INTERACTIVE MAINTENANCE TERMINAL

FAULT ISOLATION PROGRAM

by

Michael Henry Bulat

SUBMITTED TO THE DEPARTMENT OF
ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE IN PARTIAL
FULFILLMENT OF THE
REQUIREMENTS FOR THE
DEGREE OF

BACHELOR OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

January 1981

Signature of Author _____

Department of Electrical Engineering and Computer Science

January 28, 1981

Certified by _____

Albert Vezza

Thesis Supervisor

Accepted by _____

Chairman, Department Committee

Abstract

INTERACTIVE MAINTENANCE TERMINAL
FAULT ISOLATION PROGRAM

by

MICHAEL HENRY BULAT

Submitted to the Department of Electrical Engineering
and Computer Science on January 28, 1981 in partial fulfillment
of the requirements for the Degree of Bachelor of Science in
Computer Science

An interactive maintenance terminal is a stand-alone minicomputer capable of directing an operator through a fault isolation tree. In order to assess the feasibility and complexity of building such a terminal a simulation of its operating characteristics was implemented on a General Electric Training System. The study was done to obtain an appreciation of desireable display characteristics and machine response times and to refine preconceptions of how the maintenance data base should be implemented. A language called TL was developed to automate the traversing of fault isolation graphs. An interpreter and editor for TL was developed in a String Processing Oriented Machine Language (SPOML). It was found that a graph structure was capable of modelling the widest variety of fault isolation graphs. The editor facilitated the entry of the fault-isolation procedures. The project demonstrates that the interactive maintenance terminal is a feasible idea but a machine with more capability than the GE Training System would be needed to implement it effectively.

Thesis Supervisor: Mr. Albert Vezza
Title: Senior Research Scientist

Table of Contents

1. Introduction 3
    1.1. Purpose 3
    1.2. Related Work 4
    1.3. System Overview 6
2. Control Structures 9
    2.1. Generalization of Test Operations 9
    2.2. Generalization of Test Sequences to Control Structures 10
    2.3. Classifications of Control Structures 11
        2.3.1. Trees and Graphs 11
    2.4. Modifications to Control Structure Operation 13
3. Implementation 16
    3.1. General Electric Training System[2]
        3.1.1. General Features 16
        3.1.2. String Processing Oriented Machine Language 17
        3.1.3. Architecture 17
    3.2. TL Interpreter 17
        3.2.1. TL Interpreter Function 17
        3.2.2. Instruction Set 18
        3.2.3. Instruction Addressing 19
        3.2.4. Execution Order 20
        3.2.5. Paging 21
        3.2.6. Man-Machine Interface 22
        3.2.7. Sub-graphs 24
        3.2.8. State Variables 26
    3.3. TL Interpreter Structure 26
    3.4. Editor 26
4. Example TL Program 29
5. Results and Conclusions 38
I. TL Interpreter Flowchart 42

[2]GETS is a GE patented minicomputer

List of Figures

Figure 1-1: Manual Fault Isolation System Configuration and Interactions 6
Figure 1-2: System Configuration with Independant Editting Facility 8
Figure 1-3: Maintenance Terminal Configuration used for this Work 7
Figure 2-1: Test Sequence Generalization 10
Figure 2-2: Dynamic Test Sequence 11
Figure 2-3: Static Test Structure 12
Figure 2-4: Binary Tree 13
Figure 2-5: Directed Graph 13
Figure 2-6: Equivalent Sub-graphs in a Control Structure 14
Figure 3-1: Instruction Format 18
Figure 3-2: Instruction Grouping 20
Figure 3-3: Sample Instruction Node 21
Figure 3-4: Memory Map 22
Figure 3-5: Display Management 23
Figure 3-6: Linkages between Control and Data Structures 24
Figure 3-7: Sub-graph Implementation 25
Figure 3-8: TL Interpreter Flow Chart 27
Figure 3-9: Editor Display Format 28
Figure 3-10: Alternative Display Format 28
Figure 4-1: Radio Manual 30
Figure 4-2: Radio Manual (Continued) 31
Figure 4-3: Control Structure 32
Figure 4-4: Screen 1 35
Figure 4-5: Screen 2 36
Figure 4-6: Screen 3 37
Figure 5-1: Flowchart (page 1) 42
Figure 5-2: Flowchart (page 2) 43
Figure 5-3: Flowchart (page 3) 44
Figure 5-4: Flowchart (page 4) 45
Figure 5-5: Flowchart (page 5) 46
Figure 5-6: Flowchart (page 6) 47
Figure 5-7: Flowchart (page 7) 48

List of Tables

Table 3-1: Condition Codes 19
Table 3-2: Operation Codes 19
Table 3-3: Interpreter Commands 24
Table 3-4: Editor Commands 28
Table 4-1: Nodes 0 - 7 33
Table 4-2: Nodes 8 - 15 34
Table 4-3: Nodes 16 - 17 29

Acknowledgements

I would like to thank Albert Vezza (MIT) and Jack Francis (GE), for their suggestions and preperation of this report. I would also like to thank Doran Morrison for providing valuable assistance and information.

1. Introduction

1.1. Purpose

A prime requirement of many computer systems is availability, i.e., the percentage of time that the system is functional. In many applications availability is critical. While the system is being designed a variety of approaches can be taken to maximize reliability and maintainability, but regardless of the precautions taken part of any system will fail at times. When part of the system fails it must be repaired as soon as possible in order to return to its full functionality.

This thesis considers the problem of isolating faults in systems and repairing them in a minimal amount of time. An enormous variety of approaches can be taken to isolate and repair faults. Fault isolation procedures can be informal, written down, stored in software, or implemented in special equipment such as Automatic Test Equipment (ATE) [1] and/or Built In Test Equipment (BITE) [9, 5]. ATE may be connected to a system at any time in order to repair it. BITE is designed into a given system for the purposes of maintenance and fault isolation.

It was decided by General Electric that a control system designed and built at GE would have BITE to reduce down time. GE required that the BITE primarily be an automated version of the existing repair manuals. The system was conceived with the assumption that the skill level of the operators was low enough to prohibit great flexibility in test selection. The volume and complexity of the tests leave little room for the operator to design his own test sequence. After a fault is detected the BITE will have main responsiblity for isolating it and directing repairs.

At present, all failures in the General Electric Control System (GECS) must be diagnosed and repaired by an operator who uses predefined fault isolation and correction procedures. The fault isolation/correction procedures (FICP) are contained in large volumes of written records. An operator must spend an excessive amount of time searching for and executing FICPs. The operator can easily commit errors while following the FICPs. In order to reduce the diagnosis and repair time and decrease operator error, it was proposed that the FICPs be automated in the form of an Interactive Maintenance Terminal (IMT).

The IMT is designed to completely replace the written FICPs and is designed to provide additional functions. The IMT is designed to be totally responsible for guiding an operator through the FICPs. It is designed to be a graphics computer terminal capable of displaying diagrams, text, and pictures related to the GECS. It is also capable of inspecting various busses in the GECS and asserting data on them. In short, the IMT is designed to use data in a variety of forms (text, diagrams, pictures, digital) to interactively guide an operator in repairing the GECS.

In order to better understand the hardware and software necesary to perform the IMT's function an attempt was made to simulate its function on the General Electric Training System (GETS). The GETS is a stand-alone minicomputer featuring; plasma panel, floppy disk, keyboard, and sonic pen. Emphasis was put on the user interface and optimum modelling of the FICPs. In the course of modelling the FICPs, a Test Language (TL) was developed to allow their easy automation. A TL interpreter was written to execute TL programs and an editor was developed to aid in creating them. The interpreter and editor were written in SPOML (String Processing Oriented Machine Language), a language available on the GETS. Several programs were written in TL to demonstrate the proposed functioning of the IMT.

1.2. Related Work

Much work in the area of fault isolation and repair has been done. Many artificial intelligence programs have been written to localize failures in a variety of systems and many systems exist which employ BITE or ATE. Some of these programs and systems will be described below.

One example of a system with BITE capabilities is the AEGIS system [1]. An integral part of AEGIS is the Operational Readiness Test System (ORTS). ORTS is an on-line integrated test system. It can conduct on-line tests, evaluate status in the primary system, control AEGIS system initialization, and coordinate all diagnostic and maintenance activity. Tests run on the ORTS can be initiated by the operator or by a program. Once they are initiated tests run to completion under program control. If a fault is isolated the operator is referred to maintenance procedures on microfiche. There is little emphasis on manual testing in this system.

On the other hand, some systems are designed with a large number of manual operations in mind. This is usually done when operations are too costly to automate. Raytheon developed a system that guides an operator through a series of tests with its AN/DPM-22 missle test station [5]. This system uses an automated optical viewer to display drawings and diagrams to the operator. The system performs automatic diagnosis to the limits of its capabilities; then if it has not isolated the fault, it guides the operator through manual fault isolation procedures.

Much research has been done on the man/machine interface in automatic and semi-automatic testing systems. The degree of operator control over the testing procedure is important. The operator can be restricted to button pushing, or he can completely specify all tests and the order in which they are to be executed. Depending on the skill level of the operator, varying degrees of flexibility will be appropriate. One study done at a military repair shop with automatic testing capability gives a good indication of what functions are required for a useful man-machine

interface [9]. It was found that options allowing the operator to alter the normal sequence of tests were desirable. In this system all tests are set up and interpreted by the computer. Human intervention is allowed between tests only. Little emphasis is placed on computer direction of manual tests.

While not specifically designed for BITE or ATE systems, many programs have been written to isolate faults using heuristic techniques. These programs take a different approach to fault isolation and diagnosis than the IMT, but their purpose is ultimately the same. Programs desgned for debugging and diagnosis include Sussman's HACKER [13], Goldstein's MYCROFT [7], Shortliffe's MYCIN [12], Brown's SOPHIE [3], De Kleer's trouble shooting program [6], and Sussman's LOCAL [14]. Most of these programs are designed to isolate faults in a small well defined system. Some interesting features of theses programs are outlined below.

HACKER tries to solve block stacking problems by debugging old programs so that they can be applied to new problems. Bugs are explained in terms of predefined classes.

MYCROFT is designed to debug a subset of LOGO programs. Models of the intended result together with the program in question are used to infer the user's intent. MYCROFT concentrates on the interfaces of plan steps, a major source of errors.

MYCIN bases a system of rules to diagnose patients with symptoms of bacterial infection. The program questions the operator on the patient's symptoms until it is reasonably sure of the cause. It can answer questions about its decisions to any level of detail. Rules are represented hierarchally in the system and can be modified by the operator.

SOPHIE is designed to teach trouble shooting of power supplies. The program is geared to supporting a natural language interface. SOPHIE's methodology is based on converting qualitative questions to quantitative ones and predicting the results on the power supply in question.

De Kleer's trouble shooting program is designed to analyze DC circuits by looking only at the individual properties of the parts in question. Information gained from measurements and Kirchoff's laws is propagated through a circuit until multiple results are given for a point. Depending on the agreement of the results deductions can be made on the functioning of the circuit.

LOCAL is designed to isolate faults in radio receivers. LOCAL is based on a hierarchal system of experts which contain information on the radios design. Experts are called with symptoms

which they attempt to explain in terms of faults in their domain. Experts call other experts until the problem cannot be explained in terms of faults lower in the hierarchy.

A large knowledge base of detailed fault isolation procedures already exist for the GECS. It was thought that transferring these procedures to the IMT in some manner would perform as well. On this assumption a simple strategy was developed to facilitate modelling the FICPs in software. The approach taken in this work was of more modest proportions and less sophistication than the foregoing AI programs. However the program will explicitly guide the human trouble shooter in measurement taking and decision making.

1.3. System Overview

The maintenance terminal is only one part of a system including; a computer to be tested by the maintenance terminal, an operator, a set of programs to run on the terminal, and possibly a third computer for editting and maintaining the maintenance data base. Figure 1-1 shows the system configured for troubleshooting.

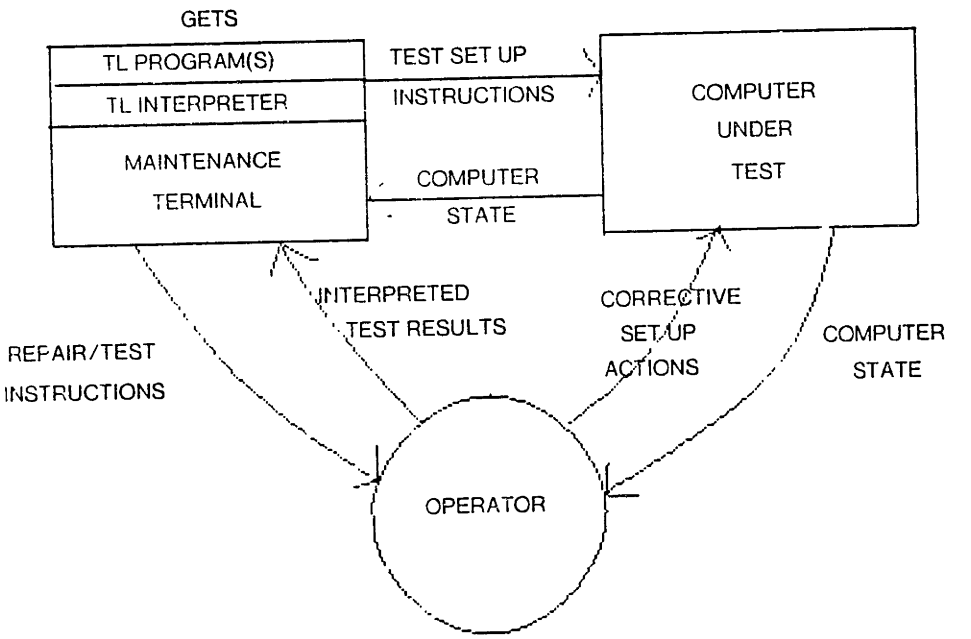


Figure 1-1: Manual Fault Isolation System Configuration and Interactions

In this system the maintenance terminal monitors the computer for faults and informs the operator if any occur. It has the ability to perform some testing automatically but its primary

purpose is to guide an operator through manual testing and fault isolation procedures. The manual FIPs require the operator to perform manual operations on the computer and interpret test results, functions the MT cannot perform by itself.

For the simulation described in this thesis only the interactions between the operator and the MT are considered. The computer under test is not simulated in any fashion. The programs developed completely ignore any interaction between the computer under test and the MT. The operator makes up his own test results for any demonstration programs. During a simulated fault isolation procedure the GETS is running a TL interpreter which in turn is runnig a TL program which guides the operator through a specific set of tests. The majority of t his thesis is concerned with the TL interpreter and the execution of TL programs.

The MT can also be considered to be part of a system for editting the TL programs. All editting can be done on the MT but another computer can be used to create and edit the TL programs. Figure 1-2 shows this configuration. For the simulation done in this thesis there is no editting computer. All editting facilities are editted on the GETS. A structured TL editor was programmed in addition to the GETS general text editor. The configuration of hardware and software developed in this thesis is shown in figure 1-3. SPOML is the language in which the TL editor, interpreter, and utility programs are written. Before these programs are described a system for descrbing the fault isolation procedures will be developed.

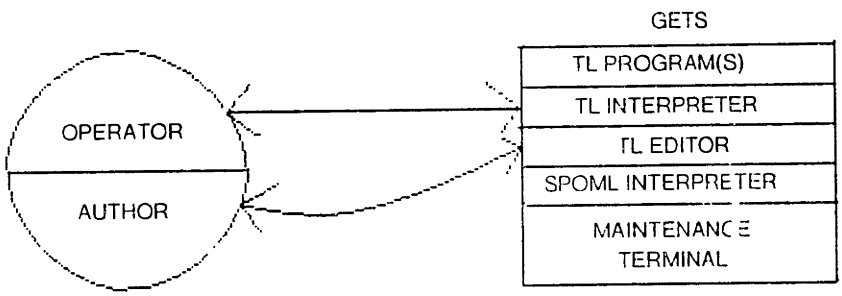


Figure 1-3: Maintenance Terminal Configuration used for this Work

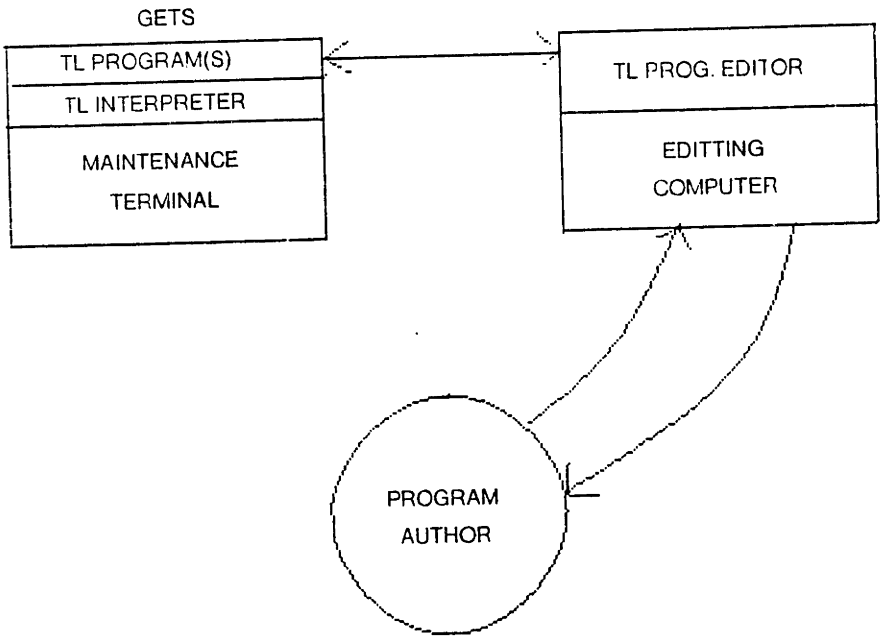


Figure 1-2: System Configuration with Independant Editting Facility

2. Control Structures

Given the existence of a collection of tests designed to isolate a fault in a machine it is useful to be able to model them. A collection of tests designed to isolate faults is termed a fault isolation tree. A fault isolation tree will be modelled by a control structure with every node representing a test and arrows representing the relationships between the tests. As information is gained in the form of test results the next test to be performed is usually chosen on the basis of these results. In the case of manual fault isolation trees this information will be represented as predefined answers to questions about the test results. The static control structure represents all the tests known and their relationships to each other. When the control structure is interpreted a sequence of tests will be performed which will hopefully find the cause of a certain problem. When a given test is being performed it will be said to be in the state corresponding to that test. A control structure is an abstract model of a set of tests designed to isolate faults in a given machine.

2.1. Generalization of Test Operations

The majority of the tests that were to be automated can be modelled by the format depicted in Figure 2-1. All of the examined tests fit the following sequence of operations. First, the machine under test is forced into some state necessary for the test to be correctly performed. This is done in one of two ways:

1. Electrically altering the machines state.
2. Instructing an operator to perform certain operations on the machine to acheive the desired state.

Because the tests are primarily manual in nature the second type of test set-up action would be more prevalent. The second phase of any test requires the operator to interpret the results of the test. This interpretation usually requires inspecting some part of the machine under test and answering a question about its state. In automatic cases, the results of the test would be available to the terminal as digital data. Third and last, the results of the test are used to decide which test to perform next. In the case of predefined test sequences the decision is made based on a predefined map of user inputs to next tests. In the automatic case this mapping is far more complicated. The machine may have to interpret values which may be acceptable over a broad range, for instance.

The machine can be forced into a state in two different ways. The first way, previously described, is to change some setting on the machine's controls or to verify that the machine is in a certain state. The second way involves taking action that can be termed "manual action" such

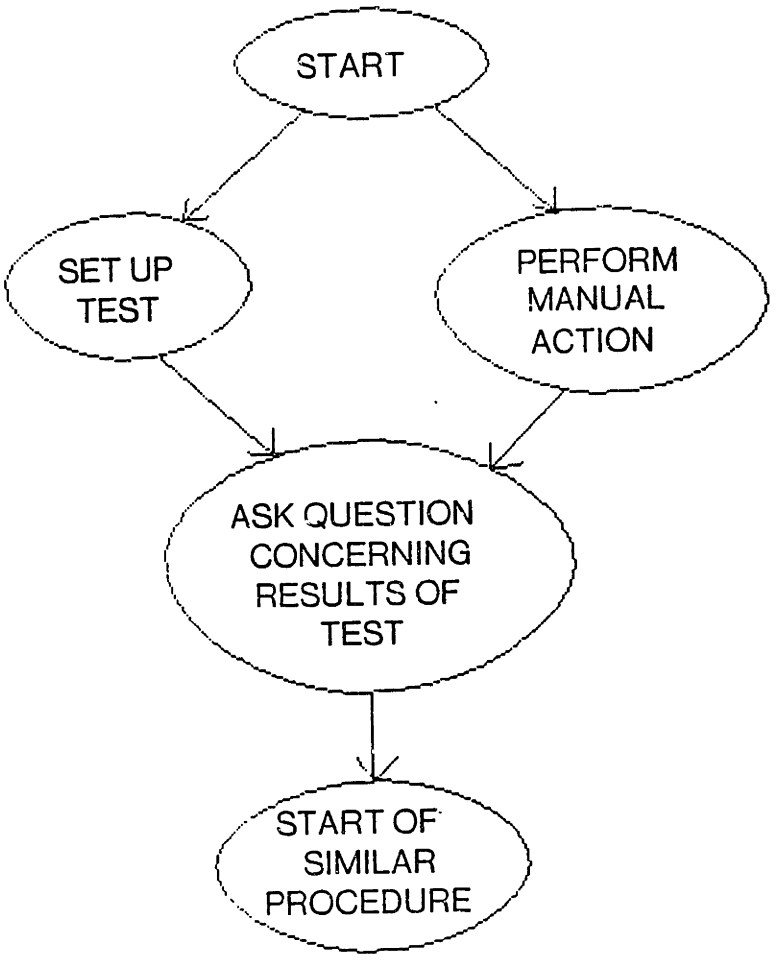


Figure 2-1: Test Sequence Generalization

as replacing or altering hardware or pushing buttons.

2.2. Generalization of Test Sequences to Control Structures

One test is rarely sufficient to isolate a fault or repair a nonfunctioning machine. Usually a series of tests is required where the results of one test determine which test is to be performed next. The series of tests taken when isolating any one fault can be represented as a linear sequence as in Figure 2-2.

The complete set of tests necessary to cover all cases of a symptom could be represented by the structure of Figure 2-3. Each test sequence, as defined above, is represented as a circle. The mapping between results and next tests is shown as an arrow labelled with a result and pointing at

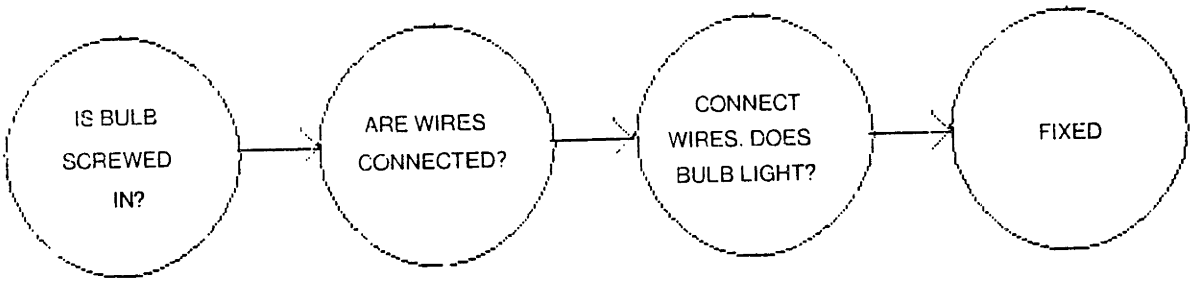


Figure 2-2: Dynamic Test Sequence

the next test. The test structure wi l be termed a control structure. It is statically similar to the mathematical concept of the graph. A graph is defined as a set of points and a set of point pairs corresponding to links between the points. Dynamically it is similar to the finite state machine (FSM), with each each test corresponding to a state. Performing a given test will be considered equivalent to being in that test state in the control structure.

Each control structure will have two types of states which will be considered different from the rest. First, each control structure will have a set of states designated as starting states. These will be defined as the only states where a test procedure can be validly started. Second, each control structure will have a set of terminating states. At the terminating states the next test is not specified in the normal manner because by definition there are no more tests to perform. What happens will be explained later.

2.3. Classifications of Control Structures

Based upon the topology of the control structure it is possible to classify them in the same manner as graphs. The relative usefulness of several classifications as models of test procedures will be looked at. In addition several changes in the dynamic control structure operation will be examined.

2.3.1. Trees and Graphs

Initially it was thought that a binary tree control structure as depicted in Figure 2-4 would be useful for modelling test procedures. The motivation for using binary trees was the great body of knowledge existing about them. However, binary trees were ruled out because the existing test procedures frequently contained topologies similar to the one in Figure 2-5.

This type of structure could represent retrying all tests after some corrective action had been

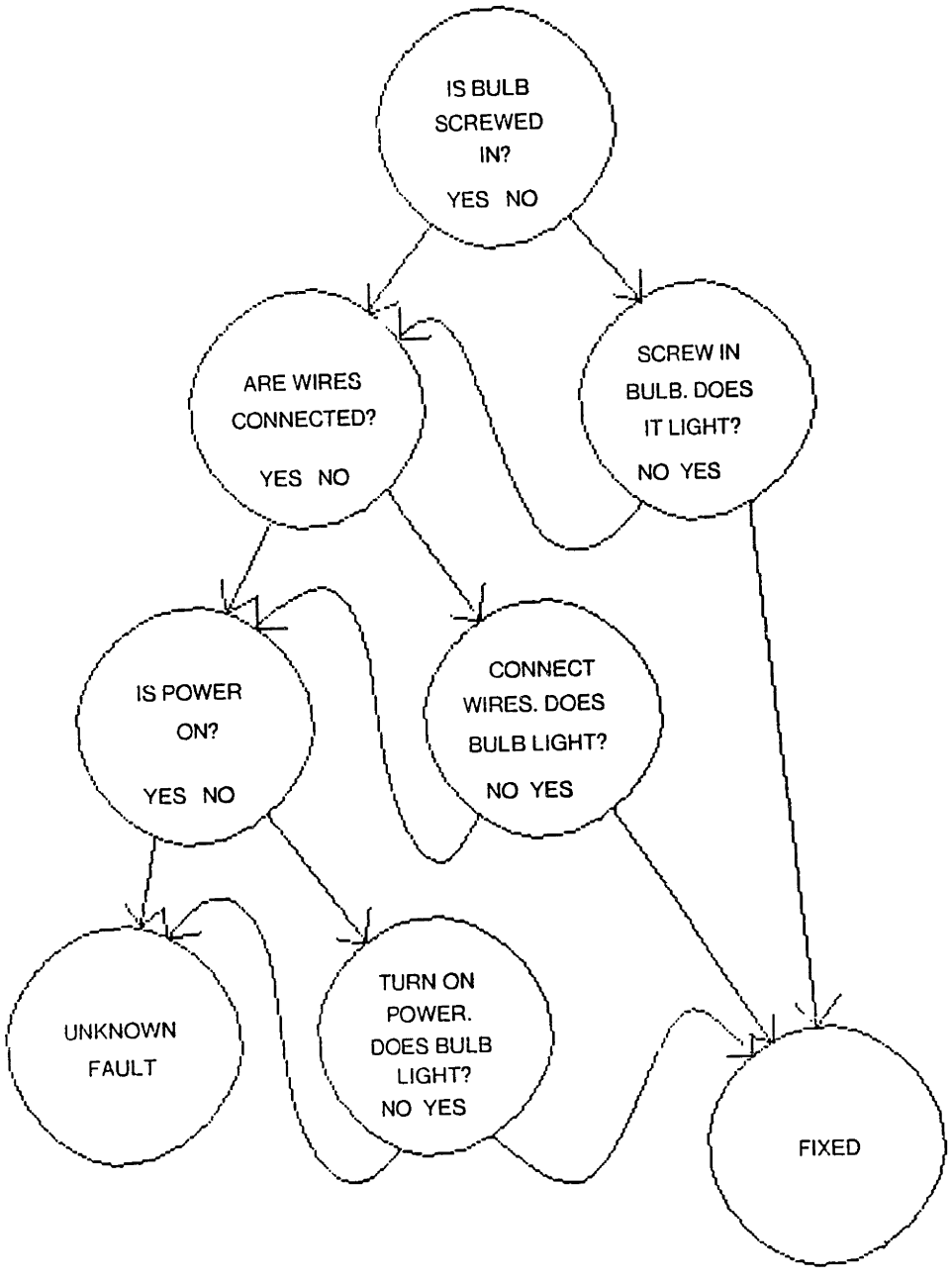


Figure 2-3: Static Test Structure

taken. Multibranch trees were ruled out for the same reason as binary trees.

The control structure of Figure 2-5 is best termed a directed graph. Directed implies that the transitions between states are one way. This reflects the idea that the results of one test imply the next test and not the inverse. For the set of fault isolation trees examined at GE the directed graph was found to be the most suitable control structure for the modelling task.

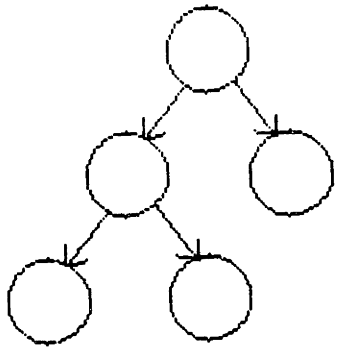


Figure 2-4: Binary Tree

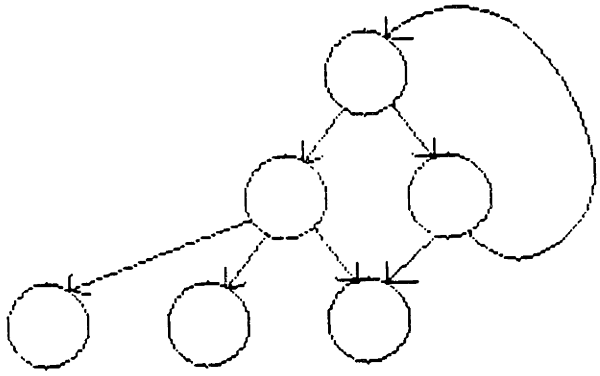


Figure 2-5: Directed Graph

2.4. Modifications to Control Structure Operation

In order to make the implementation of the control structure more efficient it was found useful to add the concepts of sub-graphs and state variables. Sub-graphs are equivalent to control structures in all ways. They can be inserted into another control structure during its operation and act exactly like a part of the other control structure. The motivation for adding this feature

was the appearance of equivalent sub-graphs at various points in a typical test procedure as in Figure 2-6.

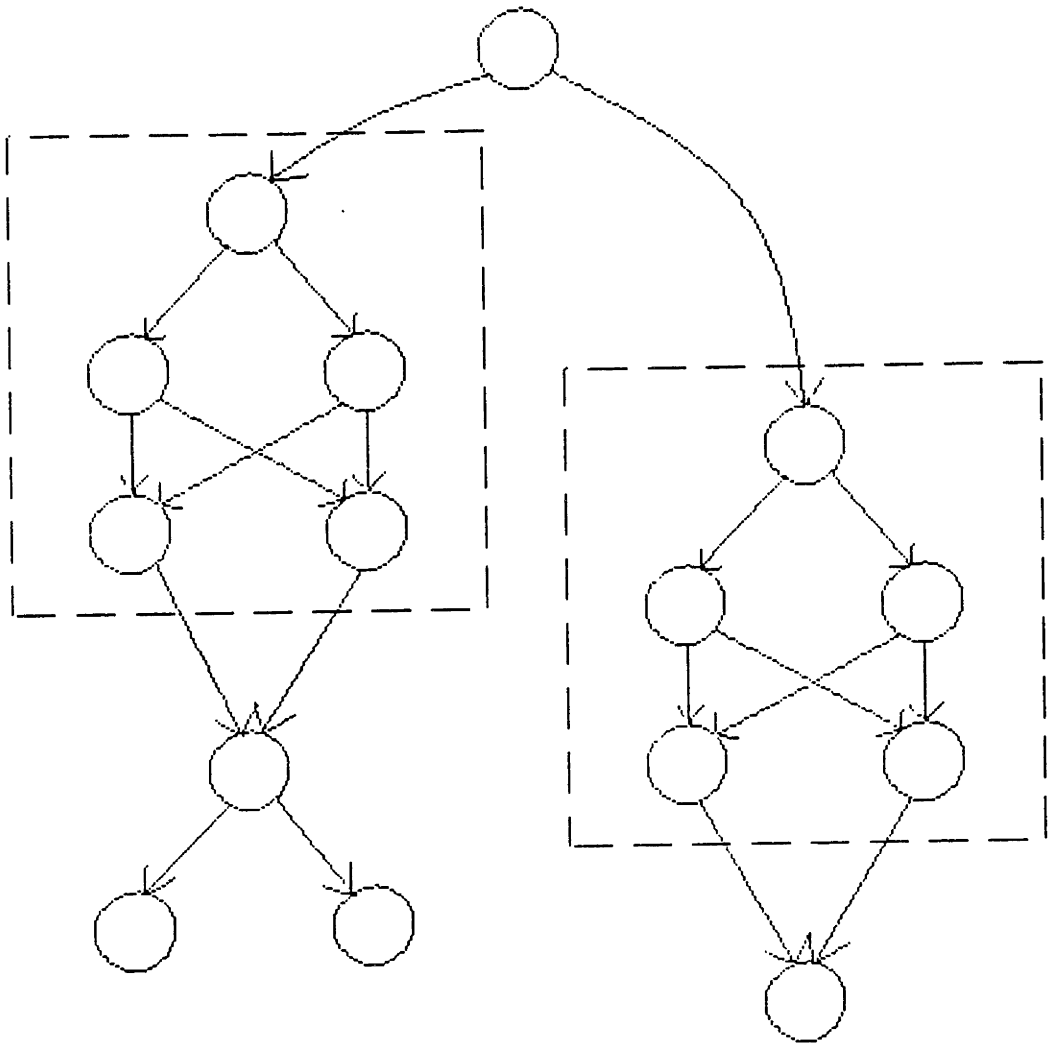


Figure 2-6: Equivalent Sub-graphs in a Control Structure

Normally all transfers from one state to another are immediately specified by the test results. When sub-graphs are entered and exited the procedure is altered. The transfer into a sub-graph is equivalent to any other transfer but information must be stored as to where to transfer control when the subgraph is exited. This information takes the form of a map from results in the sub-graph terminal states to states in the calling structure. This information is stored on a stack and used to determine return states when the sub-graph is exited.

Another change designed to reduce the size of a control structure is the addition of state variables. In the course of gathering infromation on the machine under test it was found convienent to be able to store data in variables which could be accesed from anywhere in the structure. Without these variables all information gained from the tests is represented only by the state of the control structure. At a point where a state variable can take on n values it could only be replaced by duplicating the control structure n times. This is where the savings in space occurs.

Normally the flow through the control structure proceeds forward following the direction of the arrows. Under ideal circumstances forward flow would be totally sufficient to follow the test sequence. But occasionally it may be useful to back up to a previous test to repeat it or reinterpret the results. Given a control structure with sub-graphs and state variables many implementation problems arise. One problem occurs when backing into sub-graphs. The correct map from terminating states to external states must be determined and pushed on the stack.

A more serious problem arises when previous assignments of variables must be determined. The path taken to any state must be recorded so any previous assignments can be located.

The control structures described in this chapter are similar to Wood's [15] Augmented Transition Networks (ATN). Each has the capability of calling sub-graphs, performing arbitrary operations on registers, and conditionally changing states. In control structures arbitrary operations and tests are executed at the states. In ATNs arbitrary operations are attached to the transitions. In ATNs sub-graphs are specified on transitions as conditions which must be satisfied to reach the next state. In control structures sub-graphs may have numerous terminating states each of which can return to a different node in the calling structure. In ATNs only one return state is specified. ATNs and control structures differ functionally. ATNs are used primarily for parsing languages while control structures are used to direct human operators through a sequence of actions.

3. Implementation

As stated in the introduction, the problem this thesis addresses is automatic fault isolation. The specific objective is to automate "tree" type trouble shooting procedures. If a system of programs can be developed to perform this task, the following would be accomplished:

1. The complexity and feasibility of automating the maintenance data base on an interactive display terminal would be better understood.

2. A better understandng of desireable display characteristcs and machine latencies would be gained.

3. A general data base structure for automating the maintenance data base on a minicomputer based maintenance terminal would be developed.

The General Electric training System (GETS) was selected for this task because it is a minicomputer based terminal similar to the proposed maintenance terminal and it was readily accessible for use.

3.1. General Electric Training System[1]

3.1.1. General Features

The GETS is a stand-alone microprocessor based minicomputer. The GETS features a plasma panel, slide projector, sonic pen. keyboard, and floppy disk drive. The plasma panel measures 8.5 by 8.5 inches and has a vertical and horizontal resolution of 512 points. The plasma panel is transparent allowing slides to be rear projected onto it. The projector can randomly access up to 80 slides under program control. A sonic pen is used to interact directly with the display. The sonic pen simultaneously emits a spark and an electrical signal when touched to the screen. The sound of the spark is detected by two bar microphones located above and to the left of the screen. The signals from the microphones and the pen are used to determine the pen's location on the screen. The floppy drive is a single sided, single density device with a capacity of 340K words. The only high level language implemented on the GETS is String Processng Oriented Machine Language.

---

[1] GETS is a GE patented minicomputer

3.1.2. String Processing Oriented Machine Language

All programs were written in String Processing Oriented Machine Language (SPOML) because it was more powerful than the underlying machine language. As the name implies SPOML is designed primarily for text processing. Most text processing takes place in 3 stacks. SPOML features; a nested control structure, macro subroutine capability, and the capablty of calling subroutines written in machine language.

3.1.3. Architecture

The architecture of the GETS played an important role in many of the implementation details. Processing is centered around three stacks of 256 words each, the left, right, and match stacks. SPOML programs are executed from a 2K program memory. A 2K word buffer and a 256x(11 bit) stack are also available but not ideal y suited for use by SPOML programs.

3.2. TL Interpreter

3.2.1. TL Interpreter Function

The TL interpreter is responsible for recognizing a set of instructions, translating them into SPOML, and executng them. The interpreter is responsible for maintaining the display in an orderly state and processing operator inputs. In addition the interpreter recognizes certain operator inputs as commands. The interpreter also manages the transfer of instructions from the floppy disk to the stacks. When the TL interpreter is executing it resides in a 2K program memory. TL programs reside on the floppy disk. When a particular node is to be executed it is transferred from the floppy disk into the right and left stacks. Only the instruction fields at the top of the right or left stacks are accesible to the TL interpreter. The match stack is used to store characters input by the operator through the keyboard ar sonic pen. The transfer of input characters to the match stack is performed by the GETS independant of the TL interpreter. A 256x11 Working File Buffer (WFB) stack is used to store node names during sub-graph calls. The WFB stack is used in conjunction with the WFB for editting and is free during the execution of SPOML programs. The high and low words of the return node name are stored in 2 consecutive locations on the stack. The WFB is a 2K memory used for editting. When the TL interpreter is executing the WFB is used to keep a record of all the nodes executed in a given program run. There are 64 registers available to SPOML programs. The TL interpreter reserves 26 of these, named 'a' to 'z', for use by TL programs.

3.2.2. Instruction Set

Each instruction is represented by five 8-bit words. The five words correspond to the five fields; CONDITION, COMPARE, OPERATION, DATA 1, and DATA 2 as shown in Figure 3-1.

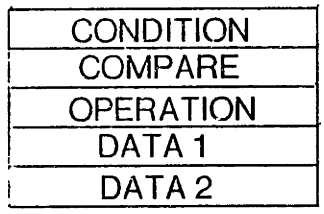


Figure 3-1: Instruction Format

The condition field together with the compare field control execution of the operation. The operation field specifies what action is to be performed. The 2 DATA fields are used for direct addressing of picture files, slides, and instructions. The DATA 2 field is used for indirect addressing of registers. The data fields can represent 2 independant integers or one twos complement integer wth a range of -32768 to +32767.

The condition codes are listed in Table 3-1. Three conditional and one unconditional type were implemented. Code 128 causes the instruction to be executed when any character is received as input. Code 129 causes the instruction to be executed immediately and unconditionally. Code 130 executes the instruction if the contents of D2 are equal to the variable addressed by D1. Code 131 executes the instruction if the contents of D1 are equal to the current input character. The current input character is created when the operator makes an entry through the keybaord or sonic pen. The current input charcter is consumed if it satisfies a condition code 131 or 128 or it fails to satisfy any of the conditional instructions in a node. The operation codes are listed in Table 3-2. It can be seen that the operation and condition fields are much larger than necessary. This is because SPOML is primarily a string processing language with few bit operations. Thus any space saved by using smaller fields would not be worth the extra time and space needed to implement these operations.

CONDITION CODE	MEANING
128	Execute instruction when any input is received
129	Execute instruction immediately unconditionally
130	Execute if <<D1>> = <D2>
131	Execute if <D2> equals current input character

Table 3-1: Condition Codes

OPERATION CODE	MEANING
132	Display slide <D2>
133	Display picture file <D2>
134	Push <D2> on control stack
135	Return - POP next control number from stack and GOTO it
136	PUSH current control number + 1 on stack and GOTO <D2>
137	GOTO control number <D2>
138	<D2> => <D1>
139	Clear screen
140	Simulate keyboard input of <D2>
141	PUSH control number on stack and GOTO <D2>
142	GOTO control number <<D2>>
143	Equivalent to instruction 140 followed by 135
144	PUSH current control number + 1 on stack and GOTO <<D1>>
145	Turn projector bulb off

Table 3-2: Operation Codes

3.2.3. Instruction Addressing

The previous section outlined all instructions recognized by the interpreter. These instructions are intended to perform all the operations needed to automate manual fault isolation procedures. A method for addressing sets of instructions together was developed. Each set corresponds to a complete indivisible test of the type described in section 2.2. It also corresponds to the abstract idea of a node in a control structure. In practice such sets of instructions are delimited by a special character followed by a double word integer uniquely indentifying that set. The format is shown in Figure 3-2. All instructions following a node delimiter up to the next delimiter are addressed as a single unit. From this point on these sets of instructions will be referred to as nodes. Nodes are addressed by the immediate data fields D1 and D2. Control transfers are also specified by the control stack when returning from sub-graph calls. No other addressing modes

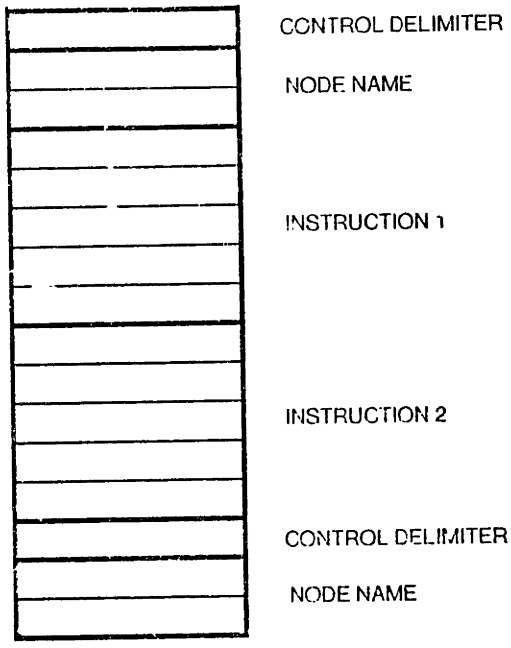


Figure 3-2: Instruction Grouping

were implemented. The effect of this addressing scheme on execution order will be discussed in the next section.

3.2.4. Execution Order

Within a node instructions are executed sequentially starting with the first instruction after the control delimiter. There is one restriction on what instructions must appear in a node. Control must be transferred to another node before the next control delimiter is encountered. If this doesn't happen an error results. When the transfer is conditional upon an operator input the interpreter will not error out if the input isn't valid instead it will read inputs until a valid one is detected. A valid input is defined as a character that matches one of the compare fields in any of the input conditional jump commands. Control can only be transferred to another node as the result of the execution of a valid instruction. When control is transferred it is always to the first instruction in a node. There is no mechanism for jumping to any other point within the node. Figure 3-3 lists the instructions of a sample node. When this node is entered the screen will be cleared and picture # 320 will be displayed (10 is the vertical height of picture 320 in characters). When the interpreter encounters the second instruction it will wait for an operator input. When an input is received it will be compared to the compare field. If it equals 'A' then control will be

CONDITION	COMPARE	OPERATION	DATA 1	DATA 2
UNCOND.	----	CLR/DSPLY	10	320
KEY COND.	A	GOTO	----	2
KEY COND.	B	GOTO	----	3

Figure 3-3: Sample Instruction Node

transferred to node #2. If it does not equal 'A' then the input is saved. The saved input is compared to 'B' in the third instruction. If the input isn't equal to 'B' then the input is thrown away and the interpreter returns to the first conditional jump instruction and waits for a new input. Any instructions after the first conditional jump will be executed multiple times if invalid inputs are received.

3.2.5. Paging

For programs of even moderate size a great quantity of memory would be required. Specifically, the total space required is given by the formula:

Total space = 3(# control nodes) + 5(total instructions)

Since the amount of memory available for processing is 256 words it is necessary to store programs on floppy disk and to transfer them to the stacks when needed. Stack sizes restricted the block size to 256 words. Since references to control nodes are by name it is necesary to provide a mechanism for mapping node names to physical locations on the disk. Ideally a map like the one shown in Figure 3-4 would be implemented for flexiblity. Instead a more rigid scheme was adapted. Arbitrarily 8 nodes, contigous in value, were assigned to every block. Given that a control structure started in block n then nodes 0 through 7 would be in block n, nodes 8 through 15 would be in block n+1, etc. When a node is addressed by name its block number can be computed by the formula:

BLOCK NUMBER = N + [NODE NAME/8]

This method was used because the control node location could be computed as a function of the control node value. This avoided an extra disk access which would have been required to retreive a memory map. The map would have to be stored on disk because lack of main memory. Because of the interactive nature of the system the flexibility ganed by the use of a map couldn't be justified in view of the time required for an extra disk access.

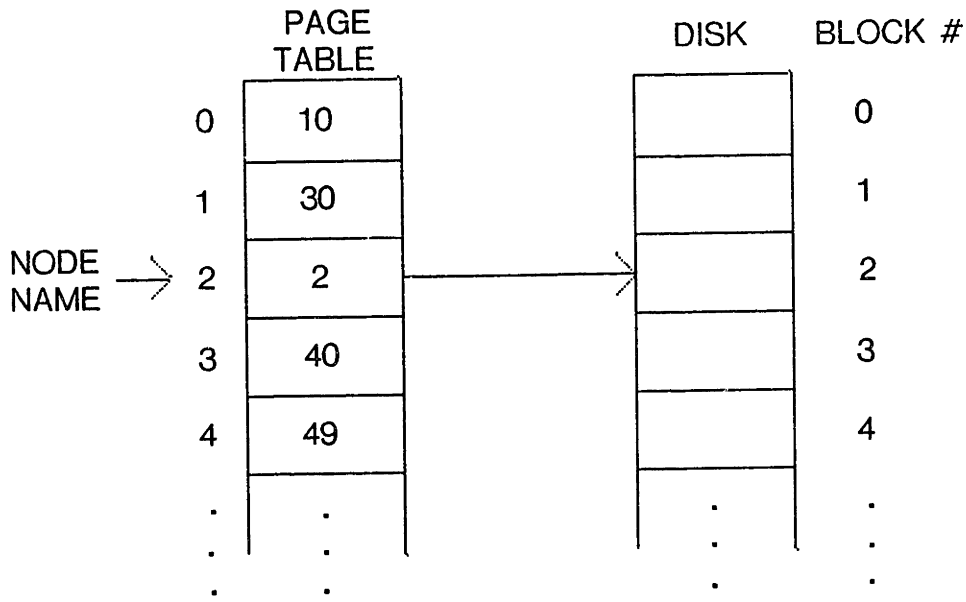


Figure 3-4: Memory Map

3.2.6. Man-Machine Interface

In addition to executng the instructions of a control structure the interpreter also maintains the display, provides visual feedback to operator inputs, and processes high level commands. As seen in section 3.2.2 there are several instructions to display picture files. These files contain information which is interpreted by the GETS to produce line drawings and text on the screen. The files are prepared independantly of the control structures and are stored on disk. Picture files are represented by an indentifying number which is mapped into a physical location on disk. Typically the picture files would contain instructions for replacing parts or setting up tests and a diagnostic question. When the question is answered an arrow is displayed below the selected response and any picture file from the next control node is displayed. When the display cannot hold the next picture the screen is cleared, the picture occupying the lowest portion on the screen is redisplayed at the top of the screen, and the next picture is displayed below it. See Figure 3-5 for an illustration of this sequence. The vertical height of each picture is given in the DATA 1 field of the display instruction. With ths information the interpreter knows when a picture will overflow the screen. In addition to the visual information in the picture file there are also sensitized areas. Portions of the text in the picture file are activated so that when they are touched by the sonic pen

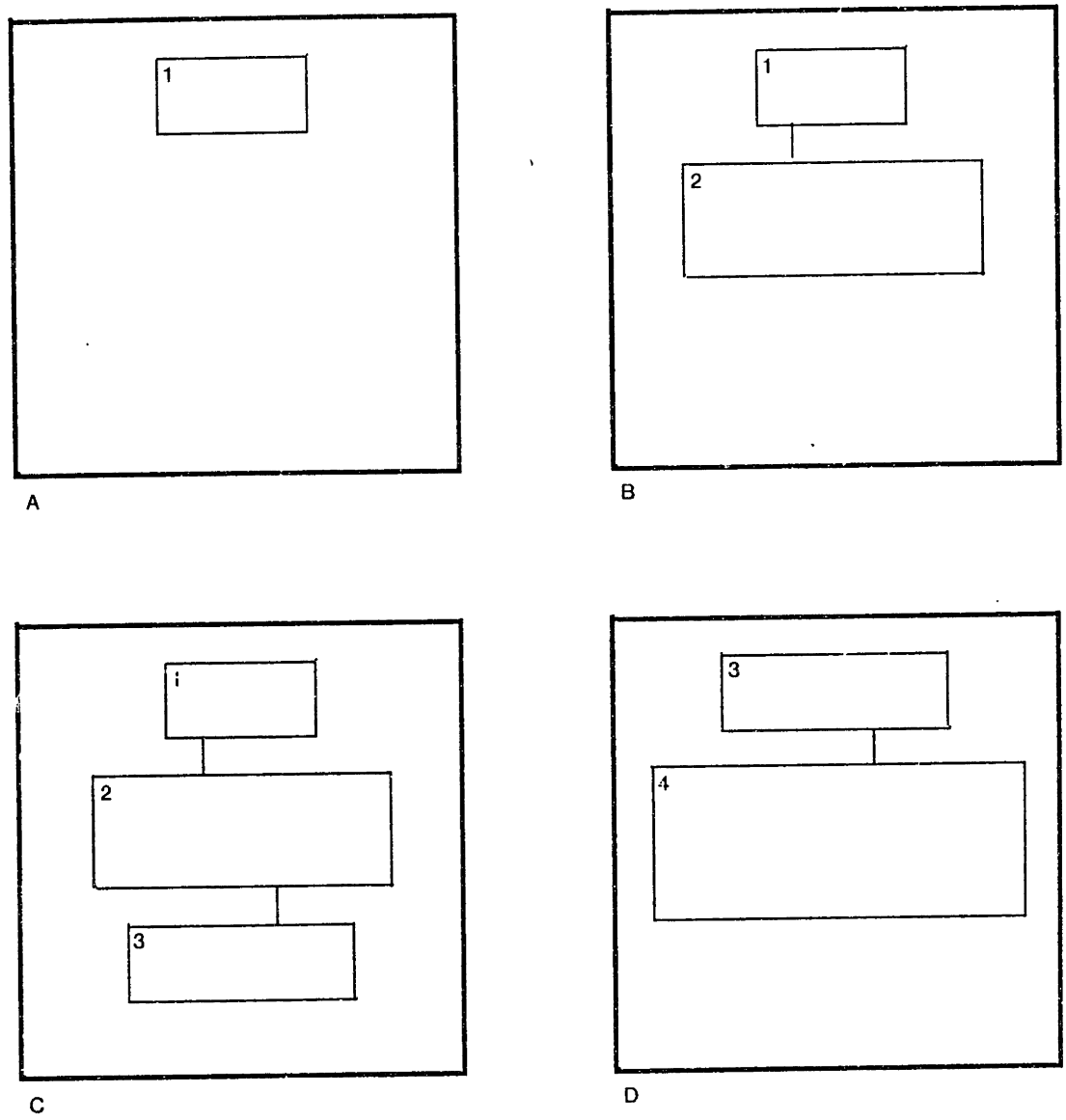


Figure 3-5: Display Management

they generate a character in a manner equivalent to a keyboard input. Any portion of text can be sensitized to generate any character. The characters generated by sonic pen hits must correspond to the compare characters in the nodes referencing the picture. Figure 3-6 shows all the linkages between the picture files and the TL program referencing them that must be filled in by the programmer.

A certain set of inputs are treated in a special way. Whenever the interpreter is wating for an operator input these commands can be entered. They are listed in Table 3-3.

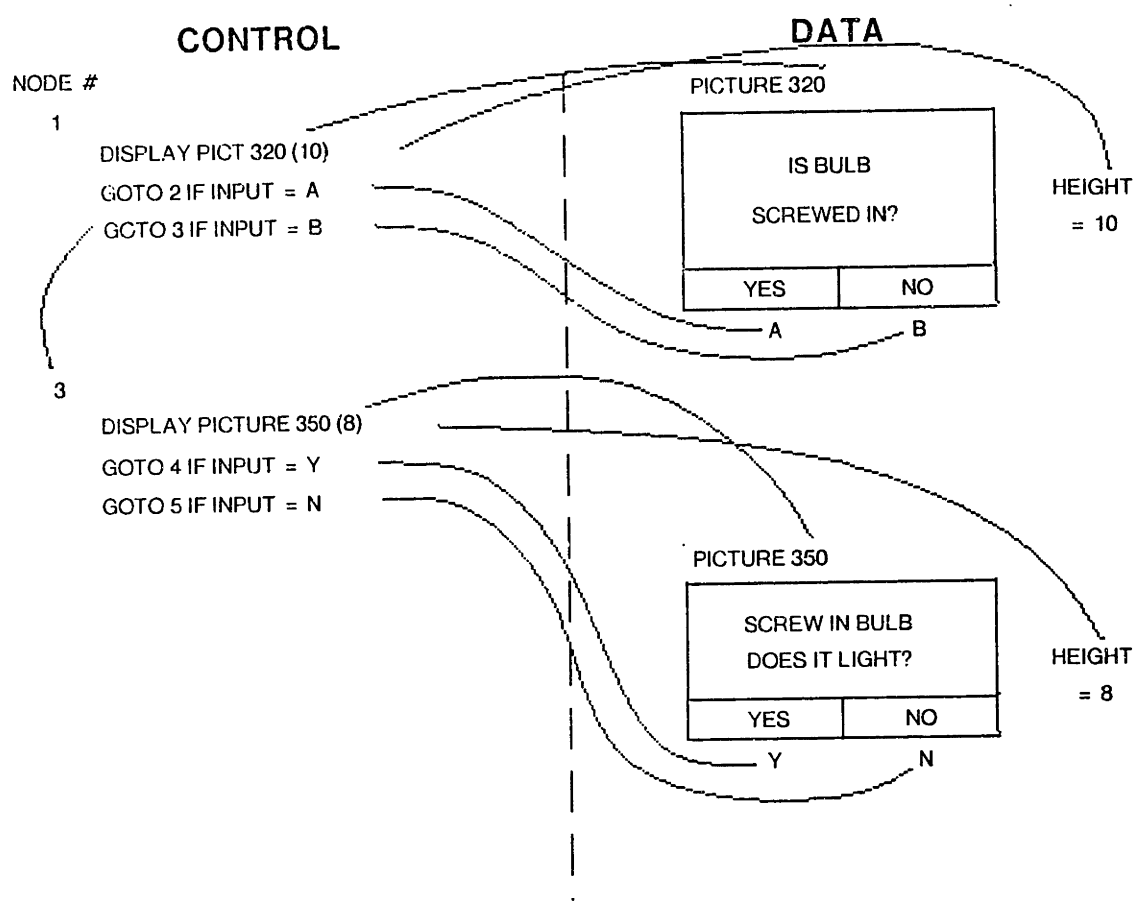


Figure 3-6: Linkages between Control and Data Structures

INPUT	OPERATION
ESCAPE	Terminate TL interpreter
NUL	Review path taken through control structure (replay function)
RETURN	Restart currently running TL program

Table 3-3: Interpreter Commands

3.2.7. Sub-graphs

Because of limitations in memory a very restricted form of sub-graph capability was implemented. When entering a sub-graph the name of return node would be pushed on a stack. The name of this node was restricted to be equal to the value of the current name plus one. This meant that all paths out of a sub-graph went to only one node, the one popped off the stack.

Some way had to be devised to transfer the results of any testing done in the sub-graph to the calling procedure. In the sub-graph a terminating state is made for all possible results of the sub-graph testing. In each of these terminating states there is an instruction to pop the control stack and obtain the next node's name. There is also a special kludge instruction which takes advantage of the GETS' ability to simulate input to its own keyboard. Thus each terminatng state would act like a unique operator input corresponding to the sub-graph's results. In figure 3-7 node Y simulates the dummy character 'D' causing node A to be entered after the subgraph is exited. After transfer was made to next node it would use the simulated input to transfer to other nodes representative of the terminating nodes in the sub-graph.

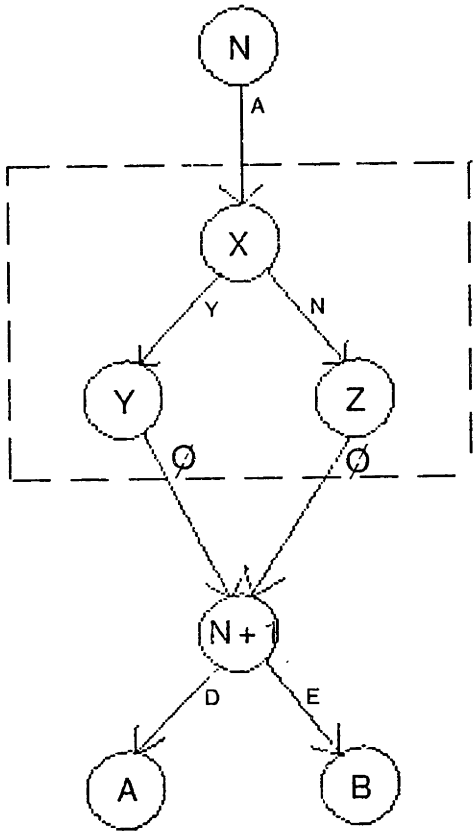


Figure 3-7: Sub-graph implementation

In Figure 3-7 node n is the calling node, node n + 1 is the return node, and nodes x, y, and z form the sub-graph. When an 'A' s received in node n the value n + 1 is pushed on the control stack and control is transferred to node x. Depending on whether the operator enters a 'Y' or an 'N' control will be transferred to node y or node z, respectively. Given that node y was selected then the next node name is popped off the stack, the character 'D' is simulated as input, and

control is transferred to node n + 1. The equivalent happens at node z with 'E' being simulated instead of 'D'. Once at node n + 1 the character 'E' or 'D' is received and used to select the next node.

Notice that transfers from the terminatng nodes to the return nodes are not driven by operator inputs as in the general case. It should be clear that with this mechanism a sub-graph can be called from multiple points within the control structure.

3.2.8. State Variables

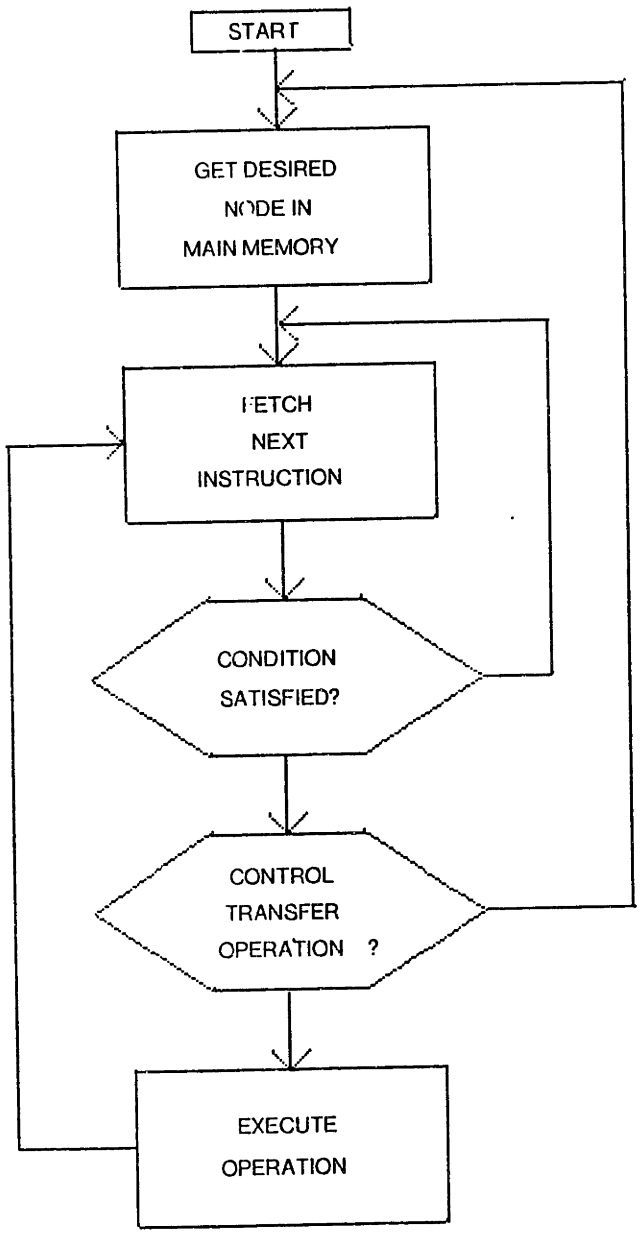
The implementation of state varial les was very limited. One was implemented. The operations on this variable were; assigning an immediate value to it, transferring control to the node named by it value, and transferring control to its value and pushing the current control name plus one on the control stack. This restricted set of variables was forced by memory and time limitations.

3.3. TL Interpreter Structure

The TL interpreter's operation can be modelled by the flow chart in Figure 3-8. A complete flow chart of the TL interpreter is given in Appendix I. The interpreter is initialized with the name of the first control node to be executed. The interpreter then gets the block containing the desired control node into the right and left stacks. Once the desired block is in the stacks a linear search for the **control delimiter - control name** pair is initiated. When the desired control name is found it is stored in the WFB as a record of program execution order. The interpreter then executes the instructions in the node. For each instruction the condition code is evaluated to determine if the operation is to be performed or not. If the instruction is not executed because a condition 130 was not satisfied then the next instruction is executed. If an instruction is not executed because a conditon 131 is not satisfied then the current input character is saved until it satisfies a code 128 or 131 or the last instruction in the node is execued. When an incorrect operator input is received it fails to satisfy any of the code 131 instructions. In this case the interpreter throws the character away and goes to the first code 131 instruction and waits for an input. If the interpreter reaches the end of a node without transferring control to another node and there are no code 131 instructions in the node then an error results.

3.4. Editor

The editor was created as an aid to writng and changing the control structures. Before it was put into use all control structures were created and altered by an independant text editting facilty. With the independent faclity all the operations needed to create the control structures were available. The problem was that the programmmer had to remember which characters



corresponded to which values and which values corresponded to which operations. For a person unfamiliar with these codes that task was virtually impossible. The editor allowed an author to create or delete nodes and create, delete, or alter individual instructions. Any field of an instruction can be altered. The complete list of editor commands is given in Table 3-4.

The feature of the editor which justified its existence was its ability to translate the fields in the instructions to english descriptions of the codes and numeric values. A sample display is given in

INPUT	OPERATION
C	Input control number and instruction number
B	Input instruction number of current control number
D	Delete current instruction
I	Insert instruction before current instruction
A	Change conditon code of current instruction
N	Change compare character of current instruction
O	Change operation code of current instruction
1	Change DATA 1 of current instruction
2	Change DATA 2 of current instruction
S	Save all changes made on current block
ESCAPE	Exit editor

Table 3-4: Editor Commands

Figure 3-9. The corresponding representation as seen with the older method is shown in Figure 3-10. When entering values for condition or operation codes the editor does not make any attempt to filter out values which have no meaning to the interpreter. On the whole, the editor was simple, easy to use, and greatly eased the programming task.

CONTROL NUMBER	CONDITION CODE	COMPARE CODE	OPERATION	DATA 1	DATA 2
1	NONE	----	DISPLAY SLD	----	10
	NONE	----	DISPLAY PICT	10	201

Figure 3-9: Editor Display Format

XRC! # BFG$ # %YGFRJ& * ~09.?";

Figure 3-10: Alternative Display Format

4. Example TL Program

In order to clarify the information presented in the previous chapters an example TL program will be developed. Details on the program's internal representation and a sample 'run' will be given. For this example a receiver has been selected as the machine to be troubleshooted. The direct links between the MT and the device it is testing will be ignored in this example, as in the rest of this thesis. Before a set of FIPs can be implemented as a TL program thay must be created or borrowed from some existing service manual. The source service manual should explicitly specify all diagnostic and repair procedures to be undertaken by the operator. The fault isolation procedures in figures 4-1 and 4-2 were created for this example. The effectiveness and accuracy of this procedure are not critical to the understanding of this example. The implementation of this procedure as a TL program is the central issue.

The fault isolation manual is represented by the control structure of figure 4-3. For the implementation of the subgraph the numbers of the calling and returning nodes must be consecutive, in this case they are 0 and 1. The rest of the node number assignments are arbitrary. 17 consecutive numbers were chosen to minimize storage requirements and reduce disk accesses.With the exception of nodes 1, 6, and 7 all nodes display textual information to the operator. Each node in figure 4-3 has the name of the picture file it references shown next to it. To better delimit each test they are enclosed by boxes. One letter inputs are shown in parentheses next to the legal responses. An operator can either type the one letter input or touch the desired box with the sonic pen to trigger the desired response. Tables 4-1, 4-2, and 4-3 show the TL program code. All of the terminating nodes are linked to the start nodes so the oprator can easily restart the program. Each table gives all the code for nodes found in one block.

NODE	COND	COMP	OPER	DATA 1	DATA 2	COMMENTS
16	129	---	133	10	14	display picture 14
	128	---	137	---	0	goto node 0 for any input
17	129	---	133	12	15	display picture 15
	128	---	137	---	0	goto node 0 for any input

Table 4-3: Nodes 16 - 17

The next sequence of figures shows the display as the operator proceeds through the program. Each picture has its number and the control node number shown next to it. The TL interpreter

Figure 4-1: Radio Manual

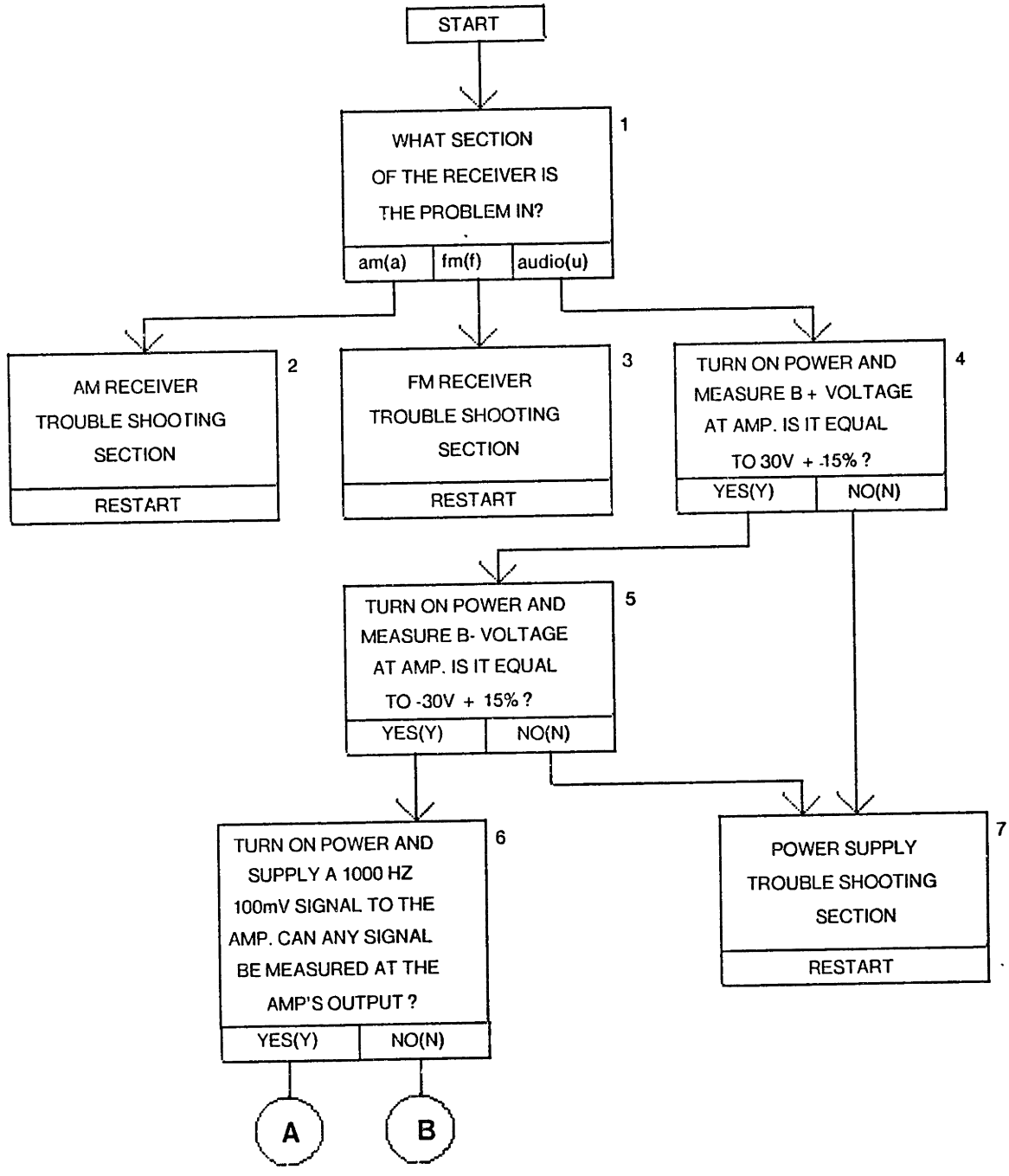


Figure 4-2: Radio Manual (Continued)

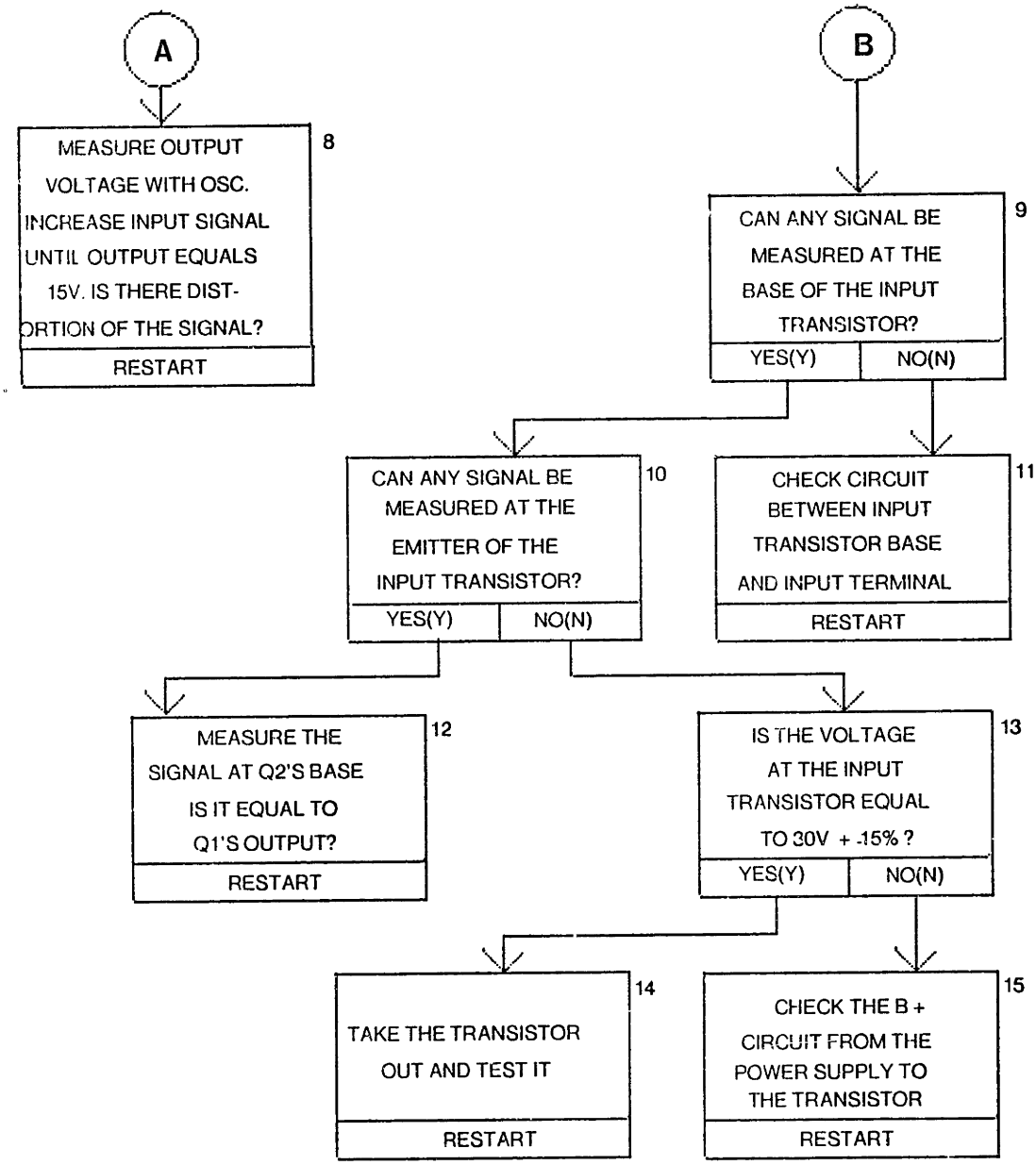
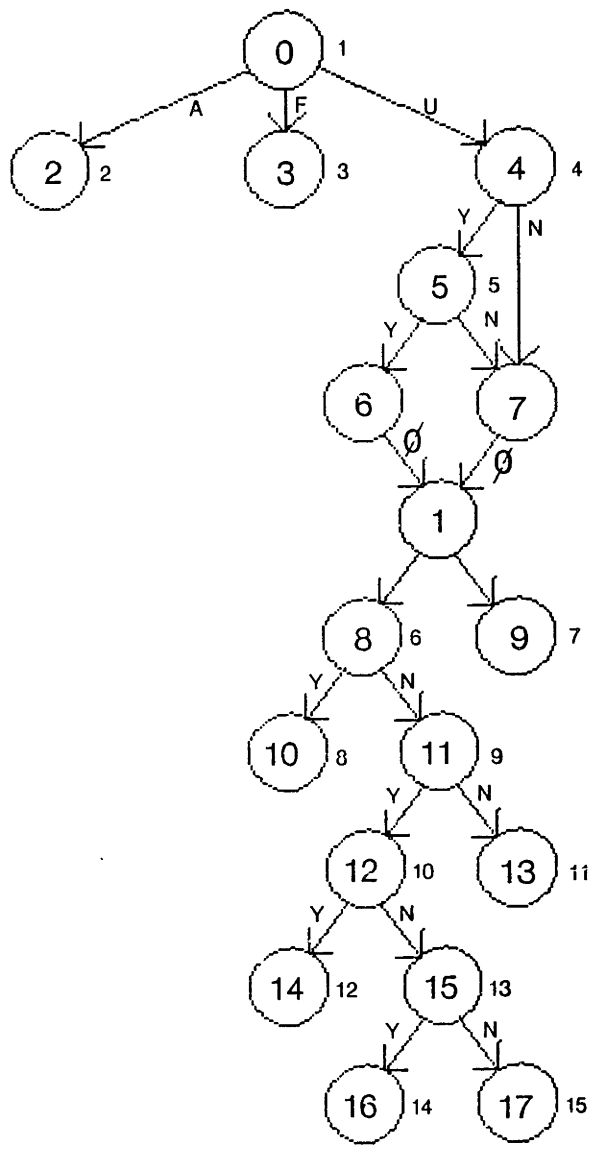


Figure 4-3: Control Structure



NODE	COND	COMP	OPER	DATA 1	DATA 2	COMMENTS
0	129	---	139	---	---	clear screen
	129	---	133	8	1	display picture 1
	131	A	137	---	2	goto node 2 if A is input
	131	F	137	---	3	goto node 3 if F is input
	131	U	136	---	4	call sub-graph starting at node 4
1	131	P	137	---	8	goto node 8 if P is input
	131	F	137	---	9	goto node 9 if F is input
2	129	---	133	8	2	display picture 2
	128	---	137	---	0	goto node 0 for any input
3	129	---	133	8	3	display picture 3
	129	--	137	---	0	goto node 0 for any input
4	129	---	133	14	4	display picture 4
	131	Y	137	---	5	goto node 5 if Y is input
	131	N	137	---	7	goto node 7 if N is input
5	129	---	133	12	5	display picture 5
	131	Y	137	---	6	goto node 6 if Y is input
	131	N	137	---	7	goto node 7 if N is input
6	129	---	143	---	(P)	simulate input of 'P' then return to node on stack
7	129	---	143	---	(F)	simulate input of 'F' then return to node on stack

Table 4-1: Nodes 0 - 7

starts executing at a predefined node , in this case node 0. Execution of node 0 causes the screen to be cleared and picture 1 to be displayed. If the oprator types an 'M' or touches the box labelled 'AUDIO (U)' control will be transferred to node 4. As soon as the input is processed an arrow appears below the 'AMP (M)' box. When node 4 is entered picture 4 will be displayed below picture 1. Entry into the sub-graph is completely transparent to the operator. Responding 'Y' to picture 4 causes node 5 to be entered and picture 5 to be displayed. At this point the screen appears exactly as in figure 4-4. Assuming the operator answers 'Y' to node 5 several things occur. First, control is transferred to node 6 which does not display a picture. Node 6 generates a 'P' (PASS) input, pops the return node number off the control stack , equal to 1, and transfers control to it. Node 1 does not alter the display. It receives the 'P' immediately and transfers control to node 8. When node 8 attempts to display picture 6 the interpreter realizes it will not fit

NODE	COND	COMP	OPER	DATA 1	DATA 2	COMMENTS
8	129	---	133	16	6	display picture 6
	131	Y	137	---	10	goto node 10 if Y is input
	131	N	137	---	11	goto node 11 if N is input
9	129	---	133	8	7	display picture 7
	128	---	137	---	0	goto node 0 for any input
10	129	- -	133	16	8	display picture 8
	128	---	137	---	0	goto node 0 for any input
11	129	---	133	10	9	display picture 9
	131	Y	137	---	12	goto node 12 if Y is input
	131	N	137	---	13	goto node 13 if N is input
12	129	---	133	10	10	display picture 10
	131	Y	137	---	14	goto node 14 if Y is input
	131	N	137	---	15	goto node 15 if N is input
13	129	---	133	10	11	display picture 11
	128	---	137	---	0	goto node 0 for any input
14	129	---	133	12	12	display picture 12
	128	---	137	---	0	goto node 0 for any input
15	129	---	133	10	13	display picture 13
	131	Y	137	---	16	goto node 16 if Y is input
	131	N	137	---	17	goto node 17 if N is input

Table 4-2: Nodes 8 - 15

on the screen. When this happens the screen is cleared, picture 5 is redisplayed and picture 6 is displayed below it. The screen now appears as figure 4-5. If the operator answers 'NO' in node 11 he will be transferred to node 13 and the screen will appear as in figure 4-6. At this point the end of one branch of the fault isolation procedure has been reached. Any input will cause control to be transferred to node 0.

Several capabilities of the TL interpreter that were not shown in the example should be mentioned. Though no diagrams were shown they could be used with any of the text. The pictures also need not follow the general format of those in the example. They may be any size or shape. Slides can also be used at any time by themselves or with text.

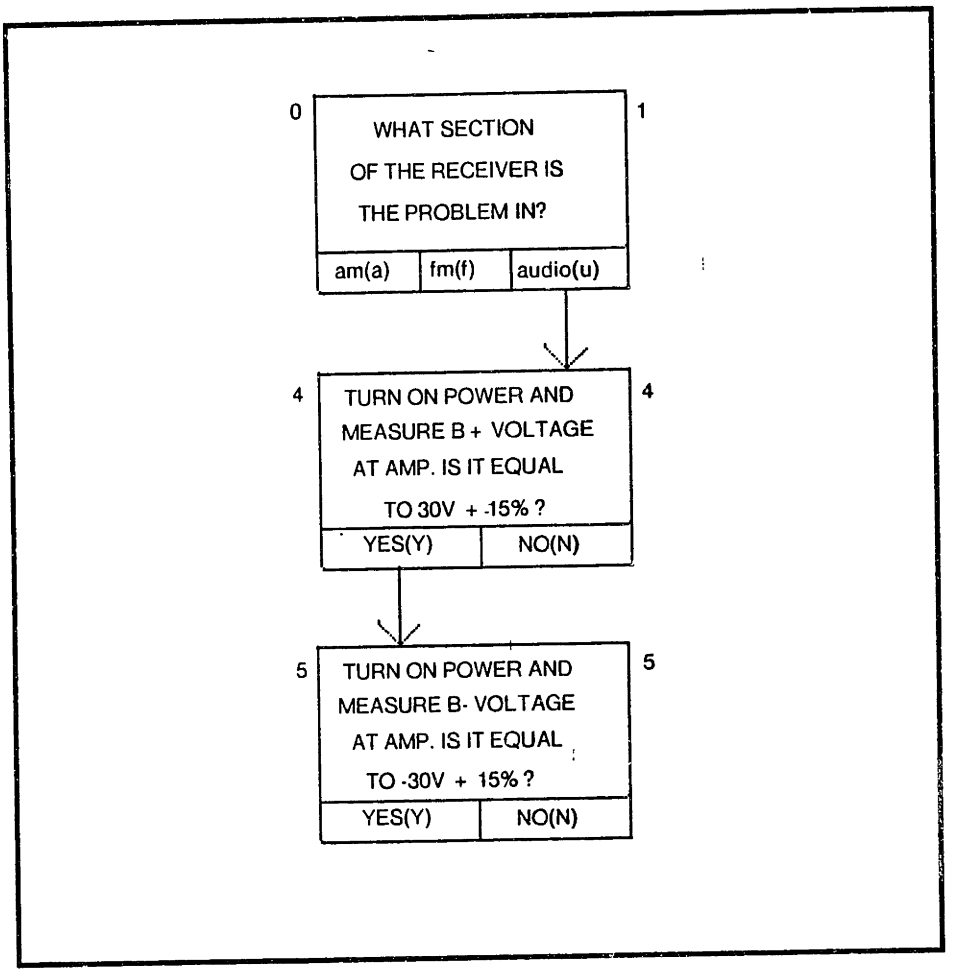


Figure 4-4: Screen 1

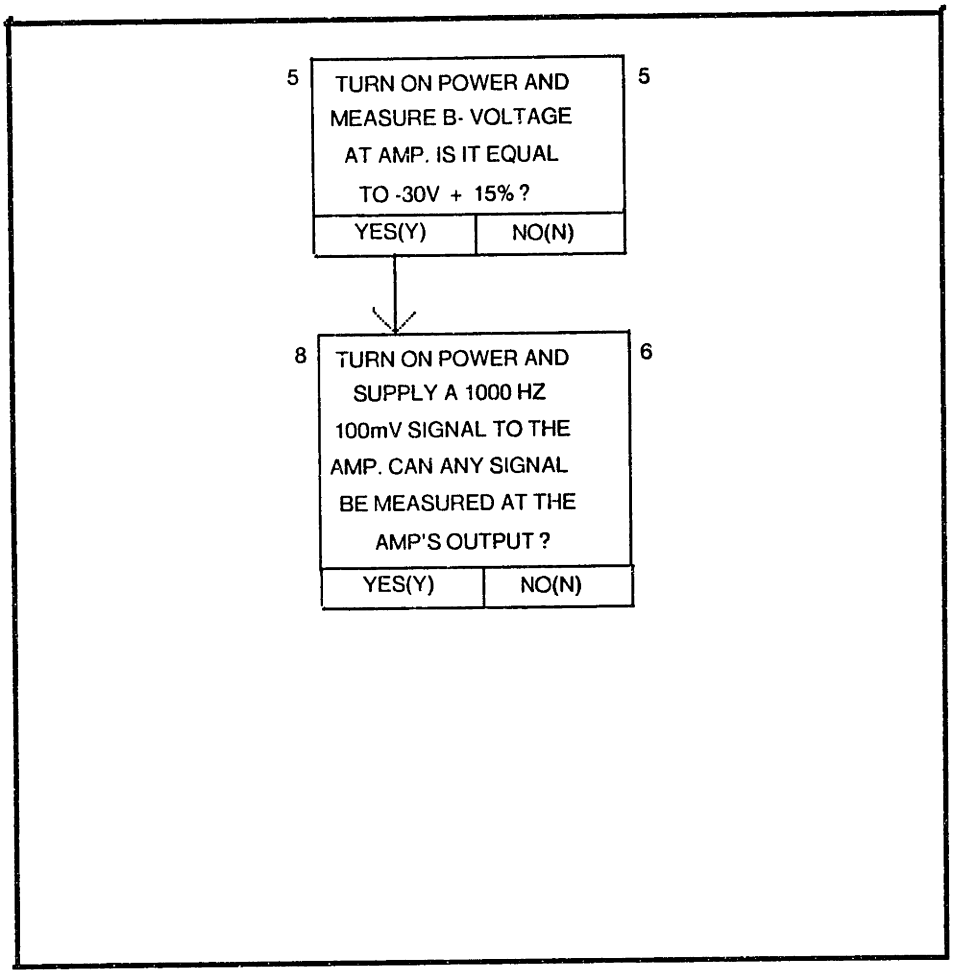


Figure 4-5: Screen 2

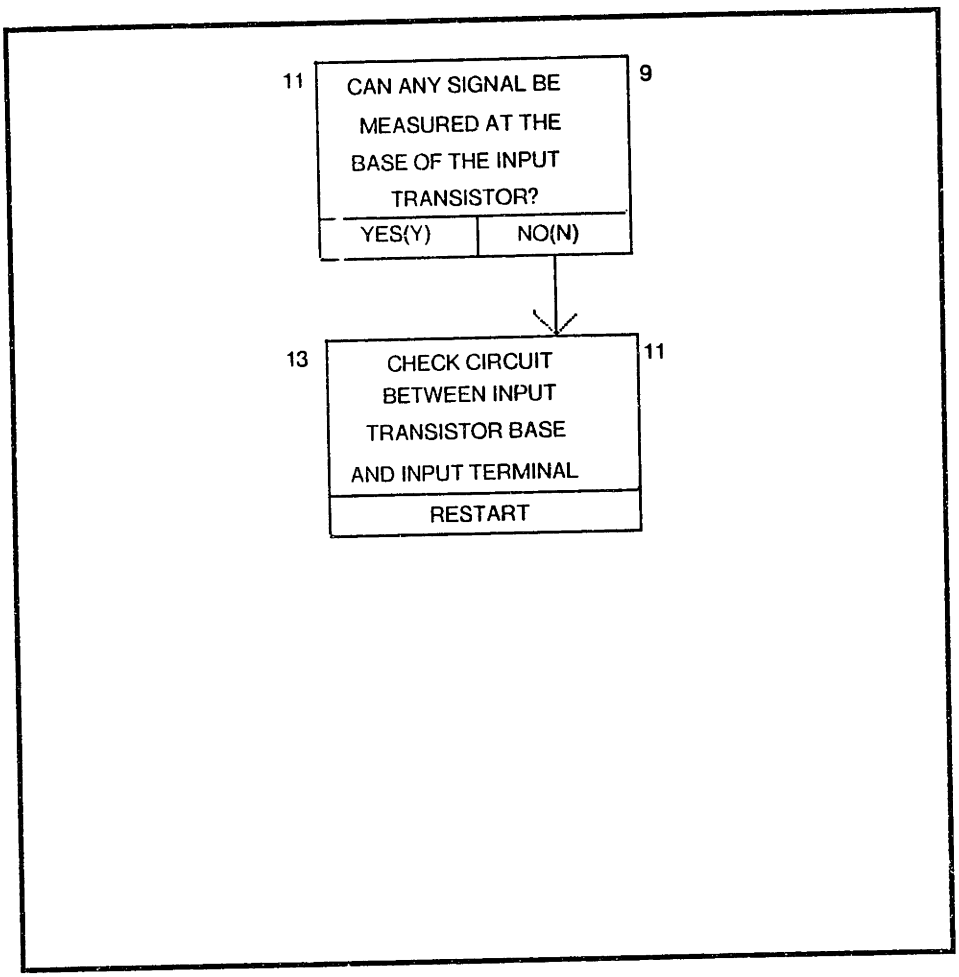


Figure 4-6: Screen 3

5. Results and Conclusions

Based on the work described in this thesis several conclusions can be drawn about the attributes which would be desireable in a maintenance terminal. In many cases it was found desireable to be able to review all the tests and their results from the first test to the most recently executed test. The replay function performed this task. Two versions of this function should be provided, one would allow the operator to only inspect the path taken through the tree, the other would allow the operator to stop inspection and take a previously unexecuted branch at any point. The inspect only replay function is useful when accidental alterations of the test sequence are undesireable. The alterable replay function can be useful in the just the opposite situations, the operator may want to perform tests that would not normally be executed. The inspect only replay function can be implemented by keeping a record of the tests executed during a run and then executing only the display instructions from these tests when the replay function is applied. This type of replay function was implemented on the GETS. The alterable replay function can be implemented by recording the tests and their results. When the function is called the maintenece terminal and the computer under test are initialized, then the tests and their results are applied automatically until the operator intervenes and enters his own results. This function would be easier to implement than the general back-up function and it would be almost as useful.

Some conclusions were reached on the general features of the user interface. The sonic pen was quick and easy to use but sometimes unreliable. In some cases it would take an operator several tries to hit a sensitized area because of misuse of the pen and electronic misadjustments. In case of such a difficulty the proper response characters were displayed on the screen so the operator could type them in. Keyboard entry is slower but more reliable. In order to show the operator that the termianl has correctly interpreted his response some visual feedback is necessary. In this project the feedback took the form of an arrow from the selected response to the resulting next test. For displaying a large quantity of information a color CRT would be preferable to the plasma panel; the resolution can reach 1024 x 1024, color can represent more information, and video mass storage techniques can be used for the test data base. Diplaying a maximum amount of information on the screen can be useful to an operator. Video disk and video tape technology can be used to store the pictures, diagrams, and text making up the test picture files.

In addition to the interpreter a structured editor is needed. Both a facility for making quick small changes and a facility for entering large quantities of data into the control and data structures are needed. A data base of picture files and a facility for entering, altering, and

deleting these files would be needed. The picture should be addressable by name and they should contain information on their size and responses. The acceptable responses displayed in a given picture are needed when the control structure is created and linked together. Another editor for the control structure would be necessary. This editor would translate from the underlying representation to a readable format. This editor should provide functions for linking picture files, responses, and next control nodes. The two editors just described can be resident in tthe maintenance terminal or they could reside in another computer. Much work on the configuration of the data bases and their editors needs to be done.

From the programs that were developed some general conclusions can be drawn about the software and hardware requirements of the maintenance terminal. A very simple instruction set appears to be adequate for implementing the maintenence terminal's functions but the simulated terminal was not required to interact with a computer under test. A more complete simulation with a computer would seem to be necessary. The general approach of modelling the control structures as finite state machines could be transferred to a microcoded sequencing approch. The terminal's functions could be implemented in bit slices for speed. Such an approach should be considered for further research.

Bibliography

[1]
Bannister, R. J.
ORTS-A Shipboard Automatic Test System.
In *Autotestcon*. IEEE, 1977.

[2]
Bergen, J. K.
A User Oriented Man/Machine Interface.
In *Automatic Support Systems*. IEEE, 1974.

[3]
Brown, J. S.; Burton, R. R.; Bell, A. G.
SOPHIE: A Sophisticated Instructional Environment for Teaching Electronic Troubleshooting.
Technical Report 2970, Bolt, Beranek, and Newman, 1974.

[4]
Bulat, M. H. and Francis, J. E.
Interactive Maintenance Terminal Fault Isolation Concept Demonstration.
Technical Information Series 79-POD-3, GE, 1979.

[5]
Colgan, J.
Automated Operator Manual for Automatic Test Systems.
In *Autotestcon*. IEEE, 1977.

[6]
deKleer, J.
Steps Toward a Theoretical Foundation for Complex Knowledge Based CAI.
Technical Report, Bolt, Beranek, and Newman, 1975.

[7]
Goldstein, I. P.
Understanding Simple Picture Programs.
AI Lab Tech Report 294, MIT, 1974.

[8]
Kaiser, G. E.
Automatic Extension of an Augmented Transition Network Grammar for Morse Code Conversations.
LCS Tech Report 233, MIT, 1980.

[9]
Rowe, G. L.
Autotest User Needs at a Base Shop.
In *Automatic Support Systems*. IEEE, 1974.

[10]

Rupp, C. R.
A Stand-alone CAI System Based on Procedural Grammars.
International Learning Technology Symposium/Exposition Washington D.C., GE, 1976.

[11]

Scully, J. K.
The Harmonization of Prime Equipment BITE with ATE.
In *Autotestcon*. IEEE, 1977.

[12]

Shortliffe, E. H.
MYCIN: A Rule-based Compu er Program for Advising Physicians Regarding Antimicrobial Therapy Selection.
AI Lab Memo 251, Stanford, 1974.

[13]

Sussman, G. J. and Brown, A. L.
Localization of Failures in Radio Circuits A Study in Causal and Teleological Reasoning.
AI Lab Memo 319, MIT, Dec., 1974.

[14]

Sussman, G. J. and Stallman, R. M.
Heuristic Techniques in Computer Aided Circuit Analysis.
In *Transactions on Circuits and Systems*, pages 857-865. IEEE, 1975.
CAS-22

[15]

Woods, W. A.
Transition Network Grammars for Natural Language Analysis.
In *Communications of the ACM*, pages 591-606. ACM, 1970.
Volume 13, Number 10

I. TL Interpreter Flowchart

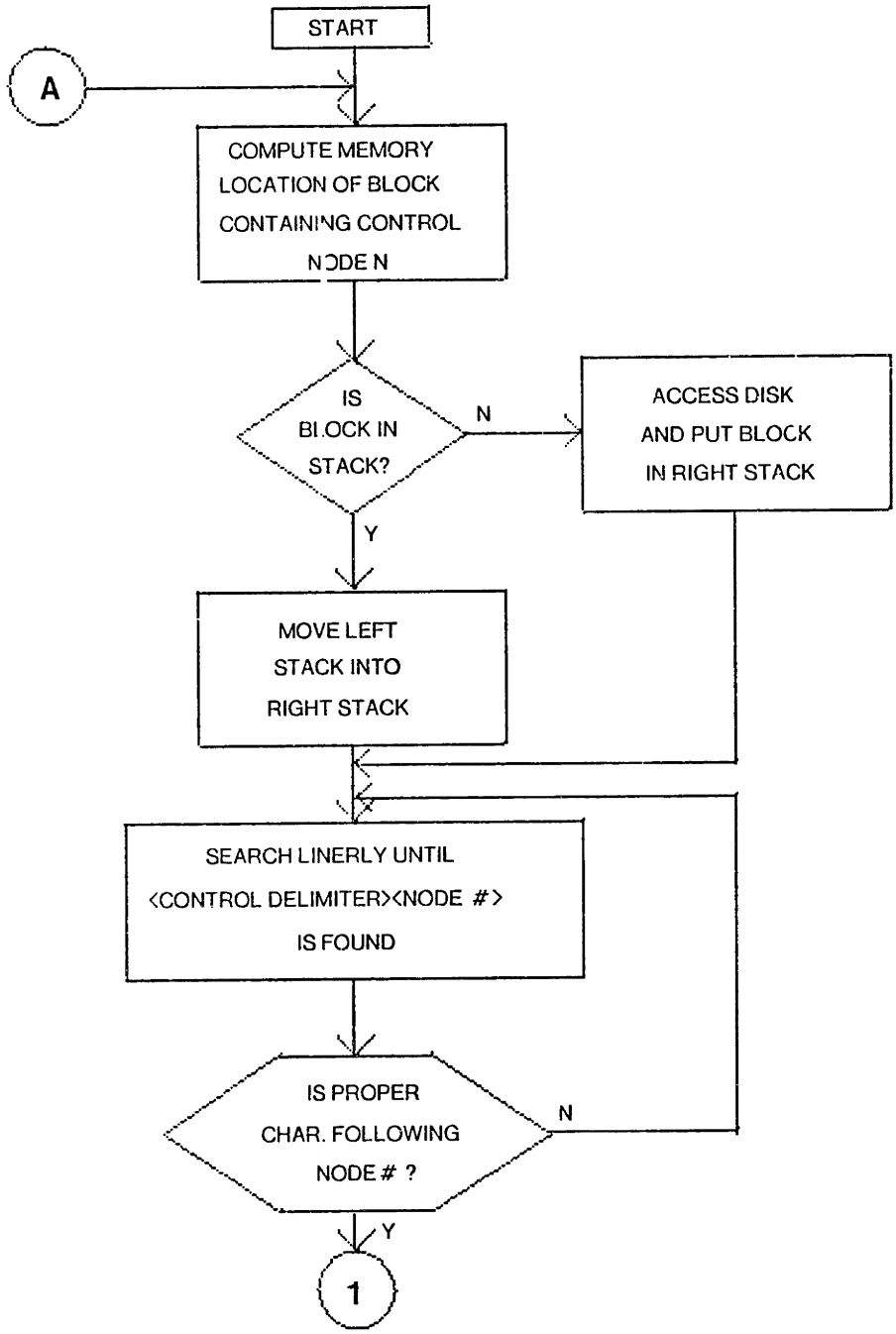


Figure 5-1: Flowchart (page 1)

Figure 5-2: Flowchart (page 2)

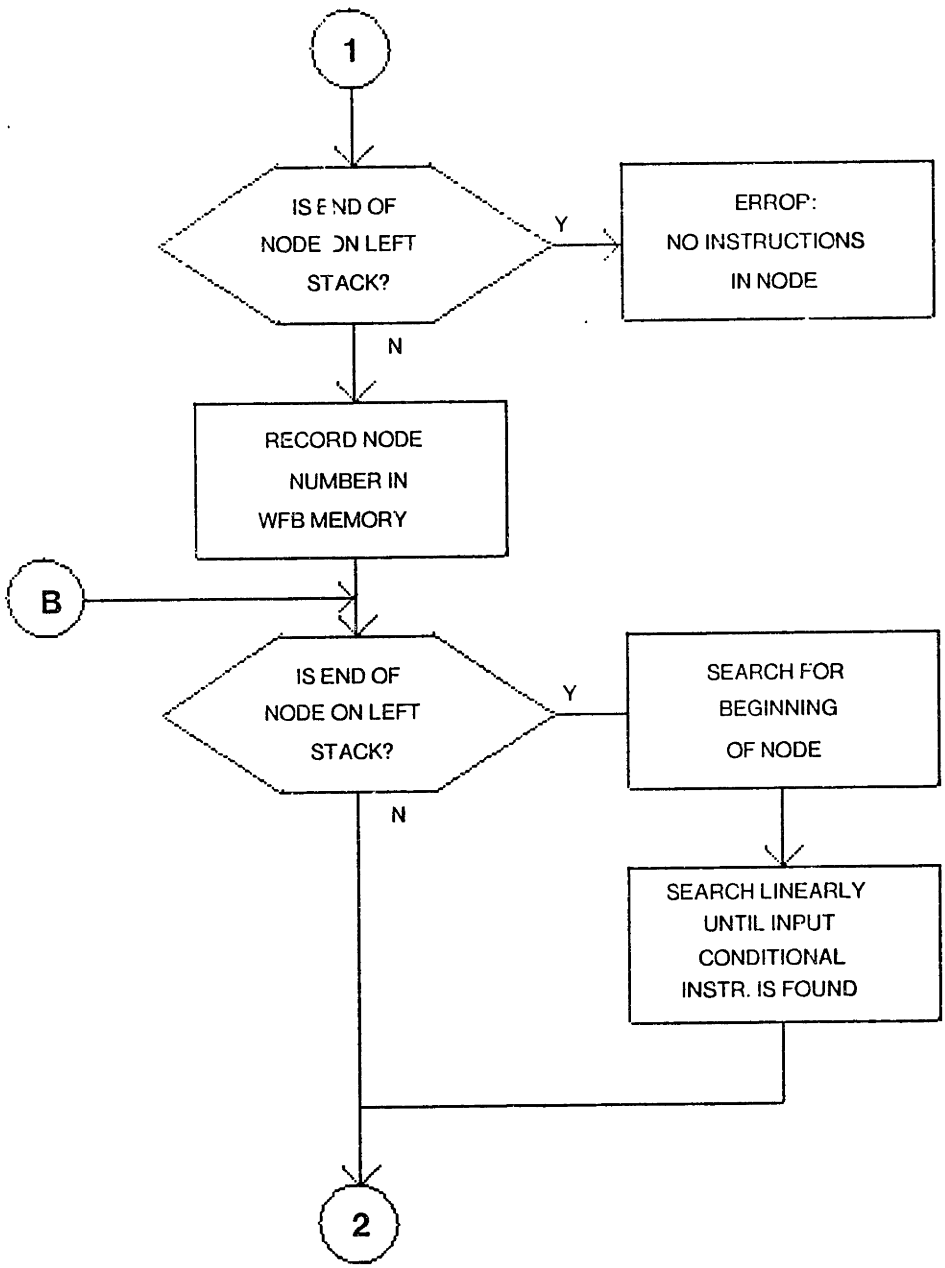


Figure 5-3: Flowchart (page 3)

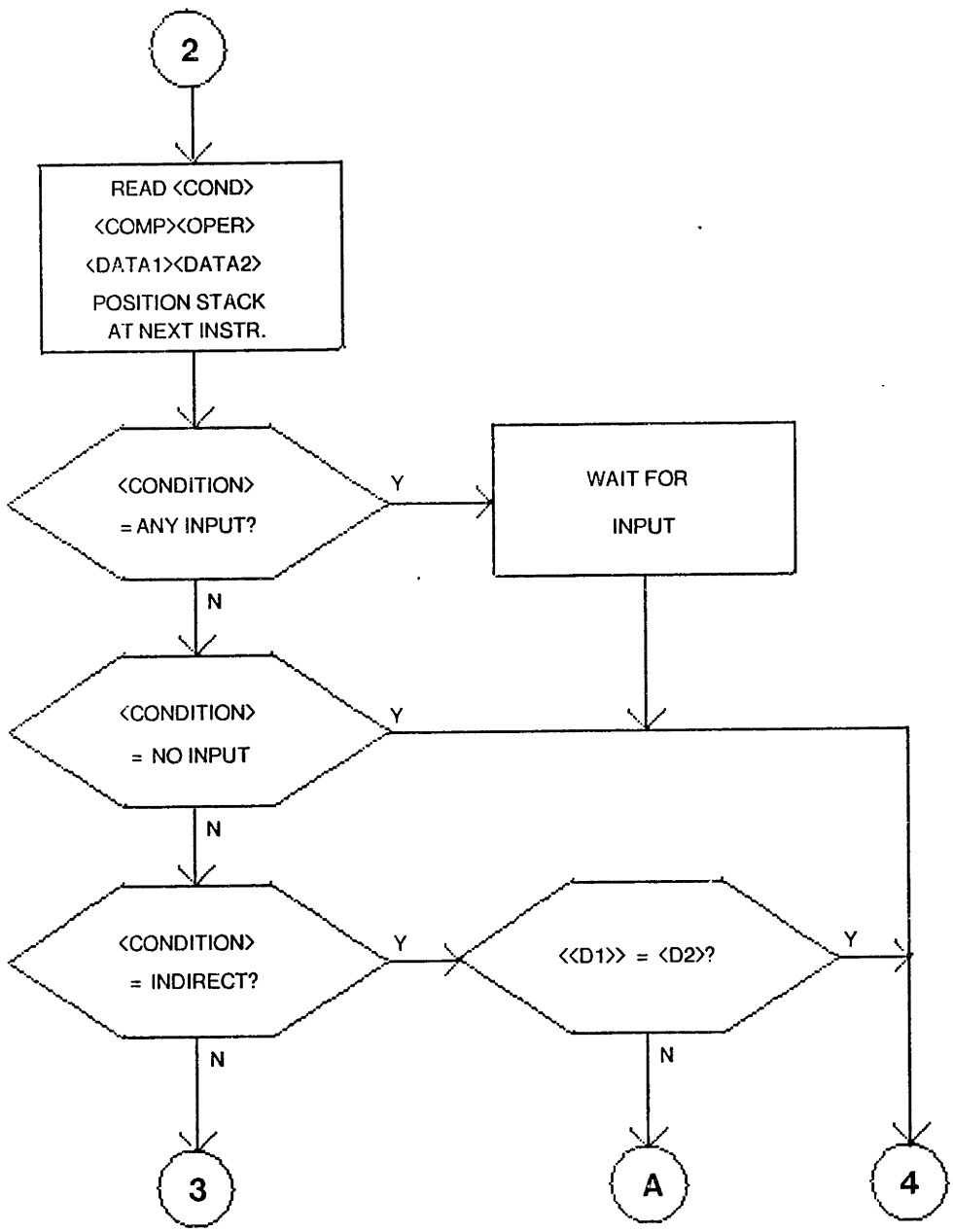


Figure 5-4: Flowchart (page 4)

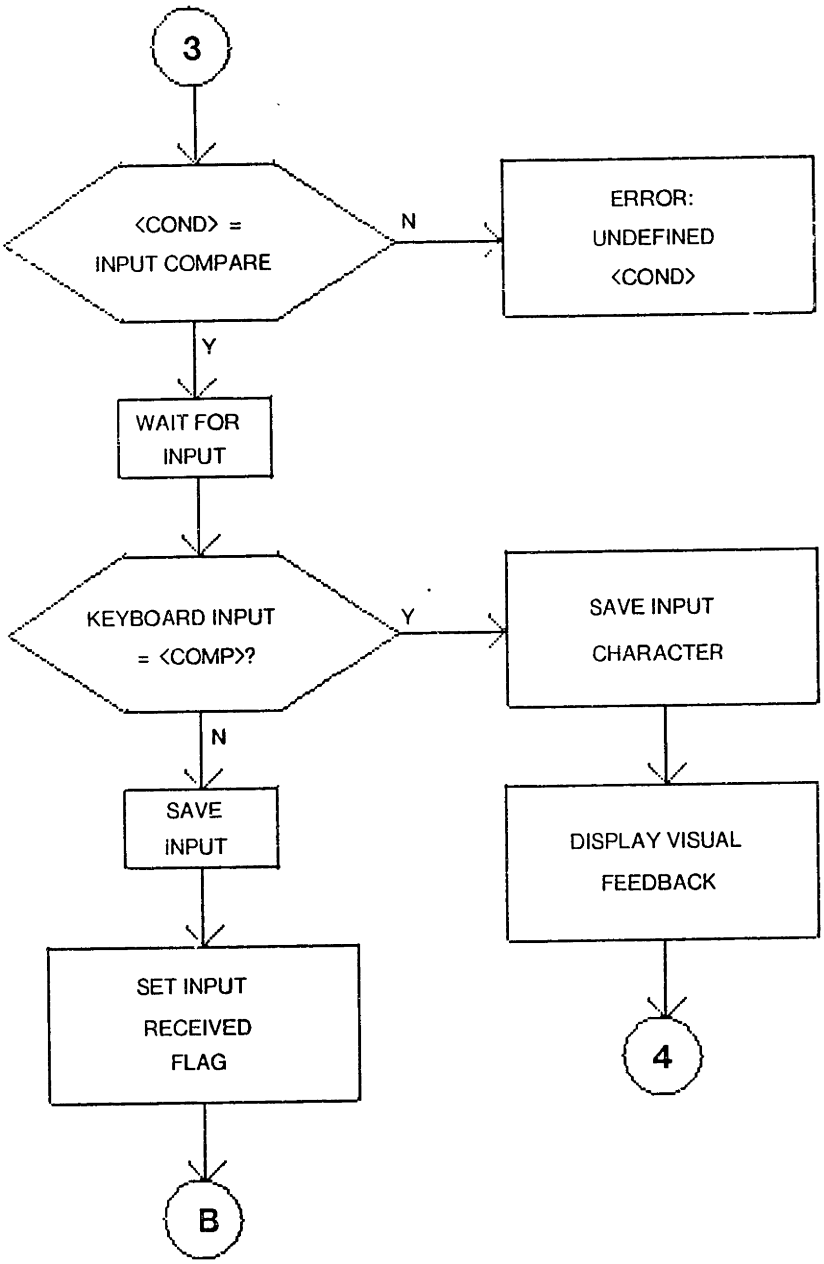


Figure 5-5: Flowchart (page 5)

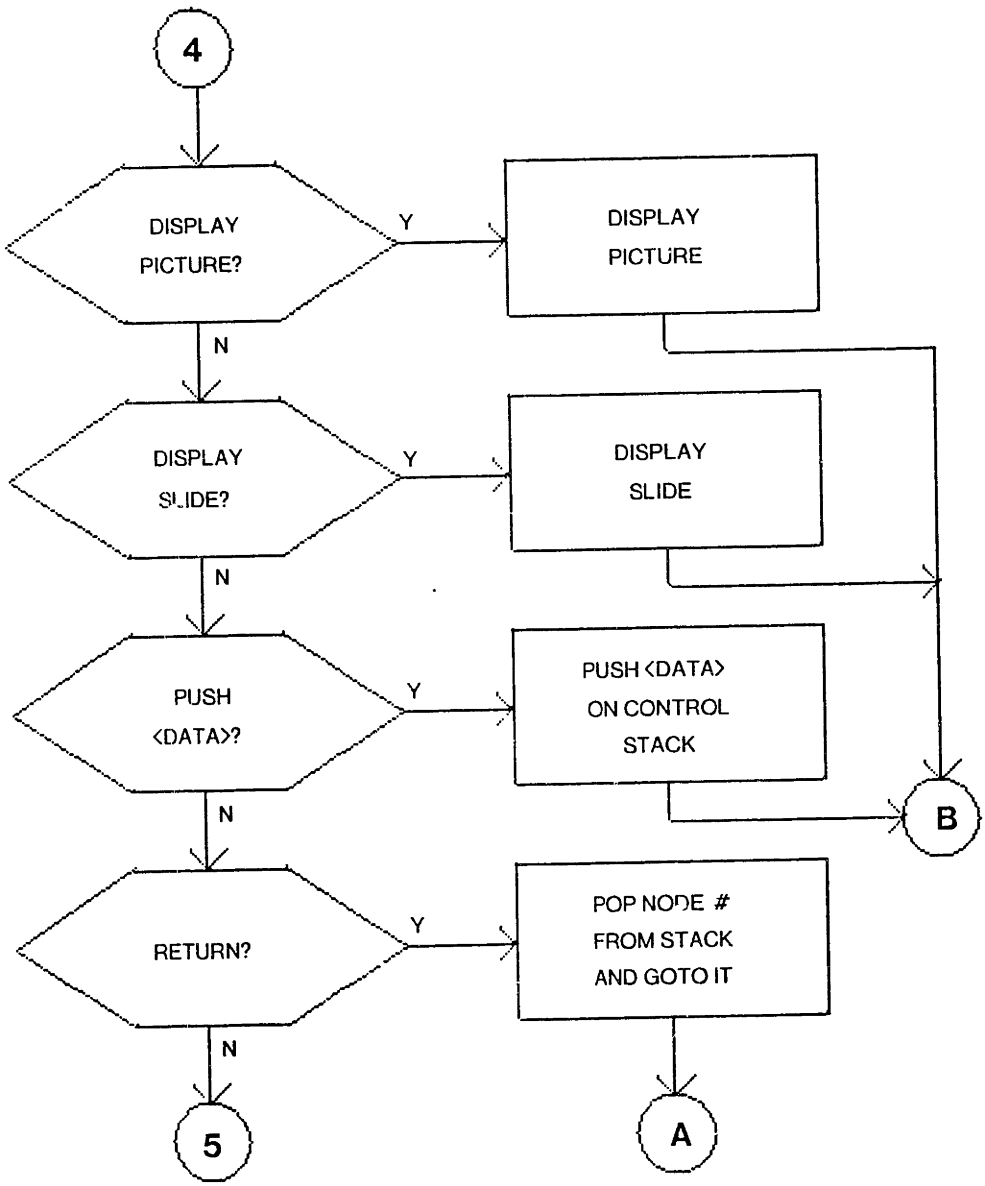


Figure 5-6: Flowchart (page 6)

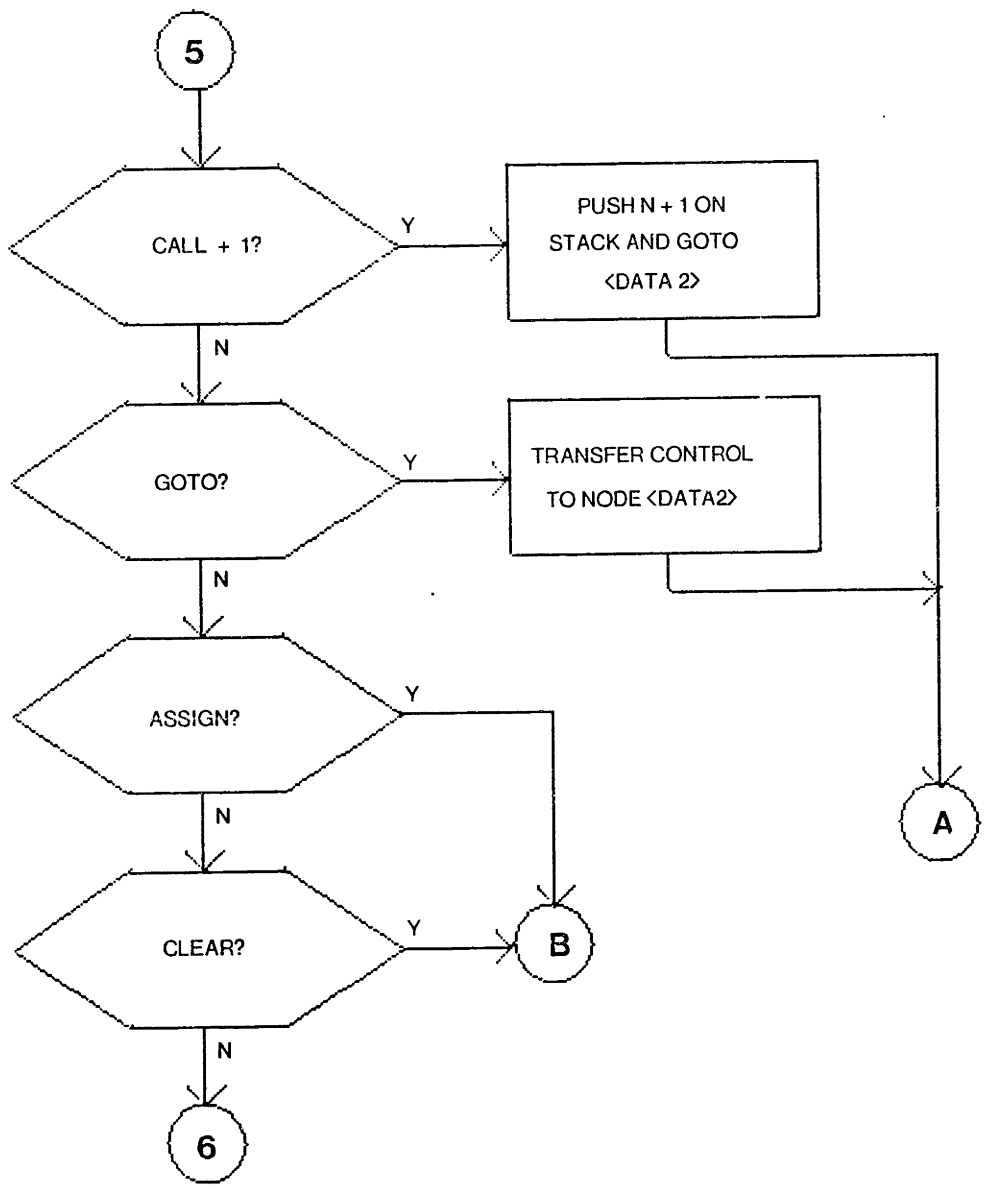


Figure 5-7: Flowchart (page 7)