To the Graduate Council:

I am submitting herewith a thesis written by Sameer Dinesh Shah entitled "Algorithms for RF Sensitivity and Airplane Flutter Tests on Television Receivers." I have examined the final copy of the thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.

Daniel B. Koch

D. B. Koch, Major Professor

We have read this thesis
and recommend its acceptance:

Accepted for the Council:

Associate Vice Chancellor
and Dean of the Graduate School

Algorithms for RF Sensitivity and Airplane Flutter Tests on Television Receivers

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Sameer Dinesh Shah

August 1995

DEDICATION

I dedicate this work to my God above and to my wonderful family. My faith has always given me strength and has helped me in all aspects of my life. My wife Kalpa, my parents, Dinesh and Nirmala, and my sister, Avnee, have supported me throughout and for that I express my deepest affection and love.

ACKNOWLEDGEMENTS

I wish to express my heartfelt gratitude and thanks to Dr. Daniel B. Koch. I certainly could not have completed this work without his support. He was so patient and understanding. I also wish to thank Dr. Herbert P. Neff and Dr. M. J. Roberts for serving on my thesis committee. Finally, I wish to acknowledge my team members during this effort: Gary Ragsdale, Eric Wrushen, Keeratpal Singh, Qin Wan, and Baishali Raychaudhuri.

ABSTRACT

This study was conducted in the area of automated testing. The purpose was to develop algorithms to perform automated RF Sensitivity and Airplane Flutter tests on NTSC televisions. In addition, an instrument driver for the HP 11759D Dynamic Ghost Simulator was developed. All of these programs were created using LabVIEW.

The RF Sensitivity test algorithm was developed using a methodology known as Cleanroom Software Engineering (CSE). The Airplane Flutter test algorithm was developed using an adapted form of CSE (CSE-A) for graphical programming. The Ghost Simulator driver was developed using design techniques invented by National Instruments.

Compared to manual procedures, the implementation of these algorithms resulted in considerable savings in time and increase in productivity. In addition, data collection and logging was done more efficiently and accurately than in the manual processes.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Chapter 1

Introduction

1.1 Background

The television industry has evolved incredibly over the past fifty years. However, some facets of it have resisted change fairly well. One of these is the area of testing. Testing refers to performance comparisons between units made by various manufacturers. In addition, it also refers to experimentation done on circuit boards ranging from prototypes to market-ready devices. Examples of such tests include measuring a television's sensitivity to Radio Frequency (RF) interference and determining a television's Flutter Frequency. For many years, the practice has been to perform such tests manually.

Continuing with this example of RF Sensitivity, the problems with a manual test environment can be brought to light through an example. In order to perform such a test, several things must be done. First, the test equipment must be configured to the required settings and the televisions must be prepared for testing. Second, the necessary audio and video signals must be routed to the televisions under test. Third, during testing, the sound is turned on and off, the channels are changed, and the appropriate audio and video

signals are routed to the desired channels. Then, at each stage the sensitivity levels are manually recorded. When the test is finished, all of the hardware must be reset so that it is in a known state for the next use. Finally, the data must be entered into a database so that it can be analyzed [1]. A similar procedure is required to determine the Flutter Frequency [2].

Reviewing this procedure makes its associated problems obvious. An extreme amount of time is consumed in having to manually configure equipment, route signals, and record data. So, labor costs and the number of errors are raised while the efficiency of the tester is lowered.

Because of these problems and shortcomings, the company's designs can lag behind its competitors' designs. Consequently, a competitive place in the market may be lost. Then, the company may attempt to make up this loss by trying to shorten the design cycle. Thus, the engineering staff will have more pressure placed on it and it will be required to increase its efficiency when in fact the problem is rooted elsewhere.

1.2 Purpose of the Project

A logical solution to the above problem was proposed in this project. The purpose of this project was to develop an automated method for performing RF Sensitivity and Airplane Flutter tests on multiple televisions. In addition, another objective of the project was to develop an instrument driver for a piece of test equipment known as the Hewlett-Packard 11759D Dynamic Ghost Simulator.

The goal was achieved by using a National Instruments' package called LabVIEW. This software (discussed in Chapters 3-6) was chosen to automate the process of making RF Sensitivity and Flutter Frequency measurements and to reduce the number of actions the technician had to take. Thus, he was allowed to pay more attention to making these measurements rather than preparing for them.

An overall block diagram of the system is given in Figure 1.1 [3]. The LabVIEW code which controls the testing is loaded on the system controller. Commands are sent to the appropriate test equipment by the system controller, via a GPIB (or RS232) connection. Error checking is also done by the controller to ensure that the equipment completes the task without any problems. Drivers allow the controller to talk to the equipment.

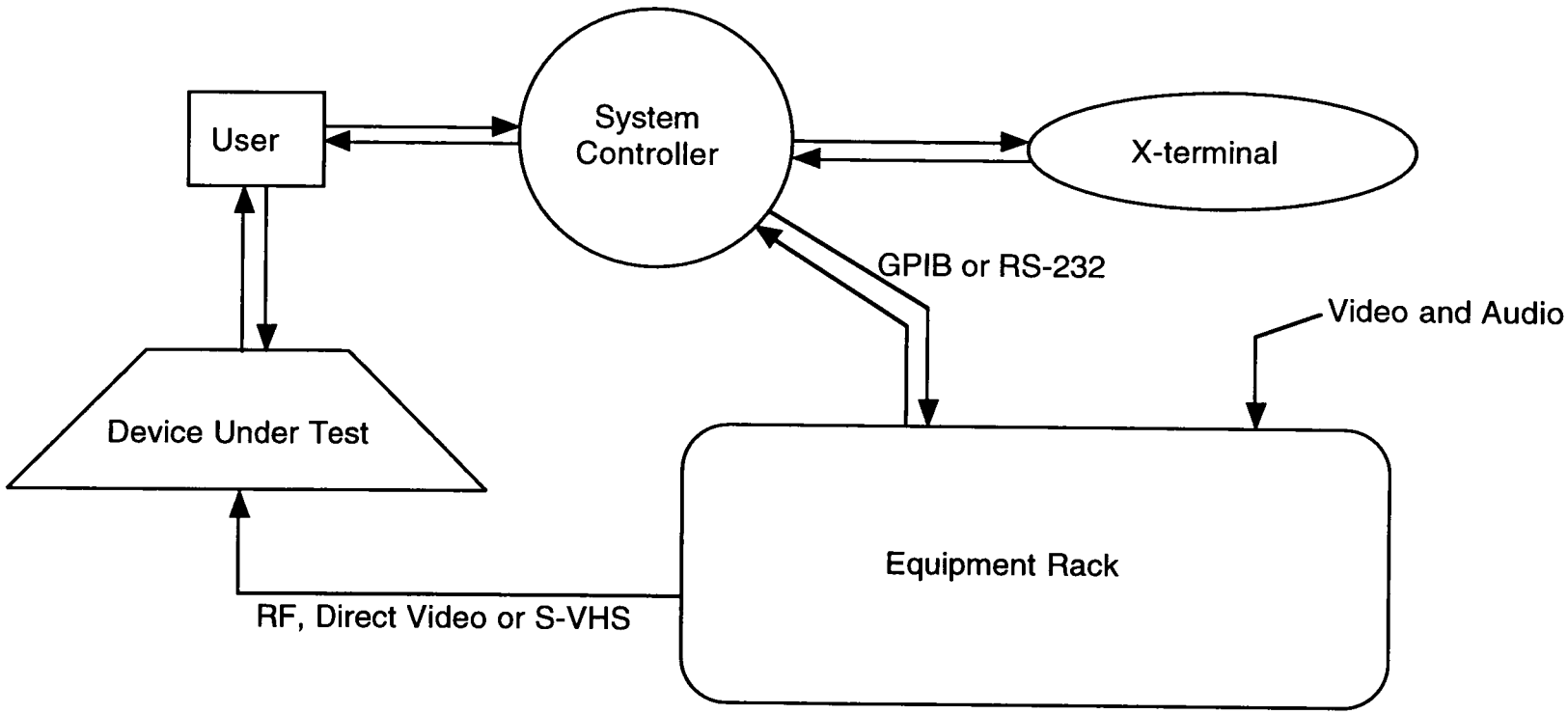


Fig. 1.1 Block diagram of the testing environment.

The user interacts with the software by means of a graphical interface. Parameters such as signal sources, attenuation levels, and the television under test can be selected. Then, the data are logged automatically to a tab-delimited file which can be read by a variety of applications.

It should be noted that the entire system--known as the Automated Test Facility--was created as a team effort. The author's portion of the work was to develop the automated RF Sensitivity test, the automated Airplane Flutter test, and the HP Ghost Simulator driver. Other drivers and tests were written by various members of the team. Therefore, all drivers needed to perform RF Sensitivity and Airplane Flutter testing will be referenced later in the work (see Chapters 4 and 5).

One other aspect of this work must be mentioned. The software was developed using two different ideologies: a modified version of a methodology known as Cleanroom Software Engineering (CSE) and an adaptation of CSE for graphical programming (CSE-A). This concept, although not entirely new to the engineering community, was only recently introduced to software engineering. Code was developed as quickly as possible and then adequate time was allotted for debugging. Consequently, low quality software is produced which will result in a dissatisfied customer. In Cleanroom

Software Engineering, soft-ware has to be developed in a much more rigorous fashion with a great deal of customer interaction. These procedures are discussed further in Chapter 3.

1.3 Remaining Chapters

A theoretical foundation for RF Sensitivity and Airplane Flutter testing are given in Chapter 2. A television receiver susceptibility to RF interference is affected by three main parts . These are: the tuner, the IF chain, and the Automatic Gain Control (AGC). The Flutter Frequency is affected most by the AGC. In addition, a theoretical background for the Ghost Simulator is given.

The Cleanroom Software Engineering process is overviewed in Chapter 3. This incorporates a discussion of how this process was modified to suit the needs of the present project. The adapted version of CSE for graphical programming is also discussed. The design methods employed in developing the Ghost Simulator driver are discussed in the last section.

Chapters 4 and 5 are dedicated entirely to the automated RF Sensitivity and Airplane Flutter test. A broader background into the testing procedure as well as the actual steps required to perform the test are provided. The LabVIEW code that was developed to do this is also presented here.

The Ghost Simulator driver is discussed in Chapter 6. The application of this driver in automated testing is briefly discussed. Then, the code is presented.

Finally, the conclusions and proposals for continuation of this work are discussed in Chapter 7.

Chapter 2

RF Sensitivity and Airplane Flutter Theory

2.1 - RF Sensitivity Theory

2.1.1. - RF Tuner

The basic television receiver is shown in Figure 2.1 [4, p.13.16]. The Tuner, IF Amplifier, and the AGC System are the main parts that affect the RF Sensitivity of the receiver. It is in these circuits that the signal is received, amplified, heterodyned down to IF, and passed on to the remainder of the receiver . Therefore, it is very important that the it be processed properly here or else the remaining circuits will not process the appropriate signal.

The block diagram for a typical VHF tuner is shown in Figure 2.2 [5, p.106]. The VHF portion can be tuned to any 6 MHz wide channel (see Figure 2.3) in the range from 54 MHz to 88 MHz and from 174 MHz to 216 MHz while the UHF portion allows channels to be tuned in the range from 470 MHz to 890 MHz.

The tuner has several responsibilities [6, p.191]. It determines which channel will be received by switching pre-tuned circuits in the RF stage and the oscillator. The impedance at its input must be matched for maximum power transfer from the

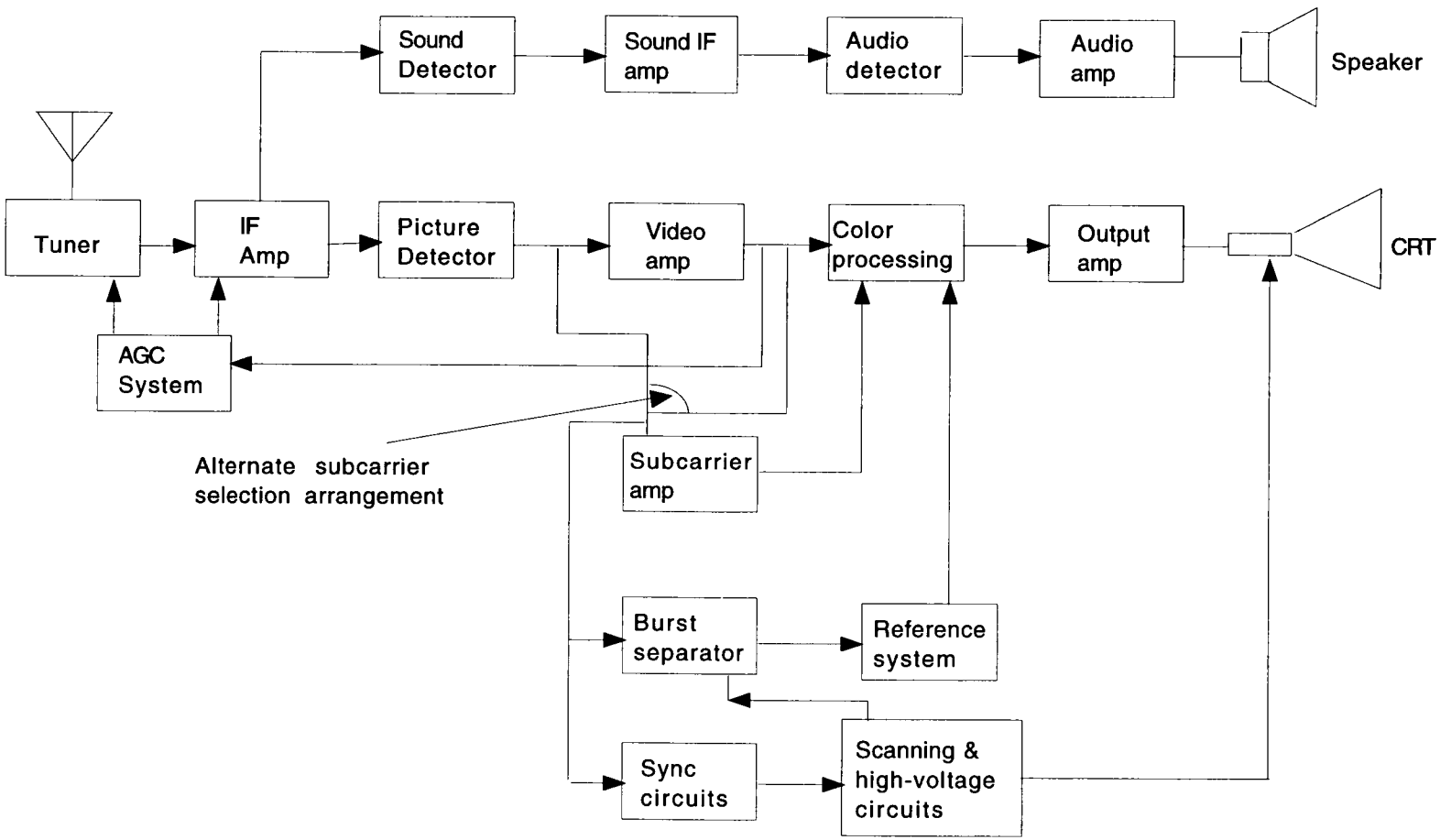


Fig. 2.1 Block diagram for a television receiver.

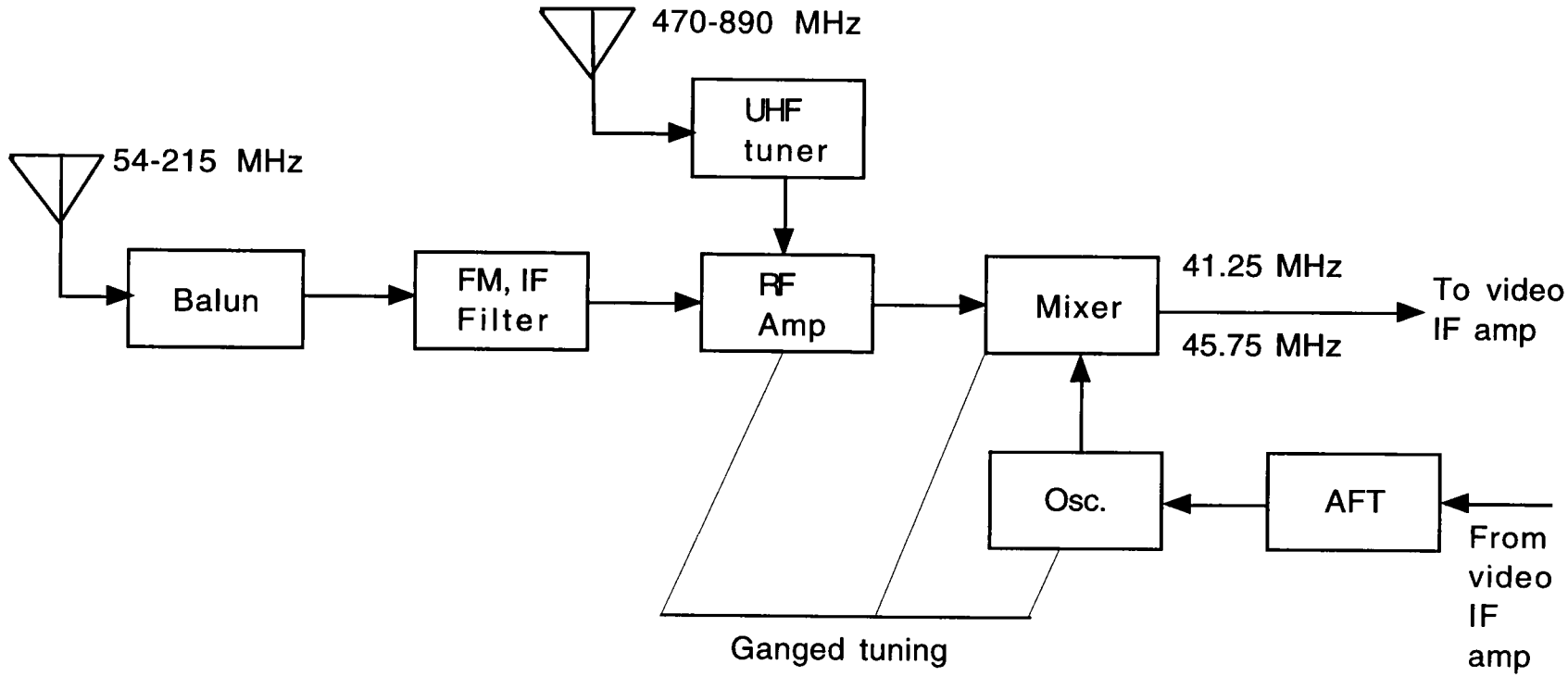


Fig. 2.2 Block diagram for a typical VHF tuner.

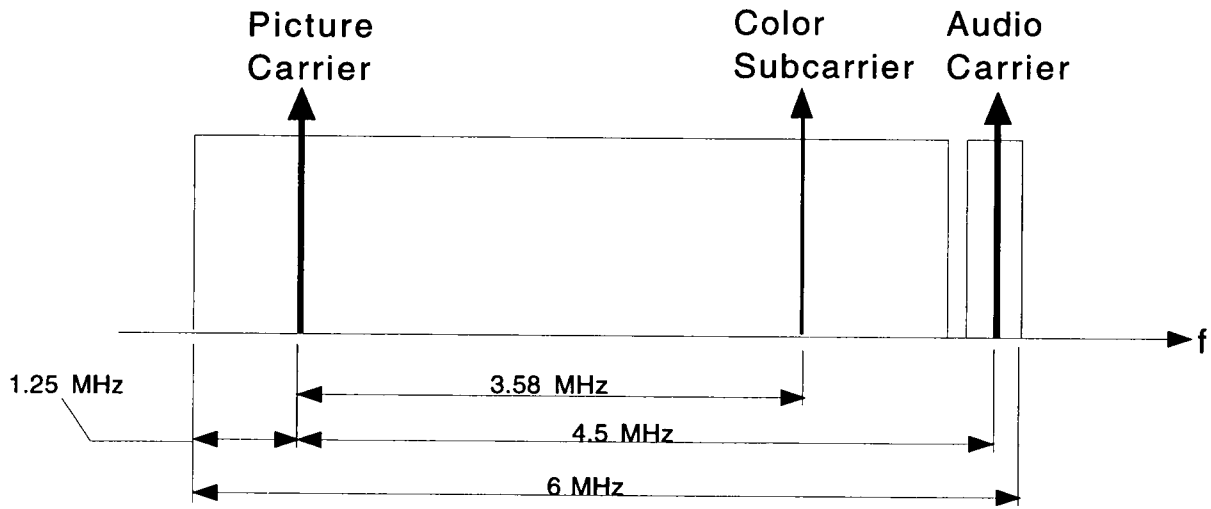


Fig. 2.3 Ideal 6 MHz NTSC channel.

antenna to the RF stage. In addition, it amplifies the signal to provide good signal-to-noise ratio. The gain of this stage is controlled by the AGC (discussed later). Then, this RF signal is down-converted to IF via the tuned local oscillator which produces a video IF carrier at 45.75 MHz and an audio IF carrier at 41.25 MHz. It should be noted that the audio IF carrier is below the video IF carrier, a reversal from the standard NTSC channel. The tuner buffers the local oscillator and consequently the IF chain from interfering signals in that range picked up by the antenna. Finally, it rejects the image frequencies by means of the RF selective circuits.

The VHF and UHF input filters provide the impedance matching necessary for maximum power transfer. In the case of VHF this is 75 ohms and for UHF it is 300 ohms. In addition, this filter helps block out reflections on the input line.

The RF Amplifier has a four-fold purpose [5, pp.122-144]. It improves the signal-to-noise ratio (SNR) of the receiver. It does this by providing adequate gain to weak signals via the AGC. This is necessary in order to provide the mixer with a high SNR because it generates noise of its own from the heterodyning operation. Mixer noise, which is the greatest source of noise in the receiver, and other internally generated noise such as random or white noise are the chief limiting factors on the RF Sensitivity of a receiver

because the weakest signal that a receiver is able to respond to must be greater than that noise level generated from within itself. The amplifier provides amplification. It must do this in order for the signal level to exceed the noise level generated by the mixer. It provides good channel tuning capabilities by isolating the mixer from the antenna. Consequently, the spurious responses and image interferences are minimized. This interference results mainly from intermodulation involving the harmonic of the oscillator plus or minus the IF, FM signals plus or minus a TV channel, or the second harmonic of FM signals. Finally, the amplifier reduces the radiation capacity of the local oscillator. It and its associated tuned circuits do this by reducing the distributed capacitance between the oscillator and the antenna. Since one is isolated from the other, the radiation and possible interference problems are minimized.

The mixer is where the IF signal is actually produced [6, p.194]. It heterodynes the RF signal with the local oscillator frequency. Since there are three carriers--picture, sound, and color subcarrier--there are three IF frequencies. These are 45.75 MHz, 41.25 MHz, and 42.17 MHz, respectively. These low frequencies were chosen in order to gain some advantages. First of all, because of the low frequencies, higher amplifier gain can be achieved. Also, the lower frequencies increase the stability of the amplifier. Third, an

improvement in tuned circuit selectivity is achieved.

The local oscillator provides the local oscillator frequency [6, p.194]. Therefore, it should be fairly stable, free from temperature drifts and minor changes in supply voltage. Also, it should always be kept separate from the mixer, otherwise it may synchronize itself with the mixer, which could result in a loss of reception. Coupled with this is the need to have the oscillator signal voltage many times larger than the incoming RF signal. This will insure that the RF signal is optimally heterodyned [5, p.127].

2.1.2 - IF Chain

A typical IF chain is shown in Figure 2.4 [4, p.13.10]. This region of the receiver has four major functions: to compensate for vestigial sideband transmission, to provide most of the selectivity required by the receiver for interference-free reception, to provide most of the picture and sound signal amplification of the receiver, and to change the IF into the video signal at the video detector.

The IF value (45.75 MHz picture IF and 41.25 MHz sound IF) is a compromise between conflicting factors. It should be as low as possible from the considerations of high and stable gain which is easier to obtain at lower frequencies. At the same time, it should be high enough to have adequate bandwidth and selectivity. These

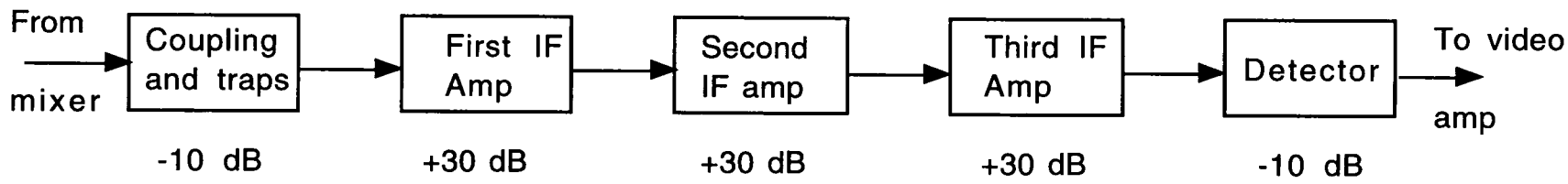


Fig. 2.4 Block diagram for a typical IF chain.

values were originally chosen for several reasons. One is that they help to minimize the interference of a television receiver with another because the local oscillator is always above each VHF band. Another reason is that the interference due to spurious reception of image frequencies, direct IF range signals, difference frequencies between frequencies of stations, etc., is minimized.

The typical IF response is given in Figure 2.5 [5, p.170]. In general, the upper adjacent-channel picture carrier and lower adjacent-sound carrier must be attenuated 40 and 50 dB, respectively, in order to avoid visible disturbances in the picture. In addition, the passband compensates for the vestigial sideband character of the transmitted television signal. The number of IF stages depends on the type of receiver and manufacturer. In general, color receivers have three or four stages of amplification whereas black and white receivers have two or three stages.

Finally, the video detector is fed from the output of the amplifiers. This component is often referred to as a second detector because it is a frequency changer in exactly the same way that the mixer stage is in the tuner. Its primary job is to lower the IF down to baseband, the frequency of the video generated by the camera at the transmitter. For black and white receivers, this consists of a composite video signal (0 MHz to 4.2 MHz), 4.5 MHz sound IF, a DC

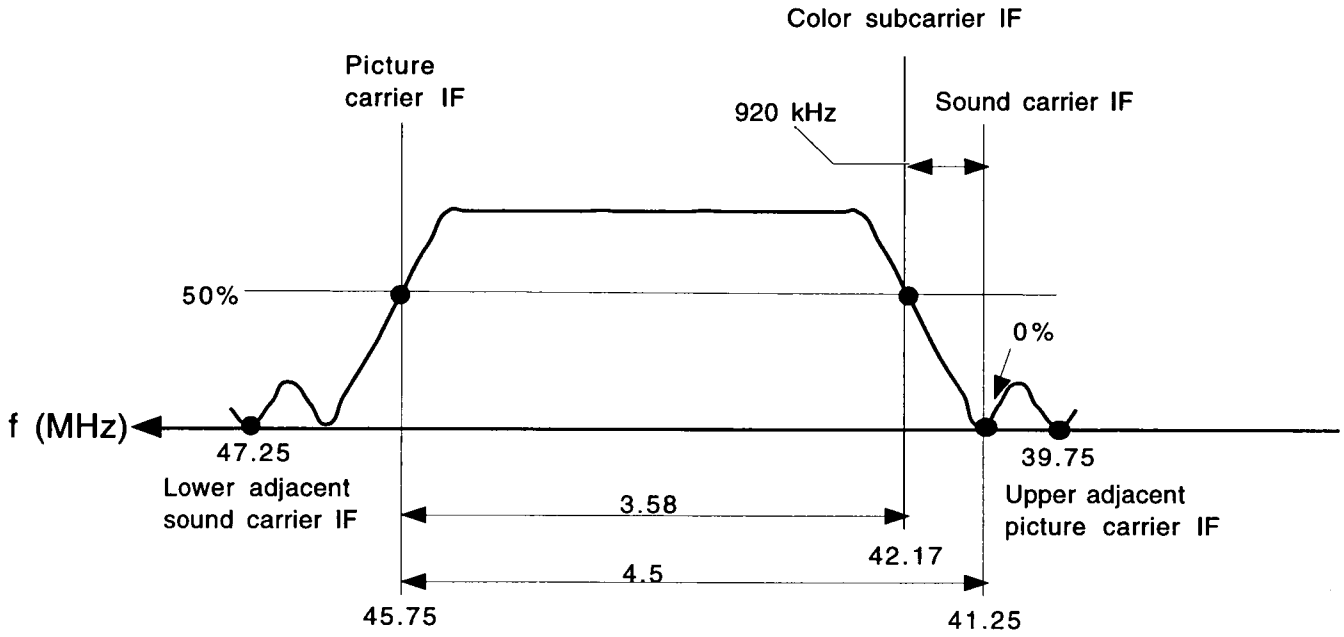


Fig. 2.5 Typical IF response.

component corresponding to the average illumination in the picture, and residual IF and higher-frequency harmonic components. Color receivers have similar video detector outputs except that the video signal signal contains the chrominance signal and the 4.5 MHz sound IF is greatly attenuated since there is a separate sound IF detector.

2.1.3 - AGC

The AGC circuit is a closed feedback loop used with all tuner and IF sections to prevent overloading from strong signals at the antenna (see Figure 2.6) [4, p.13.114]. The need for an AGC system arises from the fact that the input to the detector must be fairly constant. Therefore, the level at that input is monitored and the gain in the RF and one or more of the IF stages is adjusted accordingly. If no such control circuitry existed, a strong signal at the antenna would produce excessive contrast in the picture. If the signal were really strong, then the sync tips may get clipped, in which case, there would be a loss of synchronization and the picture would tend to "roll". In addition, negative pictures may result because overloading the video stage would cause it to no longer function as a regular amplifier with phase inversion. It might just pass the signal without inverting it resulting in a negative image on the screen .

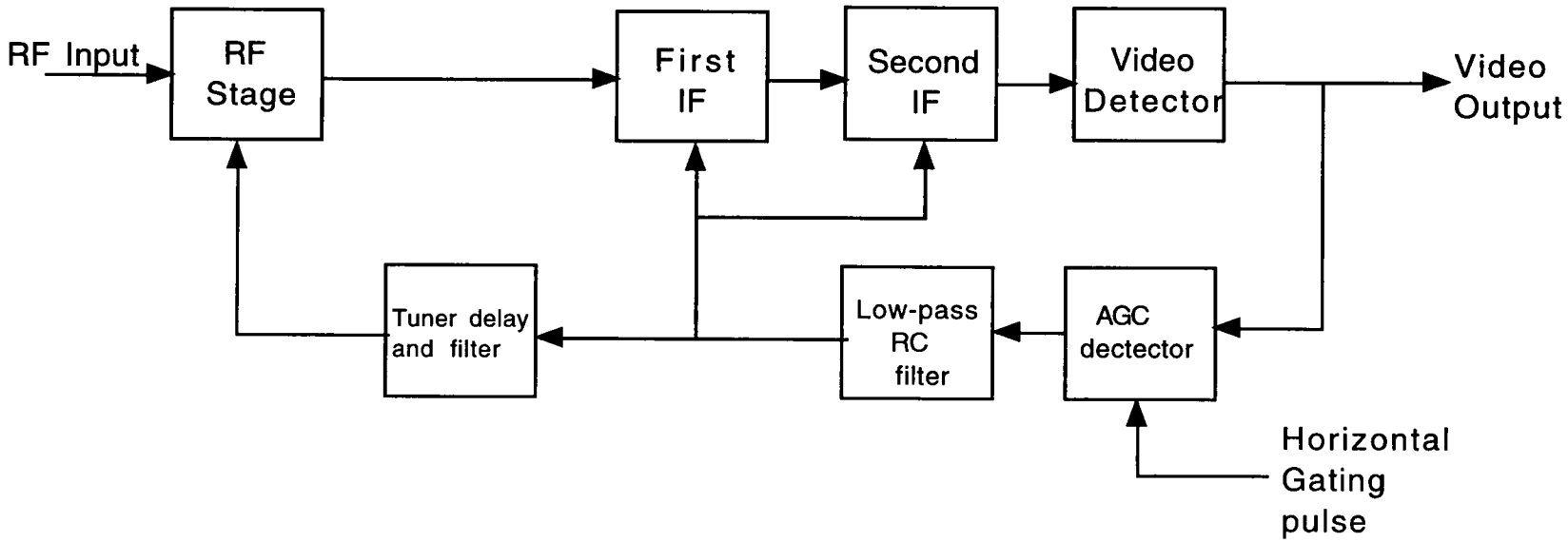


Fig. 2.6 Block diagram for a typical AGC system.

There are many different types of AGC systems. Three of the more common ones are Average AGC, Peak or sync clamp AGC, and Keyed or gated AGC. Average AGC is based on the principle of keeping the carrier level constant in the IF. Then, as the video modulation levels of the carrier change, the AGC is affected. Peak AGC compares the video sync tip level with a fixed DC value. If the sync tip level is higher than the reference, then the gain is reduced to bring it down to that reference level. Finally, Keyed AGC is similar to the Peak AGC. Here too the sync tip is compared to a reference level but only in a stage which is active when the sync is transmitted. As can be seen in Figure 2.6, the horizontal gating pulse acts as the key or gate for this stage. This method offers greater noise immunity than the other two systems.

RF Sensitivity defines the receiver's ability to handle low-level signals. If the tuner is not designed properly, then the input to the IF chain may be noisy, off frequency, or may contain images. Consequently, the IF chain will have a difficult time processing the signal. On the other hand, if the IF region is not designed properly, the input to the video detector may not be the correct level or else may contain image frequencies which made it through the passband. Related to both of these is the AGC. If it is not designed properly, then signal levels could be too high or too low, the picture could be

poor because of too much contrast, or the picture may display a loss of synchronization. Poor or faulty design of any of these will cause the receiver to have a much higher RF Sensitivity level than it should.

2.2 - Airplane Flutter Theory

When an airplane flies overhead between the transmitter and receiver, there are three main paths by which the signals arrive at the receiver. These are illustrated in Figure 2.7 [5, p.202]. The strength of the composite signal is dependent on the phase of the sky wave. Since the airplane is in motion, this will vary constantly causing rapid signal strength variations at the receiver giving rise to a picture that appears to "flutter". That is, its contrast changes very fast, at times even 20 or 30 times per second. In addition, there will be a loss of synchronization if the AGC circuitry cannot track the changing level fast enough. So, the *flutter frequency* is defined as the rate of phase change of the reflected signal with respect to the primary signal [7, pp.1-5].

Because the reflected path is longer than the direct wave, the reflected signal will arrive at the receiver a short time after the main signal. Therefore, a "ghost" image is also produced. That is, a double image of the video pattern is produced. The displacement of

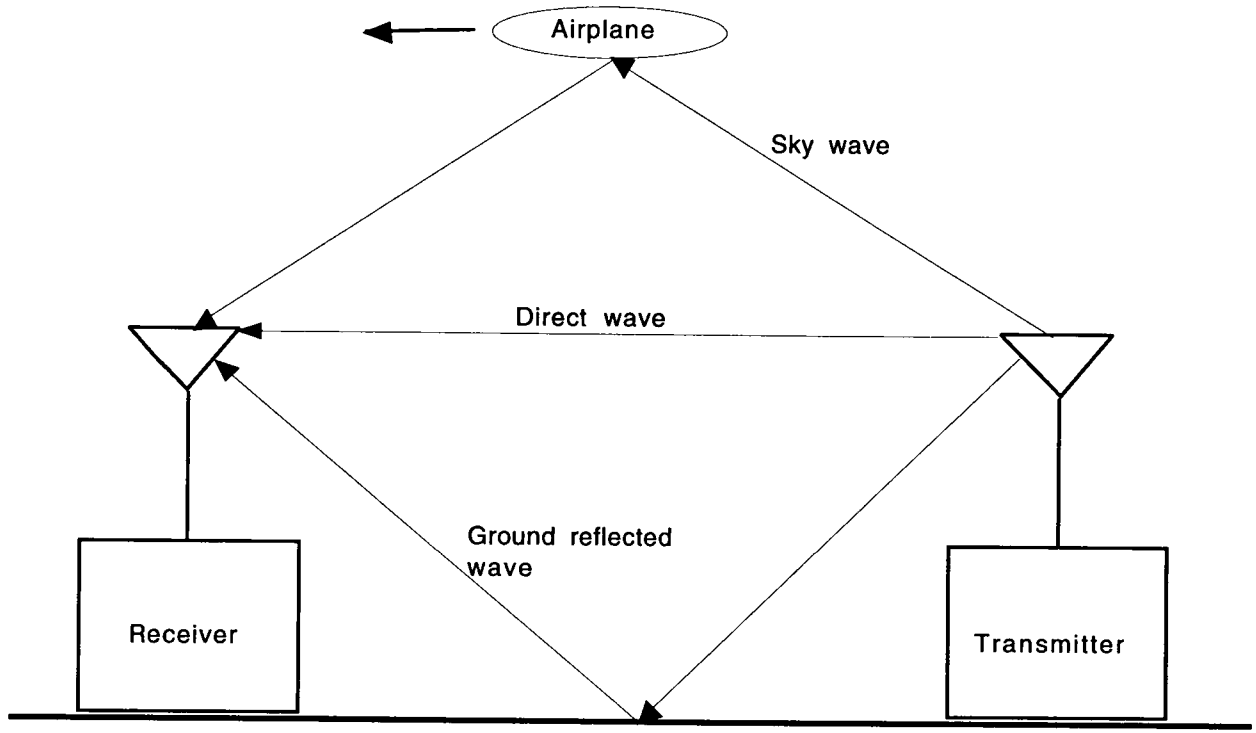


Fig. 2.7 Multipath signal reception due to an airplane flying overhead.

the image with respect to the main pattern will also change with time because the path length is changing with time. The Airplane Flutter Test tries only to determine this flutter frequency for a given receiver. It assumes that there is no ghosting.

The main part of the receiver that affects these characteristics is the AGC. As noted in the previous section, there are many types of AGC systems. However, it has been found that the primary one used to aid in reduction of airplane flutter is the Keyed AGC. A block diagram of the Keyed AGC system is shown in Figure 2.8 [5, p.209]. A simple AGC system has two inputs. One is a video signal with a positive going sync pulse that is taken from the output of a DC-coupled video amplifier. The other is obtained from the horizontal output transformer and is a large, constant-amplitude, positive-going pulse that is generated during horizontal retrace. Thus, the two occur at the same time. In Keyed AGC, the keyer circuit is arranged to be off between horizontal sync pulse intervals and to conduct only when both pulses are present. The amplitude of these horizontal sync tips determines the amount of conduction. Then, the conduction amount determines the DC voltage used for the AGC.

The video detector output may have frequencies as low as 30 Hz. In order to eliminate these from the AGC output, filters must be

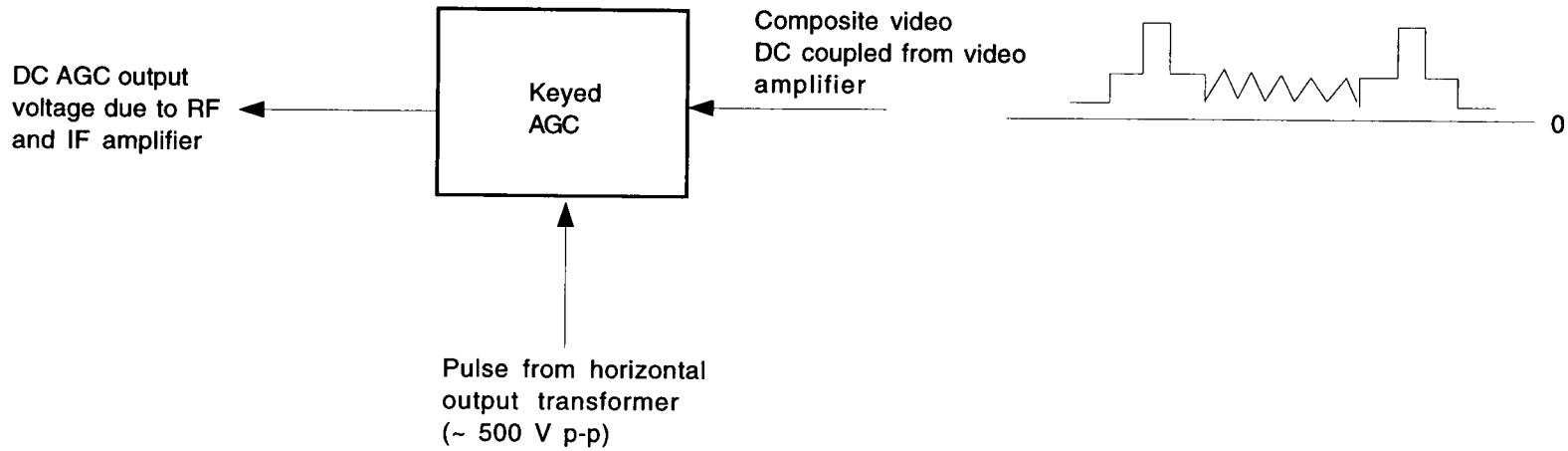


Fig. 2.8 Keyed AGC system.

used and they must have long time constants. Consequently, the AGC will experience an increase in reaction time. So, it will not be able to track rapidly-changing signals which are the main cause of airplane flutter.

In Keyed AGC, the keying pulses are at the rate of the horizontal pulses. So, the AGC current pulses conduct at the same rate producing ripples in the AGC at the same frequency. Consequently, these can be filtered out by low-time-constant filters. Therefore, the AGC will be able to track rapid signal changes much better and thus lessen the effects of airplane flutter [6, p.247]

2.3 - Ghost/Flutter Simulator Theory

The fundamental principle behind simulating airplane flutter and ghosting is shown in Figure 2.9 [7, pp.6-7]. Channel B accepts undelayed video and uses it to modulate the carrier as the reference signal. Channel A delays the video by a suitable amount. The local oscillator is delayed by the Phase Shift Amplifier and then modulated by the delayed video signal. When the delayed carrier phase is modulated between 0 and 500 Hz, the addition of the two modulated signals will yield simulated airplane flutter.

The simulation has been divided into two separate delays. The carrier is delayed at some repetitive rate between 0 and pi radians.

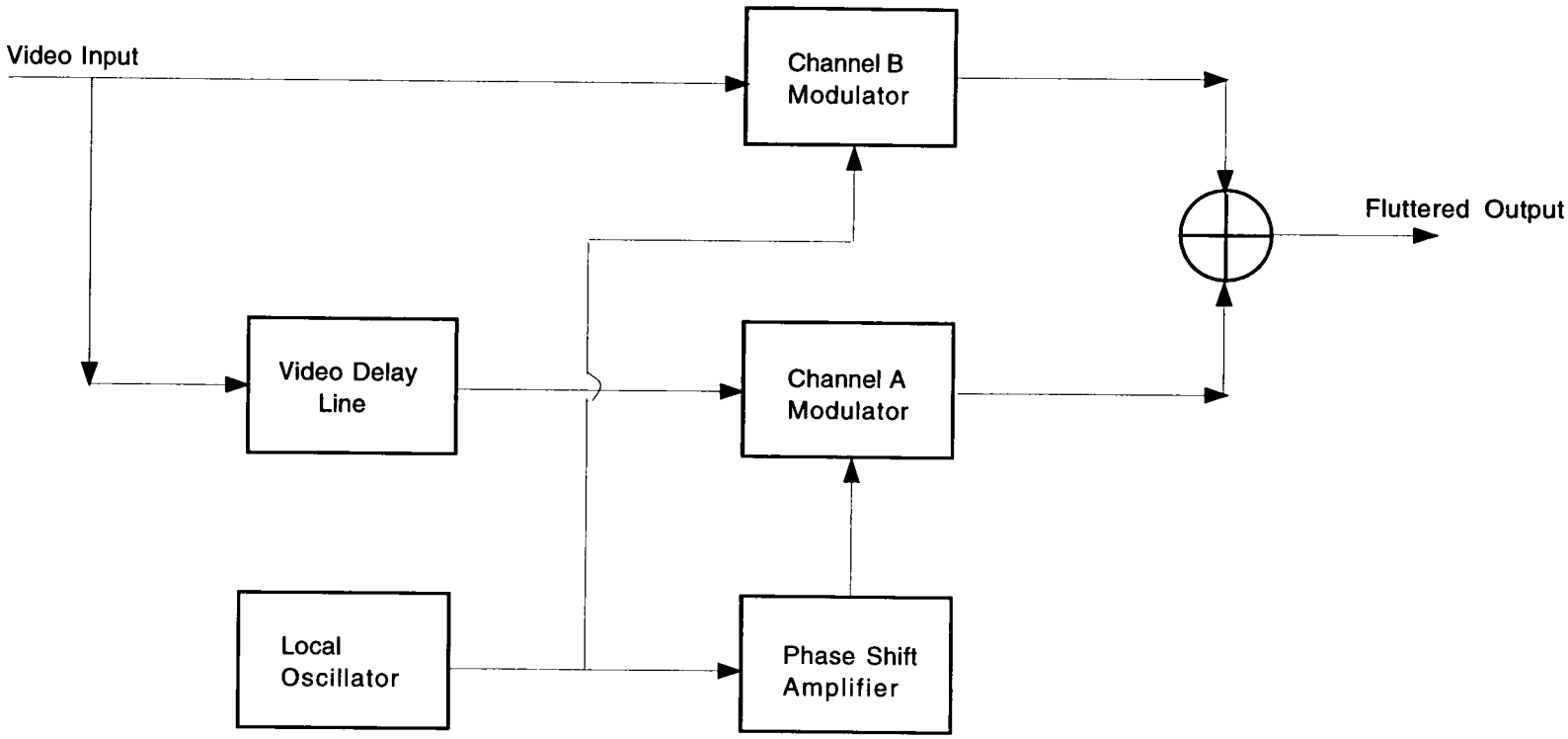


Fig. 2.9 Fundamental principle for flutter generation.

The real path difference in the signals is indicated by the delay of the modulation envelope. Therefore, the generator produces flutter by delaying the RF carrier by pi radians and delaying the modulation envelope by a period that would normally be given by much longer RF signal paths.

The HP 11759D Dynamic Ghost Simulator is designed to emulate such flutter and ghosting. It allows for one RF channel with six independently controlled paths each of which has Rayleigh fade (i.e., the attenuation in the path changes according to a Rayleigh probability distribution function), delay, and Doppler/phase shift capabilities. The block diagram for the simulator is given in Figure 2.10 [8].

The simulator is controlled via an HP Vectra VL2 486/66 computer. There is channel simulation software which is supplied with the simulator to allow access to its various functions. The two machines are connected via a Centronics parallel connector.

The RF input signal must be in the range from 40 MHz to 1 Ghz. This signal is mixed with the LO input signal which must be 6 MHz below the RF frequency. So, the output of the mixer is at a frequency of 6 MHz which is then split and fed to each of the six paths where it is processed. This processing depends on the mode of the simulator (Rayleigh, Doppler, or Phase).

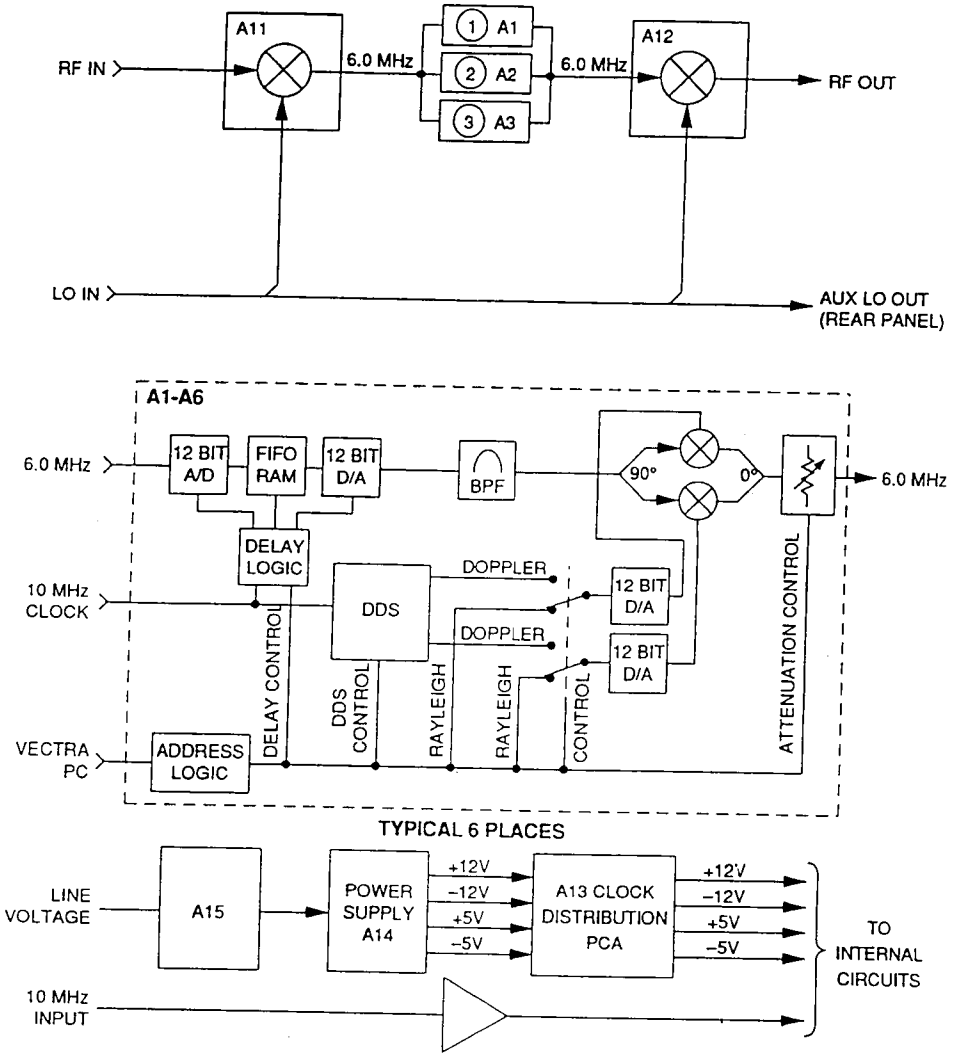


Fig. 2.10 HP 11759D Ghost Simulator block diagram.

Ghosting is produced via the delay logic. The 6 MHz signal is digitized by a 12-bit A/D. This discrete signal is fed into a First-In-First-Out (FIFO) RAM block and then is converted back to a continuous signal by a 12-bit D/A. The resulting analog signal is filtered and amplified by a bandpass filter. The amount of delay is defined by the user in the software and is controlled by the WRITE and READ signals to the FIFO which are controlled by the PC.

Airplane flutter is produced via the Doppler Shift logic. First, the amplified signal out of the bandpass filter is split into an I and a Q channel which are separated by a 90 degree phase shift. These are then each mixed with a sinusoid, recombined, and shifted to have zero phase offset. If a Rayleigh fading channel is desired, then specific data files are accessed to provide the mixing signals rather than the sinusoid. Finally, the attenuation is provided by an analog multiplier operating in a variable gain mode. The gain is controlled by a DAC, controlled by the PC.

Now that the signal has been processed, it must be up-converted back to RF. So, the processed signal is mixed with the same LO as when it was down-converted. Thus, the RF Output has the processed signal at its original RF input frequency.

Chapter 3

Code Development Methodologies

3.1 - Introduction

It was a long process to derive an efficient code development methodology. This was the first project of its kind and magnitude for all of the developers involved. Therefore, an iterative approach was necessary to arrive at an acceptable solution to the problem of code development. During the course of the project, two different approaches to code design were used: Cleanroom Software Engineering (CSE) and Adapted Cleanroom Software Engineering (CSE-A).

Until 15 or 20 years ago, the software portion of large combined hardware/software projects had been viewed as a minor task towards the completion of that effort. Consequently, very little attention was given to it. This in turn meant below-par quality code. In addition, a great deal of time was required for rework and "code fixes" (i.e., debugging).

Therefore, CSE is based on the tried and true practices of hardware design [9]. It is now one of the most successful code development methodologies. It is rooted in the idea that software--like hardware--should be designed using a top-down approach. A

designer would not sit down at his work bench and start wiring components together. Thus, why should he sit down at his computer and start writing code? He should think of the program as a system that has inputs and outputs and is made up of sub-systems (subroutines). Thus, he must start at the highest level and work his way down.

CSE is based on the assumption that the development would be done in a text-based programming environment. This is not the case with LabVIEW. So, the process had to be modified in order to use it in the LabVIEW environment. Therefore, the discussion below (Section 3.2) reflects a modified version of CSE.

In time, it became apparent that even this modified version was awkward to use with a graphical programming language. Therefore, the methodology was adapted, on a large scale, to be more efficiently used in this environment. This will be referred to as Adapted Cleanroom Software Engineering (CSE-A).

The following sections describe both of these methodologies. Section 3.2 consists of an overview of CSE, a discussion of how it was modified for this project, a discussion of conventions and standards, and a short example at the end of the section. Section 3.3 consists of a similar discussion that focuses on CSE-A. Finally, Section 3.4 has a discussion of the methodology used to develop the

HP Ghost Simulator Driver including a short example at the end of the section.

3.2 - CSE

3.2.1 - Overview of CSE

Traditionally, "hacking" has been the standard method used by engineers for developing software. This means that the software was written as fast as possible based on an underdeveloped system requirements document. In addition, there was little interaction with the customer. Therefore, a considerable amount of time was allotted at the end for debugging.

Software engineering is an alternative to hacking. The main idea in this methodology is to develop software in a way that is similar to the way hardware is developed. That is, engineering principles are used in software engineering [9].

The customer is the starting point in both processes, traditional and CSE. The system requirements are defined by the customer--as they realize them at that point in time. Then, the developer completes his task based on this immature requirements document. So, when the customer is first exposed to the software, it may be faulty or it may not be what was ordered because of the

lack of customer interaction during the design process. Therefore, more time and money must be spent in order to bring the code to proper functionality and to specification. It has been consistently shown that this process is not efficient.

In CSE, the software must be developed in a very rigorous, statistically-controlled manner from the top down rather than the bottom up. The overall process is given in Figure 3.1 [10]. The first step is to meet with the customer and iteratively develop a solid System Requirements document. This may require several iterations because the customer does not always know exactly what he wants until he sees the designer's interpretation of what he wants. The next step is to produce an Incremental Development Plan which is an approximate timetable by which the project will run. This plan has approximate dates and milestones which will need to be updated as the project progresses. Then, the Top-Level Specification is created. This usually involves two types of specifications: User Interface Specification and Functional Specification. These latter two steps involve constant customer interaction as well so that these respective documents will truly reflect the customer's wishes as well as the process that will be used to fulfill them. Only after these items have been developed will the actual design process begin. This is described in the middle of Figure 3.1. During the

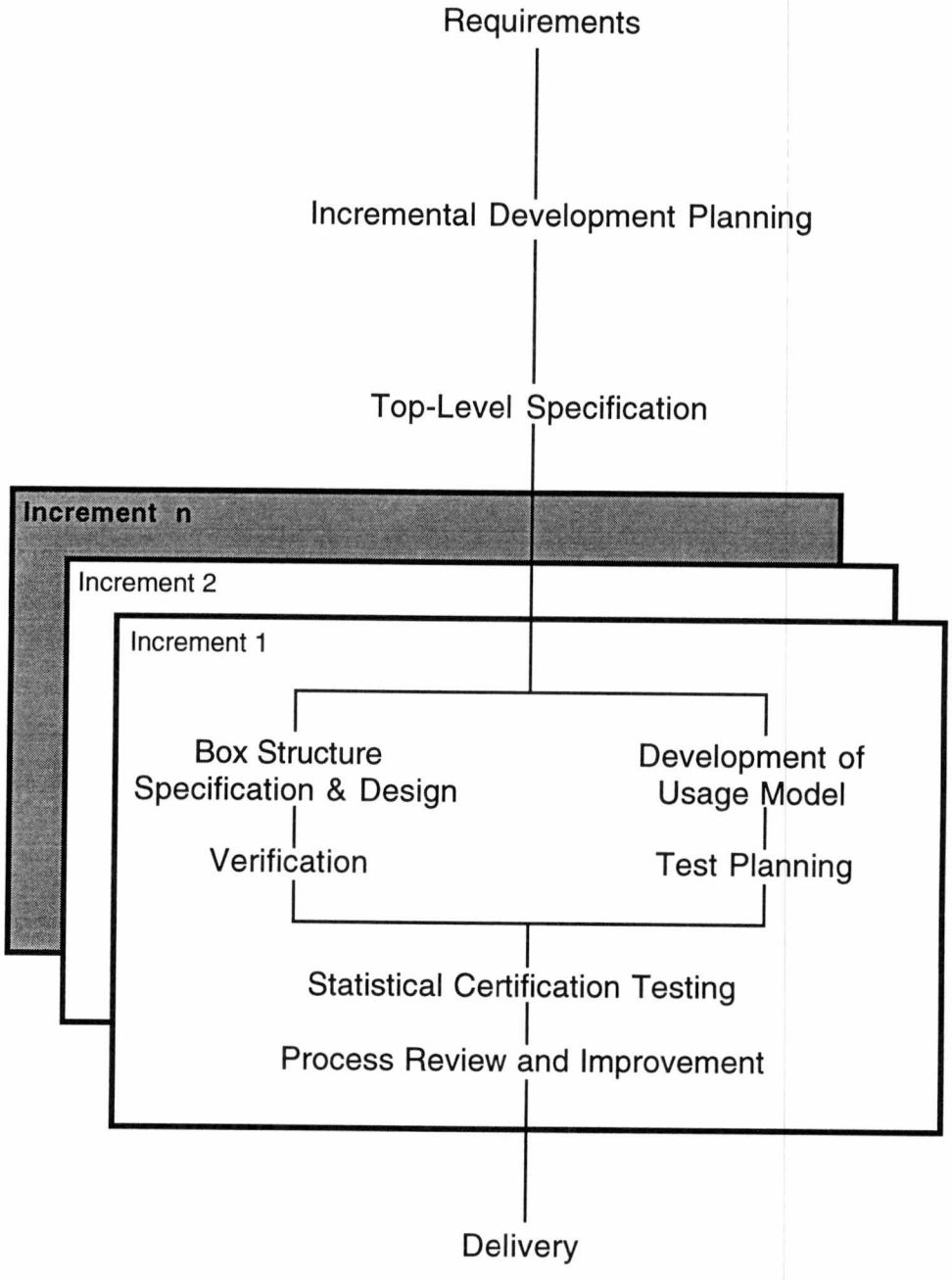


Fig. 3.1 Diagram of the CSE process.

design increments, there is also interaction with the customer after each increment (see Figure 3.2) [23]. Finally, after the final increment there is delivery of a system with very few errors and a high customer satisfaction rating.

An integral part of the project is the documentation. The various documents mentioned above must be maintained in an orderly fashion and updated during each increment. In addition to these, there is another very important document which must be maintained throughout the project. This is the Process Control Document. This document is used to record how the process was controlled so that it did not digress into the traditional method of software development. Here the design process itself is documented including such facets of the project as the method for writing code, statistical testing, usage modeling, certification of completed software, etc.

There are two facets of CSE that set it apart from the traditional means of software development. These are Box Structure Specification and Design Verification. Box Structure Specification is shown in Figure 3.3 [9]. This process should be followed when developing each routine and subroutine. First there is the Black Box. This is a stimulus-response listing for the given routine or sub-routine. At this stage, it does not matter how the routine will

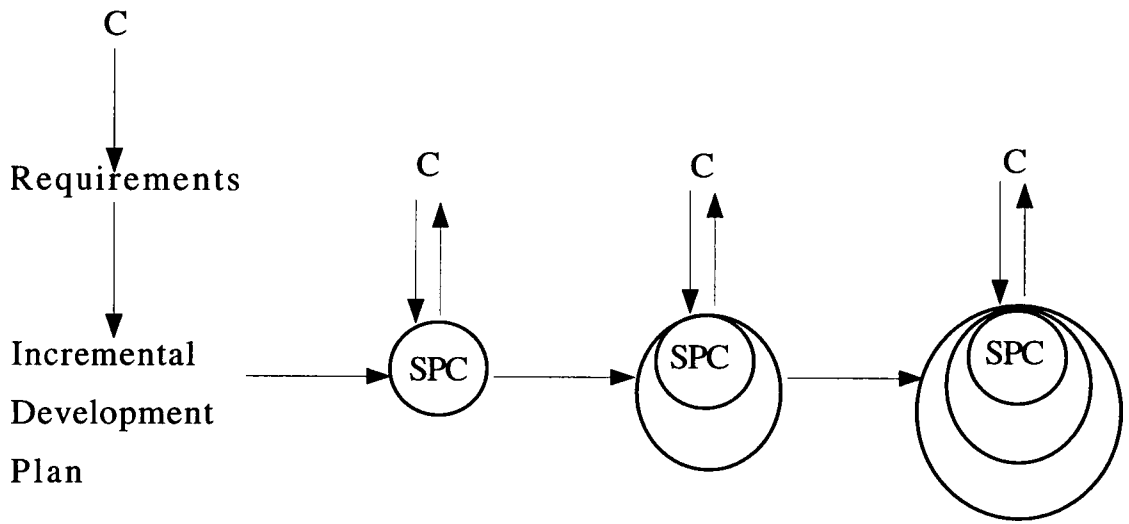


Fig. 3.2 Incremental software development under SPC.

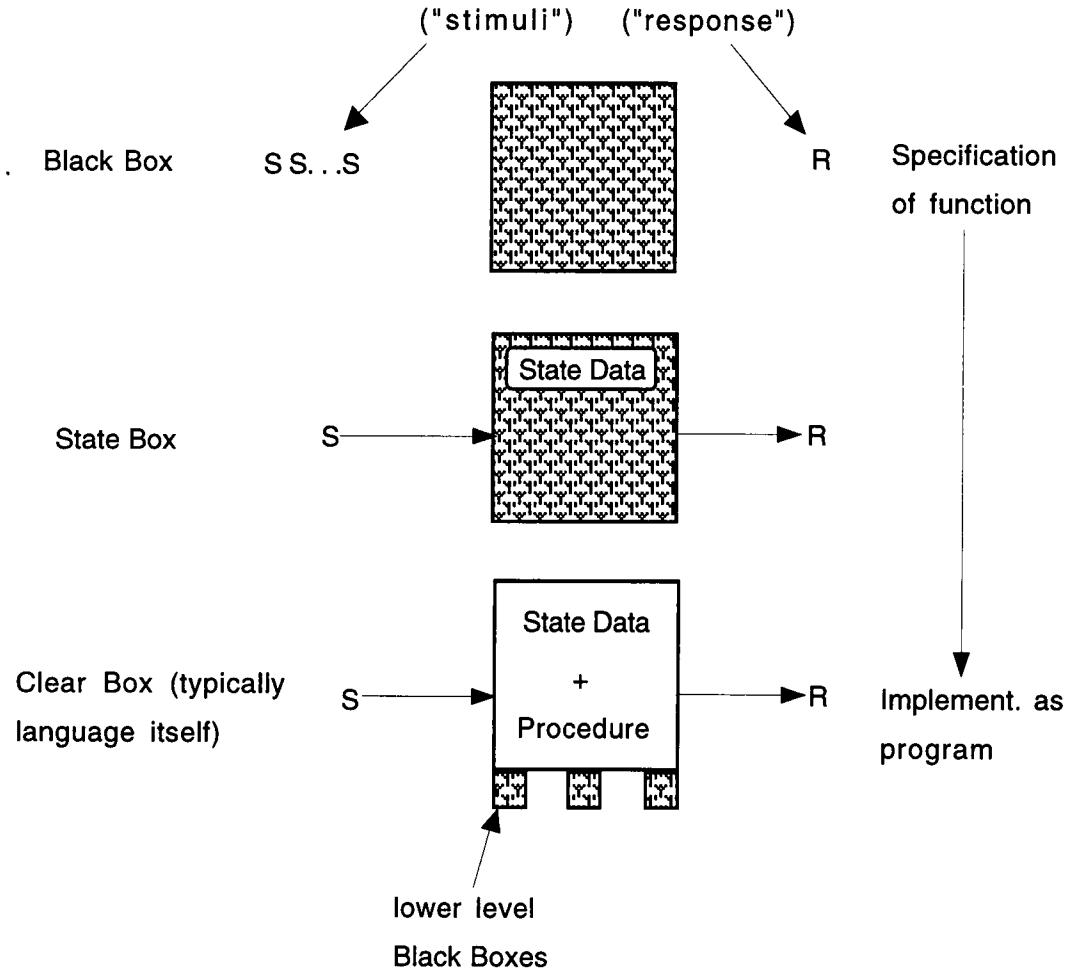


Fig. 3.3 Box Structure specification.

function, only what its inputs and outputs will be. Then, the State Box is created. To create the State Box, the Black Box is copied in its entirety. Then, the state data are determined. These include such things as shift registers and variables that change with each execution of the routine. Then, the Clear Box is constructed by starting from the State Box level and developing the procedure that will generate the shift register values and other variable values at each execution of the program. Then in turn, a given Clear Box may have lower level Black Boxes (subroutines) that it calls to perform lower level functions.

Once the Clear Box is developed for a routine, this code must go through Design Verification. This is where an independent party (that is, someone other than the designer) tries to verify that the program has been designed according to specification. The unique concept here is that the verifier begins with the Clear Box and attempts to collapse it into the State Box and then take that back into the Black Box. Basically, he tries to perform the design process for that program in reverse to see if he is able to derive the same set of stimuli and responses that the designer specified at the Black Box level. The code is not executed during verification. In this manner, many errors in the code can be found and reported to the designer before the code is ever executed. Therefore, much time is

saved in the long run and consequently much money.

Occurring concurrently with these two processes are two other processes. These are the development of Usage Models and Test Planning. The purpose of these is to answer the question: "What do you expect the user to do?". An excellent method for developing such models is to employ probabilistic techniques. So, usage scenarios are generated which help to determine where the failures will be and with what frequencies.

This problem may be approached in one of three ways, uninformed, intended, and informed. In the uninformed approach, a uniform probability distribution function (pdf) is assumed. This is the case where all usage scenarios are equally likely. In the intended approach, there are no data to assign probabilities. However, something is known about the environment in which the user will operate. So, a "guess" may be made as to the probability distribution. Finally, the informed approach is used when there is a basis for assigning probabilities [9].

Once the pdf is determined, the test planning can be done. Statistical test cases are generated to help visualize actual usage of the system. Consequently, the designer is alerted to potential failures and of possible errors in coding. Also, using such statistical sampling, generalizations can be made about the

operational usage of the system.

This concept can be best illustrated with an example. Figure 3.4 contains a diagram for a simple program wherein a user is able to select from one of three games. Then, whenever he is finished, he exits the program. Assuming the intended approach is used and assuming there are ten people who will execute this program with the known frequencies, the probability distribution may be determined. These values are given in Table 3.1. In this manner, the designer is able to identify the most frequently pursued paths in the menu structure. In addition, by having a variety of users execute the code, he is also able to identify possible errors in the code and failures in execution.

Table 3.1 Usage Model for Program in Figure 3.4.

Transitions	Frequency	Probability
0 --->1	10	1.00
1 --->2	10	0.33
1 --->3	3	0.10
1 --->4	7	0.23
1 --->5	10	0.33
2 --->1	10	1.00
3 --->1	3	1.00
4 --->1	7	1.00
5 --->0	10	1.00

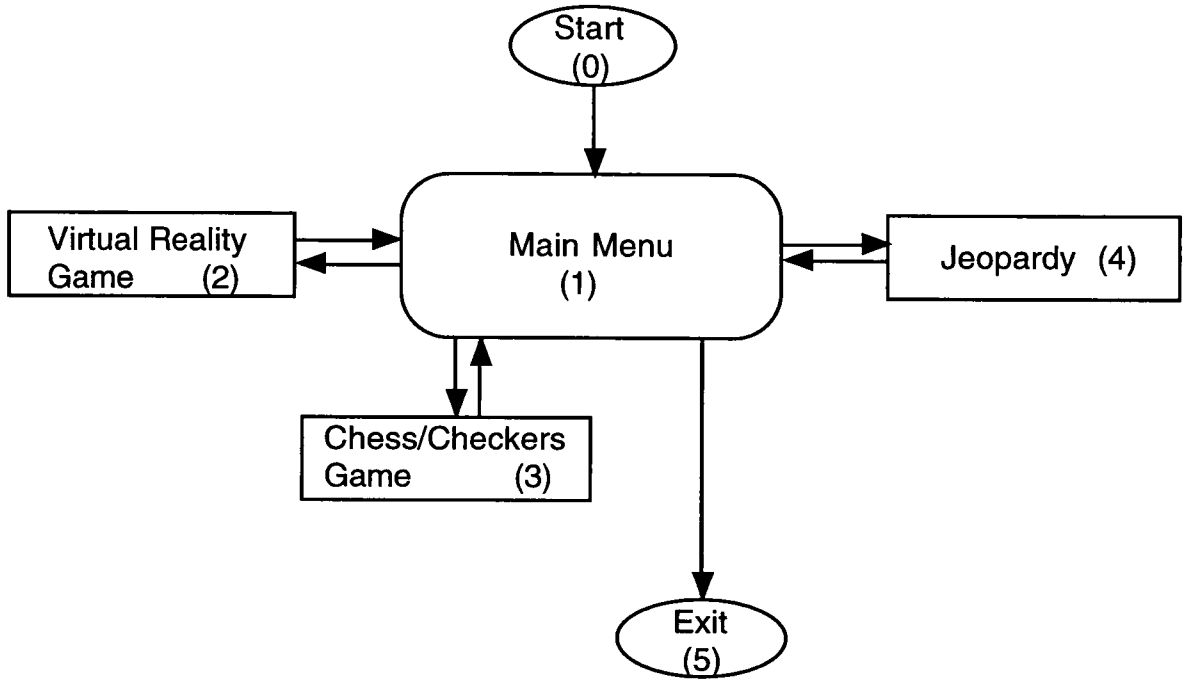


Fig. 3.4 Diagram for game selection program.

Statistical Certification Testing is an integral part of the above mentioned procedure. This involves running test cases. Since it is impossible to exhaustively test every scenario, these test cases provide a statistically sound basis for measuring the quality of the software. The certification team executes these cases and tries to determine such things as Mean-Time-to-Failure (MTF) and error conditions generated by certain sequences during execution. So, when applied properly, the same Statistical Process Control (SPC) methods that have been used for years in manufacturing can now be used in engineering software.

After each cycle or increment, the overall design process is reviewed and evaluated in order to determine where improvements could be made. Then, these changes are implemented during the next increment. This process is repeated until the product is delivered.

This process has yielded outstanding results in industrial projects of varying sizes (see Figure 3.5) [11]. The table shows errors per thousand lines of code (errors/KLOC) in the single digits. Traditionally, these have been in the 20's and 30's. So, the benefits of CSE are quite clear: a superior product that is designed quickly and efficiently with a savings in time and money.

Year	Project	Quality and Productivity
1987	IBM Flight Control: Helicopter Avionics System Component 33 KLOC (JOVIAL)	• Certification testing failure rate: 2.3 errors/KLOC • Error-fix reduced 5X • Completed ahead of schedule
1988	IBM Cobol Structuring Facility: Product for automatically restructuring COBOL programs 85 KLOC (PL/I)	• IBM's first Cleanroom product • Certification testing failure rate: 3.4 errors/KLOC • Productivity 740 LOC/PM, 5X improvement • 7 errors in first 3 years of use; all simple fixes
1989	NASA Satellite Control Project 1 40 KLOC (FORTRAN)	• Certification testing failure rate: 4.5 errors/KLOC • 50% improvement in quality • Productivity 780 LOC/PM • 80% improvement in productivity
1990	Martin Marietta: Automated documentation system 1.8 KLOC (FOXBASE)	• First compilation: no errors found • Certification testing failure rate: 0.0 errors/KLOC (no errors found)
1991	IBM System Software First increment 0.6 KLOC (C)	• First compilation: no errors found • Certification testing failure rate: 0.0 errors/KLOC (no errors found)
1991	IBM AOEXPERT/MVS™ Product 107 KLOC (mixed languages)	• Testing failure rate: 2.6 errors/KLOC • Productivity 486 LOC/PM • No operational errors from Beta test sites
1991	IBM Language Product First increment 21.9 KLOC (PL/X)	• Testing failure rate: 2.1 errors/KLOC
1991	IBM Image Product Component 3.5 KLOC (C)	• First compilation: 5 syntax errors • Certification testing failure rate: 0.9 errors/KLOC
1992	IBM Printer Application First increment 6.7 KLOC (C)	• Certification testing failure rate: 5.1 errors/KLOC
1992	IBM Knowledge Based System Application 17.8 KLOC (TIRS™)	• Testing failure rate: 3.5 errors/KLOC
1992	NASA Satellite Control Projects 2 and 3 170 KLOC (FORTRAN)	• Testing failure rate: 4.2 errors/KLOC
1993	University of Tennessee: Cleanroom tool 20 KLOC (C)	• Certification testing failure rate: 6.1 errors/KLOC
1993	IBM 3490E Tape Drive 86 KLOC (C)	• Certification testing failure rate: 1.2 errors/KLOC
1993	IBM Database Transaction Processor First increment 21.5 KLOC (JOVIAL)	• Testing failure rate: 2.4 errors/KLOC • No design errors, all simple fixes
1993	IBM LAN Software First increment 4.8 KLOC (C)	• Testing failure rate: 0.8 errors/KLOC
1993	IBM Workstation Application Component 3.0 KLOC (JOVIAL)	• Testing failure rate: 4.1 errors/KLOC
1993	Ericsson Telecom AB Switching Computer OS32 Operating System 350 KLOC (PLEX, C)	• Testing failure rate: 1 error/KLOC • 70% improvement in development productivity • 100% improvement in testing productivity
	NOTE: All testing failure rates are measured from first-ever execution.	KEY: KLOC = thousand lines of code PM = person month X = (mathematical) times

Fig. 3.5 Results in industrial projects of varying sizes.

3.2.2 - Procedures and Conventions used in CSE

In section 3.2.1 an overall process description of CSE was discussed. Now the discussion is shifted to a less general area. The focus is on CSE as it was applied to the ATF project. The complete CSE process was not used because of its extensive nature.

The current project employed the CSE process as far as the verification stage along with some efforts to review the process and improve it. There was a minor attempt at usage modeling but no attempt was made at test planning and statistical certification testing. However, there was a minor effort made towards an adapted form of certification.

The parts of the process that were utilized were used extensively. The team firmly maintained two documents: System Requirements (SR) and Incremental Development Plan (IDP). In addition, the User Interface Specification (UIS) and Process Control Document (PCD) were largely maintained.

As described previously, the SR document is developed in an iterative manner. That is, it is written and rewritten several times over the course of several months until a version is written upon which both the customer and the team can agree. This iterative method is most beneficial because it helps the customer define what

it is that he wants which in turn helps the team define what it is supposed to do.

Once the requirements are fairly well defined, the IDP can be developed. This too must done iteratively. After each increment the IDP is reviewed and revised because of unforeseen setbacks or problems such as hardware malfunctions. Such problems cause delays in the development which result in delays in the schedule.

The functional specifications for the code are in the form of test procedures (discussed further in Chapters 4 and 5) which were written by the customer. These procedures are documents which detail the method used in performing a given test on a device. Consequently, the specifications for the hardware drivers are embedded in these procedures. Therefore, these procedures are used to develop the Box Structure Specifications and Designs.

The team adapted the Box Structure Method and BDL developed by Mills [12]. Mills' work could not be used directly because the LabVIEW environment was completely graphical whereas Box Structures and BDL were developed for text-based languages. BDL is a type of pseudo code. It requires the use of a fairly rigorous syntax similar to FORTRAN and C. It is used as an intermediate step in progressing from the functional specs to the actual code. The functional specs are translated into BDL which is translated into

LabVIEW code.

Once the designer finishes a program, two members of the team are employed to verify the work. This verification was also customized to suit the needs of the current project. Ideally, an independent party rather than team members should do verification. The Verification Checklist is shown in Figure 3.6 [13].

The verifiers must go through the checklist and note all items that are not acceptable or ones that do not follow adopted conventions (discussed in the following sub-section). The main job of the verifiers is to check the logic of the program. In doing so, they try to answer two questions: Is the program precise? Is the program accurate? The program is precise if the BDL and the code agreed. The program is accurate if it actually does what it is supposed to do. Finally, if the verifiers are satisfied that the program meets all of the criteria on the checklist, then they sign off on the program. However, if they have reservations then they have the option not to sign and require the designer to correct errors (if any), modify logic, or even improve the design to be more efficient. All of this is done without executing the code at all and with each program standing alone not integrated into the rest of the system.

As mentioned above, a minor attempt was made to do certification. However, this was not statistical certification. When all

ATF PROJECT
Verification Checklist

Designer's Name_____ VI Name:_____

Item	Comments
1. COLORS	
a. BB: Pastel Yellow (R=255,G=255,B=204)	
b. SB: Olive Green (R=205,G=255,B=102)	
c. CB, etc.: Gray (R=127,G=191,B=191)	
d. i. ALL Block Diagrams=grey(229,229,229)	
ii. ALL TEXT=BLACK	
iii. BDL = Gray(204,204,204)	
2. SPACE (BB,SB,CB)	
a. Vertical Alignment	
b. Horizontal Alignment	
c. Plenty off GREY space on Block Diagram	
3. DESIGN (CB)	
a. BDL and Code match (precision)	
b. BDL accomplishes task (accuracy)	
c. BDL box shrunk to display only: "define..."	
d. GET INFO: Overview, Instructions (if necessary), I/O (if necessary), Authors' names, Modification History	
e. DESCRIPTION BLOCKS: use BDL with extra text if necessary	
f. HELP Boxes should be descriptive;NO BDL	
g. NO type conflicts; ie, no grey dots	
h. State Variables & Shift Register Variables appear in BDL	
4. MISCELLANEOUS	
a. Wiring cleanliness	
b. Descriptive Icons	

ADDITIONAL
COMMENTS_____

Verifier #1_____

Verifier#2_____

Fig. 3.6 Verification Checklist.

programs had passed verification, then they were integrated and ran for the first time by the certification team. Then, the certification team would go through any number of scenarios and sequences in order to determine failures and error in logic. Once these were corrected, the code was deemed certified.

3.2.3 - Standards Adopted for Coding

In order to give all of the code a uniform appearance and flow, it is necessary to adopt standards to be used during coding. Otherwise, each program will be developed differently and thus there would not be any consistency between programs. Therefore, such conventions were adopted and are the topic of this section.

The major reason for standardizing code development is to ensure that high quality code is written. This in turn means that less time is needed for debugging. Debugging is an expensive, time consuming, and laborious process. Another benefit is that code reuse is maximized.

The most important convention was the use of BDL. This requires the designer to initially create a Black Box (BB) which simply defines the stimuli (inputs) and responses (outputs) of a program. Then, the designer proceeds to write pseudo code using a modified version of BDL. This pseudo code is very similar to

statements in FORTRAN or C. A considerable amount of the total development time is spent on writing this pseudo code. Therefore, the time required to write the actual LabVIEW code is much less. Consequently, the result is much higher quality code because much thought is put into the design before the actual coding is done.

Another objective behind standardizing code development is to give it the appearance that one person has written it all rather than a team. Therefore, certain rules of aesthetics and function must be followed. First of all, background colors on all front panels and block diagrams must be the same according to the type of VI it is (BB, SB, or CB). These are specified on the Verification Checklist in Figure 3.6. Also, all controls and indicators with similar functions have to have the same shape and color (e.g., Quit, Logout, Done, etc.).

In order to make the designer's job easier, a library of custom controls and indicators was created. This is accessed via the Controls pull-down menu. Therefore, the conventions set for that control or indicator are automatically met. This allows all front panels that have similar inputs and outputs to have the same look and feel.

As a convenience and resource for the user, the on-line help facility in LabVIEW was used extensively. Here each item in the code (control, indicator, constant, etc.) has a "Description". There-

fore, when the mouse is run over it with the Help window open, that description is displayed. There is also the "Get Info" which gives an overall program description, I/O, Author's (Authors') name(s), and Modification Log. The descriptions for the structures (such as While-loops and Case Structures) consist of the actual BDL for that structure. In addition, the BDL is available on the block diagram. It is a inserted as a free label and "shrunk" to display only the "define" statement.

The icons have to be descriptive and indicative of the function that subVI performs. These are left to the designer's discretion and creative ability. The main thing, though, is that they have to be indicative of what the program does.

All of these items are checked during the verification process. So, not only are there conventions and standards set but there is a fairly rigorous method for checking that they are met.

3.2.4 - CSE Example

This section presents a small example to illustrate the concepts discussed previously in this section. The problem is taken from the area of Numerical Methods. The designer must develop a LabVIEW program which will use Newton's Method to approximate the real solutions of Equation 3.1 [14].

$$x^3 - x - 1 = 0 \qquad (3.1)$$

Letting

$$f(x) = x^3 - x - 1, \qquad (3.2)$$

the derivative is found to be

$$f'(x) = 3x^2 - 1. \qquad (3.3)$$

Newton's Method of approximation states

$$y_{n+1} = y_n - [g(y_n)/g'(y_{n+1})] \qquad (3.4)$$

Substituting (3.2) and (3.3) into (3.4) and simplifying yields

$$x_{n+1} = (2x_n^3 + 1)/(3x_n^2 - 1) \qquad (3.5)$$

Plotting the function in Equation (3.2) produces the graph in Figure 3.7. From the plot, it is obvious that there is only one real solution and it lies between x=1 and x=2. Thus, a program must be written that will approximate that solution. Also, the first approximation is chosen as x_1=1.5. It would also have been reasonable to choose x_1=1.0 or x_1=2.0 as initial approximations.

The first step in the process is to develop the BB. This is shown in Figure 3.8. This simply defines the stimuli and responses that will be necessary for the complete program. The next step is to write the BDL which will be the building block for the SB which will in turn be the foundation for the CB. The BDL is also shown.

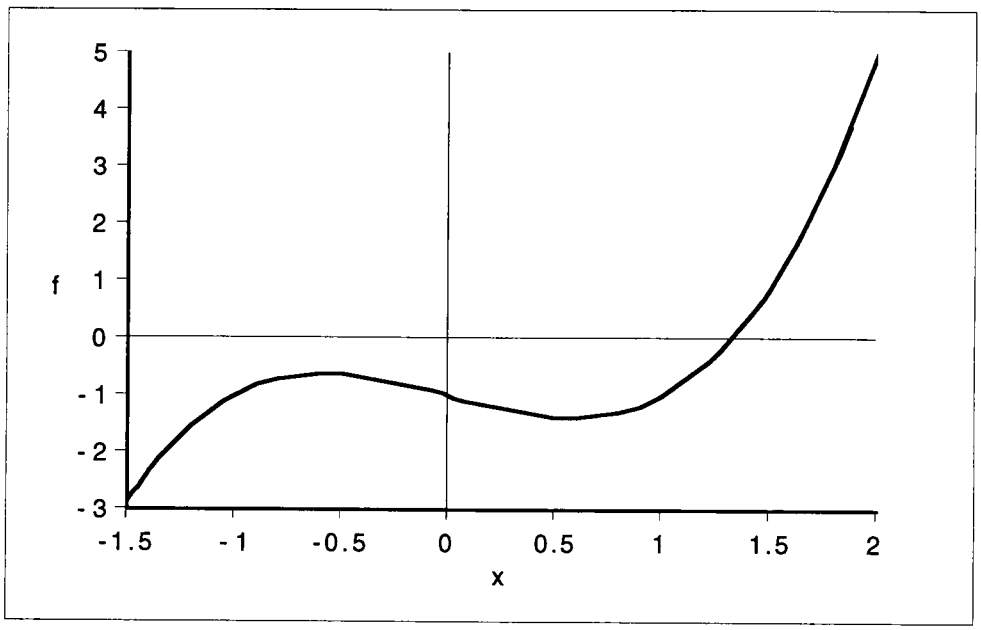
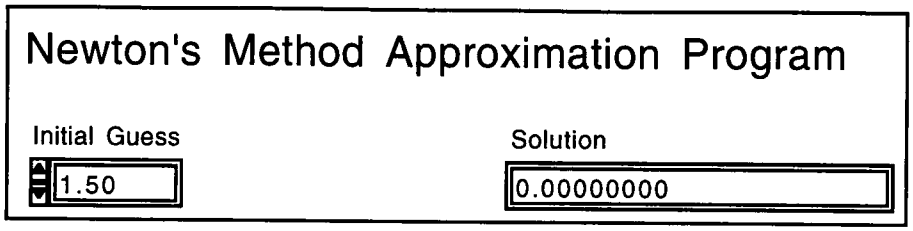


Fig. 3.7 Plot of f(x).




```
define BB Newton's Method

/*  This program was written to perform Newton's Method approximation on the example
given in subsection 3.2.4.   */

stimulus
  Initial Guess:  numeric
response
  Solution
state
  Xn:  numeric
  Xn+1 = numeric
machine
   Xn+1 = Xn - [f(Xn)/f'(Xn)], n = 1,2,3,...
end machine

proc

Xn = Initial Guess
while (Xn+1 != Xn)
do
     N(Xn) = (Xn*Xn*Xn*2.0) + 1.0
     D(Xn) = (Xn*Xn*3.0) - 1.0
    Xn+1 = N(Xn)/D(Xn)
   Xn = Xn+1
end do

solution = Xn+1

end proc

end define
```

Fig. 3.8 BB for Newton's Method example.

The SB comes next and is given in Figure 3.9. This shows the necessary elements to implement the BDL but is not the complete program. This step introduces the state machine and how it will be implemented.

Finally, the CB is shown in Figure 3.10. This represents the final program. The loop is stopped when $x_{n+1} = x_n$ for the specified digits of precision. The program yields a solution of x = 1.32471796.

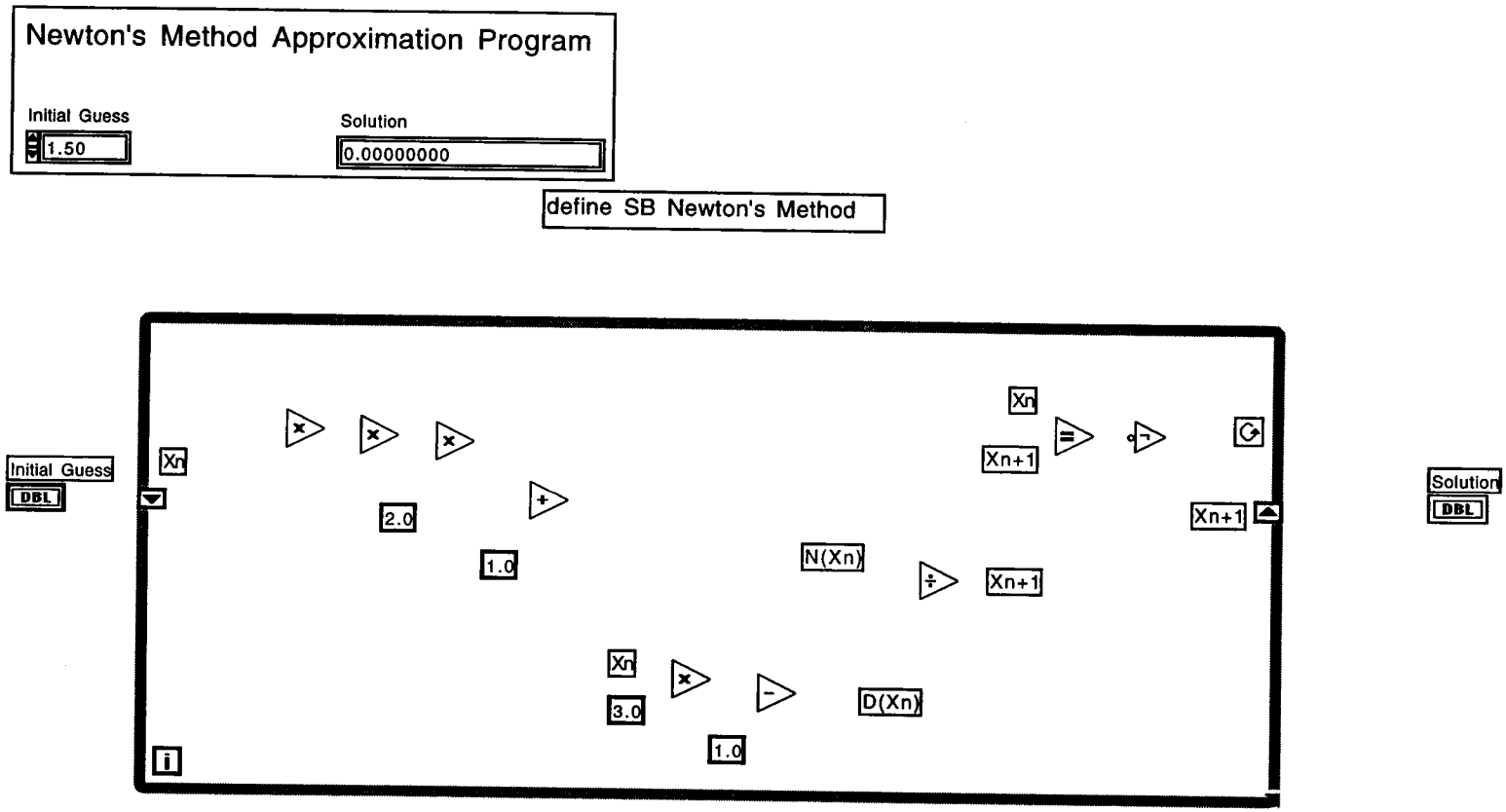


Fig. 3.9 SB for Newton's Method example.

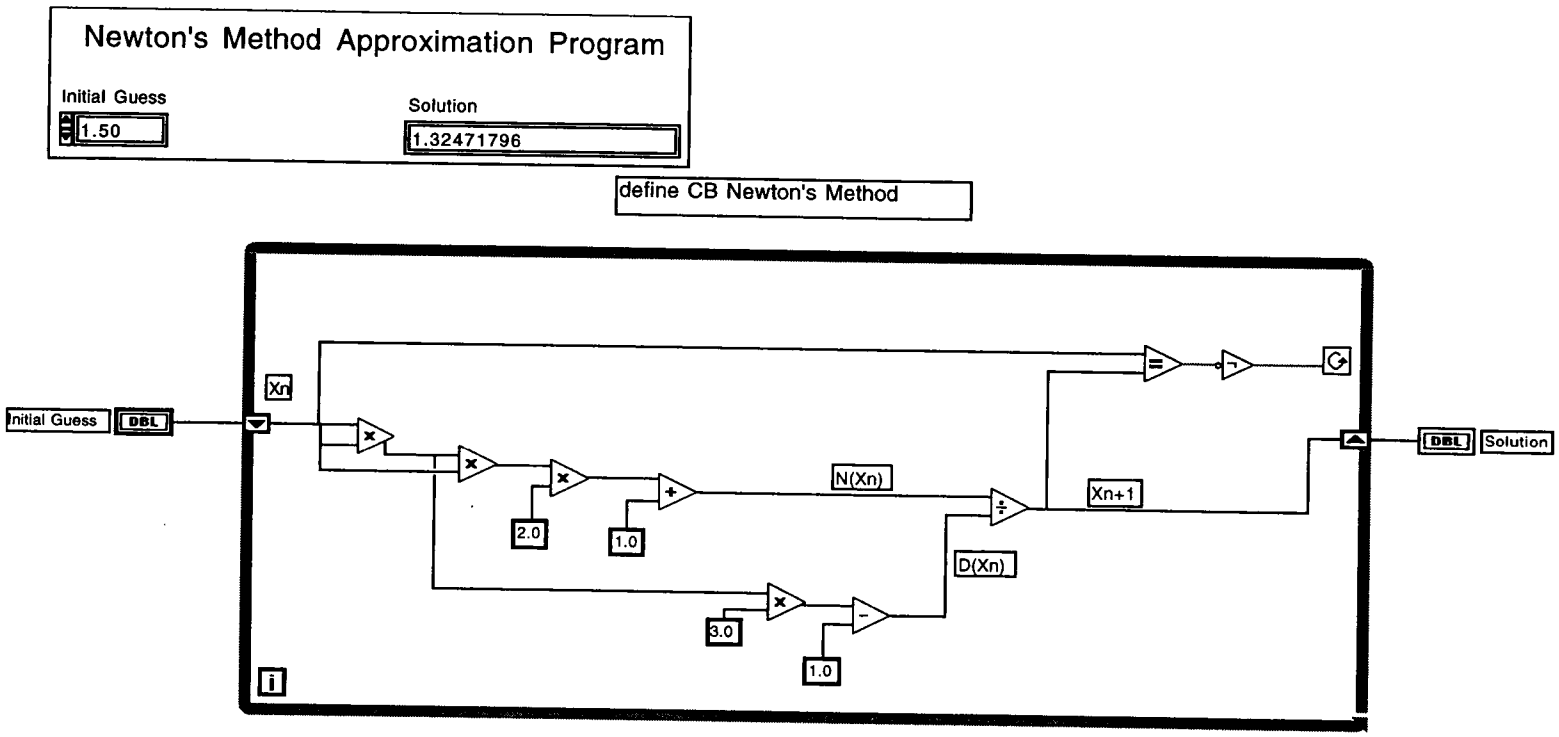


Fig. 3.10 CB for Newton's Method example.

3.3 - CSE-A

3.3.1 - Overview of CSE-A

As mentioned above, the modified version of CSE is awkward to use in a graphical environment. It is counter-productive to force graphical code development to follow a protocol designed for a text-based language. So, it became apparent that broad changes had to be made.

Consequently, a shift in philosophy was made to CSE-A. In this process, the overall idea behind CSE is maintained that the software must be developed in a rigorous fashion. However, a detour is made away from the use of BDL towards a greater use of the built-in features of LabVIEW [15].

This method is also iterative and is similar to the progression from BB to SB to CB, except each step is sequentially numbered iteration 0, iteration 1, ... Iteration 0 corresponded directly with the BB in CSE. It is important to maintain the Black Box concept. This is a nice way to include a particular subVI without requiring it to be fully developed in order to execute its calling VI. Then, instead of restricting the developer to go to a State Box and then to a Clear Box, he has the freedom to have as many iterations as necessary to produce the finished product. An analogy can be drawn

to version numbers in commercial software. Thus, when necessary the developer progresses to the next version number.

The front panel of the iteration 0 (i_0) of a VI simply has the controls and indicators necessary for it. The block diagram has BB versions of subVIs. Each of these has a description of its overall function. At this point, it is not necessary to have the inputs and outputs to these subVIs defined. The interconnectivity between them is not defined either. The purpose is to define the overall process that needs to run during the execution of this VI. Therefore, this version i_0 serves as a system-level description of the VI with BB subVIs rather than BDL in the Block Diagram.

With each iteration, the code is further refined until all of the interconnections are determined and the VI is complete. Then, the same process is followed in developing the subVIs and their subVIs, if any. Afterwards, the method is supposed to continue as before with the verification and certification steps. These had not yet been defined for CSE-A.

3.3.2 - Procedures and Conventions Used in CSE-A

As in the case of CSE, the inputs and outputs are defined first. These are placed on a LabVIEW front panel. In order to maintain the documentation and the on-line help facility, "Descriptions" are

supplied for each and the "Get Info" block is also filled in just as in the case of CSE. Up to this point, both methods are the same.

At this point, CSE-A diverges from modified CSE. In the latter case, the pseudo code would have been written and then the LabVIEW code. In CSE-A, all of the development is done in LabVIEW. First all of the major processes are defined and are represented by BB VIs. The "Get Info" of each is filled in as well so that when the mouse pointer is swept across one the Help window provides an overview of the function of that VI. In addition, any necessary case structures or loops are placed in the diagram too and are completely documented in their "Description" blocks.

Subsequent iterations are used to further define the sub-processes (subVIs) and their interconnections. Once the top level VI is well defined, then each subVI and all of its subVIs (if any) are developed using this same top down, iterative approach. Thus, the graphical LabVIEW environment is utilized to the maximum extent.

Both methods allow for modular programming. However, in using CSE-A the designer is allowed to break free from the constraints of the text-based BDL. He is able to take full advantage of LabVIEW's graphical interface. He is also able to take advantage of its on-line help facility as a means of documenting the code without the comments cluttering the code. Also, he may do this in

regular English rather than excerpts of BDL. Thus, this method is much better suited to the LabVIEW environment.

3.3.3 - CSE-A Example

This section presents a small example to illustrate the concepts discussed previously in this section. The purpose is to develop a program for a school teacher who wishes to make some calculations using the grades of her students. The program should allow selection of one of three functions: average, median, and mode. It should also sort the grades either in ascending or descending order and display them accordingly.

Using the CSE-A methodology, the i_0 version of this program must be created. This is shown in Figure 3.11. The inputs and outputs have been defined and BBs for the subroutines have been placed in the BD. Each of these has information in their "Get Info" blocks so that when the Help facility is used, that information is accessible by simply pointing to it with the mouse.

The array *Student Grades* stores the names and grades of students. The input *Function* allows selection of the type of calculation to be done. *Sort Method* allows selection of the way in which to sort the grades: ascending or descending.

There are three outputs. *Function* serves as label for *Result.*

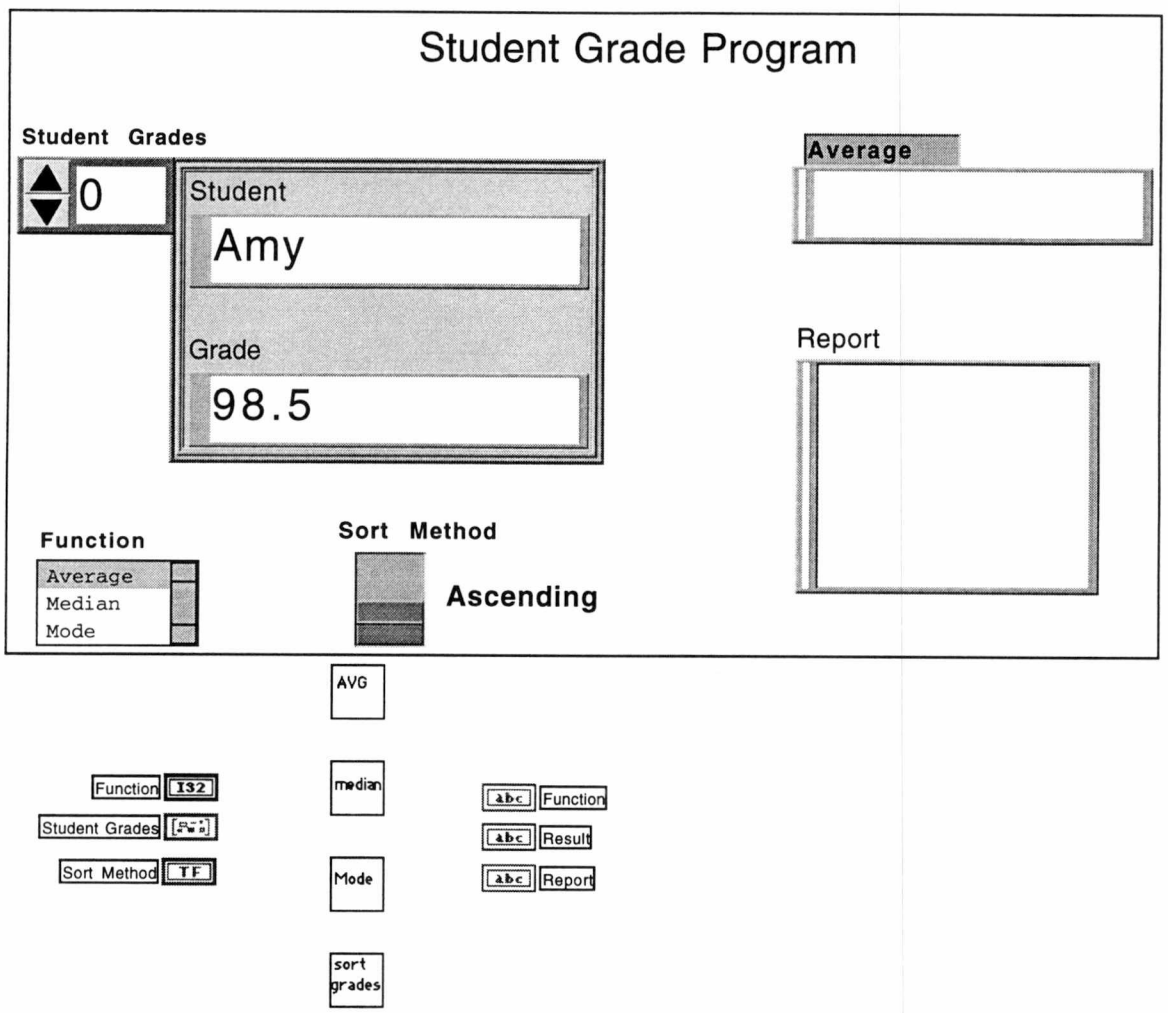


Fig. 3.11 i_0 for grades example.

That is, *Result* is used as the output indicator for all three functions. Therefore, function reads which calculation was selected and displays it as the label for the *Result* indicator. Finally, *Report* shows the ascending or descending records.

Since this is a fairly simple program, the second iteration, i_1, is the final one. This is shown in Figure 3.12. The list box *Function* controls a case structure. For it, a "Description" has been entered stating what occurs in each case.

It should be noted that the values in *Result* and *Report* are merely for example. The subVIs shown do not actually compute anything. However, this example does illustrate the CSE-A methodology. These subVIs would be developed next and then the current ones would be replaced by the final versions of those.

3.4 - Driver Design Methodology

3.4.1 - Overview

The ATF was conceived with the notion of automating the testing procedure. A key requirement was that the hardware be computer controllable. Consequently, most of the pieces were selected to meet this requirement via GPIB or RS-232. In order to control a given instrument via the computer (controller) an

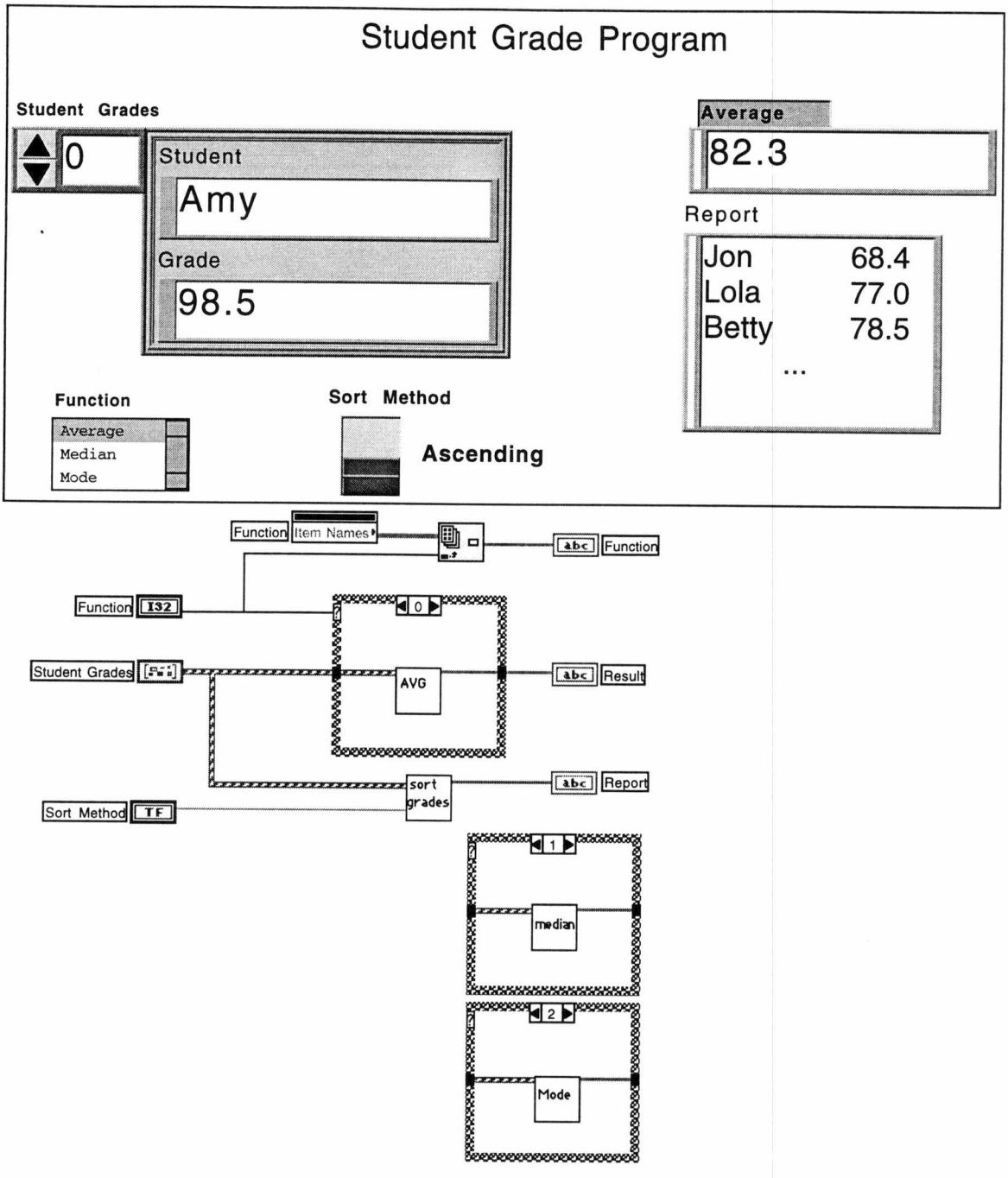


Fig. 3.12 i_1 for grades example.

instrument driver is needed for it. An instrument driver is a collection of software programs that allow the user to programmatically control a piece of hardware.

The methodology that was employed to design the Ghost Simulator driver (Chapter 6) was developed by National Instruments (NI). It is a proven and well-tested instrument driver development process. It too calls for a modular approach to design drivers. In this manner, the driver is divided into similar functions that are developed together [16, p.2-4].

The internal design model of a LabVIEW instrument driver is shown in Fig. 3.13 [16, p.2-5]. As seen, the two main divisions are analogous to axioms and theorems in mathematics. Axioms are the fundamental building blocks upon which theorems are built. Similarly, component VIs are VIs that control basic configuration and measurement functions of an instrument. Then, application VIs are combinations of these which perform the highest level functions.

NI has found that many of the component VIs are necessary for all instrument drivers. Therefore, NI created what they call template VIs. These perform common functions such as initialization, reset, self-test, etc. So, a designer may use these with minor modification. What will remain are VIs that the designer must develop. These are appropriately called developer specified VIs.

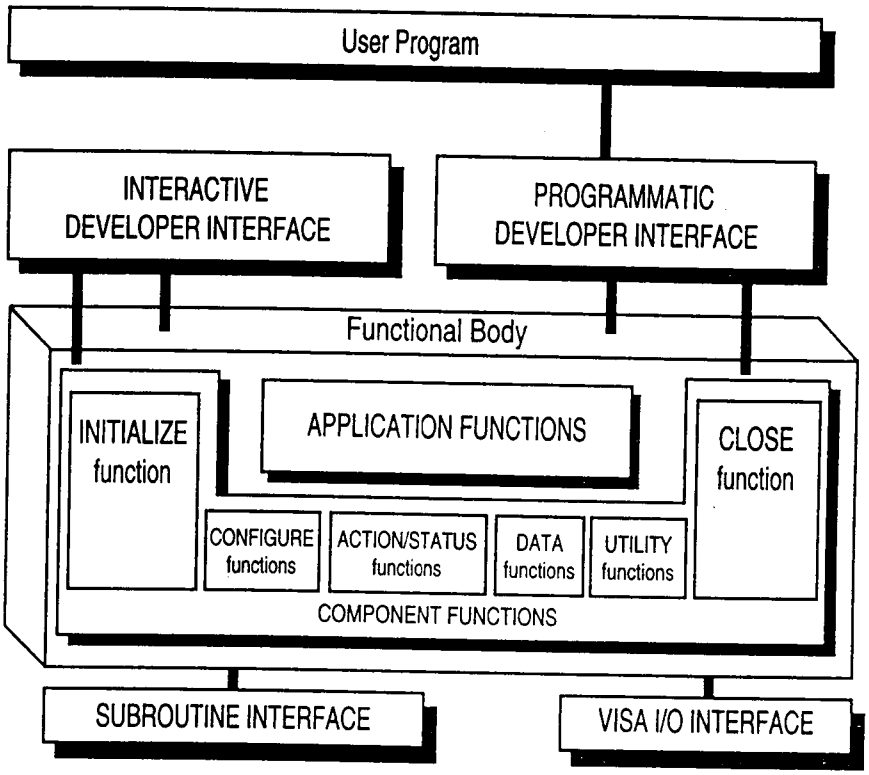


Fig. 3.13 Internal design model of a LabVIEW instrument driver.

These VIs are the ones that are specific to the particular instrument.

The LabVIEW instrument driver templates are listed in Table 3.2 along with their descriptions [16, pp.2-11 - 2-13].

3.4.2 - Development Methodology

It is important first to develop the structure for the driver. In order to do this several steps must be performed [16, p.3-2]. First,

Table 3.2 LabVIEW Instrument Driver Templates.

Name	Description
PREFIX Init	This is the first VI called when accessing an instrument driver. It configures the communications interface, manages handles, and sends a default setup command to the instrument.
PREFIX Close	This is the last VI called when controlling an instrument. It terminates the software connection to the instrument and deallocates system resources.
PREFIX Reset	This VI places the instrument in its default state.
PREFIX Self Test	If an instrument has self-test capability, this VI will cause it to initiate that self-test and returns the result.
PREFIX Revision Query	This VI outputs the revision of the driver and the firmware revision of the instrument. If the instrument firmware revision cannot be queried, the literal "Not Available" should be returned.
PREFIX Error Query	This VI queries the instrument and returns the instrument-specific error information. This VI can only be used if the instrument has error query capability.
PREFIX Error Message	This VI translates the error status information received by the Error Query VI.
PREFIX Message-Based Template	This VI is the starting point for the development of the developer specified (GPIB) instrument driver VIs. This VI has all required instrument driver controls and instructions.
PREFIX Register-Based Template	This VI is the starting point for the development of the developer specified (VXI) instrument driver VIs. This VI has all required instrument driver controls and instructions.

the designer must familiarize himself with the operation of the instrument. This is done by thoroughly reading the manuals and interactively using the instrument before any programming is done. This includes using the instrument in actual testing.

Second, the programmer's manual must be studied. It should be determined what functions are available to be executed remotely. In addition, it must be determined which are best suited for programmatic implementation.

Third, if instrument drivers are available for similar instruments or instruments in the same family, these should be studied. It may be possible to modify these or adapt them for use in the current driver.

Fourth, a determination must be made as to which template VIs may be appropriate for use in the driver. This is a fairly straightforward and systematic thing to do and includes actually modifying these VIs.

The last step requires the most developer input. Here the structure for the driver must be defined. This is done by looking for operations that are used together to perform a single task. Generally, the manual is an excellent resource for this. The instrument manufacturer has already grouped similar functions and applications in the manual. So, these groupings can be used to

delineate the various parts of the driver. The main idea is to maintain modularity and an appropriate level of granularity. These VIs should not be too high-level else they will not offer adequate control of the instrument. Nor should they be too low level else they will be tedious and complicated to use.

There are two sets of controls and indicators which should appear on every front panel. These are *instr handle in, instr handle out, error in (no error),* and *error out. Instr handle in* is used as a unique identifier for a device I/O session. It is used as the device ID in all communications with the device.

Regarding controls, it is best to use the default font (application) for all control labels. This makes portability between platforms much easier. The primary controls should be denoted with bold text and secondary controls with regular text. Both should be spelled in lower case lettering (except abbreviations such as GPIB). The default value should be included in plain text as part of the label so that it can be accessed through the help window when that VI is used as a subVI for a higher level VI.

There are a few other tips to keep in mind. The block diagram should be neat, uncluttered, and well documented by using free labels and the "Description" blocks for structures and constants. All elements should be evenly distributed with minimal bends in the

wires. The icon should be meaningful and indicative of the VI function. The connector pane should have *instr handle in* and *instr handle out* in the upper left and right, respectively; and it should have *error in* and *error out* in the lower left and right, respectively.

3.4.3 - Driver Example

A driver must be written for a computer-controlled spectrum analyzer which can measure modulation depth, carrier levels, and carrier frequencies.

After becoming familiar with the hardware and studying the manual, the structure of the driver may be defined as in Table 3.3. Then, the designer would proceed with the actual coding of the driver.

Table 3.3 Hierarchy for Spectrum Analyzer Driver.

VI Hierarchy	Type
Initialize VI	Template
Application VIs	
*Modulation Depth	Developer Defined
*Carrier Levels	Developer Defined
*Carrier frequencies	Developer Defined
Configuration VIs	
*Auto setup	Developer Defined
*Configure for signal measurements	Developer Defined
*Configure for carrier measurements	Developer Defined
Action VIs	
*Make measurement	Developer Defined
*Check device status	Developer Defined
Data VIs	
*Send Channel no.	Developer Defined
Utilities VIs	
*Reset	Template
*Self-Test	Template
*Error Query, Message	Template
Close VI	Template

Chapter 4

Algorithm for the RF Sensitivity Test

4.1 - Introduction

The purpose of the RF Sensitivity Test is given in the Functional Specifications. It is to determine the smallest incoming RF signal that will give a "barely perceptible presentation" ("just" discernible numbers on the Monoscope pattern and loss of color on the Color Bars pattern) on the video display device, and the smallest value of signal level where the sound from the system is "just noise-free". Basically, the purpose is to determine the lowest signal levels that will yield "acceptable" viewing and listening conditions.

The system hardware configuration is given in Figure 4.1 [1]. Three main measurements are made on channels 2, 5, and 12. These are video, audio, and color sensitivity. Only two measurements are made on channels 14, 45, and 69. These are video and audio.

In the video measurement, standard Monoscope pattern with 87.5% modulation depth and a monotone, 1000 Hz, 25 kHz deviation audio signal must be routed to the device under test (DUT). Then, with the DUT muted, the signal level is decreased (i.e., increased attenuation) until the numbers on the Monoscope are

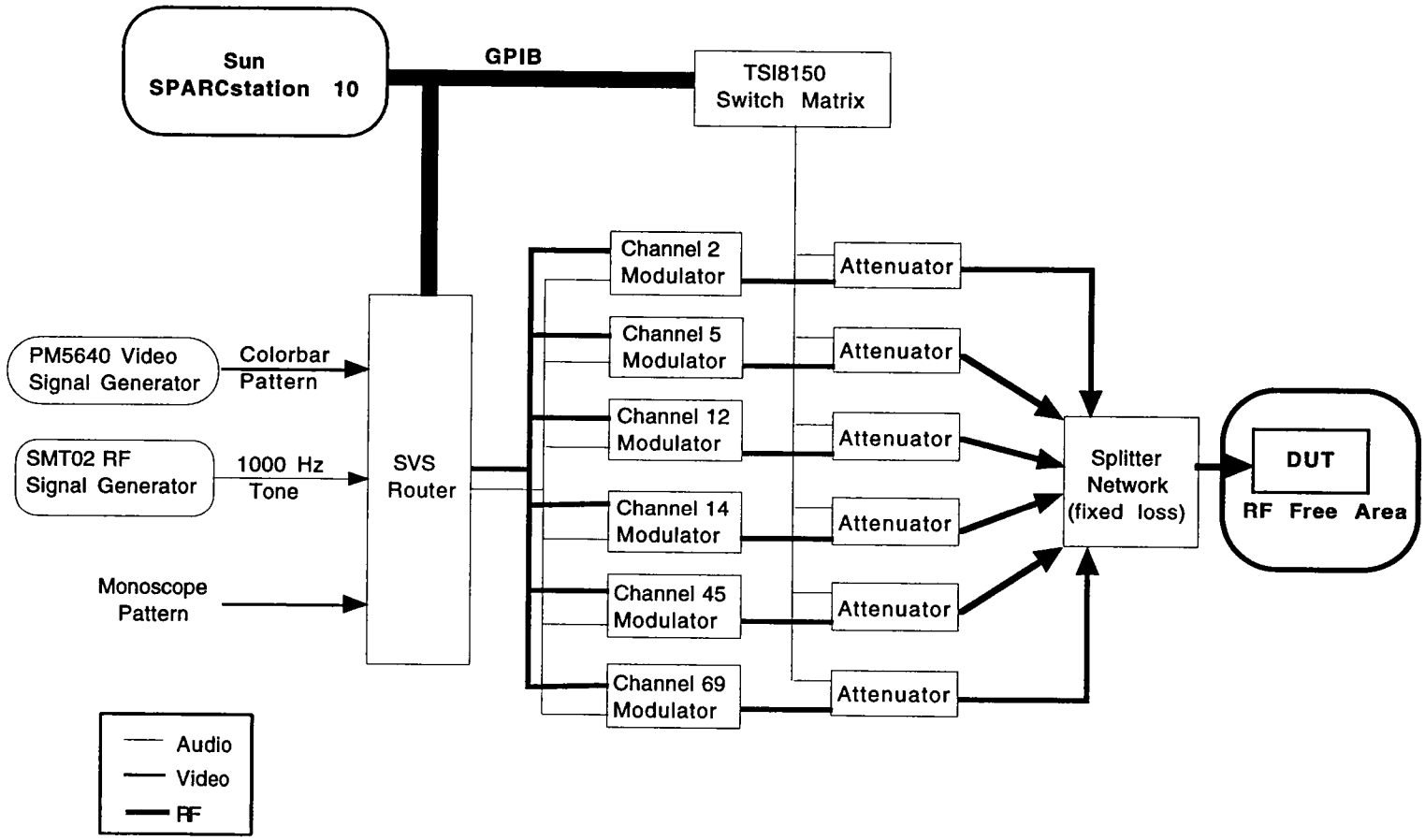


Fig. 4.1. RF Sensitivity test configuration.

"barely perceptible". This is recorded as the video sensitivity level. During the audio sensitivity measurement the DUT is not muted. Then, the signal level is decreased until the audio is "just noise free". This is recorded as the audio sensitivity level. Finally, for the color sensitivity, a 100 IRE standard Color Bars pattern must be routed to the DUT and the receiver should be muted again. The signal level is decreased until there is a loss of color. It is then increased until the color "just" remains stable. This is recorded as the color sensitivity level.

The entire ATF is set up in a menu driven hierarchy wherein the user starts from the Top Menu. Then, he may work his way down through the different paths as shown in Figure 4.2. The RF Sensitivity Test is found in the following manner. From the Top Menu, *Competitive Analysis* should be selected. Then, the user must log in at the Login Menu. Next the list of devices to be tested are entered in the *TV Header*. After which the Category Menu opens. Here the *RF Series* should be selected. Finally, from there *RF Sensitivity* should be selected to invoke the test.

The remaining sections and subsections discuss the software that was written to implement this test. Section 2 discusses Error Handling. This contains the interface to the Built-In Tests (BIT) that are to be run during off-line hours and are supposed to perform

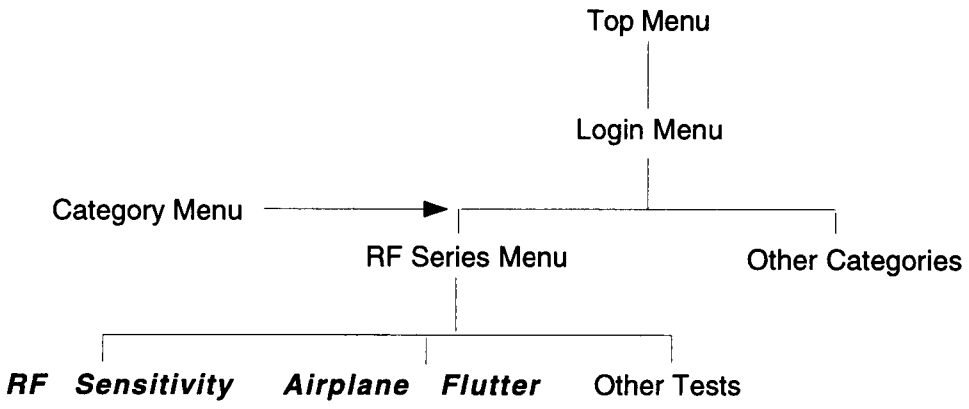


Fig. 4.2 ATF menu hierarchy.

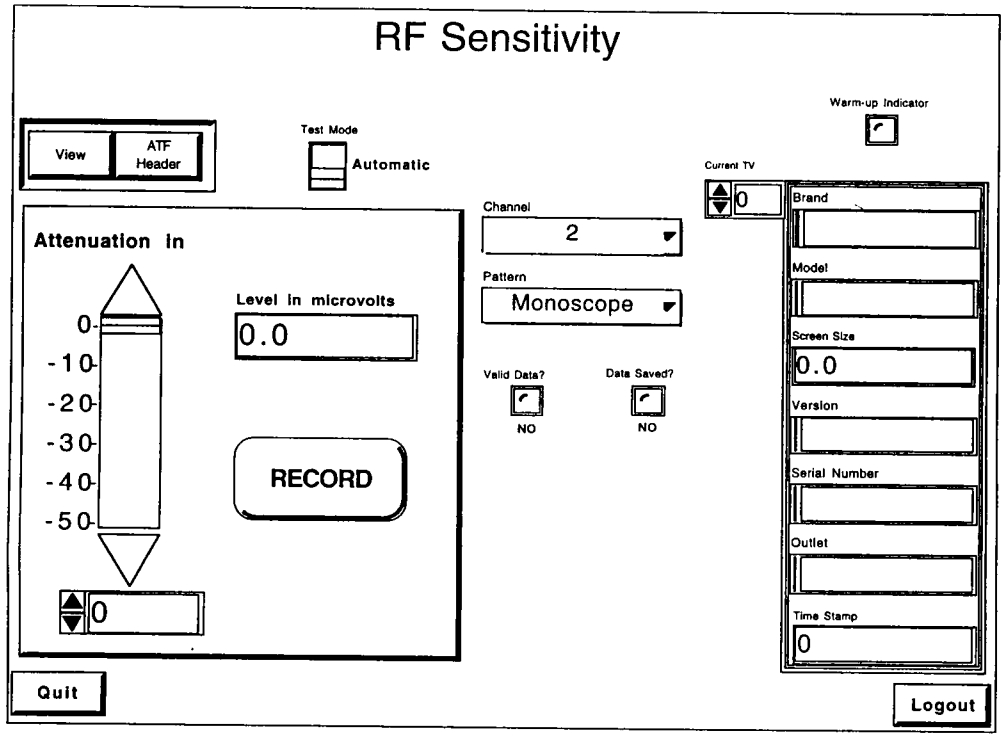
diagnostic and reliability measurements. Resetting and Initialization are the subject of Section 3. Presetting is discussed in Section 4. The test process itself is discussed in Section 5.

4.2 - Error Handling

The entire RF Sensitivity Test is shown in Figure 4.3 wherein (a) shows the front panel (FP) for the VI and the subsequent figures contain the block diagram (BD). The first step is sequence 0, the Error Handling stage. In this stage the results of the Built in Tests (BIT) are checked to determine whether or not the test can indeed be performed. As noted earlier, the BIT (when developed) will be run during the off hours and its purpose will be to determine if the equipment is functioning properly and if the system as a whole is functional.

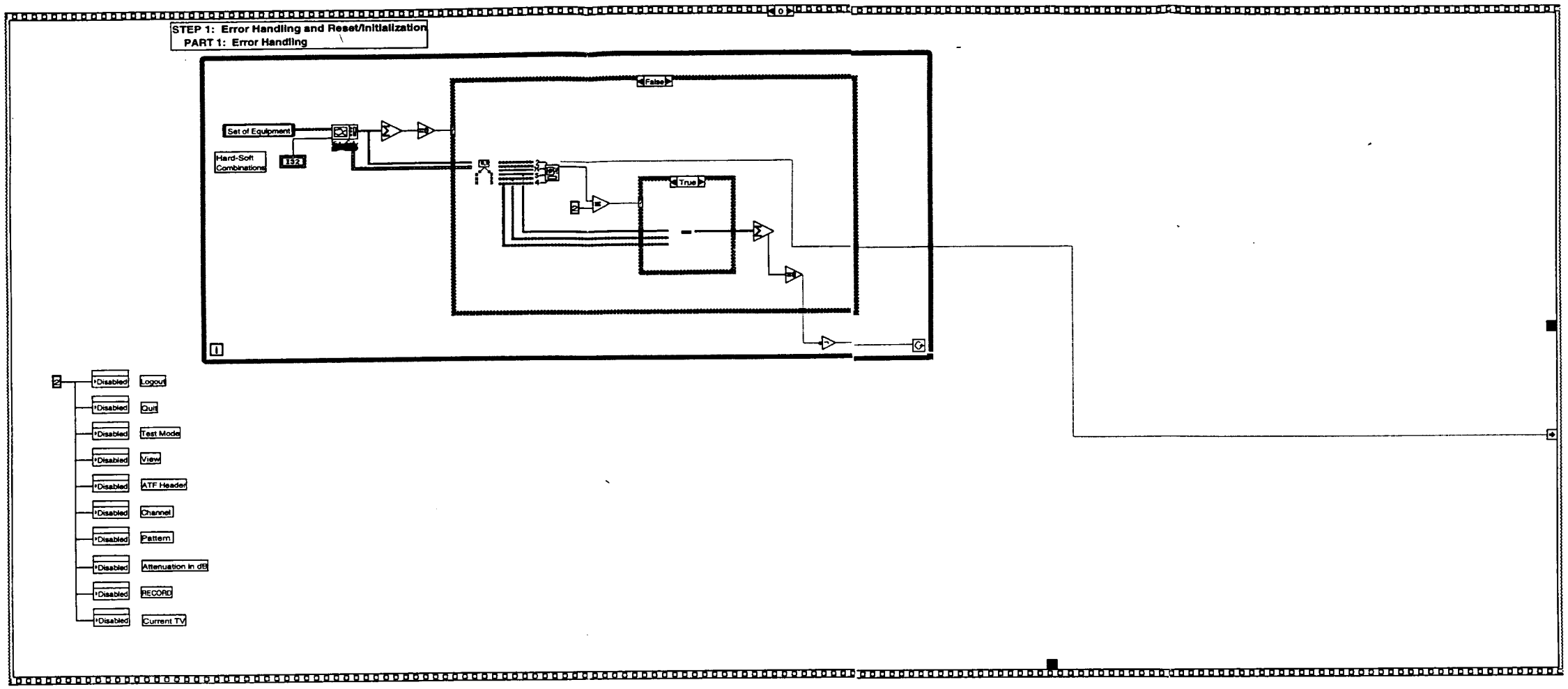
During this stage (and during Reset, Initialization, and Preset), all of the controls and indicators are grayed-out and disabled to indicate to the user that these process are occurring. Since these require several seconds to execute, the user must not be able to manipulate the FP controls. This is the reason for all of the "Disabled" attribute nodes in the lower left of the BD.

The first step is to provide the **eevtlg_c.vi** (see Figure 4.4) with the *Set of Equipment* (see Figure 4.5) global variable which



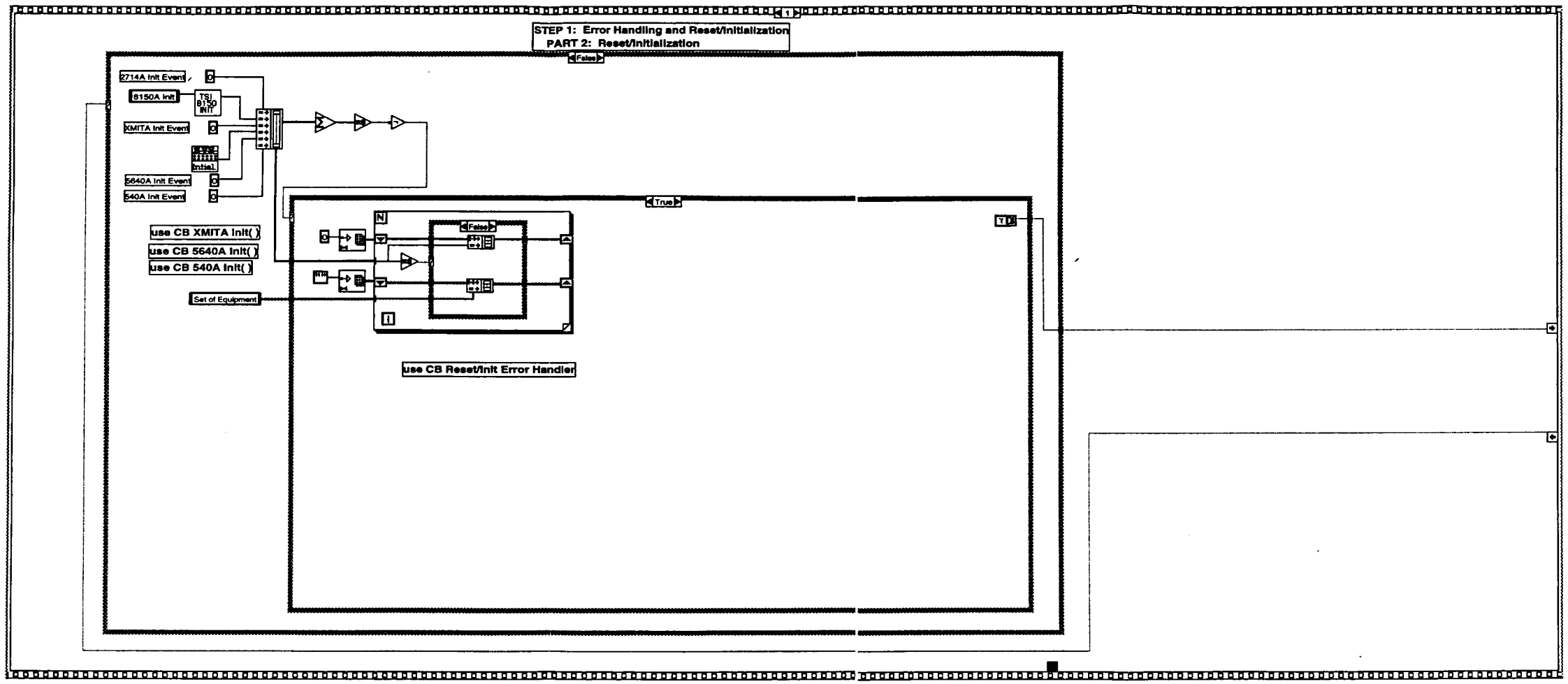
(a) Front panel.

Fig. 4.3 RF Sensitivity VI.



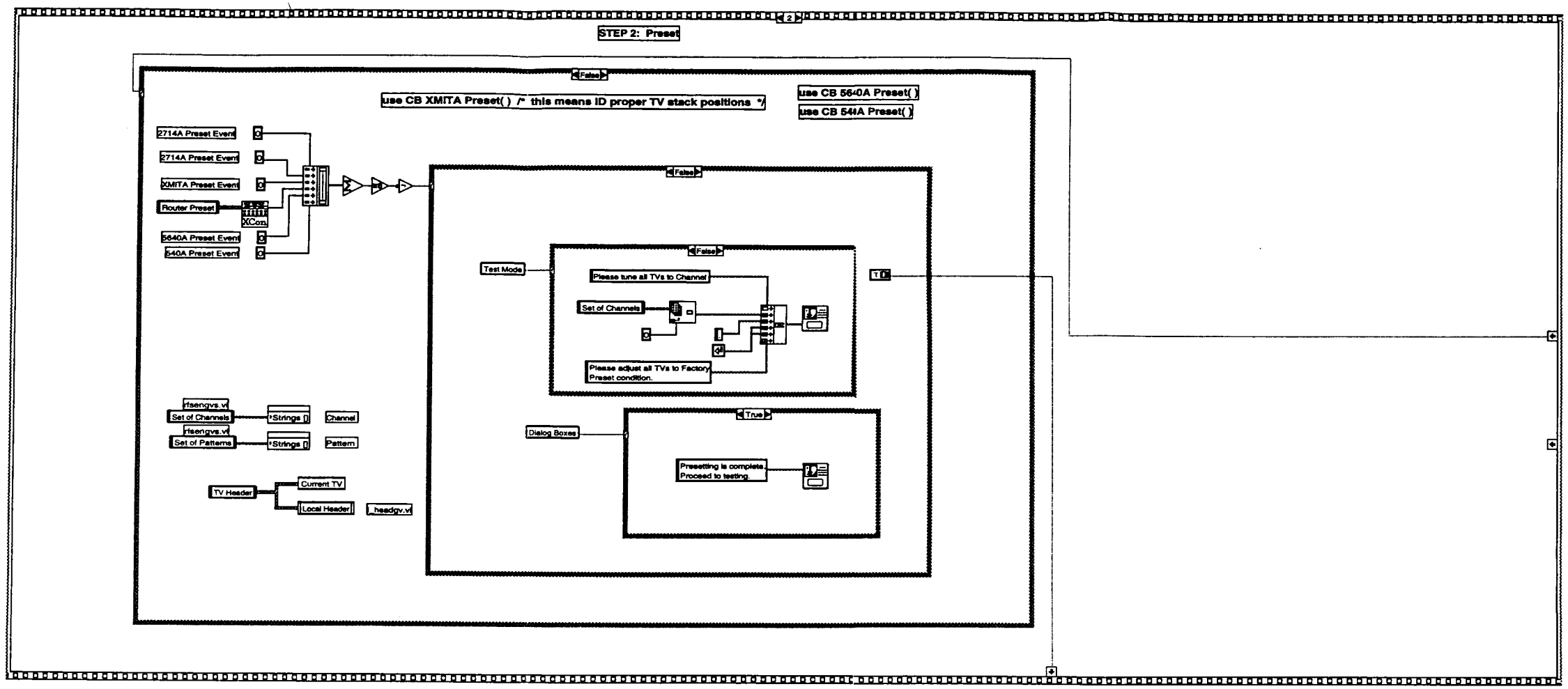
(b) Block diagram.

Fig. 4.3 (continued)



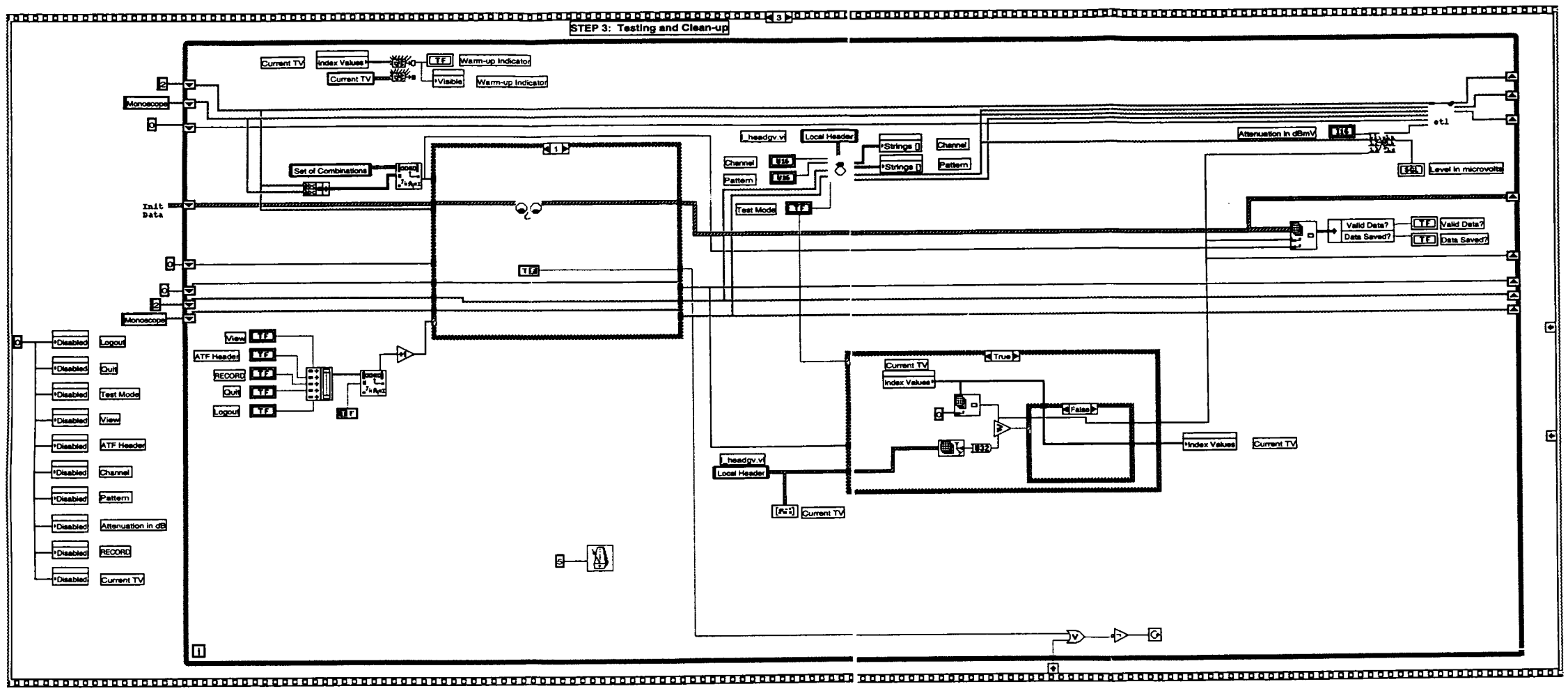
(b) (continued)

Fig. 4.3 (continued)



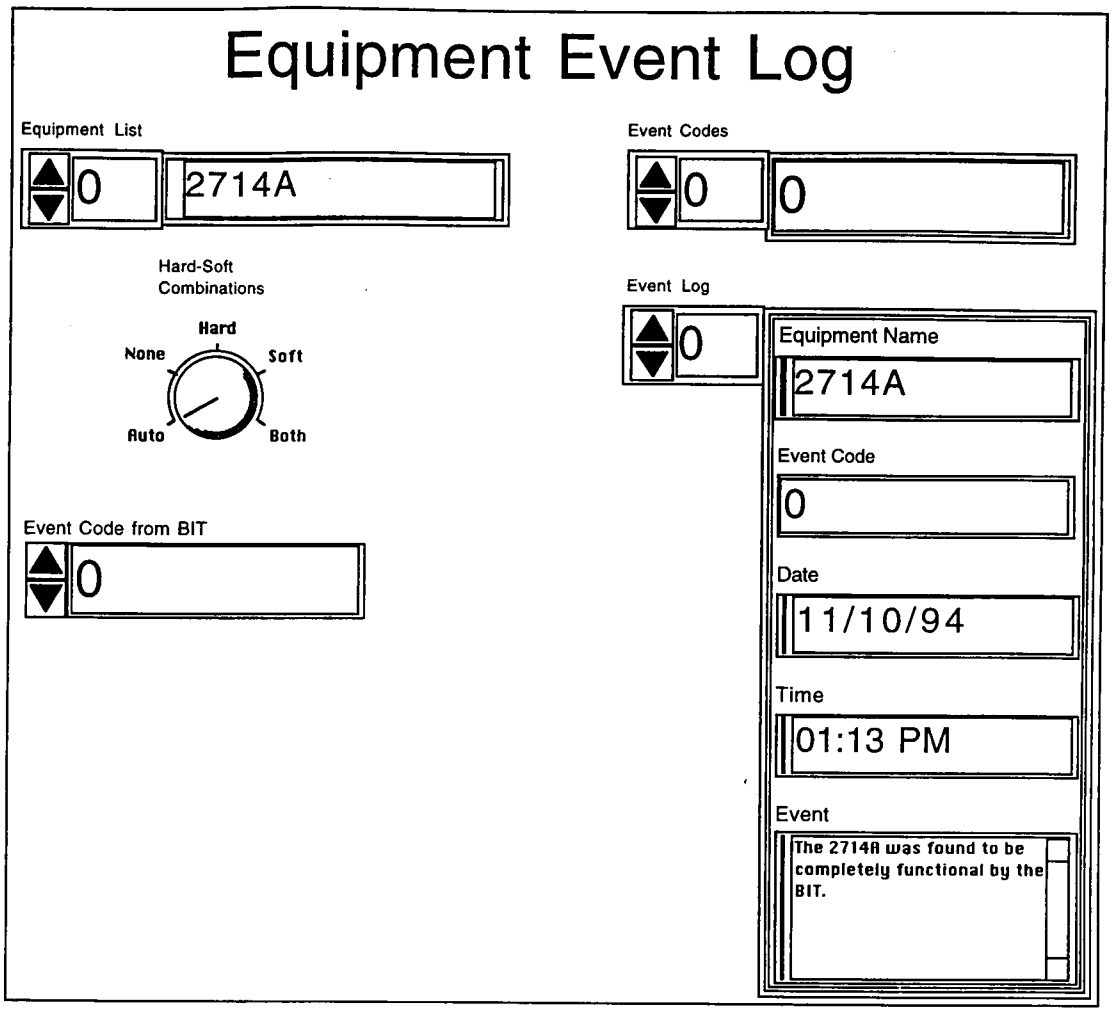
(b) (continued)

Fig. 4.3 (continued)



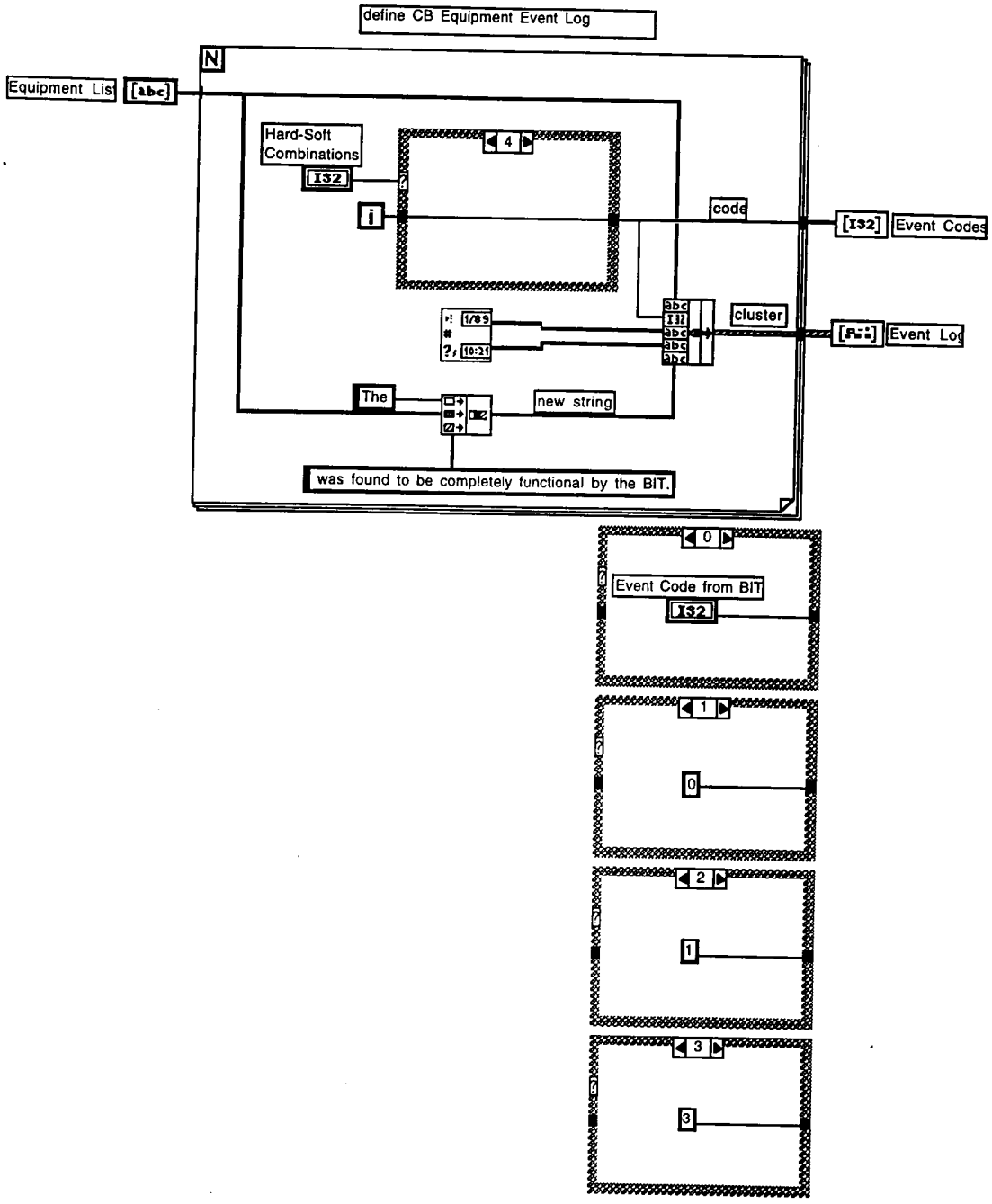
(b) (continued)

Fig. 4.3 (continued)



(a) Front panel.

Fig. 4.4 Event logger VI.



(b) Block diagram.

Fig. 4.4 (continued)

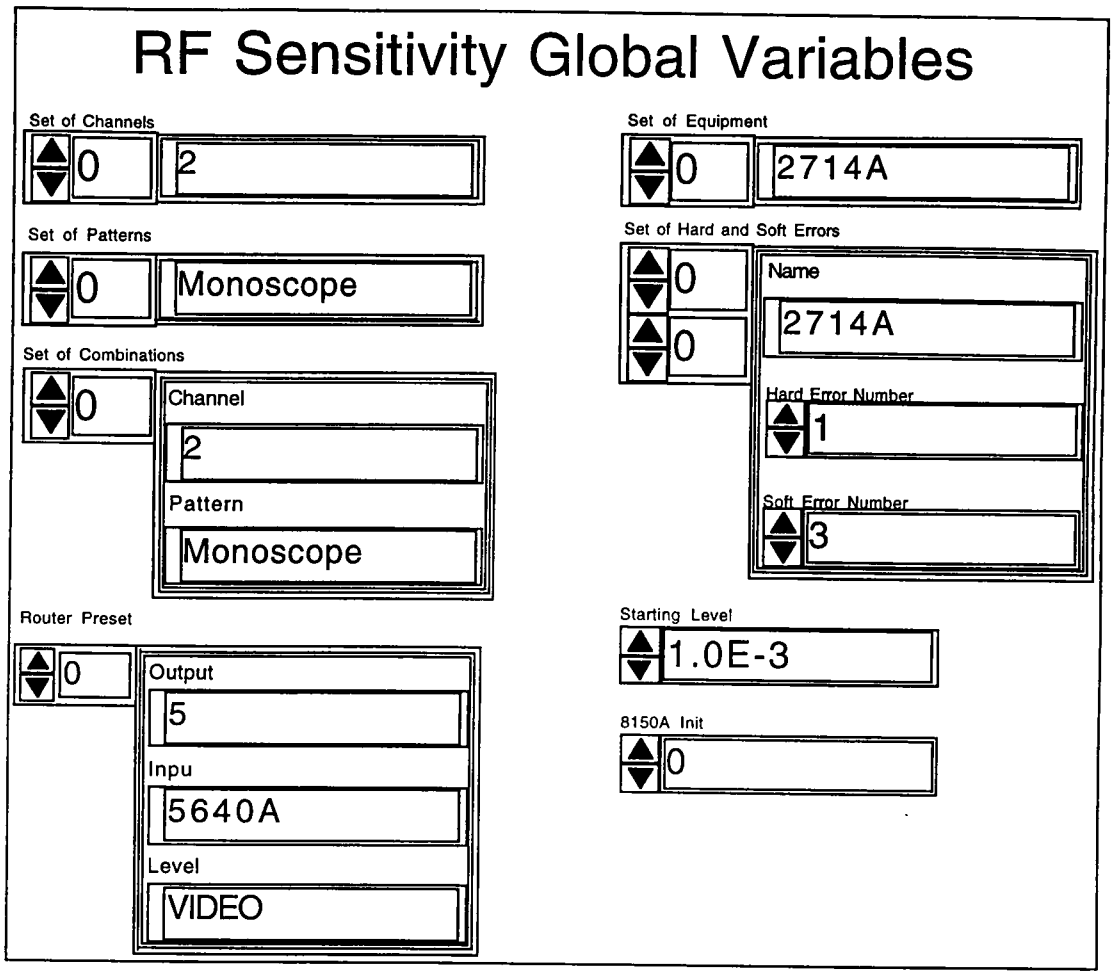
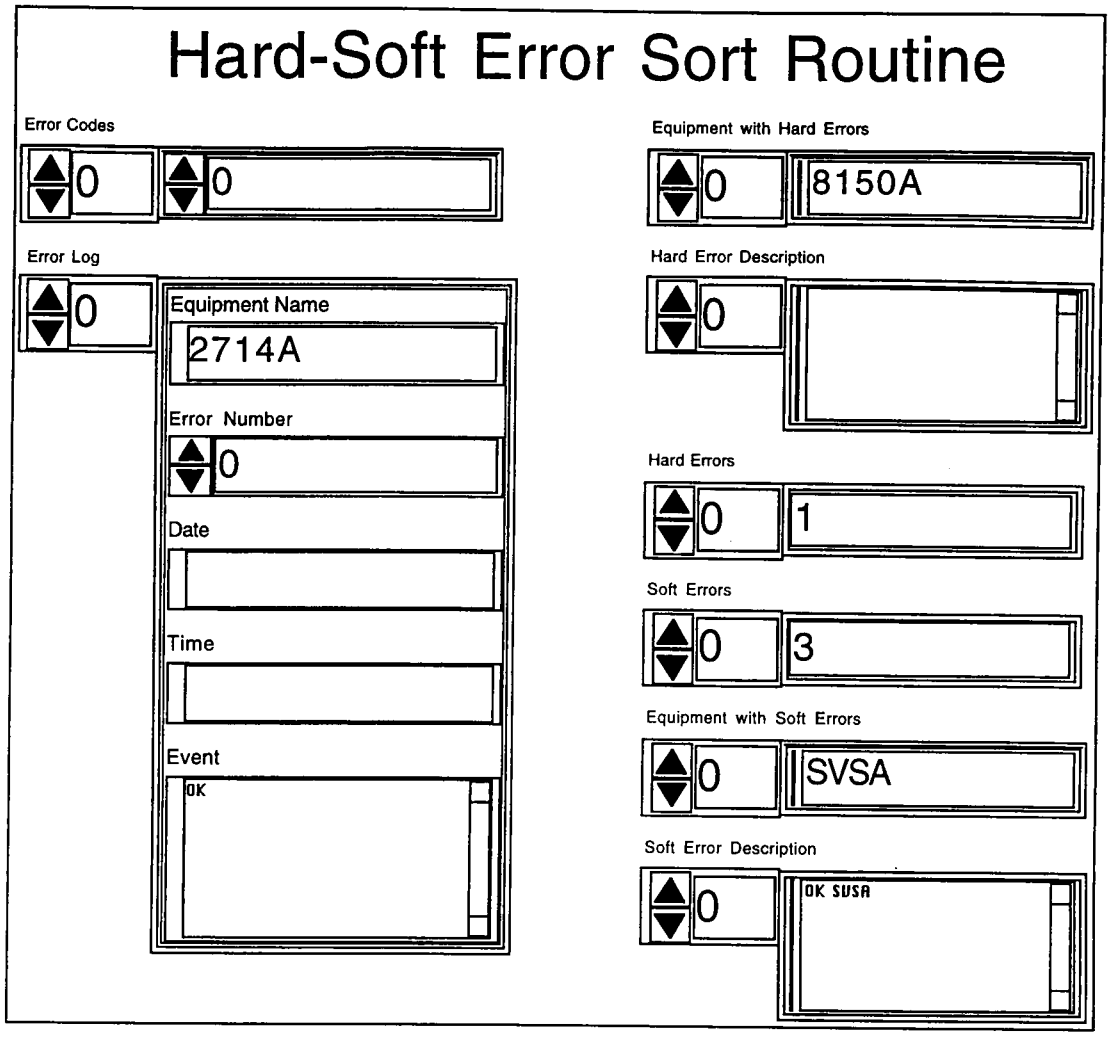


Fig. 4.5 RF Sensitivity global variables.

contains the equipment needed to perform this test. This VI was written to find equipment errors and problems as determined by the BIT. It is to be polled by the respective tests to determine if any of the equipment listed will cause problems for that test. This VI takes the *Set of Equipment* and searches the BIT global variables (to be designed later) and outputs the status of each piece of hardware. It does this in the *Event Codes* array and in the *Event Log* array. The *Event Codes* array is an array containing all of the event codes for all of the equipment in the list. These also appear in the *Event Log* array and have been duplicated for programming considerations. The *Event Log* array contains clusters for each piece of equipment. Each cluster contains the *Equipment Name, Event Code, Date, Time,* and *Event* message. It should be noted that the *Hard-Soft Combinations* input is strictly for programming purposes since the BIT was not yet developed.

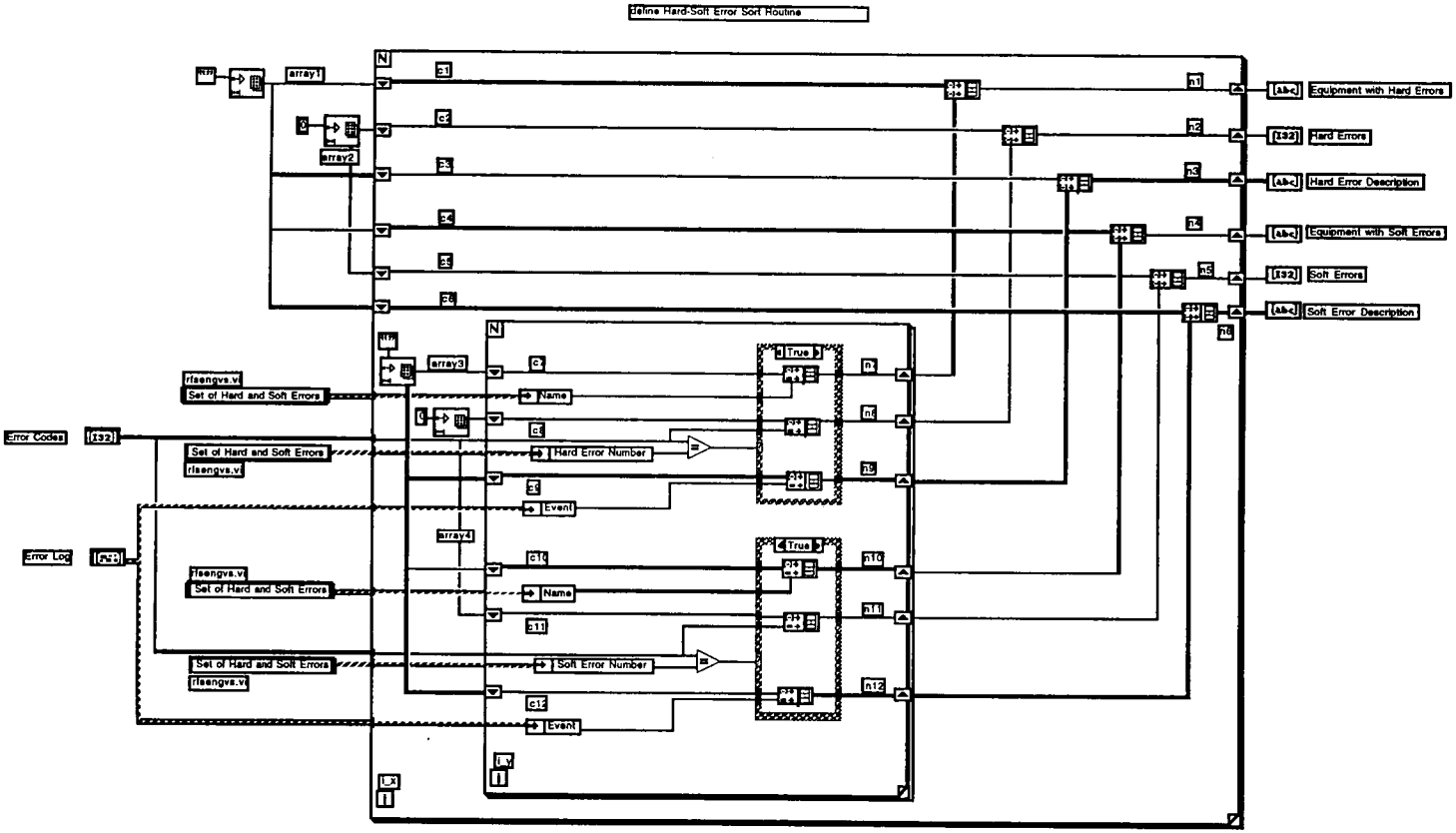
As a convention, an event code of zero (0) is defined to mean "no error". So, all of the event codes are checked via the sum array function and if that sum is zero, then there is no error and a boolean false is output for the continuation signal that is passed out through the right side of the sequence structure.

If there are errors, then **ersort_c.vi** are invoked (see Figure 4.6). This VI sorts out those errors into two kinds: hard and soft.



(a) Front panel.

Fig. 4.6 Error sorter VI.

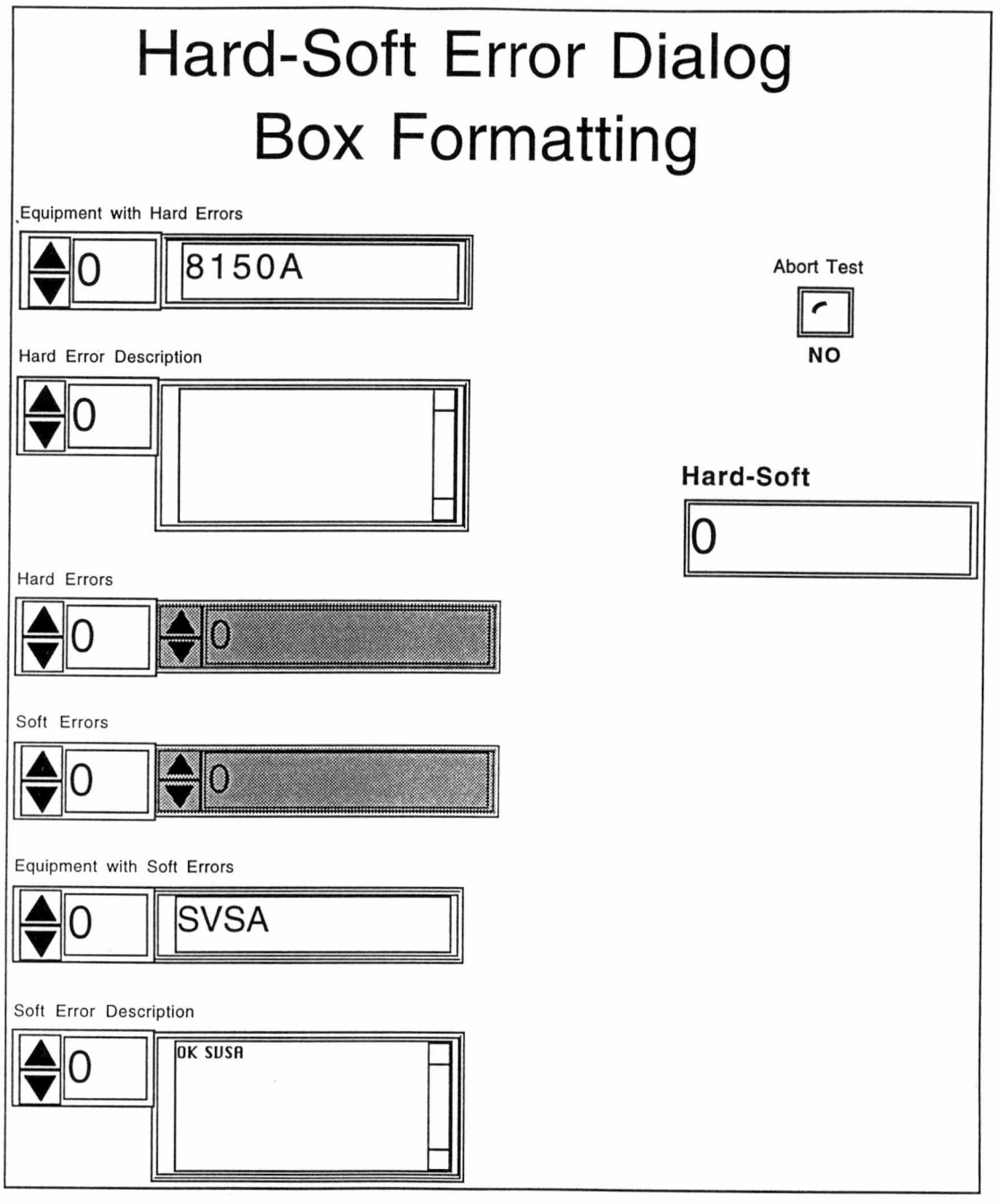


(b) Block diagram.

Fig. 4.6 (continued)

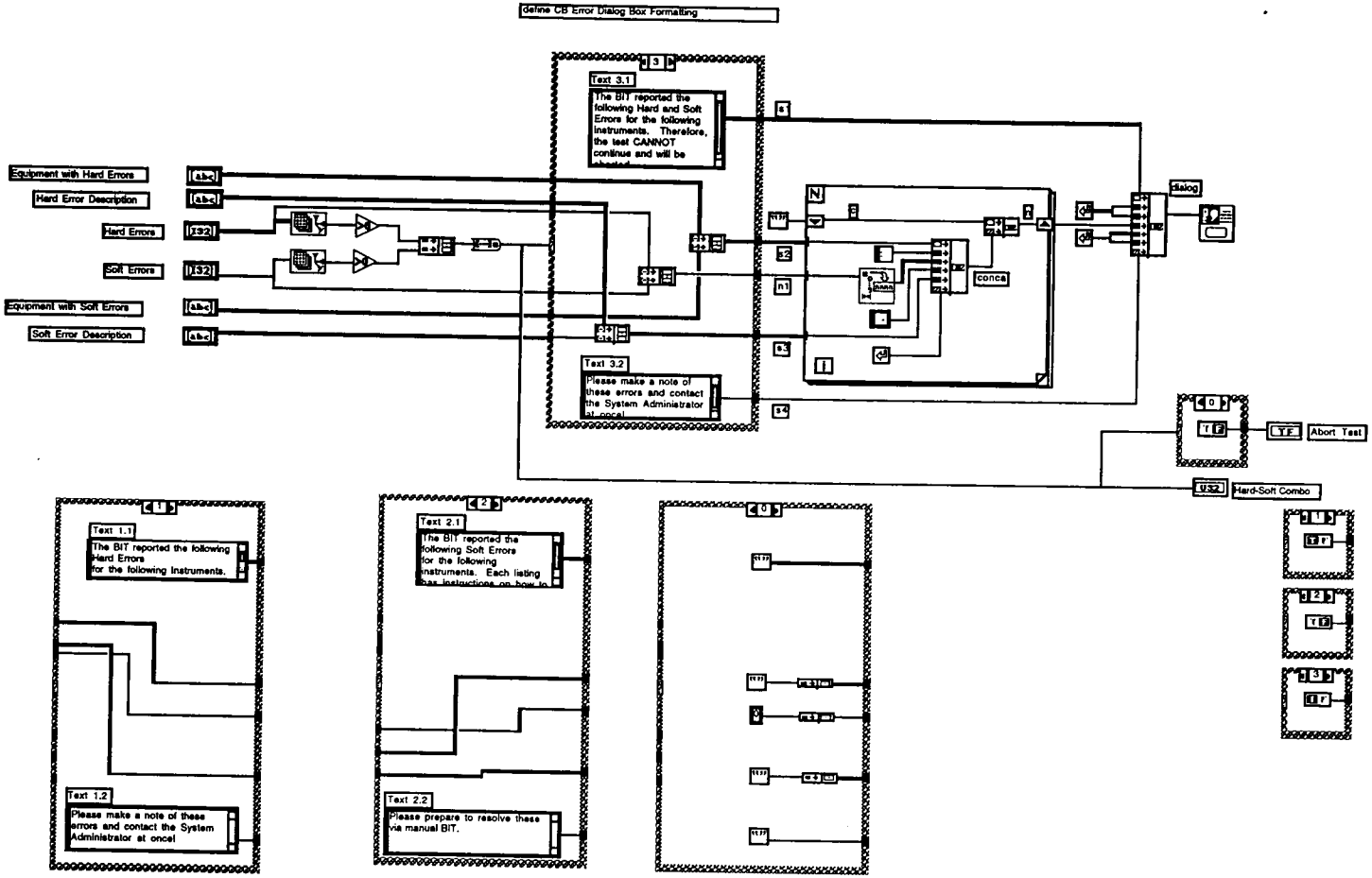
Hard errors are those that will not allow the test to be run. If an instrument is not responding at all to any kinds of polls, that would be a hard error. Soft errors are those that will allow the test to run but may produce invalid data. If an instrument is out of calibration, that would be a soft error. The *Error Codes* and *Error Log* produced by the **eevtlg_c.vi** are fed into this VI. Then, it produces a list of *Equipment with Hard Error*, *Hard Errors*, *Hard Error Description*, *Equipment with Soft Errors*, *Soft Errors*, and *Soft Error Description.* It does this based on a list of Hard and Soft error numbers which are stored in the global variables *Set of Hard and Soft Errors* (Figure 4.5). It compares the *Error Codes* supplied to the ones in that set and determines which errors are hard and which are soft. In addition, it parses the message texts from the *Error Log* and produces descriptions for the corresponding hard and soft errors.

All of the outputs of the **eevtlg_c.vi** are fed into the **errrpt_c.vi** (see Figure 4.7). This VI's purpose is to produce various text combinations which are fed into a dialog box. This text reports the equipment in which the errors occurred, the error numbers, and a brief description. If there are hard errors, then *Abort Test* will be true; otherwise, it will be false. Also, this VI outputs the *Hard-Soft Combo* which defines the types of errors that occurred according to the convention: 0 = no errors, 1 = hard errors



(a) Front Panel.

Fig. 4.7 Error Reporter VI.

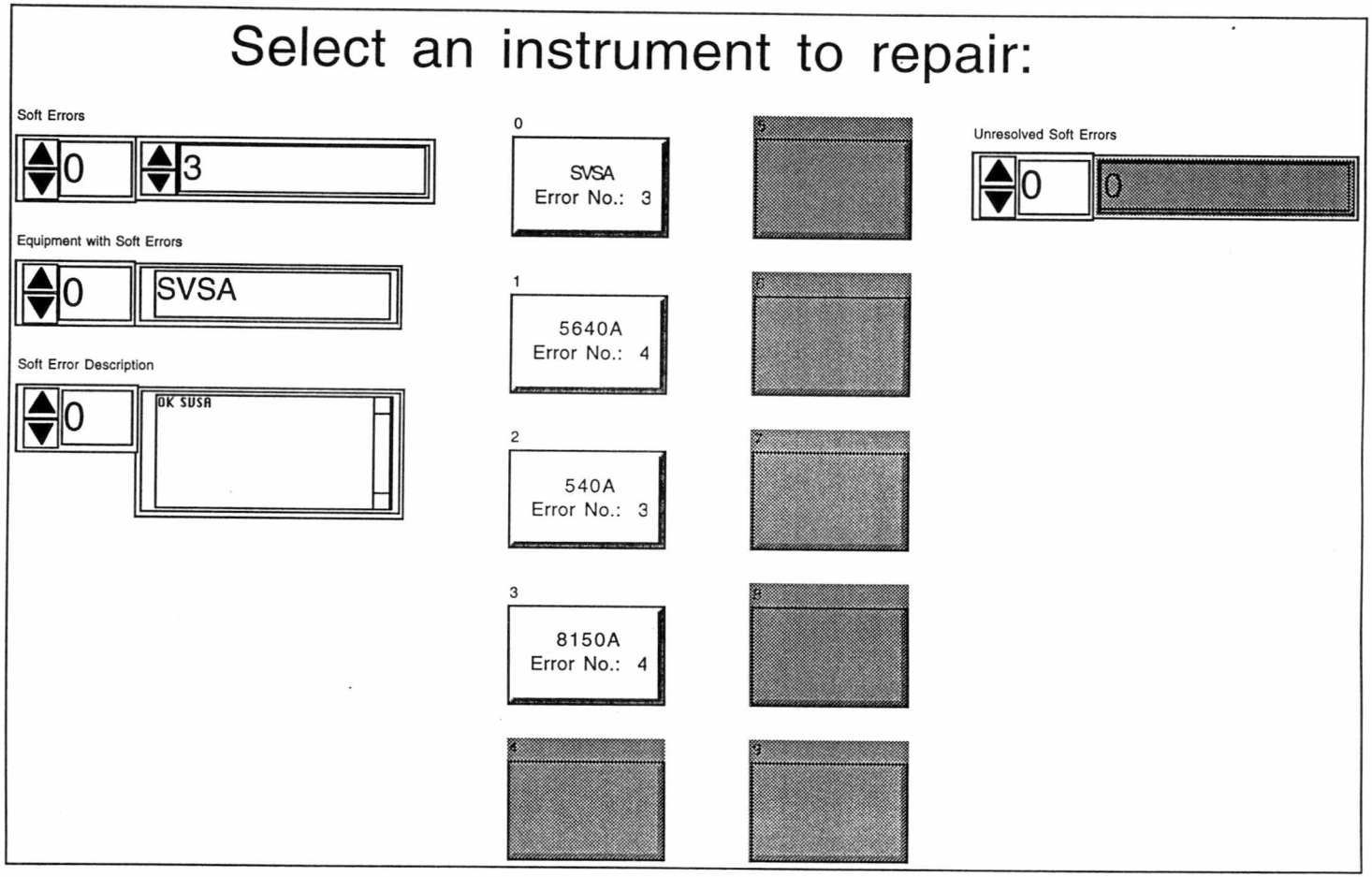


(b) Block diagram.

Fig. 4.7 (continued)

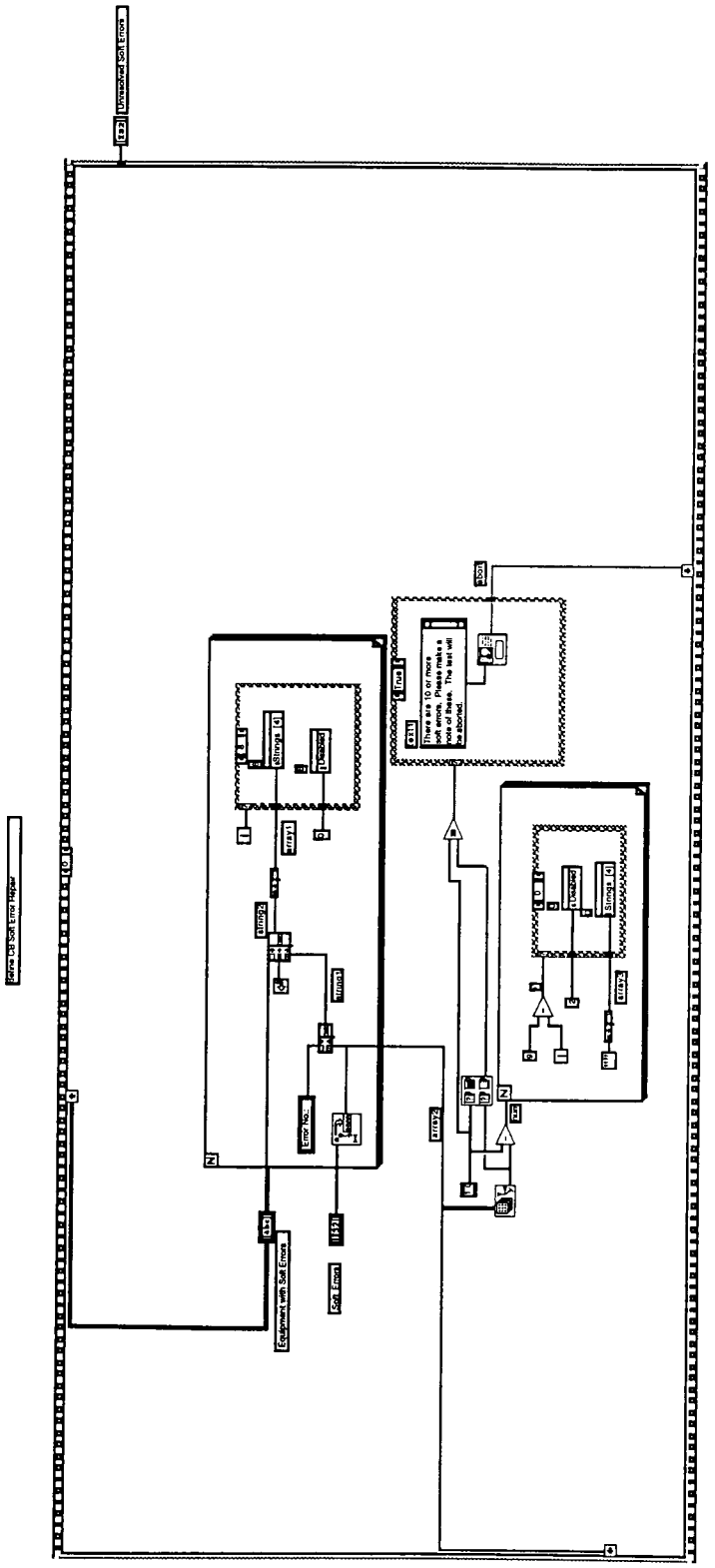
only, 2 = soft errors only, 3 = both hard and soft errors.

If only soft errors are detected, then user has the option of repairing these via a manual BIT operation. If he chooses to do so, then **repair_c.vi** is called (see Figure 4.8). This VI's purpose is to assist the user in correcting errors of the soft type. It allows a maximum of ten instruments to be "repaired". If there are more than ten, the test will be aborted. It outputs any unresolved errors so that the process may be iterated. It takes the *Equipment with Soft Errors* and the list of *Soft Errors* and produces the labels for the ten push buttons on the FP. Then, the user may press any one of these to repair the desired instrument. When he does, a dialog box is produced describing the error along with a corrective action to rectify it. Then, when he completes the action, he is to press "OK" and the instrument is checked again by a manually interfaced BIT. If it passes, then the user simply continues repairing the other instruments. If not, then it is added to the list of *Unresolved Soft Errors* and eventually this **repair_c.vi** will be called again. It should be noted that the manually-interfaced BIT software **man_bit.vi** is yet to be developed because the BIT is yet to be fully developed. Once all of the unresolved errors have been resolved, then the test may continue to the Reset and Initialization stage. These are discussed in the next section.



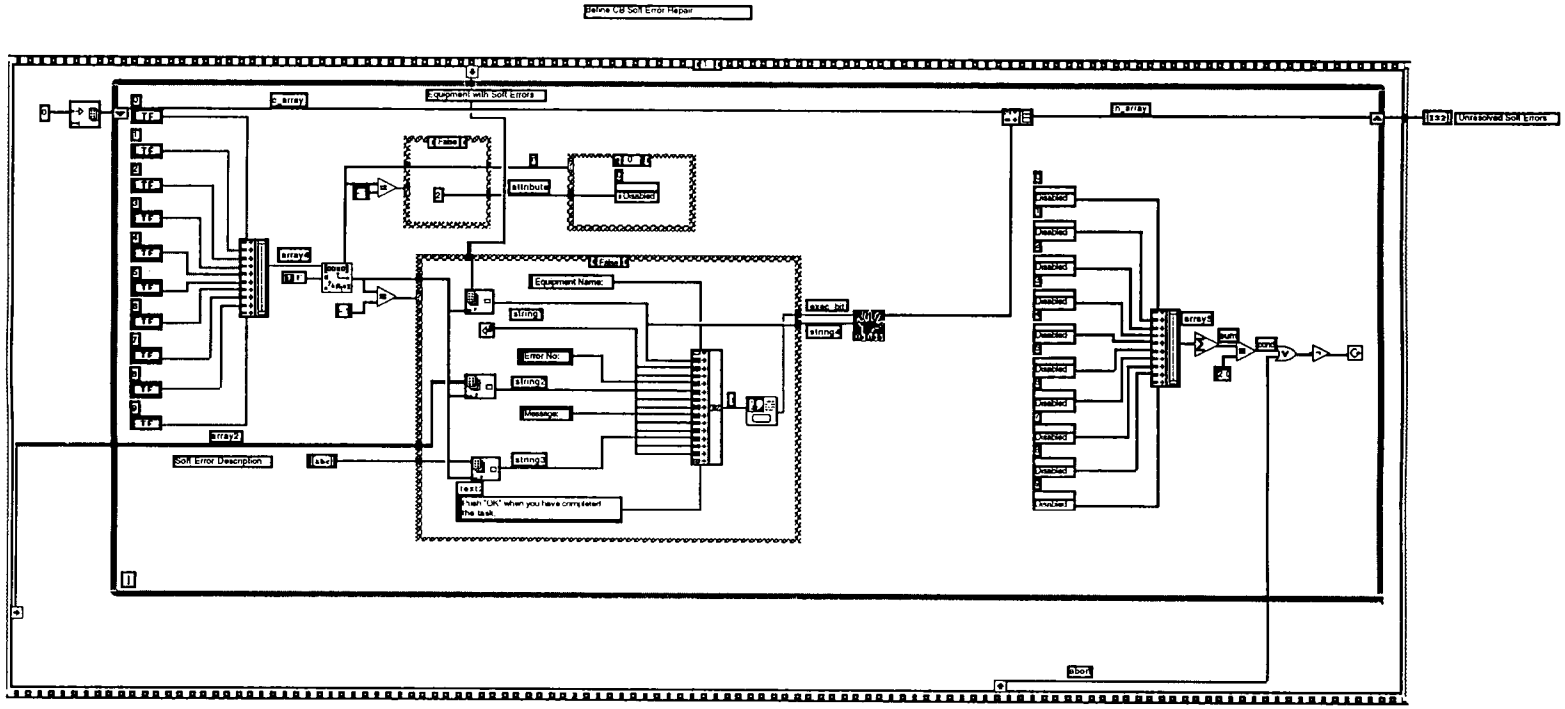
(a) Front Panel.

Fig. 4.8 Repair VI.



(b) Block diagram.

Fig. 4.8 (continued)



(b) (continued)

Fig. 4.8 (continued)

4.3 - Reset and Initialization

The purpose of this stage is to do just what its name suggests: Reset and initialize the hardware necessary for this test. Thus, the system is set to a known state. The code of interest is shown in sequence 1 of the BD. This step simply involved the initialization VI for the respective drivers. At the time of this development, only those for the TSI8150 and the SVS Router existed [17, 18]. Those for the PM5640, IR Transmitter, and the SMT02 were not yet developed.

The TSI8150 initialization routine provides full attenuation (127 dB) on all pads and it opened all RF switches. The SVS Router reset routine clears all connections in the router. Each of these returns an event code. For drivers not yet created, an event code of zero is assumed. Then, if the sum of these codes is zero, the test may proceed. Otherwise, it must be aborted because there was an error in the execution of one of those two initializations. Then, this boolean is passed through the right side of the diagram indicating whether or not the test should proceed or not. A false value indicates that the test should continue.

4.4 - Preset

The hardware is preset in this step. This is sequence number 2 in the block diagram for the RF Sensitivity Test. The purpose of this stage is to configure the hardware in the way prescribed for this test. At the time of this development, only the SVS Router driver was available. The other instruments had to be manually configured.

The signal connections for the beginning of this test are defined to be a Monoscope pattern on Channel 2 with the monaural 1000 Hz sinusoidal, 25 kHz deviation tone on Channels 2, 5, 12, 14, 45, and 69. Thus, these connections are defined in the *Router Preset* variable given in Figure 4.5. All of the other drivers are represented by zero event codes.

In addition, the *Channel* and *Pattern* controls have their text strings preset to the lists specified in *Set of Channels* and *Set of Patterns* given in Figure 4.5. The *Current TV* indicator and *Local Header* global variable (see Figure 4.9) are initialized from the *TV Header*. The *Local Header* will be discussed further in the following section.

Once again, the event codes from the driver calls are checked and if there are no errors, then the user is prompted to tune all DUTs to Channel 2 and adjust them to their Factory Preset conditions. Then, the procedure will progress to the testing stage. Otherwise, if

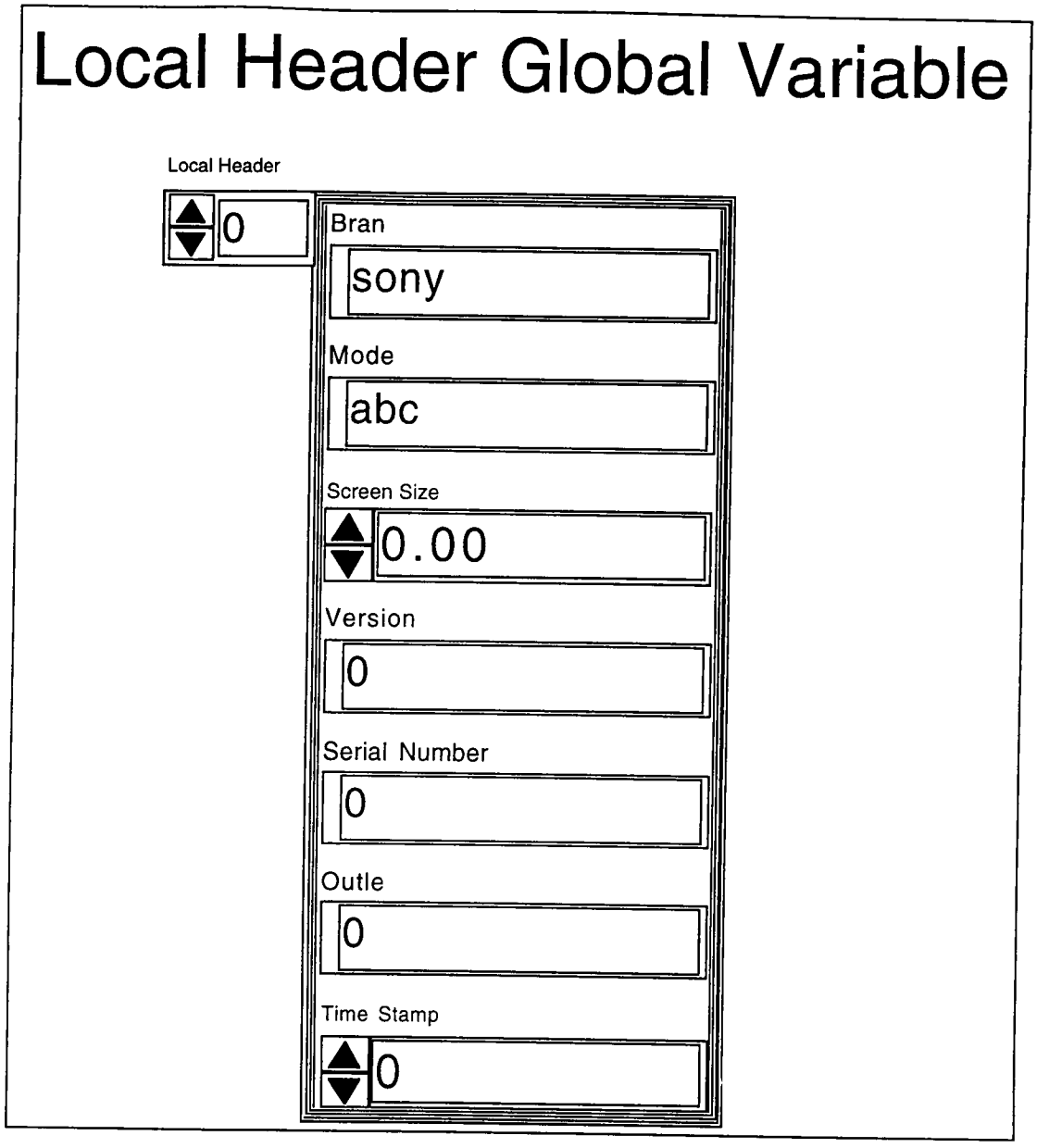


Fig. 4.9 Local Header VI.

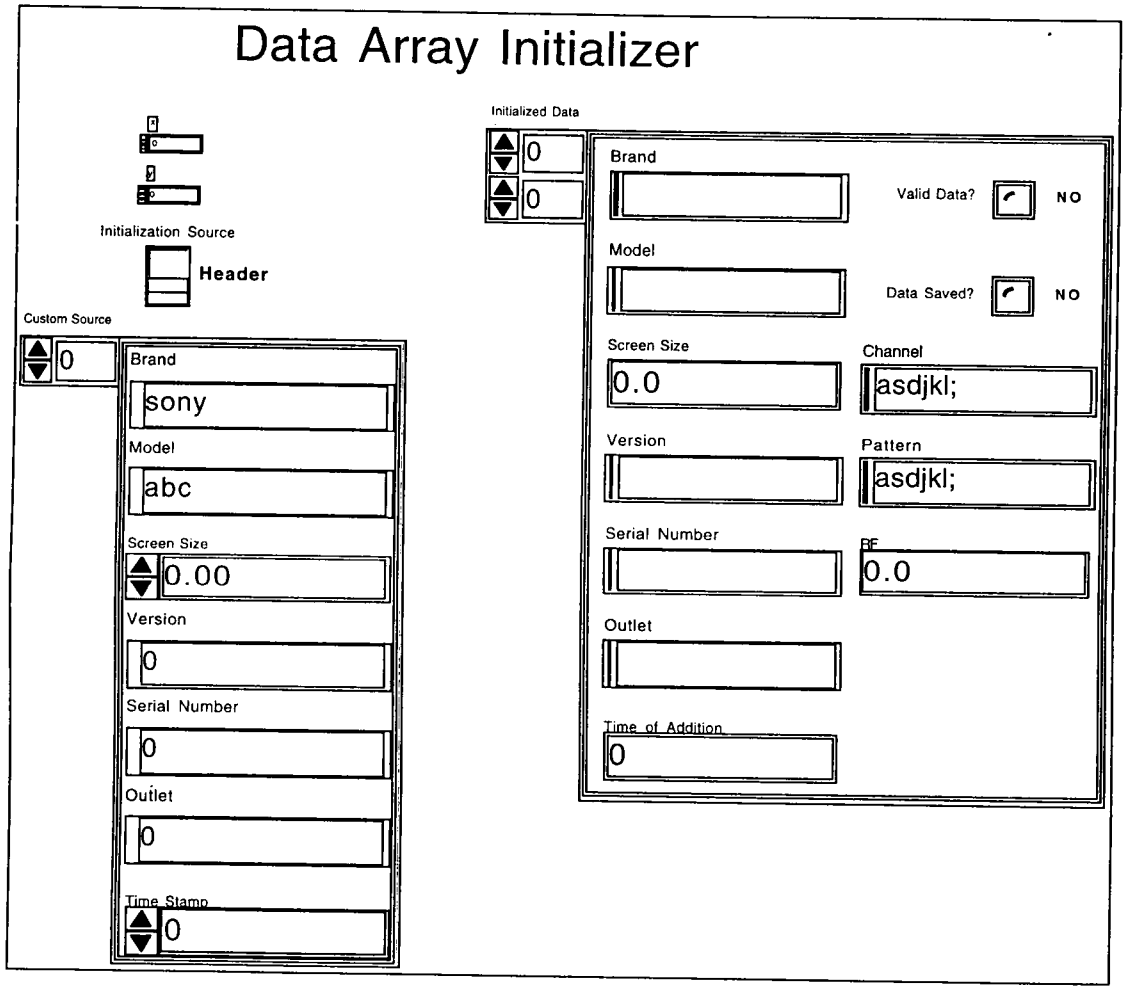
there is an error, the test will be aborted. This decision is carried by the boolean value being passed out of the right side of the BD wherein a false value indicates that the test should continue.

4.5 - Testing

This section discusses the software written to perform the RF Sensitivity Test. This is shown in sequence 3 of the BD. The first step is to enable all of the controls and indicators that were grayed-out and disabled in sequence 0. That is the reason for all of the *Disabled* attribute nodes in the lower left. Here they are assigned a value of zero which enables all of these controls and indicators.

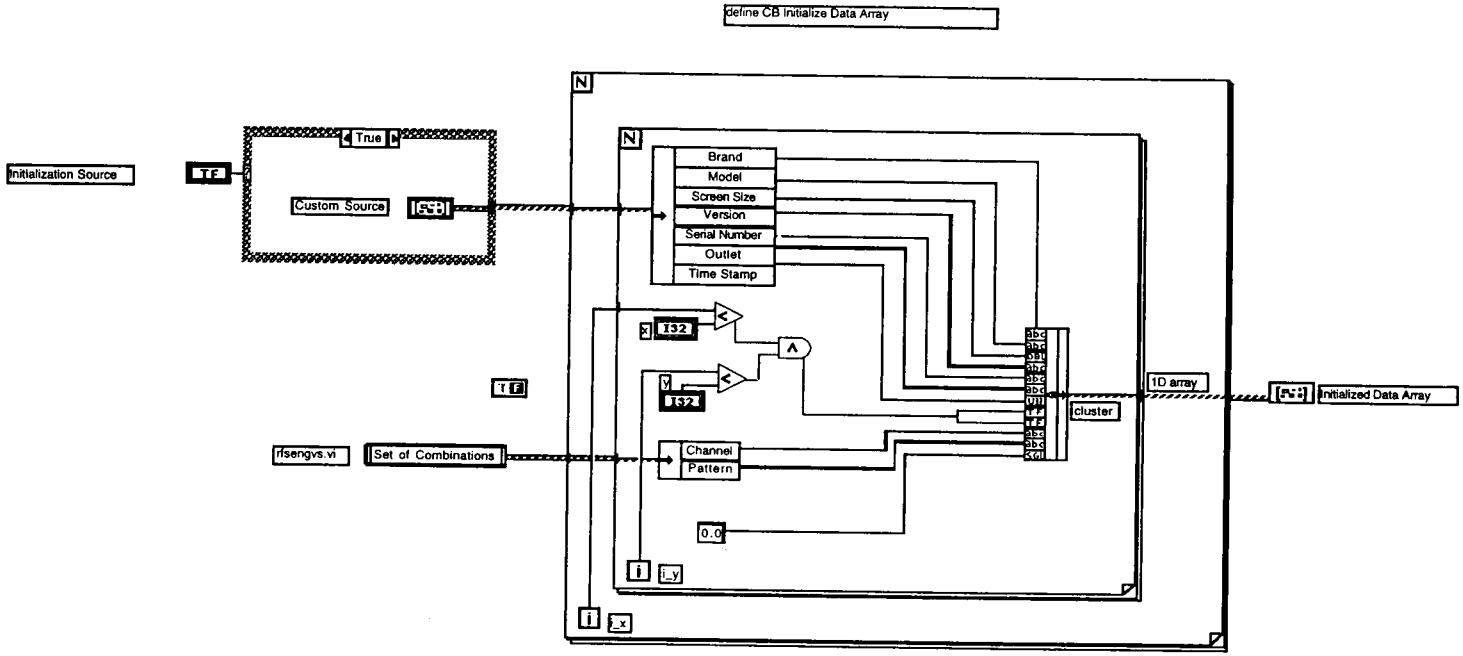
Continuing from left to right, the shift registers used during the testing must be initialized. The first three are used by the **drvctl_c.vi**, the driver controller VI. This will be discussed later. They maintain the current Channel, Pattern, and RF Level information, respectively. The lower four are used to maintain the *x-index*, the *Automatic Index*, the *Automatic Channel*, and the *Automatic Pattern.* These are used during recording operations and the automatic sequencing which are discussed later.

The shift register in the middle maintains all of the information for all of the DUTs. It is initialized by the **inidat_c.vi** (see Figure 4.10). This VI's purpose is to initialize the two-



(a) Front Panel.

Fig. 4.10 Data Array Initialization VI.

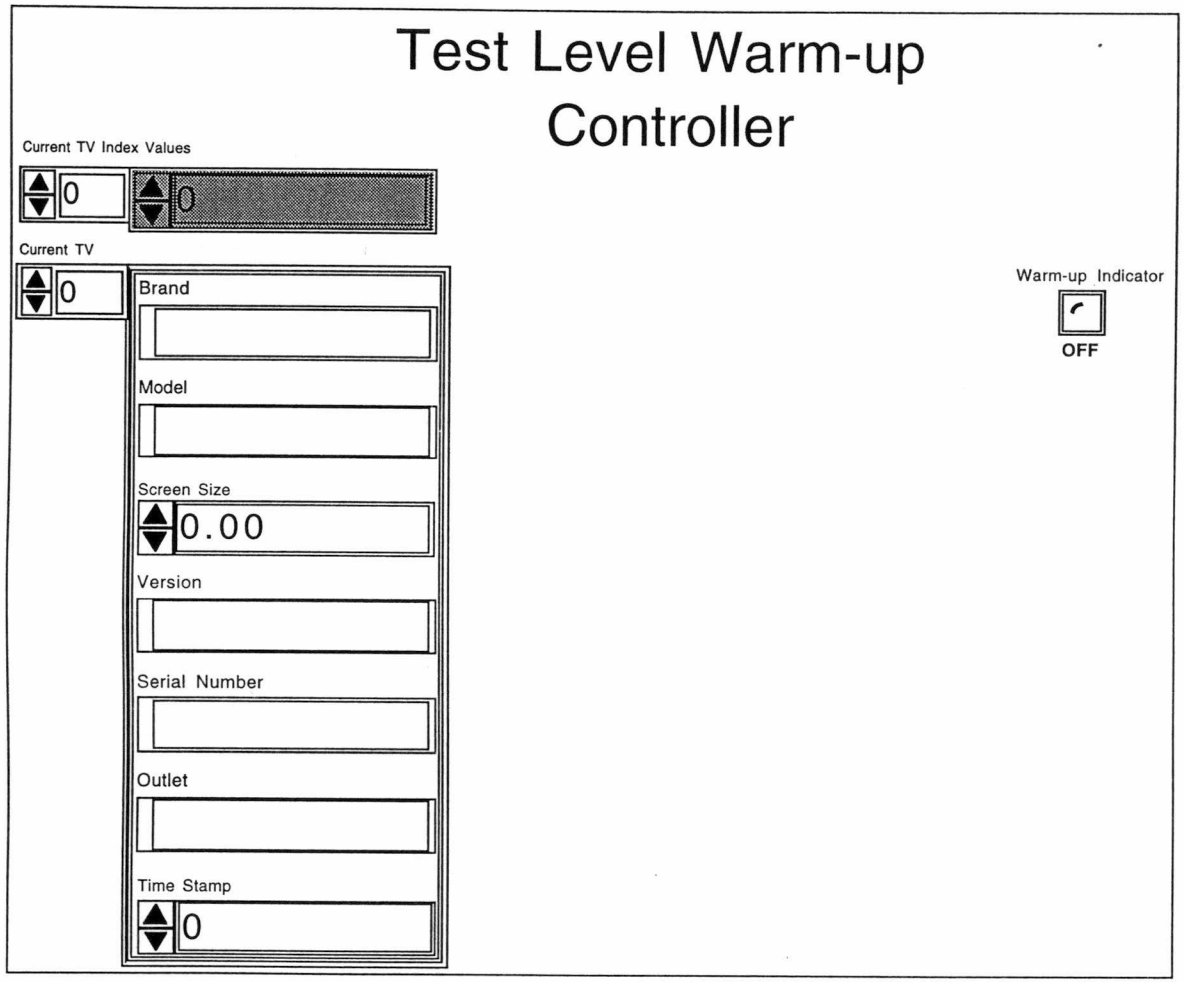


(b) Block diagram.

Fig. 4.10 (continued)

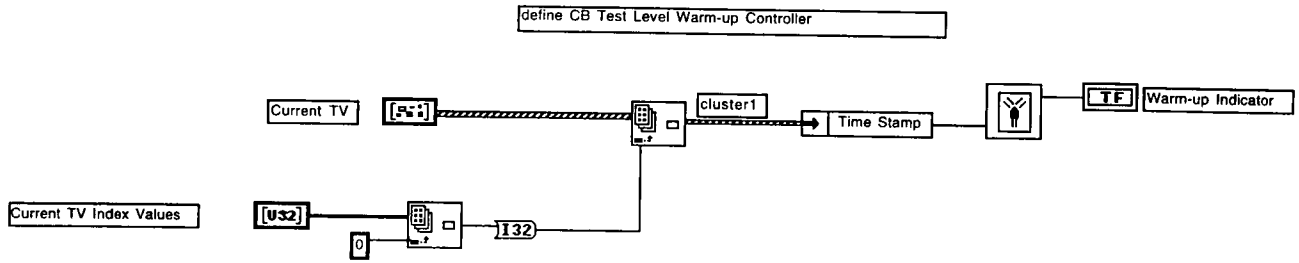
dimensional data array used to keep measurement records and DUT information. It builds this 2D array with the information from the *Local Header.* In addition, it initializes the boolean indicators *Valid Data?* and *Data Saved?* to false. It also puts in all of the possible channels and patterns specified by the *Set of Combinations* global variable as well as zero for the RF Level. Thus, the output is the *Initialized Data Array* which has DUTs as its elements in the x-dimension and measurements in the y-dimension. For example, element (0,0) would be DUT #1 and the measurement would be on Channel 2, Monoscope (video sensitivity) and element (3,4) would be DUT #4 and the measurement would be on Channel 5, Audio (audio sensitivity).

The **tstwrm_c.vi** (see Figure 4.11) is the next topic of discussion. This VI was written to control the *Warm-up Indicator* on the FP. It checks each TV's *Time Stamp* and determines whether or not that TV is "warm". If it is warm, then the indicator will be off. It relies on the **warm_c.vi** to perform this check. The **warm_c.vi** has as a default value 15 minutes [19]. So, if the *Time Stamp* of a given DUT indicates that is has been in the *Local Header* for at least 15 minutes then the indicator will be off. This means that the TV should be "warm" and so any data taken on it should be valid. A caution is necessary though. This simply indicates that 15 minutes



(a) Front Panel.

Fig. 4.11 Warm-up Check VI.



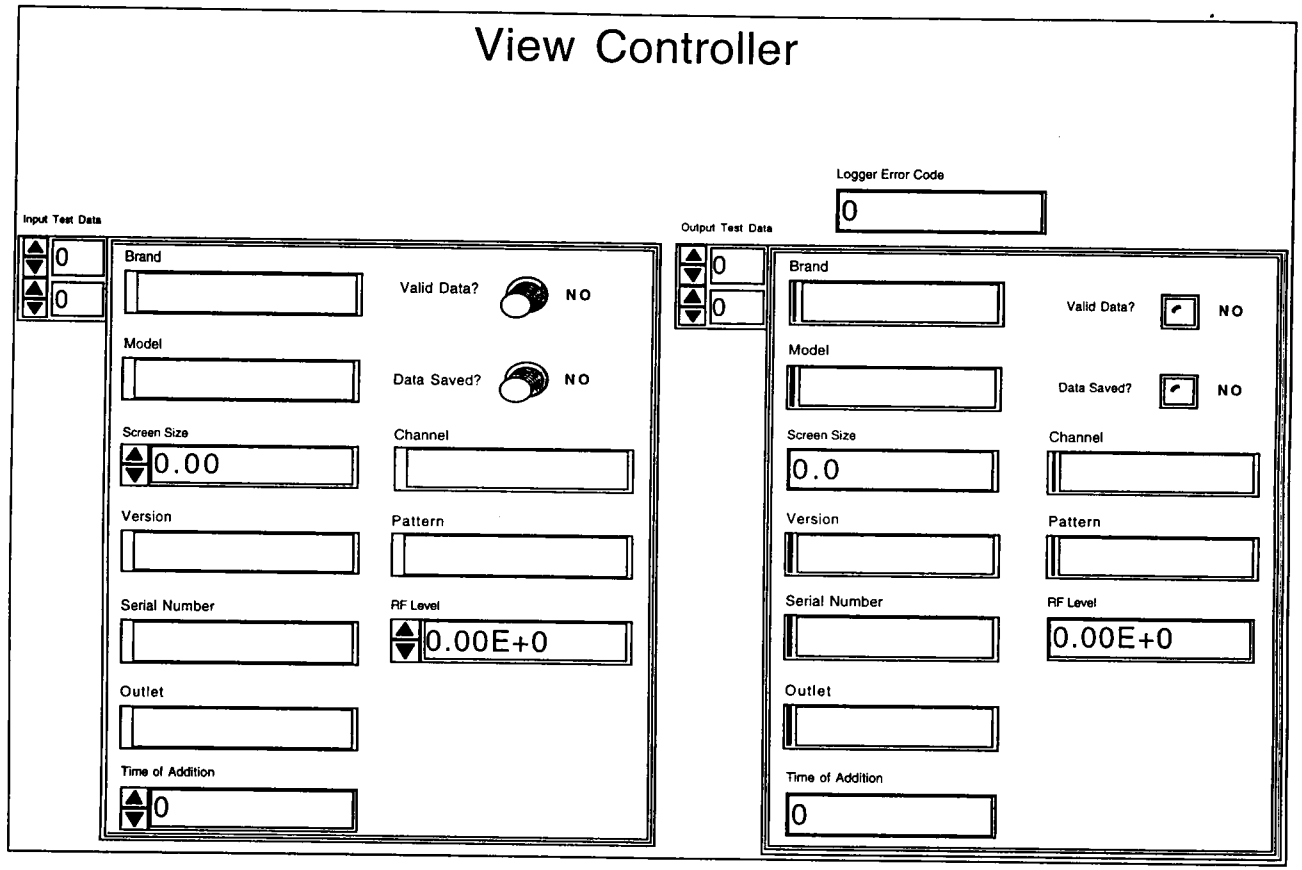
(b) Block diagram.

Fig. 4.11 (continued)

has passed since the TV was entered into the Header. It does NOT guarantee that the TV is on and has been warming up for 15 minutes. This is the job of the tester.

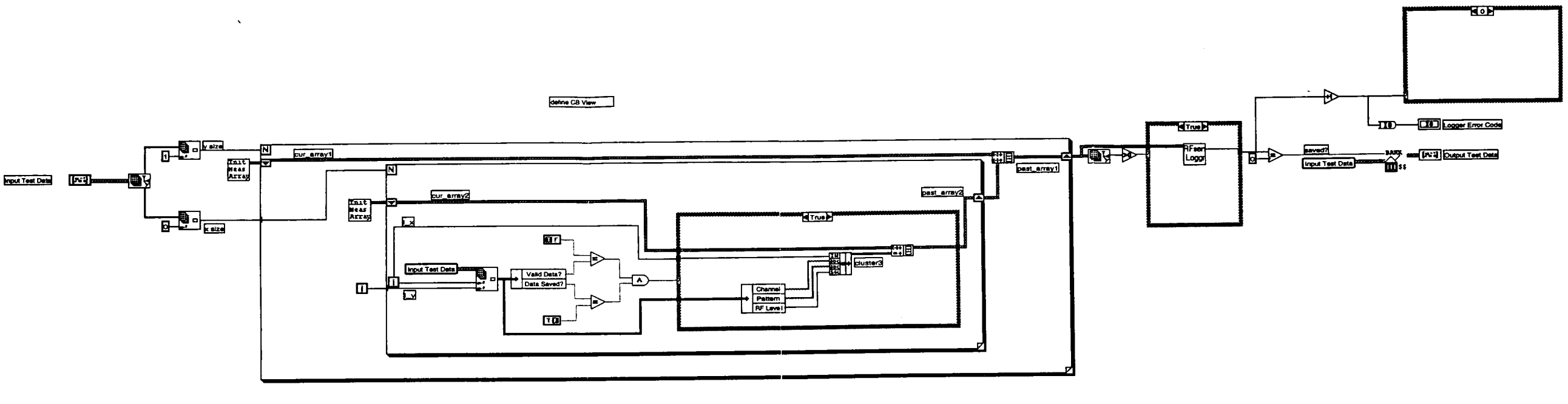
. The discussion will now focus on the case structure on the left. The case is determined by which of five buttons is pressed on the FP: *View*, *ATF Header*, *RECORD*, *Quit*, or *Logout*. If none of these is pressed, then case 0 is executed. This is the null case wherein the shift register values are simply passed through and the Test Complete? boolean value is set to false.

Case 1 is executed if the *View* button is pressed. It contains the **view_c.vi** (see Figure 4.12). This VI's purpose is to control the displaying of the data taken as well as logging it and toggling the *Data Saved?* boolean when the data is logged to disk. This VI searches through the entire 2D array for elements where *Valid Data?* is true and *Data Saved?* is false. This means that a measurement has been taken and has not yet been written to disk. If these two occur, then it appends certain information to an array of clusters which will then be passed to the Logger VI [20]. The information it appends is the Header index number for the DUT, the Channel, the Pattern, and the RF Level. Otherwise, it passes the array around in a shift register. The Logger VI then uses this information to fill out a table which is displayed. That table contains this information



(a) Front Panel.

Fig. 4.12 View data VI.



(b) Block diagram.

Fig. 4.12 (continued)

along with information it derived from the Header using the Header index number. A copy of this display is shown in Figure 4.13. The Logger VI outputs the *Logger Error Code*. These are summarized in Table 4.1. If all *Data Valid?* booleans are false, then the Logger would not be called and an error code of -1 is output. This error code is then used to control the case structure on the far right of the BD. This case structure produces dialog boxes describing the codes summarized in the table below. Finally, if the user selects "Save" on the Logger FP, then the Save VI is called. It toggles the *Data Saved?* boolean for each saved measurement to true.

Table 4.1 Logger Error Codes.

Code	Description
0	<continue--do nothing>
1	data stored properly; reinitialize TV measurement
2	Directory or path creation problem. No data was stored. Please contact the System Administrator for assistance. You may wish to PRINT the data.
3	There was a problem writing to or creating the file. No data was stored. Please contact the System Administrator for assistance. You may wish to PRINT the data before Quitting or Logging Out.
4	There was a problem writing to or creating the file. Some data was stored. Please contact the System Administrator for assistance. You may wish to PRINT the data before Quitting or Logging Out or else the remaining data will be lost.

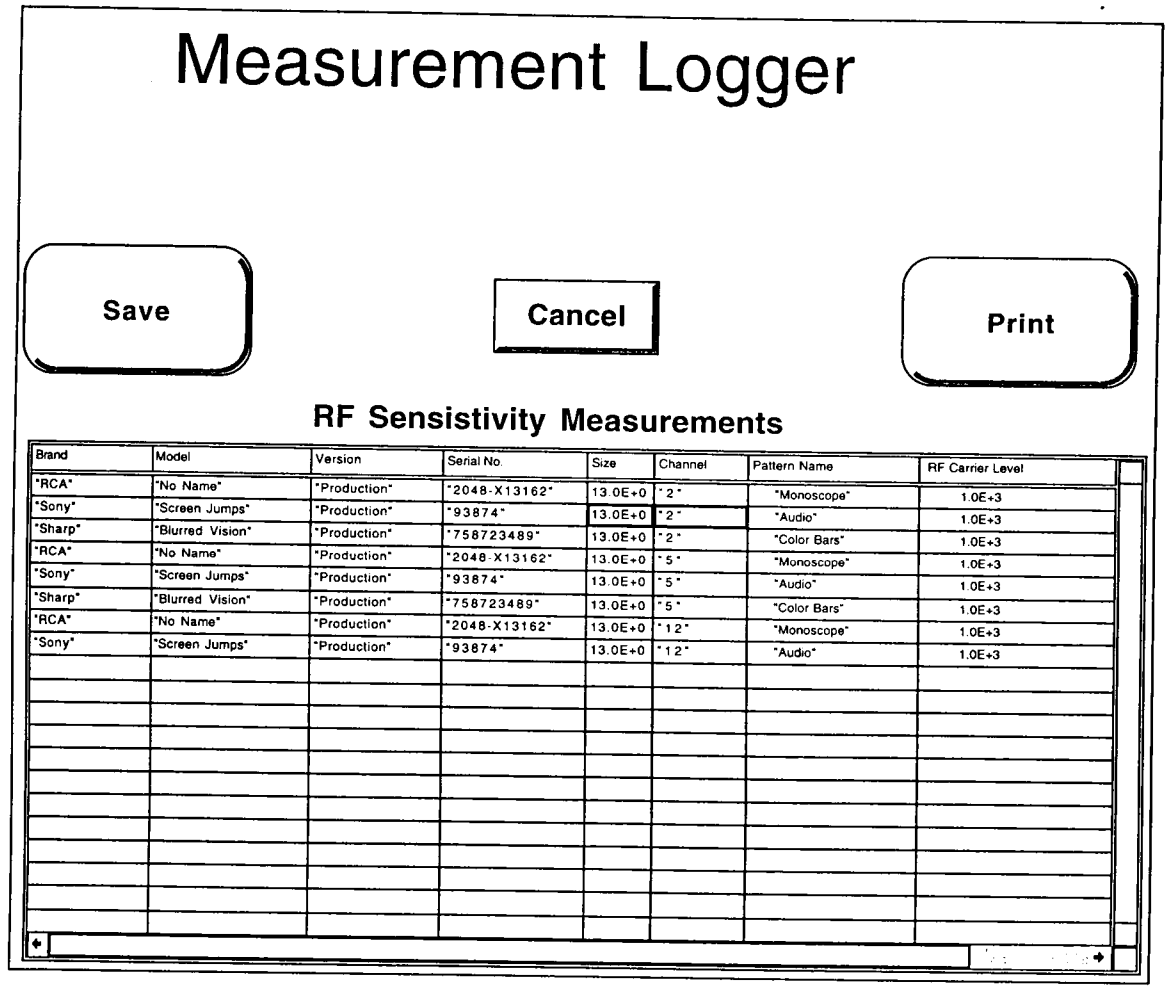
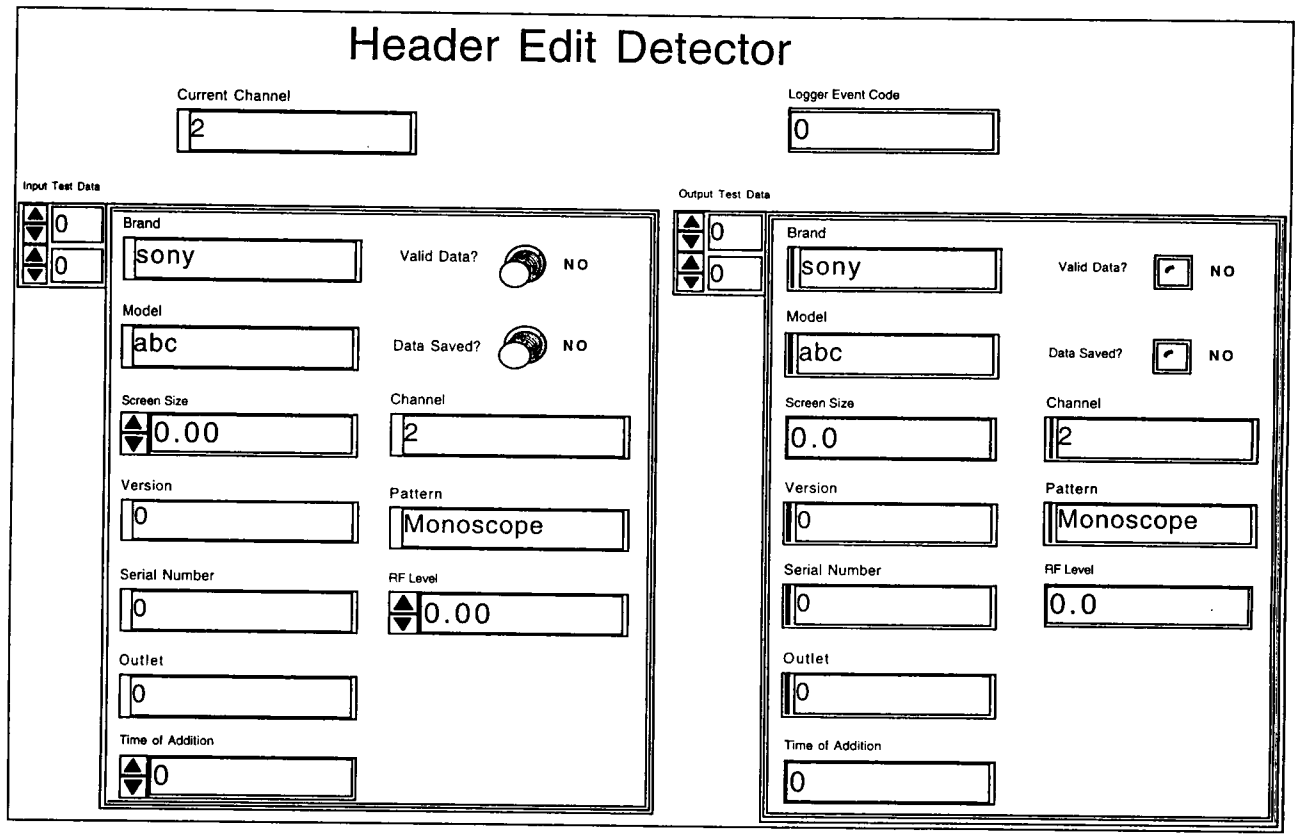


Fig. 4.13 Measurements display panel.

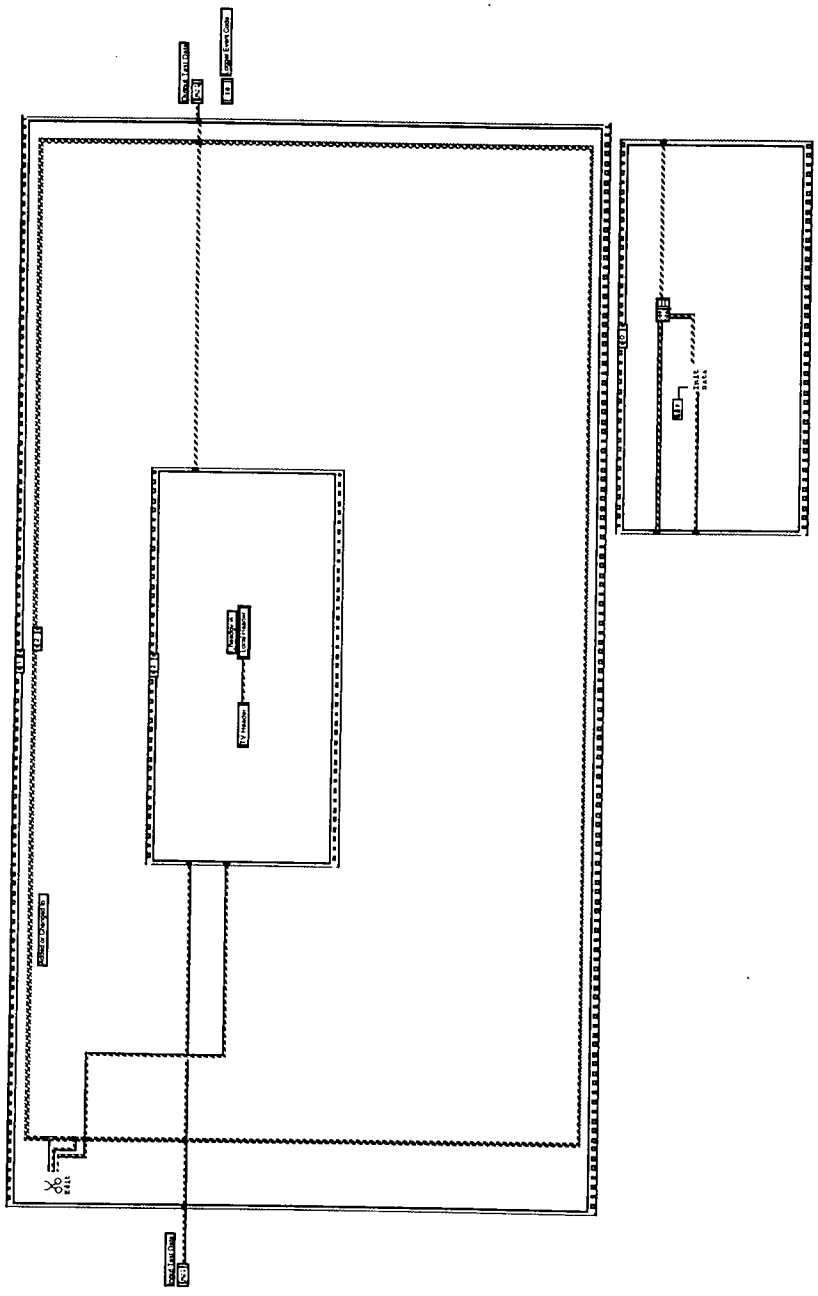
When the *ATF Header* button is pressed the **head4_c.vi** is called (see Figure 4.14). This VI's purpose is to detect changes made to the Header. It is designed to determine when additions, deletions, or changes to current entries were made to the Header. The first thing that occurs is that the Header is called. Then, once the user has made the necessary additions, deletions, and/or changes to it, the **head4_c.vi** determines what they were by comparing the *Local Header* to the *TV Header*. This can be done because the *Local Header* contains the information that was valid prior to the user's interaction with the Header. Thus, the two can be compared and the differences detected. This detection is done in the Edit VI in sequence 1 of the outer sequence structure. Once the detection is complete then the appropriate operations must be performed on the 2D data array. It must be updated to include additions, to not include deletions, and to show changes. In the event that there is a deletion or change to the Header and the item(s) that is(are) deleted or changed has measurements associated with them, the Logger is called. Then, the user may wish to save these or print them out. Overall, this was a very difficult VI to design. So, the subVIs employed to perform these operations are not presented here for considerations of brevity.

Pressing the *RECORD* button calls the **record_c2.vi** VI (see



(a) Front Panel.

Fig. 4.14 Header edit detector VI.



(b) Block diagram.

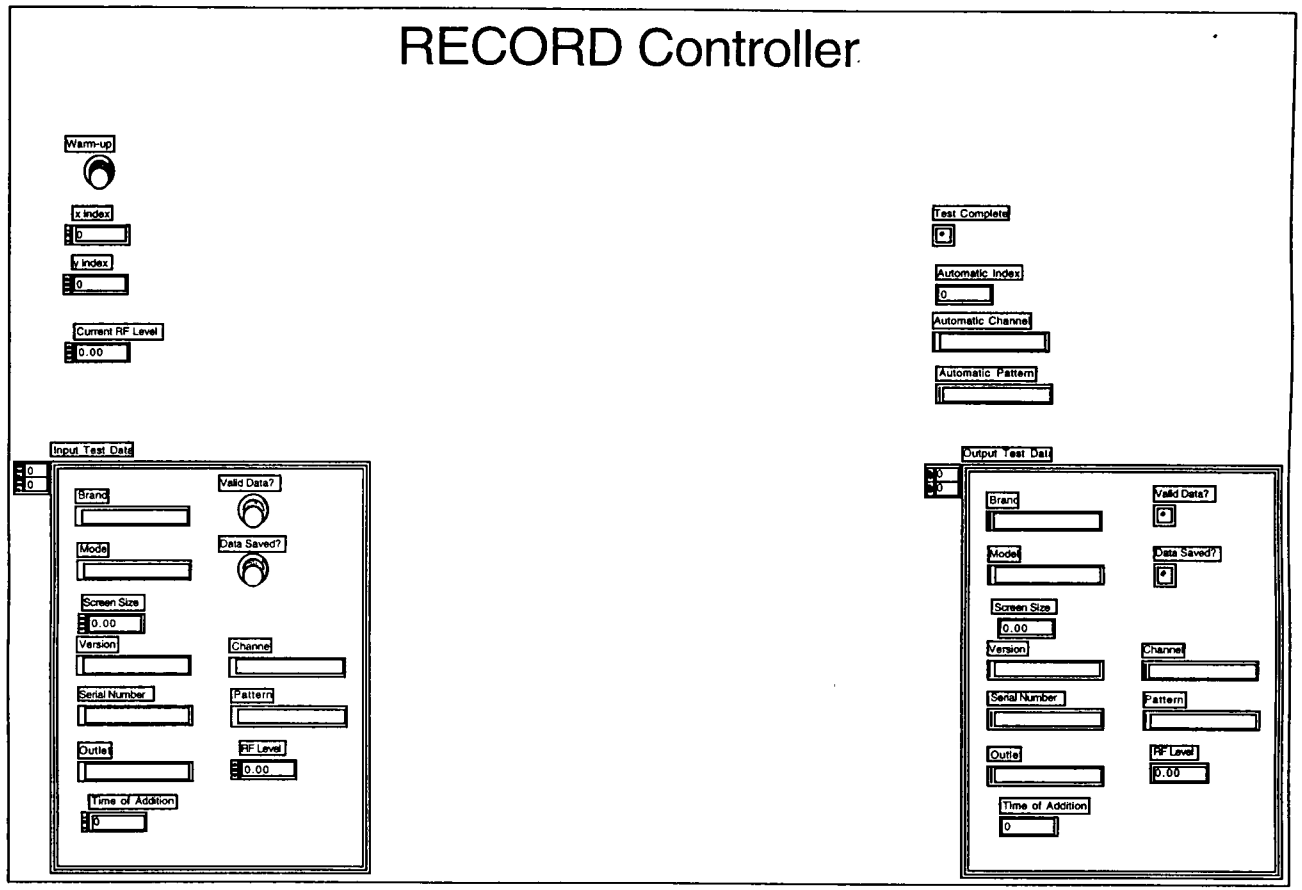
Fig. 4.14 (continued)

Figure 4.15). This VI's purpose is to actually record the measurement in the 2D data array, toggle the *Valid Data?* boolean, and determine the next step in the auto sequencing. The BD consists of .two subVIs: **recops_c.vi** and **autoseq_c.vi**.

The **recops_c.vi** (Figure 4.16) controls the operations associated with the pressing of the *RECORD* button. It determines whether or not a measurement has been made before for the given x-y combination of index numbers. If a measurement has been made, then it checks to see if that measurement has already been written to disk. Consequently, after performing these checks it prompts the user of the condition and then notifies him of his options.

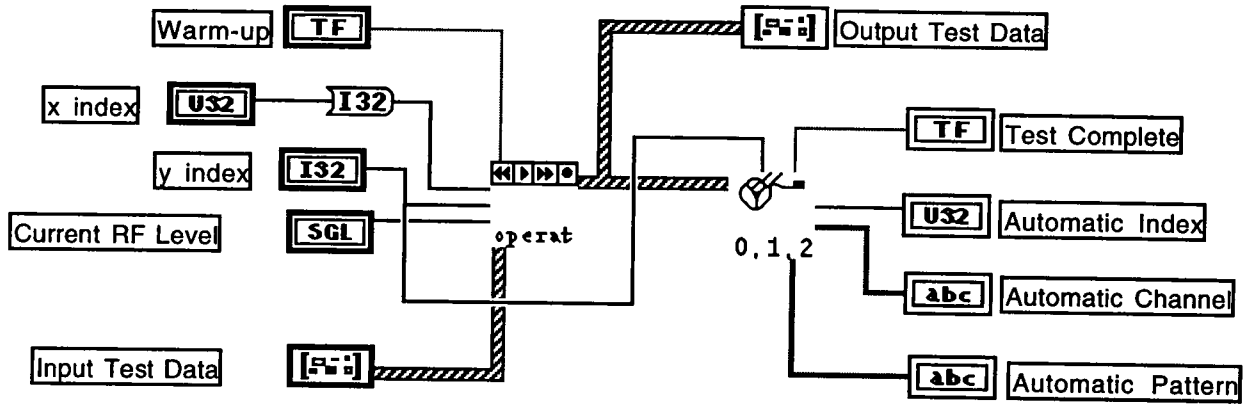
The **autoseq_c.vi** (Figure 4.17) determines the next step in the auto sequencing. If the test is running in Auto Mode, then it will sequence through the channels and patterns automatically as controlled by the *RECORD* button. So, this VI controls how that sequencing occurs. It considers warm DUTs before cold ones and it progresses in the 2D array from left to right and top to bottom. The test mode is discussed further below.

The **tstcom_c.vi** (Figure 4.18) is also in this case along with the **record_c2.vi**. It was written to determine if the test was complete. It receives the *Test Complete* boolean from the **record_c2.vi**. The value of this boolean is determined by the



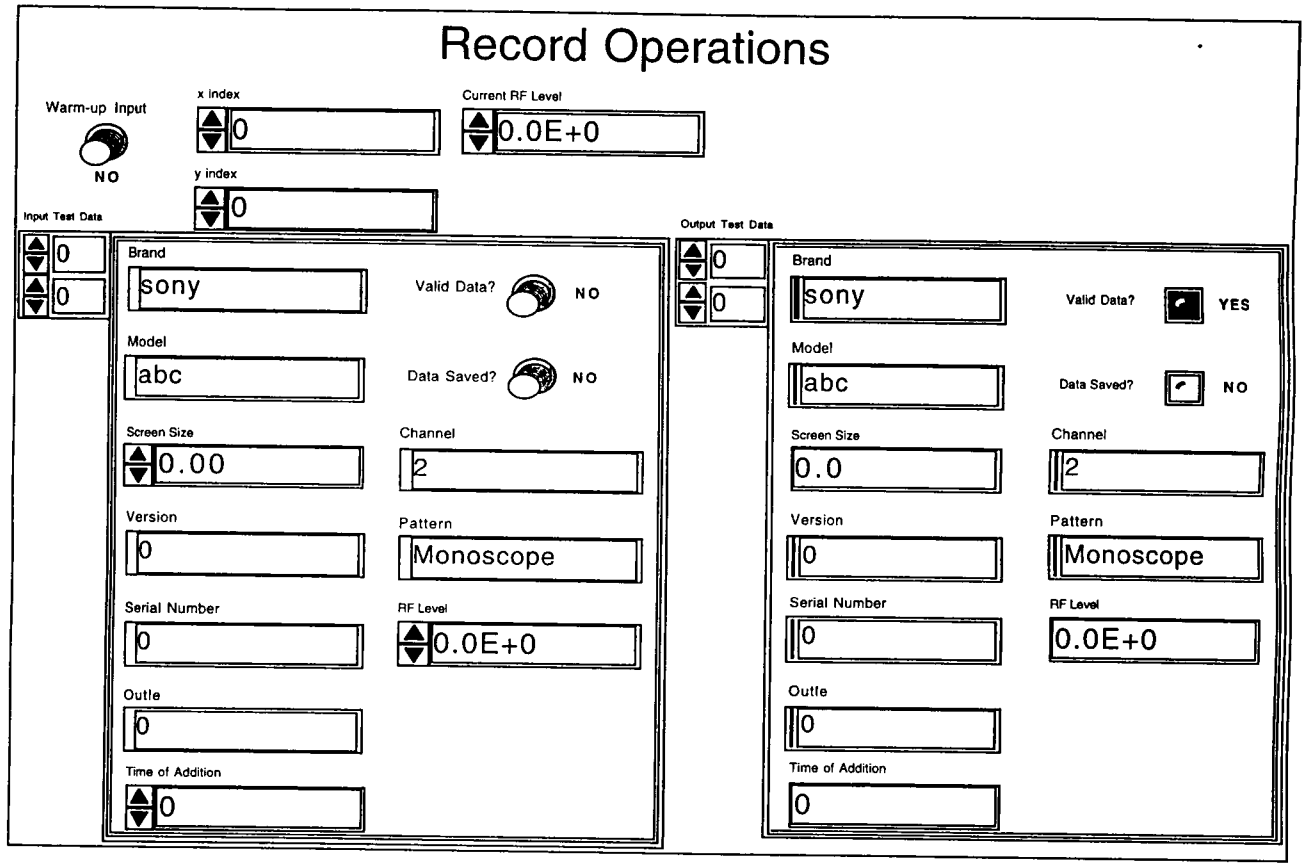
(a) Front Panel.

Fig. 4.15 Record control VI.



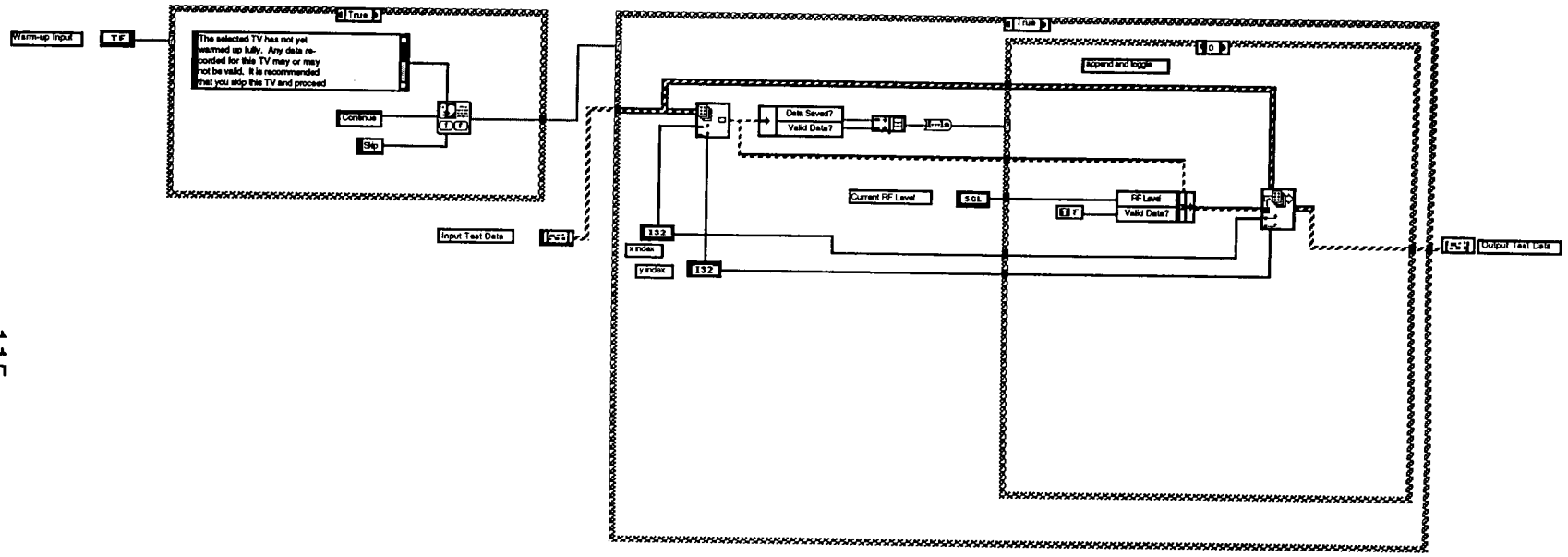
(b) Block diagram.

Fig. 4.15 (continued)



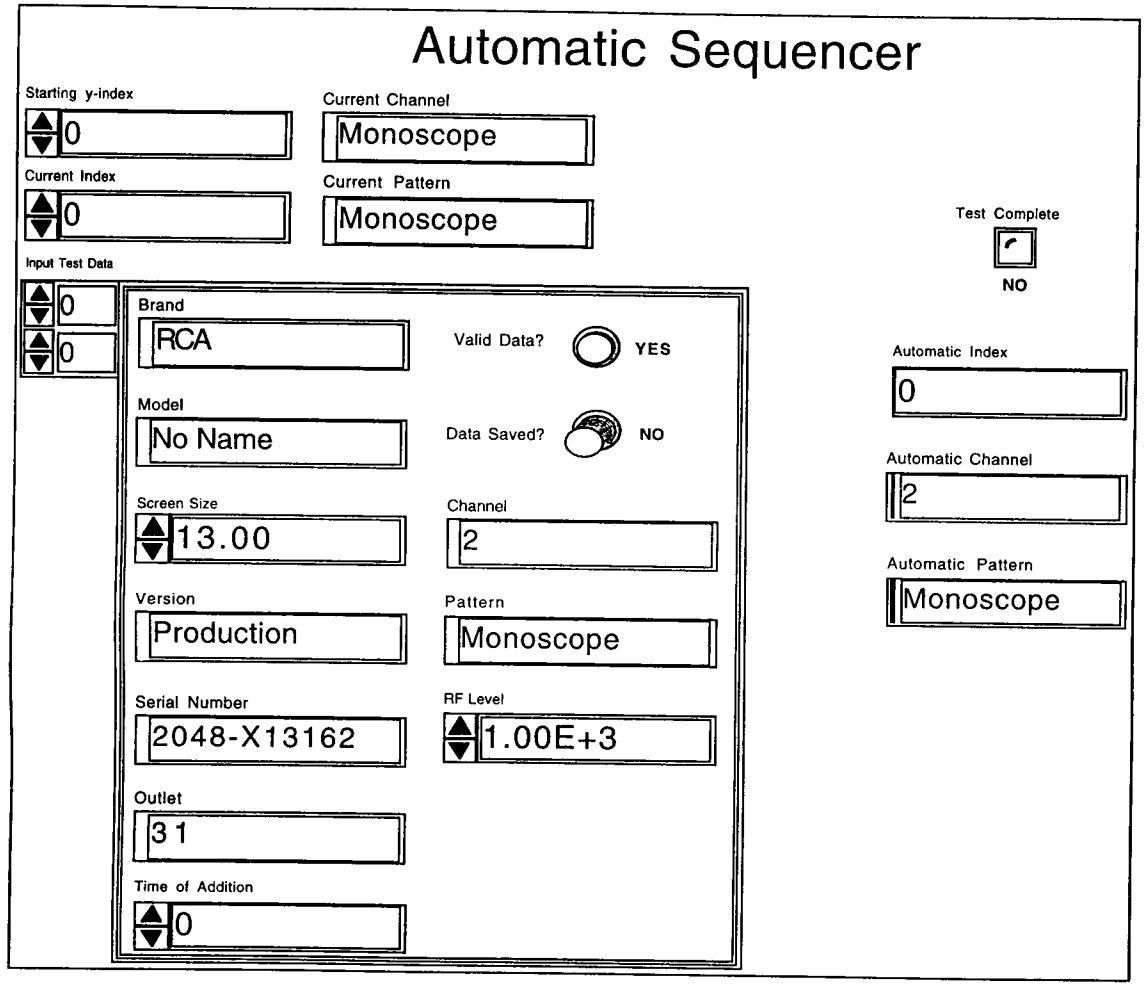
(a) Front Panel.

Fig. 4.16 Record operations VI.



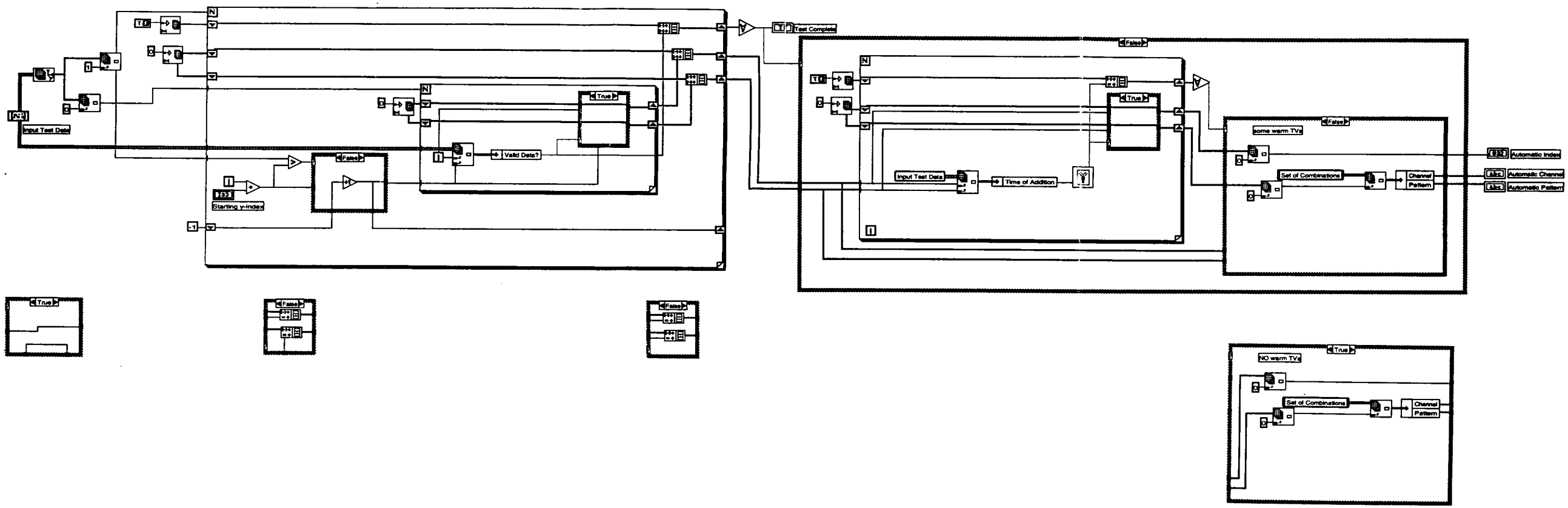
(b) Block diagram.

Fig. 4.16 (continued)



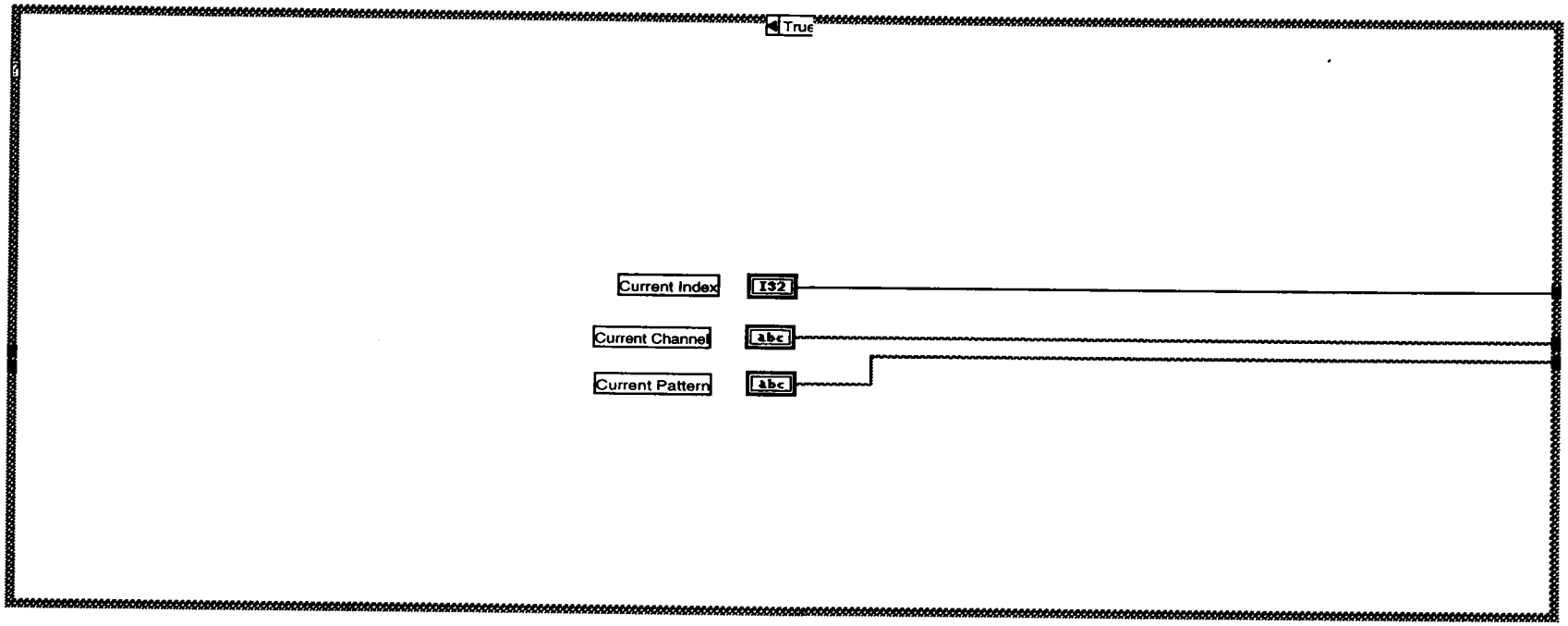
(a) Front Panel.

Fig. 4.17 Auto sequencing VI.



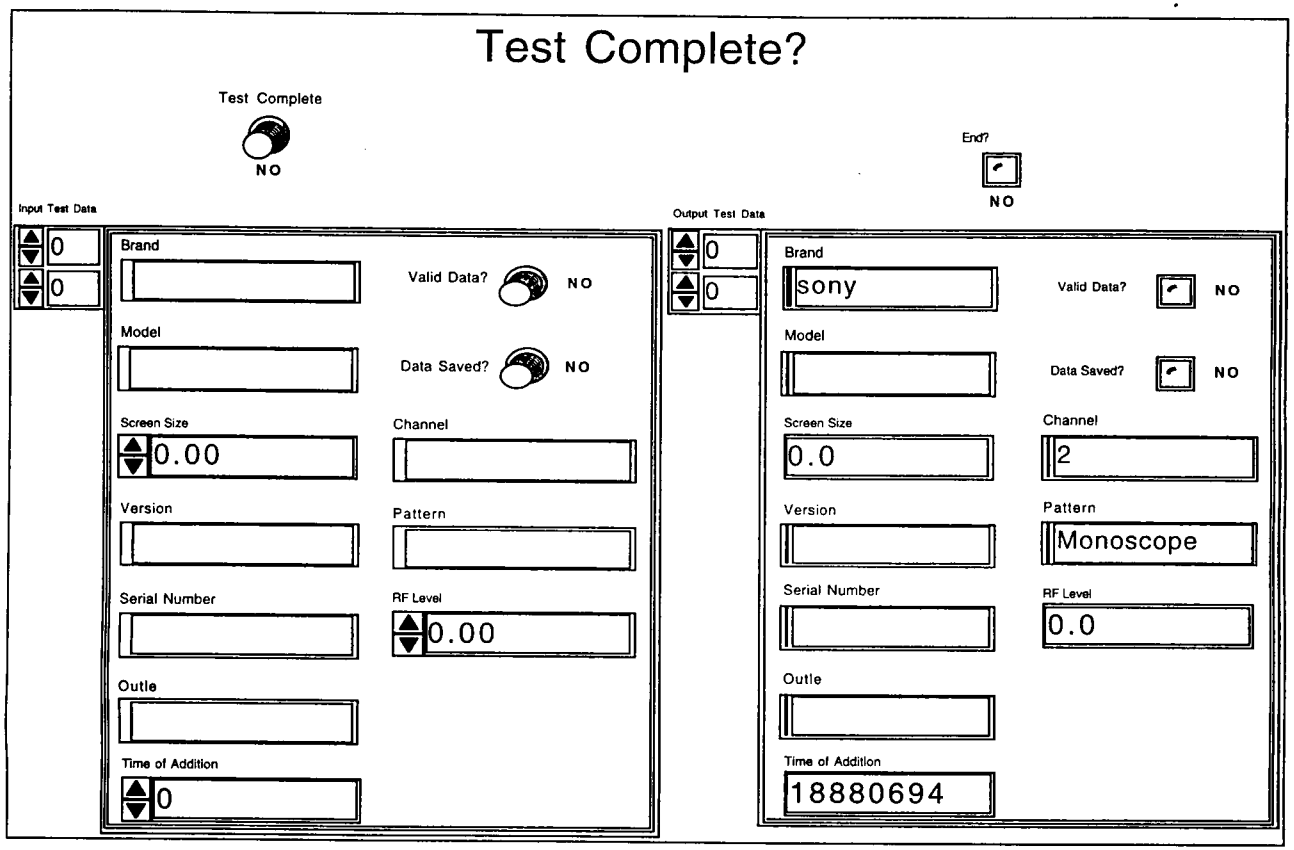
(b) Block diagram.

Fig. 4.17 (continued)



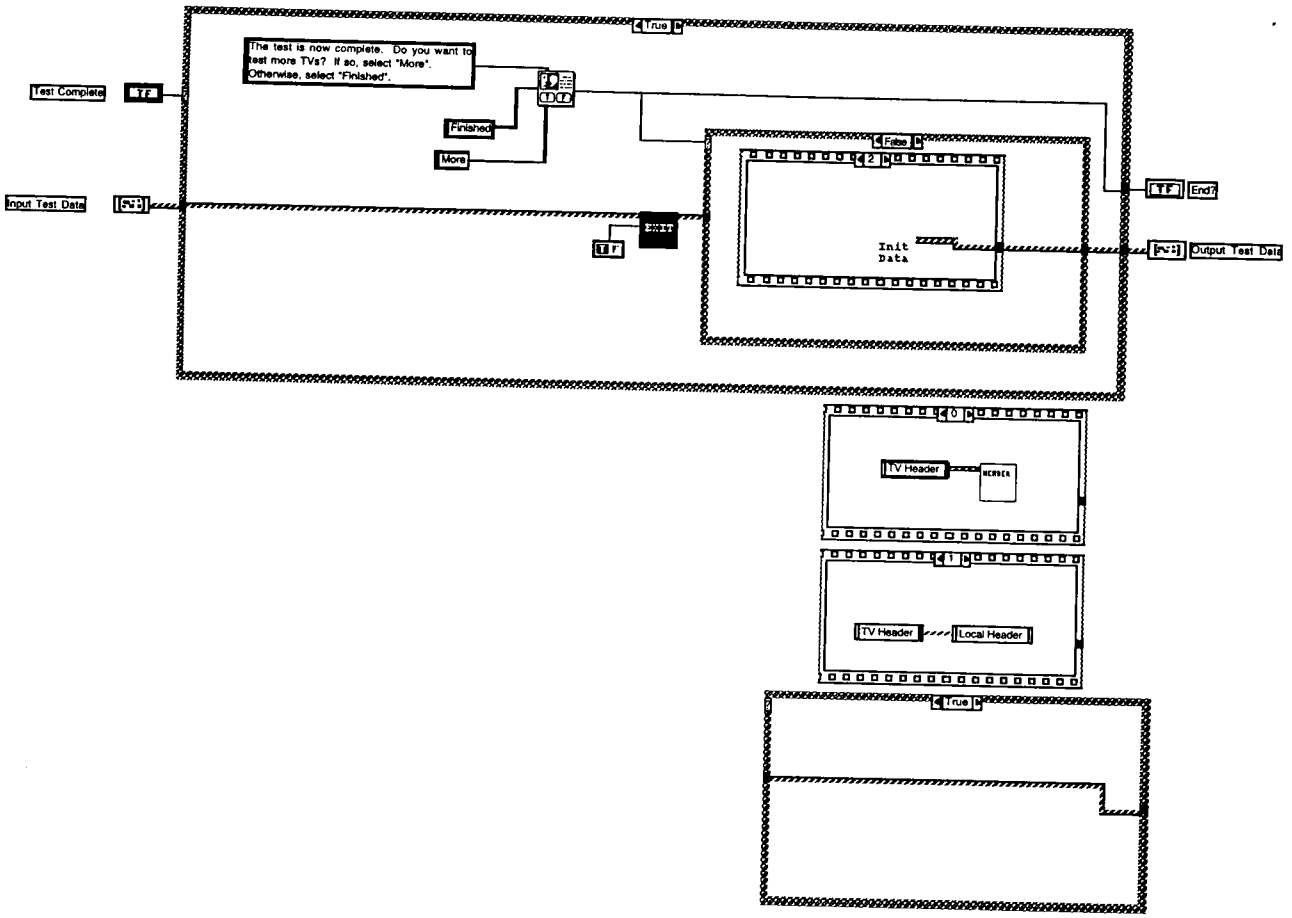
(b) (continued)

Fig. 4.17 (continued)



(a) Front Panel.

Fig. 4.18 Test complete check VI.



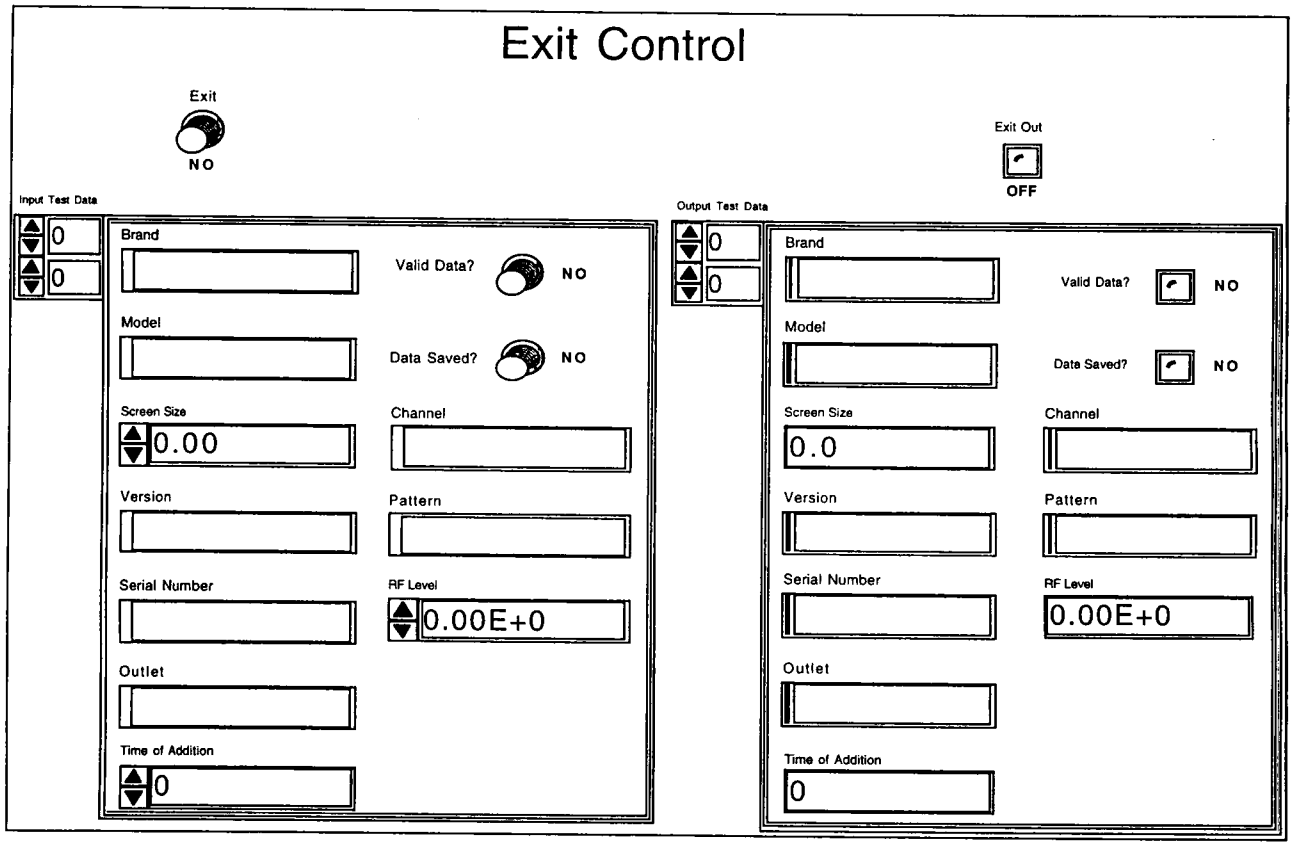
(b) Block diagram.

Fig. 4.18 (continued)

autoseq_c.vi. If the auto sequencing finds that all measurements for all DUTs have been taken then the value is true; otherwise, it is false. If it is true, then this test complete VI prompts the user that that is the case and asks him if he is finished with this test or else would like to test more TVs. If he is finished, then *End?* is true and nothing is done to the 2D data array. If he wants to test more televisions, then the Header is called, the *Local Header* is updated and the 2D data array is reinitialized with the new header information and the test starts over with Channel 2, Monoscope. In the latter case, *End?* is false.

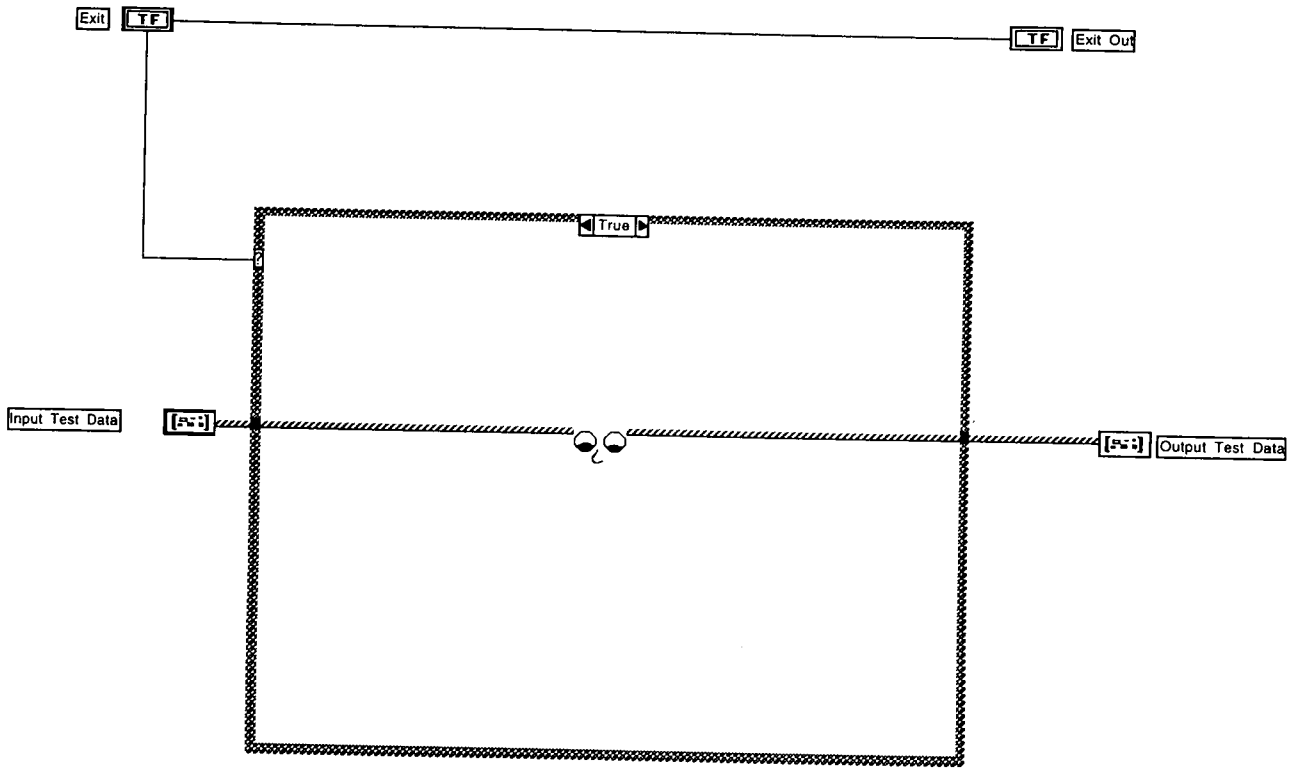
Case 4 is executed if the *Quit* button is pressed. In this case, the **exit_c.vi** VI is called. This VI is shown in Figure 4.19. It is very simple in design. If *Exit* is true, then **view_c.vi** is called. Its operations were discussed above. Otherwise, the 2D data array is simply passed out with no changes made to it. This VI is used to process any data that has not yet been saved to disk or printed. Therefore, **view_c.vi** is called.

Case 5 is very similar to case 4 except that along with the **exit_c.vi** it also contains the Resolve Logout VI [21]. The latter VI simply puts a true value in the Resolve Logout boolean global variable which is read by all panels below the Top Menu. It signals all of them that the user pressed *Logout* and so they must stop and close.



(a) Front Panel.

Fig. 4.19 Exit control VI.



(b) Block diagram.

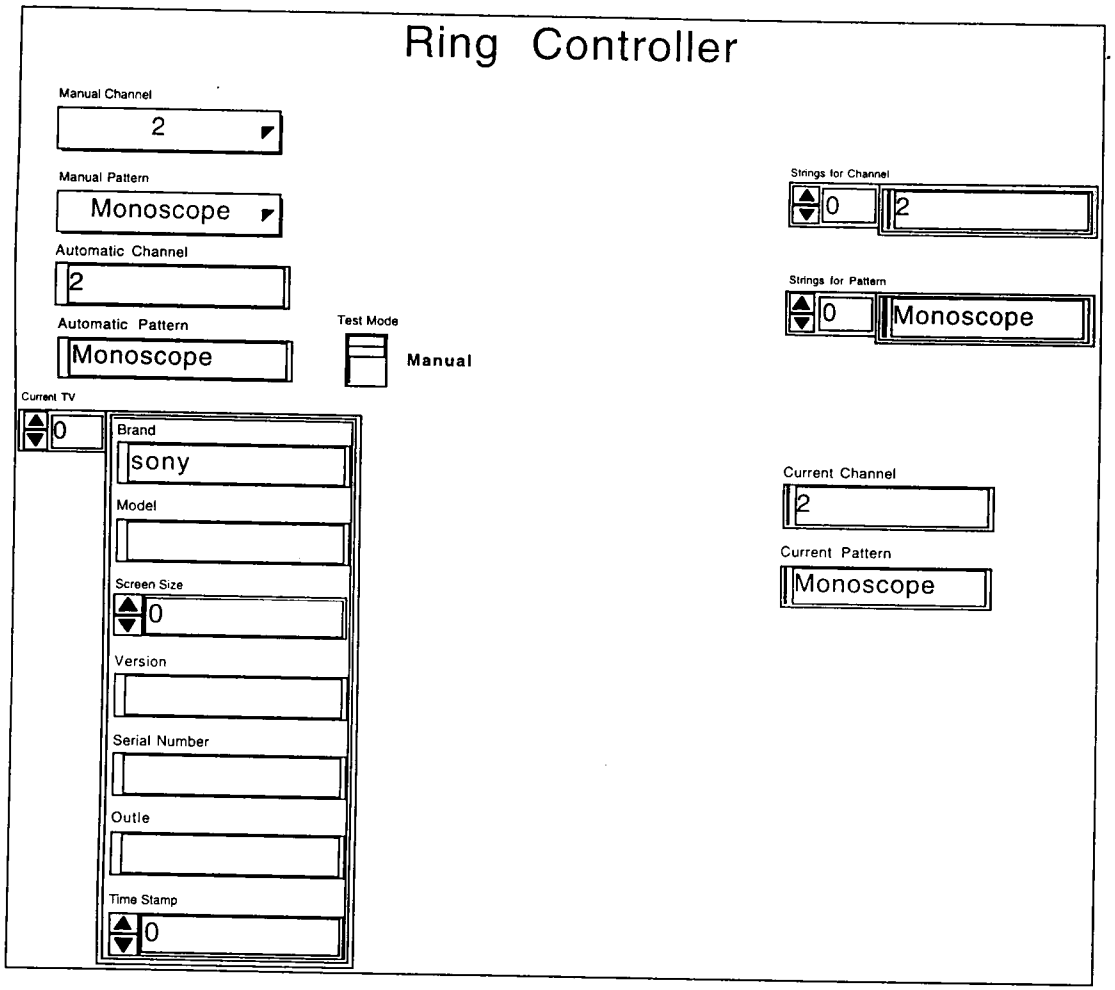
Fig. 4.19 (continued)

Thus, the user is returned to the Top Menu.

The next subVI to be discussed is the ring control VI, **rngctl_c.vi** (Figure 4.20). This VI's purpose is to control the menu rings (*Channel* and *Pattern*) and the *Current TV* array index. It does this based on the mode in which the test is functioning. If the test is in Auto Mode (*Test Mode* is false), then the test sequences through all possible channels and patterns automatically as determined by the auto sequencing VI. Thus, those rings and the array index are controlled programmatically and the user cannot change their settings. If the test is in Manual Mode (*Test Mode* is true), then the rings contain all of possible choices for channels and patterns and the array index can be changed via the up and down arrows and the user has full control of each.

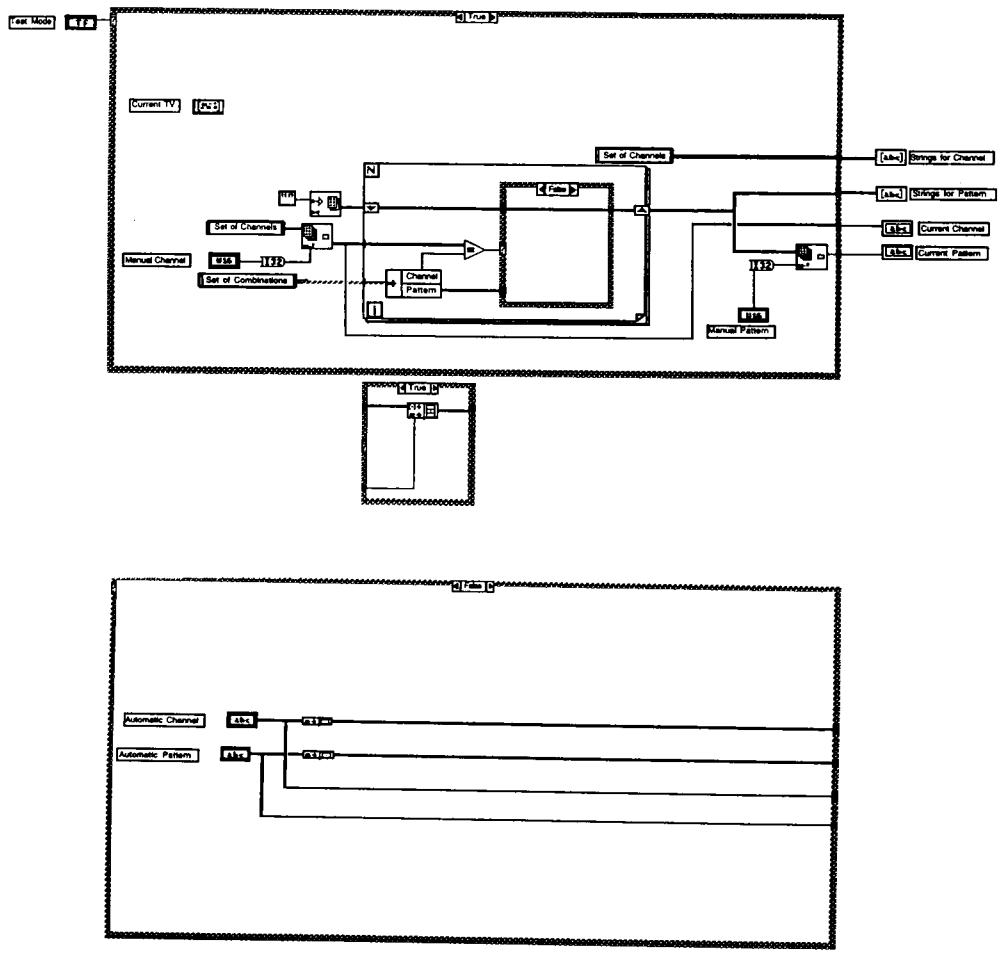
The case structure in the center of the BD is used to manipulate the array index values in Manual Mode. It is used to roll over the index value to zero when the current DUT showing is the last one. Otherwise, pressing the up arrow key would cause it to scroll upwards displaying undefined elements. For example, if there are 5 DUTs and the index value is 4 (the last one), then a push of the up arrow button will cause the index to roll over to zero.

The Level Translator VI, **lvlxla_c.vi**, shown in Figure 4.21 is used to produce a 100 μV signal level at the DUT when the slider



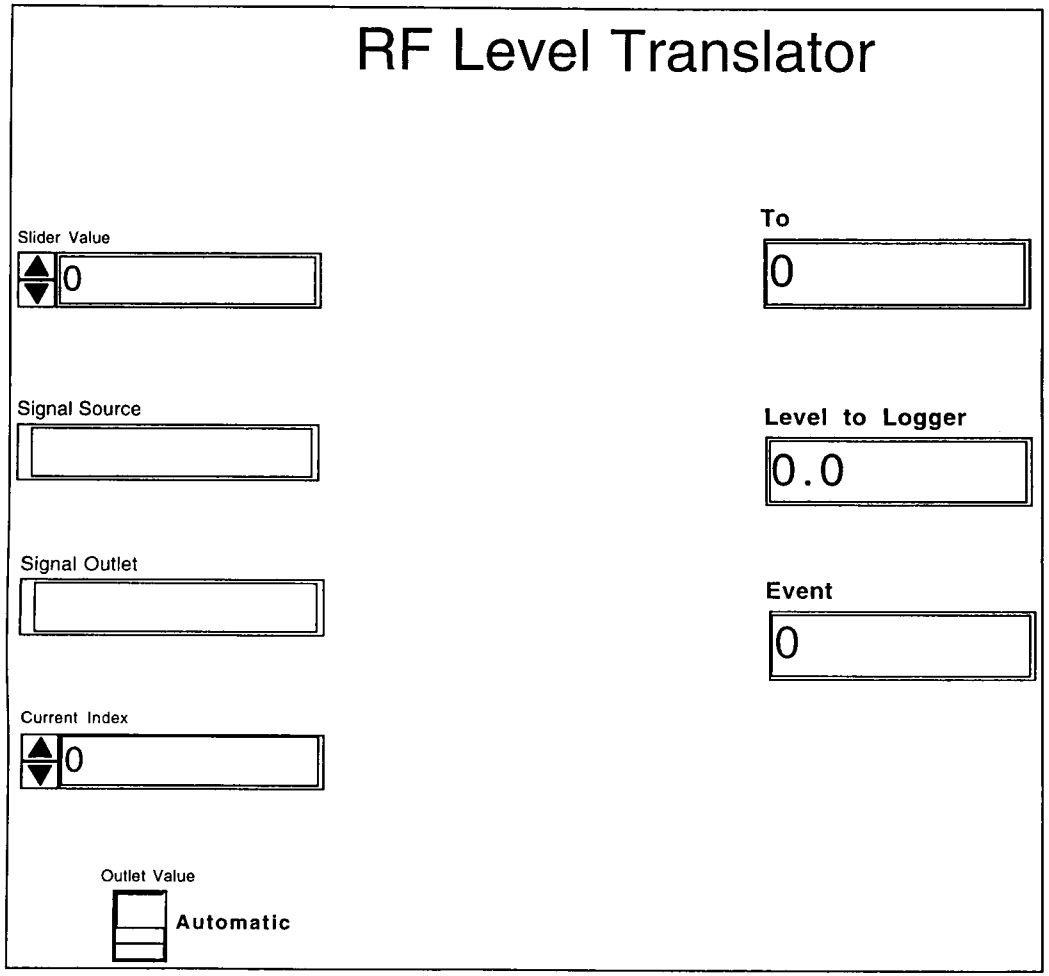
(a) Front Panel.

Fig. 4.20 Ring control VI.



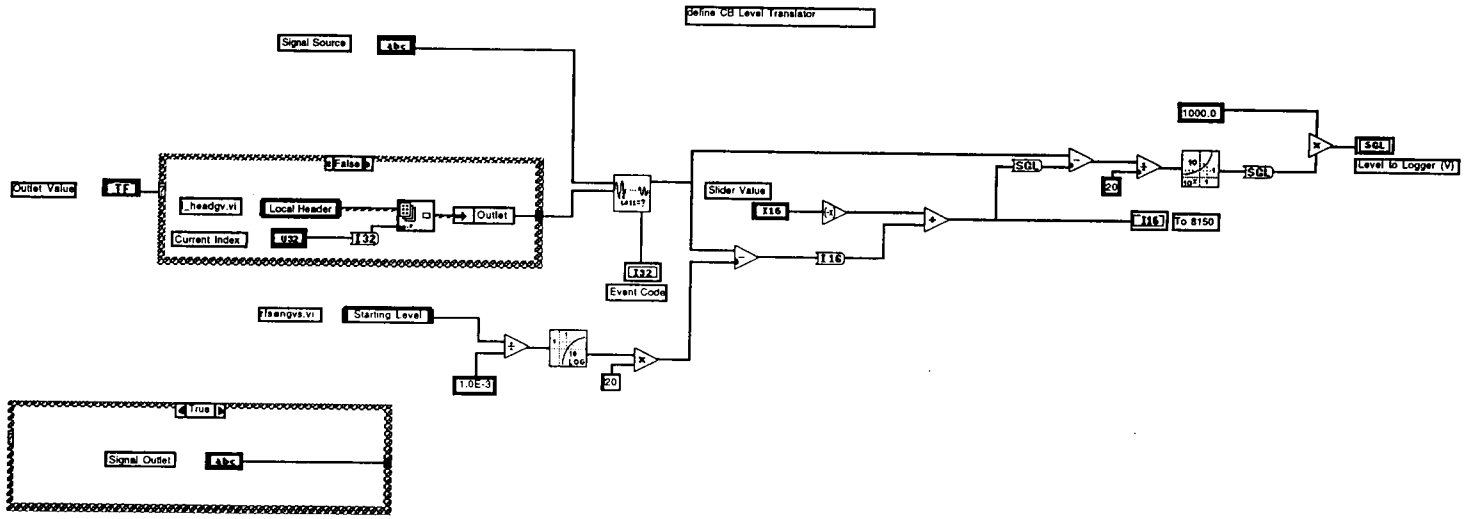
(b) Block diagram.

Fig. 4.20 (continued)



(a) Front Panel.

Fig. 4.21 Level translator VI.

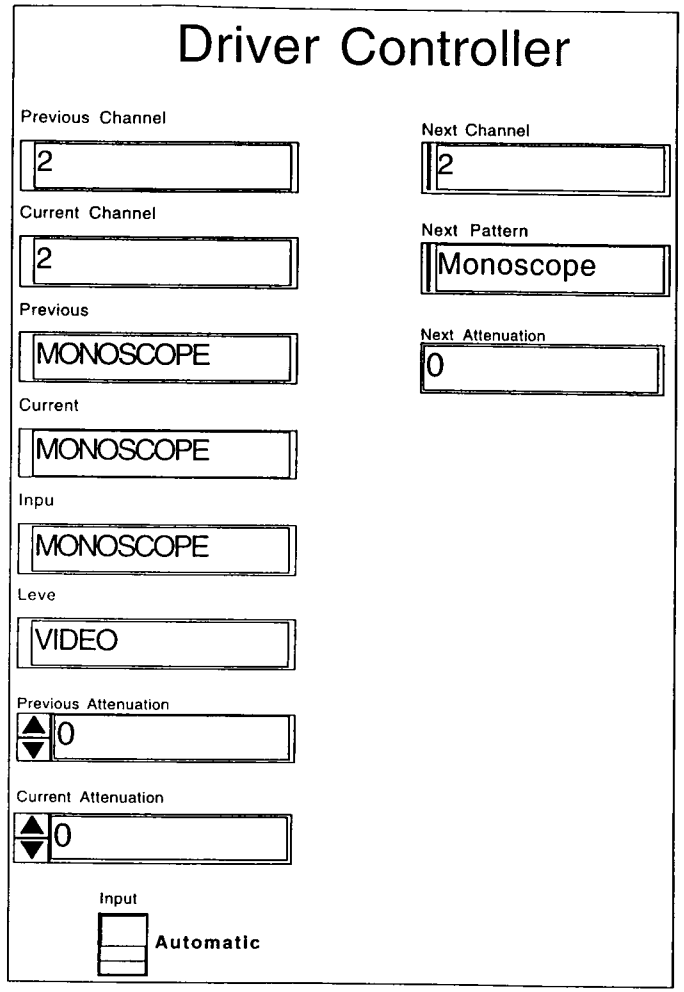


(b) Block diagram.

Fig. 4.21 (continued)

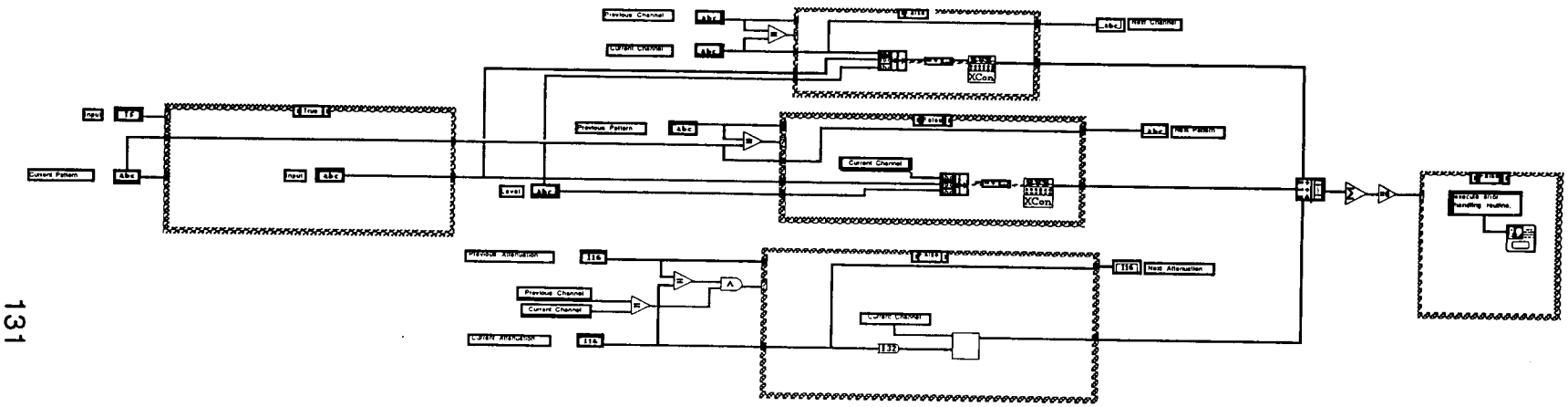
value is zero. Then, this VI factors in the slider value to determine the actual value of attenuation that must be sent to the TSI8150. It first searches the Loss Table using the given channel and outlet number to determine the signal level at that outlet when there is no attenuation in the line. It then computes the amount that must be inserted to produce the 100 μV level when the slider is zero. Then, as the slider is adjusted, it adds the slider value to this computed value to produce the desired signal level at the DUT. This actual level (in μV) is displayed on the FP in the *Level in microvolts* indicator.

Finally, the driver controller VI, **drvctl_c.vi**, will be discussed. This VI's (shown in Figure 4.22) purpose is to call the necessary drivers to make the necessary cross connections and insert the desired attenuation. This VI only calls the drivers when a change in one of the parameters (channel, pattern, or attenuation) occurs from one iteration of the while-loop to the next. Otherwise, they would be called thousands of times unnecessarily and thus the GPIB bus would be unnecessarily kept busy. So, this is where the upper three shift registers are used. They help to check whether or not the channel, pattern, and attenuation have changed since the last iteration of the loop. If so, then the respective driver is called. If the channel or pattern changes, then the SVS Router driver must be



(a) Front Panel.

Fig. 4.22 Driver controller VI.



(b) Block diagram.

Fig. 4.22 (continued)

called to make the appropriate connections on the desired channel [22]. If the attenuation changes, then the TSI8150 driver must be called [23]. Each of these calls produces an event code which must be checked to ensure that no errors were reported by the instrument drivers.

Chapter 5

Algorithm for the Airplane Flutter Test

5.1 - Introduction

5.1.1 - Overview of Airplane Flutter

The purpose of the Airplane Flutter Test is given in the Functional Specifications for the test. The purpose is to determine the ability of the TV receiver to operate under severe conditions of signal perturbations caused by signal reflections commonly referred to as "Airplane Flutter" [2]. That is, the purpose is to determine the susceptibility of a receiver to multi-path signals which are caused by reflections from airplanes flying between the transmitter and the receiver.

The system hardware configuration is given in Figure 5.1. This test requires two measurements per DUT. In each case, the flutter frequency is measured (defined in Section 2.3). The first measurement is made with a signal level of 500 μV at the DUT and the second is made with a signal level of 5 mV.

The signal setup requires that a standard Monoscope pattern with 87.5% modulation depth be routed to each DUT. No aural signal is required. Also, the flutter modulation must be 50%. This refers

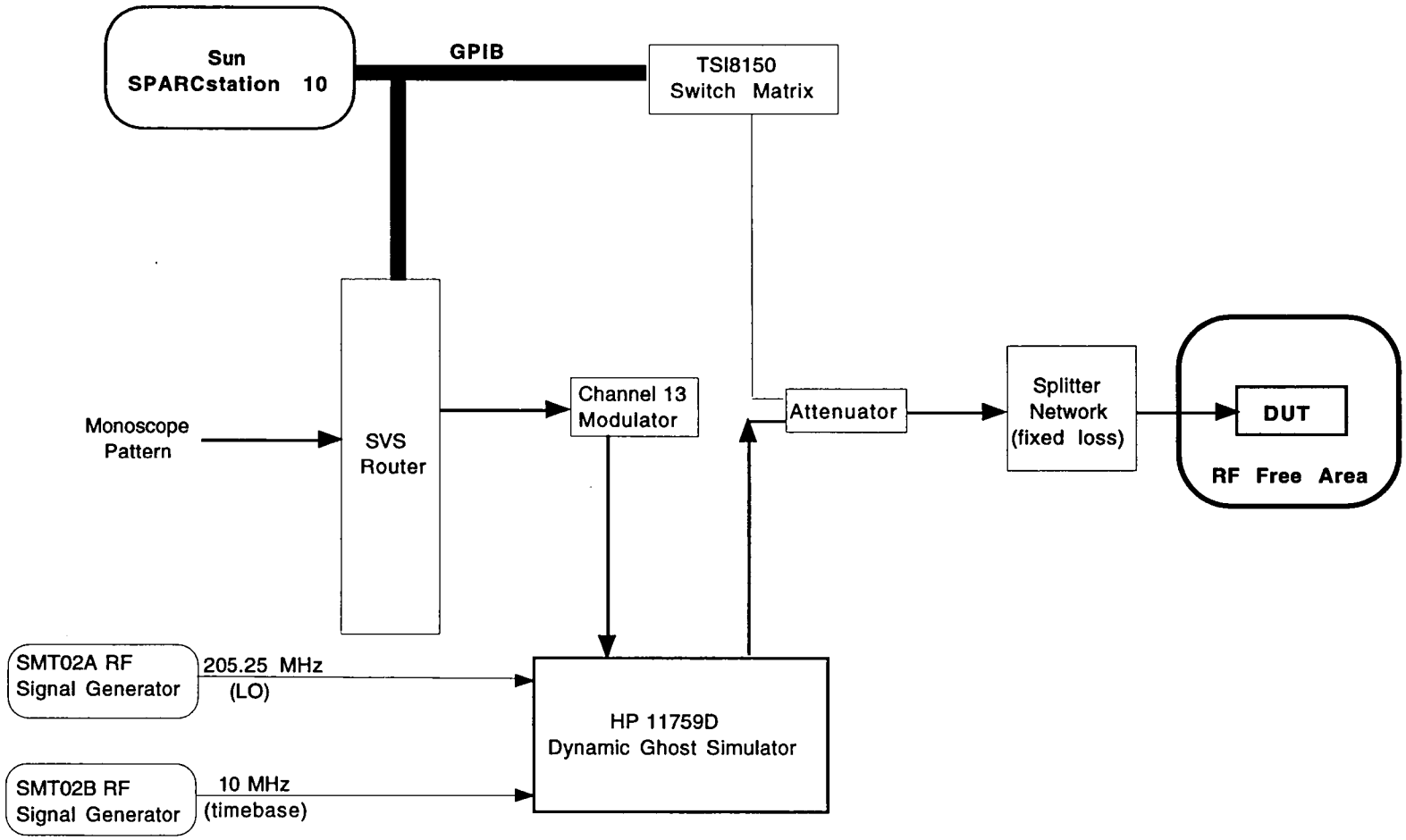


Fig. 5.1 Airplane Flutter test configuration.

to the peak-to-peak voltage level of the sinusoid modulating the 6 MHz signal in each path of the simulator. So, the peak-to-peak signal level of this sinusoid should be 50% of the peak-to-peak value of .the video signal (this sinusoid was discussed in Section 2.3). Then, each receiver has a characteristic frequency for this sinusoid beyond which it is not able to lock onto the sync pulse in the video signal. This characteristic frequency at which this loss of synchronization occurs is known as the flutter frequency.

To make each measurement, the frequency should be increased until the flutter frequency is reached. This is the Doppler frequency value inside the ghost simulator (see Chapter 6). The flutter frequency is recorded for that DUT at the given signal level.

As in the case of RF Sensitivity (Chapter 4), the Airplane Flutter test is also located in the *RF Series* of tests. So, it is executed by following the same menu tree as described in Section 4.1.

5.1.2 - Template Concept for Test Design and Architecture

During the course of the research, it became apparent that there were many subroutines that could be duplicated (or slightly modified) and used in all of the test procedures. Therefore, it became advantageous for a "Template" to be designed as a skeleton

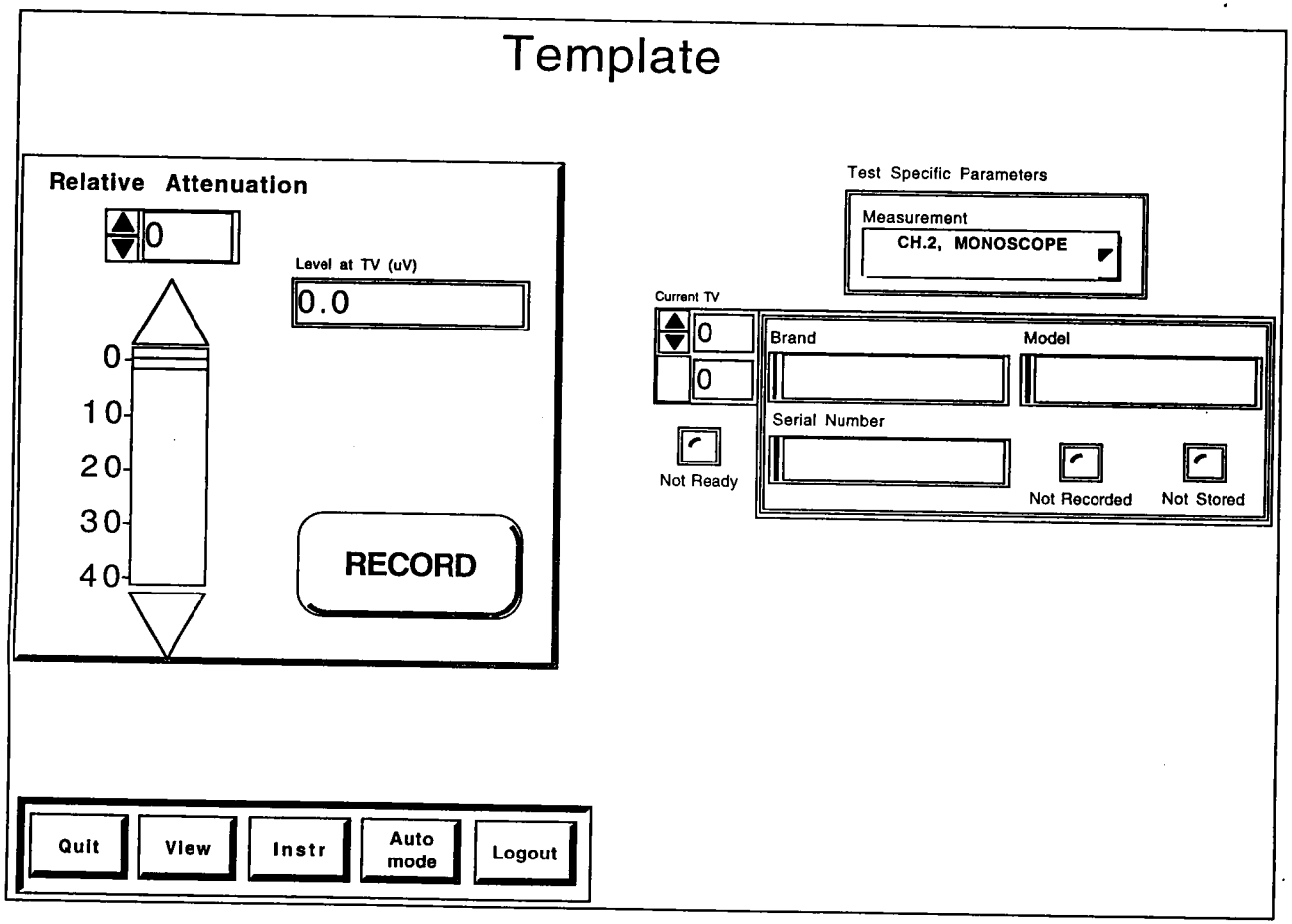
for all tests. Then, instead of the designer having to completely design each test from scratch, he could start with this template and tailor it to his particular needs. This is analogous to the NI templates for instrument driver design discussed in Section 3.4

The Airplane Flutter test (shown in Figure 5.2) was the proverbial guinea pig for this template concept. It was used as a basis for developing not only the automated test but also the template. Therefore, the following sections provide integrated discussions on the template as well as the Airplane Flutter Test. The layout of the remaining sections is very similar to those in Chapter 4.

5.2 - BIT Check Stage

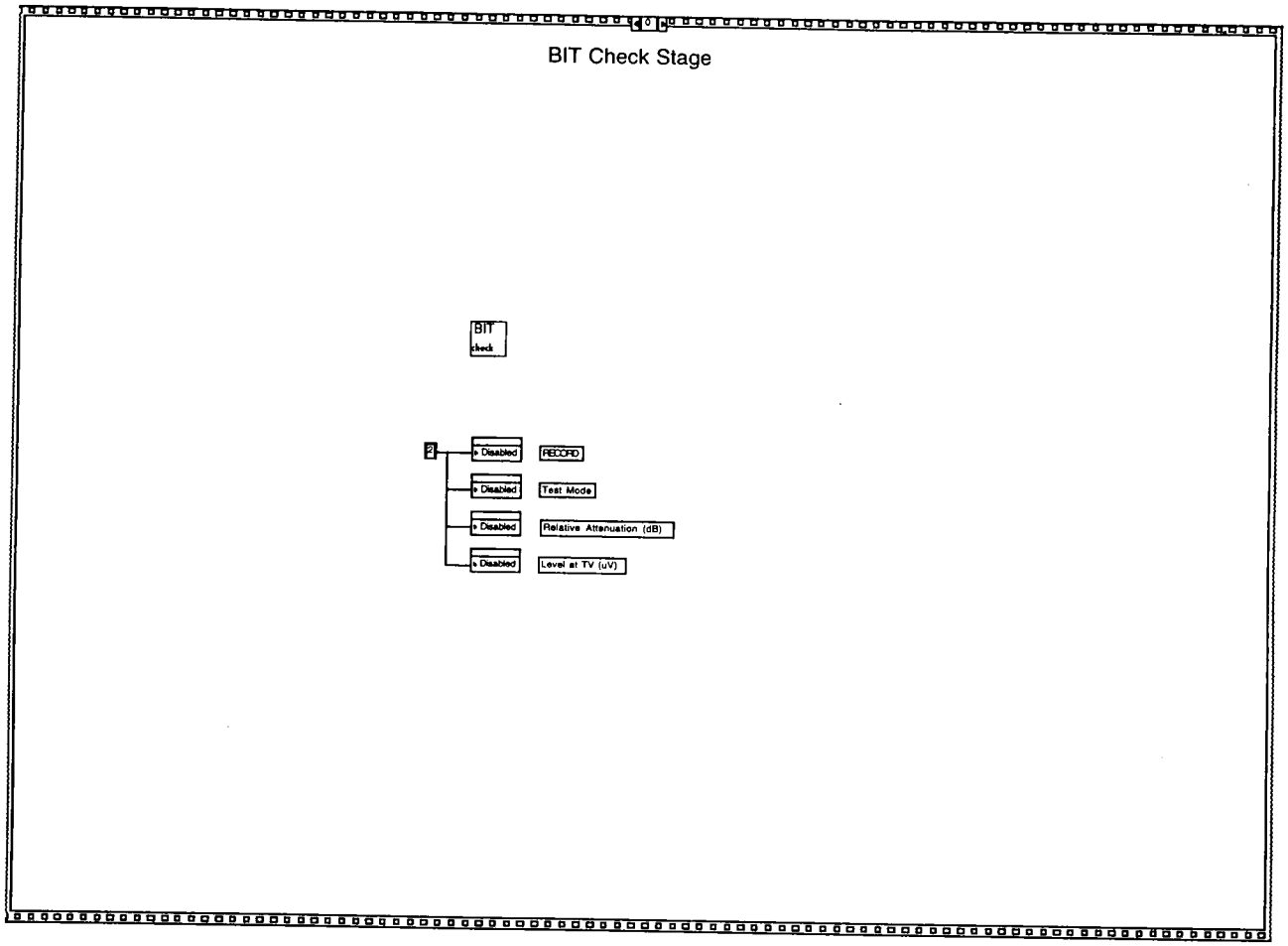
This stage is similar to the Error Checking stage discussed in Section 4.2. The only VI is **bit_chek.vi.** At the time of this development, the BIT functions were not yet developed and so the interface to them was not yet defined. Therefore, currently, this VI is basically empty as shown.

The functionality may be described as follows. This VI will take the list of equipment supplied in the test global variable (Figure 5.3) and cross check it against the results of the BIT. It will analyze its findings and output a single boolean value *BIT Forward.*



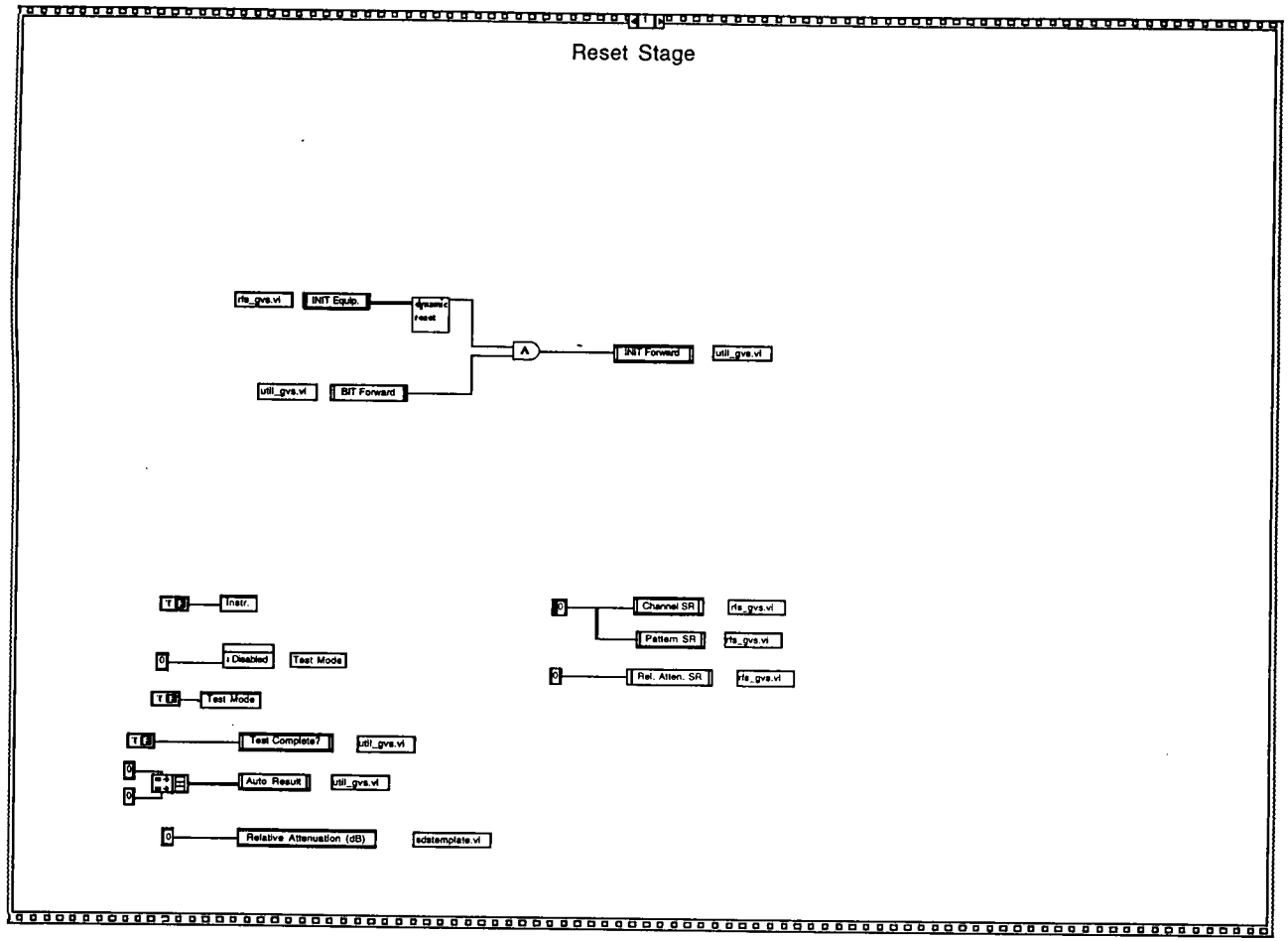
(a) Front Panel.

Fig. 5.2 Airplane Flutter VI.



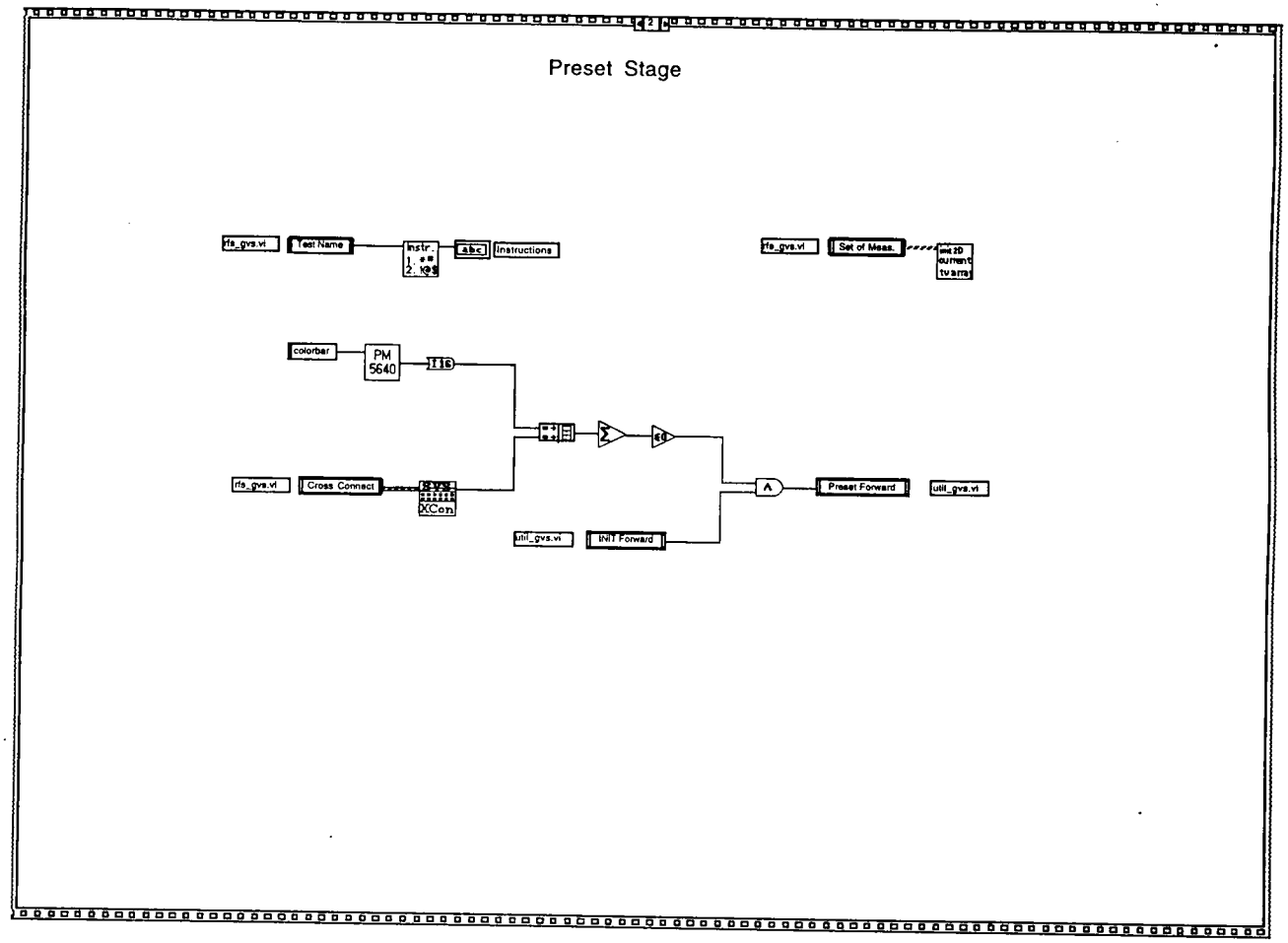
(b) Block diagram.

Fig. 5.2 (continued)



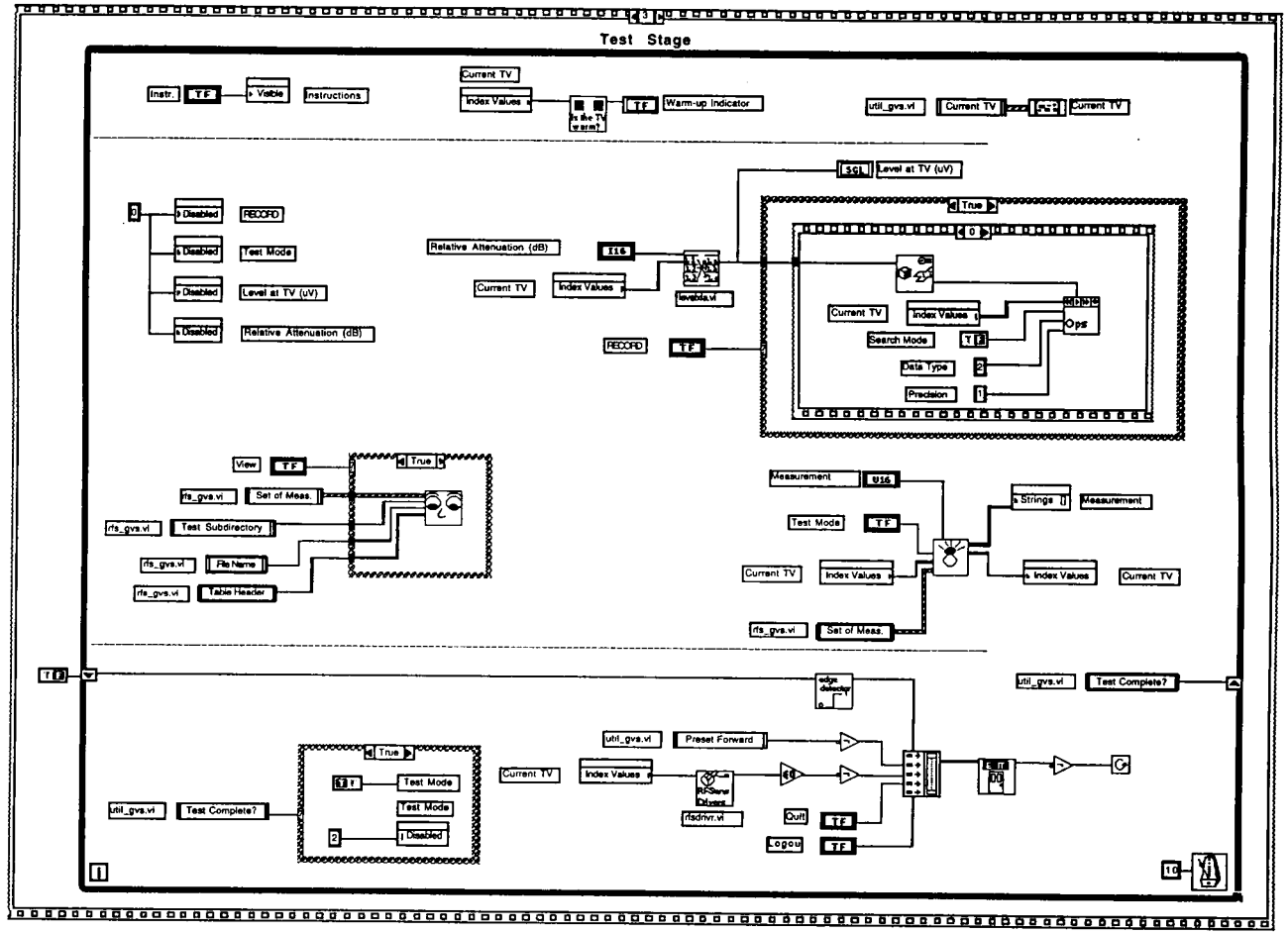
(b) (continued)

Fig. 5.2 (continued)



(b) (continued)

Fig. 5.2 (continued)



(b) (continued)

Fig. 5.2 (continued)

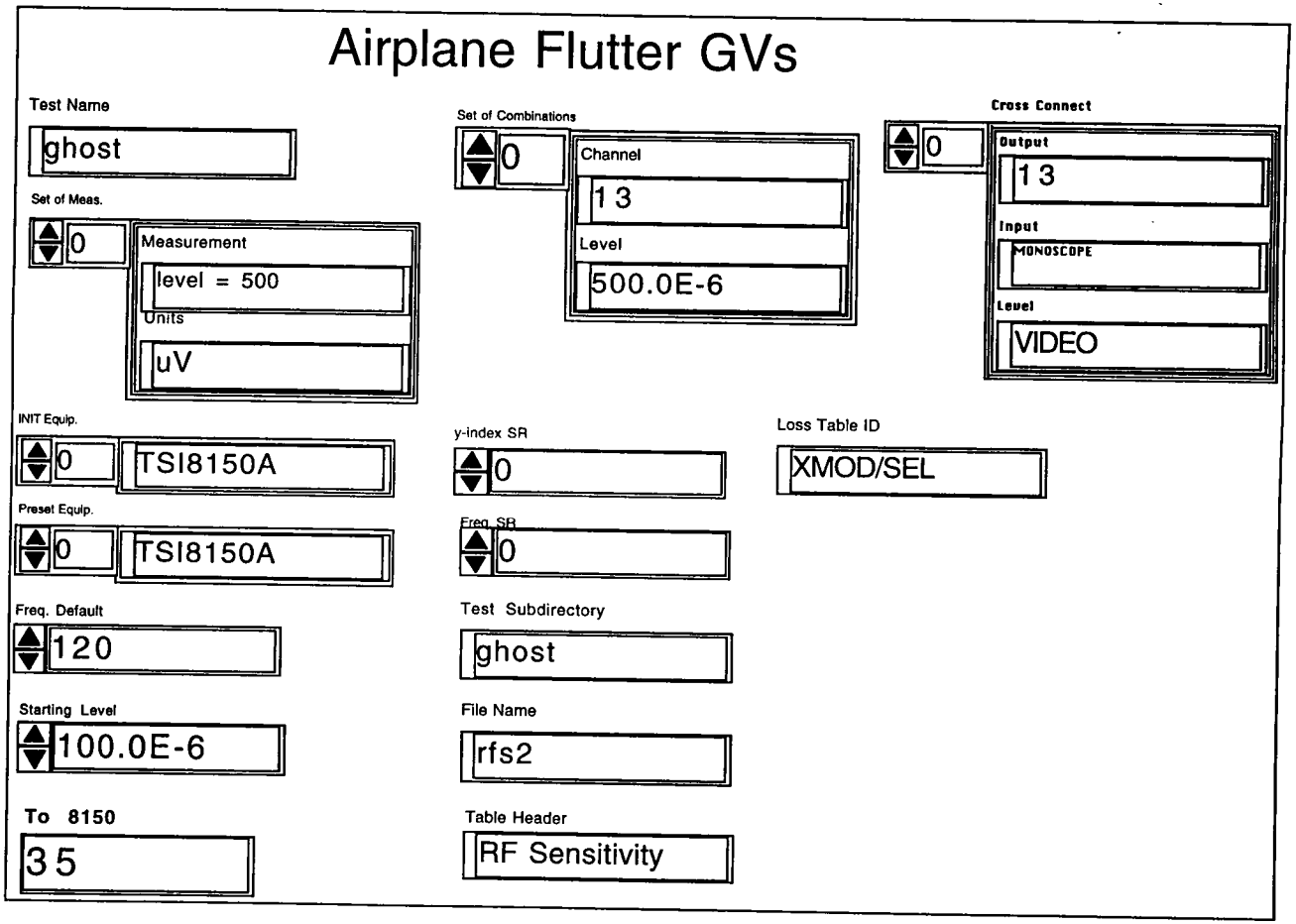


Fig. 5.3 Airplane Flutter global variables.

If the value is true, then the BIT found no errors with the hardware check and the test may continue. Otherwise, the test will be aborted because some of the hardware did not pass the BIT.

One of the advances over the previous test design is the elimination of much wiring. Global variables are used more frequently to help "clean up" the diagrams. So, instead of passing the value out from the right side of the diagram as in the Error Checking stage, it is written to a global variable. This value will be read by the Reset Stage.

The remaining logic in the BD controls the *RECORD* and *Test Mode* button as well as the test specific control *Frequency*. The value of 2 in the *Disabled* attribute node means that these will be disabled and grayed-out. This is done because the BIT, Reset, and Preset stages take several seconds to execute. So, the user is locked out until the Test Stage is reached. The grayed-out controls signal him that these activities are running in the background and so he cannot interact with the FP yet.

Another convention in the template is that all test specific controls, indicators, attribute nodes have highlighted labels to separate them from those that belong to the template. In addition, the test specific VIs have the same background color in their icons whereas the template VIs all have the same dark green background

color.

This Reset Stage is repeated in the last step of the sequence structure in order to restore the hardware to a known condition. It was determined by the team that this is a necessary operation in order to maintain the soundness of all data by the ATF. This final reset helps in making the facility more user friendly in that after any test has complete its run, the system is configured to a known, reset condition.

5.3 - Reset Stage

This stage is analogous to the Reset and Initialization step in the RF Sensitivity Test. Each test uses this stage to reset all of the hardware that it will use. Since these initialization drivers for the instruments are the same for all tests, it was observed that there was no need to actually have a given driver VI in each test that used it. As the number of tests grows, such duplication would unnecessarily use memory.

Therefore, the initialization drivers are dynamically executed. That is, they are brought into memory, called, and released from memory. This is done by **dyn_rest.vi**. This VI is shown in Figure 5.4. All it requires is a list of equipment that needs to be reset. This is given in the *INIT Equip.* global variable. It takes this list of

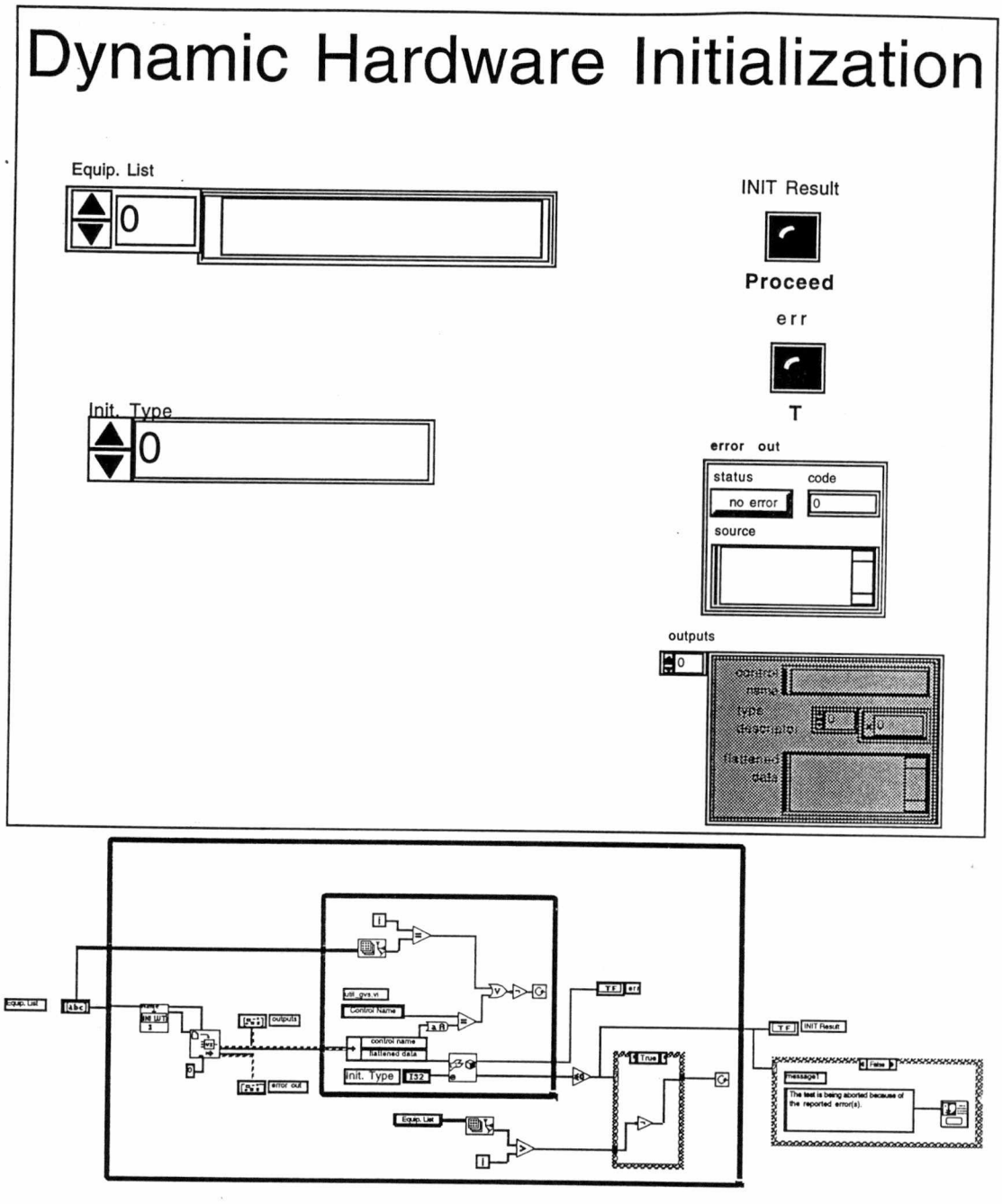


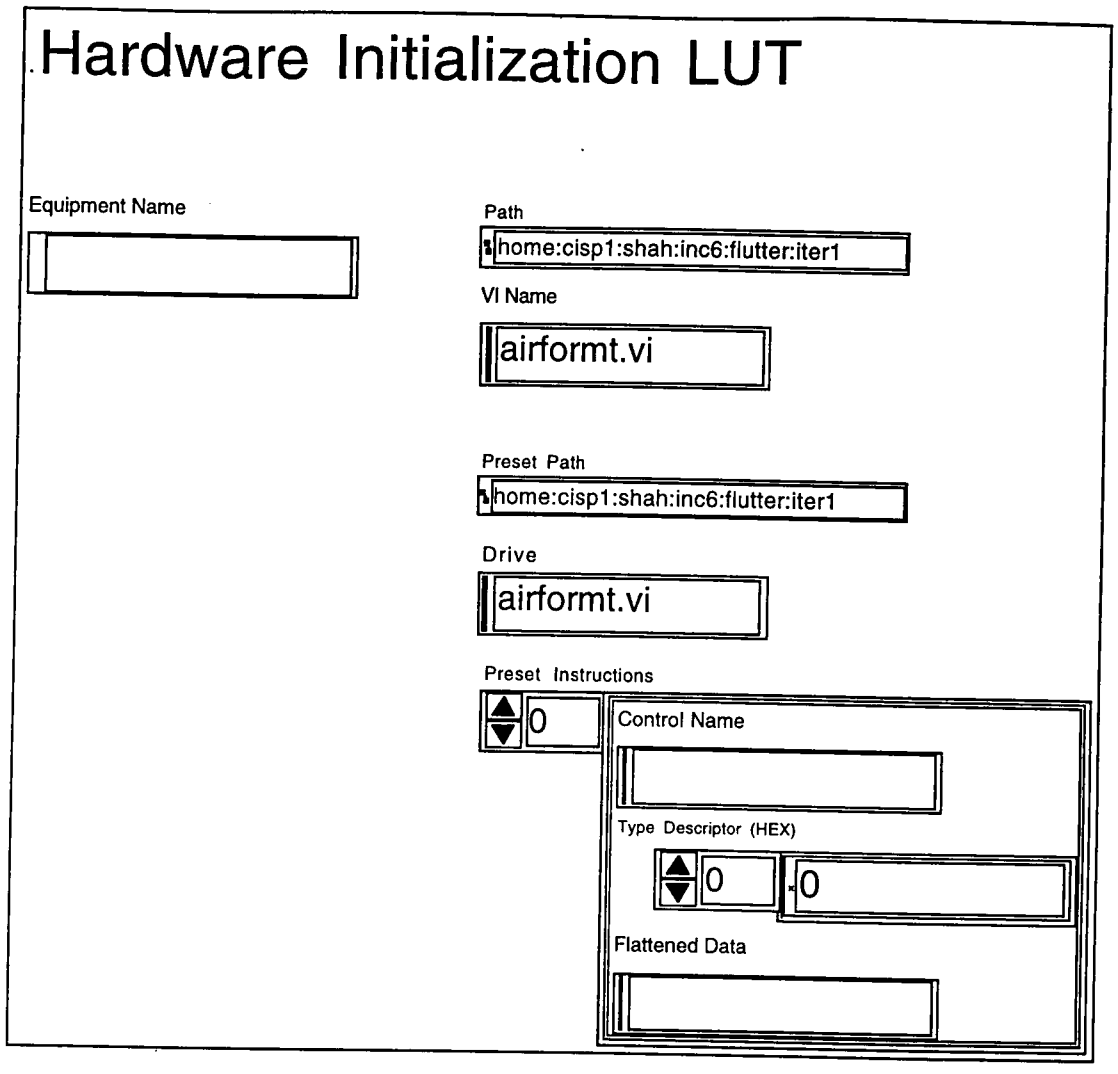
Fig. 5.4 Template dynamic reset VI.

equipment and searches through a look-up-table (LUT) to find the corresponding initialization VI to call. This search routine is performed in **init_lut.vi** shown in Figure 5.5. The latter VI outputs the *VI Name* and the *Path*. It also has some other outputs but these will be used in future work.

In order for it to always point to the same VIs, this list must be made using standard names for each piece of equipment. Otherwise, it may point to the wrong entry in the LUT or not find the entry at all. Once the entry is found, the *VI Name* and *Path* are passed to the dynamic run VI which loads the file, executes, and unloads.

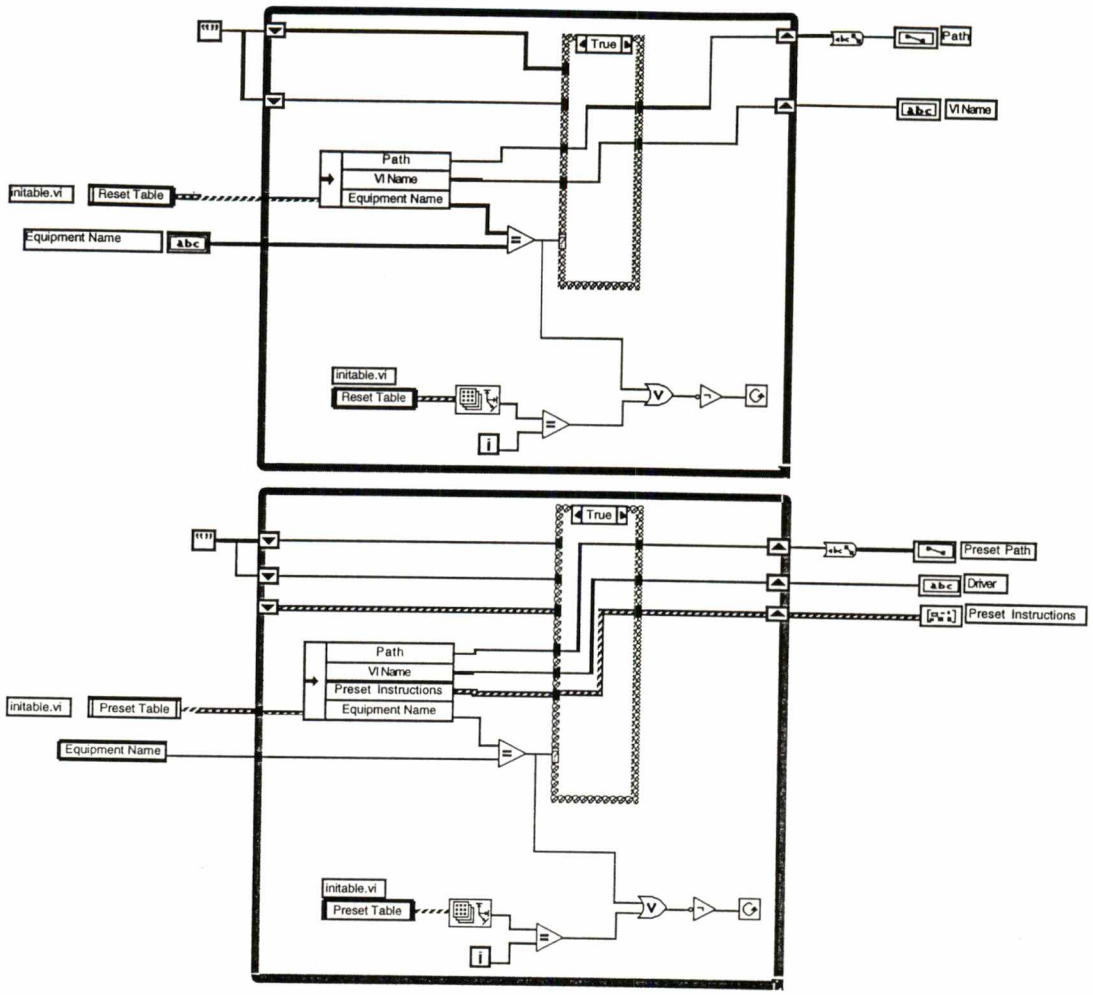
The inner while-loop is halted when one of two conditions exist. It is halted if the iteration count of the loop is equal to the number of items in the hardware list; that is, if all of the equipment has been initialized. Otherwise, for each piece of equipment, it looks at the all of the outputs and tries to match *control name* with the GV *Control Name* (shown in Figure 5.3). It is looking for the control *Event Code* which is output by all driver VIs. When it finds it, the loop is halted and value of *Event Code* is output by unflattening *flattened data*.

Whether or not this value is less than or equal to zero determines the result of the Reset Stage, *Init Forward*. As a



(a) Front Panel.

Fig. 5.5 Template initialization look-up VI.



(b) Block diagram.

Fig. 5.5 (continued)

convention, no error is indicated by a zero (or less than zero) event code. Therefore, if *INIT Forward* is true, then all of the initialization VIs executed without error and the test may proceed. If it is false, then there was one or more errors. This will be relayed to the user by the VIs where they occurred. So, the message box notifies him that the test is being aborted because of these errors.

5.4 - Preset Stage

The Preset Stage is almost identical to its counterpart in the Sensitivity test. It is shown in sequence 2 of the BD. The purpose of this stage is to configure the equipment to the necessary states for this test. Then, it checks all of the event codes from the drivers and ensures that they were all less than or equal to zero. If they were and if the *INIT Forward* is true, then the test may proceed and this is indicated by a true value in *Preset Forward*. Otherwise, the test will be aborted.

For Airplane Flutter, the required router configuration is to have the Monoscope pattern output on Channel 13. This is given in the *Cross Connect* global variable shown in Figure 5.3. The TSI8150 driver is called to setup the system in the same way it would for the Nonlinear Distortion Test. The SMT02A RF output is set for a 205.25 MHz signal at a level of +10 dBm while the SMT02A RF output is set

for a 10 MHz signal at a level of +10 dBm. These serve as the LO and Timebase inputs for the Ghost Simulator which is setup for the Simulation Test with the Spectrum set to Doppler Mode and an initial Doppler frequency of 120 Hz. It should be noted that the drivers for the SMT02s were not yet developed. Thus, they had to be configured manually and are not shown in the BD.

Also, shown in the BD are the two other VIs. One of these is the **instpanl.vi** shown in Figure 5.6. The purpose of this VI is to read the instructions file for this test and output these instructions to a string indicator on the test panel which is turned on and off via the *Instr* button. It accepts the *Test Name* as the only input and appends "_in" as a suffix. Dot extensions were avoided in order to facilitate platform portability. It reads this "Test Name_in" located at *Path to Instr. Files* and outputs the data in *Instructions*.

Also shown is **ini_2d.vi**. This VI was written to initialize the 2D data array used in all of the tests to keep track of measurements. It sets all numeric fields to 0, all measurement fields to their default value, and all booleans to false.

This 2D data array (see Figure 5.7) is similar to that one used in the Sensitivity test in that it keeps the header information and the *Valid Data?* and *Data Stored?* flags. However, instead of the last elements being test specific, it has an array of strings with two

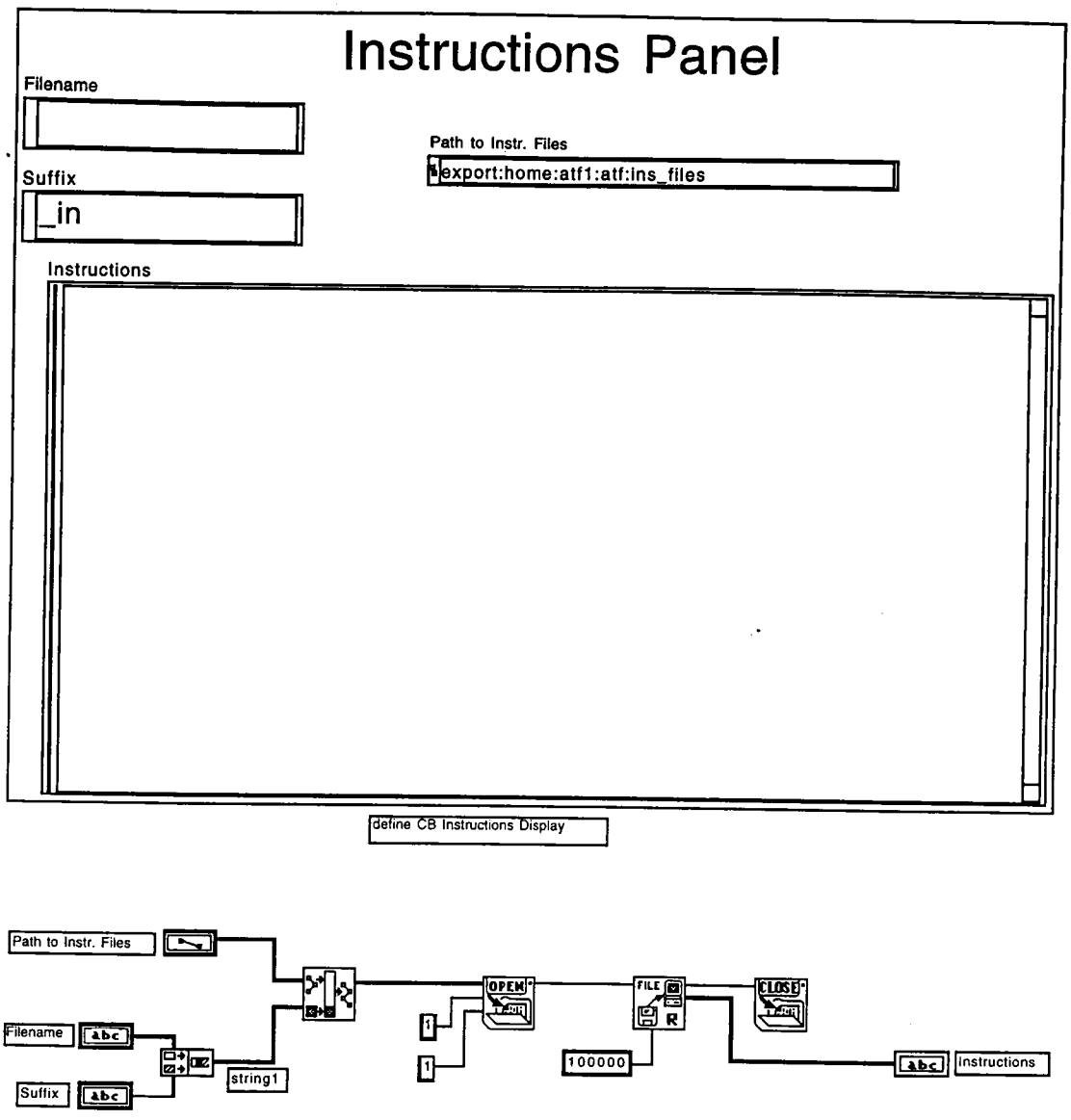


Fig. 5.6 Template instructions VI.

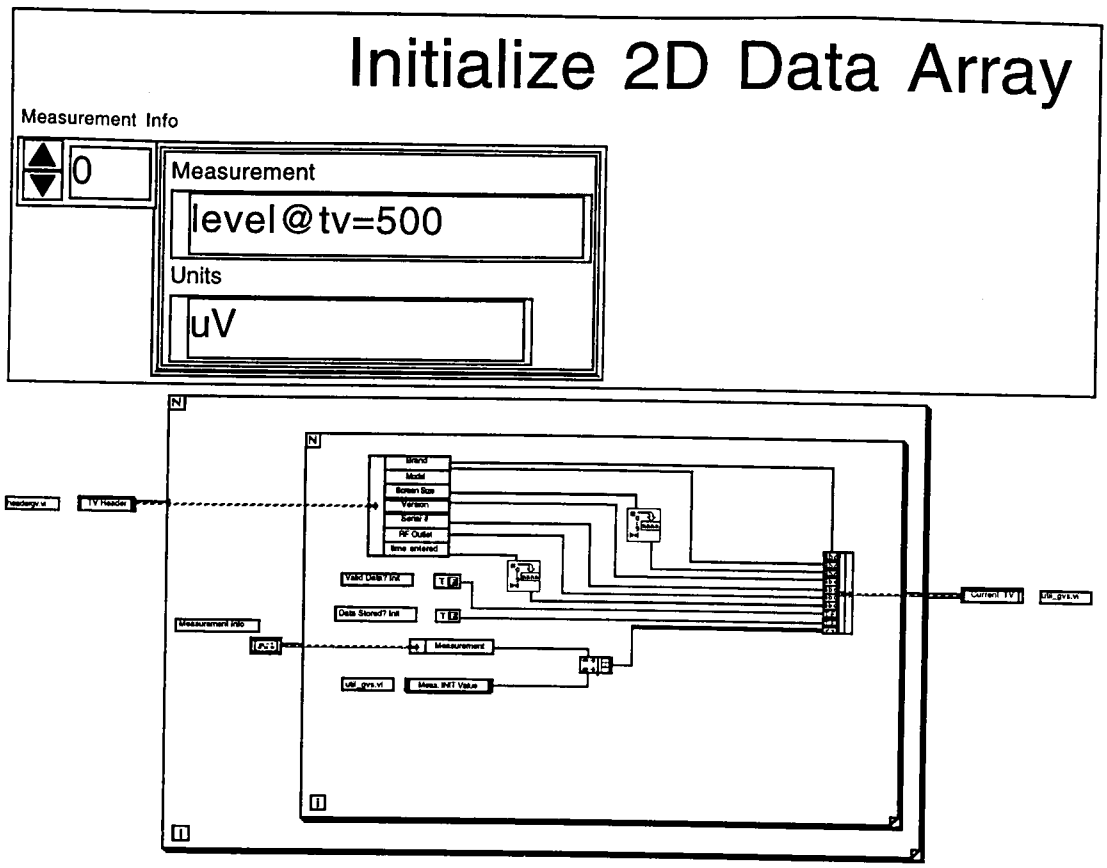


Fig. 5.7. Template initialization 2D array VI.

elements. The first is the name of the measurement (e.g., level = 500 μV or Ch.2, Audio) and the second is the value of the measurement (e.g., 120). This same 2D array is used by all test to keep track of measurement data.

By using the array of strings to maintain the measurement name and value, it became generic to all tests. Therefore, each designer does not have to change this to be specific to his test as was the case before the template concept. For example, in the RF Sensitivity Test, the last elements in the 2D array cluster maintained the Channel, Pattern, and RF Level. For the Airplane Flutter test these would be Signal Level and Flutter Frequency. Using the generic approach, both tests would use the same 2D array wherein the array of strings would have the two elements described above, measurement name and value.

5.5 - Test Stage

This stage is where the most test-specific code is required. The main test-specific VI is the driver controller (discussed below). This is where all of the calls to the necessary drivers are made (e.g. attenuation changes to the TSI8150 and signal connections made by the SVS Router). In addition, any other peripheral VIs necessary to the test are called in this stage. An example of such a peripheral VI

is the level translator VI used in the RF Sensitivity test to produce a 100 μV signal at the DUT. Otherwise, all other VIs in this stage are independent of a particular test.

The first VI to be discussed is **warmchek.vi** (Figure 5.8). This VI was written to determine whether or not the current device under testing is sufficiently warm. It accepts the index values as the only input and uses that to parse out the *Time Stamp* for the current TV. It uses this *Time Stamp* as an input to the **warm.vi** which compares this value to the value when the user logged in. If the difference is greater than 15 minutes (default), then *Warm-up Indicator* is false; otherwise, it is true. This assumes that the tester had actually turned on the DUTs prior to logging in. This is merely an indicator as whether or not 15 minutes has elapsed since login, not whether the DUTs have been on for 15 minutes.

The attribute nodes that were set to a value of 2 in the BIT Check Stage are now set to a value of 0 which means they are not are enabled. It is all right for the user to interact with these because the initialization and presetting is complete.

hpglevel.vi is shown in Figure 5.9. This VI was written to produce the appropriate signal level at the DUTs during the Airplane Flutter Test. Its functionality is similar to the **levelxla.vi** used during the Sensitivity Test.

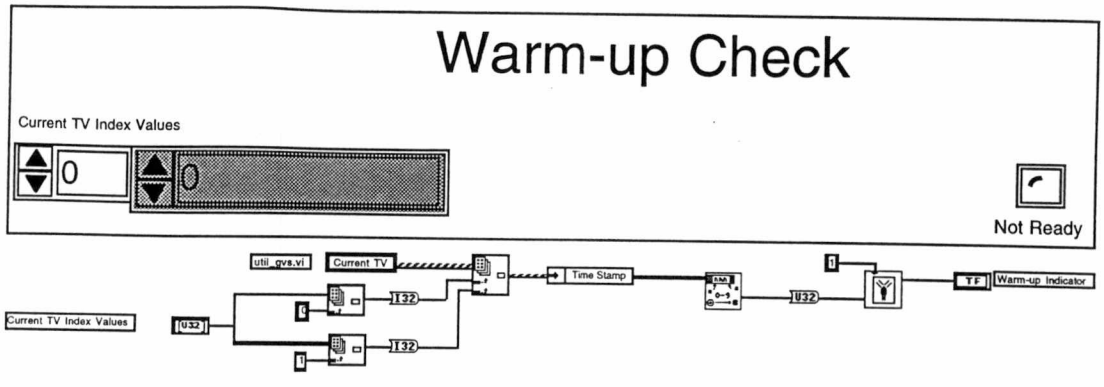


Fig. 5.8 Template warm-up check VI.

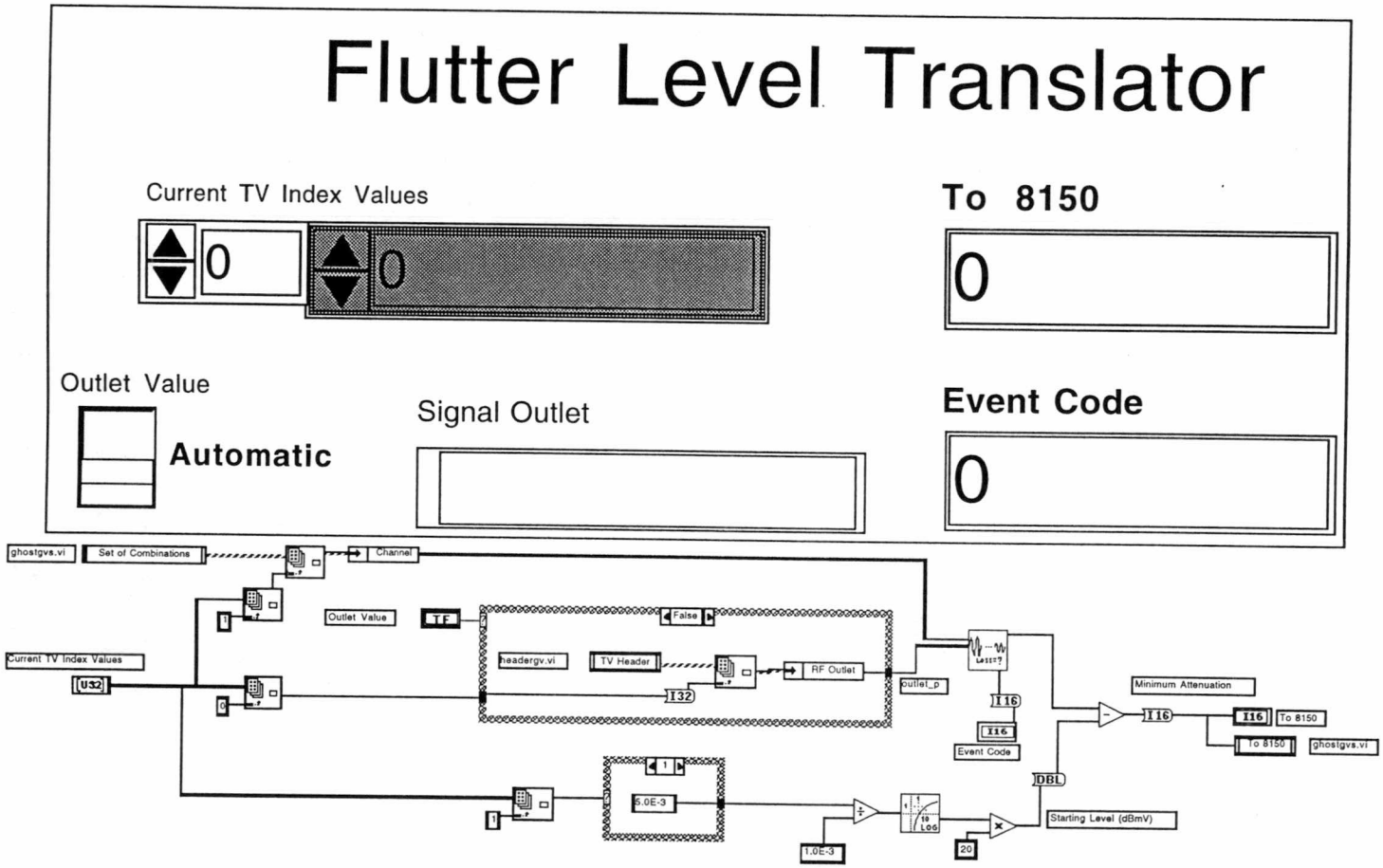


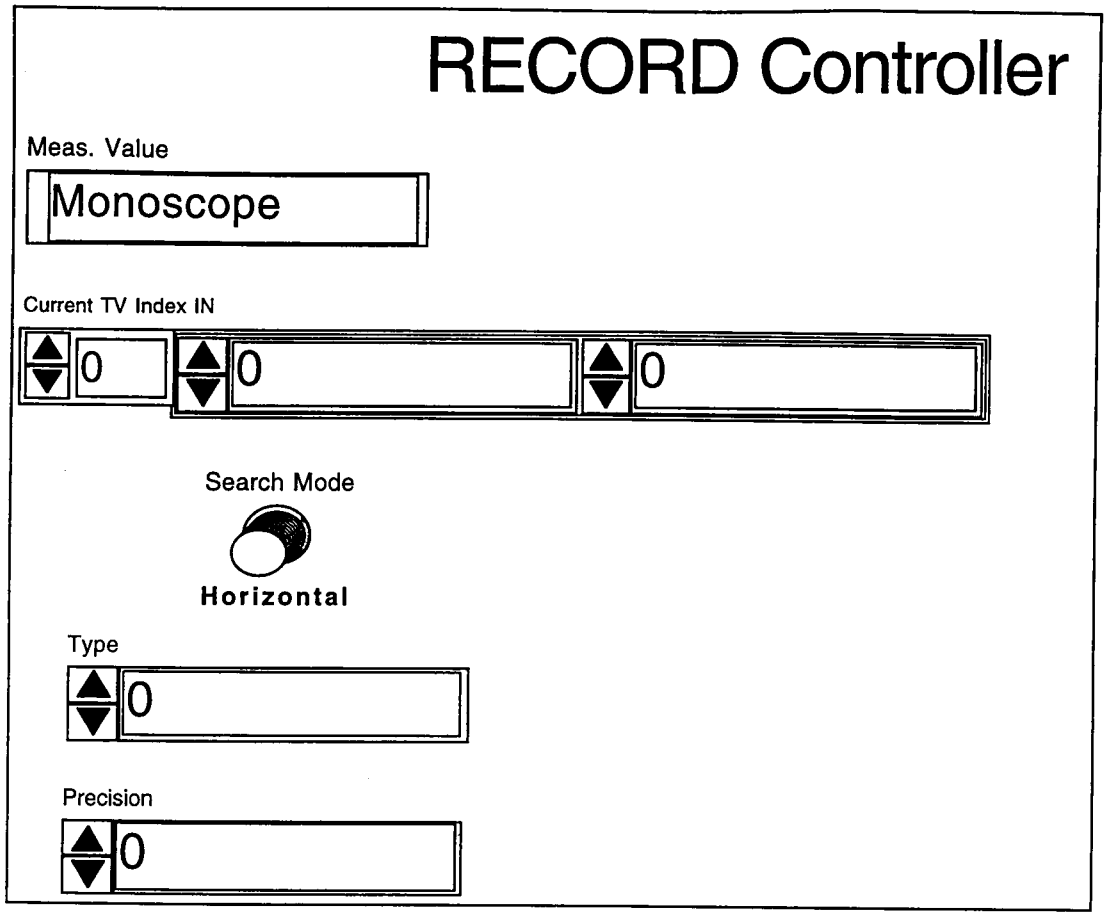
Fig. 5.9 Airplane Flutter level translator VI.

The large case structure in the upper right is controlled by the *RECORD* button. If the button is not pressed, the false case is executed in which nothing happens. When it is pressed, the true case calls in sequence 0 **rec_ops.vi** shown in Figure 5.10. This VI was written to fulfill a two-fold purpose. It is responsible for recording the measurement values and setting the *Valid Data?* flags. Also, it computes the next step in the auto sequencing.

The **Flatten To String.vi** is used in order to make this operation generic to all tests. This VI accepts any data type as its input which is convenient because some tests use integer numbers and some use floating point numbers to represent measurement values. This numeric value is then converted to a platform specific string of flattened data.

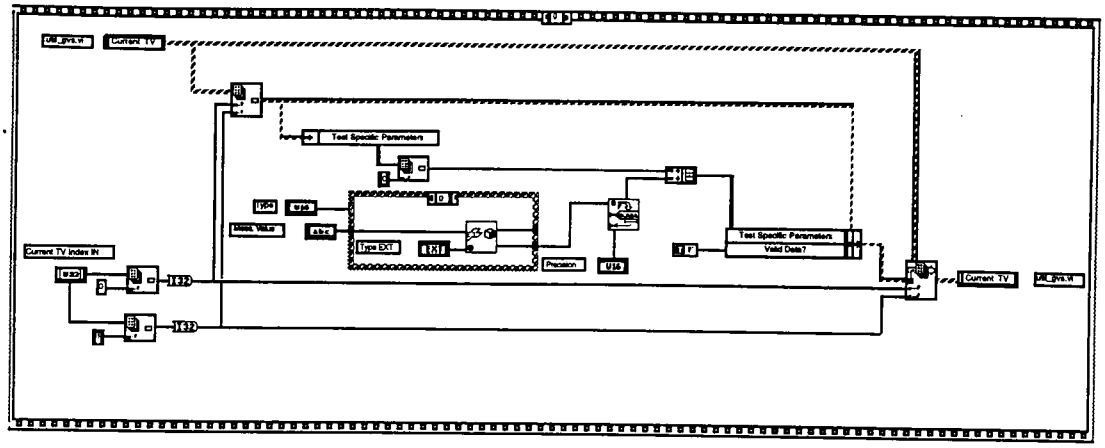
The **rec_ops.vi** program accepts this flattened data as one of its inputs in sequence 0. Internally, these data are unflattened and converted to a regular string according to the constants *Data Type* and *Precision*. Then, the index values input is used to parse the appropriate element of the 2D array and the measurement value is inserted into the second element of the Test Specific Parameters (i.e., array of strings) and then that entire element of the 2D array replaces the original one.

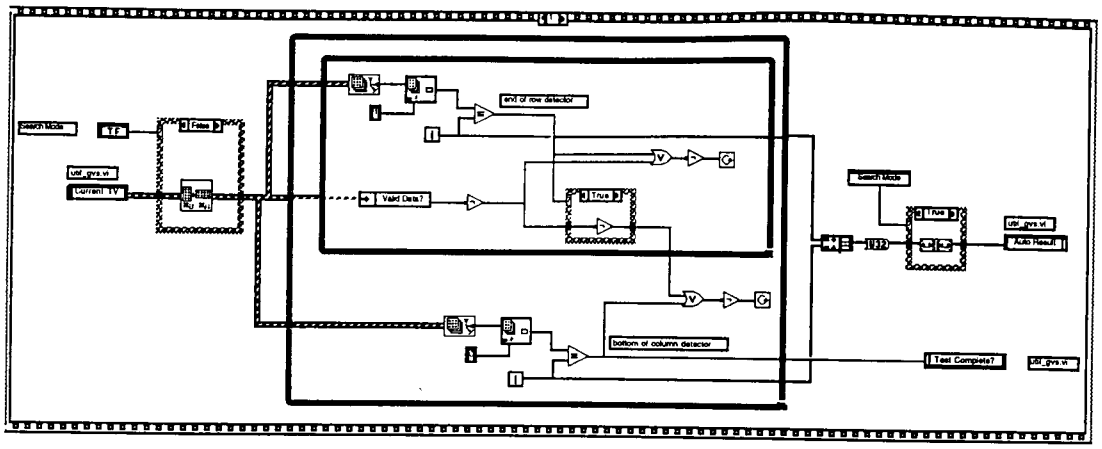
Sequence 1 of the VI is where the auto sequencing



(a) Front Panel.

Fig. 5.10 Template record operations VI.





(b) Block diagram.

Fig. 5.10 (continued)

computations occur. The *Search Mode* input determines whether the algorithm looks for the next measurement from left-to-right and top-to-bottom (*Search Mode* = false) or from top-to-bottom and left-to-right (*Search Mode* = true). In the false case, the 2D array is transposed first and then is searched while in the true case it is not transposed. So, the search routine actually always runs from left-to-right and top-to-bottom. It is this transposition which allows the option of which way to search. This option is necessary because some tests require that for a given measurement all receivers be tested before proceeding to the next measurement while other tests require that all measurements for a given receiver be made before proceeding to the next receiver. Therefore, the search algorithm has this flexibility. In addition, if an intermediate routine is necessary, then a peripheral VI can be designed to programmatically determine this *Search Mode* input rather than having a constant value.

The algorithm searches through the 2D data array examining the *Valid Data?* boolean. It outputs the index numbers in *Auto Result* of the first instance where it finds that boolean to be false. If it finds that all of these are true, then *Test Complete?* is set true signifying that all measurements for all receivers have been made.

The case structure in the center-left of the BD is actuated by the *View* button. The false case is empty. The true case calls

vw_formt.vi (Figure 5.11). This VI was written to format the measurement data in the manner prescribed by the view-print-save utility [24]. This VI is also independent of the various tests. It requires the *Set of Meas.* in order to build the row headers. The designer must also supply the *Test Subdirectory* where the data will be stored and a title for the table. The VI outputs an event code which it receives from the utility.

The view-print-save utility is shown in the center of the BD. The utility fills in the elements of a standard table used by all tests to display measured data. A sample is shown in Figure 5.12. It fills in the column headings using the information from the header global variable, *TV Header*. It produces the row headers from the *Set of Meas.* input. The data elements are taken from the 2D data array. They are extracted from the *Test Specific Parameters* and are supplied to the utility if *Valid Data?* is asserted. Otherwise, *No Meas. Ind.* is supplied which is by convention an asterisk (*). Therefore, an asterisk in the display indicates that a particular measurement has not yet been made for the particular receiver.

If the user presses *Save* on the display FP, then the measurement data are written to disk in CSV format under a hexadecimal filename which is the login time converted to HEX with a .csv extension. These files are stored in a data directory in the

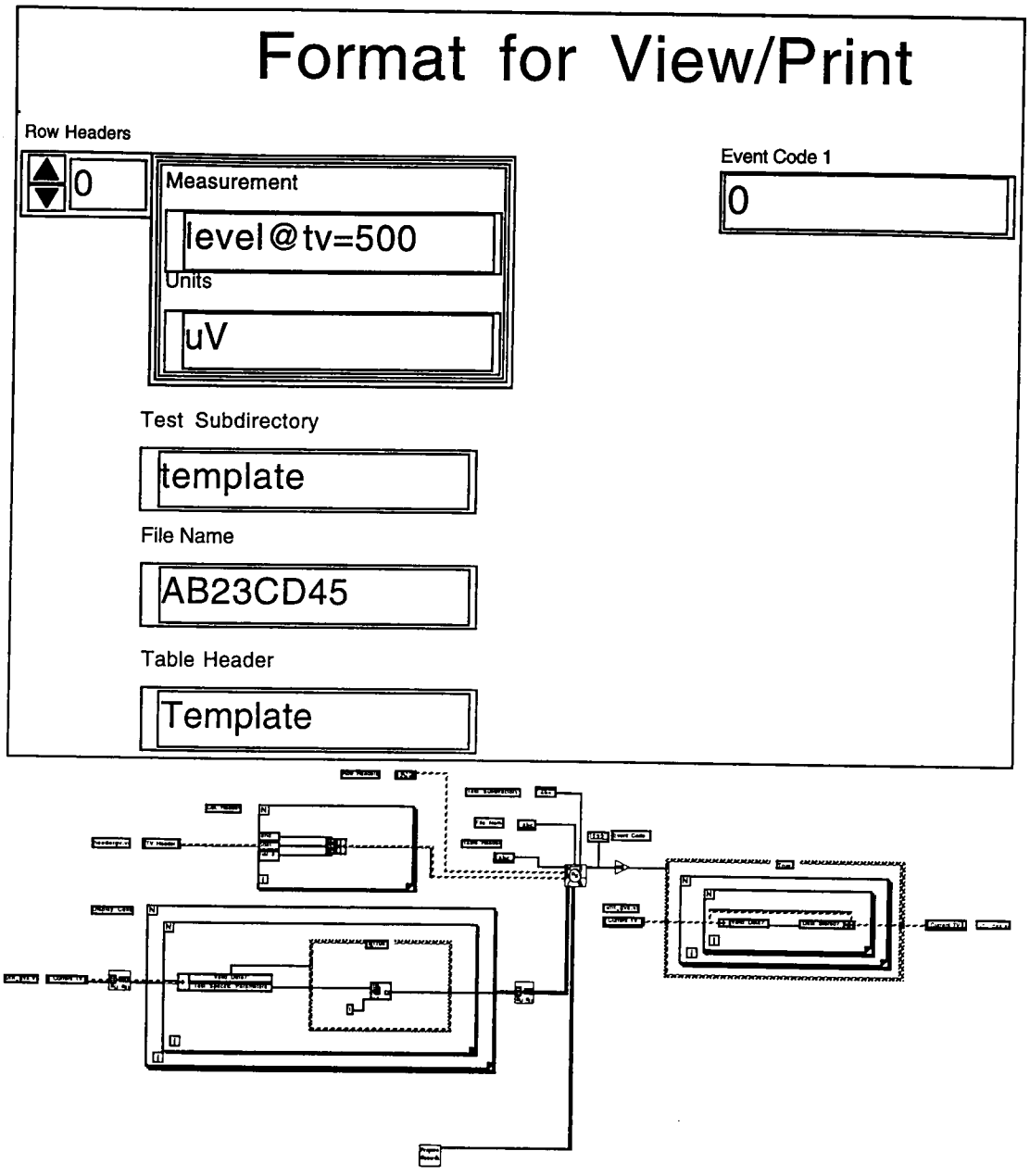


Fig. 5.11 Template data format VI.

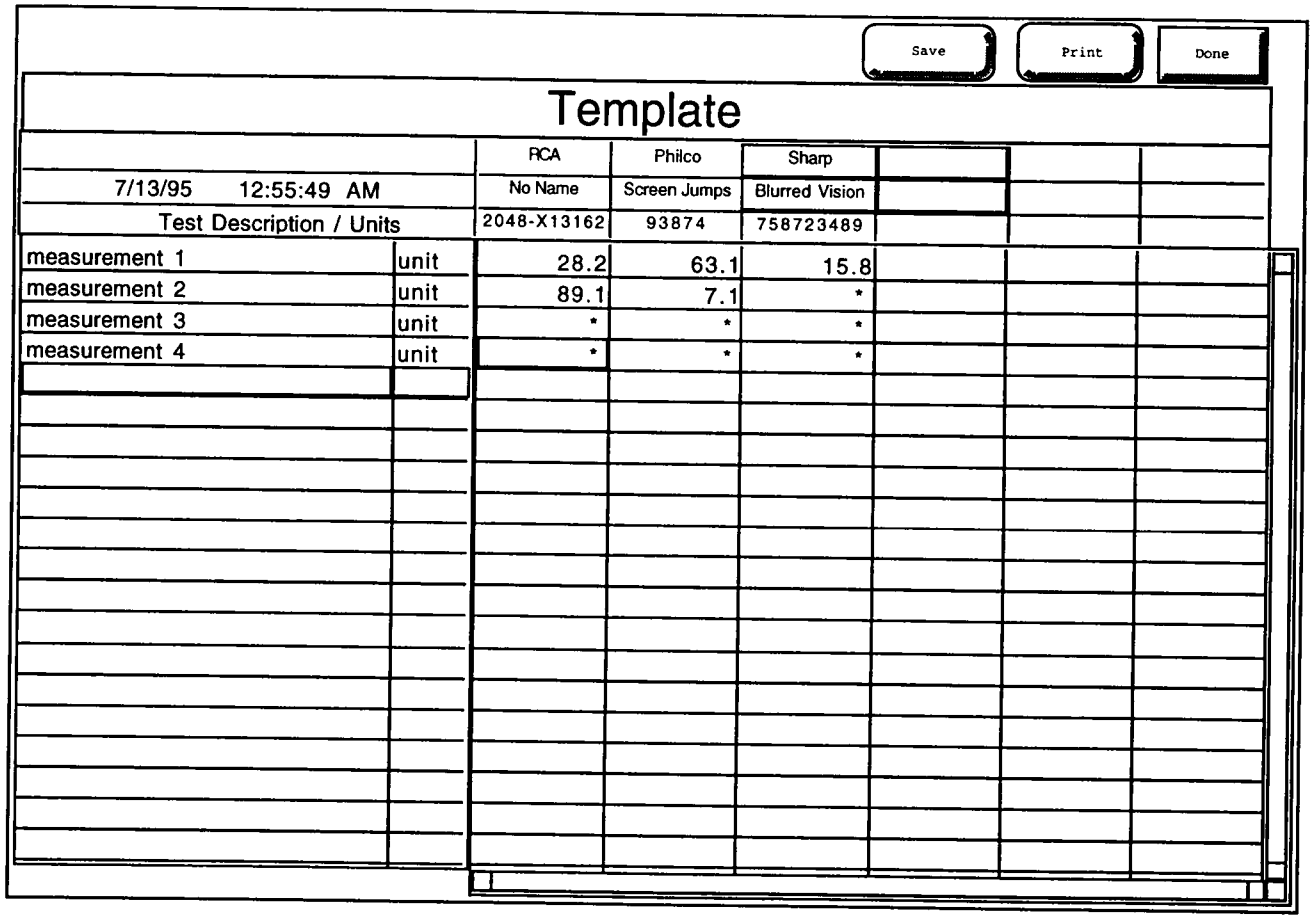


Fig. 5.12 Template measurement panel.

subdirectory specified by *Test Subdirectory*.

The data written to disk must also be in a specified format. This is done in **records.vi** which is shown in Figure 5.13. This VI outputs a 2D array of strings which is written to disk. The conventions used in the .csv format are as follows. All booleans are converted to "T" or "F" (including the quotation marks). All numeric values are written as strings but without quotation marks. All other information is written as is surrounded by quotation marks. The quotation marks are used because this data will be accessed later by some sort of database software. Then, most packages of that kind will import information without quotation marks as numeric values and all other data as non-numerics (e.g., dates, times, textual information, etc.).

Measurement values are treated somewhat differently. If a particular measurement has not yet been taken (i.e., an asterisk exists in the display for that cell) then it is written to disk as a zero value. Then, each record has the *Valid Data?* flag as a field too. Thus, if that field is set, then the measurement was taken and a zero is truly a zero value. If the field is not set, then the zero is just a place holder for the database software. Then, the database can be programmed to key on that flag in order to determine which zeros are really zeros and which are place holders.

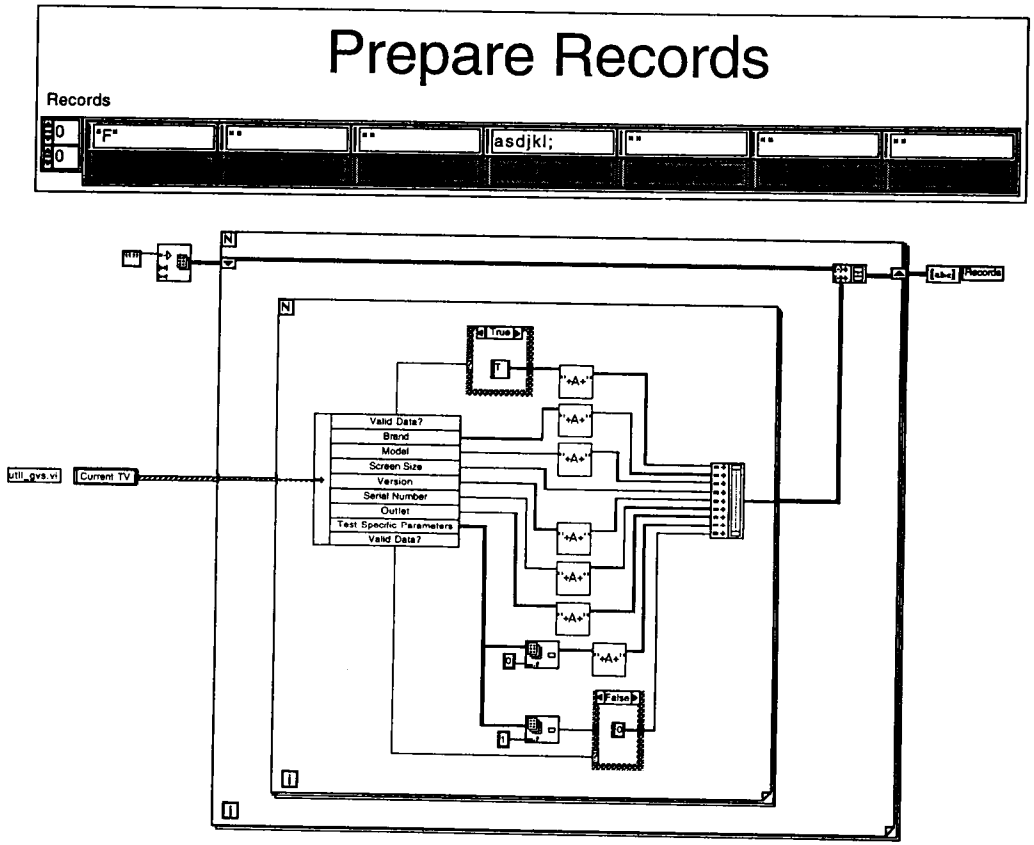


Fig. 5.13 Template records format VI.

The **mode_ctl.vi** VI will be discussed next. It is shown in Figure 5.14. It was written to control *Current TV* array and the *Measurement* menu ring on the front panel. This VI is generic also. The only test specific parameter is the *Set of Meas.* which are displayed in the menu ring.

This VI first checks the *Test Complete?* boolean which is controlled by **rec_ops.vi** (discussed above). If the test is complete, then it locks the *Test Mode* button down to true because at that point the auto sequencing is disengaged. In addition, it disables and grays out that button so that user cannot release it. Then, the user has several options (discussed below).

If the test is not complete, then the *Test Mode* controls the outputs of this VI. If the test is in Auto Mode, then the menu and the *Current TV* array are controlled programmatically. The ring is used to display the current measurement and the array is set to the current DUT. The user is not able to change either because the program has control of them.

If the *Test Mode* is set for Manual testing, then the ring displays all the possible measurements for the test and the user is able to select the one he wants. Also, he is able to select the DUT by using the up and down arrows on the upper (x) index. The lower index (y) cannot be changed. It is programmatically controlled based on

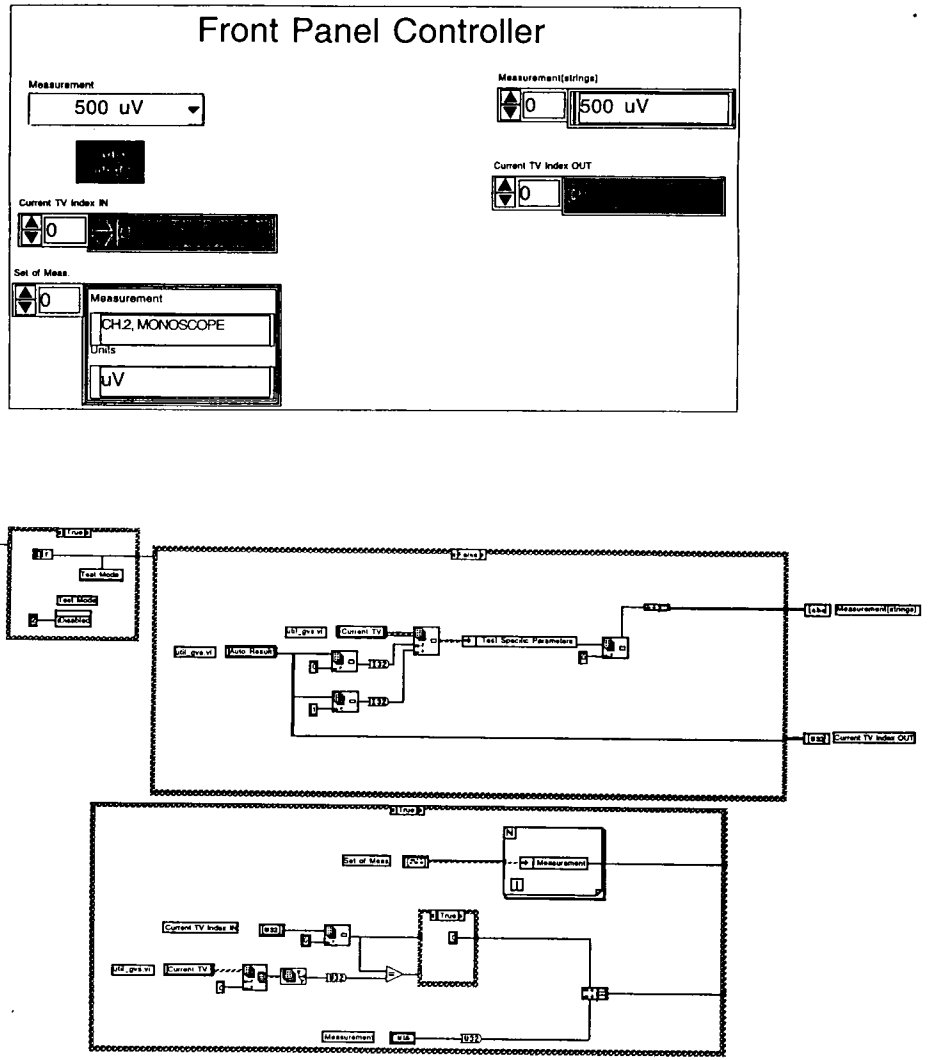


Fig. 5.14 Template ring control VI.

the setting of the menu ring. Thus, the arrows are covered.

The **edge_det.vi** (shown in Figure 5.15) was written to detect a rising edge on the *Test Complete?* variable. This was necessary to keep the test complete dialog box from continuously displaying which would be the case if the box is triggered on the level (0 or 1). Therefore, this edge detector circuit was used rather than a level detector. It simply sends the *Complete Ind.* value to the **exitchek.vi** (discussed below).

The **exitchek.vi** VI (shown in Figure 5.16) was written to check the possible conditions under which the test could exit. These are: a hardware error during the BIT Check Stage, Reset Stage, or Preset Stage (indicated by *Preset Forward*); a hardware error during testing; a pressing of the *Quit* button; and a pressing of the *Logout* button. This VI also accepts the *Complete Ind.* from the **edge_det.vi** VI. In addition, it has the same inputs as the **vw_formt.vi** because this VI calls that VI when the test is exited.

The BD shows a case structure which is controlled by the various *Exit Conditions.* Case 0 is the default case when none of the Exit Conditions is asserted. Case 1 is executed when the test is complete. A dialog box is displayed listing the options now that the test is complete. A false value is output in the *End Test* boolean. Therefore, the while-loop is not halted. Case 2 is executed if *Preset*

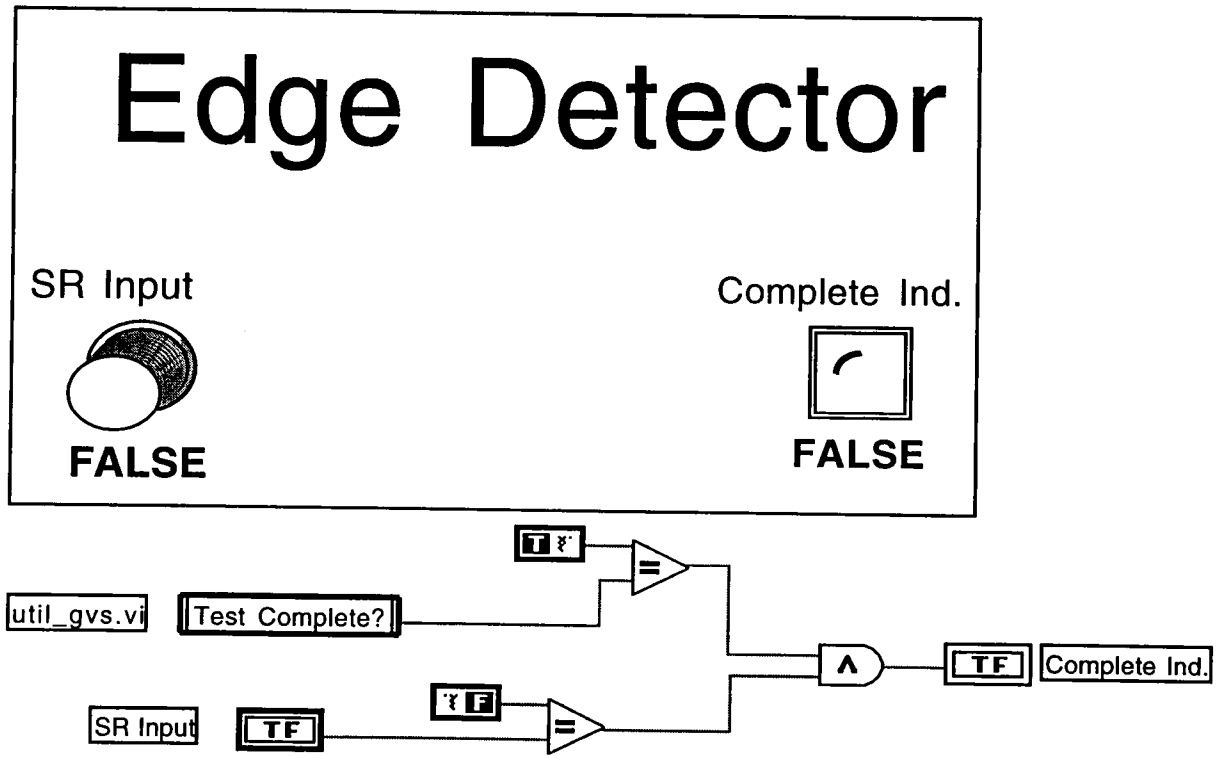
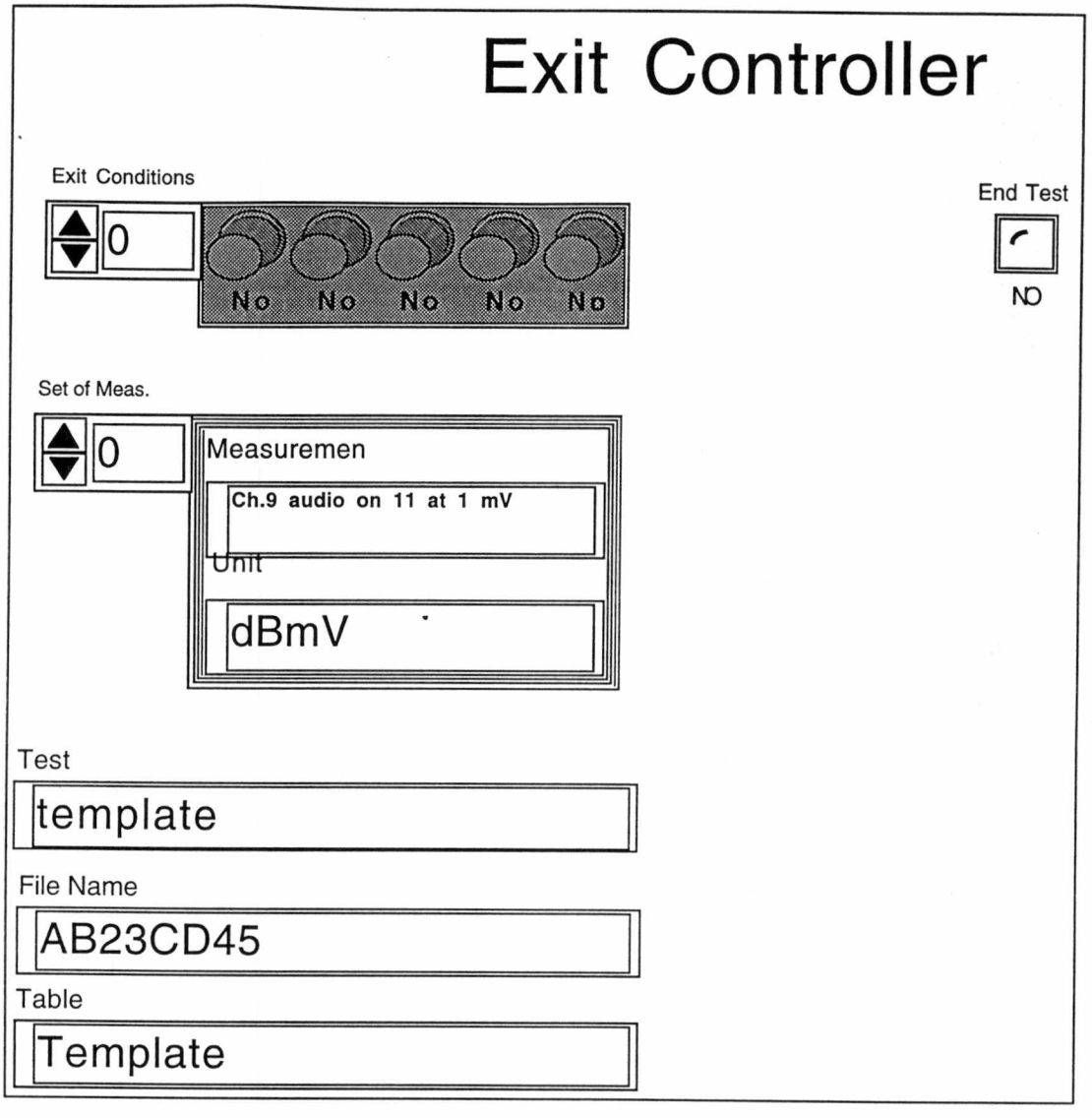
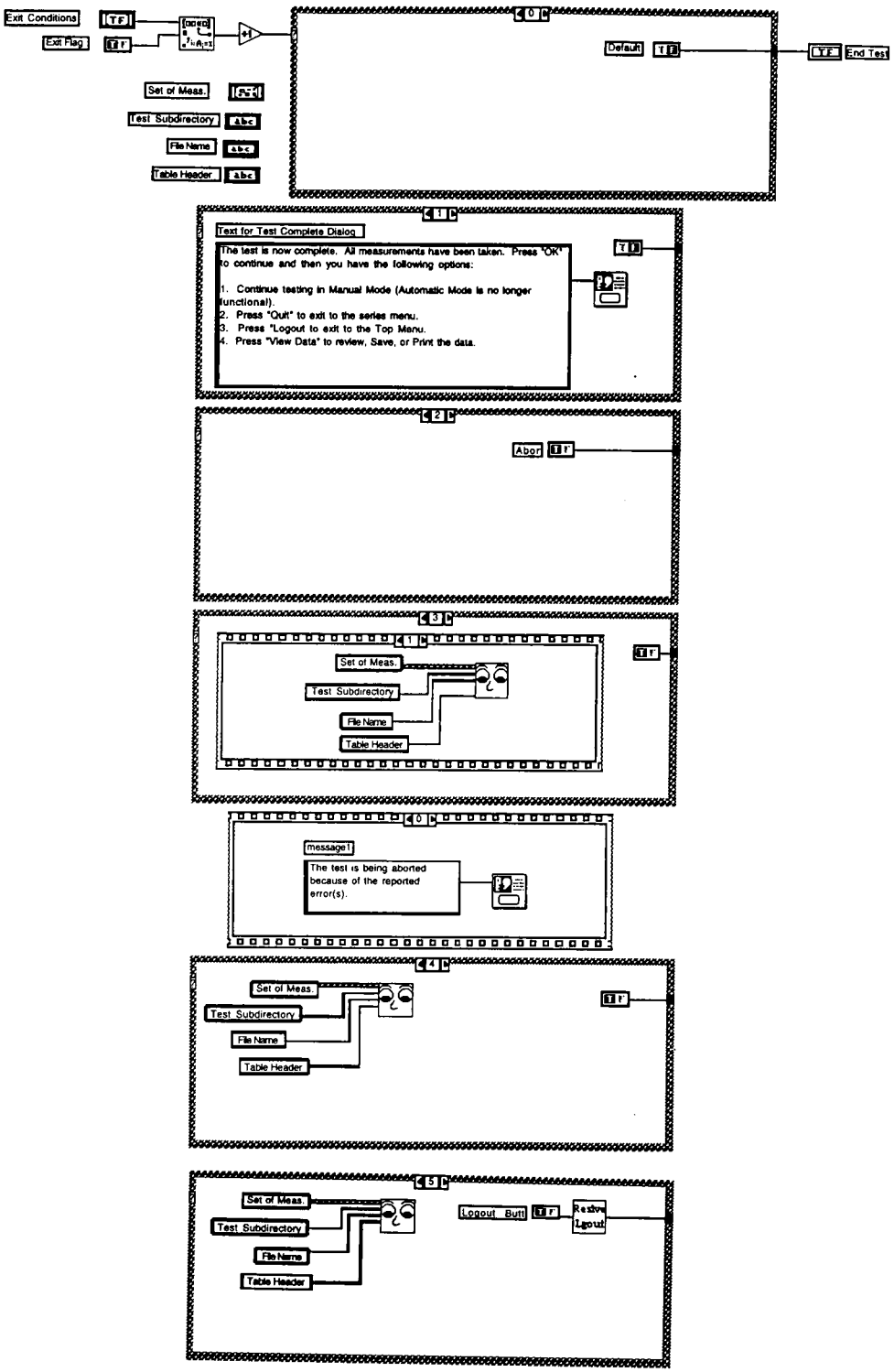


Fig. 5.15 Template edge detect VI.



(a) Front Panel

Fig. 5.16 Template exit check VI.



(b) Block diagram

Fig. 5.16 (continued)

Forward is false indicating a hardware error in the BIT Check, Reset, or Preset stages. Therefore, a true is output for *End Test* and so the while-loop is halted. Case 3 is executed if a hardware error occurs during testing. That is, the driver controller outputs a positive event code. In this case, the user is prompted that the test is being aborted for the reasons given by the driver controller and the **vw_formt.vi** VI is called. Cases 4 and 5 are executed when *Quit* and *Logout* are pressed, respectively. They both call the **vw_formt.vi** VI but Case 5 also has the Resolve Logout subVI which signals to all menus below the Top Menu that *Logout* was pressed.

Finally, the **hpgdrivr.vi** subVI is left to discuss. This VI was written to control the hardware needed for the Airplane Flutter test. It is designed similar to the driver controller for the Sensitivity test. It is shown in Figure 5.17.

The TSI8150 and the Ghost Simulator drivers are only called if their parameters are different from one iteration of the while loop to the next. That is, if the signal level required at the DUT is different, then the TSI8150 driver is called to set the required attenuation. Also, if the *Frequency* is different from one iteration to the next, then the Ghost Simulator driver is called to set the required Doppler frequency. As before, the event codes from each are checked to insure that no errors occurred during their execution.

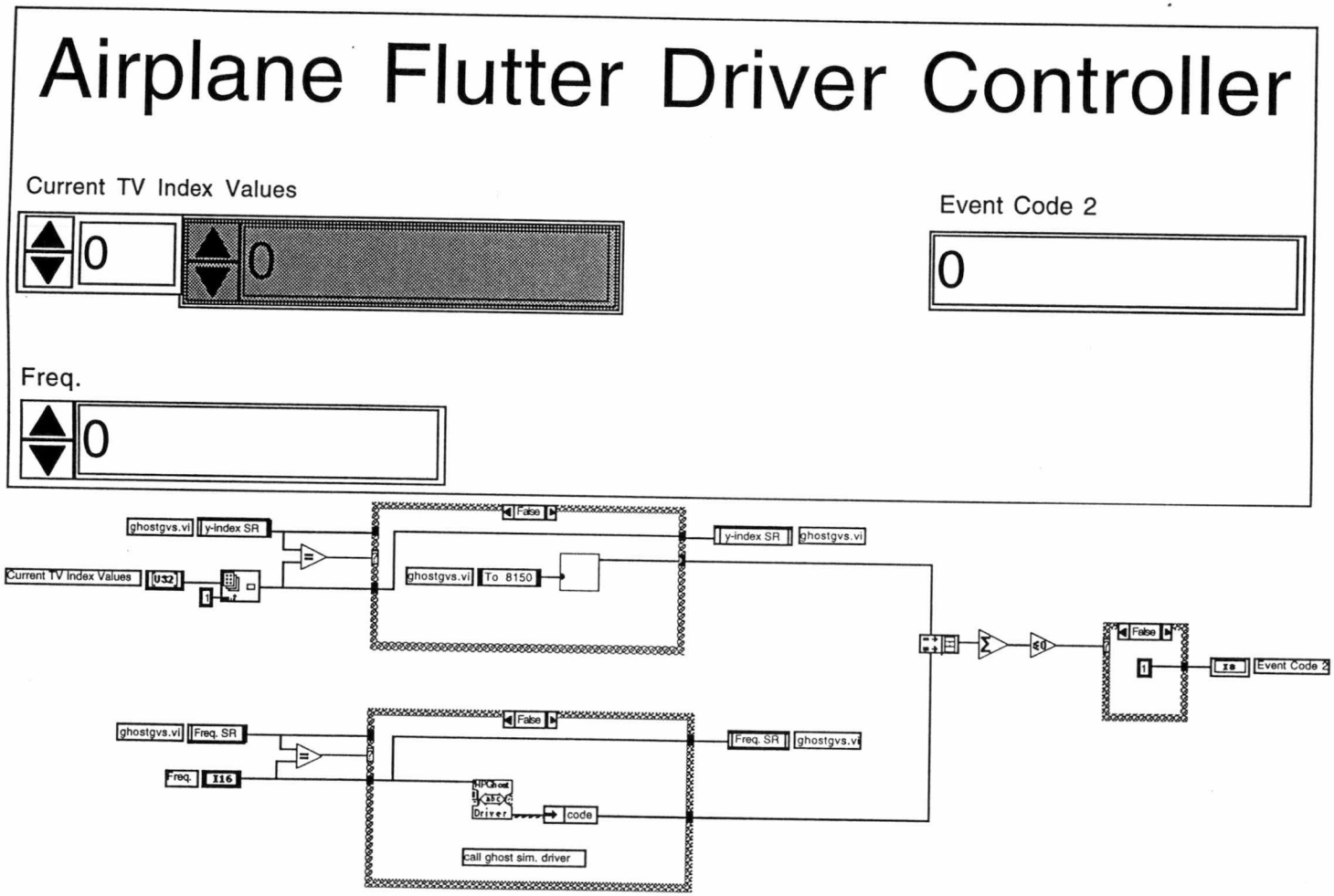


Fig. 5.17 Airplane Flutter driver controller VI.

A comparison of this architecture with that of RF Sensitivity brings to light the improvements in the code. Using this architecture, the productivity in code development was increased by a factor of four (4). Tests developed in the manner employed in developing the Sensitivity test took approximately one month to complete. Tests developed using this template approach were completed within one week. In addition, all of the tests have the same basic architecture and layout. Therefore, they are much easier to maintain and update. So, the template approach to test development was much more efficient.

Chapter 6

HP 11759D Ghost Simulator Driver

6.1 - Introduction

This chapter is dedicated to discussing the design of the instrument driver for the HP 11795D Dynamic Ghost Simulator. The Simulator is designed to simulate the time dispersion, Doppler shifts, and multipath fading/ghosting conditions seen in the television transmission channel environment. It is designed to test analog and digital television systems with occupied channel bandwidths up to 6 MHz under multipath conditions.

The device has one channel with six independent propagation paths. Therefore, a simulation can have one main signal and five other signals that interfere with it. All characteristics of each path can be independently controlled. These are delay, phase, Doppler shift, and attenuation. Also, for complex simulations, each path can be Rayleigh faded.

There are three main types of simulations that the Simulator will support. These are the Simulation test, the Travel test, and the Dynamic test. In Simulation test, the conditions are static. That is, the transmitter and receiver are assumed to be stationary with respect to each other and all paths are propagated based on a static

environment. In Travel Test, two options are available in an environment where there is a reflector of some kind (e.g., an airplane) between the transmitter and the receiver. Then, the Travel test allows for a moving reflector and a stationary receiver; or there can be a stationary reflector and a moving receiver. Finally, in Dynamic testing, prepared or supplied data files are used to produce user specified simulations. These data files specify all characteristics of each signal at certain instants in time.

The Simulator is controlled via a parallel connection to an HP Vectra VL/2 486/66 PC. The PC can also be controlled via GPIB. This was the setup for this project. Thus, the main system controller controls the PC via a GPIB connection and then the PC sends commands to the Simulator via a Centronics parallel connection.

The Simulator software is loaded on the PC and has a menu-driven interface running under DOS. From the PC, all of the features of the Simulator are accessible. Through the GPIB connection, only the Simulation Test are fully accessible. The other two tests only have one or two commands which can be executed remotely. Thus, the driver design presented here is basically a complete instrument driver even though it is dominated by the commands for the Simulation test.

6.2 - HP 11759D Dynamic Ghost Simulator Instrument Driver

In designing this driver, the procedure outlined in Section 3.4 was followed as closely as possible. The first couple of steps involved studying the instrument's operation and becoming familiar with the manual. Also, manual testing was done with the Simulator in order to understand its implementation in real testing.

The next step was to decide which features were best suited for programmatic use. These were decided to be the Simulation test, system reset, version queries, error checking, and dynamic loading of data files. Because of the limited command set for the Travel test and the Dynamic test they could not be implemented programmatically except listing, storing, and loading data files.

Then, the applicable LabVIEW template VIs had to be determined. Almost all of the VIs described in Table 3.3 were applicable with the exception of the Register-based template which is for use in systems where the instruments are controlled via a VXI bus.

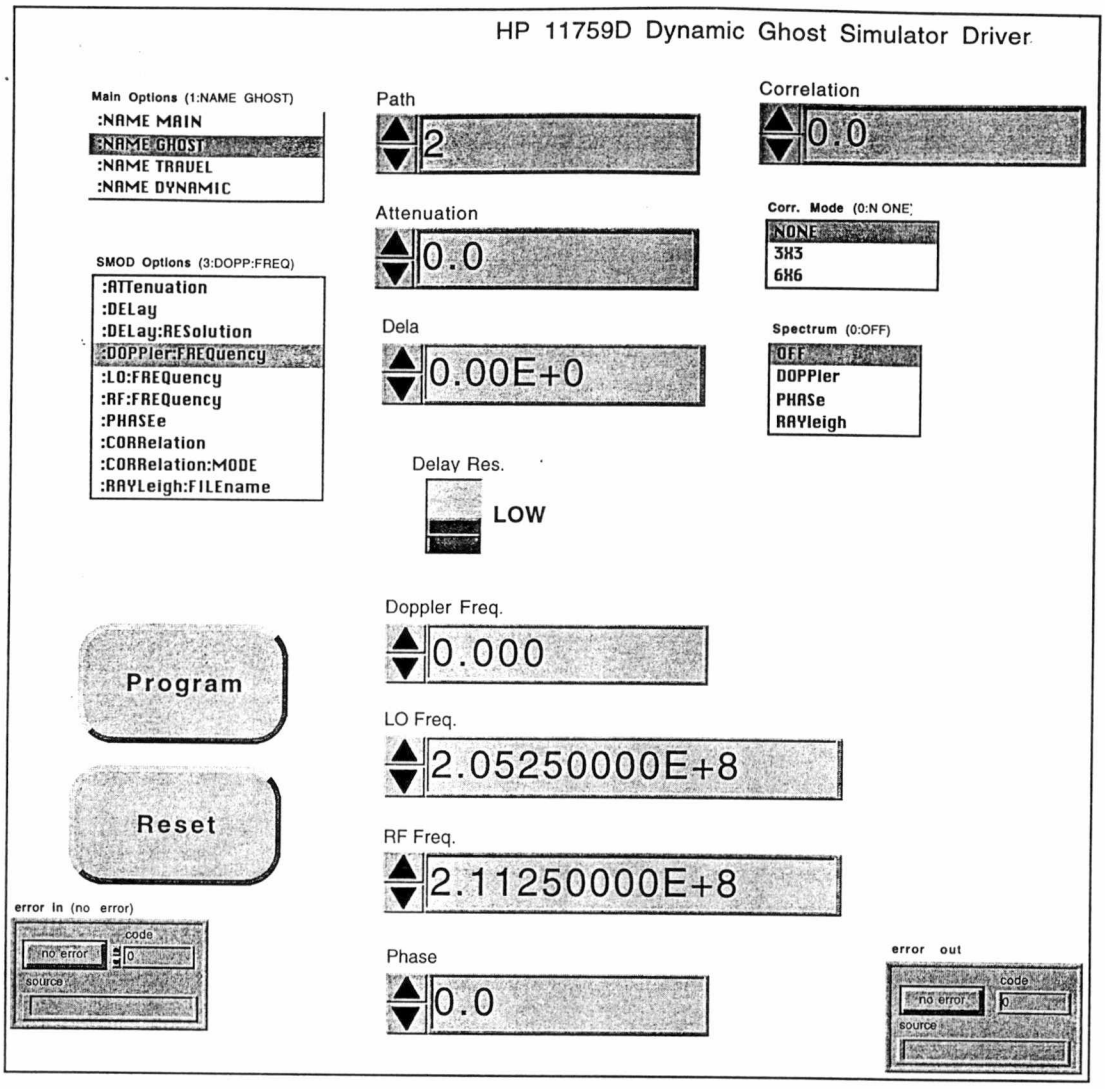
Next, the VI hierarchy had to be determined. This consists of just a few VIs which are discussed below. So, all of the simulator tests were combined into one application, again because of the limited programmatic control over the Travel and Dynamic tests.

Having completed this theoretical development, the foundation

was built for coding the driver. The top level VI is called **hpgdrivr.vi** and is shown in Figure 6.1 [16, p.4-15]. As can be seen from the BD, all of these driver VIs pass control in a token ring fashion where the *error in-error out* pair and the *instr handle in-instr handle out* pair serves as the tokens. That is, each VI (except the first which only transmits the handle out and the last one which only receives the handle in) must receive and send out these parameters. Also, each program has a constant called *VI Name* which it uses to reference errors in the error cluster. This is simply the name of the VI so that as the error cluster is passed from VI to VI, if an error occurs then the name of the VI where it occurred is included in the *source* indicator. Finally, it should be noted that all GPIB commands to the Simulator must end in a "line feed" and all responses from the Simulator are terminated with a line feed.

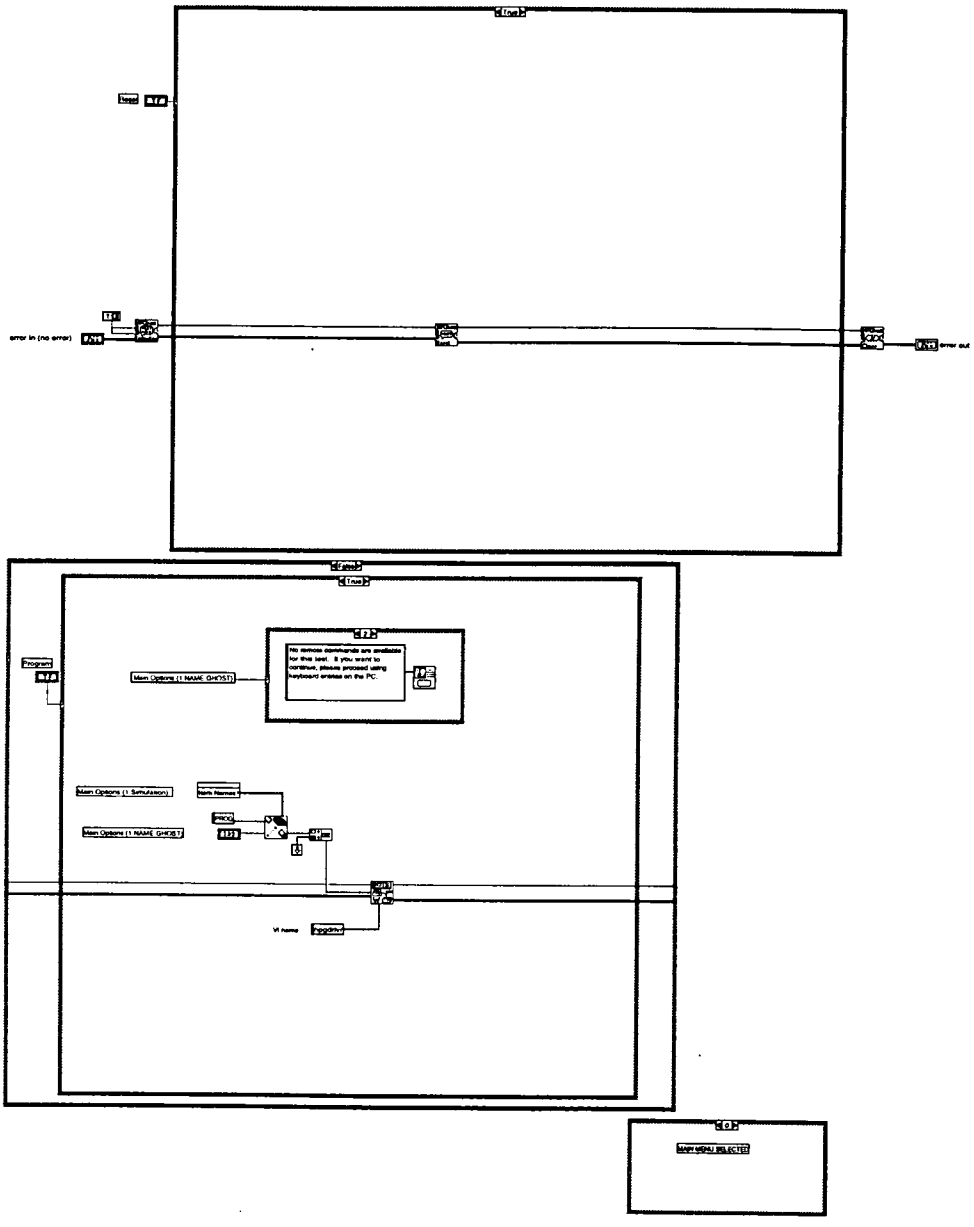
Starting from the left side of the BD, the first VI to be discussed is the **hpg_init.vi** VI (shown in Figure 6.2) [16, p.4-3]. This VI passes the addressing information in the instrument descriptor to the the Open VI and returns a unique reference to the instrument I/O session in *instr handle out*. This VI must be run first before any other driver VIs in order to obtain this instrument ID.

The user also has the two other options with this VI. He may send the instrument an ID query in order to retrieve some basic



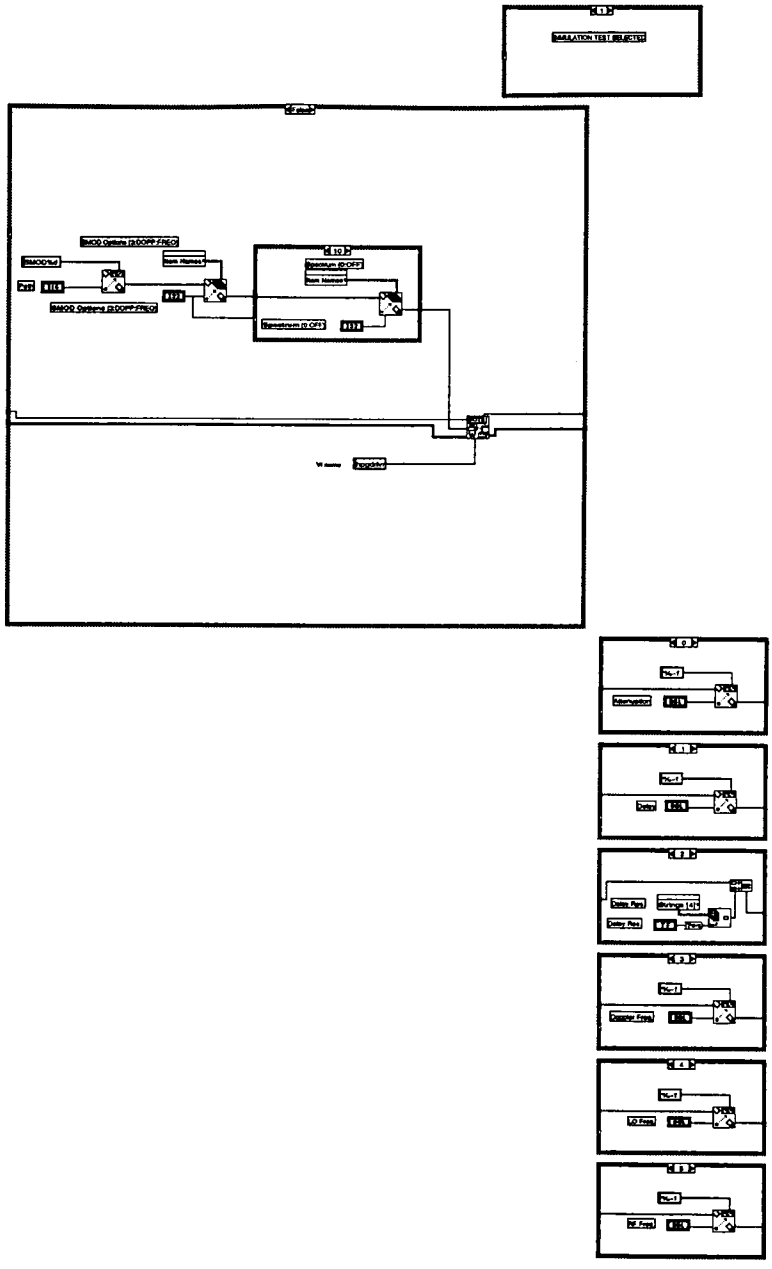
(a) Front panel.

Fig. 6.1 Ghost Simulator driver.



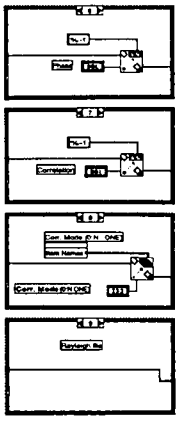
(b) Block diagram.

Fig. 6.1 (continued)



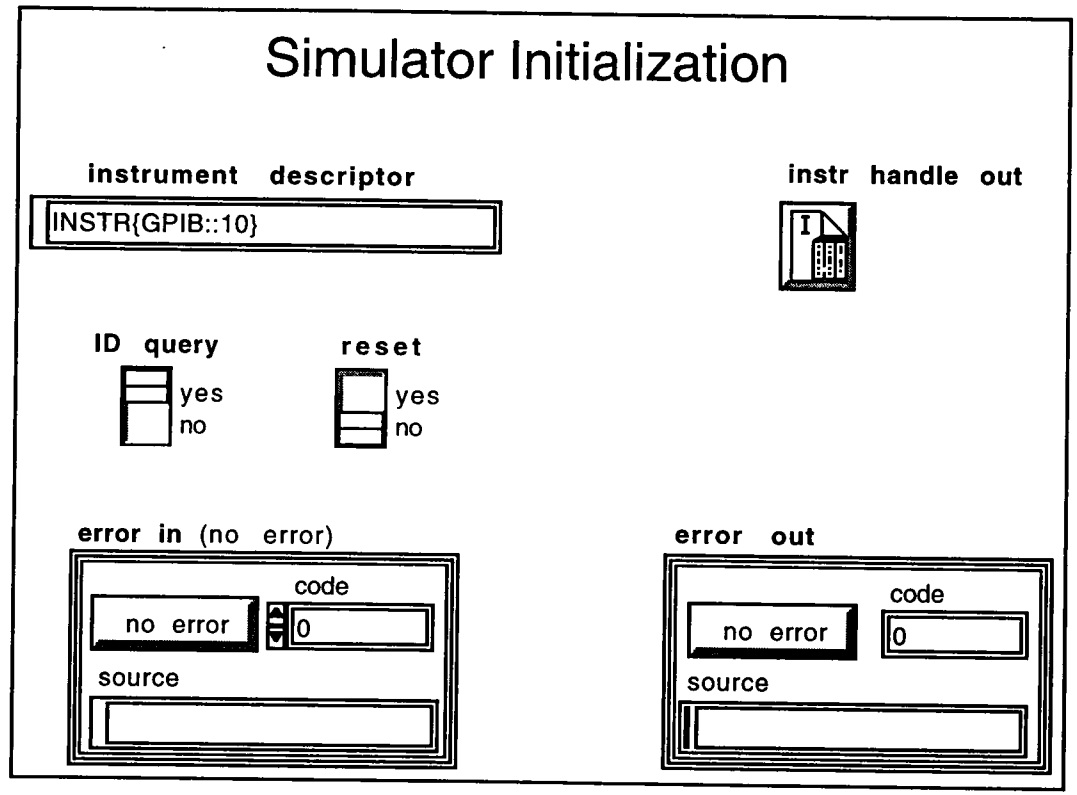
(b) (continued)

Fig. 6.1 (continued)



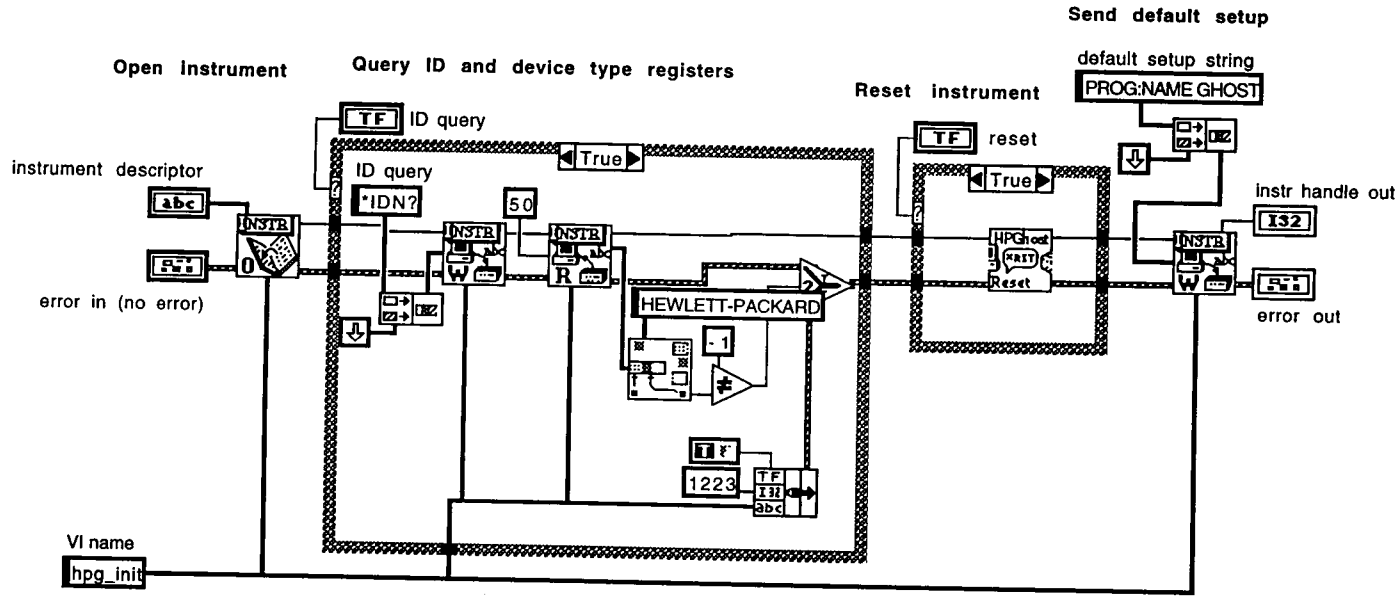
(b) (continued)

Fig. 6.1 (continued)



(a) Front Panel.

Fig. 6.2 Simulator initialization VI.



(b) Block diagram.

Fig. 6.2 (continued)

information about the instrument such as name, serial number, firmware version. Also, he has the option of resetting the instrument using the reset VI (discussed below).

. The last write command is a default setup string which is sent to the Simulator and commands it to select Simulation test from the main menu. It should be noted that the commands GHOST and SIMULATION can be used interchangeably.

The reset VI, **hpgreset.vi**, is actuated if the *Reset* button is pressed. This VI is shown in Figure 6.3 [16, p.4-10]. This is a fairly simple VI in that the standard GPIB reset command, "*RST", is sent to the Simulator which is identified by the *instr handle in*.

If reset is not asserted, then there is another case structure which is controlled by the *Program* button. If *Program* is asserted, then the *Main Options* are valid. This is analogous to being at the Main Menu screen on the PC. The user can select one of the 3 tests or he can select :NAME MAIN in order to call the Main Menu. If he chooses, the Travel or the Dynamic test, he will get a dialog box asking him to proceed from the PC because neither of these can be programmatically run.

If *Program* is not asserted and the :NAME GHOST is selected from the *Main Options*, then the Simulation test is executed. This is shown where both case structures are false. The user can first

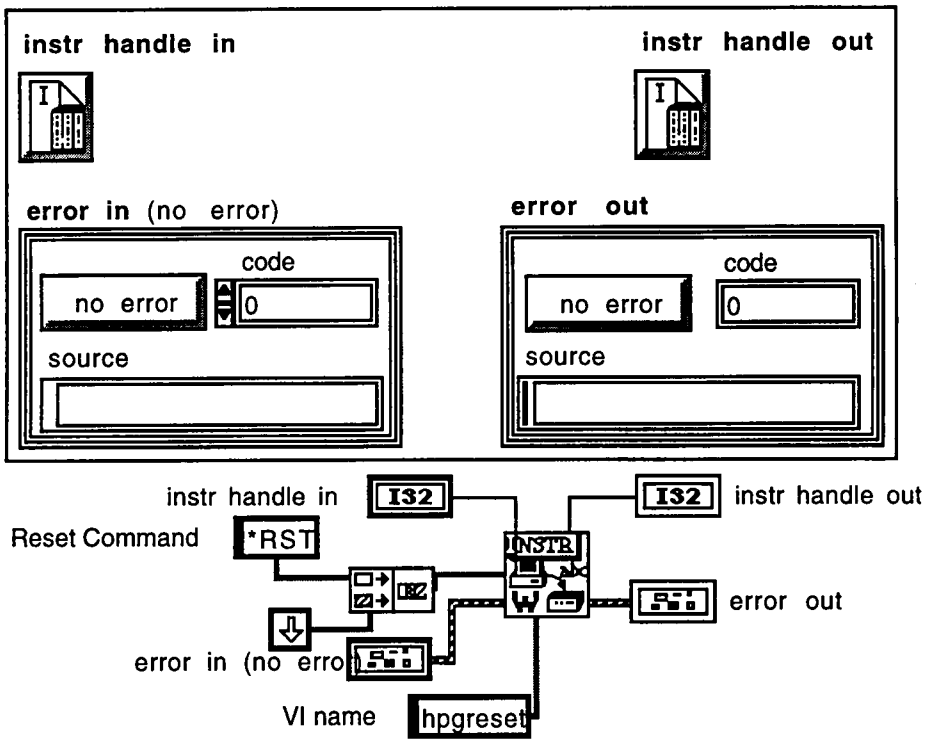


Fig. 6.3 Simulator reset VI.

select the *Path* number for which he wishes to change characteristics. Then, he may select the characteristics from the from the various *SMOD Options*. Finally, he must set the value of the particular characteristic. The program sends the appropriate command to actuate that command which is shown in the *Command* indicator.

Finally, each instrument has to close communications with the instrument. For the Simulator, this is done with the **hpgclose.vi** VI shown in Figure 6.4 [16, p.4-6]. This too is a simple VI. It calls on the **Instr Close.vi** to cease communications and deallocate resources assigned to the Simulator.

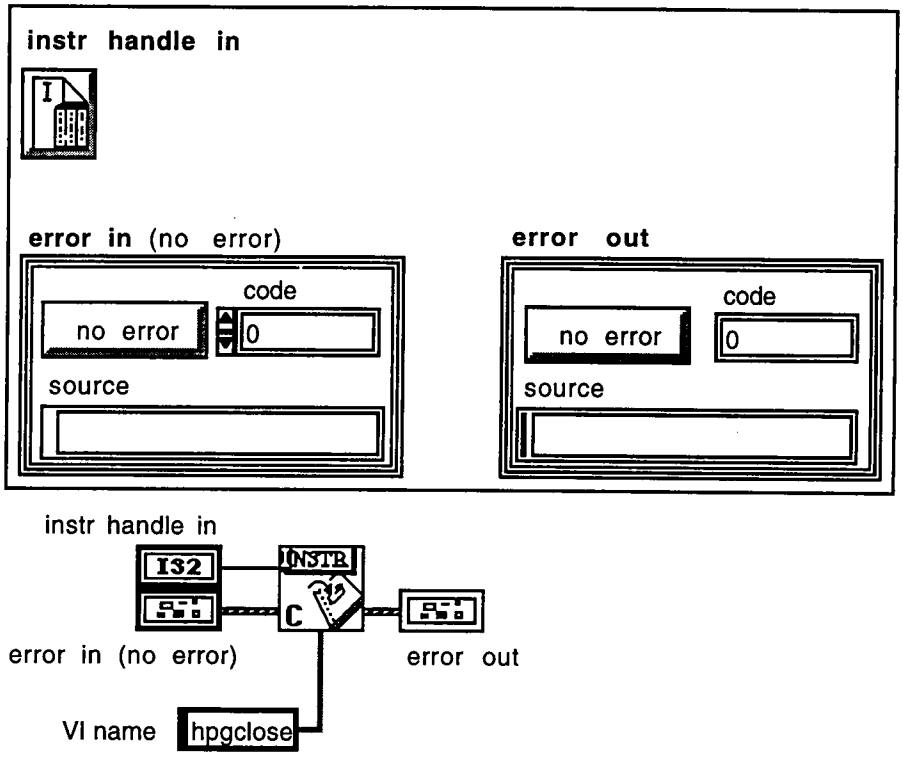


Fig. 6.4 Simulator close communication VI.

Chapter 7

Discussion

7.1 - Conclusions

The author gained much experience from this project and was able to draw several conclusions. Most of these were positive with some exceptions. This is to be expected. It would be naive to state that all of the findings in this project were positive. Even the best of projects have some negative points.

The sum total of all of the benefits was most obvious in automating the RF Sensitivity Test. From a productivity standpoint, the tester was able to double his productivity. This was the minimum value which resulted from uncontrollable variables causing delays. Otherwise, the productivity increased by more than a factor of 2. These could be computer problems, hardware problems, or even network problems between the X-terminal and the main controller. Most of these problems would not manifest themselves in a manual procedure. Thus, a lower limit on productivity is given. This productivity increase leads to increased efficiency for both the tester and his department.

There was also increased productivity in the area of data logging. Previously, the data was logged manually then transferred

to an electronic medium manually. With the development of the view-print-save utility, data logging was automated too. Consequently, it will make report generation much easier and faster. In addition, there is less chance that the data will be incorrect since they are electronically logged rather than manually.

Although the Airplane Flutter test did not yield as high an increase in productivity, it did produce a significant increase of around 1.5. For this test, the automated process simply was not much faster than the manual one. However, most of the savings in time were realized because of the automatic setup and data logging. Thus, the procedure itself was not much faster when performed automatically. However, considering this test in the context of the entire ATF, it certainly achieved its purpose.

The author was formally introduced to the CSE process as a design methodology. Without a doubt this was an excellent way to develop software. The best feature of CSE is that it requires such rigor in developing code. Then, this rigor translates into better code with much fewer run-time errors. Consequently, it increased the productivity of the author and of the entire ATF team in terms of how fast software was developed.

The LabVIEW package merits a few comments. This was a wonderful environment in which to develop software. It lends itself

very nicely to hierarchical designs such as this one. Also, its graphical interface allowed the software to be extremely user friendly. It would have been much more difficult to have completed this project in any other environment, especially a text-based one.

Overall this was an excellent and an enjoyable project. It was a team effort yet there was plenty of room for individual research and achievement. It was truly an industrial application of principles learned in education. In the end the benefits of the research could be observed upon completion of the project. These were not simply theoretical discoveries but genuine increases in efficiency and productivity.

7.2 - Proposals for Continuation of This Work

The discussion presented here was, of course, the focus of the author's research. There are many ways in which this work could be continued. In addition, there are many new projects which could be developed in areas not even touched by the current effort.

One way in which the work could be continued is by defining a more rigorous verification process. According to Mills, in order for a program to pass the verification stage, the verifier should be able to take the final code and collapse it down to the BB. This concept

needs to more fully developed. Although, the current way in which verification is performed is acceptable and does help determine failure points, it could be improved and extended to be even better.

One area that was not touched by this project is the consideration of PAL system televisions. This effort has concentrated entirely on developing algorithms for performing tests on NTSC receivers. There are millions of televisions that are configured for the various types of PAL system signals. Although the test procedure on these sets may be similar to the NTSC ones, the majority of the hardware is quite different. Thus, a new set of instrument drivers and peripheral VIs would have to be written.

Finally, the most extensive off-shoot project that could be developed is in the area of Image Processing and Vision Systems. Currently, the tester makes subjective judgments as to whether a television has noise interference "just present" or when he can "barely" read the numbers on a Monoscope pattern. Ideally, such video images should be captured by a camera, the image brought into a computer, and then processed to determine that threshold.

These are just a few suggestions for continuing this work and for new projects. There are other projects as well that have not been mentioned. It is the hope of the author that each of these will be explored by future designers.

LIST OF REFERENCES

LIST OF REFERENCES

[1] *ATF Project System Requirements*, "RF Sensitivity Test Procedure", CISP Group, University of Tennessee, Knoxville, May 1995 Version.

[2] *ATF Project System Requirements*, "Airplane Flutter Test Procedure", CISP Group, University of Tennessee, Knoxville, May 1995 Version.

[3] *ATF Project System Requirements*, "Equipment Interconnectivity Plan", CISP Group, University of Tennessee, Knoxville, Version 5.0.

[4] K. Blair Benson and revised by Jerry Whitaker, *Television Engineering Handbook*, McGraw-Hill, Inc., Revised Edition.

[5] Alvin A. Liff, *Color and Black and White Television Theory and Servicing*, Prentice-Hall, Inc., Englewood cliffs, NJ, 1979.

[6] Arvind M. Dhake, *Television Engineer*, McGraw-Hill Book Company, New Delhi, 1980.

[7] Martin Giles, "Evaluating T.V. Receiver Performance Under Conditions of Mutli-Path Signal Propagation.

[8] *HP 11759D (Including option 002) Dynamic Ghost Simulator Operating and Service Manual*, Hewlett-Packard Company, Santa Clara, CA, Part No. 11759-90017.

[9] Dr. Carmen Trammell, "CSE Workshop: Part 1", June 1994.

[10] Richard H. Cobb and Harlan D. Mills, "Engineering Software under Statistical Quality Control", *IEEE Software*, Vol. 7 No. 6, p. 44-54, November 1990.

[11] P. A. Hausler, R. C. Linger, and C. J. Trammell, "Adopting Cleanroom Software Engineering with a Phased Approach", *IBM Systems Journal*, Vol. 33 No. 1, 1994.

[12] Richard C. Linger, Harlan D. Mills, and Alan R. Hevner, *Information Systems Analysis and Design*, Academic Press Inc., Orlando, FL, 1986, Ch. 1-3.

[13] ATF Team design meeting, CISP Group, University of Tennessee, Knoxville, July 1994.

[14] Howard Anton, *Calculus with Analytic Geometry*, John Wiley & Sons, New York, NY, 2nd Edition, 1984, p. 256.

[15] ATF Team design meeting, CISP Group, University of Tennessee, Knoxville, May 1995.

[16] *LabVIEW Instrument I/O VI Reference Manual*, National Instruments, September 1994 Edition.

[17] Eric Wrushen and Keeratpal Singh, Personal Correspondence, *tslinit.vi*, LabVIEW VI, November 1994.

[18] Gary Ragsdale, Personal Correspondence, *SVSINIT.VI*, LabVIEW VI, September 1994.

[19] Gary Ragsdale and Sameer Shah, Personal Correspondence, *warm_c.vi*, LabVIEW VI, July 1994.

[20] Gary Ragsdale, Personal Correspondence, *LOGMEASR.VI*, LabVIEW VI, July 1994.

[21] Gary Ragsdale, Personal Correspondence, *rsvlogot.vi*, LabVIEW VI, October 1994.

[22] Gary Ragsdale, Personal Correspondence, *SVSXCON.VI*, LabVIEW VI, September 1994.

[23] Eric Wrushen and Keeratpal Singh, Personal Correspondence, *tsi_8150.vi*, LabVIEW VI, November 1994.

[24] Gary Ragsdale, Personal Correspondence, *vwprtsav.vi*, LabVIEW VI, July 1995.

VITA

Sameer Dinesh Shah was born on 14 September 1970, in Gholvad in the state of Maharastra, India. He came to the United States in July of 1976. He went to Wheaton Woods Elementary School in Wheaton, Maryland. He moved to Clarksville, Tennessee, in October of 1982 and currently resides there. He graduated from Clarksville High School as Salutatorian of his class in June 1988 and entered Tennessee Tech University in the fall. He graduated from Tennessee Tech in December of 1992.

Mr. Shah started graduate school the following January at the University of Tennessee in Knoxville and received his Master of Science degree in Electrical Engineering in August of 1995.

Mr. Shah owns a small electronics consulting company known as *The DAASCs Group*. He started this company in July of 1994. The company entered the import-export market in February of 1995.