

SS$_NORMAL

Service successfully completed.

SS$_IVCHAN

An invalid channel number was specified; that is, a channel number of Ø 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.

Notes:

See the VAX/VMS System Services Reference Manual for notes on the $DASSGN macro. That manual should be read before performing I/O operations in user mode.

Examples:

MACRO-32 Example:

```
$DASSGN_S CHAN_NUM
```

BLISS-32 Example:

```
$DASSGN (CHAN=.CHAN_NUM);
```

$DISCONNECT

The Disconnect RAB from FAB service of RMS is used to break the connection between an RAB and an FAB. This terminates the record stream and deallocates all I/O buffers and data structures.

MACRO-32 Format:

```
$DISCONNECT rab, [err], [suc]
```

BLISS-32 Format:

```
$DISCONNECT (RAB=rab, [ERR=err], [SUC=suc]);
```

rab

Address of the RAB to be disconnected. (The RAB will contain the address of its associated FAB.)

err (user mode only)

Address of a routine to be executed on error return from the service.

suc (user mode only)

Address of a routine to be executed on successful return from the service.

Return Status:

Note: For further details on return status values, refer to the VAX-11 RMS Reference Manual.

RMS$_NORMAL

Service successfully completed.

RMS$_IFI

The FAB's IFI field is invalid.

RMS$_ISI

Invalid stream id. The specified RAB and FAB were not connected.

RMS$_FAB

The FAB block is invalid.

RMS$_RAB

The RAB block is invalid.

Notes:

1. Table 4-5 lists the RAB fields used by the $DISCONNECT service IN STANDALONE MODE. For user mode, refer to the VAX-11 RMS Reference Manual.

Table 4-5 RAB Fields Used by $DISCONNECT (Standalone Mode)

Field Mnemonic	Field Name
Input:	
ISI	Internal stream identifier.
Output:	
STS	Completion status code. (Also returned in RØ.)

Examples:

MACRO-32 Example:

```
$DISCONNECT RAB_ADDR
```

BLISS-32 Example:

```
$DISCONNECT (RAB=RAB_ADDR);
```

$DS_ENDPASS

The End-of-Pass system service is used to indicate to the VDS that a program pass has been completed. This service must be included in the initialization code of every program. Refer to Section 3.5, Initialization Code, for an explanation of how the $DS_ENDPASS macro is to be used.

MACRO-32 Format:

```
$DS_ENDPASS_x;
```

BLISS-32 Format:

```
$DS_ENDPASS;
```

Return Status:

This service does not return a status code.

Examples:

MACRO-32 Example:

```
        $DS_GPHARD_S -
                LOG_UNIT, PTABLE_ADDR   ; Get P-table for next unit.
        CMPL    R0, DS$_ERROR           ; If all units done,
        BNEQL   10$                     ; then
        $DS_ENDPASS_S                   ;    declare end-of-pass
 10$:                                   ; else continue.
```

BLISS-32 Example:

```
IF $DS_GPHARD                           ! Get P-table for next unit.
  (DEVNAM = .LOGUNIT,                   !
   RETADR = PTABLE_ADDR)                !
EQL DS$_ERROR THEN $DS_ENDPASS;         ! If all units done,
                                        ! declare end-of-pass.
```

$DS_ERRDEV
$DS_ERRHARD
$DS_ERRPREP
$DS_ERRSOFT
$DS_ERRSYS

The five error reporting system services are used to report to the program user any errors encountered by the program that relate to failures in the device being tested.

- The $DS_ERRDEV macro is used to report device-fatal errors.
- The $DS_ERRHARD macro is used to report hard errors.
- The $DS_ERRPREP macro is used to report device preparation errors.
- The $DS_ERRSOFT macro is used to report soft errors.
- The $DS_ERRSYS macro is used to report system-fatal errors.

The error types are discussed in Section 3.9, Reporting Errors.

The error reporting system services will:

1. Display a "header message" consisting of the program title, the pass, test, and subtest numbers, and the message specified by the error macro's "msgadr" parameter (see below).
2. Cause the message routine specified by the error macro's "prlink" parameter (see below) to be called.
3. Test the VDS control flags HALT and LOOP. If HALT is set, execution of the program will be stopped. If LOOP is set, a program loop will be established (see Section 3.10, Looping).

MACRO-32 Format:

```
$DS_ERRDEV_x [num], [unit], [msgadr], [prlink], [p1],...[p6]
$DS_ERRHARD_x [num], [unit], [msgadr], [prlink], [p1],...[p6]
$DS_ERRPREP_x [num], [unit], [msgadr], [prlink], [p1],...[p6]
$DS_ERRSOFT_x [num], [unit], [msgadr], [prlink], [p1],...[p6]
$DS_ERRSYS_x [num], [unit], [msgadr], [prlink], [p1],...[p6]
```

BLISS-32 Format:

```
$DS_ERRDEV    ([NUM=num],     [UNIT=unit],      [MSGADR=msgadr],
[PRLINK=prlink], [P1=p1],...,[P6=p6]);

$DS_ERRHARD    ([NUM=num],    [UNIT=unit],      [MSGADR=msgadr],
[PRLINK=prlink], [P1=p1],...,[P6=p6]);

$DS_ERRPREP    ([NUM=num],    [UNIT=unit],      [MSGADR=msgadr],
[PRLINK=prlink], [P1=p1],...,[P6=p6]);

$DS_ERRSOFT    ([NUM=num],    [UNIT=unit],      [MSGADR=msgadr],
[PRLINK=prlink], [P1=p1],...,[P6=p6]);

$DS_ERRSYS    ([NUM=num],     [UNIT=unit],      [MSGADR=msgadr],
[PRLINK=prlink], [P1=p1],...,[P6=p6]);
```

num

An identification number assigned to the error macro. If not specified, a number is automatically assigned to the error macro at program assembly time, according to the following algorithm.

1. The error number is set to 1 at the beginning of each test and each subtest.

2. Each time one of the error reporting macros is encountered at assembly time, the macro is assigned the current error number and then the error number is incremented.

3. If a macro call possesses an argument for the "num" parameter, that argument is used and the stored error number is not incremented.

Thus if the default value for "num" is always taken, each error reporting macro within a given subtest will have a unique error number assigned to it, and for each subtest the error macros will be numbered sequentially starting with 1.

If the $DS_ERRxxxx_L form of the macro is used, the "num" argument must be specified by using the $DS_ERRNUM macro.

unit

The logical unit number of the unit currently being tested.

msgadr

Address of a counted ASCII string, to be included in the error message header. Should contain a short description of the error.

prlink

Address of an error reporting routine. Routine must be delimited by $DS_BGNMESSAGE and $DS_ENDMESSAGE macros and must use $DS_PRINTB and $DS_PRINTX macros for output.

pl through p6

One to six optional parameters that may be used to pass arguments to the error reporting routine whose address is contained in "prlink."

Return Status:

None.

Notes:

1. Registers R2 through R11 are preserved so that the routine pointed to by "prlink" can expect to find them intact.

Error Reporting Routines:

The "prlink" parameter is used to link an error reporting routine to the error macro. The error reporting system service first displays the header message, including the text pointed to by "msgadr." Then the routine pointed to be "prlink" is called. The error reporting routine must have the following properties:

1. It is called with a CALLG instruction, so it must have an entry mask.

2. It must be delimited by the $DS_BGNMESSAGE and $DS_ENDMESSAGE macros.

3. It must print the second and third levels of the error message (see Section 3.9, Reporting Errors) by using the $DS_PRINTB and $DS_PRINTX macros, respectively.

4. It can reference arguments passed via the p1 through p6 parameters. These parameters can be accessed using the symbols defined by the $DS_ERRDEF macro.

5. It must contain all of the $DS_PRINTB and $DS_PRINTX macros that are used to display the error message. (If $DS_PRINTB and $DS_PRINTX macros are used to display an error message, they must be contained in an error reporting routine.)

Examples:

Note: These examples will produce error messages that adhere to the format indicated in Section 5.6.1, Error Message Formats.

MACRO-32 Example:

```
READERRMSG:     .ASCIC  /READ error while performing block transfer./
FMT_GOODBAD:    .ASCIC  \!/!/Device base address!_:!_!SL\-
                        \!/Address of expected buffer!_:!_!SL\-
                        \!/Address of received buffer!_:!_!SL\-
                        \!/Transfer size (words)!_:!_!SL\-
                        \!/Words in error!_:!_!SL\
FMT_DUMPHDR:    .ASCIC  \!/!/ADDRESS:!_EXPECTED:!_RECEIVED:!_XOR:!/!/\
FMT_DUMPBUF:    .ASCIC  \!SL!_!SL!_!SL!_!SL!/\

BUFDUMP:
        $DS_BGNMESSAGE <R2,R3,R4,R5>
        $DS_PRINTB_S    FMT_GOODBAD,-    ; Print second level of error msg.
                        ERR$_P5(AP),ERR$_P2(AP),ERR$_P1(AP), -
                        ERR$_P3(AP),ERR$_P4(AP)
        $DS_PRINTX_S    FMT_DUMPHDR      ; Print header for buffer dump
        CLRL    R2                       ; Clear error count
        MOVAL   REC_BUF,R3               ; Get addr. of received buffer
        MOVAL   EXP_BUF,R4               ; Get addr. of expected buffer
10$:                                     ; REPEAT
        CMPW    (R3),(R4)                ;   See if this word is good.
        BEQL    20$                      ;   IF word is bad
                                         ;   THEN
        INCL    R2                       ;     Count the error.
        XORL3   R3,R4,R5                 ;     XOR good and bad data
        $DS_PRINTX_S    FMT_DUMPBUF,-    ;     Print a line of 3rd msg. level
                        R3,(R4),(R3),R5
        CMPL    R2,#8                    ;     IF eight bad words displayed,
        BEQL    30$                      ;     THEN stop.
20$:    CMPL    (R3)+,(R4)+              ;   Increment buffer pointers.
        CMPL    R3,#REC_BUF_SIZE         ;   See if top of buffer reached.
        BRB     10$                      ; UNTIL entire buffer done.
30$:    $DS_ENDMESSAGE
        .
        .
        .
$DS_BGNTEST
        .
        .
        $DS_ERRHARD_S   UNIT=LOG_UNIT, MSGADR=READERRMSG, -
                        PRLINK=BUFDUMP, -
                        P1=REC_BUF,        P2=EXP_BUF, -
                        P3=#REC_BUF_SIZE, P4=ERR_COUNT, -
                        P5=DEV_BASE
        .
        .
$DS_ENDTEST
```

BLISS-32 Example:

```
LITERAL
        REC_BUF_SIZE = 256;
BIND
        READERRMSG =
                UPLIT (%ASCIC 'READ error performing block transfer.');

OWN
        REC_BUF : VECTOR [REC_BUF_SIZE,WORD],
        EXP_BUF : VECTOR [REC_BUF_SIZE,WORD],
        LOG_UNIT,ERR_COUNT,DEV_BASE;
        .
        .
        .
$DS_BGNMESSAGE (ROUTINE_NAME=BUFDUMP)

LOCAL
        ERRORS,
        XOR_VALUE;

BIND
        FMT_GOODBAD1=
                UPLIT (%ASCIC '!/!/Device base address!_:!_!SL'),
        FMT_GOODBAD2=
                UPLIT (%ASCIC '!/Address of expected buffer!_:!_!SL'),
        FMT_GOODBAD3=
                UPLIT (%ASCIC '!/Address of received buffer!_:!_!SL'),
        FMT_GOODBAD4=
                UPLIT (%ASCIC '!/Transfer size (words)!_:!_!SL'),
        FMT_GOODBAD5=
                UPLIT (%ASCIC '!/Words in error!_:!_!SL'),
        FMT_DUMPHDR=
                UPLIT (%ASCIC '!/!/ADDRESS:!_EXPECTED:!_RECEIVED:!_XOR:!/!/'),
        FMT_DUMPBUF=
                UPLIT (%ASCIC '!SL!_!SL!_!SL!_!SL!/');


! Display the second level of the error message.

        $DS_PRINTB (FMT_GOODBAD1,P5);
        $DS_PRINTB (FMT_GOODBAD2,P2);
        $DS_PRINTB (FMT_GOODBAD3,P1);
        $DS_PRINTB (FMT_GOODBAD4,P3);
        $DS_PRINTB (FMT_GOODBAD5,P4);

! Display the third level of the error message.
! First print the header and clear the error count.

        $DS_PRINTX (FMT_DUMPHDR);       ! Print header for buffer dump.
        ERRORS = 0;                     ! Clear error count
```

```
! Now compare the expected buffer with the received buffer.  Display all
! mismatches.  If more than eight errors are found, we can stop.

        INCR INDEX FROM 0 TO REC_BUF_SIZE DO
                BEGIN
                IF .REC_BUF [.INDEX] NEQ .EXP_BUF [.INDEX]
                THEN
                        BEGIN
                        ERRORS = .ERRORS + 1;
                        XOR_VALUE =
                                .REC_BUF [.INDEX] XOR .EXP_BUF [.INDEX];
                        $DS_PRINTX (FMT_DUMPBUF,
                                         REC_BUF [.INDEX],
                                         .EXP_BUF [.INDEX],
                                         .REC_BUF [.INDEX],
                                         .XOR_VALUE);
                        END;
                IF .ERRORS EQL 8 THEN EXITLOOP;
                END;

$DS_ENDMESSAGE;
        .
        .
        .
$DS_BGNTEST (TEXT='Read tests');
        .
        .
        .
        $DS_ERRHARD (UNIT=.LOG_UNIT,     MSGADR=READERRMSG,
                     PRLINK=BUFDUMP,     P1=REC_BUF,
                     P2=EXP_BUF,         P3=REC_BUF_SIZE,
                     P4=.ERR_COUNT,      P5=.DEV_BASE);
        .
        .
$DS_ENDTEST;
```

$FAO

$FAOL

The Formatted ASCII Output ($FAO) system service provides a means by which complex messages can be formatted into ASCII character strings. This macro can be used to:

- Insert variable character string data 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

The system service constructs an output string by referring to formatted ASCII output (FAO) directives contained in a "control string" and using those directives to operate on the contents of value parameters.

The $FAOL macro performs the exact same function as the $FAO macro, but allows the specification of an address of a parameter list instead of requiring each parameter to be listed explicitly in the macro call.

MACRO-32 Format:

```
$FAO_x ctrstr, [outlen], outbuf, [p1], [p2], ...[pn]
$FAOL_x ctrstr, [outlen], outbuf, prmlst
```

BLISS-32 Format:

```
$FAO (CTRSTR=ctrstr, [OUTLEN=outlen], OUTBUF=outbuf, [P1=p1],
[P2=p2], ...[Pn=pn]);

$FAOL (CTRSTR=ctrstr, [OUTLEN=outlen], OUTBUF=outbuf,
PRMLST=prmlst);
```

ctrstr

Address of a character string descriptor (see Section 4.3) pointing to the "control string." The control string contains a set of Formatted ASCII Output (FAO) directives. These directives are described in the notes of the $DS_PRINTx macros.

outlen

Address of a word to receive length of output string constructed by the service routine.

outbuf

Address of a character string descriptor (see Section 4.3) pointing to the output buffer. The fully formatted output string is placed in this buffer.

p1 through pn ($FAO only)

0 to 20 directive parameters, contained in longwords. Depending on the corresponding FAO directive, a parameter may be a value that is to be converted, the address of a string that is to be inserted, a length, or an argument count. Parameters are listed in the order they are referenced by the control string. If more than 20 parameters must be specified, use the $FAOL macro.

prmlst ($FAOL only)

Address of a list of longwords containing the directive parameters. The list may be of any length. It may be an already existing data structure from which certain values are to be extracted.

Return Status:

SS$_NORMAL

Service successfully completed.

SS$_BUFFEROVF

Service successfully completed, but the size of the output string was greater than the maximum allowed and was truncated (see notes).

SS$_BADPARAM

An invalid FAO directive was specified in the control string.

Notes:

1. If the formatted output string is to be displayed on the user's terminal, it is important to select the proper $DS_PRINTx macro to cause the message to be displayed. Refer to the description of the $DS_PRINTx macros.

Examples:

This example will create the following string:

```
VALUES 200 (DEC) 0000012C (HEX) -400 (SIGNED)
```

MACRO-32 Example:

```
FAODESC:            ;Descriptor for output buffer
         .LONG 80        ;Output buffer length
         .LONG FAOBUF    ;Address of buffer
FAOBUF: .BLKB 80        ;80-character buffer
FAOLEN: .BLKW 1         ;Word to receive length of output

CNTRL_STRING:
         .ASCID  /VALUES !UL (DEC) !XL (HEX) !SL (SIGNED)/
VAL1:    .LONG 200
VAL2:    .LONG 300
VAL3:    .LONG -400

         $FAO_S  CTRSTR=CNTRL_STRING, -
                 OUTBUF=FAODESC, -
                 OUTLEN=FAOLEN, -
                 P1=VAL1, P2=VAL2, P3=VAL3
```

BLISS-32 Example:

```
OWN
   FAOBUF   : VECTOR [80, BYTE],
   FAODESC  : VECTOR [2]
              INITIAL (80, FAOBUF),
   FAOLEN   : VECTOR [1, WORD],
   VAL1     : VECTOR
              INITIAL (200),
   VAL2     : VECTOR
              INITIAL (300),
   VAL3     : VECTOR
              INITIAL (-400);

BIND
   UPLIT    = (%ASCID 'VALUES !UL (DEC) !XL (HEX) !SL (SIGNED)');

         $FAO    (CTRSTR=CNTRL_STRING,
                 OUTBUF=FAODESC,
                 OUTLEN=FAOLEN,
                 P1=VAL1, P2=VAL2, P3=VAL3);
```

$GET

The Get a Record service of RMS is used to read a record from a file. The file must have been previously opened and connected with the $OPEN and $CONNECT services, respectively. Records may be read from the file sequentially or by the random-by-RFA method. These access methods are discussed in Section 3.15, File Management.

MACRO-32 Format:

```
$GET rab, [err], [suc]
```

BLISS-32 Format:

```
$GET (RAB=rab, [ERR=err], [SUC=suc]);
```

rab

Address of the RAB to be associated with the FAB describing the file to which connection is to be made. (The address of the FAB is in the RAB.)

err (user mode only)

Address of a routine to be executed on error return from the service.

suc (user mode only)

Address of a routine to be executed on successful return from the service.

Return Status:

Note: For further details on return status values, refer to the VAX-11 RMS Reference Manual.

RMS$_NORMAL

Service successfully completed.

RMS$_EOF

Attempt was made to read beyond end of file.

RMS$_FAB

The FAB block is invalid.

RMS$_IFI

The FAB's IFI field is invalid.

RMS$_ISI

The RAB's ISI field is invalid.

RMS$_RAB

The RAB block is invalid.

RMS$_RER

Read error. (The device driver's return status will be in the STV field of the RAB.)

RMS$_RFA

Invalid RFA was specified in random-by-RFA mode.

RMS$_RTB

Record retrieved was too big for the buffer provided, and was truncated.

Notes:

1. Table 4-6 lists the RAB fields used by the $GET service IN STANDALONE MODE. For user mode, refer to the VAX-11 RMS Reference Manual.

Table 4-6 RAB Fields Used by $GET (Standalone Mode)

Field Mnemonic	Field Name
Input:	
ISI	Internal stream identifier.
RAC	Record access mode.
RFA	Record's address. (Used only if RAC=RFA.)
ROP	Record-processing options.
UBF	User record area address.
USZ	User record area size.
Output:	
RBF	Record address.
RFA	Record's file address.
RSZ	Record size.
STS	Completion status code. (Also returned in RØ).
STV	Status value. (See Return Status, above.)

Examples:

MACRO-32 Example:

```
   $GET RAB_ADDR
```

BLISS-32 Example:

```
   $GET (RAB=RAB_ADDR);
```

$DS_GETBUF

The $DS_GETBUF macro is used to obtain buffer space. In standalone mode, the buffer space is allocated by the VDS from its free memory pool. In user mode, the VDS calls the VMS $EXPREG system service (see the VAX/VMS System Services Reference Manual for details).

The caller indicates the number of pages desired, and the service returns the low and high addresses of the space allocated.

When the program no longer needs the allocated buffer space, it can be returned to the free memory pool with the $DS_RELBUF macro.

MACRO-32 Format:

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

BLISS-32 Format:

```
$DS_GETBUF (PAGCNT=pagcnt,
            [RETADR=retadr],
            [PHYADR=phyadr],
            [REGION=region]);
```

pagcnt

Size (number of pages) desired for buffer.

retadr

Address of a two-longword array to receive the virtual addresses of the low and high buffer limits.

phyadr

Address of a two-longword array to receive physical addresses of low and high buffer limits. This parameter is only relevant in standalone mode.

region

Memory region from which caller wishes buffer space to be allocated. Values are

0: buffer allocated from P0 space. (Default.)

1: buffer allocated from P1 space.

2: buffer allocated from system space.

In standalone mode, this parameter is only relevant if memory management is turned on.

Return Status:

SS$_NORMAL

Buffer space allocated.

SS$_ACCVIO (user mode only)

The "retadr" array cannot be written by the caller.

SS$_EXQUOTA (user mode only)

The process exceeded its paging file quota.

SS$_ILLPAGCNT

Requested page count was less than 1.

SS$_INSFWSL (user mode only)

The process's working set limit is not large enough to accommodate the increased virtual address space.

SS$_VASFULL

Insufficient virtual address space is available to fulfill the buffer request. (See note 4.)

R0 = 0 (standalone mode only)

Illegal value was given for "region" parameter.

Notes:

1. If P1 space is requested in user mode, the "retadr" array will contain the allocated space's high address as its first element and the low address as its second element.

2. In standalone mode, buffer space will always be allocated as contiguous pages. If there is not a set of contiguous pages equal to the requested buffer size, then the SS$_VASFULL status will be returned.

3. In standalone mode, buffer space is allocated starting at the lowest available physical page.

4. If there are fewer pages available than the number requested, then the number of pages available will be allocated. The beginning and ending virtual addresses of this area will be placed in the "retadr" array.

Examples:

MACRO-32 Example:

```
   $DS_GETBUF_S     #10, BUFLIMITS     ;Ask for 10 pages.
```

BLISS-32 Example:

```
   $DS_GETBUF       !Ask for 5 pages in P1 space.
          (PAGCNT=5,
          RETADR=BUF_LIMITS,
          REGION=1);
```

$GETCHN

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 can be returned, if desired.

- The primary device characteristics
- The secondary device characteristics

In most cases, the two sets of characteristics are identical. However, there are three instances in which the primary and secondary characteristics are not the same:

1. If the device is associated with a mailbox, the primary characteristics are those of the device and the secondary characteristics are those of the 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 is a logical link in a network, the secondary characteristics describe the link.

If the diagnostic program is running in standalone mode, the primary and secondary characteristics will always be identical.

This service is not available to level 3 programs.

Note: It is recommended that all newly developed level 2R programs use the VMS $GETDVI service instead of $GETCHN, because of plans to remove support of $GETCHN from VMS. Refer the VAX/VMS System Services Reference Manual.

MACRO-32 Format:

```
$GETCHN chan, [prilen], [pribuf], [scdlen], [scdbuf]
```

BLISS-32 Format:

```
$GETCHN (CHAN=chan, [PRILEN=prilen], [PRIBUF=pribuf],
[SCDLEN=scdlen], [SCDBUF=scdbuf]);
```

chan

Number of the I/O channel assigned to the device.

prilen

Address of a word to receive the length of the primary characteristics.

pribuf

Address of a character string descriptor (see Section 4.3) pointing to buffer that will receive primary characteristics. The default is Ø, implying no buffer.

scdlen

Address of a word to receive the length of the secondary characteristics.

scdbuf

Address of a character string descriptor (see Section 4.3) pointing to buffer that will receive secondary characteristics. The default is Ø, implying no buffer.

Return Status:

SS$_BUFFEROVF

Service successfully completed. Device information overflowed the buffer(s), so information was truncated.

SS$_NORMAL

Service successfully completed.

SS$_ACVIO (user mode only)

A buffer descriptor cannot be read by the caller, 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 Ø or a number greater than the number of channels available.

SS$_NOPRIV (user mode only)

The specified channel is not assigned or was assigned from a more privileged access mode.

Notes:

1. In standalone mode, the device characteristics are placed into the specified buffer in the format illustrated in Figure 4-7.

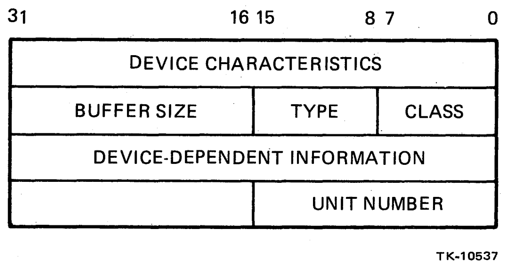

Figure 4-7 Device Characteristics Buffer (Standalone Mode)

Following the unit number is an ASCII string representing the device's generic name.

The "device characteristics" and "device dependent information" fields are the same as they are for user mode. Refer to the VAX/VMS I/O User's Guide for details.

2. In user mode, the device characteristics are placed into the specified buffers in the format detailed in the VAX/VMS I/O User's Guide.

3. Refer to the VAX/VMS System Services Reference Manual for privilege restrictions and other notes on the use of this service in user mode.

Examples:

MACRO-32 Example:

```
   CHANNUM:.WORD 0
   BUFFER:
           .LONG DIB$K_LENGTH
           .LONG BBUF
   BBUF:   .BLKB DIB$K_LENGTH
               :
               :
           $GETCHN_S CHANNUM,,BUFFER
```

BLISS-32 Example:

```
OWN
   CHANNUM : VECTOR [WORD],
   BBUF    : VECTOR [DIB$K_LENGTH, BYTE],
   BUFFER  : VECTOR [2]
             INITIAL (DIB$K_LENGTH, BBUF),
               :
               :
           $GETCHN (CHAN=.CHANNUM,,PRIBUF=BUFFER);
```

$DS_GETTERM

The Get Terminal Characteristics service can be used to obtain the type and characteristics of the user's terminal.

MACRO-32 Format:

```
$DS_GETTERM_x  termchar
```

BLISS-32 Format:

```
$DS_GETTERM    (TERMCHAR=termchar);
```

termchar

Address of a quadword to receive the terminal characteristics. See Note 1 for format of the characteristics.

Return Status:

SS$_NORMAL

Service successfully completed.

Notes:

1. The terminal characteristics are returned in a quadword with fields in the following format:

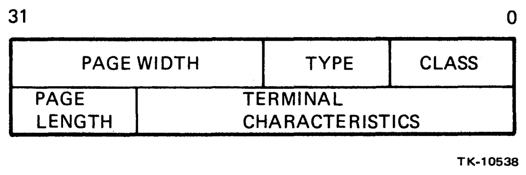

Figure 4-8 Format of Terminal Characteristics

Values for the "type" field and "terminal characteristics" are defined by the $TTDEF macro of VMS.

Note: In standalone mode, only the "type" and "terminal characteristics" fields are supplied. For terminal characteristics, only TT$M_SCOPE is provided. In user mode, all fields and all terminal characteristics are supplied.

Examples:

MACRO-32 Example:

```
TERM_INFO:      .BLKQ   1
                  :
                  :
        $DS_GETTERM_S   TERM_INFO
```

BLISS-32 Example:

```
OWN
        TERM_INFO : VECTOR [2];
                    :
                    :
        $DS_GETTERM (TERM_CHAR=TERM_INFO);
```

$GETTIM

The Get Time system service furnishes the current system time in 64-bit format. The time can be converted to ASCII by using the $ASCTIM service.

MACRO-32 Format:

```
$GETTIM timadr
```

BLISS-32 Format:

```
$GETTIM (TIMADR=timadr);
```

timadr

Address of a quadword that is to receive the current time in 64-bit format.

Return Status:

SS$_NORMAL

Service successfully completed.

SS$_ACCVIO (user mode only)

The quadword to receive the time cannot be written by the caller.

Examples:

MACRO-32 Example:

```
$GETTIM_S TIME
```

BLISS-32 Example:

```
$GETTIM (TIMADR=TIME);
```

$DS_GPHARD

The Get Hardware Parameter Table system service will provide the caller with the address of the p-table for the logical unit specified. The p-table's contents can then be accessed by the caller. The macro is used in a diagnostic program's initialization code, discussed in Section 3.5.

MACRO-32 Format:

```
$DS_GPHARD_x devnam, adrloc
```

BLISS-32 Format:

```
$DS_GPHARD_x (UNIT=devnam, RETADR=adrloc);
```

devnam

The logical unit number of the device whose p-table is being requested. Minimum value is Ø. Maximum value is determined by VDS, depending on the number of selected device units testable by caller. (See notes.)

adrloc

Address of longword to receive p-table base address.

Return Status:

DS$_NORMAL

Service successfully completed.

DS$_ERROR

The argument list does not contain exactly two arguments.

The specified logical unit number is too large.

Notes:

1. If "devnam" was initialized to Ø and incremented after each issuance of the $DS_GPHARD macro, then the DS$_ERROR return status simply means that the p-tables for all selected, testable device units have been referenced. "Devnam" should be reinitialized to Ø. See Section 3.5, Initialization Code, for details.

Examples:

MACRO-32 Example:

```
INCL    LOG_UNIT
$DS_GPHARD_S -
        LOG_UNIT, P_TABLE
```

BLISS-32 Example:

```
LOG_UNIT = .LOG_UNIT + 1;
$DS_GPHARD (UNIT=.LOG_UNIT, RETADR=P_TABLE);
```

$DS_HELP

The Display Help Text service can be used to display text contained in a help file. Help files are described in Chapter 5. This service is functionally identical to the VDS command HELP.

MACRO-32 Format:

```
$DS_HELP_x keylst
```

BLISS-32 Format:

```
$DS_HELP (KEYLST=keylst);
```

keylst

Address of a character string descriptor (see Section 4.3) that points to a list of help file keywords. This list is exactly equivalent to the keywords that would be included as parameters to the HELP command (see the VAX Diagnostic Supervisor User's Guide). To reference the help file EVXYZ.HLP, for diagnostic program EVXYZ, the first keyword in the list must be 'EVXYZ'.

Return Status:

The return status may be any status that may be returned from the $OPEN, $CONNECT, $READ, or $CLOSE services of RMS. Refer to descriptions of these services.

Examples:

MACRO-32 Example:

```
KEYSTRING:      .ASCID  /EVXYZ MANUAL OPTIONS/
                        :
                        :
                $DS_HELP_S      KEYSTRING
```

BLISS-32 Example:

```
BIND KEYSTRING = UPLIT (%ASCID 'EVXYZ MANUAL OPTIONS');
                 :
                 :
$DS_HELP (KEYLST=KEYSTRING);
```

$HIBER

The Hibernate system service allows a diagnostic program to make itself inactive. A hibernating program can be interrupted to process asynchronous events. After the diagnostic program's event handler has been executed, the program will be returned to its state of hibernation. This state will remain in effect until the program is awakened with the $WAKE system service.

MACRO-32 Format:

```
$HIBER_S
```

(Note: Only the _S form of the macro is supported.)

BLISS-32 Format:

```
$HIBER;
```

Return Status:

SS$_NORMAL

Service successfully completed.

Notes:

1. In standalone mode, the only way for a hibernating program to be awakened is for an event handler (for example, an AST routine or interrupt service routine) to call the $WAKE service.

2. In user mode, a hibernating process may be awakened by another process. Refer to the VAX/VMS System Services Reference Manual for details.

Examples:

MACRO-32 Example:

```
$HIBER_S
```

BLISS-32 Example:

```
$HIBER;
```

$DS_INITSCB

The Initialize System Control Block system service will load the VDS default values into all vectors within the SCB. It can be used to restore VDS exception and interrupt handling to all vectors if the diagnostic program has previously defined its own handlers using the $DS_SETVEC service.

This system service is only available to level 3 diagnostic programs.

MACRO-32 Format:

```
$DS_INITSCB_x
```

BLISS-32 Format:

```
$DS_INITSCB ();
```

Return Status:

SS$_NORMAL

Service successfully completed.

Examples:

MACRO-32 Example:

```
$DS_INITSCB_S;
```

BLISS-32 Example:

```
$DS_INITSCB;
```

$DS_INLOOP

The $DS_INLOOP program control macro can be used to determine if a program loop is being executed. Program looping is discussed in Section 3.1Ø.

MACRO-32 Format:

```
$DS_INLOOP_x
```

BLISS-32 Format:

```
$DS_INLOOP;
```

Return Status:

DS$_NORMAL

A program loop is being executed.

DS$_ERROR

A program loop is not being executed.

Examples:

MACRO-32 Example:

```
$DS_INLOOP_S
```

BLISS-32 Example:

```
$DS_INLOOP;
```

$DS_LOAD

The $DS_LOAD system service can be used for reading a file into a buffer area. This service may be employed when the full range of processing options provided by RMS is not needed. (The $DS_LOAD service uses RMS to implement its functionality.)

MACRO-32 Format:

```
$DS_LOAD_x file, default, length, address, retlen, retrec,
[vbn]
```

BLISS-32 Format:

```
$DS_LOAD (FILE=file, DEFAULT=default, LENGTH=length,
ADDRESS=address, RETLEN=retlen, RETEC=retrec, [VBN=vbn]);
```

file

Address of a quadword descriptor (see Section 4.3) describing a character string that represents the name of the file to be loaded. The filename format is:

```
NODE::DEV:[DIRECTORY]FILENAME.EXT;VER.
```

If any fields of the filename are missing, they will be filled in with fields specified by the "default" parameter.

default

Address of a quadword descriptor (see Section 4.3) describing a character string that represents the default fields for the filename.

length

Size, in bytes, of the buffer that will receive the file.

address

Address of the buffer that will receive the file.

retlen

Address of longword to receive the total length of the file.

retrec

Address of a longword to receive RMS file attributes of the file. The first word of the longword will contain the FAB MRS (maximum record size) field. The third byte will contain the FAB RFM (record format) field. The fourth byte will contain the FAB FSZ (fixed header size) field. Refer to the discussion of the $FAB macro for descriptions of these fields.

vbn

Virtual block number. This is the number of the first virtual block to be read. The default value is 1, which will cause reading to begin with the first block of the file.

Return Status:

The $DS_LOAD service can return any of the statuses associated with the $OPEN, $CONNECT, $READ, $DISCONNECT, or $CLOSE services of RMS. Refer to the descriptions of these services for lists of return statuses.

Examples:

MACRO-32 Example:

```
NAMEDESC:           ;Filename descriptor
        .LONG 0             ;Store filename string length here.
        .LONG BUFF          ;Address of filename string
BUFF:   .BLKB 30            ;Store filename here.

DEFDESC:            ;Default filename string descriptor
        .ASCID  /.EXE;0/

BUF_SIZE = 512
BUFFER: .BLKB   BUF_SIZE
FILE_LENGTH:
        .LONG 0
FILE_ATTR:
        .LONG 0
           :
           :
        $DS_LOAD_S NAMEDESC,DEFDESC,#BUF_SIZE, -
                        BUFFER,FILE_LENGTH,FILE_ATTR
```

BLISS-32 Example:

```
LITERAL
  BUF_SIZE = 512;
OWN
  BUFFER    : VECTOR [BUF_SIZE, BYTE],
  BUFF      : VECTOR [30, BYTE],      ! Store filename here.
  NAMEDESC: VECTOR [2]                ! Filename descriptor
            INITIAL (0,      ! Store filename string length here.
                    BUFF), ! Address of filename string
  FILE_LENGTH : VECTOR,
  FILE_ATTR   : VECTOR;

BIND
  DEFDESC =
              (%ASCID '.EXE;0');! Default filename string descriptor

            :
            :
  $DS_LOAD (FILE=NAMEDESC, DEFAULT=DEFDESC, LENGTH=BUF_SIZE,
        ADDRESS=BUFFER, RETLEN=FILE_LENGTH, RETREC=FILE_ATTR);
```

$DS_MMON

$DS_MMOFF

The Turn Memory Management On (DS$_MMON) and Turn Memory Management Off (DS$_MMOFF) system services are provided for enabling and disabling the memory management hardware in standalone mode.

Only level 3 diagnostic programs may turn memory management on or off. If a level 3 program turns memory management on or off, it MUST use these services to do so.

Memory management is discussed in Section 3.13, Memory Management and Allocation.

MACRO-32 Format:

```
DS$_MMON_x
DS$_MMOFF_x
```

BLISS-32 Format:

```
DS$_MMON ();
DS$_MMOFF ();
```

Return Status:

SS$_WASCLR

Service successfully completed. Memory management was previously disabled.

SS$_WASSET

Service successfully completed. Memory management was previously enabled.

DS$_WARNING (used with MMOFF only)

The $DS_MMOFF macro was issued, but memory management was not disabled because a SET MM ON user command had previously been issued (see the <u>VAX Diagnostic Supervisor User's Guide</u>).

Notes:

1. The user command SET MM ON has precedence over the $DS_MMOFF macro. Thus a program cannot shut off memory management if the user has turned it on.

Examples:

MACRO-32 Example:

```
$DS_MMON_S                  ;Turn on memory management.
```

BLISS-32 Example:

(This example illustrates the case of a program that cannot execute if memory management is enabled. If the program cannot turn memory management off, it aborts.)

```
! Turn off memory management.  If the user has turned it on,
! call routine to report the problem, then abort the program.

$DS_MMOFF ();
IF DS$_WARNING
THEN
        BEGIN
        REPORT_MM_ON ();
        $DS_ABORT ();
        END;
```

$OPEN

The Open Existing File service of RMS is used to make a file available for processing. Opening a file is the first step in processing the information within the file. This service uses parameters within the FAB to determine which file to open and what access attributes to assign to the file.

MACRO-32 Format:

```
$OPEN   fab, [err], [suc];
```

BLISS-32 Format:

```
$OPEN   (FAB=fab, [ERR=err], [SUC=suc]);
```

fab

Address of the FAB. The FAB is declared using the $FAB macro.

err (user mode only)

Address of routine to execute on error return from open service.

suc (user mode only)

Address of routine to execute on successful return from open service.

Return Status:

Note: For further details on return status values, refer to the VAX-11 RMS Reference Manual.

RMS$_NORMAL

Service successfully completed.

RMS$_ACC

Error accessing file.

RMS$_DME

Dynamic memory exhausted. Insufficient dynamic memory available.

RMS$_DEV

Bad device specification.

RMS$_FAB

Error in FAB.

RMS$_FNF

File not found.

RMS$_FNM

Bad file name.

RMS$_ORG

Invalid file organization. In standalone mode, file organization must be sequential.

RMS$_RER

File read error.

Notes:

1. Table 4-7 lists the FAB fields used by the $OPEN service IN STANDALONE MODE. For user mode, refer to the VAX-11 RMS Reference Manual.

Table 4-7 FAB Fields Used by $OPEN (Standalone Mode)

Field Mnemonic	Field Name
Input:	
DNA	Default file specification string address.
DNS	Default file specification string size.
FAC	File access.
FNA	File specification string address.
FNS	File specification string size.
FOP	File processing options.
FSZ	Fixed control area size; unit record devices only.
IFI	Internal file indentifier (must be Ø).
RAT	Record attributes (unit record devices only).
RFM	Record format; unit record devices only.
XAB	Extended attribute block address.
Output:	
ALQ	Allocation quantity; contains the highest numbered block allocated to the file.
BLS	Block size.
DEV	Device characteristics.
FSZ	Fixed control area size; applies only to "variable with fixed length" control records
IFI	Internal file identifier.
MRS	Maximum record size.
ORG	File organization.
RAT	Record attributes.
RFM	Record format.
STS	Completion status code (also returned in RØ).

Examples:

MACRO-32 Example:

```
    $OPEN FAB_BLOCK
```

BLISS-32 Example:

```
    $OPEN (FAB=FAB_BLOCK);
```

$DS_PARSE

The Parse Command String system service can be used in a diagnostic program for which a unique command language has been defined (see Section 3.12.2.2, Prompting the User). This service will parse a command string by searching a predefined command tree, looking for a matches between the command string and nodes of the tree. Every time a match is found, the service will dispatch to an "action routin." Details are presented in the notes below.

MACRO-32 Format:

```
$DS_PARSE_x bufadr, tree, action
```

BLISS-32 Format:

```
$DS_PARSE (BUFFER=bufadr, TREE=tree, ACTION=action);
```

bufadr

Address of a quadword descriptor (see Section 4.3) pointing to the command string.

tree

Address of the tree of valid commands. This tree should be defined by using the $DS_CLI macro.

action

Address of action routine. See notes for routine format.

Return Status:

SS$_NORMAL

Service successfully completed.

DS$_ERROR

While traversing the command tree, an error node (defined by CLI$K_ERROR, see $DS_CLI description) was encountered. In other words, an illegal command string was specified.

Notes:

1. The command string to be parsed should be fetched from the user by issuing the $DS_ASKSTR macro.

2. The $DS_PARSE system service will traverse the parse tree until a CLI$K_EXIT or a CLI$K_ERROR code is encountered (see DS$_CLI description), at which point it will return to the caller.

3. As the tree is traversed, the action routine will be called each time there is a match between the contents of the current node of the tree and the input stream. If a match is found, the action routine is called and then the next node in the current path is checked. Otherwise, a branch to the node specified by the "miss" parameter of the $DS_CLI macro occurs.

Action Routines:

Parameters will be passed to the action routine as follows:

R0 - Will contain action code specified for current node in parse tree.

R7 - Will contain current value of pointer used by VDS when traversing tree.

R8 - Will point to next unparsed character in the input string.

R9 - Will contain number of unparsed characters remaining in input string.

R10 and R11 - Will contain quadword value of last numeric string read from input buffer.

Generally, the programmer will specify a unique action code for each tree node, using the $DS_CLI macro. Sometimes a "null" action code is used, because it is not necessary for the action routine to do anything for nodes which do not completely identify a command, parameter, or qualifier. In other words, it is usually necessary to perform an action only when the parser is sure it has found something recognizable.

When the action routine is called, the action code is passed in RØ. The action routine can thus use a MACRO-32 CASE instruction or a BLISS-32 CASE expression, or some other means, to dispatch to a unique subroutine for each code. These subroutines will often just set bits in a bitmap indicating what command, command parameter, or command qualifier has been parsed. When the entire command string has been parsed, a command dispatching routine can be called. This dispatcher can examine the bitmap to determine which command processing routine to call.

An example action routine corresponding to the sample parse tree defined in the description of the $DS_CLI macro (earlier in this chapter) would be as follows:

```
ACTION_RTN::
        CASEL   RØ, #Ø, #8
1Ø$:
                .WORD   ACT_NO_ACTION-1Ø$
                .WORD   ACT_ADD-1Ø$
                .WORD   ACT_BAKE-1Ø$
                .WORD   ACT_BEAT-1Ø$
                .WORD   ACT_MILK-1Ø$
                .WORD   ACT_SALT-1Ø$
                .WORD   ACT_SUGAR-1Ø$
                .WORD   ACT_ILLCMD-1Ø$
                .WORD   ACT_BADARG-1Ø$

ACT_NO_ACTION:
        RSB

ACT_ADD:
        BISB    #1@ADD, CMD_BLOCK
        RSB

ACT_BAKE:
        BISB    #1@BAKE, CMD_BLOCK
        RSB

ACT_BEAT:
        BISB    #1@BEAT, CMD_BLOCK
        RSB

ACT_MILK:
        BISB    #1@MILK, PARAM_BLOCK
        RSB

ACT_SALT:
        BISB    #1@SALT, PARAM_BLOCK
        RSB
```

```
ACT_SUGAR:
        BISB    #1@SUGAR, PARAM_BLOCK
        RSB

ACT_ILLCMD:
        BISB    #1@ILLCMD, ERROR_BLOCK
        RSB

ACT_BADARG:
        BISB    #1@BADARG, ERROR_BLOCK
        RSB
```

Examples:

MACRO-32 Example:

This example fetches a command string, attempts to parse the string, and then either calls a command dispatcher or an error handler.

```
$DS_ASKSTR_S -                          ; Prompt user for input string.
        MSGADR=PROMPT_MSG, -
        BUFADR=STRING_BUF
CMPL    R0, SS$_NORMAL                  ; If error occurred
BNEQ    ASK_ERRHNDLR                    ; then go to error handler

MOVZBL  STRING_BUF, CMD_BUFFER          ; Put string length and string
MOVAL   STRING_BUF+1, CMD_BUFFER+4      ; address in quadword descriptor

$DS_PARSE_S -                           ; Parse the input string.
        BUFADR=CMD_BUFFER, -
        TREE=TREE_ROOT,-
        ACTION=ACTION_RTN
CMPL    R0, SS$_NORMAL                  ; If unsuccessful parse
BNEQ    PARSE_ERRHNDLR                  ; then go to error handler

BSBW    CMD_DISPATCHER                  ; Good parse - process command.
```

$DS_PRINTB
$DS_PRINTF
$DS_PRINTS
$DS_PRINTX

The Format and Print ASCII Message system services provide a means by which complex messages can be formatted into ASCII character strings and displayed on the user terminal. The macros that call these services are commonly referred to as the "print" macros. These macros can be used to

- Insert variable character string data into an output string
- Convert binary values into the ASCII representations of their decimal, hexadecimal, or octal equivalents and substitute the results in an output string

The system services construct an output string by referring to formatted ASCII output (FAO) directives contained in a "control string" and using those directives to operate on the contents of value parameters.

Once the system service creates the output string, it is automatically displayed on the user terminal.

The $DS_PRINTB macro ("print basic error message") is used exclusively to display the second message level of error messages (see Section 3.9, Reporting Errors). Display of messages generated with this macro will be inhibited if either of the VDS control flags IE2 or IE3 is set (see the VAX Diagnostic Supervisor User's Guide).

The $DS_PRINTX macro ("print extended error message") is used exclusively to display the third message level of error messages (see Section 3.9, Reporting Errors). Display of messages generated with this macro will be inhibited if the VDS control flag IE3 is set (see the VAX Diagnostic Supervisor User's Guide).

The $DS_PRINTS macro ("print summary message") is used exclusively to display program summary messages (see Section 3.7, Summary Routine). Display of messages generated with this macro will be inhibited if the VDS control flag IES is set (see the VAX Diagnostic Supervisor User's Guide).

The $DS_PRINTF macro ("print forced message") is used to display informational messages that are not related to device errors. These messages are referred to as "forced" messages because they are printed regardless of the state of the flags which inhibit message displays (see the VAX Diagnostic Supervisor User's Guide).

MACRO-32 Format:

```
$DS_PRINTB_x format, [pØ], [p1], [p2], [p3], [p4], [p5], [p6],
[p7], [p8], [p9], [pa], [pb], [pc], [pd], [pe], [pf]

$DS_PRINTX_x format, [pØ], [p1], [p2], [p3], [p4], [p5], [p6],
[p7], [p8], [p9], [pa], [pb], [pc], [pd], [pe], [pf]

$DS_PRINTF_x format, [pØ], [p1], [p2], [p3], [p4], [p5], [p6],
[p7], [p8], [p9], [pa], [pb], [pc], [pd], [pe], [pf]

$DS_PRINTS_x format, [pØ], [p1], [p2], [p3], [p4], [p5], [p6],
[p7], [p8], [p9], [pa], [pb], [pc], [pd], [pe], [pf]
```

BLISS-32 Format:

```
$DS_PRINTB (format, [pØ], [p1], [p2], [p3], [p4], [p5], [p6],
[p7], [p8], [p9], [pa], [pb], [pc], [pd], [pe], [pf]);

$DS_PRINTX (format, [pØ], [p1], [p2], [p3], [p4], [p5], [p6],
[p7], [p8], [p9], [pa], [pb], [pc], [pd], [pe], [pf]);

$DS_PRINTF (format, [pØ], [p1], [p2], [p3], [p4], [p5], [p6],
[p7], [p8], [p9], [pa], [pb], [pc], [pd], [pe], [pf]);

$DS_PRINTS (format, [pØ], [p1], [p2], [p3], [p4], [p5], [p6],
[p7], [p8], [p9], [pa], [pb], [pc], [pd], [pe], [pf]);
```

format

Address of a counted ASCII string. This is the "control string," which consists of the fixed text of the output string plus FAO directives for formatting variable data. FAO directives are listed below. Variable data is passed in parameters pØ through pf.

pØ through pf

Ø to 16 directive parameters, contained in longwords. Depending on the corresponding FAO directive, a parameter may be a value that is to be converted, the address of a string that is to be inserted, a length, or an argument count. Parameters are listed in the order they are referenced by the control string.

Return Status:

SS$_NORMAL

Service successfully completed.

SS$_BUFFEROVF

Service successfully completed, but the size of the output string was greater than the maximum allowed and was truncated (see notes).

SS$_BADPARAM

An invalid FAO directive was specified in the control string.

Notes:

1. VDS stores the output string in an internal buffer as it is being created. This buffer can contain up to 512 characters. If the output string is greater than 512 characters, the string is truncated and the truncated message is displayed.

2. If it is necessary to format a message containing more than 16 parameters, it is possible to

   - Use several PRINT macros in succession, or

- Use the $FAO or $FAOL macros to format the message. The message should then be printed using the proper print macro (for example, PRINTX for a level 3 error message).

3. In MACRO-32, the $FAO_S macro form uses a PUSHL instruction for all parameters (p1 through pn) specified with the macro call. In other words, all arguments are assumed to be values, not addresses. Therefore, if an address is desired, precede the argument with a # character, or load the address into a register.

FAO Directives:

An FAO directive has the format

!DD

where ! indicates that the following character(s) are to be interpreted as an FAO directive.

DD is a one- or two-character FAO directive. A directive may require that a parameter be included in the parameter list of the macro call. Note: All directives must be specified in uppercase letters.

Optionally, a directive may include:

- A repeat count

  A repeat count is coded as !n(DD), where n is a decimal number indicating that the directive should be repeated for the next n parameters.

- An output field length

  An output field length is specified as !lengthDD, where "length" indicates the field length that the output resulting from the specified directive should have.

A directive may contain both a repeat count and a field length, as in !n(lengthDD).

Repeat counts and output field lengths may be specified as variables, by using a # in place of an absolute numeric value. If a # is specified for a repeat count, the next argument included in the macro call must contain the count. If a # is specified for an output field length, the next argument must contain the length value.

If an output field length is specified as a variable, and a repeat count is also specified (by variable or by value), then only one length parameter will be fetched from the argument list, and each output string generated by the repeat count will have that length.

A 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 ^X21). If a literal ! is required in the output string, the directive !! must be used.

Each character in the control string that is not part of an FAO directive is copied into the output string. Thus if a portion of the message being formatted is a nonvolatile character string, that string can be placed directly into the control string.

If an invalid FAO directive is encountered in the control string, creation of the output string ceases at that point and an error status is returned to the caller.

No tests are made to determine if the correct number of parameters have been included in the macro call. If fewer parameters have been specified than are referenced by the control string, the system service routine will continue to fetch parameters past the end of the parameter list.

Table 4-8 lists the FAO directives.

Table 4-9 summarizes how the length of each field in the output string is determined, if no field length has been specified.

Table 4-8 FAO Directives

Directive	Function	Parameter(s)
Character String Substitution:		
!AC	Inserts a counted ASCII string.	Address of the string; the first byte must contain the length
!AD	Inserts an ASCII string.	1) Length of string 2) Address of string
!AF	Inserts an ASCII string; Replaces all nonprintable ASCII codes with periods (.).	1) Length of string 2) Address of string
!AS	Inserts an ASCII string.	Address of quadword character string descriptor pointing to the string
Numeric Conversion (zero-filled):		
!OB !OW !OL	Converts a byte to octal. Converts a word to octal. Converts a longword to octal.	Value to be converted to ASCII representation
!XB !XW !XL	Converts a byte to hexadecimal. Converts a word to hexadecimal. Converts a longword to hex.	For byte or word conversion, FAO uses only the low-order byte or word of the longword parameter.
!ZB !ZW !ZL	Converts an unsigned decimal byte. Converts an unsigned decimal word. Converts an unsigned decimal longword.	
Numeric Conversion (blank-filled):		
!UB !UW !UL	Converts an unsigned decimal byte. Converts an unsigned decimal word. Converts an unsigned decimal longword.	Value to be converted to ASCII representation
!SB !SW !SL	Converts a signed decimal byte. Converts a signed decimal word. Converts a signed decimal longword.	For byte or word conversion, FAO uses only the low-order byte or word of the longword parameter

Table 4-8 FAO Directives (Cont)

Directive	Function	Parameter(s)
Output String Formatting:		
!/	Inserts new line (cr/lf).	None
!_	Inserts a tab.	
!^	Inserts a form feed.	
!!	Inserts an exclamation point.	
!%S	Inserts the letter S if most recently converted numeric value is not 1.	
!%T	Inserts the system time.	Address of a quadword time value to be converted to ASCII. If Ø is specified, the current system time is used.
!%D	Inserts the system date and time.	
!n< !>	Defines output field width of n. characters. All data and directives within delimiters are left-justified and blank-filled within the field.	None
!n*c	Repeats the specified character in the output string n times.	
Parameter Interpretation:		
!-	Reuses last parameter in the list.	None
!+	Skips next parameter in the list.	

Note: 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.

Table 4-9 FAO Field Lengths and Fill Characters

How FAO Determines Output Field Lengths and Fill Characters:

Conversion or Substitution Type	**Default Length of Output Field**	**Action When Explicit Output Field Length is Longer Then Default**	**Action When Explicit Output Field Length is Shorter Than Default**
Hexadecmal byte word longword	2 (zero-filled) 4 (zero-filled) 8 (zero-filled)	ASCII result is right-justified and blank-filled to the specified length.	ASCII result is truncated on the left.
Octal byte word longword	3 (zero-filled) 6 (zero-filled) 11 (zero-filled)	Hex or octal output is zero-filled to the default output field length, then blank-filled to specified length.	
Signed or unsigned decimal	As many characters as necessary	ASCII result is right-justified and blank-filled to the specified length.	Signed and unsigned decimal output fields are completely filled with asterisks (*).
Unsigned zero-filled decimal	As many characters as necessary	ASCII result is right-justified and zero-filled to the specified length.	
ASCII string substitution	Length of input character string	ASCII string is left-justified and blank-filled to the specified length.	ASCII string is truncated on the right.

MACRO-32 Example:

```
   FMT_ERRCOUNT:
      .ASCIC       ?!/!/BYTES TRANSFERRED:!SL!_BAD:!SL!/!/?
           :
           :
      $DS_PRINTB_S FMT_ERRCOUNT, 4(AP), ERR_CNT
```

BLISS-32 Example:

```
BIND

   FMT_ERRCOUNT =
           UPLIT  (%ASCIC '!/!/BYTES TRANSFERRED:!SL!_BAD:!SL!/!/');
       :
       :
   $DS_PRINTB (FMT_ERRCOUNT, .TOTAL, .ERR_CNT);
```

$DS_PRINTSIG

The Print Signal Array system service will format and print the contents of a signal array. Signal arrays are passed to condition handlers when exception conditions occur. Refer to Section 3.14.5, Condition Handling.

MACRO-32 Format:

```
$DS_PRINTSIG_G argptr
```

(Note: Only the _G form of the macro is supported.)

BLISS-32 Format:

```
$DS_PRINTSIG (ARGPTR=argptr);
```

argptr

Address of the signal array.

Return Status:

SS$_NORMAL

Service successfully completed.

SS$_RESIGNAL

The VDS does not support condition handling for the detected condition. The signal array will not be displayed. The following conditions will always result in this return status: SS$_PAGRDERR, SS$_FAIL, SS$_DEBUG, and SS$_ARTRES.

Examples:

These examples illustrate use of the macro within a condition handler. Condition handlers receive the signal array address as the first argument on the argument stack.

MACRO-32 Example:

```
$DS_PRINTSIG_G  @4(AP)                  ;Display signal array
```

BLISS-32 Example:

```
$DS_PRINTSIG (ARGPTR = .(.AP + 4));     !Display signal array
```

$DS_PROBE

The Probe Device Address system service of the VDS may be used to determine if a device resides at a particular physical address. The service is passed the address to be checked and the logical unit number of the device that is expected to be at that address, and it will return a status code indicating whether or not the address exists.

This service is only available to level 3 programs.

MACRO-32 Format:

```
$DS_PROBE_x address, length, unit
```

BLISS-32 Format:

```
$DS_PROBE (ADDRESS=address, LENGTH=length, UNIT=unit);
```

address

The physical address whose existence is to be determined.

length

Size of the location specified by "address." Valid values are 1 for byte, 2 for word, and 4 for longword.

unit

Logical unit number of the device expected to be at the specified address.

Return Status:

$SS_NORMAL

Service successfully completed.

$DS_ERROR

An invalid value was specified for "length".

The specified address does not exist, or the device existing at address does not respond.

Examples:

MACRO-32 Example:

This example probes devices on a MASSBUS controller.

```
        .
        .
   $DS_GPHARD_S -
           LOG_UNIT, PTABLE         ; Get p-table.
        .
        .
   MOVL    PTABLE, R3               ; Get p-table address.
   MOVL    B^HP$A_DEVICE(R3),R10    ; Get MBA controller register
                                    ; base address.
   CLRL    R11                      ; Init. controller register pointer
   $DS_PROBE_S -                    ; See if the drive unit exists.
           ADDRESS = (R10)[R11]     ;
           LENGTH  = #4             ;
           UNIT    = LOG_UNIT       ;
   $DS_BERROR ERR10                 ;
        .
        .
   (Continue)
        .
        .
ERR10:     (Report error - device not there.)
```

BLISS-32 Example:

```
    $DS_GPHARD (UNIT=.LOG_UNIT, RETADR=PTABLE);
    CONTROLLER_BASE = .PTABLE [HP$A_DEVICE];
    DEVICE_ADDR = .CONTROLLER_BASE;
    WHILE .DEVICE_ADDR LSS LAST_DEVICE DO
      BEGIN
      IF NOT $DS_PROBE (ADDRESS=.DEVICE_ADDR,
                        LENGTH=4, UNIT=.LOG_UNIT)
      THEN BEGIN ...Report error - drive not there... END
      ELSE DEVICE_ADDR = .DEVICE_ADDR + NEXT_DEVICE
      END;
```

$QIO

$QIOW

The Queue I/O Request system service ($QIO) initiates an I/O operation in user mode by queueing a request to an I/O channel. The channel must have been previously assigned with $ASSIGN service. Once the I/O request has been queued, control returns to the caller. Notification that the I/O operation has completed can be accomplished by one of three methods:

1. An AST routine can be caused to execute when I/O has completed.
2. The diagnostic program can specify that an event flag be set when I/O has completed.
3. The diagnostic program can specify that an I/O status block be filled in when I/O has completed.

These methods for notification of I/O completion are discussed in Section 3.12.1.1, I/O in User Mode.

The Queue I/O Request and Wait for Event Flag system service ($QIOW) combines the operations of the $QIO and $WAITFR (Wait for Single Event Flag) system services.

The $QIO and $QIOW services may not be used by level 3 programs.

MACRO-32 Format:

```
$QIO_x efn, chan, func, [iosb], [astadr], [astprm], [p1], [p2],
[p3], [p4], [p5], [p6]

$QIOW_x efn, chan, func, [iosb], [astadr], [astprm], [p1],
[p2], [p3], [p4], [p5], [p6]
```

BLISS-32 Format:

```
$QIO    (EFN=efn,    CHAN=chan,       FUNC=func,      [IOSB=iosb],
[ASTADR=astadr],  [ASTPRM=astprm],  [P1=p1],   [P2=p2], [P3=p3],
[P4=p4], [P5=p5], [P6=p6]);

$QIOW    (EFN=efn,    CHAN=chan,      FUNC=func,      [IOSB=iosb],
[ASTADR=astadr],  [ASTPRM=astprm],  [P1=p1],   [P2=p2], [P3=p3],
[P4=p4], [P5=p5], [P6=p6]);
```

efn

Number of the event flag that is to be set at request completion. Caution: If an event flag is not specified, the default is Ø. Since event flag Ø is used by the VDS, a nonzero value for this parameter must ALWAYS be specified, for both the $QIO and the $QIOW macros, whether or not the diagnostic program actually tests this flag as a means of determining that the I/O operation has completed.

chan

Number of the I/O channel assigned to the device to which the request is directed. Obtained by using the $ASSIGN macro.

func

Function code and modifier bits that specify the operation to be performed. An introduction to function codes is provided in Section 3.12.1.1, I/O in User Mode. Complete documentation of function codes is located in the VAX/VMS I/O User's Guide.

iosb

Address of a quadword I/O status block that is to receive final completion status. See "Synchronizing I/O Completion" in Section 3.12.1.1, I/O in User Mode.

astadr

Address of the entry mask of an AST routine to be executed when the I/O completes. The AST routine will execute at the access mode from which the $QIO macro was issued. See "Synchronizing I/O Completion" in Section 3.12.1.1, I/O in User Mode.

astprm

AST parameter to be passed to the AST routine. See Section 3.14.3.

p1 to p6

Optional device- and function-specific parameters. Refer to the VAX/VMS I/O User's Guide.

The first parameter may be specified as "p1" or as "p1v," depending on whether an address or a value is required, respectively. If the keyword is not used, "p1" is the default and the argument is considered to be an ADDRESS.

P2 through p6 are always interpreted as VALUES.

Return Status:

SS$_NORMAL

Service successfully completed. The I/O request packet was successfully queued.

SS$_ABORT

A network logical link was broken.

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.

SS$_DEVOFFLINE

The specified device is offline.

SS$_EXQUOTA

The process has exceeded its buffered I/O quota, direct I/O quota, or buffered I/O byte count 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.

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 Ø or a number larger than the number of channels available.

SS$_NOPRIV

The specified channel does not exist or was assigned to a more privileged access mode.

SS$_UNASEFC

The process is not associated with the cluster containing the specified event flag.

Notes:

1. See the VAX/VMS System Services Reference Manual for discussions of privilege restrictions, resource requirements, and other notes relating to the $QIO and $QIOW macros.

2. Two potential problems exist when the $QIOW service is used:

   - If the I/O device is malfunctioning, the event flag may never be set and service will never return to the diagnostic program.

   - If the I/O device is slow or overloaded, the restriction that control-Cs be checked at least every three seconds may be violated (see Section 3.14.6, Handling Control-Cs).

   It is therefore better for diagnostic programs to use the $QIO and $WAITFR services. Additionally, the $SETIMR service should be used to limit the amount of time in which the program will wait for the event flag, in case it never becomes set.

Examples:

MACRO-32 Example:

```
    $QIO_S  EFN=#1, -                 ;Event flag 1
            CHAN=TTCHAN1, -           ;Channel
            FUNC=#IO$_WRITEBLK,       ;Virtual write function
            P1=BUFADD,-               ;Buffer address
            P2=#BUFSIZE               ;Buffer size
```

BLISS-32 Example:

```
    IF NOT (STATUS=$QIOW (EFN=32, CHAN=.LOG_UNIT,
            FUNC=IO$_SETMODE OR IO$M_ATTAST,
            IOSB = SETMODE_IOSB, P1=ATNAST)
    THEN
            BEGIN
                    (Report error.)
            END;
```

$READ

The Read File service of RMS is used to read a specified number of bytes, starting at a block boundary, from a file. The file must have been opened and connected, using the $OPEN and $CONNECT services, respectively.

MACRO-32 Format:

```
$READ rab, [err], [suc]
```

BLISS-32 Format:

```
$READ (RAB=rab, [ERR=err], [SUC=suc]);
```

rab

Address of the RAB to be associated with the FAB describing the file to which connection is to be made. (The address of the FAB is in the RAB.)

err (user mode only)

Address of a routine to be executed on error return from the service.

suc (user mode only)

Address of a routine to be executed on successful return from the service.

Return Status:

Note: For further details on return status values, refer to the VAX-11 RMS Reference Manual.

RMS$_NORMAL

Service successfully completed.

RMS$_EOF

Attempt was made to read beyond end of file.

RMS$_FAB

The FAB block is invalid.

RMS$_IFI

The FAB's IFI field is invalid.

RMS$_ISI

The RAB's ISI field is invalid.

RMS$_RAB

The RAB block is invalid.

RMS$_RER

Read error. (The device driver's return status will be in the STV field of the RAB.)

Notes:

1. Table 4-1Ø lists the RAB fields used by the $READ service IN STANDALONE MODE. For user mode, refer to the VAX-11 RMS Reference Manual.

Table 4-10 RAB Fields Used by $READ (Standalone Mode)

Field Mnemonic	Field Name
Input:	
BKT	Bucket number. Must contain the virtual block number of the first block to be read.
ISI	Internal stream identifier.
UBF	User record area address. This is where the block will be stored.
USZ	User record area size. Indicates length of the transfer, in bytes.
Output:	
RBF	Record address.
RFA	Record's file address. Contains the virtual block number of the first block transferred.
RSZ	Record size. Indicates the actual number of bytes transferred.
STS	Completion status code. (Also contained in RØ.)
STV	Status value. (See Return Status, above.)

Examples:

MACRO-32 Example:

```
   $READ RAB=RAB_BLOCK
```

BLISS-32 Example:

```
   $READ (RAB=RAB_BLOCK);
```

$READEF

The $READEF macro is used to obtain the current status of all flags within an event flag cluster. Event flags are discussed in Section 3.14.2.

MACRO-32 Format:

```
$READEF_x efn, state
```

BLISS-32 Format:

```
$READEF (EFN=efn, STATE=state);
```

efn

Number of any event flag within the cluster to be read. A flag of number 1 through 31 specifies cluster Ø, and a flag of number 32 through 63 specifies cluster 1.

state

Address of a longword to receive the status of all event flags within the cluster.

Return Status:

SS$_WASCLR (user mode only)

Service successfully completed. The specified event flag is clear.

SS$_WASSET (user mode only)

Service successfully completed. The specified event flag was set.

SS$_NORMAL (standalone mode only)

Service successfully completed.

SS$_ACCVIO (user mode only)

The address specified in the "state" parameter could not be written by the caller.

SS$_ILLEFC

An illegal event flag number was specified.

SS$_UNASEFC

In user mode, indicates that the specified common event flag (see Section 3.14.2) has not been associated with the process issuing the $CLREF macro.

In standalone mode, indicates that an event flag from 64 through 127 was specified. These flags are not valid in standalone mode.

Examples:

MACRO-32 Example:

```
$READEF_S  3, FLAGS
```

BLISS-32 Example:

```
$READEF (EFN=3, STATE=FLAGS);
```

$DS_RELBUF

The $DS_RELBUF macro is used to deallocate buffer space that was previously obtained with the $DS_GETBUF macro. The pages deallocated will be the pages that were most recently allocated. In user mode, the VDS calls the VMS $CNTREG service (see the VAX/VMX System Services Reference Manual).

MACRO-32 Format:

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

BLISS-32 Format:

```
$DS_RELBUF (PAGCNT=pagcnt,
            [RETADR=retadr],
            [REGION=region]);
```

pagcnt

Size (number of pages) of buffer space to be deallocated.

retadr

Address of a two-longword array to receive virtual addresses of low and high limit of address space deallocated.

region

Memory region from which caller wishes buffer space to be deallocated. Values are

Ø: buffer allocated from PØ space. (Default.)

1: buffer allocated from P1 space.

2: buffer allocated from system space.

In standalone mode, this parameter is only relevant if memory management is turned on.

Return Status:

SS$_NORMAL

Buffer space deallocated.

SS$_ACCVIO (user mode only)

The "retadr" array cannot be written by the caller.

DS$_FRAGBUF (standalone mode only)

The deallocated space was not contiguous. This condition could only exist if the specified page count was greater the page count specified with the most recently issued $DS_GETBUF macro, since space is always allocated in contiguous chunks in standalone mode.

SS$_ILLPAGCNT

The specified page count was less than 1.

SS$_PAGOWNVIO

In user mode, indicates that a page in the specified range is owned by a more privileged access mode.

In standalone mode, indicates that an attempt was made to deallocate more pages than had been previously allocated with GETBUF macros.

Examples:

MACRO-32 Example:

```
BUF_LIMITS:
        .QUAD 0

        $DS_RELBUF #10, BUF_LIMITS         ;Release 10 pages.
```

BLISS-32 Example:

```
OWN
        BUF_LIMITS : VECTOR [2];

        $DS_RELBUF (PAGCNT=10, RETADR=BUF_LIMITS);
```

$SETAST

The Set AST Enable system service is used to enable and disable the delivery of ASTs to the diagnostic program.

MACRO-32 Format:

```
$SETAST_x enbflg
```

BLISS-32 Format:

```
$SETAST (ENBFLG=enbflg);
```

enbflg

AST enable indicator. A value of 1 enables AST delivery, while a value of Ø disables AST delivery.

Return Status:

SS$_WASCLR

Service successfully completed. AST delivery was previously disabled.

SS$_WASSET

Service successfully completed. AST delivery was previously enabled.

Notes:

1. For notes on enabling and disabling AST delivery in user mode, refer to the <u>VAX/VMS System Services Reference Manual</u>.

Examples:

MACRO-32 Example:

```
   $SETAST_S #1              ;Enable delivery of ASTs
```

BLISS-32 Example:

```
   $SETAST (ENBFLG=0);       !Disable delivery of ASTs
```

$SETEF

The Set Event Flag system service is used to set event flags. (Event flags are discussed in Section 3.14.2.)

MACRO-32 Format:

```
$SETEF_x efn
```

BLISS-32 Format:

```
$SETEF (EFN=efn);
```

efn

Number of the event flag to be set. In user mode, the number may be from 1 through 23 or from 32 through 127. In standalone mode, flags 1 through 64 may be used.

Return Status:

SS$_WASCLR

Service successfully completed. The specified flag was previously Ø.

SS$_WASSET

Service successfully completed. The specified flag was previously 1.

SS$_ILLEFC

An illegal event flag number was specified.

SS$_UNASEFC

In user mode, indicates that the specified common event flag (see Section 3.14.2) has not been associated with the process issuing the $SETEF macro.

In standalone mode, indicates that an event flag from 64 through 127 was specified. These flags are not valid in standalone mode.

Examples:

MACRO-32 Example:

```
    $SETEF #4         ;Set event flag number 4.
```

BLISS-32 Example:

```
    $SETEF (EFN=4); !Set event flag number 4.
```

$SETIMR

The Set Timer system service allows the caller to request that an event flag be set, and optionally that an AST be delivered, after a specified amount of time has elapsed.

It is possible to make a number of concurrent timer requests. The caller will be notified (via event flag and AST delivery) when each specified time interval has completed.

MACRO-32 Format:

```
$SETIMR_x efn, daytim, [astadr], [reqidt]
```

BLISS-32 Format:

```
$SETIMR     (EFN=efn,      DAYTIM=daytim,       [ASTADR=astadr],
[REQIDT=reqidt]);
```

efn

Number of the event flag to be set after the specified time has elapsed. Note: If not specified, defaults to event flag Ø, which will cause VDS errors.

daytim

Address of quadword containing expiration time. A positive value indicates an absolute time at which the timer is to expire. A negative value indicates an offset from the current time. In standalone mode, only negative values are allowed. (See notes for specifying time.)

astadr

Address of the entry mask of an AST routine to be called when the specified time interval expires. If not specified, defaults to Ø, indicating no AST routine is to be called.

reqidt

Identification number for the timer request. Default value is Ø. A unique number may be specified for each timer request, or the same number can be assigned to several related requests. This number can be specified with the $CANTIM macro to cancel all timer requests having the specified number. Also, if an AST routine is specified, the number will be passed to the AST routine as the AST parameter.

Return Status:

SS$_NORMAL

Service successfully completed.

SS$_ACCVIO (user mode only)

The expiration time cannot be read by the caller.

SS$_EXQUOTA

- In user mode:

  Timer entry quota or AST limit quota exceeded, or insufficient system dynamic memory to complete the request.

- In standalone mode:

  The interval clock is already in use and hence is unavailable to this system service.

SS$_ILLEFC

An illegal event flag number was specified.

SS$$_INSFMEM

Insufficient dynamic memory to allocate a timer queue entry.

SS$_UNASEFC

- In user mode:

  Indicates that the specified common event flag (see Section 3.14.2) has not been associated with the process issuing the CLREF macro.

- In standalone mode:

  Indicates that an event flag from 64 through 127 was specified. These flags are not valid in standalone mode.

DS$_NOTIMP (standalone mode only)

An absolute time value was specified for "daytim." Only offset time values are allowed in standalone mode.

DS$_IPL2HI (standalone mode only)

The current IPL is too high. The IPL must be less than 2.

Notes:

1. To create a valid argument for the "daytim" parameter, first specify the time as an ASCII string, then use the $BINTIM macro to convert the ASCII string into the quadword format required by the "daytim" parameter.

2. In user mode, if the specified absolute time has already passed, the timer expires at the next clock cycle (within 1Ø milliseconds).

3. Each time the interval clock interrupts, the queue of timer requests is scanned to determine if any of the specified time intervals have expired. In standalone mode, the clock has been set up to interrupt every 1Ø milliseconds when $SETIMR requests are being processed.

4. In standalone mode, do not attempt to use the $DS_WAITUS service while $SETIMR requests are still pending.

Examples:

MACRO-32 Example:

```
   DAYTIME:
           .QUAD 0          ;Store 64-bit time here.
                   :
   .ENTRY AST_RTN, ^M<R2,R3,R4>
                   :
           ; AST routine.
                   :
           RET
                   :
           $SETIMR_S #5, DAYTIME, AST_RTN
                   :
```

BLISS-32 Example:

```
   OWN
           DAYTIME : VECTOR [2];
               :
           $SETIMR (EFN=8, DAYTIM=DAYTIME);
               :
```

$DS_SETIPL

The Set Interrupt Priority Level system service is used to change the processor's interrupt priorty level (IPL).

Only level 3 diagnostic programs are allowed to change the processor's interrupt priority level. These programs may not change the IPL without using this macro.

MACRO-32 Format:

```
$DS_SETIPL_x level
```

BLISS-32 Format:

```
$DS_SETIPL (LEVEL=level);
```

level

The level to which the IPL is to be set.

Return Status:

SS$_NORMAL

Service successfully completed.

Examples:

MACRO-32 Example:

```
$DS_SETIPL_S #31                  ;Set IPL to 31 (decimal).
```

BLISS-32 Example:

```
$DS_SETIPL (LEVEL=31);            !Set IPL to 31 (decimal).
```

$DS_SETMAP

The Set Adapter Mapping system service of the VDS will set up the mapping registers of a bus adapter so that data will be transferred to or from the desired physical address space. The service may be used to set, clear, validate, or invalidate an adapter's mapping registers.

MACRO-32 Format:

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

BLISS-32 Format:

```
$DS_SETMAP      (UNIT=unit,      FUNC=func,      PHYADR=phyadr,
[MAPBAS=mapbas], [BYTCNT=bytcnt], [DATPTH=datpth]);
```

unit

Logical unit number of the device to be tested.

func

Function code indicating the function to be performed. Function codes are listed in Note 1.

phyadr

Address of a two-longword array containing physical addresses of beginning and ending of physical address space from which or to which data is to be transferred. Commonly, this is the "phyadr" array filled in by the $DS_GETBUF service. The value specified as the ending address is used to validate the "bytcnt" parameter.

mapbas

This argument is used to optionally select the first (lowest addressed) map register to be employed in mapping virtual program addresses to physical memory addresses. The service will start with the map register specified and set up (or clear) enough map registers to map the address range indicated by "phyadr".

For MASSBUS operation, the argument must be a value from Ø to 255 (decimal), where Ø selects the first map register, 1 selects the second, and so on. The MBA Virtual Address Register will be automatically set up to point to the specified map register.

For UNIBUS operation, the argument must be a value from Ø to 495 (decimal), where Ø selects the first map register, 1 selects the second, and so on.

The default value is Ø.

For descriptions of address translation in bus adapters, refer to the VAX Hardware Handbook.

bytcnt

Number of bytes composing a data transfer. For MASSBUS operation, the 2's complement of this value is stored in the MBA byte counter. Maximum value allowed is 65535 (decimal).

For both MASSBUS and UNIBUS operation, this value is used when setting up map registers -- enough pages are mapped to handle the number of bytes specified.

The default value is Ø. If the default is used, one page (512 bytes) is mapped.

datpth

Value indicating the UNIBUS data path. The default is Ø, indicating the direct data path. Values from 1 through 15 may be specified to select one of the buffered data paths. This field is ignored if the UNIBUS adapter does not support buffered data paths.

Return Status:

DS$_NORMAL

Service successfully completed.

DS$_ERROR

The specified logical unit number is too large.

DS$_IHWE

Initial hardware error. A hardware error was detected in the bus adapter before the specified function was performed. The function was not performed. Call the $DS_CHANNEL service, specifying the CHC$_STATUS function to determine the error type.

$DS_PROGERR

An invalid function code was specified.

The byte count specified is too large to be mapped starting at the specified map register. Lower the byte count or lower the starting map register number.

The byte count specified will not fit into the buffer limits indicated by "phyadr."

Notes:

1. Function Codes

   Following is a list of valid function codes. For MACRO-32, these codes are defined by the $DS_CHMDEF macro.

   - CHM$_INVALIDATE - Clear the "valid" bits for all map registers in the bus adapter to which the device unit specified by "unit" is attached.

   - CHM$_MFWDN - Set up map registers for a forward transfer according to "phyadr," "mapbas," and "bytcnt" parameters, and set the "valid" bit in each register used. Do not invalidate any registers. If MASSBUS, load MBA virtual address register and MBA byte counter.

   - CHM$_MFWDNO - Set up map registers for a forward transfer according to "phyadr," "mapbas," and "bytcnt" parameters, and set the "valid" bit in each register used. Do not invalidate any registers. Indicate that a byte offset transfer will be performed (UNIBUS only).

- CHM$_MFWDV - Invalidate all map registers. Set up map registers for a forward transfer according to "phyadr," "mapbas," and "bytcnt" parameters, and set the "valid" bit in each register used. If MASSBUS, load MBA virtual address register and MBA byte counter.

- CHM$_MFWDVO - Invalidate all map registers. Set up map registers for a forward transfer according to "phyadr," "mapbas," and "bytcnt" parameters, and set the "valid" bit in each register used. Indicate that a byte offset transfer will be performed (UNIBUS only).

- CHM$_MREVN - Set up map registers for a reverse transfer according to "phyadr," "mapbas," and "bytcnt" parameters, and set the "valid" bit in each register used. Do not invalidate any registers. If MASSBUS, load MBA virtual address register and MBA byte counter.

- CHM$_MREVNO - Set up map registers for a reverse transfer according to "phyadr," "mapbas," and "bytcnt" parameters, and set the "valid" bit in each register used. Do not invalidate any registers. Indicate that a byte offset transfer will be performed (UNIBUS only).

- CHM$_MREVV - Invalidate all map registers. Set up map registers for a reverse transfer according to "phyadr," "mapbas," and "bytcnt" parameters, and set the "valid" bit in each register used. If MASSBUS, load MBA virtual address register and MBA byte counter.

- CHM$_MREVVO - Invalidate all map registers. Set up map registers for a reverse transfer according to "phyadr," "mapbas," and "bytcnt" parameters, and set the "valid" bit in each register used. Indicate that a byte offset transfer will be performed (UNIBUS only).

- CHM$_NFWDN - Do not alter map register contents. If MASSBUS, load MBA virtual address register and MBA byte counter for forward transfer.

- CHM$_NREVN - Do not alter map register contents. If MASSBUS, load MBA virtual address register and MBA byte counter for reverse transfer.

Examples:

MACRO-32 Example:

```
   BUF_SIZE = 1024
   LOG_UNIT:         .BLKL 1
   BUFFER:           .BLKQ 1
                .
                .
           $DS_SETMAP_S LOG_UNIT, #CHM$_MFWDV, BUFFER,,#BUF_SIZE
```

BLISS-32 Example:

```
   LITERAL
           BUF_SIZE = 1024;
   OWN
           LOG_UNIT : VECTOR,
           BUFFER   : VECTOR [2];
                           .
                           .
           $DS_SETMAP (UNIT=.LOG_UNIT, FUNC=CHM$_MFWDV,
                           PHYADR=BUFFER, BYTCNT=BUF_SIZE);
```

$SETPRT

The Set Protection on Pages system service allows a program to change the protection code associated with one or more pages of virtual memory.

MACRO-32 Format:

```
$SETPRT inadr, [retadr], [acmode], prot, [prvprt]
```

BLISS-32 Format:

```
$SETPRT   (INADR=inadr,    [RETADR=retadr],    [ACMODE=acmode],
PROT=prot, [PRVPRT=prvprt]);
```

inadr

Address of a two-longword array containing the starting and ending virtual addresses of the pages for which the protection code is to be changed. Specifying the same value for the starting and ending addresses will cause the protection of one page to be changed. Only the virtual page number portion of the address is used; the low-order nine bits are ignored.

retadr

Address of a two-longword array to receive the starting and ending virtual addresses of the pages that had their protections changed. See Note 2.

acmode

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 result must be equal to or more privileged than the access mode of the owner of the pages being changed.

This parameter is ignored in standalone mode.

prot

New protection, in bits Ø through 3. Symbolic names for the various page protection codes are described by the $PRTDEF macro which is defined in LIB.MLB.

prvprt

Address of a byte to receive the protection previously assigned to the last page whose protection was changed. Useful if only one page was changed.

Return Status:

SS$_NORMAL

Service successfully completed.

SS$_ACCVIO

- User mode:
  - The input address array cannot be read by the caller, or the output address array or the byte to receive the previous protection cannot be written by the caller.
  - An attempt was made to change the protection of a nonexistent page.
- Standalone mode:

  The specified address range was in the reserved virtual address space (CØØØØØØØ to FFFFFFFF).

SS$_EXQUOTA (use mode only)

The process exceeded its paging file quota while changing a page in a read-only private section to a read/write page.

SS$_IVPROTECT (user mode only)

The specified protection code has a numeric value of 1 or is greater than 15.

SS$_LENVIO

In user mode, a page in the specified range is beyond the end of the program or control region.

In standalone mode, a page in the specified range is beyond the end of the program, control, or system region.

SS$_NOPRIV (user mode only)

A page in the specified range is in the system address space.

SS$_PAGOWNVIO (user mode only)

Page owner violation. An attempt was made to change the protection on a page owned by a more privileged access mode.

DS$_PROGERR (standalone mode only)

The specified address range was improperly formatted.

Notes:

1. In standalone mode, setting page protection is only meaningful if memory management has been enabled.
2. If an error occurs while changing page protections, the return array, if specified, will contain the start and ending address of the pages that were changed before the error occurred. If no pages were changed, the return address array will contain a -1.

Examples:

MACRO-32 Example:

```
ADDR_RANGE:     .BLKQ 1
                :
                :
        $SETPRT INADR=ADDR_RANGE, PROT=#PRT$C_UW
                :
```

BLISS-32 Example:

```
OWN
  ADDR_RANGE : VECTOR [2];
                 :
                 :
  $SETPRT (INADR=ADDR_RANGE, PROT=PRT$C_UW);
```

$DS_SETVEC

The Set Exception or Interrupt Vector system service is used to load an exception or interrupt vector with the address of a service routine.

Only level 3 diagnostic programs may use the $DS_SETVEC macro. Vector contents may not be changed by any means other than the use of this macro.

MACRO-32 Format:

```
$DS_SETVEC_x vector, srvadr, [code]
```

BLISS-32 Format:

```
$DS_SETVEC (VECTOR=vector, SRVADR=srvadr, [CODE=code]);
```

vector

The vector address, relative to the base of the System Control Block (SCB). Refer to the VAX Architecture Handbook for a list of vector addresses in the SCB. See Note 1.

srvadr

The address of a service routine which is to receive control when an exception or interrupt is delivered through the specified vector. The address must be on a longword boundary.

code

Used to indicate the stack on which the event is to be serviced.

Can be Ø or 1. (The default is Ø.)

- If Ø, service the event on the kernel stack unless already running on the interrupt stack, in which case service on the interrupt stack. Behavior of the processor is undefined for a "kernel stack not valid" exception with this code.

- If 1, service the event on the interrupt stack. If the event is an exception, raise the IPL to 1F(16).

The value specified for this parameter is loaded into bits <1:Ø> of the specified vector.

Return Status:

DS$_NORMAL

Service successfully completed.

DS$_IVADDR

Address specified for "srvadr" routine does not start on a longword boundary.

DS$_IVVECT

Address specified for "vector" is not a valid vector address.

DS$_ICBUSY

The caller specified the interval clock's vector, and the interval clock was already active.

Notes:

1. When setting device interrupt vectors, remember that the SCB is several pages long. The page on which a particular device interrupt vector resides is determined by both the bus adapter(s) to which the device is attached and the processor being used.

   For instance, to find the SCB offset for a particular UNIBUS device's vector address, read HP$W_VECTOR in the device's p-table, then OR this value with the contents of HP$W_VECTOR in the p-table associated with EACH adapter existing between the device and the processor. The number and type of adapter depend on the processor type. (The device's p-table contains the actual UNIBUS vector, and the adapters' p-tables contain relative offsets into the SCB for the bases of the vector areas for the adapters.)

It thus becomes obvious that referencing device vectors in the SCB will cause a diagnostic program to become processor-dependent. Using the $DS_CHANNEL service for I/O operations eliminates the need to load SCB vectors and thus keeps diagnostic programs processor- independent.

Examples:

MACRO-32 Example:

```
   $DS_SETVEC_S     VECTADDR, SERV_RTN
```

BLISS-32 Example:

```
   $DS_SETVEC (VECTOR=.VECTADDR, SRVADR=SERV_RTN, CODE=1);
```

$DS_SHOCHAN

The Show Channel Status system service of the VDS will display on the user's terminal the contents of internal bus adapter registers. This service should be used whenever the $DS_CHANNEL or $DS_SETMAP services detect adapter faults.

The display will consist of the name of each register, the mnemonic name of each bit field within the register, and the current value of each bit field.

This service is only available to level 3 diagnostic programs.

MACRO-32 Format:

```
$DS_SHOCHAN_x unit
```

BLISS-32 Format:

```
$DS_SHOCHAN (UNIT=unit);
```

unit

Logical unit number of device currently being tested. Adapter to which this unit is attached will be the one whose registers are displayed.

Return Status:

$DS_NORMAL

Service successfully completed.

$DS_ERROR

Logical unit number is too large.

Notes:

1. It may be useful to include the $DS_SHOWCHAN macro in an error reporting routine (refer to the error reporting macros, $DS_ERRxxxx).

Examples:

MACRO-32 Example:

```
    $DS_SHOCHAN_S LOG_UNIT    ;Display adapter status.
```

BLISS-32 Example:

```
    $DS_SHOCHAN (UNIT=.LOG_UNIT);
```

$DS_SUMMARY

The Print Summary system service will cause the diagnostic program's summary routine to be executed. Summary routines are discussed in Section 3.7. Note that the summary routine will also be executed when the $DS_ENDPASS service is called, if the requested number of program passes have been executed.

MACRO-32 Format:

```
$DS_SUMMARY_x
```

BLISS-32 Format:

```
$DS_SUMMARY;
```

Return Status:

No status returned.

Examples:

MACRO-32 Example:

```
$DS_SUMMARY_S
```

BLISS-32 Example:

```
$DS_SUMMARY;
```

$UNWIND

The Unwind Call Stack system service allows a condition handler to "unwind" the procedure call stack to a specified depth. "Unwinding" is the process of stepping through a specified number of call frames on the stack so that when the condition handler returns, the specified call frame will be used instead of the one placed on the stack when the condition handler was called. In other words, the normal execution flow is altered. Optionally, an address can be specified that will be placed in the return PC argument of the call frame that was stepped to.

For a further discussion of unwinding, refer to sections on condition handling in the VAX/VMS System Services Reference Manual.

MACRO-32 Format:

```
$UNWIND_x [depadr], [newpc]
```

BLISS-32 Format:

```
$UNWIND ([DEPADR=depadr], [NEWPC=newpc]);
```

depadr

Address of a longword indicating the depth to which the stack is to be unwound. A depth of Ø indicates the call frame that was active when the condition occurred (the frame that would normally be used when the condition handler returns), 1 indicates the caller of that frame, 2 indicates the caller of the caller of the frame, and so on. If the depth is specified as Ø or less, no unwind occurs and a successful status code is returned. If no value is specified for this parameter, the unwind is performed to the caller of the frame that established the condition handler.

newpc

Address to be given control when the unwind is complete. This value is placed in the return PC argument of the call frame that is stepped to. If no value is specified for this parameter, the return PC argument is not altered.

Return Status:

SS$_NORMAL

Service successfully completed.

SS$_ACCVIO (user mode only)

The call stack is not accessible to the 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 the specified number of frames.

SS$_NOSIGNAL

No signal for an exception condition is currently active.

SS$_UNWINDING

An unwind is already in progress.

Notes:

1. The actual unwind does not occur when the service is called. The service simply modifies the return addresses in the call frames so that when the condition handler returns, an "unwind" procedure is called from each frame that is being unwound.

Examples:

In this example, the $UNWIND will cause the return PC of the call frame created by the CALLS ROUTINE1 instruction to be replaced by OUTADDR, and the RET instruction on the condition handler will cause that call frame to be referenced.

MACRO-32 Example:

```
ROUTINE1:
                :
        CALLS   ROUTINE2
                :
        RET

ROUTINE2:
                :
        (condition handler is called.)
                :
        RET

COND_HNDLR:
                :
        MOVL    #1, DEPTH
        $UNWIND_S DEPTH, OUTADDR
                :
        RET

OUTADDR:
                :
```

BLISS-32 Example:

```
ROUTINE ROUTINE1 =
        BEGIN
            :
        CALLS   ROUTINE2
            :
        END;

ROUTINE ROUTINE2 =
        BEGIN
            :
        (condition handler is called.)
            :
        END;

ROUTINE COND_HNDLR =
        BEGIN
            :
        DEPTH = 1;
        $UNWIND_S DEPTH, ERRORS
            :
        END;

ROUTINE ERRORS =
        BEGIN
            :
        END;
```

$WAITFR

The $WAITFR macro calls a system service that will wait until a specified event flag is set before returning. Event flags are discussed in Section 3.14.2. If the specified flag is already set, the service routine returns immediately. Otherwise, control is not returned to the caller until the flag has been set.

MACRO-32 Format:

```
$WAITFR_x efn
```

BLISS-32 Format:

```
$WAITFR (EFN=efn);
```

efn

Number of the event flag to wait for.

Return Status:

SS$_NORMAL

Service successfully completed.

SS$_ILLEFC

An illegal event flag number was specified.

SS$_UNASEFC (user mode only)

In user mode, indicates that the specified common event flag (see Section 3.14.2) has not been associated with the process issuing the $SETEF macro.

In standalone mode, indicates that an event flag from 64 through 127 was specified. These flags are not valid in standalone mode.

Notes:

1. While the system service routine is waiting for the event flag to be set, ASTs can interrupt the service. Program control will return to the $WAITFR system service after execution of the AST routine has completed.

Examples:

MACRO-32 Example:

```
$WAITFR_S #4
```

BLISS-32 Example:

```
$WAITFR (EFN=5);
```

$DS_WAITMS

The Millisecond Wait system service is used to create a delay of a specified number of milliseconds. When the service routine is called, control is not returned to the caller until the requested amount of time has elapsed (unless an asynchronous event occurs that causes a routine containing a $CANTIM or $DS_CANWAIT macro to be executed; see Note 1).

MACRO-32 Format:

```
$DS_WAITMS_x time, [tag]
```

BLISS-32 Format:

```
$DS_WAITMS (TIME=time, [RETTIM=tag]);
```

time

Length of delay in time units. One time unit equals 10 milliseconds. Value must be greater than 0.

tag

Address of longword to receive amount of unused time, if delay was canceled before all requested time was used up (see note 1).

Return Status:

SS$_NORMAL

Service successfully completed.

DS$_PROGERR

An invalid value was specified for the "time" parameter.

SS$_EXQUOTA

- In user mode:

  Timer entry quota or AST delivery quota exceeded, or insufficient dynamic memory space.

- In standalone mode:

  The interval clock is already in use and hence is unavailable to this system service.

Notes:

1. If an asychronous event (AST delivery or hardware interrupt) occurs, and the routine handling the AST or interrupt issues a $CANTIM or $DS_CANWAIT macro, then the $WAITMS service will, on regaining program control after return from the event handler, store the unused delay time in the address specified by "tag" and return control to the caller.

Examples:

MACRO-32 Example:

```
$DS_WAITMS_S #100, TIME_LEFT
```

BLISS-32 Example:

```
$DS_WAITMS (TIME=200, RETTIM=TIME_LEFT);
```

$DS_WAITUS

The Microsecond Wait system service is used to create a delay of a specified number of microseconds. When the service routine is called, control is not returned to the caller until the requested amount of time has elapsed (unless an asynchronous event occurs which causes a routine containing a $CANTIM or $DS_CANWAIT macro to be executed; see Note 1).

This macro may only be used by level 3 diagnostic programs.

MACRO-32 Format:

```
$DS_WAITUS_x time, [tag]
```

BLISS-32 Format:

```
$DS_WAITUS (TIME=time, [RETTIM=tag]);
```

time

Length of delay in time units. One time unit equals 1Ø microseconds. Value must be greater than Ø.

tag

Address of longword to receive amount of unused time, if delay was canceled before all requested time was used up (see notes).

Return Status:

SS$_NORMAL

Service successfully completed.

DS$_PROGERR

An invalid value was specified for the "time" parameter.

SS$_EXQUOTA

- In user mode:

  Timer entry quota or AST delivery quota exceeded, or insufficient dynamic memory space.

- In standalone mode:

  The interval clock is already in use and hence is unavailable to this system service.

Notes:

1. If an asychronous event (AST delivery or hardware interrupt) occurs, and the routine handling the AST or interrupt issues a $CANTIM or $DS_CANWAIT macro, then the $DS_WAITUS service will, on regaining program control after return from the event handler, store the unused delay time in the address specified by "tag" and return control to the caller.

2. Do not attempt to use the $DS_WAITUS service if $SETIMR requests have been issued and are still pending.

Examples:

MACRO-32 Example:

```
$DS_WAITUS_S #50, TIME_LEFT
```

BLISS-32 Example:

```
$DS_WAITUS (TIME=40, RETTIM=TIME_LEFT);
```

$WAKE

The Wake system service reactivites a process that is in hibernation as a result of execution of the $HIBER system service.

MACRO-32 Format:

```
$WAKE_x [pidadr], [prcnam]
```

BLISS-32 Format:

```
$WAKE ([PIDADR=pidadr], [PRCNAM=prcnam]);
```

pidadr (user mode only)

Address of a longword containing the process indentification of the process to be awakened.

prcnam (user mode only)

Address of a character string descriptor (see Section 4.3) pointing to the process name string.

Refer to the VAX/VMS System Services Reference Manual for details on the interpretation of these parameters.

Return Status:

SS$_NORMAL

Service successfully completed.

SS$_ACCVIO (user mode only)

The name string or string descriptor cannot be read by the caller, or the process id number cannot be written by the caller.

SS$_IVLOGNAM

The process name string is invalid.

SS$_NONEXPR

Warning. The specified process does not exist, or an invalid process id was specified.

SS$_NOPRIV

The caller's process does not have the privilege required for waking the specified process.

Notes:

1. In standalone mode, the only meaningful use of this macro is to place it in an event handler that will be executed while the diagnostic program is in hibernation. This will awaken the program so that it may continue executing.

Examples:

MACRO-32 Example:

```
$WAKE_S
```

BLISS-32 Example:

```
$WAKE ();
```

$WFLAND

The $WFLAND macro calls a system service that will wait until a specified group of event flags is set before returning. Event flags are discussed in section 3.14.2. All of the event flags must be in the same event flag cluster. If the specified flags are already set, the service routine returns immediately. Otherwise, control is not returned to the caller until all specified flags have been set.

MACRO-32 Format:

```
$WFLAND_x efn, mask
```

BLISS-32 Format:

```
$WFLAND (EFN=efn, MASK=mask);
```

efn

Number of any event flag in the cluster being used.

mask

32-bit mask in which bits set to 1 indicate event flags that must be set before the system service returns.

Return Status:

SS$_NORMAL

Service successfully completed.

SS$_ILLEFC

An illegal event flag number was specified.

SS$_UNASEFC

In user mode, indicates that the specified common event flag (see Section 3.14.2) has not been associated with the process issuing the macro.

In standalone mode, indicates that an event flag from 64 through 127 was specified. These flags are not valid in standalone mode.

Notes:

1. While the system service routine is waiting for the event flags to be set, ASTs can interrupt the service. Program control will return to the $WFLAND system service after execution of the AST routine has completed.

Examples:

MACRO-32 Example:

```
$WFLAND_S #0, FLAG_MASK

$WFLAND_S #0, #000000F0
```

BLISS-32 Example:

```
$WFLAND (EFN=0, MASK=.FLAG_MASK);

$WFLAND (EFN=0, MASK=%X'000000F0');
```

$WFLOR

The $WFLOR macro calls a system service that will wait until any one of a specified group of event flags is set before returning. Event flags are discussed in Section 3.14.2. All of the event flags must be in the same event flag cluster. If any one of the specified flags is already set, the service routine returns immediately. Otherwise, control is not returned to the caller until one of the specified flags has been set.

MACRO-32 Format:

```
$WFLOR_x efn, mask
```

BLISS-32 Format:

```
$WFLOR (efn, mask);
```

efn

Number of any event flag in the cluster being used.

mask

32-bit mask in which bits set to 1 indicate event flags that are to be tested by the system service.

Return Status:

SS$_NORMAL

Service successfully completed.

SS$_ILLEFC

An illegal event flag number was specified.

SS$_UNASEFC

In user mode, indicates that the specified common event flag (see Section 3.14.2) has not been associated with the process issuing the macro.

In standalone mode, indicates that an event flag from 64 through 127 was specified. These flags are not valid in standalone mode.

Notes:

While the system service routine is waiting for an event flag to be set, ASTs can interrupt the service. Program control will return to the WFLOR system service after execution of the AST routine has completed.

Examples:

MACRO-32 Example:

```
$WFLOR_S #0, FLAG_MASK

$WFLOR_S #0, #000000F0
```

BLISS-32 Example:

```
$WFLAND (EFN=0, MASK=FLAG_MASK);

$WFLAND (EFN=0, MASK=%X'000000F0');
```

4.7 SYMBOL DEFINITION MACROS

$DS_BITDEF

The $DS_BITDEF macro defines (for MACRO-32 programs) a bit mask for each bit from Ø through 31. For BLISS-32 programs, these symbols may be referenced without first issuing the $DS_BITDEF macro.

Symbols defined are:

```
        BITØ     =       ØØØØØØØ1 (HEX)
        BIT1     =       ØØØØØØØ2 (HEX)
        BIT2     =       ØØØØØØØ4 (HEX)
          :                  :
          :                  :
        BIT31    =       8ØØØØØØØ (HEX)
```

$DS_CFDEF

The $DS_CFDEF macro defines (for MACRO-32 programs) symbolic names for the fields of a call frame. For BLISS-32 programs, these symbols may be referenced without first issuing the $DS_CFDEF macro.

Symbols defined are:

CF$L_ONCOND	- Address of condition handler
CF$W_PSW	- Processor status word
CF$W_MASK	- Register mask
CF$L_AP	- Saved AP
CF$L_FP	- Saved FP
CF$L_PC	- Saved PC
CF$L_REG	- Start of saved RØ through R11

Notes:

1. These symbols are used as offsets from the current FP, as in 'CF$W_PSW(FP)'.

$DS_CHCDEF

The $DS_CHCDEF macro defines (for MACRO-32 programs) the symbolic names of the function codes associated with the $DS_CHANNEL service. For BLISS-32 programs, these symbols may be referenced without first issuing the $DS_CHCDEF macro.

Symbols defined are:

```
CHC$_INITA
CHC$_INITB
CHC$_ENINT
CHC$_DSINT
CHC$_ABORT
CHC$_PURGE
CHC$_CLEAR
CHC$_STATUS
CHC$_SETDFT
CHC$_CLRDFT
```

$DS_CHMDEF

The $DS_CHMDEF macro defines (for MACRO-32 programs) symbolic names for the function codes associated with the $DS_SETMAP service. For BLISS-32 programs, these symbols may be referenced without first issuing the $DS_CHMDEF macro.

Symbols defined are:

```
        CHM$_INVALIDATE
        CHM$_MFWDN
        CHM$_MFWDNO
        CHM$_MFWDV
        CHM$_MFWDVO
        CHM$_MREVN
        CHM$_MREVNO
        CHM$_MREVV
        CHM$_MREVVO
        CHM$_NFWDN
        CHM$_NREVN
```

$DS_CLIDEF

The $DS_CLIDEF macro defines (for MACRO-32 programs) symbolic names for the "traversal codes" used in associated with the $DS_CLI macro.

Symbols defined are:

CLI$K_ALNUM
CLI$K_ALPHA
CLI$K_NUM
CLI$K_SYMBOL
CLI$K_FILE
CLI$K_SPACE
CLI$K_COMMA
CLI$K_SLASH
CLI$K_VALUE
CLI$K_EOL
CLI$K_DEC
CLI$K_HEX
CLI$K_OCT
CLI$K_STRING
CLI$K_BR
CLI$K_BIF
CLI$K_CALL
CLI$K_RETURN
CLI$K_BIFS
CLI$K_EXIT
CLI$K_ERROR

$DS_DSDEF

The $DS_DSDEF macro defines (for MACRO-32 programs) symbolic names for status codes returned by system services that begin with the prefix 'DS$_'. Status codes beginning with the 'SS$_' prefix are defined by the $SSDEF macro in LIB.MLB. For BLISS-32 programs, these symbols may be referenced without first issuing the $DS_DSDEF macro.

Symbols defined are:

DS$_NORMAL	DS$_WARNING	DS$_ERROR
DS$_SEVERE	DS$_OVERFLOW	DS$_NULLSTR
DS$_ILLCHAR	DS$_PROGERR	DS$_TRUNCATE
DS$_NOTDON	DS$_IVVECT	DS$_IVADDR
DS$_VASFUL	DS$_INSFMEM	DS$_MMOFF
DS$_IHWE	DS$_FHWE	DS$_LOGIC
DS$_ILLPAGCNT	DS$_FRABUF	DS$_MCHK
DS$_KRNLSTK	DS$_POWER	DS$_TRANSL
DS$_CHME	DS$_NOTIMP	DS$_IPL2HI
DS$_ICERR	DS$_ICBUSY	DS$_ARITH
DS$_UNEXPINT	DS$_CHMK	DS$_BADTYPE
DS$_BADLINK	DS$_NEEDUNIT	DS$_ILLUNIT
DS$_DEVNAME	DS$_NOPCS	DS$_NOSUPPORT

$DS_DSSDEF

The $DS_DSSDEF macro defines (for MACRO-32 programs and BLISS-32 programs) the symbolic names of entry points for the system services.

For BLISS-32 programs, the macro must be defined globally in at least one source module, as follows:

```
GLOBAL $DSSDEF;
```

Symbols defined are:

DS$ABORT	DS$ASKDATA	DS$ASKADR
DS$ASKLGCL	DS$ASKSTR	DS$ASKVLD
DS$ATTACH	DS$BGNSUB	DS$BRANCH
DS$BREAK	DS$CANWAIT	DS$CHANNEL
DS$CKLOOP	DS$CLRVEC	DS$CNTRLC
DS$CVTREG	DS$ENDPASS	DS$ENDSUB
DS$ERRDEV	DS$ERRHARD	DS$ERRPREP
DS$ERRSOFT	DS$ERRSYS	DS$ESCAPE
DS$FREEDBGSYM	DS$GETBUF	DS$GETMEM
DS$GPHARD	DS$HELP	DS$INITSCB
DS$INLOOP	DS$LOAD	DS$LOADPCS
DS$MAPDBGBLOCK	DS$MMOFF	DS$MMON
DS$MOVPHY	DS$MOVVRT	DS$PARSE
DS$PRINTB	DS$PRINTF	DS$PRINTS
DS$PRINTSIG	DS$PRINTX	DS$PROBE
DS$RELBUF	DS$RELMEM	DS$SETIPL
DS$SETMAP	DS$SETPRIEXV	DS$SETVEC
DS$SHOCHAN	DS$SUMMARY	DS$WAITMS
DS$WAITUS	SYS$ALLOC	SYS$ASCTIM
SYS$ASSIGN	SYS$BINTIM	SYS$CANCEL
SYS$CANTIM	SYS$CLOSE	SYS$CLREF
SYS$CONNECT	SYS$DISCONNECT	SYS$DALLOC
SYS$DASSGN	SYS$FAO	SYS$FAOL
SYS$GET	SYS$GETCHN	SYS$GETTIM
SYS$LKWSET	SYS$NUMTIM	SYS$OPEN
SYS$QIO	SYS$QIOW	SYS$READ
SYS$READEF	SYS$SETAST	SYS$SETEF
SYS$SETIMR	SYS$SETPRI	SYS$SETPRT
SYS$SETRWM	SYS$SETSWM	SYS$ULKPAG
SYS$ULWSET	SYS$UNWIND	SYS$WAITFR
SYS$WFLAND	SYS$WFLOR	

$DS_ERRDEF

The $DS_ERRDEF macro defines (for MACRO-32 programs) the symbolic names of the parameters associated with the $DS_ERRxxxx macros. These symbols will most likely be used in error reporting routines.

For BLISS-32 programs, these symbols are not used in error reporting routines because expansion of the $DS_BGNMESSAGE macro produces a parameter list for the error reporting routine.

Refer to descriptions of the $DS_BGNMESSAGE and $DS_ERRxxxx macros for examples of referencing $DS_ERRxxxx parameters in error reporting routines.

Symbols defined are:

```
ERR$_NUM
ERR$_UNIT
ERR$_MSGADR
ERR$_PRLINK
ERR$_P1
ERR$_P2
ERR$_P3
ERR$_P4
ERR$_P5
ERR$_P6
```

Notes:

1. These symbols are used as offsets into the argument list, for example, ERR$_P3(AP).

$DS_HPODEF

$DS_HPO_DECL

The $DS_HPODEF macro defines (for MACRO-32 programs) the symbolic names of the device-independent fields of a p-table.

The $DS_HPO_DECL is used for BLISS-32 programs. The format of this macro is as follows:

```
   $DS_HPO_DECL ($DS_xxxx_DEF);
```

where "xxxx" represents the name of the device for which p-table fields are to be defined, such as $DS_HPO_DECL ($DS_RH780_DEF).

Symbols defined are:

```
        HP$Q_DEVICE      - Quadword descriptor of device name
        HP$W_SIZE        - Total length of p-table
        HP$B_FLAGS       - Initialization flags
        HP$B_DRIVE       - Unit number
        HP$T_DEVICE      - Start of device name string
        HP$A_DEVICE      - Hardware address of device
        HP$A_DVA         - Base of address space assigned to device
        HP$A_LINK        - Address of p-table for device's link
        HP$W_VECTOR      - Device's vector
        HP$T_TYPE        - Start of counted string for device type
        HP$A_DEPENDENT   - Start of device-dependent portion of p-table
        Device-dependent fields
                    .
                    .
                    .
```

Notes:

1. These symbols should be used as offsets from the base of the p-table. For example, if the p-table base address was placed in R2, then the vector field could be referenced as 'HP$W_VECTOR(R2)'. Refer to Section 3.2.4, Referencing P-Tables from a Diagnostic Program.

$DS_PARDEF

The $DS_PARDEF macro defines (for MACRO-32 programs) symbolic names for values that can be used with the "radix," "defalt," and "exword" parameters to the $DS_ASKxxxx macros. For BLISS-32 programs, these symbols may be referenced without first issuing the $DS_PARDEF macro.

Symbols defined are:

```
        PAR$_BIN
        PAR$_DEC
        PAR$_HEX
        PAR$_OCT

        PAR$_NO
        PAR$_YES

        PAR$V_NODEF      PAR$M_NODEF
        PAR$V_ATLO       PAR$M_ATLO
        PAR$V_ATHI       PAR$M_ATHI
        PAR$V_ATDEF      PAR$M_ATDEF
```

$DS_PTDDEF

The $DS_PTDDEF macro defines (for MACRO-32 programs) symbolic names for the flags associated with the $DS_$NAME p-table descriptor macro. For BLISS-32 programs, these symbols may be referenced without first issuing the $DS_PTDDEF macro.

Symbols defined are:

PTD$M_UNIT	PTD$V_UNIT
PTD$M_CONTROLLER	PTD$V_CONTROLLER
PTD$M_NAME	PTD$V_NAME
PTD$M_INHERIT_PRE	PTD$V_INHERIT_PRE
PTD$M_INHERIT_CON	PTD$V_INHERIT_CON
PTD$M_INHERIT	
PTD$M_DEVICE	
PTD$V_ENDDEVICE	

$DS_PSLDEF

The $DS_PSLDEF macro defines (for MACRO-32 programs) symbolic names for fields of the process status longword. For BLISS-32 programs, these symbols may be referenced without first issuing the $DS_PSLDEF macro.

Symbols defined are:

```
PSL$V_CBIT      PSL$M_CBIT
PSL$V_VBIT      PSL$M_VBIT
PSL$V_ZBIT      PSL$M_ZBIT
PSL$V_NBIT      PSL$M_NBIT

PSL$K_KERNAL
PSL$K_EXEC
PSL$K_SUPER
PSL$K_USER
```

$DS_SCBDEF

The $DS_SCBDEF macro defines (for MACRO-32 programs) symbolic names for the vector offsets in the system control block. For BLISS-32 programs, these symbols may be referenced without first issuing the $DS_SCBDEF macro.

Symbols defined are:

```
SCB$L_ZERO
SCB$L_MACHCK
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
```

$DS_DEFDEL

The $DS_DEFDEL macro is used to conserve memory space during program assembly time. Some of the symbol definition macros cause memory space to be allocated. If the $DS_DEFDEL macro is issued AFTER the symbol definition macros, then any memory space allocated during the symbol defintion process will be deallocated. This will not affect the symbol definitions themselves.

CHAPTER 5
CREATING A VDS DIAGNOSTIC PROGRAM

5.1 INTRODUCTION

The previous chapters have presented the building blocks needed to contruct a diagnostic program that will execute under the VAX Diagnostic Supervisor. This chapter describes the steps required to create a VDS diagnostic program, from the program's inception to its completion. It also specifies all standards and conventions to which a diagnostic program must adhere.

5.2 PROGRAM DEVELOPMENT PROCESS

5.2.1 Overview

Creating a diagnostic program involves several distinct, consecutive phases. Each phase is required, and the phases must be entered in same order that they are described here.

5.2.2 Consultation Phase

The consultation phase of program development consists of informal gathering and exchanging of information relating to the hardware product for which the diagnostic program is to be written. This phase should begin soon after an engineering or product management group has made a commitment to develop a new product.

Goals of this phase are to formulate a testing strategy for the product (what types of diagnostic programs should be developed), identify a few key project milestones (dates), and estimate staffing and funding requirements.

The consultation phase begins before staffing and funding commitments have been negotiated. Typically, the result of this phase is a cursory project plan.

Participants will include management and senior technical personnel from the engineering group or product line developing the product, the future program's user community (generally field service and manufacturing personnel), and the diagnostic programming group.

An important note: If it is desirable for the hardware design of a new product to provide aids that will enhance the fault detection of a diagnostic program, then the diagnostic programming group must request these aids as soon as possible in order to ensure that they will be incorporated into the device's final design. Negotiations for design changes to aid diagnosis should thus commence during this phase of the project.

5.2.3 Planning Phase

This phase begins after staffing and funding commitments have been made. This and all following phases are performed by the diagnostic program's project leader and his or her staff.

The goal of the planning phase is to develop a plan for implementation of the project. This project plan will include a description of the diagnostic program and will specify project goals, schedules, development requirements, training requirements, and maintenance requirements.

The result of this phase is a Diagnostic Engineering Project Plan adhering to the format specified by Section 7C3-1.A of the Software Development Policies and Procedures.

5.2.4 Functional Specification Phase

After the project plan has been completed, the task of defining the functional operation of the diagnostic program begins.

The goal of this phase is to clearly define the functions that the diagnostic program will perform. A functional specification must answer the question, "What will the program do?". (On the other hand, it should NOT approach the question of HOW the function will be implemented.)

Additionally, a functional specification will include specific statements about the program's intended uses and users, plus goals regarding the program's performance and run-time parameters.

The result of the functional specification phase is a Diagnostic Engineering Functional Specification that adheres to the format specified by Section 7C3-2.A of the Software Development Policies and Procedures.

5.2.5 Design Phase

The program's design phase may be entered when the functional specification phase has been completed.

The goal of the design phase is to develop a design specification that defines the methods that will be used to implement the functionality defined in the functional specification. This phase answers the question, "How will the program's functionality be provided?". For example, if the functional description states that the program will test a certain section of the device's logic, then the design specification will describe the algorithm to be used to perform the test.

Some of the methods that may be used to specify designs are:

1. Detailed hierarchy charts
2. Interface specification blocks
3. HIPO diagrams
4. Structured flowcharts
5. Program Design Language 1 (PDL1) (See below.)

The result of this phase will be a Diagnostic Engineering Design Specification, adhering to the format specified in Section 7C3-3.A of the Software Development Policies and Procedures. This document also describes PDL1.

5.2.6 Design Implementation Phase

After the design has been completely specified, it may be implemented. Design implementation is, of course, the phase in which coding and debugging take place.

The schedule on which coding and debugging of the various pieces of the program is based depends greatly upon the availability of product hardware. Programs that are written for new hardware are typically in the process of development concurrently with the hardware itself. Therefore it is important to create a schedule for program development that matches the hardware development's schedule.

Implementation of programs for new hardware must often be carried out in two stages. These stages are referred to as "prototype support" and "final product support."

Prototype support involves providing the engineering group responsible for the product with a preliminary version of the program. This version will be used to help verify the integrity of the hardware design. The engineering group will generally expect this version to be ready for use within a matter of days after the hardware is "powered up" for the first time. Specific requirements for prototype support depend on the particular product. These requirements should be specified in the Project Plan and Functional Specification.

Unfortunately, it may be necessary to provide prototype support before the planning and specification phases just described have been completed. Therefore, it is important to carefully coordinate all phases of program development so that the needs of all users can be met on schedule. For example, some portions of the design specification or even the functional specification may have to be delayed until debugged code supporting the prototype hardware has been provided.

Final product support involves development of the program that will be used with the final, error-free version of the hardware product. This is the version of the program that will be released for general use. User requirements for the final product may be different from user requirements for prototype support. Knowledge of the hardware's operation that was gleaned by the programmer during development of prototype support will, of course, aid him or her in creating a program that provides high degrees of fault detection and isolation.

Because hardware development and diagnostic program development occur concurrently for new hardware products, it is necessary to carefully coordinate the two development processes. Hardware design engineers and manufacturing personnel will often desire working versions of the diagnostic program before the scheduled completion date. It is thus common for diagnostic programmers to provide "prerelease" versions of the program before the final program has been completed. A prereleased program may or may not provide the full functionality that will exist in the final program.

5.2.7 Design Verification Phase

Once the final program has been completed, its functionality and operation must be assessed to ensure that the program meets all of the functionality goals that were originally set and that it adheres to all applicable operating standards (such as using VDS macros properly). Assuring overall program quality is performed by following the steps indicated in Section 5.10, Quality Assurance.

5.3 PROGRAM STRUCTURE

Chapter 3 described all of the required and optional components of a VDS diagnostic program. Since all VDS diagnostic programs are made up of the same components, it is useful to arrange these components in the same order and format in the source code of every program. This will aid program maintainers by ensuring a large measure of consistency from one program to the next.

In all diagnostic program sources, program components should be divided into a series of source modules. There should be a "header module" and one or more "test modules."

5.3.1 Header Module

The header module contains all of the tables used by the VDS, the initialization, cleanup, and summary routines, plus any routines used globally by the diagnostic program. Components of the header module should be arranged in the following order:

- Module cover page (copyright statement, title and author, maintenance history)
- Functional description of module
- Declarations of libraries and BLISS require files
- User-defined macro definitions
- Symbol definitions
- Diagnostic header ($DS_HEADER)
- Dispatch table ($DS_DISPATCH)
- Statistics table ($DS_BGNSTAT, $DS_ENDSTAT) (optional)
- Section names declaration ($DS_SECTION)

- Device mnemonics list ($DS_DEVTYP)
- ASCII text:
  - Register and bit names for $DS_CVTREG calls
  - Other ASCII strings
  - Error message strings
- Initialization code ($DS_BGNINIT, $DS_ENDINIT)
- Cleanup code ($DS_BGNCLEAN, $DS_ENDCLEAN)
- Summary routine ($DS_BGNSUMMARY, $DS_ENDSUMMARY)
- Error reporting routines ($DS_BGNMESSAGE, $DS_ENDMESSAGE)
- Other (optional) global subroutines, including interrupt service routines, condition handlers, and so on.

Note: If a program has many global routines and data structures, they should be placed in a separate module.

5.3.2 Test Modules

Each test module will contain one or more tests. The number of tests modules and the number of tests per module are unrestricted. Each test module should be formatted as follows:

- Module cover page (copyright statement, title and author, maintenance history)
- Functional description of module
- Declarations of library and require files
- User-defined macro definitions
- Symbol definitions
- Section names declaration ($DS_SECDEF)
- For each test in module:
  - Test name ($DS_SBTTL)
  - $DS_BGNTEST
  - Test header
  - For each (optional) subtest in test:

    Subtest header

    $DS_BGNSUB

    Subtest code

    $DS_ENDSUB

  - $DS_ENDTEST

5.3.3 Module Templates

To help the programmer follow the above formats, template files have been created. There is a header module template and a test module template. Each template contains the program-independent fields of each program component. The programmer simply fills in the program-dependent fields of each module. These templates are named HEADER.MAR and TEST.MAR for MACRO-32, and HEADER.B32 and TEST.B32 for BLISS-32. The templates are reproduced in Appendixes A and B.

5.4 PROGRAM DOCUMENTATION

5.4.1 Introduction

A diagnostic program should be considered to be made up of two parts -- the code and the documentation. Each of these parts is of equal importance. Documentation should NEVER be thought of as auxiliary to the code, to be hurriedly added at the end of the project, if time permits. The best documentation is that which is developed before and during code development.

Diagnostic program documentation serves two purposes:

1. Users of diagnostic programs probably refer to and depend on program documentation more than users of any other software. This is because identification of hardware failures requires a very exact understanding of what function is being performed by a particular section of code and what areas of the hardware circuitry are likely to be activated to carry out that function. It is sometimes necessary for the program user to read the program's listing files to see what signals are being activated within a test or subtest.

2. As is the case with any software product, program maintenance is usually performed by persons other than the product's author. Those who must enhance, correct, or otherwise update a diagnostic program depend on the documentation for understanding of the program's function, design, and implementation.

Documentation for VDS diagnostic programs consists of three parts. These are:

1. A documentation file containing hardware requirements, operating instructions, and functional descriptions of the program's tests

2. Source code documentation providing detailed functional descriptions of every test, subtest, routine, and line of code

3. "Help" files that the user can access with the VDS HELP command, and that summarize the program's operating instructions

5.4.2 Documentation File

The documentation file will be distributed with the diagnostic program. The documentation file for program EVXYZ will be called EVXYZ.DOC. A template for the documentation file is available in both RUNOFF and non-RUNOFF formats. A reproduction of the template can be found in Appendix C.

The documentation file will contain the following information:

- Cover page

  The cover page contains identification information such as the program's name, release date, and maintainer, along with copyright and disclaimer statements.

- Table of contents

- Abstract

  The abstract is a short description of the program, summarizing information found in later sections of the document. This section should identify which types of hardware will be tested. It should also state the program level (level 2R or level 3).

- Hardware requirements

  This section lists the minimum hardware required for the program to execute, plus any optional hardware. Include special connectors or other special hardware required by the program.

List the processor types with which the program is compatible. Do NOT make generalized statements, such as "all VAX processors," since the program may not be executable on future processors.

- Software requirements

  List the software required, including the VAX Diagnostic Supervisor. Any auxiliary data files should be included here.

- Prerequisites

  This section should list the program's hardcore requirements, that is, the hardware that must be operating properly in order for the diagnostic program to correctly diagnose faults on the hardware being tested.

- Operating instructions

  In most cases, the VAX Diagnostic Supervisor User's Guide should be the only reference needed for operating instructions.

  - Options

    If the program has special instructions (such as using a user-defined command language), that information should be provided in this section.

  - Event Flags

    If any user-controllable event flags are used by the program, they should be listed.

- Program functional description

  - Program overview

    This is a general functional description of the program. The program's purpose and testing strategy should be included.

  - Program size

    The load time and run-time memory requirements should be specified. Include memory required by any auxiliary data files.

- Program run times

  The execution time of each program section is listed here. If a QUICK mode is provided, include its execution time also.

- Run-time dynamics

  Indicate how the program allocates resources during execution time. Include both memory and device allocations. Specify the mimimum buffer space needed.

- Fault detection

  Describe the fault coverage (include percentage) and error resolution the program is capable of.

  Include sample error messages, if error reporting routines are used.

- Performance during hardware failures

  Indicate how the program will handle unexpected exceptions resulting from hardware failures, power failure, and the like.

- Program applications

  List the uses this program was designed for, such as manufacturing, customer services, engineering, customers, or whomever.

- Test descriptions

  For each test, include:

  A functional description of the test

  The step-by-step flow of the test

  "Debug aids" - hints to the program user about what should be looked at next if the test fails. Very important for logic tests.

● Maintenance history

Each time the program is updated, the update must be described here. The description must include the date of the change and the program's version number.

5.4.3 Source Code Documentation

5.4.3.1 Diagnostic Codes - Each diagnostic program released by DIGITAL is assigned a "diagnostic code" that uniquely identifies it. Codes for VAX diagnostic programs consist of five characters, the first of which is "E." The code is assigned by the Release Engineering group.

5.4.3.2 Module Names - For the diagnostic program having the diagnostic code "EVXYZ," the header module should be named EVXYZØ.MAR if it is a MACRO-32 program, or EVXYZØ.B32 if it is a BLISS-32 program. Test modules should be named EVXYZ1.MAR (or .B32), EVXYZ2.MAR (or .B32), and so on.

5.4.3.3 Module Cover Page - Each module must have a cover page. The cover page will include:

1. Module and program names, including version numbers (see above).

2. Copyright statement

3. Module abstract

4. Author

5. Maintenance history (see below).

The format of the cover page is illustrated in the header module template example contained in Appendix A.

5.4.3.3.1 Maintenance History - Each time the module is updated, the update must be described here. The description must include the date of the change and the module's version number.

5.4.3.4 Test and Subtest Prefaces - Each test and each subtest must possess a preface. Prefaces for tests and subtests must contain the following information:

1. Test description

   This will contain a detailed description of WHAT is being tested and HOW the test is implemented.

2. Assumptions

   List assumptions being made about the state of the hardware before the test is executed. For example, if this test will not function properly unless certain parts of the hardware are good, list those parts.

3. Test steps

   In this section list the test steps. A pseudolanguage is very useful for this purpose.

4. Errors

   Provide a detailed description of all errors reported by this test.

5. Debug

   This section should provide information that might be helpful to someone attempting to determine the cause of a hardware error. For example, there might be a statement of the form "If error number X is reported, then Y might be broken."

The format of test and subtest prefaces is illustrated in the test module template in Appendix B.

5.4.3.5 Subroutine Preface - Each subroutine must possess a preface. Subroutine prefaces must contain the following information:

- Functional description

  This must be a DETAILED description of WHAT function the routine performs and does and HOW the function is performed.

- Calling sequence

  Indicate how the routine is to be called, for example:

  ```
              CALLS #4,ROUTINE or CALLG ARGPTR,ROUTINE
     or       BSBW ROUTINE
     or       Entered via exception vector
  ```

- Inputs

  List here all input parameters that are explicitly passed to the routine. Explicitly passed input parameters are those pushed onto the stack before a routine is called. (In BLISS-32, explicit input parameters are those that are listed in parentheses after the routine name.)

- Implicit inputs

  List here all input parameters that are not explicitly passed on the stack. This list will include ANY variable referenced by the routine but not defined locally in the routine and not passed explicitly. For example, parameters passed in registers are implicit inputs.

  Note: Use of implicit inputs should be kept to a minimum. They adversely affect program maintainability and routine transportability.

- Outputs

  List all output parameters that are explicitly passed back to the caller. Explicitly passed output parameters are those that are:

  - Loaded onto the stack by the routine, or

  - Loaded into locations whose addresses were explicitly passed to the routine.

- Implicit outputs

  List all output parameters that are implicitly returned to the caller. Implicit output parameters are ANY variables that are modified by the routine but were not explicitly passed to the routine. For example, if a variable stored in a register is updated, that variable is an implicit output.

  Note: Use of implicit outputs should be kept to a minimum. They adversely affect program maintainability and routine transportability.

- Completion codes

  Indicate all completion codes that could be returned by this routine. If the routine passes along completion codes received from subordinate subroutines, these codes must also be listed. Also indicate how the completion code is passed. (Placing the code in RØ is the normal method.)

- Side effects

  List here any actions taken by this routine that could affect the operation of other routines. Examples are initializing data structures or altering the state of global flags.

  Also, if the routine places the hardware in some unusual or indeterminate state, indicate that here.

- Registers used

  Identify the purpose of each general purpose register used by the routine, so anyone reading the code can quickly determine the functions of the registers.

The format of a routine preface is illustrated in the header module template of Appendix A.

5.4.3.6 Source Code Comments - It is extremely important that the source code be very accurately commented. Comments within the source code can take three forms; they may be "block comments," "group comments," or "line comments."

5.4.3.6.1 Block Comments - Block comments are used to identify major functions within a routine. They have the following format:

MACRO-32

```
<skip>
;++
; This is a block comment.  It begins at the left-hand margin
; and extends fully across the page.
;--
<skip>
```

BLISS-32

```
<skip>
!++
! This is a block comment.  It begins at the left-hand margin
! and extends fully across the page.
!--
<skip>
```

5.4.3.6.2 Group Comments - Group comments are used within blocks of code delimited by block comments. The are useful when it is desirable to make a comment stand out on the page. Group comments have the following format:

MACRO-32

```
;
; This is a group comment.  It is indented the same amount as
; the code being commented, and extends fully across the page.
;
```

BLISS-32

```
!
! This is a group comment.  It is indented the same amount as
! the code being commented, and extends fully across the page.
!
```

Group comments should be used extensively in BLISS-32 programs. BLISS-32 programs use group comments instead of the line comments (below) used by MACRO-32 programs.

An illustration of the use of group comments in BLISS-32 code is as follows:

```
   !
   ! Explain what the IF-THEN-ELSE statement will do.
   !

   IF ... THEN ....
   ELSE
           BEGIN
             :
             :

           !
           ! Explain what the REPEAT-UNTIL loop will do.
           !

           REPEAT
             :
           UNTIL ... ;
             :
           END;
```

See the example in the next section for an illustration of group comments in MACRO-32 code.

5.4.3.6.3 Line Comments - Line comments are those that appear at the end of, and on the same line as, a MACRO-32 instruction or BLISS-32 statement. These comments are extremely important in MACRO-32 programs, and EVERY MACRO-32 instruction should be followed by a line comment.

In BLISS-32 programs, line comments are generally not used, because group comments are preferable. Since lines of BLISS-32 code are self-documenting (if they are written properly), line comments are unnecessary. Group comments should be placed before most blocks to describe what will occur within each block.

Line comments (and group comments) are illustrated in the following MACRO-32 example:

```
           .
           .
           .
;
; Clear the data buffers.
;

   CLRL    R6                  ; Clear buffer pointer
15$:                           ; REPEAT
   CLRL    W^GOOD_DATA[R6] ;     Clear longword of good data buffer
   CLRL    W^BAD_DATA[R6]  ;     Clear longword of bad data buffer
   AOBLSS  #16, R6, 15$    ;     Increment pointer and branch back
                           ; UNTIL entire buffer is cleared

;
; Compare expected and received data, one longword at a time.  If
; they do not match, store the expected and received values in the
; good data buffer and bad data buffer, respectively, so they can be
; printed later.
;
   MOVL    4(AP), R2       ; Put byte count in R2.
   MOVL    8(AP), R3       ; Put address of received data in R3.
   MOVL    12(AP), R4      ; Put address of expected data in R4.
   CLRL    R1              ; Clear error count.
   CLRL    R5              ; Clear buffer pointer.
           .
           .
```

This example illustrates several concepts:

1. Every MACRO-32 instruction has a comment.

2. It is useful to indicate structured programming constructs where applicable. Notice the REPEAT-UNTIL construct in the example. IF-THEN-ELSE, WHILE-DO, CASE constructs, and so on, can be flagged similarly, enhancing readability. Capitalize keywords and indent comments within a construct.

3. Comments provide useful information. For example, the last comment in the example says, "Clear buffer pointer." It does NOT say "Clear R5," which would be useless to anyone reading the code.

5.4.4 Help Files

5.4.4.1 Description of Help Files - A "help file" is a text file that is referenced when the VDS HELP command is used. Text within the file is displayed to the user. Arguments specified with the HELP command are used to determine which portions of the text to display.

A help file must be provided for every diagnostic program. For program EVXYZ, the help file must be named EVXYZ.HLP. A user can reference this file by typing 'HELP EVXYZ'.

The purpose of a diagnostic program's help file is to provide the program user with a quick reference source that will summarize the program's unique characteristics. Information contained in a help file will include:

- A program abstract

- ATTACH procedures

- A list containing the name and function of each program section
- Descriptions of devices not supported by the VDS (devices for which p-table descriptors reside in the diagnostic program instead of in the VDS)
- A list containing the number and use of any user-selectable event flags referenced by the program
- A description of the program's "quick mode" operation
- Descriptions of tests requiring manual intervention
- The format of the program's summary message, if one exists

5.4.4.2 Creating Help Files - Help files consist of "keywords" and associated text. Keywords are used by the VDS to locate the proper text to display. For instance, if a user typed HELP EVXYZ SECTIONS, the VDS would search the help file named EVXYZ.HLP for the keyword "sections," and then display the text following that keyword. There are two types of keywords, referred to as "numbered keywords" and "qualifier keywords."

5.4.4.2.1 Numbered keywords - Each numbered keyword is preceded by a number from 1 through 5. This number indicates the keyword's "level." Level 1 is the highest level, and is used to indicate the file's main topics. Keywords with larger numbers are considered to be subtopics of those with smaller numbers. If the file contains a level 1 keyword followed by several level 2 keywords, followed by another level 1 keyword, then the level 2 keywords between the first and second level 1 keywords are subtopics of the first level 1 keyword. If the second level 1 keyword was followed by another set of level 2 keywords, they are subtopics of the second level 1 keyword.

The level number must be the first character of a new line. There must be one or more spaces or tabs between the level number and the keyword.

When the user types a HELP command, the VDS will display the text following the specified keyword. It will also display the keywords (but not the text) of the next-lowest level subtopics associated with the specified keyword. For example, suppose a portion of a help file consisted of the following:

```
              .
              .
1 SECTIONS
 Program EVXYZ contains the following sections.  Type

   HELP EVXYZ SECTIONS section-name

 for details on a particular section.
2 DEFAULT
 (Text describing DEFAULT section.)

2 MANUAL
 (Text describing MANUAL section.)

2 READ_TESTS
 (Text describing READ_TESTS section.)

2 WRITE_TESTS
 (Text describing WRITE_TESTS section.)

1 ATTACH
              .
              .
```

If the user typed 'HELP EVXYZ SECTIONS', the following would be displayed:

```
   SECTIONS

     Program EVXYZ contains the following sections.  Type

           HELP EVXYZ SECTIONS section-name

     for details on a particular section.

   Additional information available:

   DEFAULT MANUAL  READ_TESTS      WRITE_TESTS
```

Any time a topic is specified with a HELP command, the VDS displays the text associated with the topic and lists the subtopics (keywords with next higher level number) associated with the topic.

All of the subtopics of a topic are listed directly underneath the topic in the help file. Thus all the level 3 subtopics associated with a level 2 keyword would directly follow that level 2 keyword.

Thus in the above example, suppose the user typed, 'HELP EVXYZ SECTIONS DEFAULT'. The VDS would display the text associated with the level 2 keyword "default," and then would list any level 3 keywords that follow the text for "default." (The sample help file above does not associate any level 3 keywords with "default.")

5.4.4.2.2 Qualifier keywords - It is unlikely that a diagnostic program's help file will require qualifier keywords, since they are only used to indicate command line qualifiers. They are not preceded by a level number; instead, they begin with the "/" character. However, a level number is implicitly associated with a qualifier keyword; that number is one greater than the number specified in the most recently specified numbered keyword. That keyword should be 'Qualifiers'. This is illustrated in the following example:

```
        .
        .
1 START
 Execute a previously loaded image.

Format:
        START [qualifiers]
2 Qualifiers
/SECTION:section-name
 Select a program section to be executed.
/TEST:first:last
 Select a range of tests to be executed.
        .
        .
```

The '/' character must be the first character of a new line. The keyword must immediately follow the '/'. Immediately following the keyword there may be an additional string, as in "/QUAL:string."

Note: If one qualifier keyword directly follows another, with no text in between, then the second qualifier keyword will be treated as part of the text for the first. This is useful for qualifiers of the form "/qual" and "/NOqual."

5.4.4.2.3 Text - Text must immediately follow the keyword with which it is associated. It must start on a new line. Each line of the text must be indented one space from the left margin. Text associated with level 1 keywords should not extend beyond column 65. Text associated with keywords of any other level should not extend beyond column 6Ø. The text is more easily readable if it does not exceed the length of the display screen (no scrolling should occur).

5.4.4.3 Contents of help files - Help files for diagnostic programs must contain the following level 1 keywords and associated text:

- ATTACH - Describe the attach procedures for the program. That is, list the set of ATTACH commands that are necessary to create the proper links from the unit under test to the processor.

- DEVICE - Under this keyword, include a level 2 keyword for every device tested by the diagnostic program. Under each level 2 keyword, provide either of the following:

  - For devices with p-table descriptors contained in the VDS, the text should state, "Type HELP DEVICE device-type for device description."

  - For devices with p-table descriptors contained in the diagnostic program, provide a device description similar to the device description that is obtained from typing "HELP DEVICE device-type."

- EVENT - List any user-selectable event flags referenced by the program and describe their function.
- HELP - This text should contain an abstract of the program. The text associated with the HELP keyword is displayed when a user types 'HELP EVXYZ' without including a keyword. In other words, this is the default keyword.
- QUICK - Describe the operation of the program when the QUICK flag is set.
- SECTIONS - List and describe each section of the program. Be sure to include the DEFAULT section. If a MANUAL section exists, clearly detail the actions that must be performed by the user.
- SUMMARY - If the program contains a summary routine, provide an explanation of the information displayed by that routine.

The above keywords must appear in every help file. Other keywords should be added to provide information on unique program characteristics.

The keywords must be placed in the help file in alphabetical order.

A sample help file is provided in Appendix D.

5.5 RUN-TIME ENVIRONMENT CONSIDERATIONS

One of the main purposes of the VAX Diagnostic Supervisor, as stated in Chapter 2, is to insulate the diagnostic program from the various runtime environments that exist for diagnostic programs.

Thus if all of the rules, guidelines, and conventions described in this manual are followed, any diagnostic program written should be capable of executing in any of the run-time environments under which diagnostic programs are expected to run.

Possible run-time environments for VDS diagnostic programs include (but are not limited to):

1. User mode
2. Standalone mode
3. Automated Product Test (APT)
4. Remote Diagnosis (APT/RD)

For details on any of these environments beyond what is provided in this manual, contact the 32-Bit Systems Diagnostic Engineering Group, who can provided related documents.

5.6 CUSTOMER-RUNNABLE DIAGNOSTICS (CRD)

For both the standalone mode and the user mode environments, a system has been developed by which DIGITAL customers can easily and automatically run diagnostic programs. This system is referred to as Customer-Runnable Diagnostics (CRD). CRD provides the following modes of operation:

- "Auto mode," in which the user can type one command (TEST) which that cause a set of diagnostic programs to be executed. This set of programs will completely test the user's system.

- "Menu mode," which allows the user to select (by menu) the testing of specific devices.

CRD auto mode is provided only in the standalone mode run-time environment. CRD menu mode is provided in both the standalone mode and user mode environments.

Diagnostic programs that are to be executed under the CRD system may have certain constraints placed on them. These constraints may include limitations on maximum execution time, prohibition of the use of any manual intervention (see Section 5.7.5), or other run-time conditions. Constraints placed on programs running under the CRD system are described in the document called VAX Diagnostic Requirements for CRD. Any program that will be executed under the CRD system must obey the rules set forth in that document.

5.7 CODING CONVENTIONS

5.7.1 Error Message Formats

As stated in Chapter 3, error messages are displayed by invoking the $DS_ERRxxxx services. Error messages consist of three levels. They should adhere to standard formats.

The format of the first message level (the message header) is controlled by the VDS.

The formats of the second and third message levels are controlled by the programmer. These parts of the error message are contructed with the error reporting routines called by the $DS_ERRxxxx services and delimited by $DS_BGNMESSAGE and $DS_ENDMESSAGE macros.

When error reporting routines are constructed, messages should be formatted as follows:

- Invalid contents of a register:

  A message that reports invalid contents of a register should indicate the expected contents, the actual (received) contents, and an exclusive-OR (XOR) of the expected and received values. Mnemonics of bits set in the XOR value should be displayed. Indicate the radix of all values displayed.

  Example:

  ```
  EXPECTED:       5Ø68(X)
  RECEIVED:       ØØØØ(X)
  XOR:            5Ø68(X) ;TIE,SAE,RIE,MSE,MAINT,FUNC=READ
  ```

- Reporting data comparison errors for buffers

  When data comparison errors are detected in data transfer buffers, the error message should include:

  - The base address of the failing device
  - The address of the buffer
  - The size of the data transfer
  - The number of comparision errors
  - The buffer address and contents of all bad data

  Example:

  ```
  Device base address      :      6ØØ1Ø5ØØ(X)
  Expected buffer address  :      ØE1Ø(X)
  Received buffer address  :      1Ø1Ø(X)
  Transfer size            :      256 words
  Words in error           :      4

  Address:        Expected:       Received:       XOR:

  ØE1Ø4           1Ø1Ø            1ØØØ            1ØØØ
  ØE11Ø           1Ø1Ø            1ØØØ            1ØØØ
  ØE1CØ           1Ø1Ø            1ØØØ            1ØØØ
  ØE1FØ           1Ø1Ø            1ØØØ            1ØØØ
  ```

If there are a large number of errors, only display the first eight.

- Register dumps

  When dumping the contents of a set of registers, list the registers in order of address. Display the register mnemonic, the register's contents (and radix), and the bit mnemonic for each set bit.

  Example:

```
RPCS1  :    144270(O)    ;SC,TRE,DVA,RDY,FUNC=WRITECHECK
RPWC   :    777710(O)
RPBA   :    001000(O)
RPDA   :    001001(O)    ;TRACK=2,SECTOR=1
RPCS2  :    040203(O)    ;WCE,OR,UNIT=3
       .
       .
   (etc.)
       .
```

5.7.2 Volume Verification

All diagnostic programs that write onto magnetic media must provide a mechanism to ensure that a customer's data base is not inadvertently destroyed.

Some disks provide that a portion of the medium (called "maintenance tracks") is always reserved for diagnostic purposes. If a diagnostic program writes only on the maintenance tracks, then the customer's data base will not be affected.

If a device being tested does not provide maintenance tracks, or if the diagnostic program does not limit itself to only using the maintenance tracks on a device that does provide them, then the entire medium must be protected; a method must exist for verifying that the medium loaded in the device under test may be written on.

Thus, for devices that do not provide maintenance tracks, diagnostic programs must check the volume name of a storage medium before executing any tests that will write on that medium. By convention, media that contain no stored data and hence are available for the writing of test patterns by diagnostic programs are named "SCRATCH."

Volume verification must take place in a program's initialization code.

The program must read the storage medium's home block to determine the medium's volume name. (Refer to the FILES-11 On-Disk Structure Specification for a description of the home block's format.)

If the volume name is "SCRATCH," the medium may be used and testing may thus begin.

If the volume name is anything other than "SCRATCH," the program must ask the user (via the $DS_ASKLOGICAL system service) if it is all right to use the medium. If the response is "no" (the user does not wish the medium to be used), then the program should issue a $DS_ABORT call. A DEFAULT RESPONSE MUST BE PROVIDED FOR THE $DS_ASKLOGICAL SERVICE, AND THE DEFAULT MUST BE "NO." This will ensure that if the OPERATOR flag is cleared and a nonscratch medium has been mistakenly placed in the unit under test, then the medium will not be used.

The volume verification code should only be executed the first time through the initialization code (use the $DS_BPASSØ or $DS_BNPASSØ macro). Otherwise, the user would have to respond to the $DS_ASKLOGICAL question for every program pass.

Note: Previous editions of this guide have indicated that, when asking the user if it is all right to use a nonscratch medium, the user prompt passed to the $DS_ASKLOGICAL service must begin with a null character. This null will force the VDS to check the user terminal for a response to the question, even if the program is being run by a command file (script). (If the program is being run by a command file, all responses are obtained from the command file, unless the prompt string begins with a null.)

This is not a good practice, because it forces limitations onto the user regarding how the program may be executed. It should be the user's decision whether a question's response is to be fetched from a script or from the terminal, not the programmer's decision. Therefore, prompt strings should never be preceded with a null character. (Refer to the VAX Diagnostic Supervisor User's Guide for a description of command files.)

5.7.3 Long Silences

A "long silence" is a long period of time in which there is no communication between the diagnostic program and the user. Sometimes long silences are good and sometimes they are bad.

A long silence is good when a diagnostic program is running for a long period of time (either because the program's execution time per pass is long, or because a large number of passes has been selected by the user) and the user's terminal is a hardcopy terminal. Long silences save paper and lessen the risk of paper jams when no one is around.

On the other hand, a long silence is bad when a user is present at the terminal, monitoring the program's progress. In this case the user would like to be kept abreast of the program's status during long executions in order to be assured that the program is not "hung." If a long silence occurs, the only way a user can monitor program progress is to type a control-C, then SHOW STATUS, then CONTINUE.

Thus a diagnostic program must have the capability of both eliminating and providing long silences.

To eliminate long silences in programs with long execution times per pass, the program should cause a message to be displayed at least once per minute. An AST routine may be used for this purpose. The message should be a simple, succinct indication to the user that program execution is progressing properly.

To provide for long silences when the user desires them, a means of disabling the above-mentioned AST routine should be provided. For example, the AST routine should check the status of the OPERATOR flag (by using the $DS_BOPER or $DS_BNOPER macros) and only print the message if the flag is set.

5.7.4 Hardware Preparation

"Hardware preparation" is the act of setting the device under test in some physical state before testing begins. Hardware preparation may include setting switches, connecting a cable, loading a special medium into the device, and the like.

Ideally, diagnostic programs should be written so that no hardware preparation has to take place. If this is not possible, hardware preparation should be kept to an absolute minimum, since it lengthens testing time and is a nuisance for the program user.

All hardware preparation should occur before the program is started. If the program requests hardware preparation during execution, it is referred to as "manual intervention" (see next section) and is considered to be even more of a nuisance.

If a program detects a preparation error (hardware not set up correctly), the $DS_ERRPREP service should be used to report the error.

5.7.5 Manual Intervention

The term "manual intervention" refers to user actions during program execution. A program requiring manual intervention is one requiring the program user to perform a duty at some point during the program's execution. This duty might be as involved as adding a piece of hardware to (or removing one from) the system under test, or it might be a simpler action, such as typing a response on the terminal.

Ideally, no diagnostic program should ever require manual intervention, because manual intervention complicates the operation of the program from the user's point of view.

If inclusion of manual intervention cannot be avoided, the following rules must be followed:

1. If the manual intervention involves ANY actions OTHER THAN responding to questions at the user terminal, the tests that require these actions must be placed in a program section called "MANUAL." Examples of such actions are setting a write-enable switch, connecting a cable, or watching patterns generated by a program that tests video display terminals.

   Each test within the MANUAL section must use the $DS_BOPER or $DS_BNOPER macro to determine if a user is present. If a user is not present, the test must call the $DS_ABORT service.

2. Communication with the user must be performed by using the $DS_ASKxxxx macros.

3. If $DS_ASKxxxx macros are included in the MANUAL section, it is not necessary to provide default responses.

   If $DS_ASKxxxx macros are used anywhere OTHER THAN in tests within the MANUAL section, default responses MUST be provided. If default responses are included, and if the user clears the OPERATOR flag, then the default responses will automatically be used and the user will not have to be present. (This is of course true for the DEFAULT section, also.)

5.7.6 Quick Mode

"Quick mode" is a mode of program execution in which the main objective is to provide a relatively fast execution time per pass. It is a convenient mode to provide in programs having long execution times. It should provide a fast pass/fail testing capability, with little or no fault isolation. It will be employed when a user wants a quick verification of hardware integrity.

The decision of whether or not a diagnostic program will provide a quick mode is one shared between the programmer and the program's users. Specific functions of a particular program's quick mode are also to be decided by mutual agreement between the programmer and users.

If quick mode operation is provided, it is to be executed only if the user selects it by setting the VDS control flag QUICK. The program will use the $DS_BQUICK or $DS_BNQUICK macro to determine the state of the QUICK flag.

5.7.7 Naming Symbols

For the sake of consistency from program to program, it is important to obey certain conventions when creating names for symbols. These conventions are as follows:

1. The dollar sign '$' character is included in all publicly defined symbols located in the VDS and in all other system level software provided by DIGITAL. To differentiate private symbols (those available only to the program in which they are defined) from public symbols, private symbols should not include the '$' character. Since ALL symbols defined in diagnostic programs are private, the '$' should never be used.

   Note: There is one exception to this rule; since p-table descriptors are public, their names should include dollar signs. See Section 3.2.3, P-Table Descriptors, for details and examples.

2. To determine the characters allowed in a symbol name, and the maximum length of a symbol name, refer to the reference manual for the language in which the program is being written.

3. Global variable names are of the form:

   Gt_variablename

   where "t" is a letter indicating the variable type (see Table 5-1).

4. Global arrays are of the form:

   A_arrayname

5. Structure field names are of the form:

   structure_t_fieldname

   where "t" is a letter indicating the variable type (see Table 5-1).

6. Entry points to global routines having nonstandard calls are of the form:

   entryname_Rn

   where registers RØ through Rn are not preserved by the routine and thus must be saved by the caller.

7. When naming bits and bit fields in hardware registers, use the bit mnemonics specified in the hardware documentation.

Table 5-1 contains letters used for data types.

Table 5-1 Letters Used to Indicate 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 DIGITAL as an escape to other codes
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

Some examples of symbol names are:

A_RP_REG	- Address of storage array for RPxx controller registers
RP_REG_L_RPDS	- Offset RPDS into array RP_REG
GW_BYTE_COUNT	- Address of global word containing byte count

5.8 LINKING A DIAGNOSTIC PROGRAM

Before a diagnostic program is released, it must be linked as a "system image," using the command line:

```
   LINK/SYSTEM=512 EVXYZ1, EVXYZ2, ...
```

where EVXYZ1, EVXYZ2, and so on, are the source modules for program EVXYZ.

If the symbolic debugger for diagnostic programs (VDSDEBUG) is to be used during program development, another linking procedure must be used. Refer to the VAX Diagnostic Debugger User's GUide for a description of that procedure.

5.9 DEBUGGING A DIAGNOSTIC PROGRAM

Two facilities are available for aiding in debugging diagnostic programs.

The VDS command language provides several commands that are useful for debugging programs. Commands are available for examining and altering locations within the diagnostic program, setting breakpoints, and "single-stepping" through the program. Refer to the VAX Diagnostic Supervisor User's Guide for details.

More debugging capabilities are provided by the VAX Diagnostic Debugger (VDSDEBUG). This is a separate program that can run under the VDS in conjunction with a diagnostic program. It provides such features as breakpoints, watchpoints, queue traversal, referencing program locations by their symbolic names, plus examining and depositing contents of program locations as numeric data, character strings, or MACRO-32 instructions. Refer to the VAX Diagnostic Debugger User's Guide for details and operating instructions.

5.1Ø QUALITY ASSURANCE

5.1Ø.1 Quality Requirements

All diagnostic programs must meet certain quality standards. Quality standards must be met in all of the following areas before a program will be accepted as a usable product:

- Documentation quality - The diagnostic programmer must provide accurate, detailed documentation that gives both users and maintainers all the information they will need to perform their jobs. Documentation must adhere to the guidelines spelled out earlier in this chapter.

- Functional quality - The program must provide all of the functional capabilities contained in the functional specification.

- Operational quality - The program must operate in accordance with the rules set forth in this manual.

5.1Ø.1.1 Documentation Quality - Following is a list of the documention that must be provided with every diagnostic program:

1. Documentation file - The documentation file must adhere to the format presented in Appendix C.

2. Map file - For program EVXYZ, the map file EVXYZ.MAP produced by the linker must be provided.

3. Listing file including cross-reference table - For program EVXYZ, the listing file EVXYZ.LIS produced by the MACRO-32 assembler or BLISS-32 compiler must be provided. For MACRO-32 programs, a cross-reference table must be included. Within the listing, the guidelines spelled out in Section 5.4.3, Source Code Documentation, must be followed.

4. Help file - A help file must be provided. It must match the format presented in Section 5.4.4, Help Files.

5.1Ø.1.2 Functional Quality - The program developer must ensure that all functions described in the program's functional specification have been properly inplemented.

5.1Ø.1.3 Operational Quality - To guarantee the execution quality of a diagnostic program, the following steps must be performed:

1. Load and normal start

   The following steps must be performed IN THE ORDER SHOWN:

   1. Load the VDS.

   2. Issue the proper ATTACH and SELECT commands.

   3. Load and start the program with the LOAD and START commands or the RUN command.

   The program should execute without errors and stop after one program pass.

2. For EACH SECTION of the program, the following should be performed:

   - Trace mode

     Issue the SET TRACE command, then START. Check that test numbers and trace messages coincide with program documentation for the section being executed.

   - Multiple passes

     Execute the section again, specifying a pass count of at least 1Ø.

3. For EACH TEST of the program, the following steps must be performed:

   - Reverse order testing

     Execute each test, one at a time, starting with the highest-numbered test and ending with test number 1. Allow each test to complete one pass.

   - Multiple loop-on-test

     Execute each test individually, specifying a pass count of at least 1Ø.

   - Multiple loop-on-subtest

     Execute each subtest of each test individually, specifying a pass count of at least 1Ø.

   - Control-C response

     For each test, start the test and type control-C. A response to the control-C should occur within three seconds. When the VDS prompt is displayed, type CONTINUE. The program must continue from where it was interrupted and must successfully complete the pass.

   - Event flags

     Check that all event flags are used only as indicated by the program's documentation.

   - Power off

     Shut off the power for the device under test. The program must display a message stating that the device is without power.

- Write Protection

  Write-protect the device under test. Tests that write to the device should display messages indicating the that device is write-protected.

- Off line

  Place the device off-line. The program should state that the device is off-line.

- Minimum hardware configuration

  Set up a hardware configuration that matches the minimum hardware configuration specified in the functional specification. All tests must execute on this configuration.

- Maximum hardware configuration

  Set up a hardware configuration that matches the maximum hardware configuration specified in the functional specification. All tests must execute on this configuration, and all units of the device under test must be tested.

- Module extender board

  Place each logic module of the device under test on an extender board, one at a time, and verify that each test will execute successfully.

- Transportability

  Repeat all of the steps in this section on every VAX processor type on which the program is supposed to run.

- Marginal testing

  If the program has been specified to be executed successfully under marginal conditions (voltage, timing, and so on), then execute each test under these conditions.

- Error reporting and loop-on-error

  - Make sure that no $DS_ERRxxxx macros are ever executed when the cleanup code is run (Typing ABORT will cause the cleanup code to be run.)

- Set the LOOP and HALT flags. Cause every error reporting macro ($DS_ERRxxxx) to be executed. (This can be accomplished either by causing hardware failures on the device under test or by temporarily patching the program.)

- For EVERY $DS_ERRxxxx macro, do the following:

  1. CLEAR the IE1, IE2, and IE3 flags.

  2. Make sure that all error messages are printed, and that they are of the proper format (see Section 5.7.1, Error Message Formats).

  3. Make sure that the entire message has been printed before the DS> prompt is displayed.

  4. Clear the IE3 flag.

  5. Type CONTINUE, and make sure that a loop begins executing.

  6. The $DS_ERRxxxx macro should be reexecuted, but this time the third level of the error message should not be displayed.

  7. When the DS> prompt appears, clear the IE2 flag.

  8. Type CONTINUE.

  9. The $DS_ERRxxxx macro should be reexecuted, but this time the second and third levels of the error message should not be displayed.

  10. Clear the IE1 flag.

  11. Type CONTINUE.

  12. The $DS_ERRxxxx macro should be reexecuted, but this time none of the error message should be displayed.

  13. Set the IE1 flag and clear the HALT flag.

  14. Type CONTINUE.

  15. Allow the loop to execute several more times.

4. The following step must be performed for the DEFAULT section.

   - No operator

     Clear the OPERATOR flag, then execute the DEFAULT section for one pass. The program must execute successfully, and the user must not be required to type any characters on the terminal or perform any other form of manual intervention.

5. The following additional steps must be performed for programs that execute in standalone mode:

   - Memory Management on

     Turn memory management on and execute each test for several passes. Each test should execute successfully unless the program is not supposed to be executed with memory management turned on, in which case the program should abort without errors.

   - Invalid address

     Using the ATTACH command, specify an incorrect device address. The program should display a message indicating that an invalid address has been specified.

   - APT compatability

     To verify that the program will execute under the APT run-time environment, run the program under APT for eight hours.

6. The following step must be performed for programs that execute in user mode:

   - Make sure that all units are properly deallocated after the diagnostic program has finished. Issue the following VDS and VMS commands:

     1. ATTACH device-name
     2. SELECT device-name
     3. RUN program-name
     4. Type control-C
     5. ABORT
     6. Type control-Y
     7. SHOW DEVICE

None of the devices that were tested, used for error logging, or made use of in any way by the diagnostic program should be still allocated.

7. The following steps must be performed for programs that execute under CRD:

   - Issue ATTACH and SELECT commands as indicated in the program's "UUT Support Data File" for CRD.
   - Clear all VDS flags, then set any flags indicated in the UUT Support Data File.
   - Run the program for one pass, specifying the section indicated in the UUT Support Data File.
     - The time required to complete one pass should match the time specified in the UUT Support Data File.
     - The program should not request manual intervention.
   - Purposely perform incorrect device preparation (as indicated in the UUT Support Data File), and run the program. The program should display an error message describing the incorrect preparation.
   - If the UUT Support Data File specifies certain configurations of the device under test which should not affect successful execution of the program, set up these configurations and run the program. The diagnostic program should execute successfully.

5.10.2 Automated Quality Assurance

In order to aid the programmer in ascertaining the quality of a diagnostic program, the VDS provdes an automated quality assurance feature, called "Auto-QA." This feature will automatically perform some (but not all) of the quality assurance checks listed above.

Auto-QA is invoked by including the '/QA' qualifier with the RUN or START command. Operating instructions for Auto-QA are described in the VAX Diagnostic Supervisor User's Guide.

Following is a list of the quality assurance checks performed by Auto-QA. Note that Auto-QA only checks the DEFAULT program section. Quality assurance of other program sections must be performed manually.

1. Normal Start Check

This check will perform a normal load and execution of the diagnostic program with the TRACE flag set.

The program must make an error-free pass, printing out the normal trace messages and terminating with End-of-Pass. If the program does not execute an error-free pass, an appropriate QA error message will be printed. The trace messages must be visually checked by the user.

This check also makes sure that the DEFAULT section does not request input from the user. (The OPERATOR flag is cleared.)

This check is equivalent to the following sequence of VDS commands:

```
DS> CLEAR FLAG ALL
DS> SET FLAG TRACE
DS> RUN diagnostic-program-name
DS> CLEAR FLAG TRACE
```

2. Multiple Passes Check

This check will execute ten passes (by default) of the diagnostic program. The program must make ten error-free passes and terminate after the tenth pass. If this does not happen, an error message will be printed.

The number of passes executed by the diagnostic program can be changed by the user.

This check is equivalent to the following VDS command:

```
DS> START/PASS:10
```

3. Infinite Loop-On-Test Check

This check will execute each test in the diagnostic program's DEFAULT section 100 times (by default). The diagnostic must execute each test the given number of times. If the diagnostic does not execute properly, an error message will be printed.

The number of times each test is executed can be changed by the user.

This check is equivalent to the following VDS commands:

```
   DS> START/PASS:100/TEST:1:1
   DS> START/PASS:100/TEST:2:2
                .
                .
                .
   DS> START/PASS:100/TEST:n:n
```

where "n" is the highest numbered test in the DEFAULT section. The tests are executed in ascending order.

4. Infinite Loop-On-Subtest Check

This check will execute each subtest in each of the tests in the diagnostic program's DEFAULT section 100 times (by default). The program must loop on each subtest the given number of times. If the program does not execute properly, an error message will be printed.

The number of times each subtest is executed can be changed by the user.

This check is equivalent to the following Supervisor commands:

```
   DS> START/PASS:100/TEST:1:1/SUBTEST:1
   DS> START/PASS:100/TEST:1:1/SUBTEST:2
                    .
                    .
                    .
   DS> START/PASS:100/TEST:1:1/SUBTEST:m1
   DS> START/PASS:100/TEST:2:2/SUBTEST:1
   DS> START/PASS:100/TEST:2:2/SUBTEST:2
                    .
                    .
                    .
   DS> START/PASS:100/TEST:2:2/SUBTEST:m2
                    .
                    .
                    .
   DS> START/PASS:100/TEST:n:n/SUBTEST:1
   DS> START/PASS:100/TEST:n:n/SUBTEST:2
                    .
                    .
                    .
   DS> START/PASS:100/TEST:n:n/SUBTEST:mx
```

where "n" is the highest-numbered test in the DEFAULT section, and "mx" is the number of subtests in test "x."

The tests and subtests are executed in ascending order.

5. Run Individual Tests in Reverse Order Check

This check executes the tests in the diagnostic program's DEFAULT section in reverse order. This check ensures that a test does not depend on results from a previous test, and that each test is a standalone entity. If the diagnostic program does not execute properly, an error message will be printed.

This check is equivalent to the following VDS command:

```
DS> START/TEST:n:n
DS> START/TEST:n-1:n-1
          .
          .
          .
DS> START/TEST:1:1
```

where "n" starts at the highest numbered test in the DEFAULT section, and descends to the first test. That is, the tests are executed in descending order.

APPENDIX A TEMPLATE FOR THE VDS DIAGNOSTIC PROGRAM HEADER MODULE

A.1 HEADER MODULE TEMPLATE FOR MACRO-32 PROGRAMS

This is a template to aid in the development of the header module of a VAX diagnostic program. It is not intended to be a tutorial for writing the program.

Areas that must be deleted or replaced by the programmer are enclosed between matching sets of triple stars.

Areas that may be optionally modified are enclosed between matching sets of double stars.

Comments marked with one star are for information purposes and should be deleted.

```
    .TITLE   *** PROGRAM NAME ***
    .IDENT   /Ø1/
    .LIST    MEB
    .NLIST   CND
    .PSECT   HEADER, LONG, NOWRT  ;* CHANGE ALIGNMENT TO PAGE FOR DEBUG
    .DEFAULT DISPLACEMENT, WORD   ;* CHANGE THIS TO LONG FOR DEBUG

;
; COPYRIGHT (C) 1983
; DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASSACHUSETTS Ø1754
;
; 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.
;
; THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT NOTICE
; AND SHOULD  NOT BE CONSTRUED  AS A COMMITMENT  BY DIGITAL  EQUIPMENT
; CORPORATION.
;
; DEC ASSUMES  NO  RESPONSIBILITY  FOR  THE USE OR  RELIABILITY OF ITS
; SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DEC.
;

;++
; FACILITY:         VAX DIAGNOSTIC.
;
; ABSTRACT:         *** Short description of program. ***
;
; ENVIRONMENT:      VAX DIAGNOSTIC SUPERVISOR.
;
; AUTHOR:  *** NAME        DATE ***        VERSION Ø1.
;
; MODIFIED BY:
;--
```

```
   .PAGE
   .SBTTL  DECLARATIONS
;
; INCLUDE FILES:
;

.LIBRARY   \SYS$LIBRARY:DIAG.MLB\  ; VAX FAMILY DIAGNOSTIC LIBRARY.
;** Declare programmer-defined libraries here.
;*  (Libraries are searched in reverse to the order listed.)
;
; MACROS:
;
;*** USER MACROS (OPTIONAL). ***


;
; EQUATED SYMBOLS:
;
   $DS_BGNMOD      *** ENVIRONMENT ***
   $DS_DSSDEF      GLOBAL          ;SUPERVISOR SERVICE ENTRY VECTORS

;*** USER EQUATED SYMBOLS ***
```

```
   .PAGE
   .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(HEX).
;
;--
   $DS_HEADER      <***PROGNAME***>, REV=01, UPDATE=0, NUNIT=**1**


   .SBTTL  DISPATCH TABLE.
;+
;
;  THE DISPATCH TABLE IS A COLLECTION OF ADDRESSES GENERATED AT THE
;  BEGINNING OF EACH TEST AND GROUPED TOGETHER INTO A CONTIGUOUS
;  LIST BY THE LINKER.  THIS IS DONE BY DEFINING A PSECT CALLED
;  DISPATCH.
;
;-

   $DS_DISPATCH
   .PAGE
   .SBTTL  PROGRAM GLOBAL DATA SECTION.
   .PSECT  DATA,LONG
;++
; FUNCTIONAL DESCRIPTION:
;
;***       ALL DYNAMICALLY MODIFIED DATA SHOULD BE PLACED IN THIS SECTION. ***
;***       THIS IS THE ONLY PSECT WHICH WILL NORMALLY BE WRITE ENABLED. ***
;
;--

;+
; STATISTICS TABLE.
;-
   $DS_BGNSTAT
   $DS_ENDSTAT

;*** OTHER GLOBAL DATA (OPTIONAL). ***
```

```
   .PAGE
   .SBTTL  PROGRAM TEXT SECTION.
;++
; FUNCTIONAL DESCRIPTION:
;
; THIS SECTION CONTAINS ALL OF THE DATA STRUCTURES THAT ARE MADE UP OF
; CHARACTER STRINGS.
;--

;+
; PROGRAM SECTION NAMES.
;-
   $DS_SECTION     <*** SECTION NAMES ***>

;+
; DEVICE MNEMONICS LIST.
;-
T_DEVICE:
   $DS_DEVTYP      <*** DEVICES ***>

;+
; NAMES OF DEVICE REGISTERS AND BIT MNEMONICS
;-
;*** ASCII NAMES OF DEVICE REGISTERS AND THEIR BITS (OPTIONAL) FOR  ***
;*** USE WITH THE $DS_CVTREG MACRO ROUTINE. ***

;+
; FORMATTED ASCII OUTPUT STATEMENTS.
;-
;*** MESSAGES TO THE OPERATOR, ETC. (OPTIONAL). ***

;+
; STRINGS USED TO REPORT ERRORS
;-
;*** ERROR REPORT MESSAGES. (OPTIONAL) ***
```

```
   .PAGE
   .SBTTL  INITIALIZATION CODE.
;++
; FUNCTIONAL DESCRIPTION.
;
;  THIS ROUTINE WILL BE EXECUTED AT THE BEGINNING OF EACH PASS.
;***        DESCRIPTION OF YOUR ROUTINE. ***
;
; CALLING SEQUENCE:
;
;  THE DIAGNOSTIC SUPERVISOR CALLS THIS ROUTINE WITH A CALLG INSTRUCTION.
;
; INPUT PARAMETERS:
;
;  ** NONE **
;
; IMPLICIT INPUTS:
;
;  ** NONE **
;
; OUTPUT PARAMETERS:
;
;  ** NONE **
;
; IMPLICIT OUTPUTS:
;
;  ** NONE **
;
; COMPLETION CODES:
;
;  ** NONE **
;
; SIDE EFFECTS:
;
;  ** NONE **
;
;--

   $DS_BGNINIT
;*** DEVICE INITIALIZATION CODE. ***
   $DS_ENDINIT
   .PAGE
   .SBTTL  CLEAN-UP CODE.
;++
; FUNCTIONAL DESCRIPTION:
;
;  THIS ROUTINE IS EXECUTED AT THE COMPLETION OF THE LAST PROGRAM PASS.
;***        DESCRIPTION OF YOUR ROUTINE. ***
;
; CALLING SEQUENCE:
;
;  THE DIAGNOSTIC SUPERVISOR CALLS THIS ROUTINE WITH A CALLG INSTRUCTION.
;
; INPUT PARAMETERS:
;
;  ** NONE **
;
; IMPLICIT INPUTS:
;
;  ** NONE **
;
; OUTPUT PARAMETERS:
;
;  ** NONE **
;
; IMPLICIT OUTPUTS:
;
;  ** NONE **
;
; COMPLETION CODES:
;
;  ** NONE **
;
; SIDE EFFECTS:
;
;  ** NONE **
;
;--

   $DS_BGNCLEAN
;*** DEVICE "SHUT-DOWN" CODE. ***
   $DS_ENDCLEAN
   .PAGE
   .SBTTL  SUMMARY REPORT CODE.
;++
; FUNCTIONAL DESCRIPTION:
;
;  THIS ROUTINE ISSUES A SUMMARY REPORT UPON REQUEST FROM THE
;  OPERATOR OR WHEN A $DS_SUMMARY_G CALL IS MADE.
;***       DESCRIPTION OF YOUR ROUTINE. ***
;
; CALLING SEQUENCE:
;
;  THE DIAGNOSTIC SUPERVISOR CALLS THIS ROUTINE WITH A CALLG INSTRUCTION.
;
; INPUT PARAMETERS:
;
;  ** NONE **
;
; IMPLICIT INPUTS:
;
;  ** NONE **
;
; OUTPUT PARAMETERS:
;
;  ** NONE **
;
; IMPLICIT OUTPUTS:
;
;  ** NONE **
;
; COMPLETION CODES:
;
;  ** NONE **
;
; SIDE EFFECTS:
;
;  ** NONE **
;
;--
   $DS_BGNSUMMARY
;*** SUMMARY REPORT CODE. (OPTIONAL) ***
   $DS_ENDSUMMARY
   .SBTTL  GLOBAL SUBROUTINES.

;***       OPTIONAL GLOBAL SUBROUTINES, SUCH AS ERROR REPORTING ROUTINES,
   INTERRUPT SERVICE ROUTINES, CONDITION HANDLERS, ETC.

;++
; FUNCTIONAL DESCRIPTION:
;
;
; CALLING SEQUENCE:
;
;  ** NONE **
;
; INPUT PARAMETERS:
;
;  ** NONE **
;
; IMPLICIT INPUTS:
;
;  ** NONE **
;
; OUTPUT PARAMETERS:
;
;  ** NONE **
;
; IMPLICIT OUTPUTS:
;
;  ** NONE **
;
; COMPLETION CODES:
;
;  ** NONE **
;
; SIDE EFFECTS:
;
;  ** NONE **
;
; REGISTERS USED:
;
;  ** NONE **
;--
;***
   $DS_ENDMOD
   .END
```

A.2 HEADER MODULE TEMPLATE FOR BLISS-32 PROGRAMS

This is a template to aid in the development of the header module of a VAX diagnostic program. It is not intended to be a tutorial for writing the program.

Areas that must be deleted or replaced by the programmer are enclosed between matching sets of triple stars.

Areas that may be optionally modified are enclosed between matching sets of double stars.

```
%TITLE '*** title ***'
MODULE  *** module_name ***   (
             IDENT = '01-00'
             ) =
BEGIN

!++
!                    COPYRIGHT (c) 1983 BY
!         DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASS.
!
! THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED AND  COPIED
! ONLY  IN  ACCORDANCE  WITH  THE  TERMS  OF  SUCH  LICENSE AND WITH THE
! INCLUSION OF THE ABOVE COPYRIGHT NOTICE.  THIS SOFTWARE OR  ANY  OTHER
! COPIES  THEREOF MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY
! OTHER PERSON.  NO TITLE TO AND OWNERSHIP OF  THE  SOFTWARE  IS  HEREBY
! TRANSFERRED.
!
! THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE  WITHOUT  NOTICE
! AND  SHOULD  NOT  BE  CONSTRUED  AS  A COMMITMENT BY DIGITAL EQUIPMENT
! CORPORATION.
!
! DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE OR  RELIABILITY  OF  ITS
! SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DIGITAL.
!
!--

!++
! FACILITY:      VAX-11 DIAGNOSTIC
!
!
! ABSTRACT:      *** abstract ***
!
!
! ENVIRONMENT:   VAX-11 DIAGNOSTIC SUPERVISOR
!
!
! AUTHOR: *** your name ***, DATE: *** date ***, VERSION: V01.0
!
! MODIFIED BY:
!
!--
```

```
%SBTTL 'Declarations'

!++
!   TABLE OF CONTENTS:
!--

FORWARD ROUTINE
    *** routine names *** ;

!++
!   EXTERNAL DECLARATIONS:
!--

EXTERNAL ROUTINE
    *** routine names *** ;                                 ! In module...

!++
!   INCLUDE FILES:
!--
 *** Declare user-defined libraries and "require" files here ***
LIBRARY 'SYS$LIBRARY:STARLET';              ! VMS MACRO LIBRARY
LIBRARY 'SYS$LIBRARY:DIAG';                 ! VAX DIAGNOSTIC FAMILY LIBRARY

!++
!   MACRO DEFINITIONS:
!--

MACRO
    *** OPTIONAL USER-WRITTEN MACROS *** %;

!++
!   DIAGNOSTIC SUPERVISOR MACROS:
!--

$DS_BGNMOD (ENV = *** environment ***);
$DS_DISPATCH;
GLOBAL $DS_DSSDEF;
$DS_DSADEF;

!++
!   PROGRAM SECTION NAMES:
!--

$DS_SECTION (*** section names ***);

!++
!   DEVICE MNEMONICS LIST:
!--

$DS_DEVTYP (STRINGS = (*** device types ***),
       ADDRESSES = (*** addresses of PT-desc ***));
```

```
%SBTTL 'Program Header Data Block'

!++
!   FUNCTIONAL DESCRIPTION:
!
!         The progra header data block contains the parameters which
!         allows the Diagnostic Supervisor to control the program.
!         The Diagnostic Supervisor looks for the header information
!         beginning at virtual address 200(HEX).
!--

$DS_HEADER (PNAME ='*** program name ***',
            REV = 1,
            UPDATE = Ø,
            NUNIT = *** number of units ***);
```

```
%SBTTL 'Program Global Data Section'

!++
! FUNCTIONAL DESCRIPTION:
!
!***       ALL DYNAMICALLY MODIFIED DATA SHOULD BE PLACED IN THIS SECTION. ***
!
!--

!++
! DEVICE REGISTER CONTENTS TABLE
!--

$DS_BGNREG;
$DS_ENDREG;

!++
! STATISTICS TABLE.
!--

$DS_BGNSTAT;
$DS_ENDSTAT;

!++
!   EQUATED SYMBOLS:
!--

GLOBAL LITERAL
    *** enter literals *** ;

!++
!   OWN STORAGE:
!--

GLOBAL
    *** enter variables *** ;
```

```
%SBTTL 'Program Text Section'

!++
!   FUNCTIONAL DESCRIPTION:
!
!           This section contains all the character strings.
!
!--


!++
!   NAMES OF DEVICE REGISTERS AND BIT MEMONICS:
!--

GLOBAL BIND
    *** ascii names of device registers and their bits (optional)
   for use with the $DS_CVTREG macro rotine ***

!++
!   FORMATTED ASCII OUTPUT STATEMENTS:
!--

    *** messages to the operator, etc. (optional) ***

!++
!   STRINGS USED TO REPORT ERRORS
!--

    *** enter statements ***;
```

```
%SBTTL 'Initialization Code'

!++
!   FUNCTIONAL DESCRIPTION:
!
!         This routine will be executed at the beginning of each pass
!         of the diagnostic.
!
!   FORMAL PARAMETERS:
!
!         ** NONE **
!
!   IMPLICIT INPUTS:
!
!         ** NONE **
!
!   IMPLICIT OUTPUTS:
!
!         ** NONE **
!
!   COMPLETION CODES:
!
!         ** NONE **
!
!   SIDE EFFECTS:
!
!         ** NONE **
!
!--

$DS_BGNINIT;
BEGIN

*** initialization code ***

END;
$DS_ENDINIT;
```

```
%SBTTL 'Clean-up Code'

!++
!   FUNCTIONAL DESCRIPTION:
!
!         The cleanp-up code is executed at the completion of the last
!    program pass.
!    *** Description of your routine goes here. ***
!
!   FORMAL PARAMETERS:
!
!         ** NONE **
!
!   IMPLICIT INPUTS:
!
!         ** NONE **
!
!   IMPLICIT OUTPUTS:
!
!         ** NONE **
!
!   COMPLETION CODES:
!
!         ** NONE **
!
!   SIDE EFFECTS:
!
!         ** NONE **
!
!--

$DS_BGNCLEAN;
BEGIN

*** cleanup code ***

END;
$DS_ENDCLEAN;
```

```
%SBTTL 'Summary Report Code'

!++
!   FUNCTIONAL DESCRIPTION:
!
!         This routine displays a summary report when the operator types
!    a SUMMARY command or when a $DS_SUMMARY call is issued.
!  *** Description of the summary routine goes here. ***
!
!   FORMAL PARAMETERS:
!
!         ** NONE **
!
!   IMPLICIT INPUTS:
!
!         ** NONE **
!
!   IMPLICIT OUTPUTS:
!
!         ** NONE **
!
!   COMPLETION CODES:
!
!         ** NONE **
!
!   SIDE EFFECTS:
!
!         ** NONE **
!
!--

$DS_BGNSUMMARY;
BEGIN

*** summary code ***

END;
$DS_ENDSUMMARY;
```

```
! *** Optional global subroutines, such as error reporting routines,
!     interrupt service routines, condition handlers, etc, should
!     be placed here. ***

%SBTTL 'Global Subroutines'

!++
!   FUNCTIONAL DESCRIPTION:
!
!
!
!   FORMAL PARAMETERS:
!
!         ** NONE **
!
!   IMPLICIT INPUTS:
!
!         ** NONE **
!
!   IMPLICIT OUTPUTS:
!
!         ** NONE **
!
!   COMPLETION CODES:
!
!         ** NONE **
!
!   SIDE EFFECTS:
!
!         ** NONE **
!
!   REGISTERS USED:
!
!    %SBTTL 'Summary Report Code'

!++
!   FUNCTIONAL DESCRIPTION:
!
!
!
!   FORMAL PARAMETERS:
!
!         ** NONE **
!
!   IMPLICIT INPUTS:
!
!         ** NONE **
!
!   IMPLICIT OUTPUTS:
!
!         ** NONE **
!
```

```
!    COMPLETION CODES:
!
!          ** NONE **
!
!    SIDE EFFECTS:
!
!          ** NONE **
!
!    REGISTERS USED:
!
!     ** NONE **
!
!--


$DS_ENDMOD;
END
ELUDOM
```

APPENDIX B TEMPLATE FOR VDS DIAGNOSTIC PROGRAM TEST MODULES

B.1 TEST MODULE TEMPLATE FOR MACRO-32 PROGRAMS

This is a template to aid in the development of a test module of a VAX diagnostic. It is not intended to be a tutorial for writing the program.

Areas that must be deleted or replaced by the programmer are enclosed between matching sets of triple stars.

Areas that may be optionally modified are enclosed between matching sets of double stars.

Comments that contain only one star are for informational purposes and should be deleted.

```
    .TITLE  *** PROGRAM MODULE NAME ***
    .IDENT  /Ø1/                ;***  VERSION NUMBER ***
    .LIST   MEB
    .NLIST  CND
    .DEFAULT DISPLACEMENT, WORD ;* CHANGE THIS TO LONG FOR DEBUG

;++
; COPYRIGHT (C) 198Ø
; DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASSACHUSETTS Ø1754
;
; 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.
;
; THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT NOTICE
; AND SHOULD  NOT BE CONSTRUED  AS A COMMITMENT  BY DIGITAL  EQUIPMENT
; CORPORATION.
;
; DEC ASSUMES  NO  RESPONSIBILITY  FOR  THE USE OR  RELIABILITY OF ITS
; SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DEC.
;++

;++
; FACILITY:         VAX DIAGNOSTIC.
;
; ABSTRACT:         *** Short description of this module. ***
;
; ENVIRONMENT:      VAX DIAGNOSTIC SUPERVISOR.
;
; AUTHOR:  *** NAME        DATE ***        VERSION Ø1.
;
; MODIFIED BY:
;--
```

```
   .PAGE
   .SBTTL  DECLARATIONS
;+
; INCLUDE FILES:
;-

.LIBRARY   \SYS$LIBRARY:DIAG.MLB\  ; VAX FAMILY DIAGNOSTIC LIBRARY
;** List programmer-defined libraries here.
;** (Libraries are searched in reverse order.)

;+
; MACROS:
;-

;*** PROGRAMMER-DEFINED MACROS (OPTIONAL). ***

;+
; EQUATED SYMBOLS:
;-

;*** SYMBOLS FOR LOCAL USE AND SUPERVISOR INTERFACE ***
;*** AND USER EQUATED SYMBOLS (OPTIONAL). ***

$DS_BGNMOD <*** ENVIRONMENT ***>,TEST=*** NUMBER OF FIRST TEST IN MODULE***

$DS_CHDEF  GLOBAL           ; CHANNEL SERVICE SYMBOLS (LEVEL 3)
$DS_DSSDEF GLOBAL           ; SUPERVISOR SERVICE ENTRY VECTORS

;+
; PROGRAMMER-DEFINED LOCAL AND GLOBAL STORAGE
;-


;+
; SECTION DEFINITIONS:
;-
   $DS_SECDEF      <*** SECTION NAMES ***>
   .PAGE
   $DS_SBTTL        <*** TEST NAME ***>
;++
; TEST DESCRIPTION:
;
;  THIS WILL CONTAIN A BRIEF DESCRIPTION OF WHAT IS BEING TESTED
;  AND HOW THE TEST IS IMPLEMENTED.
;
; ASSUMPTIONS:
;
;  *** ASSUMPTIONS MADE BEFORE THE TEST IS RUN, SUCH AS
;      WHAT PARTS OF THE HARDWARE MUST BE FUCTIONING PROPERLY
;      BEFORE THIS TEST IS EXECUTED. ***
;
; TEST STEPS:
;
;  *** DETAILED DISCRIPTION OF THE TEST AND TEST FLOW ***
;  1) FIRST STEP, INITIALIZATION
;  2) SECOND STEP
;  3) THIRD STEP
;
; ERRORS:
;
;  *** DETAILED DISCRIPTION OF THE ERRORS DETECTABLE AND REPORTED ***
;  ERROR Ø1:
;  ERROR Ø2:
;  ERROR Ø3:
;
; DEBUG:
;
;  THIS SECTION WILL CONTAIN INSTRUCTIONS ON HOW TO USE THIS
;  TEST IN DEBUGGING THE UNIT UNDER TEST.
;
;--
   $DS_BGNTST       <*** SECTION NAMES ***>,ALIGN=BYTE ;* CHANGE THIS TO
                                                       ;*   PAGE FOR DEBUG

;+
; BLOCK COMMENTS TO EXPLAIN WHAT A SPECIFIC BLOCK OF CODE
; IS DOING
;-
```

```
;+
; SUBTEST DESCRIPTION:
;
;  *** BRIEF DESCRIPTION OF WHAT THE SUBTEST CHECKS ***
;
; SUBTEST STEPS:
;
;  *** DETAILED FLOW OF TEST SEQUENCE ***
;
; ERRORS:
;
;  *** BRIEF DESCRIPTION OF EACH OF THE ERRORS
;      THAT CAN BE DETECTED BY THIS TEST ***
;
; DEBUG:
;
;  *** HELPFUL HINTS FOR TRACKING HARDWARE FAULTS ***
;
;-
   $DS_BGNSUB




;+
; BLOCK COMMENT
;-


   $DS_ENDSUB
   $DS_ENDTEST
   $DS_ENDMOD
   .END
```

B.2 TEST MODULE TEMPLATE FOR BLISS-32 PROGRAMS

This is a template to aid in the development of the header module of a VAX diagnostic program. It is not intended to be a tutorial for writing the program.

Areas that must be deleted or replaced by the programmer are enclosed between matching sets of triple stars.

Areas that may be optionally modified are enclosed between matching sets of double stars.

```
%TITLE '*** title ***'
MODULE   *** module_name ***   (
             IDENT = '01-00'
             ) =
BEGIN

!++
!                      COPYRIGHT (c) 1983 BY
!         DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASS.
!
! THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED AND  COPIED
! ONLY  IN  ACCORDANCE  WITH  THE  TERMS  OF  SUCH  LICENSE AND WITH THE
! INCLUSION OF THE ABOVE COPYRIGHT NOTICE.  THIS SOFTWARE OR  ANY  OTHER
! COPIES  THEREOF MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY
! OTHER PERSON.  NO TITLE TO AND OWNERSHIP OF  THE  SOFTWARE  IS  HEREBY
! TRANSFERRED.
!
! THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE  WITHOUT  NOTICE
! AND  SHOULD  NOT  BE  CONSTRUED  AS  A COMMITMENT BY DIGITAL EQUIPMENT
! CORPORATION.
!
! DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE OR  RELIABILITY  OF  ITS
! SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DIGITAL.
!
!--

!++
! FACILITY:     VAX-11 DIAGNOSTIC
!
!
! ABSTRACT:     *** abstract ***
!
!
! ENVIRONMENT:  VAX-11 DIAGNOSTIC SUPERVISOR
!
!
! AUTHOR: *** your name ***, DATE: *** date ***, VERSION: V01.0
!
! MODIFIED BY:
!
!--
```

```
!++
!    INCLUDE FILES:
!--

*** List all programmer-defined libraries and "require" files here. ***
LIBRARY 'SYS$LIBRARY:DIAG';

!++
!    SUPERVISOR MACROS
!--

$DS_BGNMOD (ENV = *** environment ***, TEST = *** starting test number ***);
$DS_DSADEF;
$DS_DSSDEF;
$DS_SECDEF (*** section names ***);

!++
!    EXTERNAL DECLARATIONS
!--

EXTERNAL ROUTINE
    *** routine name *** ;

EXTERNAL
    *** names *** ;
```

```
%SBTTL '*** subtitle ***'
!++
!   TEST DESCRIPTION:
!
!         *** This will contain a brief description of what is being tested
!         and how the test is implemented.  ***
!
!   ASSUMPTIONS:
!
!         *** Assumptions made before the test is run, such as
!        which portions of the hardware must be functioning
!        properly before this test is executed. ***
!
!   TEST STEPS:
!         *** Detailed description of the test and test flow ***
!         1)   *** First step, Initialization ***
!         2)   *** Second step ***
!         3)   *** Third step ***
!
!   ERRORS:
!         *** Detailed description of the errors detectable and reported ***
!         Error Ø1:  *** description ***
!         Error Ø2:  *** description ***
!         Error Ø3:  *** description ***
!
!   DEBUG:
!
!         *** This section will contain instructions on how to use this
!         test in debugging the unit under test. ***
!
!
!--

$DS_BGNTEST (SECTION = ***  section names  ***,
           TEST = '***  test name  ***');

!++
!   *** Block comment to explain what a specific block
!       of code is doing ***
!--

BEGIN
```

```
%SBTTL '*** subtitle ***'

!++
!   SUBTEST DESCRIPTION:
!
!         *** Brief description of what the subtest checks ***
!
!   SUBTEST STEPS:
!
!         *** Detailed flow of test sequence ***
!
!   ERRORS:
!
!         *** Brief description of each of the possible errors detected ***
!
!   DEBUG:
!
!         *** Helpful hints for tracking hardware faults ***
!
!
!--

$DS_BGNSUB;

!++
!   *** Block comment to explain what a specific block
!       of code is doing ***
!--

BEGIN
*** subtest code ***
END;

$DS_ENDSUB;


END;

$DS_ENDTEST;

$DS_ENDMOD;
END
ELUDOM
```

APPENDIX C TEMPLATE FOR DIAGNOSTIC PROGRAM DOCUMENTATION FILE

This is a template for VAX diagnostic documentation files. Everything to be changed, added, or deleted is enclosed in matching double angle brackets, '<<' and '>>'.

APPENDIX

IDENTIFICATION

Product code:	ZZ-<< maindec code, including version >>
Product name:	<< program name >>
Product date:	<< submission date >>
Maintainer:	<< diagnostic engineering group >>

APPENDIX

Table of Contents

C.1 ABSTRACT

<< program abstract; from 3 to 2Ø lines

C.2 HARDWARE REQUIREMENTS

<< minimum hardware configuration; optional hardware

C.3 SOFTWARE REQUIREMENTS

<< software environment, e.g. VAX Diagnostic Supervisor

C.4 PREREQUISITES

<< hardware that should be verified before running this program

C.5 OPERATING INSTRUCTIONS

<< Refer to the "VAX-11 Diagnostic System User's Guide" (EK-DS78Ø-UG-ØØ2) for instructions on how to load and start the Diagnostic Supervisor and how to load and execute programs under the Diagnostic Supervisor. The operator must ATTACH and SELECT the device << e. g., KA78Ø before starting this program.

C.5.1 Options

<< any operator options, such as MANUAL section

C.5.2 Event Flags

<< The following event flags are used by this program.

1. <<event flag 1

2. <<event flag 2

3. << etc.

C.6 PROGRAM FUNCTIONAL DESCRIPTION

C.6.1 Program Overview

<< purpose, strategy, transportability

C.6.2 Program Size

<< names and sizes of all associated files

C.6.3 Program Run Times

<< quick verify, default, with options

C.6.4 Run-time Dynamics

<< memory allocations, side effects, sequence of testing on multilpe units

C.6.5 Fault Detection

<< error resolution, error message formats, fault coverage (%)

C.6.6 Performance During Hardware Failures

<< unsuspected traps, power failure

C.6.7 Program Applications

<< field service (RD), manufacturing (APT), customers, engineering

C.6.8 Test Descriptions

<< for each test/subtest, "Test description", "Test steps", and "Debug aids"

C.7 MAINTENANCE HISTORY

<< date, version: description of changes

APPENDIX D
SAMPLE HELP FILE

```
1 ATTACH
 The CPU  must  be  attached.   Type  "HELP  DEVICE  KA780"  for
 information on a VAX-11/780.  A VAX-11/750 CPU is a KA750, etc.

 Example:
    ATTACH KA780 SBI KA0 NO NO 0 0

1 HELP
 This program exercises  the  VAX  native  mode  floating  point
 instruction  set, which  can  be  executed  in any mode, i.e.,
 non-priviledged  instructions.   The  program  is  capable   of
 running   under   the   Diagnostic  Supervisor  in  either  the
 standalone environment or as a user task under VMS.  It is also
 designed to run on any member of the VAX family of computers.

1 DEVICE

2 KA730
 Type "HELP DEVICE KA730" for more information.

2 KA750
 Type "HELP DEVICE KA750" for more information.

2 KA780
 Type "HELP DEVICE KA780" for more information.

1 EVENT
 The following event flags have the described  effects  on  this
 program:

 Event Flag 2: Disable  the interval  timer  interrupting  during
 instruction execution.

 Event Flag 3: Enable the  interval timer interrupting while page
 faulting  is also enabled.

 Event Flag 4: Enable the continuation  of  a  subtest  after  an
 error (normally the subtest is aborted).

1 QUICK
 The QUICK flag disables the exection of the  instructions  with
 page  faulting  or  interrupting, so that each instruction test
 case  is  only  executed  once  for  each  addressing  mode
 combination.

1 SECTIONS

2 DEFAULT
```

The DEFAULT section includes all of the tests making up the other four sections.

2 F_FLOATING
Single Precision Floating Point Instructions: MOVF, MNEGF, CVTBF, CVTWF, CVTLF, CVTFB, CVTFW, CVTFL, CVTRFL, CMPF, TSTF, ADDF2, ADDF3, SUBF2, SUBF3, MULF2, MULF3, DIVF2, DIVF3, EMODF, and POLYF.

2 D_FLOATING
Double Precision Floating Point Instructions: MOVD, MNEGD, CVTBD, CVTWD, CVTLD, CVTDB, CVTDW, CVTDL, CVTRDL, CVTFD, CVTDF, CMPD, TSTD, ADDD2, ADDD3, SUBD2, SUBD3, MULD2, MULD3, DIVD2, DIVD3, EMODD, and POLYD.

2 G_FLOATING
Extended Range Double Precision Floating Point Instructions: MOVG, MNEGG, CVTBG, CVTWG, CVTLG, CVTGB, CVTGW, CVTGL, CVTRGL, CVTFG, CVTGF, CMPG, TSTG, ADDG2, ADDG3, SUBG2, SUBG3, MULG2, MULG3, DIVG2, DIVG3, EMODG, and POLYG.

2 H_FLOATING
Extended Range Quadruple Precision Floating Point Instructions: MOVH, MNEGH, CVTBH, CVTWH, CVTLH, CVTHB, CVTHW, CVTHL, CVTRHL, CVTFH, CVTDH, CVTGH, CVTHF, CVTHD, CVTHG, CMPH, TSTH, ADDH2, ADDH3, SUBH2, SUBH3, MULH2, MULH3, DIVH2, DIVH3, EMODH, and POLYH.

1 SUMMARY
The summary report gives an error count by test number. No report is generated if there were no errors.

INDEX

VAX DIAGNOSTIC DESIGN GUIDE **Reader's Comments**

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 *Documentation Products Directory*, 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
Accessories and Supplies Group
P.O. Box CS2008
Nashua, New Hampshire 03061

Attention: Documentation Products
Telephone: 1-800-258-1710

Order No. EK-1VAXD-TM

ZKO

Fold Here

Do Not Tear – Fold Here and Staple

digital

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 33 MAYNARD, MA.

POSTAGE WILL BE PAID BY ADDRESSEE

32-Bit Systems Diagnostic Engineering, TWO/F17
Digital Equipment Corporation
1925 Andover Street
Tewksbury, MA 01876

EK-1VAXD-TM-003

VAX Diagnostic
Design Guide

IMPORTANT

To automatically receive updates of this manual, fill out the following information:

Internal:	External:
Name__________	Name__________
Group__________	Title__________
Mail Stop__________	Company__________
DTN__________	Street __________
	City__________
	State/Country__________
	Zip__________

Fold Here

Do Not Tear – Fold Here and Staple

digital

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 33 MAYNARD, MA.

POSTAGE WILL BE PAID BY ADDRESSEE

Digital Equipment Corporation
Educational Services/Quality Assurance
12 Crosby Drive, BU/E08
Bedford, MA 01730