System Features to Aid in the On-line Diagnosis of Computer Peripherals

by

Steven Douglas Krueger

Submitted in Partial Fulfillment of the Requirements
for the Degrees of Master of Science and Bachelor of Science

at the

Massachusetts Institute of Technology

June 1980

Signature of Author _____
Department of Electrical Engineering and Computer Science
June, 1980

Certified by _____
Prof. J. H. Saltzer, Thesis Supervisor

And _____
Company Thesis Supervisor

Accepted by _____
Chairman, Departmental Committee on Graduate Students

System Features to Aid in the On-line Diagnosis of Computer Peripherals

by

Steven Douglas Krueger

Submitted to the
Department of Electrical Engineering and Computer Science
on May 19, 1980 in partial fulfillment of the
requirements for the Degrees of
Master of Science and Bachelor of Science.

ABSTRACT

A set of operating system features are developed so that peripheral diagnostic programs may run under control of a general purpose operating system. Such a program is termed an "on-line diagnostic". The hardware and software environment for on-line diagnosis is modeled for typical computer systems. A new class of virtual resource for the device-under-test is developed and its characteristics identified. A manager for device-under-test virtual resources is described and its interfaces to the on-line diagnostic program are discussed. A trial implementation on the Hewlett-Packard HP 300 small business computer is discussed and evaluated.

Thesis Supervisor: Dr. J. H. Saltzer
Title: Professor of Computer Science

Acknowledgements

Thanks are in order to the many people who were of help to me in the course of this work and in the writing of this thesis. Foremost in my mind are the Hewlett-Packard Company, General Systems Division and its people. The research for this thesis was conducted while on VI-A assignment at HP. Many people there were of great help and encouragement to me. I am especially greatful to Carson Kan who supervised my work. He gave me enough supervision to keep my progress forward and enough freedom to perform seemingly useless subtasks, many of which later proved their worth. I am also especially greatful to Roger Ruhnow, my office mate and later house mate, who often listened to my babble.

I also thank Professor Jerome Saltzer, my thesis supervisor. Professor Saltzer has, on many occasions, displayed the uncanny ability to ask just the right question to crystalize my hopelessly jumbled thoughts into a clear idea. I also appreciate the several pep talks that restored my faltering confidence.

Special thanks also go to the staff of 6.032, Professors Steve Ward and Bert Halstead, my fellow teaching assistants Mike Patrick, Jon Powell, and Jim Troisi, and course secretary Eva Tervo. With out their help to share the load at a few critical moments I could not have succeeded.

And last, to Rich Kuntz, who provided inspiration both to impede and expedite the completion of this thesis.

Table of Contents

Chapter 1: Introduction

In this thesis, features are identified and a design method developed so that peripheral programs can be run under control of a general purpose operating system. We call such a program an on-line diagnostic. The basic strategy for on-line diagnosis described here is compatible with typical general purpose computer systems available today, as a goal was to describe a strategy that could be added to existing systems to extend their serviceability. A trial implementation of this scheme to show its viability is described.

The source of both the difficulty and the power of on-line diagnosis is its execution environment. The environment is a general purpose computer that supports multiprocessing. The nature of other activities proceeding concurrently in the system is not known. Additionally, the operating system provides a number of services that are available to the application. The most important services for our purpose are virtual input and output devices that have more uniform characteristics and are "better behaved" than the physical devices that implement them. This is especially true in the case of errors and malfunctions. For on-line diagnosis information about the nature of each error or malfunction must be available to the testing program. One of the functions of "well behaved" virtual devices is that much error information is hidden so that the virtual devices appear more uniform to the application. This hiding of error information is a difficulty in on-line diagnosis. The types of devices that may be accessed ranges from terminals to communications lines to disk storage.

The power of on-line diagnosis stems from two features of its environment. Since the computer system can perform many activities concurrently, diagnosis can be performed while other activities continue. On-line diagnosis does use system

resources, decreasing the availability of resources for other activities and thus delaying the completion of some activities. It does not stop all other activities, however, in contrast with other diagnostic strategies. The second source of power for on-line diagnosis is the availability of system services. The availability of these resources gives the diagnostic application programmer a powerful environment for the execution diagnostics.

1.1: Nature of the problem

The problem is to identify system features necessary to support on-line diagnostics. The goal is to provide an interface through which an application program running under the operating system can perform diagnosis of a peripheral device. Some of these features will be implemented in the hardware of the peripherals, others in the operating system software.

The key issue is disruption of other activities performed concurrently in the system. Disruption can result from access timing conflicts at the peripheral device, destruction of data stored in the peripheral that is used by some other activity, or destruction of data used by the operating system to implement services. Another issue of concern is providing a means of communicating device-specific information to the application in a system designed to hide device-dependant details.

1.2: Relationship to other diagnostic and service strategies

Since the first computers, failures in the hardware of the system have decreased system availability, to the dismay of users. Manufacturers have improved availability over the years by attacking the two components of availability, reliable operation and timely repair. Reliability has improved through advances in component technology and through robust design. Timely repair is based on efficient operations (according to Fitzsimons, "have the right man with the right data and the right part in the right place at the right time") and on serviceability of the computer system.

Serviceability has been enhanced through advances in packaging and modularity of design. But the biggest advances in serviceability have been from the area of diagnostic tools. Diagnostic tools give the service technician powerful help in locating a faulty component or subassembly, hence aiding him in quickly repairing the computer system.

Test and diagnostic programs have long been a part of the service strategy for computer systems, but these programs have most often been designed to make exclusive use of the computer system. These programs execute in an environment without other activities, so that no other work is done by the computer system during diagnosis. Furthermore, switching from the normal environment to the diagnostic environment is itself disruptive.

We explore, here, the relationship between current diagnostic and service strategies and on-line diagnosis. The present diagnostic strategy can be viewed as a hierarchy of three levels. At the highest level a special set of diagnostic programs uses a subset of the system to diagnose the remainder of the system. This requires an operator with a small amount of training. At the next lowest

level, a skilled technician operates the self-testing capability of a subsystem, requiring no specialized knowledge to interpret whether the subsystem passed or failed the test. At the lowest level, diagnosis is performed by a highly trained technician using test equipment and specialized knowledge.

More should be said about self-test. Self-test is a comprehensive, low level, functional test that a subsystem can perform internally. With the integration of microprocessors into subsystems, particularly into I/O devices, self-test can now easily be incorporated into most of the subsystems of a computer system. Self-test is entirely internal to the subsystem, requiring only power, to test the subsystem. This means that each subsystem may be tested individually to hasten system integration or to aid in diagnosis of multiple faults. Self-test is initiated in several ways. In is initiated automatically at power-on. It can be initiated manually from switches and testpoints internal to the subsystem. Or, for peripherals, it can be initiated by a command from the computer.

There are several features of the present diagnostic method that should be noted. Each level is less costly than the one below it because each higher level requires less time and less expertise to diagnose. The lowest level is slow and requires a highly trained technician with specialized knowledge about the subsystem being diagnosed. This level also requires test equipment, often specialized. Self-tests, the next diagnostic level, are quick to perform and can locate most subsystem failures. They, however, require a trained technician to invoke because they are initiated from switches and testpoints internal to the computer. The diagnostic programs can be run by a technician with little training or by an unskilled operator under the telephone direction of a technician. Thus the customer may perform diagnosis by running diagnostic programs under the direction of a service engineer, so that the service engineer can be assured of having the

required replacement sub-assemblies when he arrives at the customer site.

Another feature to note is that all of these diagnostic procedures are performed after stopping normal system activity. Diagnostic programs run in a special environment, without possible disruption from or disruption of other activities. Self-tests assume no interference from outside the subsystem under test; in most cases no interference means that the system must be stopped to prevent interference. Because diagnosis using test equipment is by largely ad hoc methods, it may be possible to diagnose some subsystems in this way without stopping system activities, but in general other activities must be stopped and in practice they nearly always are stopped.

A third property of this strategy is that although the higher level methods are more cost effective, they cannot totally replace lower level diagnostic methods. The small kernel of subsystems required for testing may not be operable, forcing diagnosis of individual subsystems by using self-tests. There are failures that may not be found by self-tests, forcing diagnosis by a technician using test equipment.

On-line diagnosis fits into this hierarchy as a fourth level of diagnostic tools that can be used without stopping normal system activity. Since on-line diagnosis can be performed without stopping other activities, it is particularly useful to verify subsystem function when no failure has yet been detected. A wary user might wish to verify that the printer is working before beginning a long application that will eventually use the printer. Verification is an important use of on-line diagnosis.

On-line diagnosis is also useful when the system can continue to operate without the failing subsystem. Peripherals, in particular, may fail without disabling the

entire computer system. For example, most computer systems can continue to operate without a printer (for a while) by storing printer-bound output in files for later printing (spooling).

The discussion above shows situations where a user might invoke on-line verification or diagnosis. Several other strategies can be envisioned for invoking an on-line diagnostic, running with the aid of operating system services. Since the communications services of the operating system are available the diagnostic could be invoked from a remote computer (or at least receive commands from a remote location and send diagnostic results back to the remote location). Thus, an engineer at a field service office could run diagnostics on a customer's machine before going to the customer's site.

Another strategy for using on-line diagnosis is to schedule periodic diagnostic tests to execute automatically. These periodic verifications of system peripherals might detect problems before they become manifest to the user. The system might tell the user, "Device PRINTER removed from service because it failed a periodic verification test. Call your service engineer and inform him/her that subassembly XXX has been diagnosed as faulty." A the computer system might go further and send the message to the service center over a computer network or over dial-up lines.

Just as each of the other levels of diagnostic tools cannot completely replace lower levels of diagnostics, on-line diagnosis cannot replace other forms of troubleshooting. Some failures will cripple essential system services, requiring diagnosis using the smaller kernel of subsystems used by the stand-alone diagnostics or requiring that the subsystems be tested individually with self-test.

On-line diagnosis is useful only when the system can continue to operate normally or nearly normally with the failed subsystem. Diagnosing this class of failures without stopping other system activities provides for graceful degradation of services, permitting useful work to be done even while the failed subsystem is being diagnosed and repaired.

1.3: Related work

Most of the previous work in diagnosis of computer systems and reliable computing has been in the areas of component level diagnosis, particularly for production testing. Production testing is a very different environment from system diagnostic testing or on-line diagnosis. There is also much literature on super-reliable computers, as used in aerospace applications. The cost of super-reliable computing puts it out of the general purpose computer market; super-reliable computing techniques are, therefore, not usable in the general purpose computing environment. There is, however, some literature that addresses the concerns of on-line diagnosis.

The articles concerning the Electronic Switching Systems (Beuscher et al. and Downing et al.) provide insight into the problems of super-reliable computing in a real-time environment. These provide a good introduction to the power of on-line diagnosis.

Clary and Sacane summarize the methods used to provide "built-in tests", those that test occasionally like self-test and those that test continuously such as error checking coding. This paper in a good overview of the methods available to the hardware and firmware designers to provide "built-in-tests".

Gowan explains the features of the Hewlett-Packard HP 300 small business computer that enhance its serviceability. It outlines the presently used service strategy for the HP 300. It is particularly pertinent since my work in exploring on-line diagnosis has been with this system.

Taneda, Oku, and Nambe approach my subject most closely. They studied the the problems of on-line diagnosis and describe an implementation of an on-line test program for the Dendenkosha Information Processing System of Nippon Telegraph and Telephone Public Corporation. They identify three design objectives: detailed tests, no service interruptions, and ease of use. The former and latter in my system are left, for the most part, to the application program performing the diagnosis. My understanding of their work leads me to believe that what they mean by interruption is disruption. Preventing the disruption of normal service is performed in my system by a number of mechanisms discussed in later chapters.

Taneda, Oku, and Namba identify sources of difficulty in meeting their design objectives. In particular they identify three sources of interruption (disruption) of normal service by the testing programs. One source is incorrect access to a device other than the device under test. Another source is garbage left on the test device media after testing is stopped either normally or abnormally. The third source of interruption (disruption) is conflicts over system resources between the test program and other system activities. These sources of disruption will be controlled by the system proposed here.

1.4: Method of Presentation

This thesis is broken into 5 chapters. Of the remaining four, the next three divide the coverage of on-line diagnosis into environment, design, and implementation.

Chapter 2 identifies the hardware and software structures involved in input and output and provides a perspective into the environment into which on-line diagnosis will be fitted.

Chapter 3 develops the diagnostic manager as the manager of a new class of virtual resource. First the diagnostic manager is placed in the software hierarchy of chapter 2. Then services it must provide are motivated and developed. Also covered in chapter 3 are three functions performed by each I/O device for diagnosis. Next, methods of avoiding interference with other system activities are developed.

Chapter 4 details the trial implementation of this strategy on the HP 300 computer system. Covered are both hardware and software implementations. The purpose of the trial implementation was to develop an understanding of the real implementation problems encountered. It was also useful in showing that this strategy could be post-fitted to an existing system.

Chapter 5 is a summary of the work presented here. Implications of the results are considered. Suggested, too, are directions for further work in on-line diagnosis.

Chapter 2: System Environment Structure

We begin by choosing a system structure that is representative of typical computer systems available today. It is this structure that will serve as the basis for adding on-line diagnosis.

The structure outlined here is similar to most computer systems and, therefore, provides a wide base of applicability to the development of techniques for on-line diagnosis. It is similar to the hardware and software structure of the Hewlett-Packard HP 300 small business computer on which this work was done. This is not, however, meant to be a description of the HP 300.

2.1: Hardware organization

The hardware structure for the input/output part of a computer system can be viewed as a hierarchy. There is a single computational cluster of the system, consisting of the processor and the memory. One or more I/O channels each interface one or more I/O devices to the computational cluster. This structure is diagrammed in figure 2.1. What follows is a discussion of the function of each level in the hierarchy.

2.1.1: Computational Cluster

The computational cluster of the system controls input and output operations. It initiates operations by issuing requests to the I/O channels. The cluster is the source of data for output operations and is the destination of data for input operations. During on-line diagnosis, the computational cluster serves as the

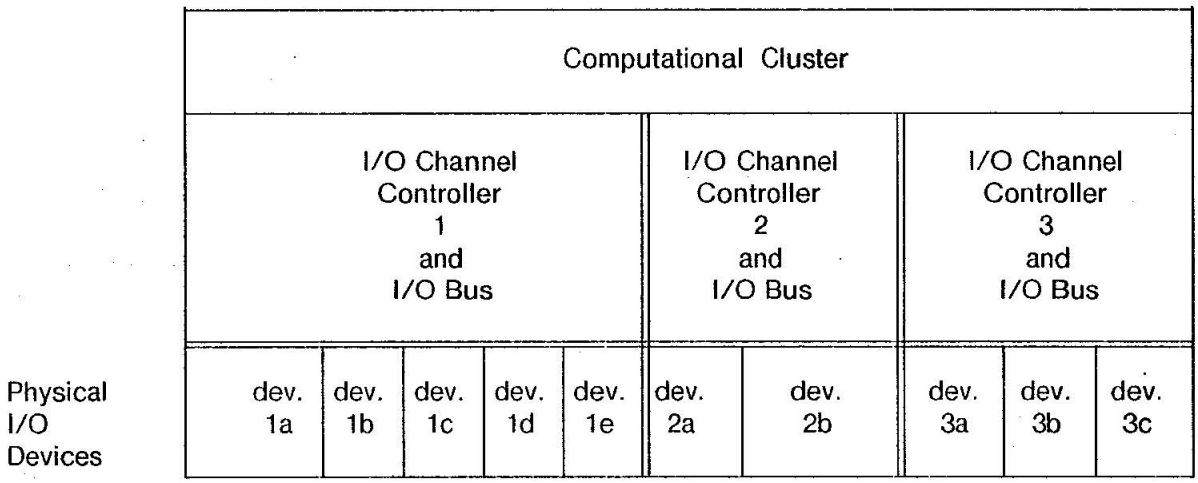

Figure 2.1 I/O Hardware Hierarchy

diagnostician, accessing a peripheral in order to determine if the device is functional. Programs executed by the computational cluster perform the diagnosis. Accesses to the device under test (DUT) are initiated by issuing a request to the channel controller associated with the device.

Errors encountered by either the channel or the DUT are reported to the computational cluster. Reporting these errors allows the diagnostic programs to detect them and may allow the diagnostic test program to test the error detection mechanisms, as well, by introducing known errors. Some of the errors detectable by these mechanisms cannot be forced by the diagnostic program because they are internal on the device or the channel controller, or because they reflect physical errors and malfunctions of the device. Some errors detectable by these mechanisms should not be forced in the on-line environment as they might affect other I/O devices and other operations proceeding concurrently in the system.

In order to perform on-line diagnosis we assume that the computational cluster is functioning correctly. This is reasonable since without a functional computational

cluster, the operating system could not run and diagnosis cannot proceed on-line.

2.1.2: I/O Channel Controller and I/O Bus

The principal functions of the I/O channel controller are managing the usage of the shared I/O bus, relaying requests from the computational cluster to the I/O devices, and controlling data transfers to and from the devices. As such, its function is communication between the computational cluster and the devices.

The shared I/O bus connects several devices to the channel controller. Since the channel controller is the interface to the computational cluster and the computational cluster controls all I/O, the channel controller must be the source or the destination of each I/O bus transfer. Hence, direct device to device transfers are prohibited. We assume for concreteness that the channel controller performs I/O bus allocation and arbitration.

The relaying of requests from the computational cluster to the devices and the managing of the data transfers often involve considerable mechanism in implementations. In many systems, requests are presented as channel programs to be interpreted by the channel controller, producing a series of commands to the I/O device. We will consider channel programs part of the physical device driver software. This presents the difficulty for on-line diagnosis that channel programs can be written that perform error handling without notifying the diagnostic program. This will be discussed in Chapter 3. Data transfer management usually involves keeping counts of data transferred and performing address calculations for directly accessing the memory in the computational cluster. We need do no more than note its possible presence in the system, as these functions do not interfere with on-line diagnosis.

The I/O channel controller and I/O bus are introduced into this system model because they present a problem in on-line diagnosis. They present a point in the access path from the computational cluster to the DUT where access to several devices share the same path. A failure in this path will affect all the devices serviced by it. Furthermore, a failure of one of the devices may affect all the devices by causing the access path to fail. For example, if a bus driver for one of the devices becomes permanently active it could jam the bus, causing errors on bus transfers between the controller and any device.

Last, it should be noted that some computer systems connect I/O device directly to the computational cluster without the shared I/O channel controller and I/O bus. This arrangement does not invalidate this model but is a special case of it. In this case each I/O device has its own channel controller. This simplifies the problem above greatly by removing the channel imposed problem discussed above; however, in most systems there is some other point of shared access path so that this same problem appears elsewhere.

2.1.3: I/O Devices

The I/O devices connect to the system through the I/O bus. Requests from the computational cluster reach the device as a series of commands from the channel controller. The commands fall into four classes: data reads, data writes, status reads, control writes. I/O devices cover a wide variety of apparatus. Two important classes of I/O devices are storage devices and hardcopy devices. For concreteness, consider a storage device to be a disc pack drive and a hardcopy device to be a lineprinter.

Asynchronous signaling of request completion and of status changes is usually implemented but need not be detailed here. For our purposes it can be included in our model of the channel controller's bus allocation function.

2.2: Software organization

The software involved in I/O operations is also organized in a hierarchy. Each level presents an interface with more uniform characteristics than that below. Physical devices are transformed to logical devices and logical devices are used to make virtual resources such as files and virtual communication circuits. These virtual resources are used by applications to solve the user's problems. Figure 2.2 diagrams this hierarchy. All software executes in the computational cluster.

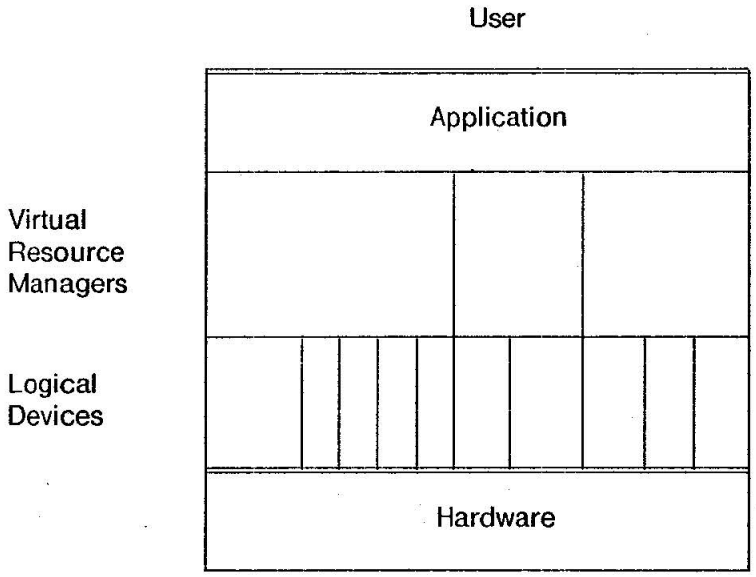

Figure 2.2

2.2.1: Physical Device Driver

The physical device driver transforms the physical device interface into a logical device interface. It is the software module that issues requests to the I/O channel controller. As a result it must be an expert on the device. Since devices vary widely, there is typically one physical device driver for each model of device connected to the system.

The logical devices presented by the physical device driver are uniform in their recognition of generic requests and uniform within a device class for other requests. For example, the only differences between two different disc drives at the logical device interface are the amount of storage on the device and its speed of operation.

Physical device drivers also perform device specific error handling, often involving retrying the operation. This is a problem for on-line diagnosis as the test program should know about each error that occurs during testing; however, other accesses to the device should not have to know about recoverable errors. The problem of error reporting will be approached in Chapter 3.

2.2.2: Virtual Resources

The logical devices of the physical drivers provide a somewhat uniform interface, but require considerable management to be useful. An additional layer of software provides this management, relieving the application from the burden of management and protecting the system from mismanagement. The virtual resources provided by these managers are more powerful than the logical and physical resources that implement them. For example, the files and directories

provided by a file manager present a simple interface to the application but may implement more storage than is available on a single disc drive.

The other aspect of virtual resource management is coordination. In a multiprocessing environment, care must be taken to control multiple accesses. Accesses for diagnosis must not cause errors in normal accesses. Since diagnosis may place the device into states that are illegal in normal operation, in most cases, other accesses to the device must be blocked during diagnosis.

In order to provide virtual resources the manager must keep data about the state of the virtual resource and about the state and characteristics of the logical device. The virtual resource manager must protect this data against inconsistency by verifying that all requests are consistent with the current state. The state data is not accessible to the application. Not only must an on-line diagnostic be careful not to leave the device in an inconsistent state, but it must also be careful not to leave any virtual resource management data in an inconsistent state.

2.2.3: Applications

Applications are all users of virtual resources. These programs do not deal directly with devices or synchronization. They have at their disposal all virtual resources in the system. All user programs and most system utilities are written as application level programs. The wide variety of resources and system services, such as multitasking and memory management, make this a powerful environment for program execution. Thus it is desirable that an on-line diagnostic be written to run as an application program.

2.3: Summary

An environment has been developed in this chapter for the development of an on-line diagnostic strategy. The hardware hierarchy was built outward from computational cluster through the I/O channel controller to the peripheral device. The software hierarchy of physical device driver, virtual resource manager, and application was also developed. Along the way problems for on-line diagnosis were identified.

In the next chapter we will explore the placement of a diagnostic interface in this hierarchy and what functions it should perform. Solutions will be sought for the problems mentioned in this chapter.

Chapter 3: Diagnostic Manager

It is evident from the discussion in chapter 2 that the virtual resources provided in computer systems have very different characteristics than the devices used to implement them. A new class of virtual resource is proposed here whose characteristics are more similar to those of the device under test.

3.1: Characteristics of the Device-Under-Test Virtual Resource

The characteristics of a device-under-test virtual resource (DUT) should be chosen to aid on-line diagnosis. To that end the DUT should mimic the state of the physical device implementing it, as it is this device that is being tested. In addition access to the DUT should not cause misoperation of any other activity in the system.

To aid on-line diagnosis a DUT virtual resource must report an error whenever the physical device that implements it encounters an error. Failed operations should not be automatically retried. Furthermore, the errors reported should map one-to-one onto the physical device errors producing them. In other words, each device error should have a distinct error representation at the DUT interface. We will call this characteristic truthful error reporting.

The power of access to the low level characteristics of the device must be controlled so that other activities can proceed unharmed. Certainly, the truthful error reporting characteristic can easily be provided by allowing access directly to the physical device, bypassing the physical device driver. However, this uncontrolled access can interfere with other activities. Device access must be coordinated among the various activities using the device. Furthermore,

accesses to the device must not make illegal access to the I/O channel controller or leave the device in an illegal state. While the later could be solved by careful device design, in most cases it is not.

A better solution is to provide at the virtual resource level a diagnostic manager as a manager of DUT's which has the responsibility for synchronizing access to the physical device with other activities and for assuring that the channel is never improperly accessed and that the device is left in a state so that "normal" accesses can continue after testing.

In order for the diagnostic manager to report device errors truthfully, all errors must be reported truthfully by the physical device driver. We said, however, in chapter 2 that the physical device driver handles all device specific errors, often by retrying. Retrys violate the characteristic of truthful error reporting by hiding retry recoverable errors from the diagnostic manager.

The only apparent solution to the retry problem is to add new requests to the logical device interface (between the physical device driver and the diagnostic manager). These requests should provide accesses to the device without invoking the normal retry mechanism should an error be encountered, but report the error instead.

This, then, is the basic architecture of on-line diagnosis: the DUT virtual resource is managed by the diagnostic manager accessing logical devices through special requests that report errors truthfully. The software hierarchy is diagrammed in figure 3.1 below. The remainder of this chapter adds further detail to this strategy and discusses how to solve other problems mentioned in chapter 2.

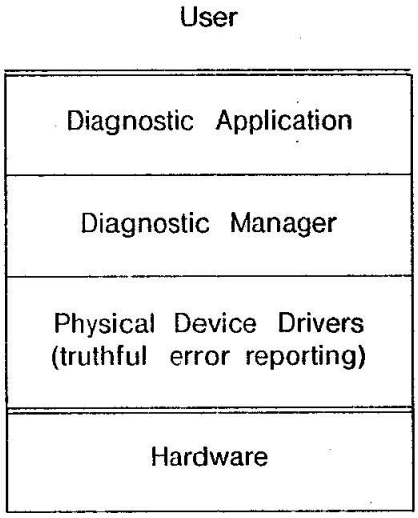

Figure 3.1

3.2: Diagnostic Manager Interfaces

In this section we will explore several diagnostic interfaces to the application program. First we will explore some administrative interfaces and then some useful test functions. These interfaces are service procedures that may be called by any program that has the right to use them. Since sufficient protection for other activities is provided by the diagnostic manager this is no reason that all programs cannot have the right to use these interfaces. On the other hand, the diagnostic interfaces may breach the security system. In that case the right to access the diagnostic interfaces should be controlled in the interest of maintaining security.

3.2.1: Begin Testing

The interface to create a DUT virtual resource is called *diagnostic open*. The name is indicative of its analogy to file management *open*. Like file management *open*, the user calls this procedure to create an instance of the virtual resource in the environment of the application. Like the file manager, the diagnostic manager handles most synchronization when the virtual resource is created. Local data used to store the state of the virtual resource is allocated and initialized at this time.

The *diagnostic open* interface accepts a device name to indicate which physical device is to be tested. This name is translated by the diagnostic manager into a logical device address. For consistency all virtual resources should use the same device names.

Coordination by the diagnostic manager at a diagnostic open consists basically of checking whether any other use is being made of the device and if it is not already in use, allocating it for the exclusive use as a DUT for a particular user. If the physical device is already in use the application is denied access. This strategy of exclusive allocation is simplistic and not always desirable. Therefore, we will return to it later.

3.2.2: End testing

At the conclusion of testing the application calls *diagnostic close* to handle deallocation of the virtual resource and its implementation data in the DUT manager. *Diagnostic close* also releases synchronization control by the diagnostic from the device. In the case of exclusive access, the lock is removed so that other

activities can access the device.

There is one additional function that may be performed by *diagnostic close*. If one of the diagnostic tests fails, *diagnostic close may, at the request of the diagnostic application,* mark the device as off-line so that other programs cannot access the failed device. This function should be performed by diagnostic close if there is the chance that device may be allocated to some other activity immediately after being closed. This is not a problem in systems where the common allocation and synchronization mechanism allows a device to be in use but not available for further allocation. In that case an additional interface should be available to remove the device from service.

3.2.3: Device Diagnostic Functions

Device diagnostic functions provide the interface to perform the actual diagnosis. Two different approaches can be taken to choosing these interfaces. These routines could have access to all of the functions at the logical device level, thus the diagnostic manager would only perform synchronization with other activities and pass all other calls unchanged to the proper logical device interface. This strategy gives the diagnostic application programmer a low level interface to the device and assures that each diagnostic application will work with only a small number of different devices.

The other approach is to provide higher level diagnostic functions such as write-read-compare. Diagnostic applications can then be written with little or no device specific knowledge. To that end, five high level test functions are proposed in this section.

Three of the diagnostic functions are based on physical device commands that are recognized by devices in the HP 300 computer system. They were designed by foresighted engineers to aid in system diagnosis. Their use is presented here because they aid the diagnostic writer greatly. With the limited set of diagnostic functions presented here they are necessary for any meaningful diagnosis. With another set of diagnostic functions they might not be needed. The other two tests are diagnostic write and diagnostic read. These provide for end-to-end functional testing.

3.2.3.1: Identify

The *identify diagnostic* function uses the identify command to request a particular device to respond with its identify code. This code indicates the type of device (ie printing) and the particular model.

While this command was conceived to aid in automatic configuration, it is also useful for diagnosis. Identify provides an initial check that the DUT is present at the channel address expected and is alive. Alive here means at least partially functional. Additionally, a garbled identify code usually indicates that two devices are sharing the same I/O address.

The *identify function* returns "failed" if the device does not respond with the identify code that is expected for that device. *Identify* should be used at the beginning of every diagnostic so to ensure that diagnosis is being performed on the right device. It can also save much time if the device is powered-off or disconnected from the I/O bus.

3.2.3.2: Loopback over I/O bus

Loopback over the I/O bus consists of sending data to the peripheral device interface and then reading it back. This allows the access path from the computational cluster to the device to be checked. Loopback only tests the device interface to the I/O bus not the device function. The goal of loopback testing is to isolate testing the I/O bus interface, I/O bus, and channel controller from testing the device functions. Patterns can be chosen to test for "stuck at" and shorted faults in the I/O bus.

The *loopback function* uses the loopback device command to effect the transfers. The data returned is compared to the data sent. If any data differ *loopback* returns "failed". *Loopback*, too, should be performed before any functional tests.

3.2.3.3: Self-test

Self-test is a comprehensive, low level, functional test that a subsystem performs internally. Self-test is a powerful change in the way diagnosis is performed. Self-test places the burden of fault isolation on the device, providing an object-oriented approach to diagnosis (command the device to diagnose itself). An advantage of self-test is that in testing itself, the device has much greater accessibility to internal nodes than any of the software in the computational cluster can, thus diagnostics are less involuted easier to write and to understand.

By using the same encoding for "passed" the interface to each device can be the same for all others except for the interpretation of the "failure" code that indicates which subassembly to replace. Verification programs that test whether a device is functional can be written so that they will test any physical device.

3.2.3.4: Diagnostic Write

Diagnostic write performs otherwise normal accesses to the device, but reports all errors truthfully. Naming the destination of writes could be handled in several ways. The names could be addresses on the logical device. This violates the goals of high level interface and device independence and may compromise system security. The destination could always be a reserved "test area". Reserving the extra space is wasteful of storage space on the device for the infrequently used diagnostic. Also, the fact that the space is "special" indicates that accesses to it are not the same as normal accesses and may not use the same mechanism.

The destination could be a file on the tested device already *opened* by the diagnostic application using the file management *open* interface. The diagnostic manager must be carefully designed to avoid deadlock in the access control mechanisms governing files and DUT's when the file is implemented in storage on the DUT. Another concern when using application provided files for testing is the handling of the file on abnormal program termination. Most file managers provide files that are in a legal state after abnormal termination. If, in addition, the file manager provides a means to declare a file as temporary so that its space will automatically be reclaimed after use, the problem of collecting diagnostic garbage is solved as well. If the file manager does not provide temporary files of this type the diagnostic application will need to perform diagnostic garbage collection as best it can. While application provided files are a flexible choice for naming the destination of *diagnostic writes*, this choice does not compromise system security since all security features of the file manager intermediate in the file

open and in the diagnostic accesses as well.

3.2.3.5: Diagnostic Read

Diagnostic read reads data from the device implementing the DUT, reporting any errors truthfully. Like *diagnostic write*, *diagnostic read* should be implemented using as much of the normal mechanism for reading as is possible. Normal mechanism in most cases means that access should be through the file manager's *read* mechanism.

As in *diagnostic write*, there are several alternatives for determining the address for reading from the device. The most important consideration is that *diagnostic read* and *diagnostic write* be consistent in their interface to the application. As discussed above, application provided files seem the most flexible and easiest to use.

3.3: Synchronization of Accesses to the Physical Device

Care must be taken so that accesses to the device for diagnosis do not interfere with normal accesses. The simplest solution to this problem is to give the diagnostic exclusive access to the device while diagnosis is being performed. Exclusive access, however, is not always best. In some cases diagnostic accesses and normal accesses can be interleaved, just as accesses to data may in some cases be interleaved. The advantage of interleaving accesses is that diagnosis can minimize its impact on system performance in verification usage by testing a device that is in service.

There are three ways that tests can be grouped, according to whether and how they can be interleaved safely with other accesses. Some tests, such as *identify*, do not alter the state of the device in any way, are performed by a single request, and do not demand internal consistency in the device state or the data stored by it. A test of this class can be performed at any point in any stream of requests to the logical device, without interfering. We will term this class of tests, concurrent.

Some tests do alter the state of the device, but transform the device state from one legal state to another. Normal *read* and *write* transactions have similar properties. One way they alter the device state is by advancing the read/write point by performing a seek operation or by advancing the media past the read/write point. Additionally, these tests may make several requests to the logical device that must be performed without intervening requests. This is similar to *update* access to data, the state of the device must not change between requests. We will call this class of tests, shared access. Shared access requires blocking other accesses to the device at the beginning of a test and releasing it at the end of the test. *Self-test* is a shared access test.

The third class of tests require exclusive access to the device. Exclusive access is required whenever two or more tests must execute in sequence without any intervening state changes in the device or whenever a test leaves the device in an illegal state. Depending on the details of how the operating system implements multiprocessing the diagnostic may be able to measure timing of operations on a device being tested with exclusive access. The diagnostic manager must return the device to a legal state (usually reset) before returning the device to normal service.

Specification of the class of test as either concurrent, shared, or exclusive may occur when the DUT is *opened*, just as the access mode of a file is specified at *open*. The diagnostic manager must be certain that all access control for a DUT is released when the device is *closed* or when the diagnostic terminates abnormally.

3.4: Diagnostic Data Bases

In order to implement the diagnostic interface for a wide variety of devices, the diagnostic manager needs a data base. The diagnostic data base contains useful and necessary information about each physical device. The data base includes such information as which diagnostics test functions may be performed on the device and the size of the loopback buffer provided by the device.

The diagnostic manager also needs access to other data. It must have access to the mapping from device names as known by the user to logical device addresses. It must also access synchronization data for control of accesses to the device. Synchronization data must be shared by all virtual device managers to prevent interference at the device between accesses by different virtual resource managers. Of course, the diagnostic manager must observe the same protocol as the other resource managers in accessing the synchronization data to protect the integrity of the synchronization data.

3.5: Diagnostic Manager Summary

In this chapter we have developed a diagnostic manager that manages physical devices during diagnosis and verification. Its chief purposes are to protect other activities from the diagnostic activity and to provide higher level interfaces for diagnosis, removing much low level detail. Basic diagnostic interface functions were introduced. *Diagnostic open* and *diagnostic close* control the allocation and deallocation of diagnostic virtual resources and control some synchronization. The test functions *identify, loopback over I/O bus, self-test, diagnostic write,* and *diagnostic read* provide basic diagnostic test capability. The issues of synchronization and of diagnostic data bases were also discussed.

In the next chapter specifics are given of a trial implementation on an HP 300 computer system.

Chapter 4: Implementation on the HP 300

In this chapter a trial implementation on the HP 300 small business computer is detailed. The purpose of implementing the diagnostic manager is to show the viability of this approach and also to show that this approach can be built into an existing operating system of the common form outlined in chapter 2.

While the implementation is not complete, it is viable for demonstrating the approach. Only *diagnostic open, diagnostic close, identify,* and *self-test* were implemented. *Loopback, diagnostic write,* and *diagnostic read* were not implemented. In addition an error message formatter provides English language translations of error codes provided by the diagnostic test functions.

4.1: Diagnostic Manager Implementation

The diagnostic manager consists of a set of applications interface procedures and a diagnostic data base. The interface procedures are the diagnostic interfaces developed in chapter 3. These interfaces manipulate the diagnostic data and issue requests to the logical devices.

The diagnostic manager implementation was complicated by implementation time limitations that prevented the modification of all physical device drivers to recognize diagnostic requests and report errors truthfully. As a solution a very simple physical driver was written that recognizes only the diagnostic requests for identify, loopback, and self-test and reports their errors truthfully. This physical driver works properly with most devices in the system for these commands. Installing this diagnostic device driver as a temporary physical driver for a device is a delicate process which will be described, briefly, in this section.

4.1.1: Diagnostic Data Bases

The diagnostic manager stores data for four purposes. Static data is stored concerning the diagnostic capabilities of the physical devices in the system and the physical drivers for those devices. A static symbol table associates device names as known by the user with logical devices. Synchronization data is shared with all other virtual device managers. And, last, the state of each allocated DUT is kept by the diagnostic manager.

Diagnostic capabilities of the devices vary. Not all devices implement self-test, for example. To avoid making invalid requests to a device, the diagnostic manager should know which requests the device can perform. The diagnostic manager also requires device specific constants that are parameters to the requests. The size of the device input buffer used for loopback is such a device specific constant and is the only device specific constant used in the trial implementation.

For each device, an indication is kept of whether its normal physical device driver can recognize diagnostic requests and report errors truthfully. This indication makes possible for a gradual transition from having no device drivers recognize the diagnostic requests to having all devices recognize them. If the normal device driver does not recognize the diagnostic commands the diagnostic device driver replaces it temporarily.

The *device name table* is a system wide data base that associates device names as known by the user with logical device addresses. The contents of the *device name table* is fixed when the hardware configuration is defined. Additionally, each

process has an *active equate table* which allows local renaming of devices and files. To determine the logical device address the *active equate table* is searched for the device name supplied by the application. If the name is found in the *active equate table*, it is translated to the corresponding global name, if not, the name is already a global name. The global name is looked up in the *device name table* and logical device address is retrieved.

Synchronization data is maintained to control multiple accesses to each of the devices. The synchronization data is shared by all virtual resource managers so that accesses by different virtual resource managers to the same device are properly controled. Careful adherence to the established protocols for accessing the synchronization data are necessary so that synchronization errors do not occur.

Each instance of a DUT has some state information associated with it, called the diagnostic control block. This information includes the logical device address of the device being tested, pointers to buffer space used for I/O transfers, and a list of system resources that can be allocated for implementing a particular DUT. System resources include buffer space, *id's*, files, etcetera. These system resources must be deallocated when the DUT is closed and on abnormal termination of testing. State information also includes the state of the diagnostic physical driver if used by this DUT.

The *id's* provided by the HP 300 operating system are its means of providing an object interface to virtual resources. *Open* creates an *id* which is only valid in that user's process. All further references to the virtual resource use the *id number* corresponding to that *id*. The *id number* is used by the virtual resource manager to access the state (control block) of the virtual resource. Additionally,

id's are typed. Virtual resource managers check the type of the *id's* passed to them to ensure that each manager only operates on its own type of object.

Additional mechanism in the HP 300 operating system provides for termination deallocation of virtual resources. For each process a list is kept of all currently held *id's*. On either normal or abnormal process termination the list is scanned and the proper virtual resource manager's termination handler is invoked for each virtual resource still allocated. The proper virtual resource manager can be determined by examining the type of the *id*.

In order to implement a new type of object for DUT, minor changes were made to the *id* manager. More major changes were required to the termination handler to properly handle DUT *id's*. The changes to the termination handler were not implemented.

4.1.2: Application Interfaces

Five diagnostic interface procedures were implemented, *diagnostic open, diagnostic close, identify, self-test,* and *diagnostic error message formatter.* The first four are implementations of diagnostic interfaces developed in chapter 3. *Loopback, diagnostic write,* and *diagnostic read* were not implemented.

Diagnostic open creates an instantiation of a DUT. It takes as one argument the name of the device to be tested. The device name is translated to a logical device address by look up in the system wide device name table. The diagnostic control block is allocated and initialized with device specific information from the diagnostic data base and parameters supplied by the application as arguments to *diagnostic open*. One argument is the class of test. If the class of test is

exclusive an attempt is made to allocate the device to the application for exclusive access. If the class of test is shared an attempt is made to allocate the device for shared access. Of the class of test is concurrent the device is not allocated to the application. An id of type DUT is allocated and is initialized with a pointer to the diagnostic control block. If any error is encountered during *open*, all allocations are backed out and the error is reported to the application.

Diagnostic close deallocates the resources allocated by *diagnostic open* and other temporary resources used by the diagnostic tests that for some reason were not previously deallocated. Additionally, if the device failed any of the diagnostic tests, it is marked unavailable for use and a warning is sent to the application. *Diagnostic close* will not abort if it encounters an error in attempting to deallocate a resource but will continue attempting to deallocate the other resources. If an error occurs the application will always receive an error indication; however, if multiple errors are encountered, the application will only receive a single error indication corresponding to one of the errors actually encountered.

Identify and *self-test* are diagnostic tests that rely on the hardware diagnostic capabilities of the peripherals on the HP 300. They issue requests to the logical device. If the device fails the diagnostic test, the failure is marked in the diagnostic control block and an error code and failure syndrome are returned to the application.

Because the system physical device drivers do not recognize diagnostic requests, a special diagnostic physical driver must be installed temporarily as the physical driver for the device being tested. The installation and removal of the diagnostic physical driver is performed at the beginning and end respectively of the diagnostic tests. Installation involves interfacing to memory management to

make the diagnostic driver present in physical memory at a fixed physical address, then manipulating hardware tables under protection against interruption.

The *diagnostic error message formatter* accesses the diagnostic error message text data base to translate error codes returned by the test routines into English text. For some errors additional information about the error is returned by the test routine. The additional information, called the error syndrome, is vital for ascertaining the exact fault in the device. For example, *self-test* returns the error code for "failed self-test" and, in addition, the failing sub-test number returned by the device. The *diagnostic error formatter* could translate the error information into text such as "Self-test failed testing SEEK function on device FLEXDISC. Replace subassembly C."

4.2: Physical Device Driver

The physical device driver must present a logical interface with normal functions for I/O operations plus additional functions for diagnostic requests. These additional functions provide truthfully reported versions of normal requests and additional requests that use the diagnostic commands of the devices. To add these functions would require substantial changes to each physical device driver. However, since for these functions, no error processing is required and since the interface to device diagnostic commands is uniform, a single *diagnostic device driver* can serve all of the devices for these functions.

4.3: Limitations of the Diagnostic Subsystem Implementation

The implementation of the diagnostic subsystem suffers from several limitations. The limitations are not inherent in the strategy of chapter 3, but present in this implementation. All are the result of incomplete implementation or incomplete implementation design.

The most noticeable limitation is the inflexibility of the application interface. The only diagnostic test interfaces provided are *identify* and *self-test*. The application does not have many options in testing a device. Typically, *identify* is performed followed by *self-test*. If the device passes both tests, no further and more rigorous tests are available. With all of the test interfaces described in chapter 3 considerably more choices are available for the application to pursue additional testing. It is my feeling, however, that additional diagnostic test interfaces, beyond those described in chapter 3, need to be defined to increase the flexibility of the application interface.

Another serious problem with the implementation is that the test routines were not designed to work with all types of I/O channel controllers. The implementation design was for a general I/O channel controller. The HP 300 may, however, have other types of channels. In particular, the *asynchronous data communication controller* I/O channel, which interfaces terminals to the computer via asynchronous communication facilities, is not supported by the diagnostic manager. The difficulty lies in different logical device commands and different channel structure. The capability to access different types of channels for diagnosis would have expanded the base of applicability of the implementation; however, it also would have complicated the diagnostic manager. More seriously, though, diagnostic test interfaces appropriate to all channels, such as *diagnostic read* and *diag-*

nostic write must be developed. For example, the semantics of loopback in a communication channel are not immediately apparent.

A third limitation of the diagnostic manager implementation involves abnormal termination of the diagnostic. The problem is that program termination is delayed until all outstanding I/O to the logical devices has completed. This is a design feature of the HP 300 operating system. On some devices, however, the self-test request may take as long as a minute to complete or to fail due to a timeout. The termination strategy in the HP 300 was not designed for very long I/O transactions. As a result, termination of the diagnostic program may be delayed for up to a minute while diagnostic I/O completes. A clever design will be needed to correct this problem in the HP 300 termination strategy. The problem is not fundamental, but is a problem in this operating system.

Another limitation of this implementation is that the disc drive containing the *system volume* may not be tested. The reason for the difficulty may be seen by comparing the system structure of the HP 300 to the structural model of chapter 2. The *system volume* is used both as ordinary file storage and as backing store for the virtual memory subsystem. The only way to fit the virtual memory function of the disc into the model of chapter 2 is to place that function of the disc into the computational cluster. Since the diagnostic application and the diagnostic manager execute in the computational cluster, the diagnostic manager may not test the *system volume* without disrupting the operation of the computational cluster on which the diagnostic manager must run. Unless the system provides for dynamic reconfiguration of the virtual memory backing store so that a different disc may serve, at least temporarily, as backing store, the *system volume* cannot be tested by on-line diagnostics. The problem with testing virtual memory backing store is fundamental to this diagnostic strategy, stemming from the

incompatibility with the underlying system model.

4.4: Sample Diagnostic Application

A sample diagnostic application program was written to exercise the diagnostic manager and demonstrate it. The application allows the user to select a device for test by giving its name, and then, execute the diagnostic test of either of the diagnostic test interfaces by pressing a "soft-key" beside the name of the test on the screen. The application takes advantage of the multi-window display of the HP 300 console to present a history of the tests performed and errors encountered, as well as the menu of tests, an environment window stating that the display is for the "On-line Diagnostician", an input window showing user input, and an error window showing input errors.

The interface to the user puts diagnosis and verification in terms that are familiar to the user. Device names are the same names used in all dialog with the computer system. Even the display is familiar to the user as the diagnostician's usage of the display is similar to that of other subsystems. In all, the user need not feel intimidated by diagnostics or the on-line diagnostician.

Much of the elegance of the user interface is attributable to having operating system services to extend the diagnostic application environment. Services to enhance user interface are seldom if ever found in stand alone diagnostic monitors, but are usually found in general purpose operating systems.

4.5: Implementation Summary

The implementation of the diagnostic manager on the HP 300 was an educational experience. Experimentation with the implementation pointed the way for much of this thesis. But, also, the implementation substantiates the thesis by showing that it is a workable approach. It also shows that the diagnostic strategy of chapter 4 can be built after-the-fact into an existing computer system.

Chapter 5: Conclusions

In this thesis features were identified and a design methodology developed to implement an interface so that on-line diagnostic programs can be written. An implementation was discussed that was undertaken to gain greater understanding of the problem and to show the viability of the design methodology developed. In this chapter the results of this thesis will be summarized and their implications noted. Additional directions for research will be suggested.

5.1: Summary of the Results

In chapter 2 a model was developed of the hardware and software environment of concern for diagnosis in a general purpose operating system. A design methodology was developed for on-line diagnosis in systems of the form modeled in chapter 3. The basis of the design methodology is a virtual resource manager for device-under-test virtual resources (DUT's). A diagnostic application program accesses a DUT to diagnose a physical device. The diagnostic manager makes diagnostic requests to the logical device which is implemented by the physical device driver. The physical device driver accesses hardware.

The virtual resources supplied by the diagnostic manager have characteristics that aid on-line diagnosis. In chapter 3, characteristics of a DUT to aid on-line diagnosis and protect other activities from disruption were developed. These characteristics are truthful error reporting and controlled access synchronized with accesses by other activities.

Also, in chapter 3, seven interfaces that the diagnostic manager might provide to the application are described. These interfaces fall into three categories.

Diagnostic open and *diagnostic close* are administrative interfaces, for allocating and deallocating DUT's. *Diagnostic write* and *diagnostic read* are truthfully reported versions of interfaces available from other virtual resource managers. *Identify, loopback,* and *self-test* are special purpose diagnostic functions utilizing special device capabilities for diagnosis.

In chapter 4, a trial implementation is described. The implementation both served as an arena for developing the ideas presented here and as a verification of their viability.

5.2: Implications

The availability of the capability for on-line diagnosis greatly reduces the cost of using diagnosis and verification. Use of the diagnostic capabilities to verify that peripherals will particularly benefit from the low cost of on-line diagnosis. The low cost takes several forms. Since other activities are not stopped to run a diagnostic, availability is enhanced since the computer system is still available during diagnosis. Additionally, a trained technician is not needed to diagnosis the system; the user may diagnosis a peripheral without aid. Also, the ease of use and the immediacy of being able to perform diagnosis while other work proceeds will enhance user satisfaction with the computer.

5.3: Additional Research

Several areas of additional research are suggested. As suggested in chapter 4, more work is needed in defining the diagnostic manager interfaces to the application. The set presented here is restrictively small.

The problem of diagnosing the backing store for virtual memory is typical of the problems of diagnosing any resource that cannot be accessed exclusively, except for a very short time. This problem, too, deserves attention as its solution will allow the I/O channel controllers and the components of the computational cluster to be tested on-line in addition to peripherals.

While not discussed previously here, it became apparent during this research that there exists a close logical link between the system error logging facility and the diagnostic monitor. Some means of integrating the two seems a profitable undertaking. One possible link is in automatic invocation of diagnosis when there is an excessive error rate on a particular device. Another link might be to use trend analysis to aid diagnosis. It seems clear that the diagnostic manager and the error log exist to serve similar purposes. Both serve to aid maintenance and service.

5.4: Concluding Remarks

On-line diagnosis is a tool that can change the user's perception of computer system reliability. The availability of tools to verify and diagnose parts of the system easily by users and service technicians alike will give the user added confidence in the computer system by encouraging verification of system parts using the diagnostic tools and also increase system availability by allowing diag-

nosis to be performed without stopping other productive activities.

Bibliography

Beuscher H. J., Fessler G. E., Huffman D. W., Kennedy P. J., Nussbaum E.; "Administration and Maintenance Plan" (No. 2 ESS); Bell System Technical Journal, Vol. 48, 8 (Oct. 1969); pp 2765-2815.

Clary J. B., Sacane R. A.; "Self-testing Computers"; Computer, Vol. 12, 10 (Oct. 1979), pp 49-59.

Downing, R. W., Nowak J. S., Tuohenoksa L. S.; "No. 1 ESS Maintenance Plan"; Bell System Technical Journal, Vol. 43, 5, part 1 (Sept. 1964); pp 1961-2019.

Fitzsimons R. M.; "TRIDENT -- A new maintenance weapon"; Proc. AFIPS Fall Joint Computer Conf. (1972), Vol. 41 Part I, pp 255-262.

Gowan C. R.; "Serviceability features of the HP 300 small business computer"; AFIPS Conference Proceedings, Vol. 48 (1979), pp 493-497.

Taneda A., Oku H., Nambe D.; "An on-line test .program for peripheral devices"; Proceedings of the AFIPS National Computer Conference (1976), pp 1001-1006.