DEVELOPING TRUE DIAGNOSTIC SOFTWARE

Michael Prusynski

Oregon State University

MSCS Research Paper & Project (CS501)

Submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Science

Presented at Oral Exam:
December 9, 1988
Final Revision:
December 23, 1988

ABSTRACT

This document describes the need for, and design of, diagnostic software that provides circuit verification, fault isolation, and troubleshooting aids. The difference between verification software and true diagnostic software is explained. A modular design of diagnostic software is proposed, consisting of a series of 3-part tests, a test manager, results analyzer, and user interface. The design guidelines are described in general terms so that they may be applied to other computer-based boards or systems. An implementation of these design principles is described for a circuit board in a Tektronix product -- the DAS9200 (Digital Analysis System).

CONTENTS

1. THE PROBLEM: LACK OF TRUE DIAGNOSTICS 1
   1.1 Purpose and Scope of This Paper 1
2. INTRODUCTION TO TESTING 2
   2.1 The Need for Diagnostics 3
3. A SOLUTION: PROPER SOFTWARE DESIGN 4
   3.1 Hardware and Software Partitioning 5
   3.2 Test Module Design 6
   3.3 Test Manager Design 11
      3.3.1 The RAM Test Anomaly 14
   3.4 The Results Analyzer 15
   3.5 The User Interface 17
   3.6 Other Troubleshooting Aids 19
4. THE PRODUCT IMPLEMENTATION 19
5. CONCLUSION 24
6. ACKNOWLEDGMENTS 25
7. BIBLIOGRAPHY 26

DEVELOPING TRUE DIAGNOSTIC SOFTWARE

Michael Prusynski

OSU CS501 - MSCS Research Paper & Project

1. THE PROBLEM: LACK OF TRUE DIAGNOSTICS

Although the true purpose of "diagnostic" software should pertain to diagnosing (as its name implies), its major shortfall is, in fact, its inability to diagnose. Much so-called diagnostic software of today does little more than verification -- exercising the hardware and reporting when a test failure occurs. This type of software may detect all the problems if it truly tests all the hardware, but it does not really diagnose the problems it finds. This fact has been re-iterated by complaints from both manufacturing technicians and customer service personel, based on their experiences of being left unassisted by the software once a failure has been detected. True diagnostic software not only needs to indicate **when** there is a hardware problem, it needs also to identify **where** the failing component or circuit is, or at least provide troubleshooting aids for faults that it cannot identify.

1.1 Purpose and Scope of This Paper

Because many diagnostic software packages fall short in at least one of the above areas (verification, fault identification, and troubleshooting), this research paper proposes a design of software that can satisfy the main requirements of all three categories. This paper can be considered a plea to the engineering community to do a better job of designing diagnostic software. Since the purpose of diagnostic software is to test hardware, any discussion of diagnostic design must also include some hardware discussion. However, this paper is meant to address the computer science field, so the discussion of hardware will be kept to a minimum. Many related topics have been researched, including the areas of Automated Test Equipment (ATE), Design for Testability (DFT), Artificially Intelligent (AI) expert systems, user interfaces, fault classification and

modeling, and test algorithms. Related terms and phrases are explained and referenced as they are used. This paper attempts to remain focused on techniques for the proper design of diagnostic software that is executed by a CPU within the unit under test. These design techniques are not intended to produce a diagnosis down to a single faulty component, although this is possible for RAM tests. With these techniques however, software can be designed that will either diagnose by isolating a fault to a circuit path, or help a human troubleshooter diagnose by providing him the software tools that allow him to use his skills to pinpoint the fault and repair it. Implementation examples are provided to further clarify these design techniques.

2. INTRODUCTION TO TESTING

Two of the main approaches to digital electronic testing can be termed structural and functional testing [Susskind73]. Structural testing, as the name implies, is based on the component structure. A primary example is a "bed-of-nails" implementation of an Automated Board Test (ABT) that connects to the inputs and outputs of the components on a circuit board. The board is tested by applying signals to the component's inputs and monitoring its outputs, which must comply with its structural description. This approach tests the discrete components independently, but does not test the overall functionality of the board. Functional testing verifies that the hardware behaves as intended, and this testing can be done at the system, board, or component level. Typical diagnostic software takes a functional approach that lies somewhere between board and component level.

When considering the purpose for testing, there are two main categories that are known under various names [Ligouri74]: (1) go/no-go, pass/fail, verification, acceptance or qualification testing - which determines that a unit is "good" or "bad", and (2) fault location, isolation, identification, or diagnostic testing - which indicates the probable cause of a failure. This second category of testing is usually applied to a unit that has failed a test of the first category.

2.1 The Need for Diagnostics

With the increase in hardware complexity and component counts, structurally testing each component becomes less economical and less feasible, while the concern for product reliability increases [Breuer76]. Diagnostics can provide an alternate means of verifying a product, however, testing a product with its normal operating software is sometimes good practice and often appears more economical. If using operating software can produce a system-level test relatively quickly, why bother developing diagnostics? One major flaw in this reasoning is apparent when considering **overall test coverage** of the hardware. With today's virtual memory operating systems the user often does not know what physical memory is being addressed. How can one ensure that **all** address lines are tested? How does one verify **all** memory locations in RAM or ROM? Another major drawback appears when the product inevitably fails a "system-level" test. With only a "go/no-go" functional test for a complex system, **how does the repair person identify the source of the problem**? These concerns can be resolved by building true diagnostic software into the product. Properly designed diagnostic software verifies hardware functions rather than software functions, so it can provide extensive test coverage of the hardware as well as localizing faults when they occur.

Once designed and implemented, internal diagnostic software brings with it other advantages [David79]. Internal diagnostics can reduce the manufacturing test time, the troubleshooting and re-work time, as well as reducing the need for external test fixturing and costly external test equipment. Since internal diagnostics go with the product, they can be executed by the customer (or at powerup) for verification, thereby increasing customer confidence in the reliability of the product. They can provide built-in troubleshooting aids, thereby reducing the time, labor, and cost of at-customer-site service calls.

The major differences between testing in manufacturing (production testing) and support testing in the "field" (usually a customer site) involve the number of faults in the unit under test,

and the repair strategy [Greenspan73]. The field service person can assume that a failed unit was once operational, and that the failure is due to a single fault. He may not require component-level fault identification, because the repair strategy is typically to "swap" the bad board with a good one rather than locating and repairing the faulty component at the customer site. However, localizing the fault will enable him to efficiently determine the faulty board, and provide an alternative to board swapping if the diagnosis indicates a problem that is easy to remedy. Eventually, a board that has been "swapped out" will need to be repaired, and component level diagnostics will again be required. When a unit is first tested in manufacturing, there is the possibility of catastrophic faults (voltages shorted to ground, etc.) which should be checked for, and the potential for multiple faults make good diagnostics invaluable. When a major fault or multiple faults prevent the diagnostics from executing, a hardware feature which forces a CPU op-code on the data bus can provide a starting point for troubleshooting. (This was incorporated into the DAS9200 board, but since it is a hardware design feature, further discussion is left to [Dobrin84].)

Diagnosis becomes more difficult as hardware complexity increases [Chang74], which may discourage the diagnostic designer in his efforts to incorporate true diagnosis capability into the software. However, this also indicates that a repair person is more likely to need diagnosis assistance from the software when complex hardware is involved. In order to provide true diagnosis software, the diagnostic designer must place himself in the role of "servant to service", in that he must consider the needs of the repair person as his highest priority. By designing diagnostic software for the more stringent troubleshooting and fault isolation requirements, the software package as a whole will provide verification as well.

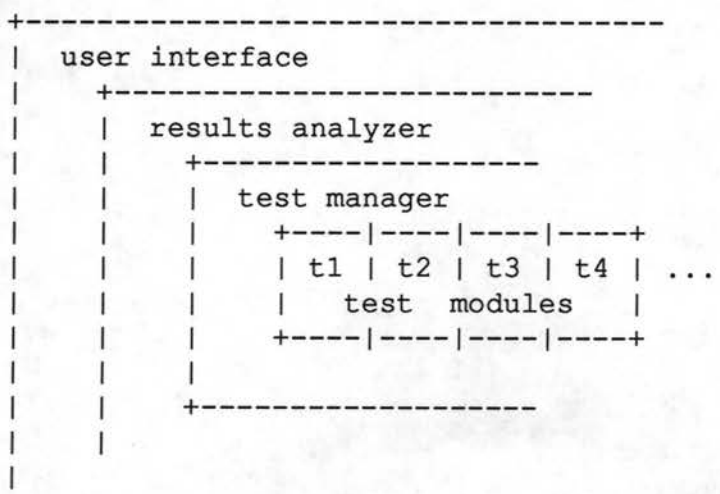
3. A SOLUTION: PROPER SOFTWARE DESIGN

Much has been published in recent years on hardware designed for testability [Bennetts84, Fujiwara85], fault modeling, path sensitization techniques [Cortner84], and other test algorithms

[IBM84], but much less attention has been given to techniques for effective diagnostic software design. This section does not describe specific algorithms or test approaches, but rather describes a set of general guidelines for the design of software that is "diagnostic" in more than name only.

3.1 Hardware and Software Partitioning

"Design for testability" means designing hardware that is testable, and should be the responsibility of the diagnostic engineer as well as the hardware design engineer [Dougherty88]. Just as the diagnostic engineer should encourage the design of hardware that can be divided into testable units, he must design the diagnostic software likewise. The design process starts in a top-down fashion by studying the circuitry, partitioning it into major test areas, and further dividing these areas into the smallest testable hardware units, such as the CPU, ROM, RAM, busses, selection circuitry, and so forth [Cortner84]. Each of these hardware units is tested by a respective "test module". The series of test modules make up the lowest level of diagnostic software. A second software layer should be a test manager that controls the test sequence, looping, and reporting of test results. An optional, but valuable, third software layer is a mini "expert system" [Morley86] program that analyzes the results of the tests as a whole. The top layer of software must provide a useful interface to the end user. Refer to the simplified conceptual diagram below:



3.2 Test Module Design

Because the test modules form the foundation of diagnostic software, proper design of this software layer is a major emphasis of this paper. The most important guideline is that each test module should be restricted to verify a minimum of the hardware, such as one circuit path. The test modules should be grouped according to circuit functions, and arranged in a ordered execution sequence that verifies from the hardware "kernel" outward, in small increments. This has several advantages. Because each test is narrowly focused, corresponding test error messages are more focused and more meaningful. As an extreme example, contrast a "system test failed" error message with one such as "XX interrupt enable line stuck". Also, tests restricted to a minimum of hardware are more apt to run properly and produce the proper diagnosis in the presence of multiple faults. Because the tests have a functional grouping, a sequential failure of tests within a particular group immediately identifies a failure of a major hardware block. It is worth noting that this guideline is contrary to typical efficiency goals for minimizing the number of tests for a product, because it encourages many small tests, including some that are redundant or overlapping. However, redundancy in tests can be a great aid in fault location [Susskind73]. If a component or circuit path cannot be verified by a unique test, but test1 verifies units A & B, test2 verifies units C & D, and test3 verifies units B & C, then tests 1 & 2 are sufficient for fault detection, and all three tests can be used for fault location. This redundancy can be further exploited by implementing a results analyzer (discussed in Section 3.5).

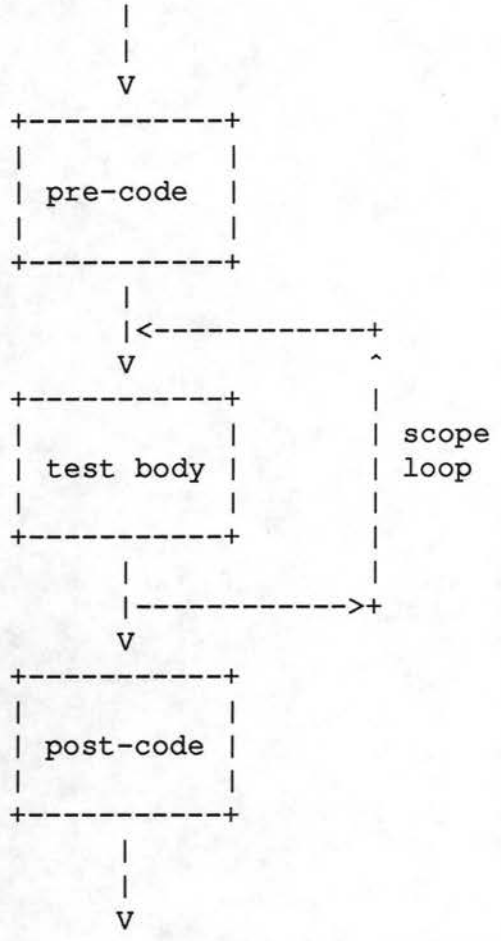
Other test design guidelines originate from troubleshooting requirements, and help create tests that can be used further by a troubleshooter in the event that the software diagnosis is insufficient. Typical troubleshooting practice is to probe the circuitry with an oscilloscope or logic probe, following circuit paths until a suspicious or unexpected signal is discovered, or backtracking from the point of failure to a point where the signal appears good [Cortner87]. Two more test design guidelines aimed at assisting this troubleshooting practice are as follows:

1. To allow the troubleshooter to select and run individual tests, each test should be designed to run without relying on initialization from other tests.

2. To provide the troubleshooter with a pulse that repeats rapidly enough to be usable by a scope, each test must be designed so that the repeated portion is kept to a minimum (called a "scope loop").

The common theme in these guidelines is that of reducing the tests to their smallest functional elements. Tests which stimulate a minimum of circuit paths also allow the troubleshooter to probe rapidly with a scope, rather than resorting to a logic analyzer to filter out the event he wishes to inspect from all the extraneous circuit activity (sometimes a very time consuming and frustrating experience).

These general requirements can be fulfilled by a test module design that consists of three parts, as shown in the diagram below.



The block marked "pre-code" represents initialization code that must be executed immediately prior to the actual test, and can remain unchanged during that test (sometimes called "loop-invariant" code). An example of this is code that initializes areas of memory for interrupt vectors or for multi-processor communication. Therefore, this initialization would be performed once prior to the initial entry to a test. On subsequent loops (repeated runs of a single test), the initialization would be bypassed. The main test body is where the actual test occurs. This is the minimum code that must be executed to actually perform the test, but includes test initialization that must be repeated for each run of the test. A simplified example is code that resets a signal, reads its state to verify that it is deasserted, then asserts the signal and re-verifies its new state. The "post-code" block is "cleanup" code that need to be done only once after the test completes, such as restoring interrupt vectors to their previous values. In some cases, a test for additional

errors or additional error analysis could be done in post-code. The test body should leave the test "cleanup" to the post code, to provide an immediate exit if an error occurs. This helps to "freeze" the hardware in the state in which the error occurred, in case further manual troubleshooting is necessary. The pre-code, test body, and post-code should be separate callable routines, in order to provide a common interface to the test manager. Note in the figure above that looping is not performed within any part of the test itself, but by the test manager which is explained in the Section 3.3.

A second major point is to maximize the use of readback hardware. Early input from the diagnostic designer to influence the hardware design towards testability is encouraged, which is another subject in itself [Bennetts84, Fujiwara85]. One write and read port is all that is required for testing of a data bus, which may span a major portion of the circuit board. Address bus tests can make use of available RAM, since a wide range of addresses with write and read capability is needed. If readback hardware has been incorporated into the circuit under test, the diagnostic software designer should use those readbacks to identify the location of the problem when a test failure occurs. A readback midway through a long circuit path may eliminate half of that path from suspicion if that test fails. Failure messages can say more than "XXX test failed" if readback hardware is utilized. These messages can instead be correlated to component numbers on a schematic diagram, thereby giving tests the ability to truly diagnose. In the latter case, the software designer must ensure that these diagnosis messages do not get truncated back into the popular "pass/fail" flag on their way through the various software layers before reaching the user.

The following example describes a test that utilizes readback hardware during post-code error analysis. In this case, a test must generate an interrupt and verify that it occurred. This is commonly done by pointing the interrupt vector to a special routine which sets a flag to say "I got here!". When designing a test of a specific signal, it is good practice to anticipate related failures. A good design should also take into account a stuck interrupt line or enable line, and verify that

each can be properly asserted and deasserted. Consider the following algorithm in "pseudo code":

```
< pre-code >
   initialize interrupt vector
< main test body >
   deassert interrupt signal
   clear "interrupt-received" flag
   enable interrupt
   poll interrupt flag in wait loop
   if interrupt flag is set before timeout,
      report "ERROR: interrupt stuck on" and exit
   (else)
   disable interrupt
   attempt to assert interrupt signal
   if interrupt flag is set before timeout,
      report "ERROR: interrupt enable stuck on" and exit
   (else)
   enable interrupt
   poll interrupt flag in wait loop
   if interrupt flag is not set before timeout,
      report "ERROR: interrupt not received"
   (else no errors -- test passed)
< post-code >
   if no interrupt
      read the interrupt port
      if the interrupt can be read
         report "Interrupt signal sensed at component XX"
              "Check components YY and ZZ"
              (the fault is between the readback point
               and the interrupt input to the CPU)
      else (the interrupt cannot be read)
         report "Can't sense interrupt signal at component XX"
              "Check components AA and BB"
              (fault is between the interrupt source
               and the readback point)
   restore original interrupt vector
   disable interrupts
```

This example assumes that the readback hardware is for diagnostics only, thus it is used only to isolate the failure and is not used if the interrupt occurs successfully. Note also that the interrupt flag is polled during each iteration of the wait loop, so that the loop terminates as soon as the interrupt occurs rather than waiting for the full loop count. This minimizes the time interval during repeated scope loops of the test.

A final point is to maximize the use of proven diagnostic algorithms (such as that for busses, ROM, and RAM) in common re-usable routines. After these general-purpose routines have been implemented, they have the obvious software development advantages. They become building blocks that allow the specific tests to be implemented quite rapidly, and making enhancements or bug corrections in the implementation usually involves a single base routine, rather than replicating changes in each of similar tests. But more importantly, the diagnosing ability of the base routines is automatically passed on to every dependent test. If several interrupt sources exist, the interrupt test algorithm above can provide a general-purpose base routine. All bus tests can use a common routine that implements an algorithm based on a single error-correcting Hamming code [Srini78], so that random errors in RAM or other readback devices do not produce an erroneous fault diagnosis. All ROM tests should use a common routine based on a proven checksum algorithm. A proven RAM test algorithm can be made general-purpose by requiring only the start and end addresses of the RAM, and possibly a bit map for ignoring specified bits on its data bus. The intent here is not to debate the effectiveness of the many test algorithms that have been published over the past 10 years, but to emphasize the value of re-using proven algorithms with good diagnosis capabilities.

3.3 Test Manager Design

This software layer provides test sequencing, looping, and message reporting services based on control flags set by the user or by the executing test. The advantage to this type of design is that the control checking for looping, error reporting, and "what next?" needs not be duplicated within each test. The test merely sets a flag corresponding its pass or fail result, and returns control to the test manager. This allows new tests to be implemented and added quite rapidly, once the test manager is operating.

For sequencing, the manager software maintains a list of pointers to each tests' pre-code, test body, and post-code. Some tests may require identical initialization, so their precode pointers would all reference the same routine. Some tests may require that their initialization and/or cleanup be performed within the test body, and have no need of the pre- or post-code. In this case, a "null" or zero pointer indicates to the test manager that this portion of a particular test does not exist (and is not to be executed). The test sequence is dictated by the operational mode, ultimately decided by the user (discussed in the user interface section). Complete verification requires a straight-forward execution of all available tests (possibly requiring that external fixtures are installed), and a corresponding list of all test pointers. Alternate test sequences (again determined by the user) can be handled simply by reorganizing the list so it contains only the selected tests' pointers in the proper order.

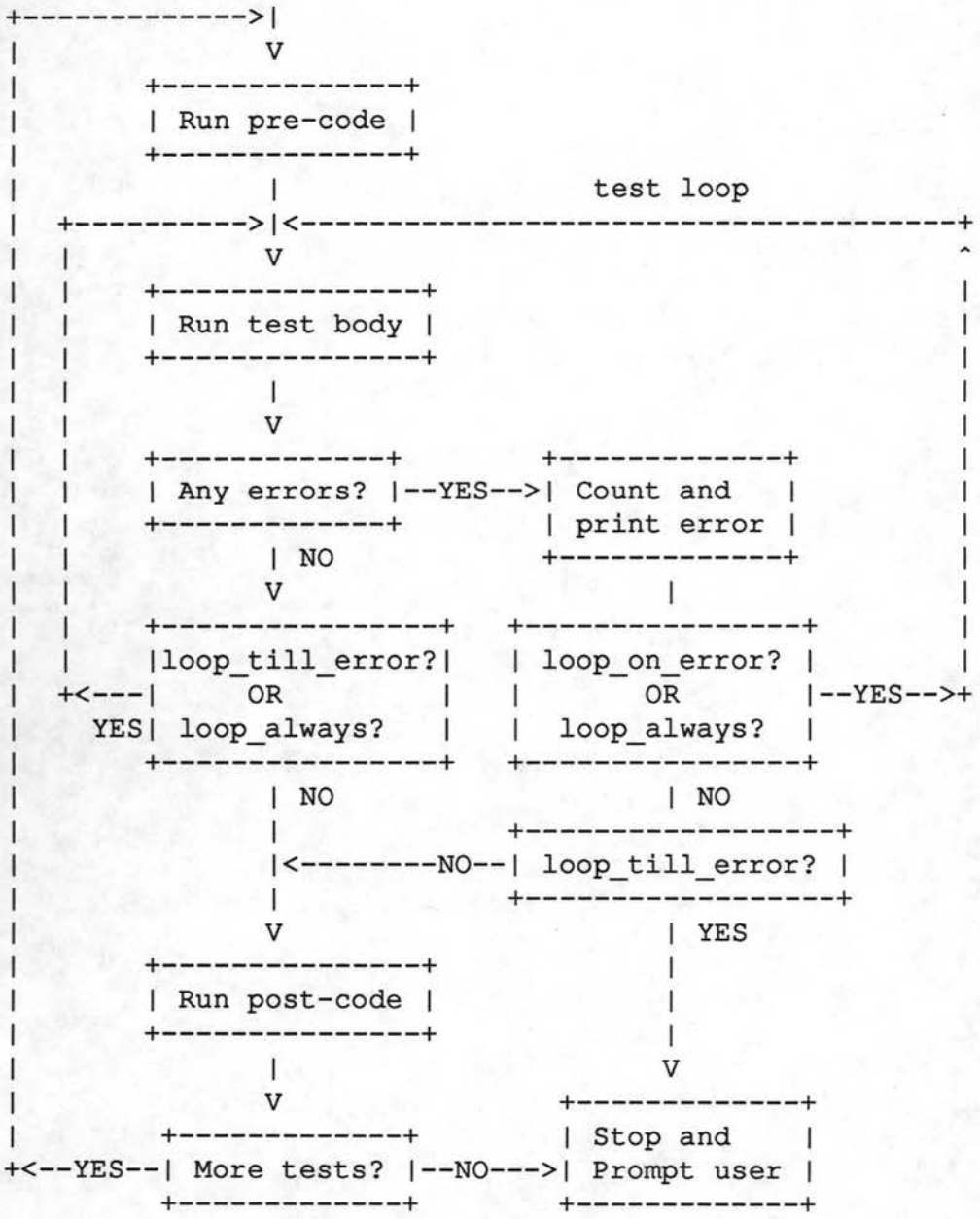
Looping capability is not only valuable for troubleshooting as discussed earlier, but is critical to reliability verification as well. Situations such as product "burn-in" require repeated execution of a sequence of all available tests, usually indicated by a loop control flag. Looping on the entire sequence of tests is sufficient for burn-in if the unit under test (UUT) never fails, but the purpose of burn-in is to "screen out" components, boards, or products that fail prematurely [Hanlon83]. Therefore, diagnostics should provide additional looping capabilities, such as looping continuously on the first test that fails, looping on the test as long as the fault is present, or stopping the verification loop after the first error. The looping capabilities provided are somewhat dependent on the error reporting capabilities (discussed next), and the recording device (printer, hard disk, etc.)

The test manager should provide at least two display modes, "no display" and "error display". As with sequencing and looping, the message reporting function should be responsibility of the test manager, independent from the test modules. The test module calls the test manager with a pointer to the message to be sent (typically an error report), but the test manager

decides how much to actually report based on the display mode selected by the user. For a tight scope loop, the display flag would be set to "no display". When called, the display routine would first check the flag and return immediately without reporting anything. The normal display mode is to report errors only, but a sometimes useful third display mode is to report the status of tests as they progress. This optional mode would require more reporting from within the test modules, similar to the technique of inserting "print" statements when debugging code. This "display all" mode should be handled in the same manner as displaying errors in that the corresponding display routine would decide whether to actually report the message.

These display modes have been implemented in various ways. For printing on a local terminal, the "display all" and "display error" routines check their corresponding flags, and return to the test if not set. Otherwise they each call a base print routine (similar to the C-language "printf" function) which handles the formatting and print controls.

The test sequencing, looping, and error printing are depicted in the flowchart below:



3.3.1 The RAM Test Anomaly

An often overlooked requirement of RAM tests is a unique looping/display mode needed when RAM test errors occur. Because of today's large RAM arrays and the need to minimize the time for testing RAM, contiguous RAM is typically considered a single test element. Regardless of the number of discrete RAM components, the RAM chips are usually verified by a single test.

Under normal looping conditions, a single test runs only until the first error, and then repeats. Although reporting one RAM failure is sufficient during verification, it is often valuable to find all RAM errors to diagnose the problem, especially when address bus and data bus tests for the RAM are not available. A special loop mode should be provided so that when RAM failures occur, the test can report the failure and continue at the next address to find any additional problems, rather than restarting at the first address of RAM. This situation can be generalized to non-RAM components as well, whenever multiple identical components are verified by a single test, or when numerous functions of a single VLSI component are exercised by a single test. This anomaly should not be overlooked by diagnostic designers.

Depending on the implementation, this situation can be handled by the test manager or within the test itself. In an earlier implementation, the special looping was handled in the RAM test's post code. If the post code detected that the RAM test failed, it incremented the failing address, and modified the test manager's "next test pointer" to re-execute the RAM test body using "failing address + 1" as the new starting address of the test. In my most recent implementation, the display flag is checked within the RAM test itself (after an error) and if set to "display all", the test continues through the remainder of RAM and reports any further errors before returning back to the test manager. (Sample output is shown in Section 4.)

3.4 The Results Analyzer

A test that is designed to run independently, without relying on initialization from previous tests is prevented from making run-time decisions based on previous test failures. An independent test's error reporting does not consider results of other tests. As a result, most diagnostics do not sufficiently analyze the summation of failure data to identify the problem and isolate it to a component or area. If an integral component failed (a fault on a major data bus for example), diagnostics could display numerous individual error messages, each with a different assessment

of where the fault is located. In this situation, the repair person is made aware of various failures, but is frustrated by the software's confusing diagnosis which does not help him pinpoint and rectify the problem.

This is an optional layer of diagnostic software that is too often omitted. The value of a results analyzer is that it takes a "macro" view of test failures (seeing the "forest" instead of only "trees"). The results analyzer could "know" that a bus test failure invalidates more specific error messages displayed by subsequent tests (which could be quite confusing to the repair person). By diagnosing RAM error messages (failing addresses, actual and expected data), the expert system could identify a bus or decoder fault, or pinpoint a single faulty RAM chip. Based on the complexity and cost of the system under test, the results analyzer could have the capabilities of large expert systems [Rich83]. A more moderate approach ranges in capability from analyzing error message content and changing the run-time sequence based on its conclusions, to a minimum approach which analyzes only the sequence of test failures. The communication interface between the test manager and results analyzer depends on the desired analysis capabilites. Minimally, a mapping of each test with its pass or fail result is required for post analysis of the failure sequence. For a more extensive diagnosis, error messages must be either stored for later inspection or analyzed real-time (as the errors are reported).

Some simplifying assumptions were made during the implementation. Unlike expert systems that must deal with with weighted probabilities caused by the uncertainty of human symptoms or measurements, this simplified version has exact, "digital" input data -- an indication that a test either passed or failed -- there is no "middle ground". This implementation does not contain knowledge of electronic component structures [Rehfuss84], and it does not query the user for additional details as it progresses through its diagnosis. This simple expert system reasons backward from its top-level goal: find the cause of the test failure(s) [Laffey86].

Only the minimal "bit-map" approach mentioned above has been implemented, for reasons explained in Section 4. The result analyzer's input data is a static record of failure data, where a bit position in the record corresponds to a specific test. As the test sequence is run, the pass/fail result of each test is recorded in sequential bits of memory. This expert system contains a knowledge base specific to the unit under test, consisting mainly of "if-then-else" cases, such as "if test1 & test2 failed, but test3 passed, then the fault is...". It uses a "fault dictionary" approach where the failure pattern is matched with pre-determined causes. Different faults cause a different failure pattern, so those failure patterns are programmed into the knowledge base and correlated to the components that cause them.

3.5 The User Interface

Since this paper deals with general design guidelines, this user-interface discussion will be kept "device-independent". That is, an ASCII input/output device is assumed, and special graphics devices are not discussed. A specific product implementation is described in the next section.

Verification and troubleshooting pose different requirements for the user interface. Verification requires a "run all" mode of sequential test execution, possibly with looping on the entire sequence, that is easily selectable. A menu-based user interface [Rubenstein84] can be generally applied and can be made quick and easy to use. To simplify the user's selections, some interfaces have "built-in" loop and display options, such as looping that occurs automatically on a test that fails. A single menu with options for "run all" or individual test selection is sufficient for verification, but troubleshooting requires more flexibility. Some menus lead the user down a very restrictive path, frustrating the troubleshooter because he cannot get the diagnostics to do what he wants. So although the loop and display control mechanisms are sometimes inter-related, they should be presented to the troubleshooter as separate selectable options. On the contrary, verification users frequently follow the same menu path, and are annoyed by too many selections

that get repetitious and seem needless.

The acceptable number of items per menu and levels of menus has been the subject of much study [Paap86], and the optimum user interface can be an entire subject in itself. One solution to the above requirements is a top-level menu that presents an immediate choice between verification or troubleshooting. The second-level menus that appear next differ based on that initial choice, with the verification menus being minimal in number and options, and the troubleshooting side having more options, possibly distributed over several menu levels. The figure below depicts a previous implementation that follows this scheme with multiple levels of small menus.

```
                    MODE SELECT MENU
                    ----------------
                1 - AUTOMATIC VERIFICATION MODE
                2 - TEST SELECTION MODE

                Enter desired number followed by RETURN key: _
```

If menu item #1 was selected:

```
      AUTO MODE MENU
      --------------
1 - RUN ALL TESTS ONCE
2 - RUN ALL TESTS CONTINUALLY
```

If menu item #2 was selected:

```
SELECT MODE MENU
----------------
1 - RUN TEST AAA
2 - RUN TEST BBB
3 - RUN TEST CCC
  (and so on...)

LOOP MODE MENU
--------------
1 - LOOP ON ERROR
2 - LOOP UNTIL ERROR
3 - DO NOT LOOP
```

An acceptable alternative is a top-level menu that presents all the basic options (looping, display, run all tests, run one test), but requires only one user selection and uses "defaults" for selections not made by the user. An example of this user interface is shown in the implementation section.

3.6 Other Troubleshooting Aids

Several features can be added to diagnostic software to help the troubleshooter diagnose a problem, if the software diagnosis is insufficient:

1. "Breakpoints" to stop test execution (RAM-based software only) and freeze the state of the hardware for inspection. This item is unnecessary if the tests can be forced to terminate as soon as the error occurs, without running the "cleanup" code.

2. The ability to exercise signals which are otherwise untested because they cannot be verified automatically, such as a repeated toggling or pulsing of a signal which allows it to be verified manually with a scope or logic probe.

3. The ability to read and write memory, with looping if desired. Commands for "read byte" and "write byte" would be minimally sufficient, with looping as a useful addition for troubleshooting. This feature is depicted below, in the implementation section.

4. THE PRODUCT IMPLEMENTATION

This section describes an actual implementation based on the guidelines presented in this paper. This implementation gives credence to these design guidelines, and hopefully the implementation examples will further clarify the guidelines. The implementation is described in a top-down fashion by illustrating the user-interface and its use. Some of the test names have been generalized to avoid confusion.

The user-interface consists of basically a one-level menu, with temporary look-up menus that list the tests and test groups. (Test groups are called "functions" in the actual implementation, because of a pre-established precedence.) The main menu and its default selections are shown below, with self-explanatory examples of user input and program output. Notice how the user "prompt" gives an immediate indication of the current selections (function, print mode, loop

mode). The one or two characters after the ">" symbol of the prompt represents what is typed in by the user.

```
****  DIAGNOSTICS DEBUG MONITOR  ****
?N  Show available functions (?f) or tests (?t)
fN  Select function (N = function Number in decimal)
tN  Select test (N = test Number in decimal)
/   Unselect function (selects all tests)
lN  l0=no loop, l1=loop on error, l2=loop till error, l3=loop always
pN  p0=no print, p1=print errors, p2=print all test info

r   Run selected tests
a   Analyze test results
h   Display this help menu
q   Quit (exit from diagnostics)

Run all tests, PrintErrors, NoLoop > ?f

Available functions:
  f0-Kernel
  f1-Bus tests
  f2-Interrupts
  f3-LAN circuitry
```

```
Run all tests, PrintErrors, NoLoop > ?t

Available tests:
  f0-Kernel Tests
     t0-RAM chips
     t1-EPROM checksum
     t2-CPU
  f1-Bus Tests
     t0-RAM address bus
     t1-RAM data bus
     t2-I/O data bus
     t3-Bus error
  f2-Interrupts
     t0-Clock interrupt
     t1-Port Interrupt
     t2-Interface interrupt
  f3-LAN Circuitry
     t0-LAN PROM check
     t1-LAN data bus
     t2-LAN interrupt
     t3-LAN self-test
     t4-LAN address setup
     t5-LAN internal loopback
     t6-LAN SIA loopback
     t7-LAN external loopback
```

Notice that only the tests for the selected group are listed. Above, all tests are listed when "Run all tests" is the selected function. Below, only the kernel tests are listed because function "f0" has been selected.

```
Run all tests, PrintErrors, NoLoop > f0
f0-Kernel tests, PrintErrors, NoLoop > ?t
Available tests:
  t0-RAM chips
  t1-EPROM checksum
  t2-CPU

f0-Kernel tests, PrintErrors, NoLoop > t0
f0-Kernel, t0-RAM chips, PrintErrors, NoLoop >
```

Test failure information is given by a run-time error message, followed by a "Fault Analysis" message after the test completes its looping (if looping was selected). The first line of the error message alerts the user that the test failed, and the second line identifies the faulty area

after isolating the failure through readback hardware or additional error analysis. An error in the LAN interrupt test, for example, may produce the following error display (the "<>" symbol denotes the schematic page):

```
f0-Kernel, t0-RAM chips, PrintErrors, NoLoop > f3t2
f3-LAN circuitry, t2-LAN interrupt, PrintErrors, NoLoop > r
Error: LANINT not received by CPU
Fault Analysis: LANINT present at <7>U348 but not at CPU; check <2>U364
```

The RAM test loop/print anomaly is depicted in the following example. When the RAM chip test is first run with "loop always" mode, notice how the same (first) error is repeated. When "print all" is selected, all RAM errors are displayed, and in this example it becomes apparent that the errors are caused by a stuck data line rather than a failure within a RAM chip.

```
f3-LAN circuitry, t2-LAN interrupt, PrintErrors, NoLoop > f0t013
f0-Kernel, t0-RAM chips, PrintErrors, LoopAlways > r
Error: Location= 80000  Expected data= 55  Actual data= 65
Error: Location= 80000  Expected data= 55  Actual data= 65
Error: Location= 80000  Expected data= 55  Actual data= 65
                  .
                  .
                  .
f0-Kernel, t0-RAM chips, PrintErrors, LoopAlways > p2
f0-Kernel, t0-RAM chips, PrintAll, LoopAlways > r
Error: Location= 80000  Expected data= 55  Actual data= 65
Error: Location= 80001  Expected data= 55  Actual data= 65
Error: Location= 80002  Expected data= 55  Actual data= 65
Error: Location= 80003  Expected data= 55  Actual data= 65
                  .
                  .
                  .
```

The memory manipulation commands were implemented as "read byte" and "write byte" and "dump memory". These commands were added to the main menu, and they are illustrated in the figures below:

```
****  DIAGNOSTICS DEBUG MONITOR  ****
dN  Dump memory starting at hex address N
rN  Read byte at hex address N (obeys Loop_all and Print_all)
wM N Write hex byte M to hex address N (obeys Loop_all and Print_all)
?N  Show available functions (?f) or tests (?t)
fN  Select function (N = function Number in decimal)
tN  Select test (N = test Number in decimal)
/   Unselect function (selects all tests)
lN  l0=no loop, l1=loop on error, l2=loop till error, l3=loop always
pN  p0=no print, p1=print errors, p2=print all test info

r   Run selected tests
a   Analyze test results
h   Display this help menu
q   Quit (exit from diagnostics)


f0-Kernel, t0-RAM chips, PrintAll, LoopAlways > l0
f0-Kernel, t0-RAM chips, PrintAll, NoLoop > r80067
  80067= 07

f0-Kernel, t0-RAM chips, PrintAll, NoLoop > w80067 55
  Write completed.

f0-Kernel, t0-RAM chips, PrintAll, NoLoop > l3
f0-Kernel, t0-RAM chips, PrintAll, LoopAlways > w80067 55
Looping on write to 80067(hex), press any key to quit _

f0-Kernel, t0-RAM chips, PrintAll, LoopAlways > d80000

        0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
80000= 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F ................
80010= 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F ................
80020= 74 68 69 73 20 69 73 20 61 6E 20 41 53 43 49 49 this is an ASCII
80030= 72 65 70 72 65 73 65 6E 74 61 74 69 6F 6E 20 0F representation .
80040= 6F 66 20 74 68 65 20 73 61 60 65 20 64 61 74 61 of the same data
80050= 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F ................
80060= 00 01 02 03 04 05 06 55 08 09 0A 0B 0C 0D 0E 0F .......U........
80070= 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F ................
80080= 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F ................
80090= 28 43 29 54 65 6B 74 72 6F 6E 69 78 31 39 38 38 (C)Tektronix1988
800A0= 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F ................
800B0= 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F ................
800C0= 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F ................
800D0= 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F ................
800E0= 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F ................
800F0= 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F ................
Press space bar to continue or any other key to quit _
```

The results analyzer is invoked with the "a" command, and sample output is shown below:

```
f0-Kernel, t0-RAM chips, PrintAll, NoLoop > a

Recording test results...
Searching for failure patterns...
  All interrupt tests failed --
  -fault probably at <2>U364: enable stuck hi or PORTINT stuck lo
Failure pattern analysis completed.
```

As mentioned earlier, the results analyzer (as implemented) only considers a pass/fail bit for each test, rather than analyzing error message content. An inherent limitation with this approach is that all conclusions are a result of a match with predicted failure patterns (developed apriori). Failure patterns may show up in manufacturing that were not predicted, and will not draw any matching conclusion from the results analyzer. This implementation resides in EPROM on the circuit board and is not easily modified once the board is into production. However, this approach was implemented because 32 bits of test result information can be passed to the "system controller" on another board in the DAS9200 product. The system controller can perform the results analysis in an identical fashion, with the program loaded from a diagnostic floppy disk which is maintained for manufacturing use. It is my intent to upgrade a floppy-based results analyzer whenever unpredicted failure patterns appear in manufacturing.

5. CONCLUSION

So-called "diagnostic" software that provides only verification is the bare minimum a diagnostic software designer should be responsible for. The previous discussion has shown that in manufacturing and other service areas there is a definite need for software that is "diagnostic" in more than name only. The design techniques proposed and discussed in this paper can produce test software worthy of the name "diagnostic", so that it provides verification, fault identification, and troubleshooting aids. They are a result of much research and my eight years of experience as a diagnostic design engineer and manufacturing test engineer. These techniques have been

authenticated by their implementation in a real product, and their worth should be realized whenever this implementation is used by manufacturing and field service.

6. ACKNOWLEDGMENTS

I wish to recognize Maria Agoston, David Maguire, and John Richartz, who as former coworkers, helped develop some of the ideas presented here. I also acknowledge Forest Ross, who originated the Diagnostic Debug Monitor (DDM) user interface for the DAS9200.

7. BIBLIOGRAPHY

Of the references listed, the book "Digital Test Engineering" by J. Max Cortner is by far the best single reference on fault modeling, test algorithms, diagnostics, troubleshooting, and other related topics.

Agoston, M. & Dale, I. & Irlandez, L. & Hoke, T. "Building an Expert System for Digital Board Diagnosis using HIPE", Oregon State University Project Report, 1986.

Bennetts, R.G. "Design of Testable Logic Circuits", London: Addison-Wesley, 1984.

Breuer, M.A. & Friedman, A.D. "Diagnosis & Reliable Design of Digital Systems", Rockville, MD: Computer Science Press, 1976.

Chang, H.Y. & Manning, E. & Metze G. "Fault Diagnosis of Digital Systems", Huntington, NY: R.E. Krieger, 1974.

Cortner, J. Max. "Digital Test Engineering", New York: John Wiley & Sons, 1987.

David, John G. "The Value of Internal Diagnostics to Tektronix" Tek Labs Internal Publication, May 1979.

Dobrin, A. & Novak, F. "Freerunning the M68000", Electronics Test, p. 124, April 1984.

Daugherty, David W. "Taking on Design-for-Test in the Real Engineering World", Electronics Test, pp. 16-20, February 1988.

[Greenspan73] see Ligouri, Fred.

Fujiwara, Hideo. "Logic Testing & Design for Testability", Cambridge, MA: MIT Press, 1985.

Hanlon, Everett. "Intelligent Burn-in for High Density Memories", Electronics Test, pp. 60-64, January 1983.

Hewlett Packard. "Computer Aided Test Symposium", Spring 1988.

IBM. "Coding and Error Control", IBM Journal of Research & Development, Vol.28 #2, March 1984. (The entire issue is devoted to related articles.)

Kirkpatrick, Donald C. "Diagnostic Fundamentals for the System Planner", Technology Report (Tektronix Internal Publication), August 1981.

Laffey, T.J. & Perkins, W.A. & Nguyen, T.A. "Reasoning About Fault Diagnosis with LES", IEEE Expert, pp. 13-20, Spring 1986.

Liguori, Fred (Editor). "Automatic Test Equipment: Hardware, Software, & Management", New York: IEEE Press, 1974. This is a collection of reprinted IEEE articles. The most noteworthy are on pages 57-64 and 93-100 and are referenced (respectively) by their authors:
Greenspan, Arnold M. "Automatic Test Systems", IEEE Transactions on Instrumentation and Measurement", November 1973.
Susskind, A.K. "Diagnostics for Logic Networks", IEEE Spectrum, October 1973

Morley, R. & Taylor, W. "Why Bother With Expert Systems?", Digital Design, pp. 47-51, July 1986.

Paap, K. & Roske-Hofstrand, R. "The Optimal Number of Menu Options per Panel", Human Factors Journal, pp. 377-385, August 1986.

Pradhan, D.K. (Editor). "Fault Tolerant Computing", Englewood Cliffs, NJ: Prentice-Hall, 1986.

Rehfuss, S. & Freiling, M. & Alexander, J. "Particularity in Engineering Data", Oregon State University Technical Report, 1984.

Rich, Elaine. "Artificial Intelligence", New York: McGraw-Hill, pp. 284-291, 1983.

Rubenstein, R. & Hersh H. "The Human Factor", Bedford, MA: Digital Press, 1984

Srini, Vason P. "Fault Location in Semiconductor RAM", IEEE Transactions on Computers, pp. 349-358, April 1978.

[Susskind73] see Ligouri, Fred.