Development of the Second-Generation IMTS (Intelligent Monitoring and Trending System) and WOT (Wizard of Tech) Expert System for Rotating Machinery

by

E.C. Pawtowski

Thesis submitted to the faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

Master of Science

in

Mechanical Engineering.

APPROVED:

Dr. R. Gordon Kirk

Dr. D. J. Nelson

Dr. L.D. Mitchell

April 29, 1996

Blacksburg, Virginia

Keywords: Expert System, Turbomachinery, Data Acquisition, Data Trending

Development of the Second-Generation IMTS (Intelligent Monitoring and Trending System) and WOT (Wizard of Tech) Expert System for Rotating Machinery

by

Eric Pawtowski

(Abstract)

IMTS and WOT form a PC-based hardware and software system designed to continuously monitor large numbers of rotating machinery, evaluate each machine's condition through a series of user-definable standards, and alert operators to potential problems. This system requires a rack of data acquisition equipment located near the machines being monitored and a PC that can be located remotely. This system has been tested under actual plant conditions at the Virginia Tech Power Plant. The software operates under Windows 3.1, and allows data to be acquired and evaluated simultaneously. This thesis discusses the development of this system over earlier versions and the installation procedures and first runs at the Power Plant. It discusses in detail the operation of some of the main programs that comprise the Intelligent Trending and Expert System.

Acknowledgements

The IMTS and WOT systems have been made possible by joint industry/Virginia Center for Innovative Technology grants CAE-88-011, CAE-88-011-01, and CAE-92-010, with additional support from Bently-Nevada, Dresser-Rand, Ingersoll-Dresser Pump Company, Elliott Company, and Virginia Power.

In addition, the Virginia Tech Physical Plant allowed use of their plant for testing and development of this system. I am especially grateful for Charles Page, chief of Instrumentation at the physical plant, and his assistant Donald Doss. Their guidance and assistance during the system's installation at the plant and their input into the system's user interface is appreciated.

I would also like to thank Roy Mondy, Rotating Equipment Specialist at Virginia Power, for his input early in the development of this system, James R. Hoglund, creator of the original off-line WOT expert system, Marcello Typrin, co-developer of the first-generation IMTS system, and Joseph Cochrane, who programmed the early versions of the second-generation IMTS display module.

I would especially like to thank the chairman of my committee, Dr. R. Gordon Kirk, for his assistance with this project and thesis and for providing the full resources of the Virginia Tech Rotor Dynamics Laboratory. Thanks also go to fellow Rotor Lab graduate student Erik Swanson for his technical support in the lab.

Further thanks go to my parents, Frances and Cloud, and my brother, Daniel, for making this project possible. Special thanks go to my fiance', Yvonne Bennett, and her parents, Warren and Bonnie, for their support through the difficult periods during this project. Finally, members of VTSFFC, the Science Fiction and Fantasy Club at Virginia Tech, for their camaraderie and support.

Table Of Contents

List of Figures

1.0 Introduction and Background

Over the last ten years, there has been an explosive growth in the use of expert systems for engineering applications. Over 150 commercial expert systems are currently on the market for engineering-specific tasks. The only fields which have seen a greater number of expert systems developed are the business, medical, and manufacturing fields [1]. Many of these systems are dedicated for use in power plants and other heavy industrial applications. Most of these (particularly those designed for nuclear power plants) have been designed to aid plant personnel sort through and interpret the assorted alarms and warnings the plant equipment generates [2,3]. Few, however, have been designed to be low-cost systems for smaller power plants, capable of reading data off of standard plant sensors and creating it's own set of alarms and bringing them to the attention of the operator.

This thesis discusses the on going effort at the Virginia Tech Rotor Dynamics Laboratory to develop an inexpensive system to monitor, evaluate, and diagnose rotating machinery in industrial applications, such as power plants, through the use of a personal digital computer. This project began with the development of an off-line rule-based expert system. [4,5,6,7] The extension of this off-line system into an on-line expert monitoring system capable of multiple machine monitoring and trending forms the basis of the current research [8,9,10]. This research is similar to the VARMINT system that has been developed at Design Maintenance Systems in North Vancouver, BC [11].

This thesis will discuss the various tasks that were necessary to further this project. These include extensive modifications to the original version of the off-line WOT expert system, the logic that was used to determine what form the on-line WOT system and monitoring system should take, the

development and form of the "standards" used to evaluate data, the flow of the various parts of the system, and a discussion of each of the major parts. Finally, there is a discussion of the installation and operation of the system at the Virginia Tech physical plant.

2.0 The WOT Expert System

The WOT expert system is a PC-based system designed for diagnosing turbines, compressors, and similar pieces of industrial machinery, although it could be used for other applications. It was written in the Prolog programming language, and is designed to be simple and user-friendly, without requiring any special programming or database knowledge to use.

The WOT system is what is known as a "rule-based" expert system. It operates by considering a list of rules, each one of which represents a potential problem for a machine (such as "loose bearing shell") and inferring which ones are most likely to be true, given a specific machine and a series of input data from that machine. The likelihood of a rule being true is referred to as the "confidence" the system has in that rule. Rules are evaluated by asking a series of questions, or "conditions", each one of which represents a possible symptom that relate to the rules (such as "is the 1x vibration high?)". Every rule is linked to at least one condition, all of which can be answered by "yes", "no", or "don't know". Whenever a condition is answered "no" or "don't know", the confidence of every rule linked to that condition is decreased. Weighing factors, called "confidence factors", are used in every link between a rule and a condition, to establish how important each condition is to a rule's confidence. The process by which rule confidence is determined is discussed in "An Expert System for Off-Line

Analysis of Rotating Equipment", by James R. Hoglund [8].

In addition to conditions, rules can be linked to "references", "standards", and "more to do" items. The first two types of items contain information about published reference texts or industrial standards that are applicable to the type of problem the rule describes. A "more to do" item contains a suggested method of dealing with the problem.

2.1 The On-Line WOT Intelligent Monitoring and Trending System

The first form of the WOT system was operated in a completely off-line mode. This system was not created by this author, but rather by his predecessors in this project [6,8] Conditions were answered by an operator when the system was executed. Many improvements have been made to the off-line system by this author, mostly as a result from user feedback. However, the ultimate goal of this project was to give WOT the ability to automatically evaluate conditions from data acquired directly from an operating machine.

To determine if a measured machine parameter or series of parameters (such as speed, vibration, or phase angle) meet a condition in the expert system database, it is necessary to have some means of automatically evaluating the parameters. The easiest way to accomplish this would be to individually assign each condition in the database to a parameter, and have the condition be evaluated as "yes" or "no" based on the value of the parameter. Such a system would have severe limits, however. It would have to be completely reconfigured for each different machine. It would have difficulty handling conditions that depended on more than one parameter at a time, and the connections between the data and the conditions would not be very intuitive.

Therefore, it was decided to go with a more complex system that would mimic the way a human expert evaluates a

machine. It would take data from the machine and compare it to a series of standards, which will be discussed in detail in the next section of this report. The results of these comparisons would be used to answer the conditions in the expert system. The acquired data, as well as the results of the comparisons, would be displayed to the user. Standards can easily be generalized to cover a wide array of machines, and, more importantly, they provide an intuitive connection between the data being acquired from the machine and the machine's status.

These abilities turned out to be so useful that they were developed separately as the Intelligent Monitoring and Trending System (IMTS). Various "hooks" were placed in the monitoring system so that the remaining parts of the expert system could be attached later [6,9,10].

2.2 The WOT Expert System Editor

Whether operated in on-line or off-line, the WOT expert system requires a large database of rules, conditions, and other facts to operate. These databases are created and/or modified by the WOT system editor. Like the expert system itself, the editor is written in the Prolog language and is menu-driven.

Creating a new database from scratch must be done by a human "expert". This expert would devise a list of potential problems and potential symptoms that a machine might display, and let each one become a rule or condition. Each condition would be assigned to the various applicable rules along with estimated confidence factors.

For many applications, a database does not need to be created from scratch. The WOT system comes with a number of pre-generated databases for assorted types of machinery, such as turbines, generators, motors, and fans. Naturally, since

these databases were not created with any specific machine or plant in mind, they are extremely general. While they can certainly be used as is, they are meant to be modified by the end user, using both the WOT editor and the automatic answer program, which is a portion of the IMTS system. The editor would be used to modify the confidence factors of the built-in conditions and to add any special conditions that may apply to particular cases. The automatic answer program can be used to permanently "set" the answers to any conditions that would always be answered the same way (such as "does the machine have oil seals?).

Anytime the WOT system is used, there is an opportunity to further refine the database and improve its accuracy. When a database has been run through, the WOT system will note the confidence factors of each of its predictions, and can show how it arrived at this confidence through application of the various conditions. At this point, the operator can alter how each condition effects the various rules to improve the system's ability to more accurately predict the problem the next time.

2.3 Editor Revisions

The original version of the editor, created by James Hogland [8], operated correctly with operated with small, developmental databases. However, when used with large, operational databases, it tended to run out of RAM and crash after correctly performing a few database operations. The number that could be performed seemed directly proportional to the size of the database. This behavior was puzzling, at first, because the program uses the "external" database structure, as defined by PDC Prolog (Prolog Development Corporation). External databases store most of their data on disk, and only pull into RAM the portion of the data currently

in use.

Investigations by the author showed that the memory problem was related to the manner in which the editor presented menus that allowed the user to select an item from a list, such as a list of rules or conditions. Each menu was defined as a single Prolog routine, called a "predicate". (A predicate is similar to a subroutine in more conventional languages. See Appendix A for more details on the Prolog language). These predicates would make the list used by the menu with data from an external database with a recursive sub-predicate, present it on the screen, accept a key (typically a function key) as input, and then jump directly to another predicate, in a manner somewhat similar to a GOTO statement in FORTRAN. Unfortunately, the list remained in RAM permanently. After enough operations were performed, all available memory would become clogged with lists, causing a crash.

The solution that was eventually determined was to completely restructure all menu predicates used by the program so that, after the selection was made, the program would backtrack across the predicate that formed the list. Since any predicates backtracked across are effectively "undone", the lists in RAM are eliminated. The specific form of menu currently used is presented in detail in Appendix A.

Other additions and improvements were made to the editor, mostly as a result of feedback from actual users of the first release. Many complaints were filed because the first release could not delete anything from a database besides a rule. This was apparently done to avoid having a rule lined to a non-existent item. The next release of the program was given the ability to delete conditions, recommended fixes, library references, and standards that are unused: i.e., not assigned to a rule. As an additional advantage, attempting to delete any of these items that are used will cause to program to list

all rules that the item is assigned to. The previous version had no way of providing this information, short of printing out the entire database and going through it manually. Options were added to allow program to search through a list of items and display those that contained a search string given by the user. Users report that both of these abilities are highly useful when working with large databases.

The initial release was unable to handle an empty list of rules, conditions, fixes, references, or standards. If the user attempted to display an empty list of such items, the editor would abort to a "create" option and force at least one item to be created. To avoid this annoyance, new menus were added, which give the option of creating a new item or returning to a different menu.

One of the most unexpected user requests that was implemented was the ability to change the order in which conditions are assigned to a given rule, which is the order in which the expert system asks them at run time. While the order does not affect the evaluation of the rules, many users evidently felt uncomfortable answering the questions in certain orders. This ordering ability was also given to the assignment of recommended fixes, references, and standards.

3.0 Development and Purpose of the IMTS

The monitoring system began as an outgrowth of the on-line expert system project. The idea behind this project was to allow the computer to answer conditions in the WOT expert system database using one or more parameters (such as speed, vibration levels, or phase angles) that are measured from an operating machine, instead of requiring an operator to manually answer each condition. This requires that the computer be equipped with hardware that allows it to monitor data directly, and with software capable of analyzing this

data and comparing it to the WOT database.

Developing custom hardware was not necessary for the project, as many varieties of data acquisition equipment for PC's have been available for some time. It was decided to make the system as independent of the specific brand of data acquisition hardware as possible. This was accomplished by making the system a series of interconnected programs, rather than a single program. Only one program, the simplest, controls the hardware. This program, the acquisition program, acts as the interface between the rest of the IMTS software and whatever hardware is being used. This modular software allows the system to be installed in existing installations with a minimum of modification. It also allows hardware to be upgraded without requiring a simultaneous rewrite of the software.

Thus, the basis of the IMTS project are these various software modules. Besides the acquisition module, there is an evaluation module, which evaluates data, the display module, which acts as the main operator interface when the system is running, a setup module that is used to teach the system about the installation that is to be monitored, and standards generation programs. All of these modules were written for the DOS operating system. The assorted run-time modules are designed to be multitasked using Windows.

Conceptually, the most difficult part of designing the IMTS system was determing how the data would be used to answer WOT conditions. The simplest way would be to individually assign each condition in the database to a parameter, and have the condition be evaluated as "yes" or "no" based on the value of the parameter. Such a system would have severe limits, however. It would have to be completely reconfigured for each different machine, since what might be considered a high vibration on one generator might be barely above runout on

another. It would have difficulty handling conditions that depended on more than one parameter at a time, and the connections between the data and the conditions would not be very intuitive.

Therefore, it was decided to go with a more complex system that would mimic the way a human expert evaluates a machine. It would take data from the machine and compare it to a series of standards. The results of these comparisons would be used to answer the conditions in the expert system. The ability to simply evaluate acquired data with standards turned out to be so useful that it was developed separately as the Intelligent Monitoring System.

As previously mentioned, the system has a number of different programs, or modules. This allows the program to be as independent of its hardware as possible. Making the system modular had additional advantages: it allows the system to take advantage of the strengths of many different programming languages. For instance, the data acquisition module was written in QBASIC, primarily to simplify any modifications that must be done to adapt to different kinds of data acquisition hardware. The display and system setup modules were written in Pascal, to take advantage of the advanced screen handling routines available in that language. The data evaluation and standards configuration modules were written in Prolog, because of that language's inherent ability to easily manipulate and interpret large amounts of data. This modularity also allowed the overall system to be more complex than would normally be possible with a single DOS program.

Making the system modular did have its drawbacks, however. The chief among these were the difficulties involved with getting multiple programs written in different languages to interact with each other in a reasonably efficient manner. The solution eventually settled upon was to have all inter-

module data transfer done through standard ASCII text files. Data is passed sequentially, from the acquisition module, to the analysis module, to the display module. Various assorted commands are also sent between modules in this way. The format of each transfer file was carefully chosen so the system could be easily adapted to different situations.

The current version of the IMTS is, in fact, its second generation. The previous version (described in "IMTS (Intelligent Monitoring and Trending System, A PC-based monitoring System for Rotating Machinery" [9]) was similar in that it used a number of modules that communicated via ASCII text files while running under Windows, and had the same overall structure. Preliminary tests found severe shortcomings in the first generation system. It relied on various pieces of specialized hardware to collect data, such as a Bently-Nevada Digital Vector 3 to collect vibration data, and a Rapid Systems FFT to provide vibration spectrum data. These devices were controlled through a variety of interfaces, such as an IEEE-488 bus, an RS-232 port, and specialized manufacturer-supplied plug-in PC boards. This hardware was originally chosen to simplify the IMTS code and speed the system's operation. However, it raised the cost of the installed system by a substantial amount. It also limited the system's versatility. The software required to control this hardware was so specialized that it seemed unrealistic for an end user to be able to adjust the system to efficiently run with different hardware. The first generation system was unable to collect data from more than four separate machines, with a maximum of thirty-two vibration probes, twenty thermocouples, and sixteen generalized analogue probes between them. The user interface on both the configuration programs and the display program were crude and difficult to use.

The second generation of the system has overcome these

shortcomings. Rather than utilizing specialized data acquisition hardware, all data is acquired through simpler boards equipped with digital-to-analog converters. The boards proved much cheaper than the hardware used by the previous version. Although the IMTS acquisition software is still written for the specific model of analog-to-digital boards that were used, customizing it for different hardware would involve little more than changing minor details, such as the names of function calls, and re-compiling under BASIC.

The main disadvantage to using this method is that the IMTS software must handle all data processing and manipulation itself. The original system farmed out some of these tasks to the specialized hardware attached to the PC, and only read in the results of the manipulation. In order to handle the additional work, the IMTS code became larger and more complex, with many new procedures. Rather than slowing the system down, however, these changes sped the system up. This can mostly be attributed to the high speed of the PC used and the ability of the data acquisition card to acquire data through Direct Memory Access (DMA). With DMA, most of the computation needed to acquire data is performed on the card, requiring little PC processor time. The shear speed of the PC used (an IBM-comparable PC with an Intel 486 main processor and a 33 MHz clock) simply outstripped the older data acquisition hardware employed by the first generation system.

Besides the modifications needed to handle the hardware conversion, other major changes were made in the way the second generation IMTS software is used. The configuration and display modules were completely redesigned, using a sophisticated series of routines supplied by Borland, Inc, in their Borland Pascal 7.0 compiler. The data analysis modules were modified to better handle increased amounts of data. Representatives from Virginia Power were consulted to

determine what user interfaces would best suit actual use in a power plant or other similar installation.

3.1 IMTS STANDARDS

In order to understand IMTS, the standards that it uses to evaluate data must be explained. The IMTS standards were designed to mimic standards that a human expert would use. A review of published standards showed that many of them were expressed as graphs of an independent variable (often a vibration level) and a dependent variable (typically speed in RPM). Therefore, the first standards created for the monitoring system were of this type. They are created by the IMTS program GRAPH.EXE.

A sample graphical standard for a hypothetical compressor is shown in Fig. 1. It consists of four lines, each defined by a number of points. Each of the lines divides the area of the graph into regions, in this case, "excellent," "good," "marginal," and "unacceptable." The monitoring system's evaluation module evaluates acquired data against this standard by effectively plotting it against the graph and determining which is the lowest line that passes over the data point. The name of this line is passed on to the display module, where it is eventually displayed to the operator.

After consulting with some of the potential users of the system, it was decided that graphical standards alone would not be sufficient to fully evaluate the data. In particular, graphical standards could not easily be used to evaluate vectors formed by plotting amplitude of vibration and phase angle. A special kind of vector standard was created to handle this case. Figure 2 shows a sample vector standard.

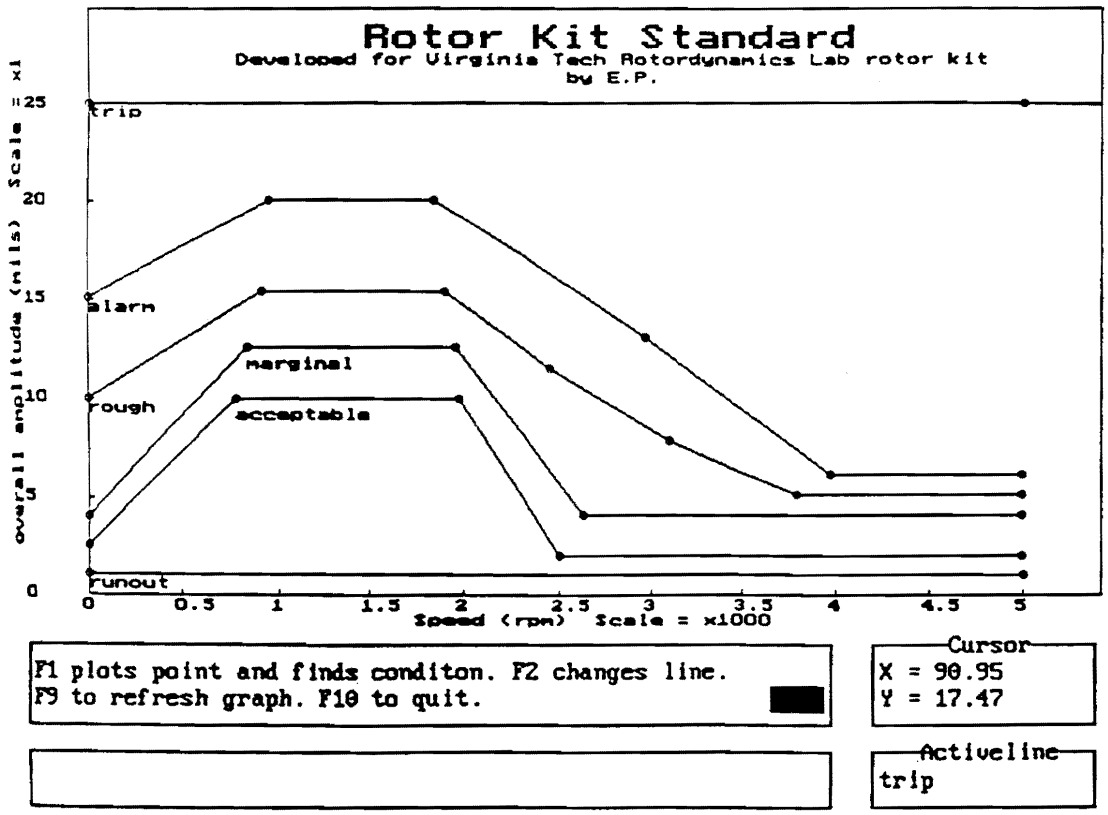


Figure 1

Graphical Standard for Overall Vibration

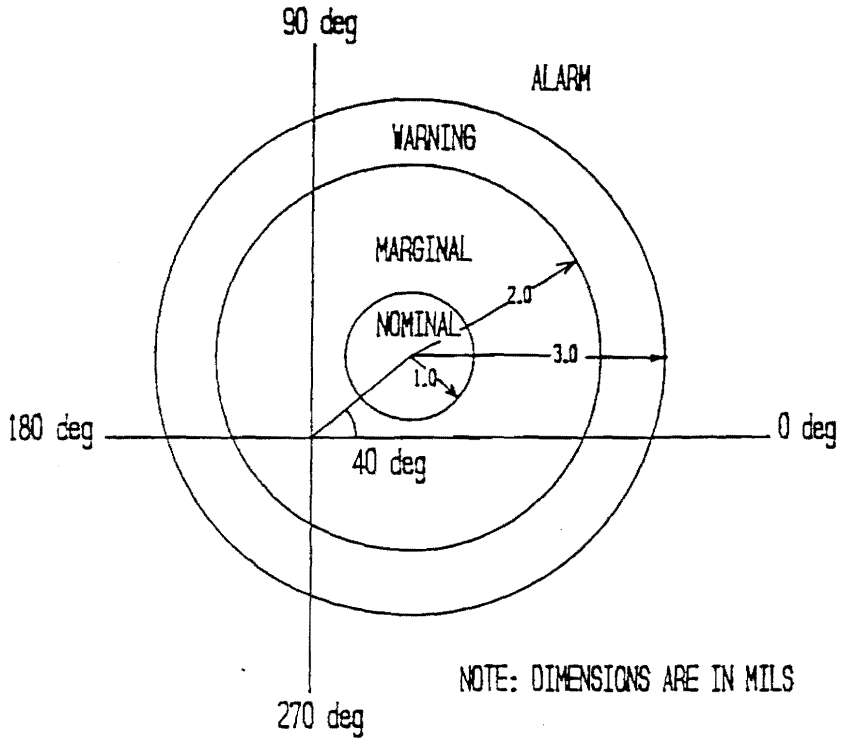


Figure 2

Vector Standard for Synchronous Amplitude and Phase

Similar to graphical standards, the evaluation module works by making a polar plot of acquired data and determining which is the smallest circle that contains the data point.

The simplest type of standard are the scalar standards. This type evaluates a single parameter only, and may be thought of as a number line, as shown in Fig. 3. The line is divided into a number of regions. Data is evaluated by determing in which region it would plot into. This particular standard is used to evaluate gap volts, which represents the static displacement of a shaft, and is thus symmetric about its center point. Unsymmetric scalar standards can also be defined, which judge the data not only on how far it is from the nominal region, but whether the data is higher or lower than nominal. Scalar standards have proven to be the most popular kind used by the system, because they are so simple to create and interpret. Since most industrial turbomachines operate at a fixed speed, scalar standards can be used to check their vibration. In this case, there is little need to plot the data on a speed vs. amplitude graph.

Scalar and vector standards are alike in that they are defined in terms of one- or two-dimensional spaces around a center point. Scalar standards can be defined as fixed, in which case the numerical value of the centerpoint (and all of the regions above and below it) cannot be changed. Alternatively, some scalar and all vector standards can be reset, in which case the centerpoint can be reset while the system is operating. All regions above and below the centerpoint will change by the same amount. When a standard is reset, the program sets it to be equal to the current conditions the machine is operating under. Resetable standards are meant to be reset when the machine has reached

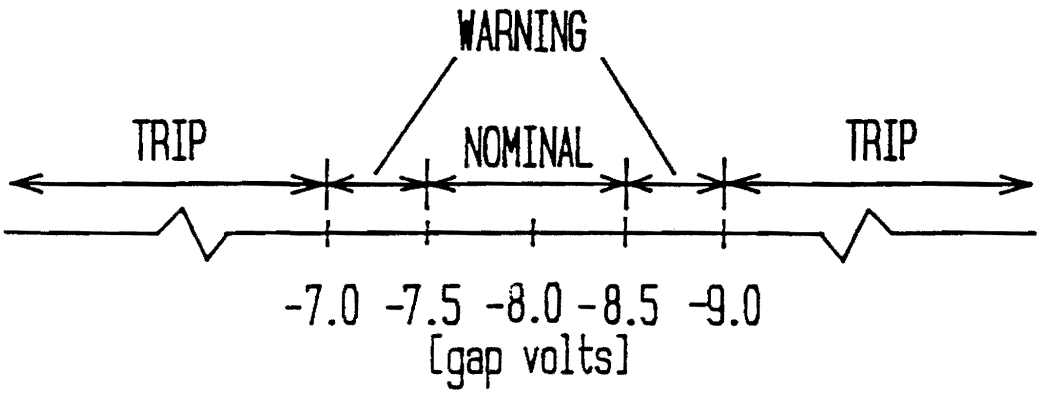


Figure 3

Scalar Standard for Gap Volts

a stable operating state. Once reset, they will indicate if the machine has strayed from that state, even if it remains within the allowable operational states, which are defined by the non-resetable standards. Graphical standards cannot be reset, since they have no form of centerpoint.

Standards are used not only to determine if the machine has strayed from its nominal state, but to call attention to a machine that is experiencing serious difficulty. Any standard can be set with one or more "alarm" states. When one or more of a machine's parameters enters one of these states, the display program will immediately call that machine to the operator's attention.

3.2 IMTS Hardware

The IMTS system has been developed to run on an IBM-compatible PC running the Windows 3.1 multitasking system in 386 enhanced mode, with at least six megabytes of RAM. It can operate with less memory, using disk space as simulated RAM, but only with a substantial reduction in speed. All files used to transfer data and commands among the various modules are located on a RAM disk to increase speed. The system was developed on a Swan PC with a 80486 main processor and a clock speed of 33 MHz. This computer was used in the installation at the Virginia Tech Power Plant.

The system acquires data through a series of external and internal boards. Figure 4 diagrams how these boards connect to each other, the raw data inputs from the machines being monitored, and the control computer. Analog data is acquired through a Keithley-Metrabyte DAS-1602 data acquisition board, mounted in one of the PC's expansion slots. This board has eight input channels, and a maximum data collection rate of 100,000 Hz that can be divided among those channels. Only

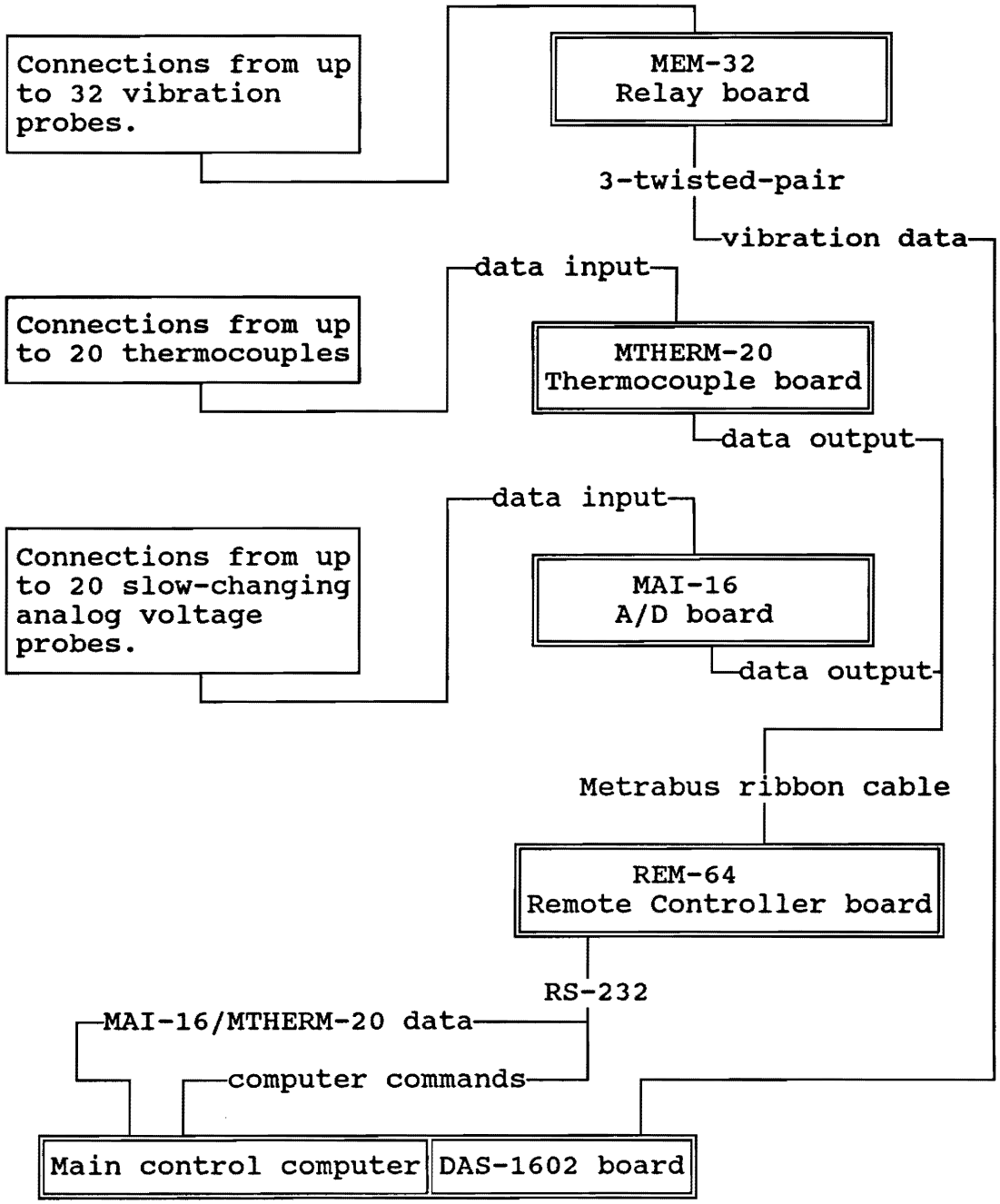


Figure 4

IMTS Hardware Connection Diagram

three channels are currently used in the IMTS system: two for vibration data, one for keyphasor data.

These three channels are connected by twisted-pair wiring to a series of one or more external relay boards, which are typically located much closer to the system being monitored than the controlling PC. These relays switch the various probes and sensors on the machines between the three channels. The system uses MEM-32 relay boards with 32 relays each.

Besides the relay boards which pass analog data directly into the DAS-1602's high-speed A/D converter, other, more specialized boards that contain their own slower A/D converters. These include the MTHERM-20 thermocouple board, which reads the voltage from all standard types of thermocouples and converts it directly into a digital temperature. There is also the MAI-16 A/D board, which can convert any analog voltage into a digital voltage.

All of the external boards are controlled by a Metrabyte REM-64 remote controller board. This external board communicates with the PC via a standard RS-232 serial communications line. The REM-64 can control up to sixteen data acquisition boards, which means that the IMTS system can monitor up to 512 vibration probes, or a lesser number of other probes.

This setup was chosen primarily to reduce the wiring that would be required to install IMTS in an actual plant. The probes on each machine need only be connected to the external relay boards, which can be located as close to the operating machines as desired. These boards are connected to the computer running the IMTS by a maximum of up to three twisted pairs for the DAS-1602 vibration channels, and one serial cable for the remote controller board. Thus, the computer need not be located near the machines being monitored, but may be placed at a more convenient location in the plant.

3.3 IMTS SOFTWARE

When in operation, the standard IMTS system consists of three executable programs, two configuration files, and several transient files for data and command passing. In addition, up to four setup programs may be required to create the configuration files. Figure 5 shows how each of the three programs interact with each other and the assorted data files. Data is acquired directly from relays and switches by the acquisition program, AQUMOD.EXE. This program knows which relays to acquire from and what sorts of data each one contains by reading the configuration file EXPSYS.CFG, which is created by the setup program EXPSETUP.EXE (not shown in Fig. 5). Any applicable FFT standards are contained in FFTSTD.DAT, a configuration file created by FFTSTD.EXE (also not shown). From the acquisition module, data is written into the file EVAL.PLG and passed directly to the filter program, FILDATA.EXE. This program evaluated the data with the standards contained in the file ODD.SET, which is created by the setup program WOTSETUP.EXE (not shown). After being evaluated, the data and evaluations are written into the file EVAL.DAT where they are read by the display program, DISPPROG.EXE. This is the only program that the operator would routinely interact with. Both of the files that contain the acquired data (EVAL.PLG and EVAL.DSP) are conventional ASCII files. Each program creates them by writing data as it is acquired or evaluated into a temporary file, closing this file, then renaming it to the names given below (for example, the acquisition module writes data into "EVAL.TMP" until it has acquired one piece of data from every probe in the system, then it closes EVAL.TMP and renames it to EVAL.PLG). The next program "down the stream" constantly searches for a file with the appropriate name (FILDATA.EXE constantly searches for a new EVAL.PLG file, and begins processing as soon as it finds

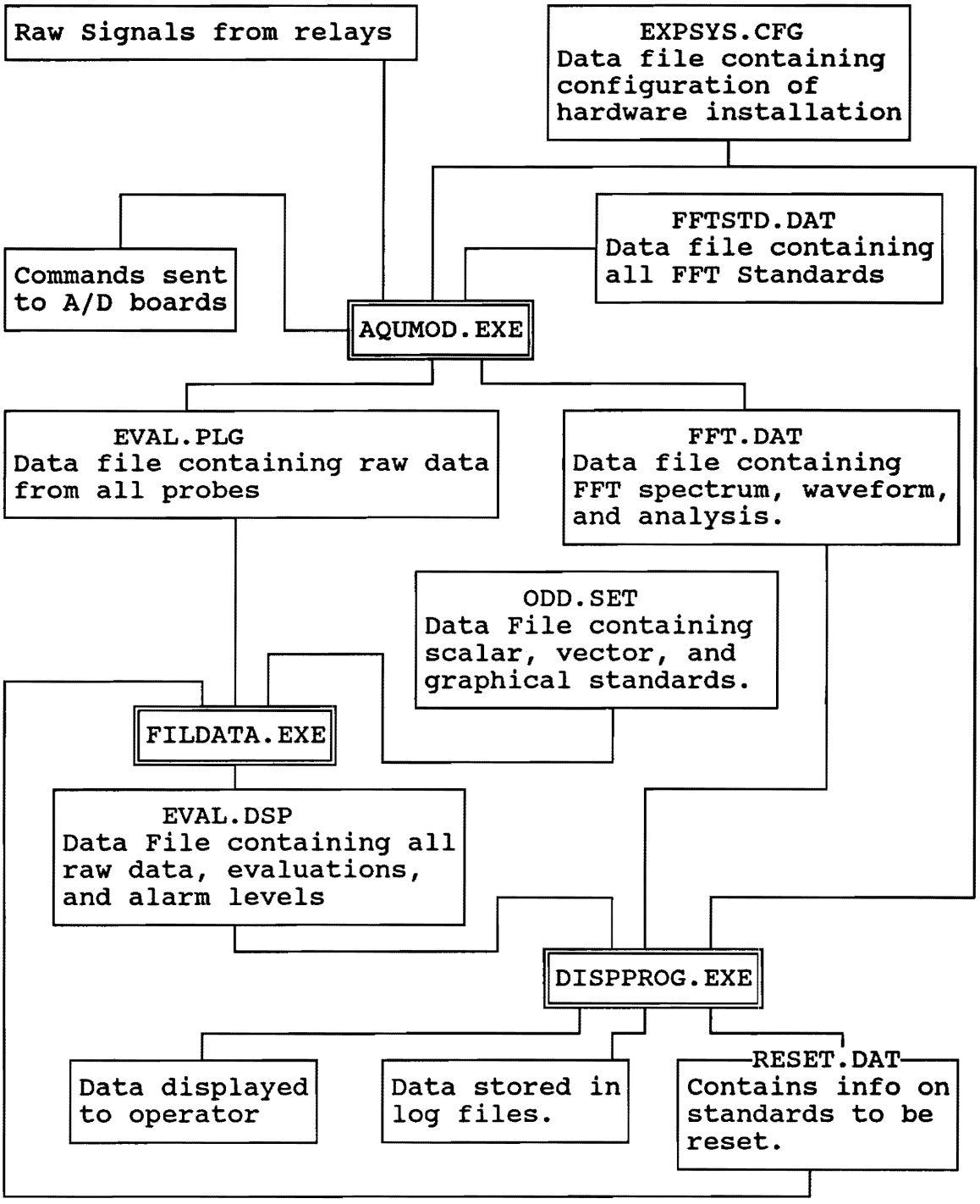


Figure 5

Flowchart of IMTS Program Operation and Data Management

one). This elaborate setup was necessary because each of the modules was written in a different language. The only types of files that could be freely passes between them are ASCII text files. The system of writing to one name and then renaming is used to prevent two programs from attempting to read the same file at the same time and "colliding", which would crash the system.

3.4 IMTS Main Display Program

The display program is the only portion of the IMTS system that is designed for day-to-day interaction with the system operators. This program is ultimately responsible for the display of gathered data and evaluations to the user, and for passing commands from the user to the rest of the system. In addition, this program is responsible for storing data into trend files, and for deleting information from these files when, in the user's opinion, the information has become too old to be of use.

Figure 6 shows a typical display screen for the display program, taken from the system's test installation at the Virginia Tech power plant. It contains a pull-down menu bar at the top for executing various system commands and a large display area for machine information. In its default mode, the program will display a series of small windows, each containing certain information from one machine. This information includes the name of the machine, which is divided into three parts- "unit", "system" and "machine", each separated by dashes. (This is not needed for this particular installation, since the Virginia Tech Power Plant only has one turbine and generator, but would be necessary were the system installed at a larger plant with multiple turbogenerator

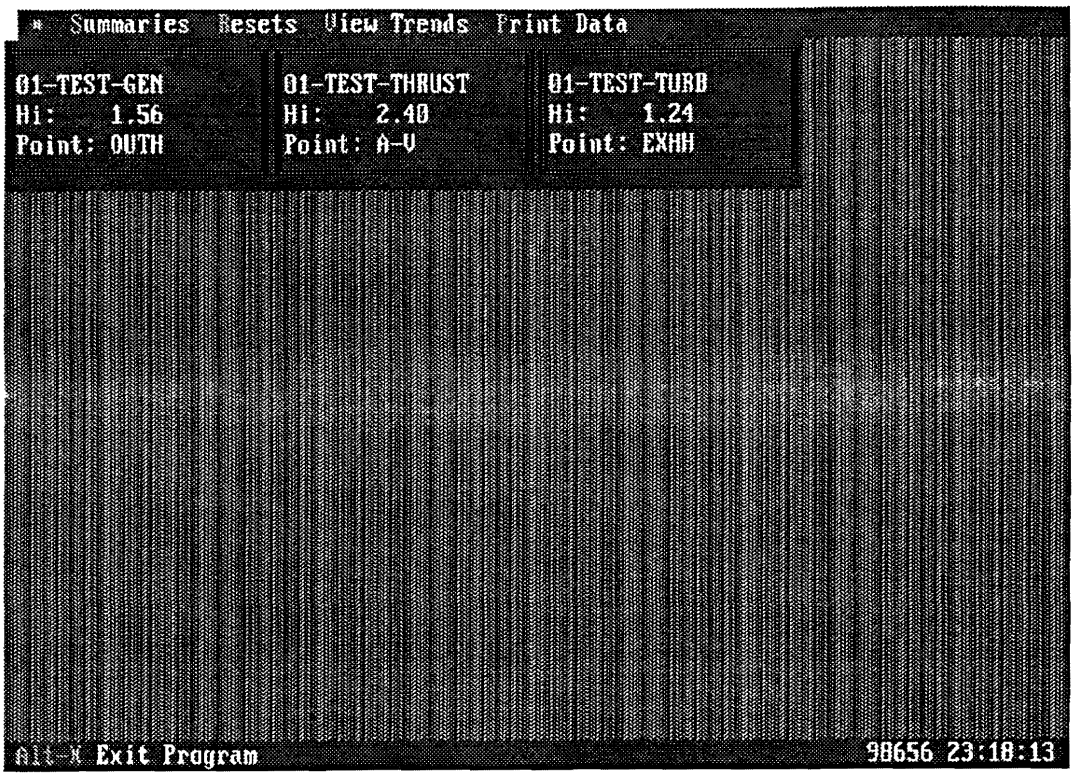


Figure 6

Typical Screen for IMTS Display Program

sets). This is based on the nomenclature system used at most power plants. Below the machine name is noted the highest level of vibration currently registering at that machine, and the name of the probe (a text string) which that information is from. The color of this box identifies the state of the machine in question. Tan indicates that the machine has not met any of the "alarm" conditions in an IMTS standard. Flashing red indicates that the machine is currently in an alarm condition on at least one standard. Steady red indicates that a machine has been in an alarm state in the past, but is not currently in such a state. The drop-down menu item "reset" can be used to "acknowledge" an alarm condition. Acknowledging an alarm will cause its boxes color to change from red back to tan, until more data that is in an alarm condition is received by the program. The menu allows all alarms on the system to be acknowledged at once, all the alarms on a given machine, or an alarm on an individual probe.

More detailed information about any particular probe can be obtained by clicking on one of the small windows. Doing this will pull up two menus in turn allowing the user to first select which kind of probe is of interest (vibration, thermocouple, or generic analog probe) and then an individual probe. The program will then display a "large" window overlaid on the small windows on the screen. Figure 7 shows a typical large window for a vibration probe. This window gives the name of the machine and probe as well as all information gathered about that probe. In an optimal case, this information consists of the rotary speed of the machine, the overall vibration level from this probe, the 1x vibration speed, the phase angle of the 1x vibration, and the gap volts of the probe. This information is in the first column in the window. In addition, this window will display the results of any evaluations against IMTS standards that have been done

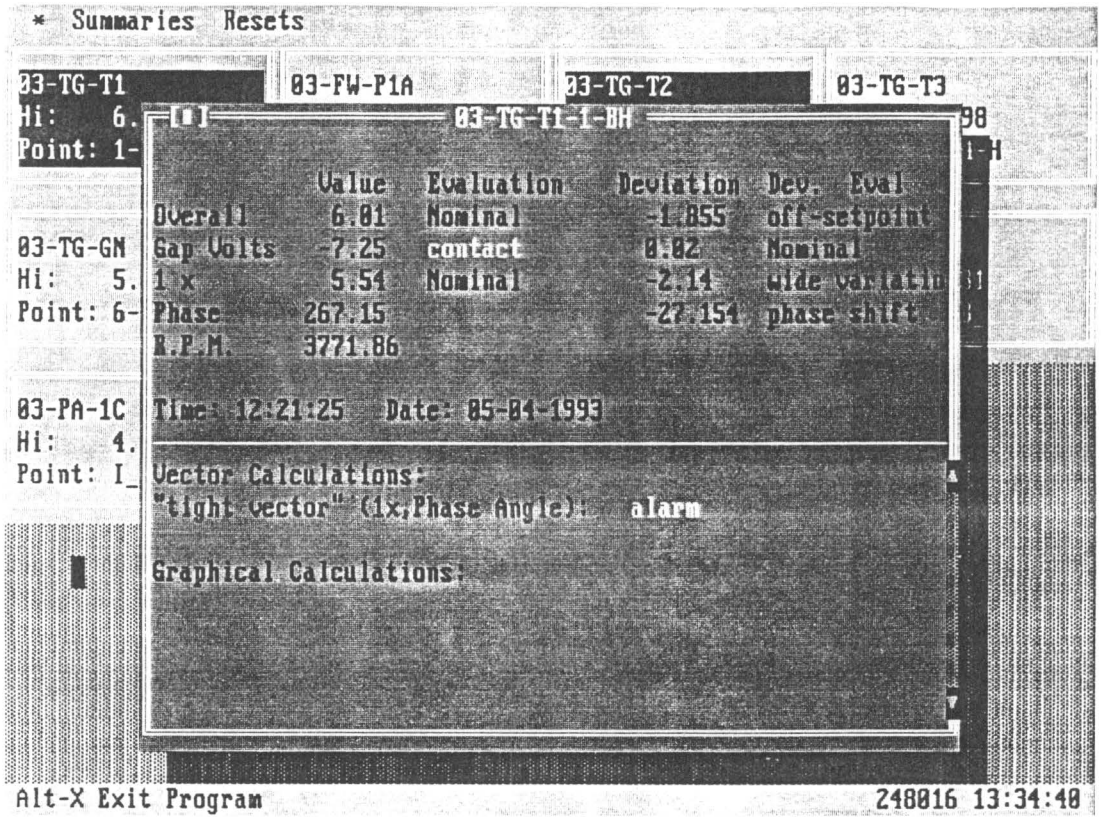


Figure 7

Typical Vibration Display Window for IMTS Display Program

with particular pieces of data. The next column displays the results of any comparisons to fixed scalar standards that have been made. The third column displays the deviation between each data point and any relevant resettable scalar standards, and the fourth and final column gives the results of the comparison to these resettable standards. If a data point is written in black, then the standard is not in an alarm state. Data that has caused an alarm will be written in either flashing white or steady white, depending on whether or not the data is currently in alarm or not. (Red was not used primarily because it proved too hard to read). Resetting the data will cause all standards to go back to being black. Below this data is the time and date at which the information was gathered by the acquisition module.

In the bottom of this window are two sections that are used to display the results of any vector or scalar standards. This section identifies which data are being compared, as well as the evaluation, which is coded by being black, white, or flashing white.

A large window can be "picked up" with the mouse and moved around by clicking on the top bar, and removed by clicking on the top left-hand side. Multiple windows can be opened at once, to allow simultaneous monitoring of many different probes at the same time. Approximately six of these windows can be fit on the screen before they begin to significantly obscure each other. Naturally, any small windows beneath a large window cannot be seen as long as the large window is active.

A persistent problem developed with these large windows. Occasionally, usually when data was received, a large window would become "inactive"- its text would go from highlighted to dimmer, and it could not be picked up, moved, or deleted, and the data on it would not update. Much effort was spent in

attempting to find the cause of this fault to no avail. It appears to be due to some bug in the Pascal compiler itself. The best solution that could be devised was to set the program such that hitting the space bar while a large window was active would refresh all of them. This would "re-activate" the window, and allow it to be handled normally.

The small and large windows are only used for the display of the most current data acquired by the machine. The system is also capable of graphing data that has been acquired over long periods of time. This is done by selecting the "Trends" menu option and then going through a series of menus to design what sort of graph is required. The design of these graphs and the menus required to produce them was one of the most difficult parts of creating the entire display program.

The procedure used is outlined in Fig. 8a-c. From the main pull-down menu option are the four basic functions (shown in Fig. 8a) that are involved with examining trended data. The first, "Select Machine", is used to create a new plot of the data. If this is chosen, the user is prompted for a filename to store this graph's parameters (if none is entered, it will use the default filename "default.gra"). Creating a graph can be a very tedious process. By storing the various parameters in a file, it is possible to call up that graph again using the "Previous Plot" option off the "Trends" menu item.

Figure 8b-c shows how these assorted parameters are chosen. First, the machine and probe that the data came from are selected off of lists. Next, the user is presented with the the "Specify Parameters" menu. Figure 9 shows a typical select parameters menu for a vibration probe that has a keyphasor. The user decides which piece of data from the probe be is to be plotted. In this case, the choices are overall vibration, 1x vibration, phase angle of the 1x

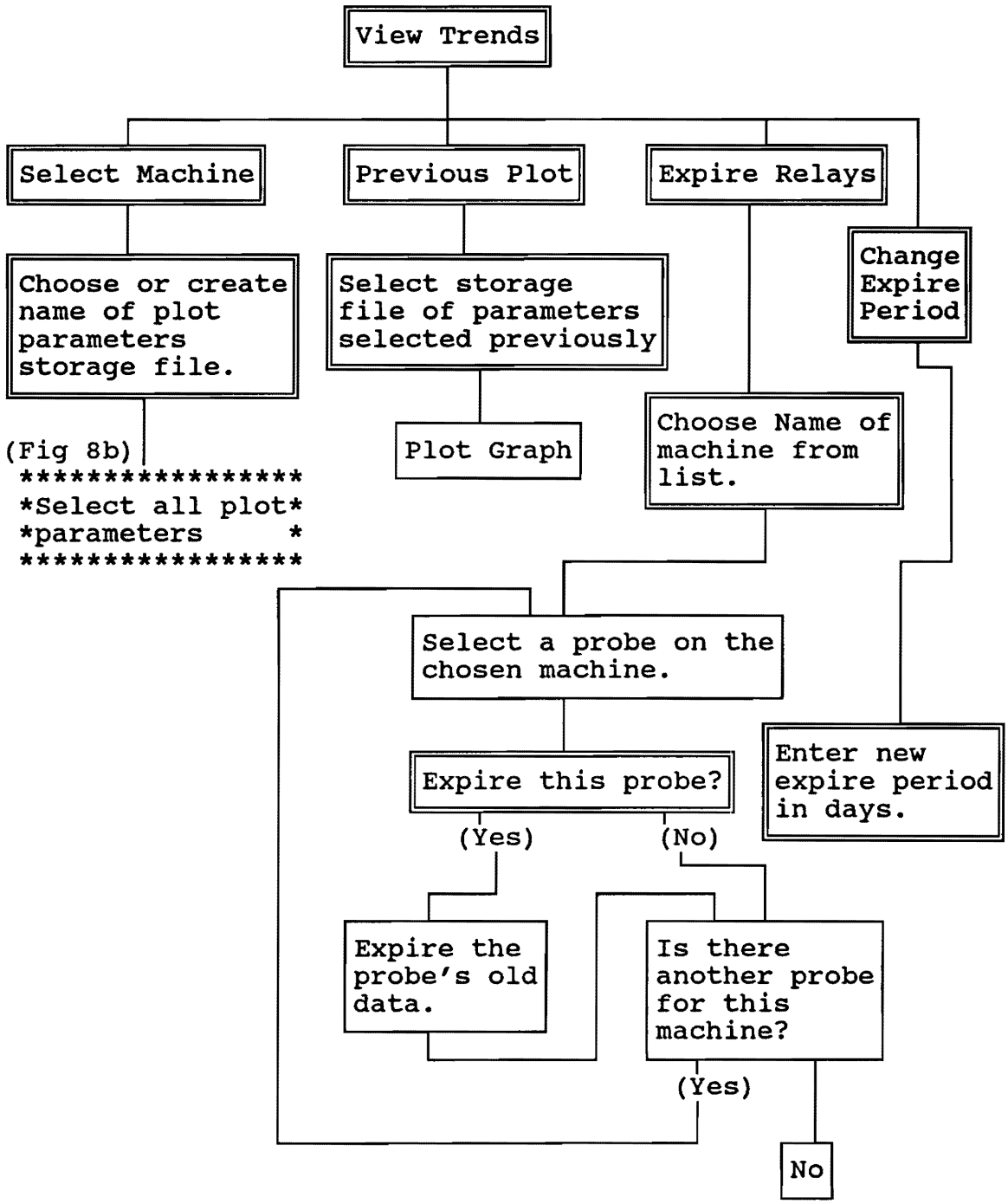


Figure 8a.

Use of IMTS Display Program to Plot Probe Data

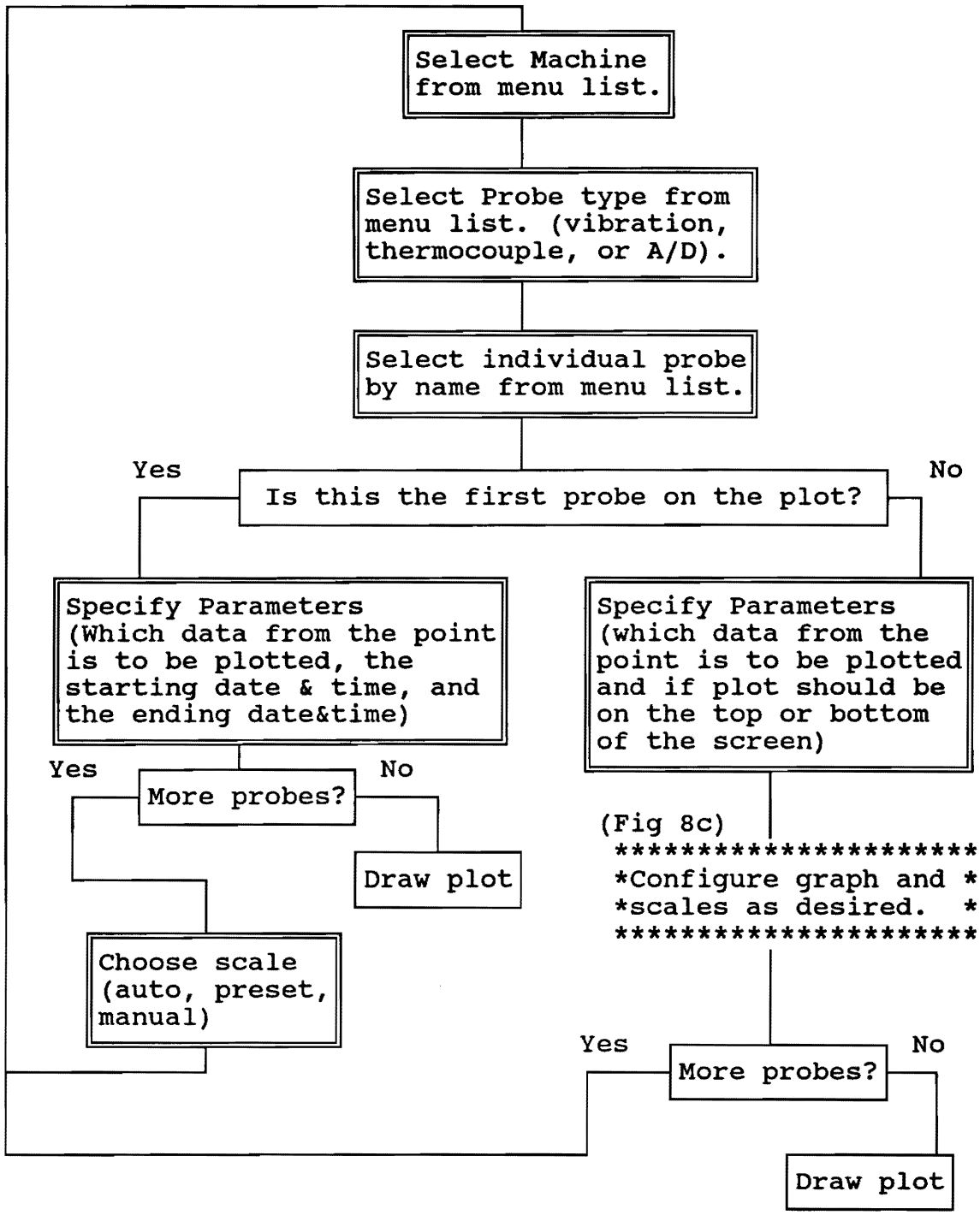


Figure 8b.

Choosing Data Points and Plot Parameters

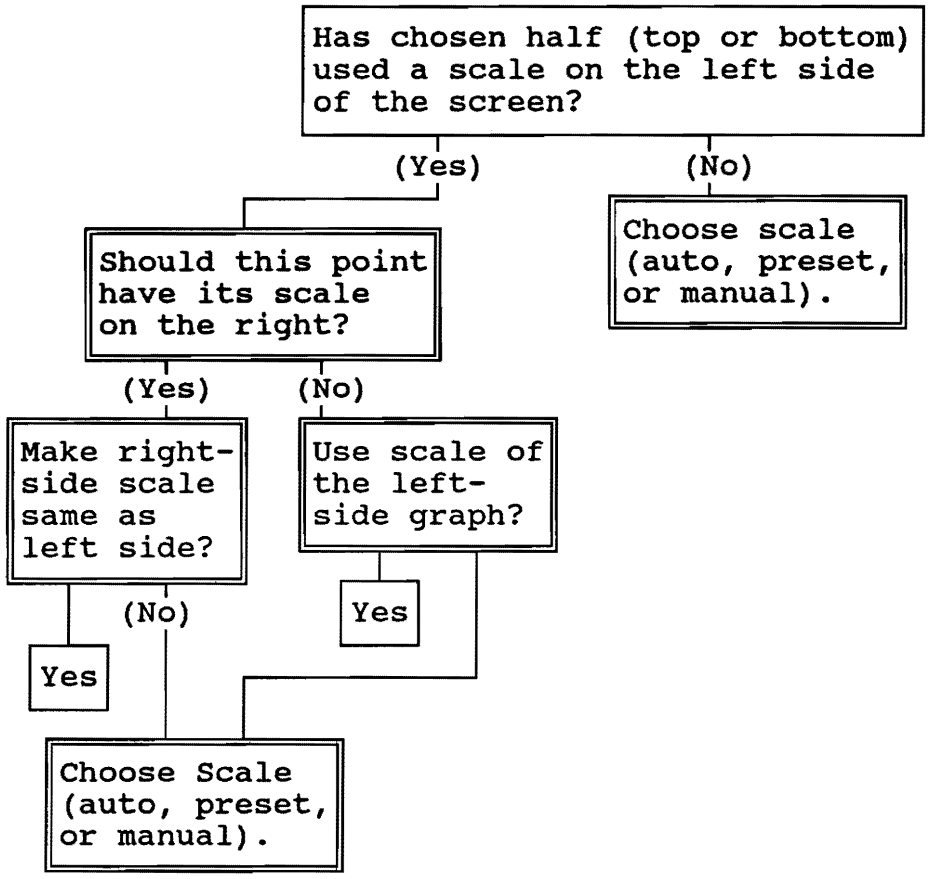


Figure 8c.

Selecting Placement of Graph and Labeling of Axes

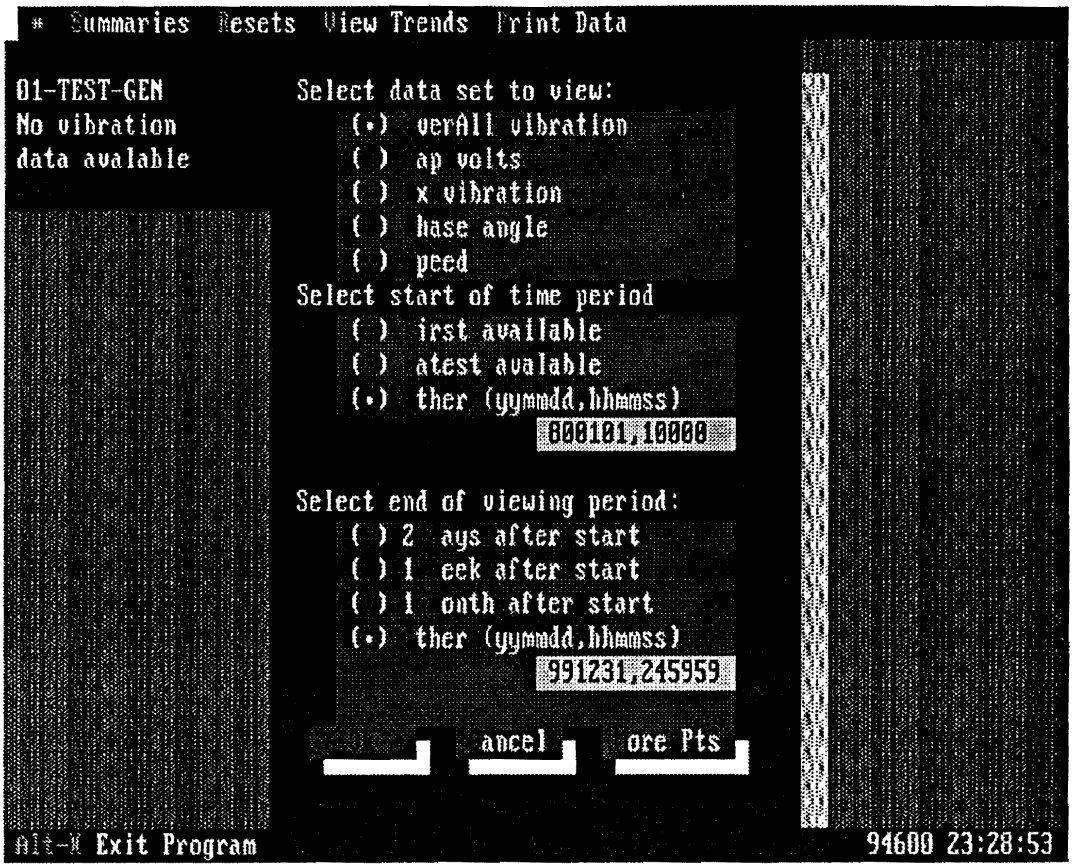


Figure 9

Sample Select Parameters Menu for a Vibration Probe

vibration, and gap volts. (phase angle would not be listed if the probe did not have a keyphasor). Other types of probes, such as temperature and A/D probes, would not have this choice, since they only have one piece of data. The other parameters are the starting and stopping time and date of the data to be plotted. This can be specified in one of three ways. The first is to manually enter a starting date and ending date. The other options are to either start the graph with the earliest piece of data available, or to end it with the latest piece of data in the file. With either of these options, the total time period of the graph can be specified as two days, one week, or two week's worth of data.

Once the first data point has been selected and its starting and stopping points set, other points can be selected (they will all be plotted over the same length of time as the first plot). Up to five points of data can be plotted on the screen at any one time. The lines indicating each plot are indicated by their color, and a color-coded key is present at the bottom of the plot. An attempt was made to devise a scheme that distinguished data points by non-color means, such as using lines of dashes and asterisks, as would be done on a paper graph. Unfortunately, all of these techniques made it very difficult to read small peaks and valleys in the data on a computer screen, and were abandoned. The five plots can be overlaid on one graph, at the top of the screen, or on two graphs on the top and bottom of the screen.

Figure 10 shows a plot with one graph, Fig. 11 shows a plot with two graphs. The single graph has the advantage of a more complete key on the lower half of the screen. Using two graphs makes it easier to separate the various lines from each other. Scales for graphs can be placed on either the

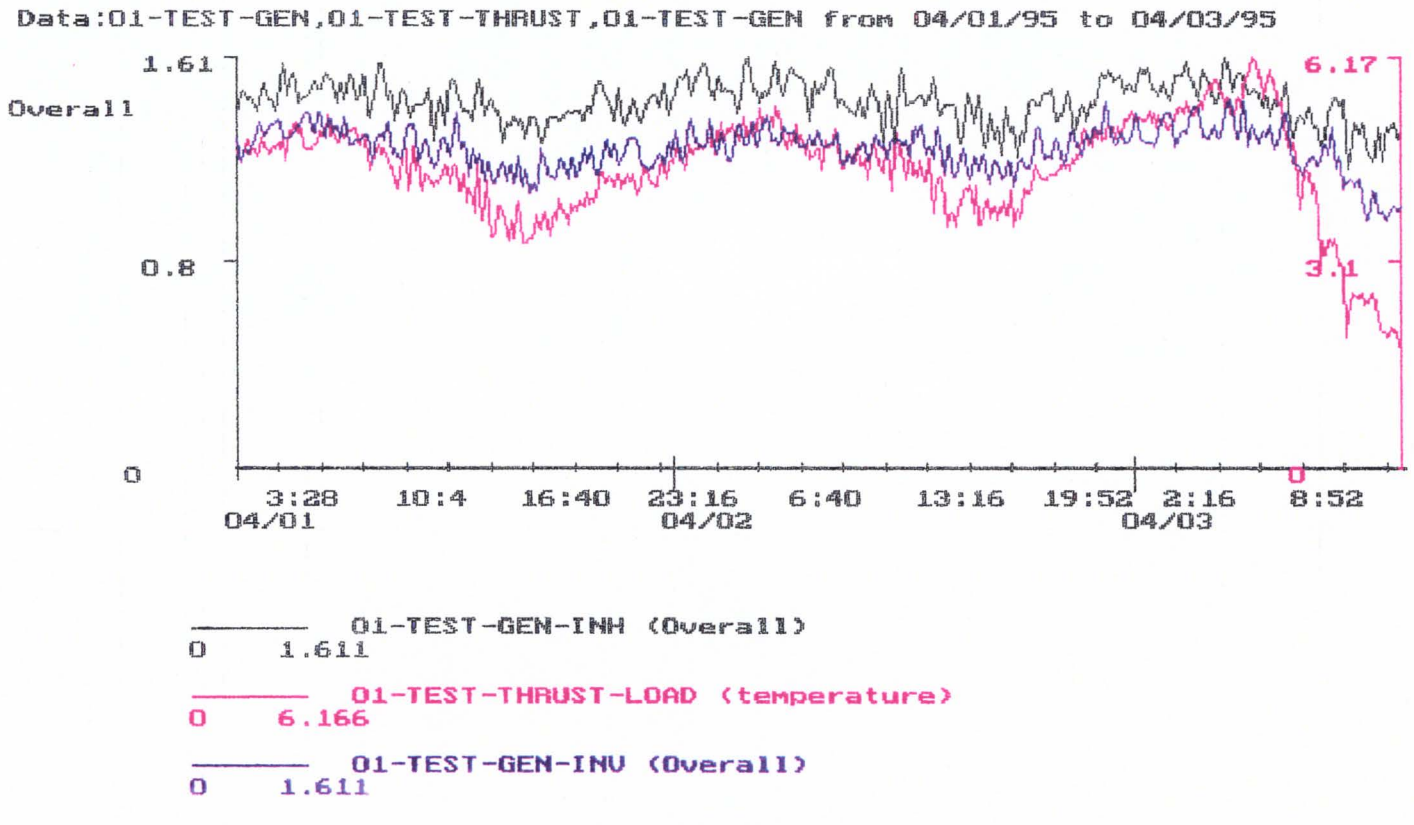


Figure 10

Sample Single Graph from IMTS System

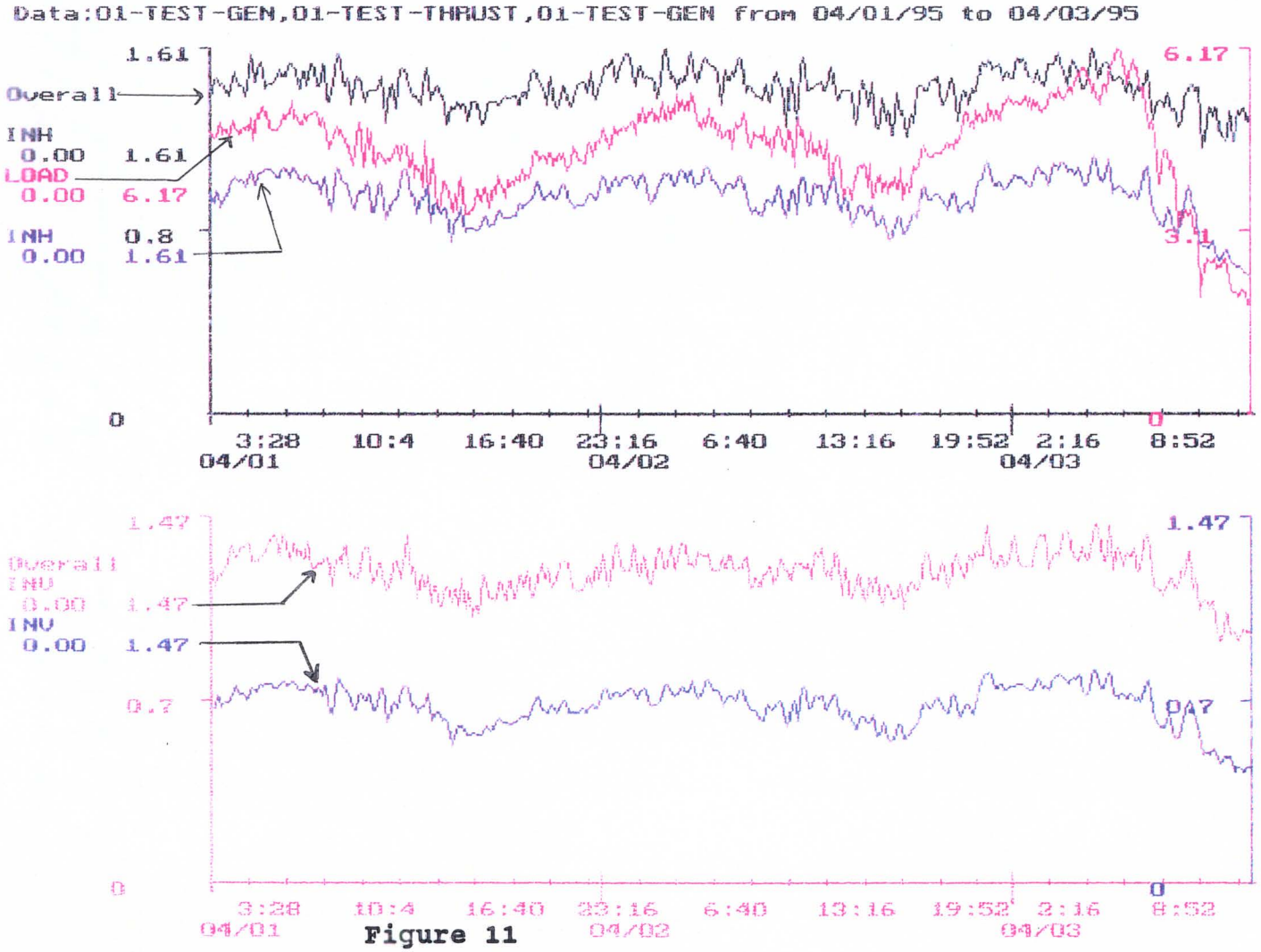


Figure 11

Sample Double Graph from IMTS System

left or right side. Figure 8c outlines the procedure used to pick how the graph scales are used. Any individual graph can be scaled automatically or manually. In automatic mode, program will adjust the scale to read from zero to the maximum in the case of positive data, zero to the minimum in the case of negative data, or minimum to maximum if the data crosses the zero axis. In manual mode, the user must enter in a maximum and minimum, and the plot will be scaled accordingly. There is also a "preset", which will use a maximum and minimum that have been entered manually (there is a different preset for vibration data, thermocouple data, and A/D data).

The horizontal axis of the graph can be along the top of the graph, down the middle, or at the bottom, depending on if the data is positive, positive and negative, or only negative. There can be up four scales on the screen at once (one on the left and right of each half of the screen). Each of the first four data points can have its own scale, use a scale from a point that was previously picked, or have no scale at all. The graph key will show the maximum and minimum for each piece of data. The scale at the bottom has tick marks for each six hours if the time period shown is two days or less, and each day if longer. These tick marks are labeled as closely as is possible without overlapping.

3.5 IMTS Acquisition module

The purpose of the IMTS acquisition module is to provide the interface between the rest of the system's software and the data acquisition hardware. It operates a single high-speed A/D converter, one or more banks of relays that serve as analog multiplexers, and other specialized A/D boards. This module can gather data from proximiter probes, velocity probes, accelerometer probes, thermocouples, and generic voltage measuring probes. A simple, user-adjustable scale

factor is applied to the output from each vibration probe, and a polynomial of up to 10th order may be applied to the output from the voltage probes. Thus, it is capable of monitoring any sensor that produces a DC voltage proportional to the parameter that it monitors, without any reprogramming. Some reprogramming would be required if the system were used to monitor severely nonlinear probes, or probes with an output other than voltage.

The acquisition module has been written in the QUICKBASIC programming language, which is a form of compiled BASIC, as opposed to interperatave forms of BASIC. The module is currently configured to control a Keithley-Metrabyte DAS-1602 data acquisition board and a Metrabyte MEM-64 remote controller board (see Fig. 4), through a series of manufacturer-supplied library routines. The DAS-1602 board is capable of sampling at up to 100,000 samples per second through DMA. Three channels are dedicated for vibration data (typically for vertical and horizontal components of vibration plus a keyphasor). The MEM-64 board is capable of controlling up to sixteen relay boards, each with thirty-two individual relays, for a total of up to five hundred and twelve relays.

These particular types of boards were chosen because they are the main data acquisition boards used by Virginia Power, a former sponsor of the IMTS project. The module was written in BASIC so that it could be quickly adjusted and re-compiled to control different hardware.

As shown in Fig. 5, the acquisition module reads information about each machine from the hardware configuration file, EXPSYS.CFG. It closes groups of relays, which allows the voltage from specific probes to reach the channels of the A/D board. Once digitized, this voltage is interpreted as data by the module, and finally passed on to the evaluation module.

Data is not gathered continuously from any given probe. Rather, each probe is sampled at periodic intervals. Each machine is checked in the order that it is listed in the configuration file. For any given each machine, vibration probes that are paired (usually to sample the horizontal and vertical components of vibration in a single 2-dimensional plane) are checked first, if the machine has any such probes. Then vibration probes that are paired (if any) are sampled. If the machine is equipped with a keyphasor, it is always sampled at the same time as the vibration probes. Then, any thermocouple probes are sampled. Lastly are any miscellaneous voltage probes. Once the last machine in the configuration file is checked, the module starts again with the first machine in the file. Vibration probes and keyphasors are sampled long enough by the DAS-1602 to get 1024 data points. The other probes are monitored long enough to get a single stable reading from the slower A/D converters built into the various boards. When the system is first started up, the first vibration probe on the first machine is sampled for a longer interval, and a 400-line Fast Fouier Transform (FFT) is conducted of it to obtain a full vibration spectrum. The rest of the probes are sampled normally. On the next pass through the probes, an FFT is taken of the second probe of the first machine, etc., until every probe of each machine has had a full spectrum taken. Then the process repeats with a spectrum of the first probe.

This method was taken because it seemed to be a reasonable compromise between system speed and the need to obtain spectral information from each probe. If a full spectrum was taken of each probe at every sampling, the system would run unacceptably slowly, possibly taking hours to sample each probe in the database. By sampling only once per pass, the basic information from each probe will never be more than a

few minutes old no matter how many probes the system monitors, and every probe will have a spectrum taken at least once every few hours.

The operation of the program is outlined in Fig. 12a-b. This program is quite straightforward in most of its operation. All assorted arrays and initialization are performed, then the DAS-1602 board is initialized with the steps provided by the manufacturer. The program's output file, EVAL.TMP, is created (blank), then all internal program counters for an individual machine are initialized and the hardware configuration file (EXPSYS.CFG) is opened. The program then acquires data from the first machine in that file and writes all necessary output to the data file. If there is another machine in the configuration file, the process is repeated until there is none. When the last machine is done, the output file is closed and passed along to the evaluation module (by changing it's name from EVAL.TMP to EVAL.PLG) and the cycle repeats. If the file EVAL.PLG currently exists (presumably because the evaluation module has not finished processing the last one yet) then an error condition is generated and the program loops, continually trying to rename the file until the error condition is no longer created and the renaming can occur. This loop can be broken if a capitol "C" (for "Cancel") is depressed while the program is in the active window (this is meant as the main way of terminating the program). In actual tests of the system, the acquisition program always took longer to process one batch of machines than the evaluation program, so the error condition is never generated.

In processing each machine (Fig. 12b) a certain procedure is followed. For any machine with vibration probes, the

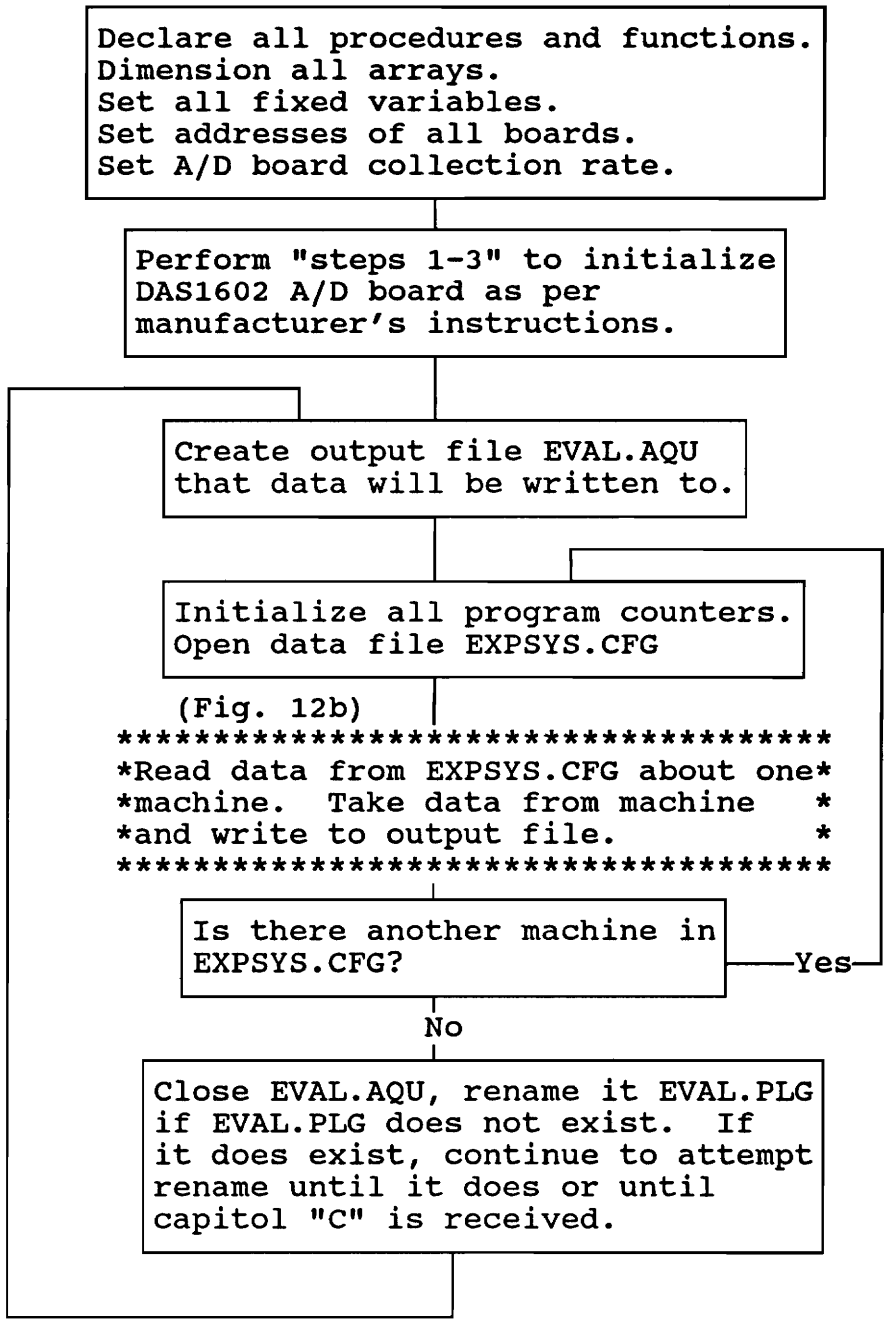


Figure 12a

Operation of the IMTS Acquisition Module

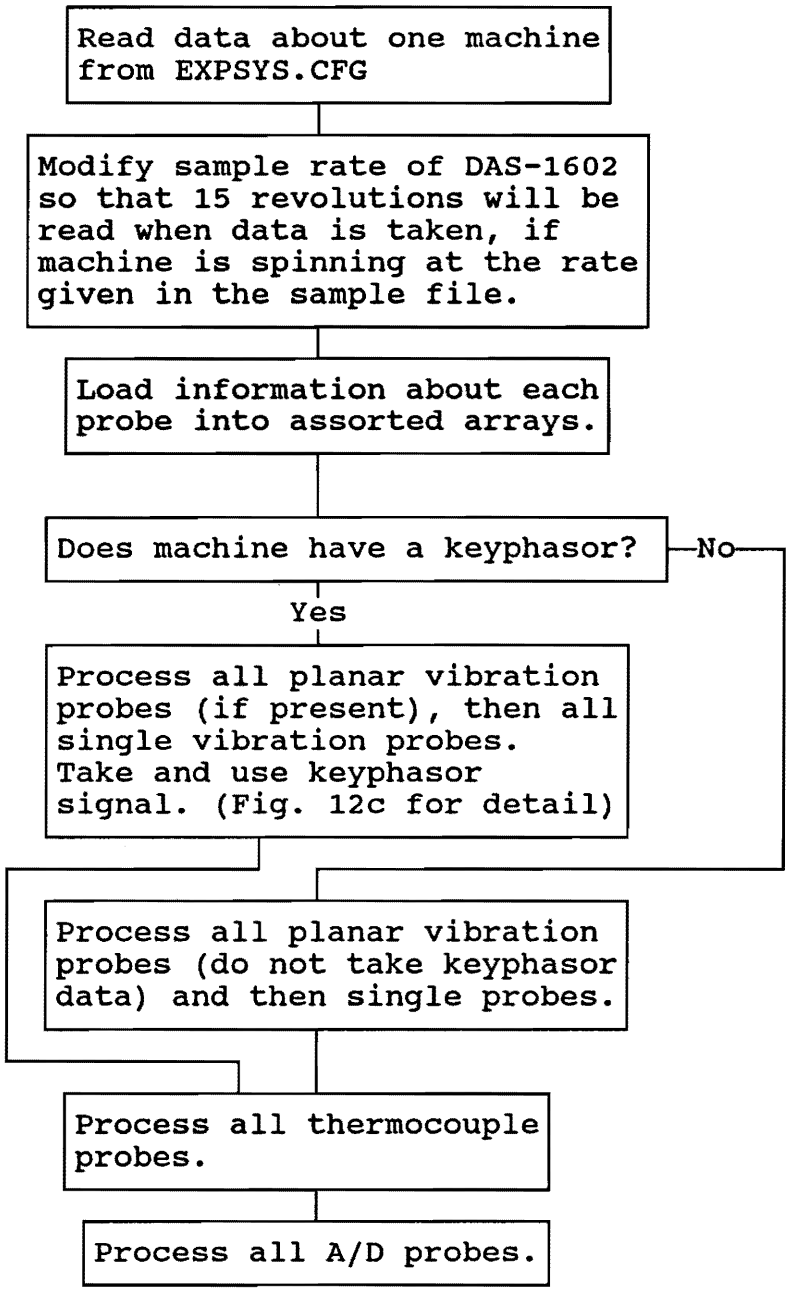


Figure 12b

IMTS Acquisition Module Processing One Machine

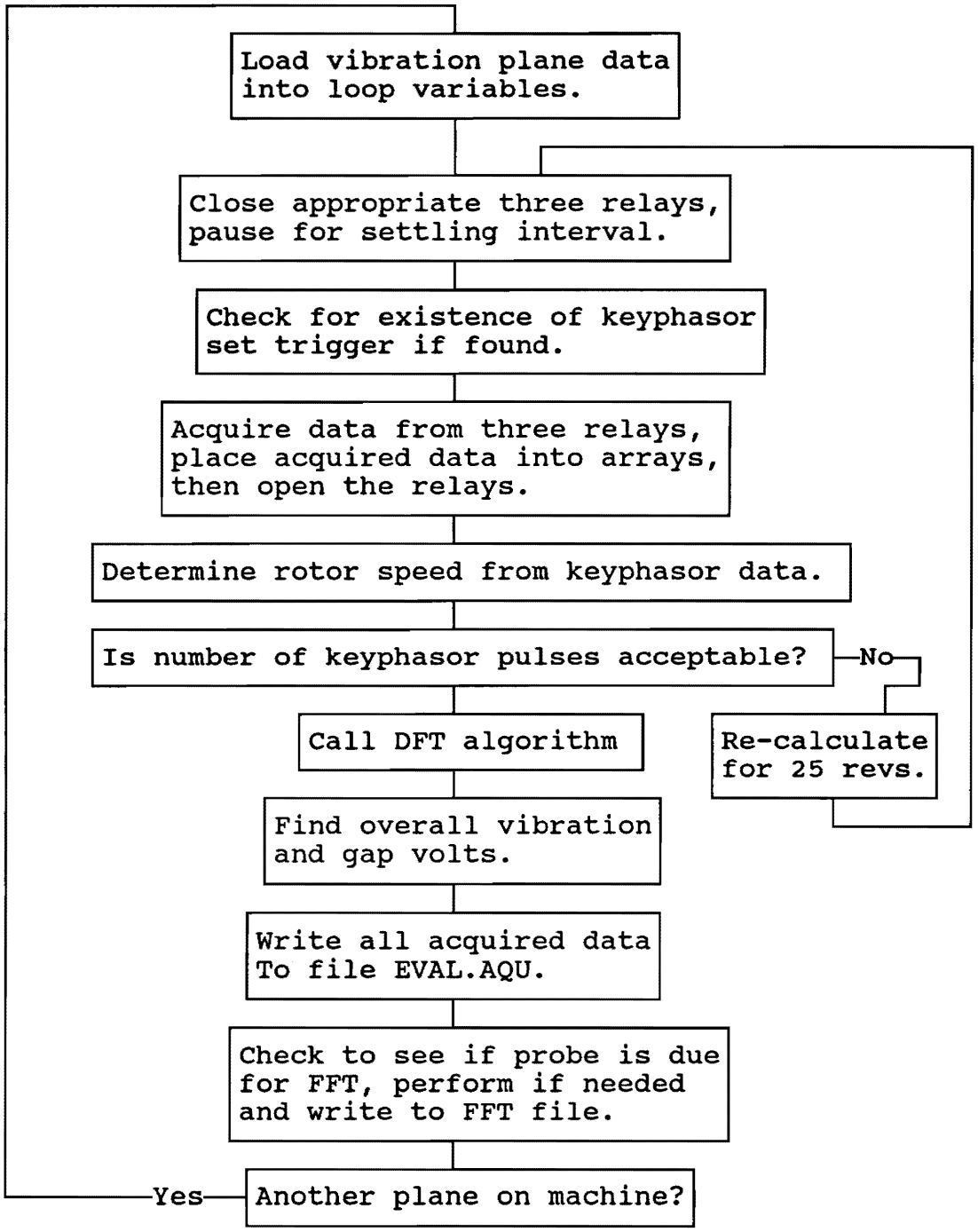


Figure 12c

IMTS Acquisition Module Processing Vibration Probe

configuration file will have the nominal speed that the machine is supposed to spin at. The program will alter the data acquisition rate of the DAS-1602 board so that, assuming the machine is spinning at this rate, there will be 25 revolutions of the shaft while the 1024 data points are gathered (within reason, the program will not accept RPM's slower than 200 or faster than 10,000). This number of revolutions was determined mostly by trial and error. It was noted that, in order to accurately determine the 1x and overall vibration levels, the program needed to sample between 10 and 80 revolutions of the shaft. Fifteen was chosen as a compromise number within that range.

The machine then processes each of the machine's probes in turn. First are the planar vibration probes, then the single vibration probes, then the thermocouple probes, then the A/D probes last of all. As each probe is finished, the data from it is written to EVAL.AQU. If a machine has none of any particular kind of probe, then a "0" is written to the output file for the line that corresponds to that kind of probe (it is possible, although pointless, to give the system a machine with no probes, in which case no data is taken and the output file will have four lines with a zero).

Figure 12c shows the details of the loop that handles a planar pair of vibration probes on a machine with a keyphasor. This is the most complicated vibration probe loop the program has. Single probes and probes on machines without keyphasors are handled by what amounts to the same loop with relevant elements deleted, so this is the only loop that will be gone over in detail in this document.

The loop starts by loading information about both probes in the plane from the array they were stored in to loop variables. There are eight pieces of information per vibration probe: The ID number of the board where the relay

that switches this probes data is on, the ID number of the relay on that board, the minimum interval between which data from this probe should be placed into the log file (in minutes), a string that identifies the type of probe involved (proximity probe, velocity probe, or accelerometer), a string that indicates if it is a normal vibration probe or a thrust probe, a real number multiplier that indicates the number of volts per vibration unit, a string for the name of the probe, and an integer that indicates what range the FFT spectrum should be.

The program then closes the three required relays (one for each probe and one for the keyphasor) then calls a subroutine that checks for the keyphasor. This routine collects data from the keyphasor channel and checks to be sure that a spike in the voltage has occurred. If so, then the DAS-1602 will be set to trigger data acquisition off of a pulse on the keyphasor signal (this ensures that the data is always acquired starting on the same part of the shaft's rotation).

If the keyphasor is not found, the program will treat the vibration probes as if they were on a machine that is not equipped with a keyphasor. This check is necessary because of the way this board works. If it is set to trigger off a pulse on a specific channel and that pulse never arrives, then the program will wait indefinitely for it. There is no way to insert a timer that makes it break out after a certain interval. Thus, this check was added to make sure that if a keyphasor probe breaks, the IMTS will not lock indefinitely (until someone notices and shuts it off). There is a small fraction of a second between this check and the start of data acquisition. If the correct keyphasor breaks during this interval, then the program will lock. This possibility was considered an acceptable risk, however.

For machines that have a keyphasor, the next step is to determine the rotor speed (for machines with no keyphasor or a malfunctioning one, the machine is assumed to be spinning at the speed given in the configuration file, if the machine is a fixed-speed machine). To do this, the program first finds the maximum and minimum values in the keyphasor data and finds the average and median. It sets a "crossover" point at slightly below the median and counts the number of times that the data crossed this point, at at what points in the data stream those crossovers happened. The time between crossovers is found (since the data acquisition interval is known), which enables the program to determine the time of each revolution that was sampled. Originally, the program simply averaged these times determine the overall rotor speed while data was being collected. However, a problem was found. Occasionally, (in about one out of every twenty data samples) one of the keyphasor data pulses would be missing from the acquired data. Much time was spent trying to find the reason for this, but to no avail. When this happened, the program would think that the machine had been reduced to 1/2 its correct speed for one revolution and average that with the other speeds, resulting in the overall estimate of rotor speed to be too low, and completely ruining the estimates of speed and 1x vibration for that particular data cycle. The problem was eventually corrected for in the following manner: the program computes the standard deviation of the intervals of each revolution. If the standard deviation is greater than 500, the program drops the longest interval from it's calculations and re-calculates. This was found to completely eliminate the problem. If, in the end, rotor speed is calculated to be less than 200 rpm, the program sets it to be 199 rpm (the evaluation and display modules know that rpm's less than 200 are to be treated as error conditions).

Next is the routine that checks how many times a keyphasor pulse was detected in the data. If this number is fewer than 15 or greater than 50, it re-adjusts the DAS-1602's sample rate and re-acquires the data (The routine cannot set acquisition speed to be faster than 80 clock ticks between sample rates, since it was found that the DAS-1602 starts to becomes inaccurate if there are fewer than 50, which on a MHz machine corresponds to a sample rate of KHz. This is consistent with the bounds claimed by the manufacturer).

Next, the program determines the 1x component of the vibration signal in each of the probes by calling the subroutine DFT (Direct Fourier Transform) for each channel. This routine uses a DFT technique to find the component of the vibration in a narrow range around the rotor speed (either the speed determined by the program or the speed in the configuration file, whichever is available). A DFT was used rather than a Fast Fourier Transform (FFT) to increase the program's speed. While a FFT technique can perform a complete spectral analysis of a vibration signal considerably faster than repeatedly applying separate DFT filters from zero to high frequencies, in this case the program does not want a complete analysis. Only a narrow band of frequencies in the middle of the potential frequency range needs to be checked. It was found that the computer took much longer to perform a complete FFT spectrum than to perform a DFT algorithm a small number of times.

Figure 13 contains the lines of code from the program that performs the DFT. This is an exceedingly simple algorithm for a DFT. This algorithm first determines if the DFT filter will be in the correct position by using the following equations:

```
wrun = 2 * pi * (rpm / 60)
wo = (2 * pi) / (Nsamples% * tstep)
LineNum = INT(wrun / wo)
Divide = wrun / wo
Lfactor = Divide - LineNum
IF Lfactor > .3 THEN
   wo = wrun / LineNum
   Ntstep = (2* pi) / (wo * Nsamples%)
   Resamp = 1
ELSE
   Resamp = 0
END
If Resamp = 1 then END SUB
IF LineNum > 2 THEN
   startline = LineNum - 1
ELSE
   startline = 1
END IF
stopline = LineNum + 3
  FOR ifreq = startline TO stopline
       gfre(ifreq) = 0
       gfim(ifreq) = 0
       I% = 0
       cossim = COS(0)
       sinsim = SIN(0)
       cosdelta = COS(ifreq * wo * tstep)
       sindelta = SIN(ifreq * wo * tstep)
       FOR I% = FirstCross% TO LastCross%
        cs = cossim * cosdelta - sinsim * sindelta
        ss = sinsim * cosdelta + cossim * sindelta
        cossim = cs
        sinsim = ss
        gfre(ifreq) = gfre(ifreq) + waveform(I%) * cossim * 2
           / (LastCross% - FirstCross%)
        gfim(ifreq) = gfim(ifreq) - waveform(I%) * sinsim * 2
           / (LastCross% - FirstCross%)
       NEXT I%
       total(N%) = SQR(gfim(ifreq) * gfim(ifreq) + gfre(ifreq)
          * gfre(ifreq)) / (SFC% / 10)
       Phase(N%) = -1 * ATN(gfim(ifreq) / gfre(ifreq))
          N% = N% + 1
  NEXT ifreq
```

Figure 13

Acquisition Module DFT Code

```
wrun = 2*pi*rpm/60
wo = (2*pi)/(Nsamples*tstep)
LineNum = INT(wrun/wo)

where: wrun = running speed in radians/sec
       rpm = running speed in revs/minute
       wo = width of the DFT filters
       Nsamples = number of samples taken
       tstep = time delay between each sample
```

The variable "LineNum" represents which spectral line the machine's running speed is closest to, which is which filter of the DFT should be used. Of course, there is still the possibility that the running speed will fall towards the edge of the DFT filter, since the filter has a very narrow width. Thus, the program checks to see how close the centerpoint of the spectral line falls to the acquired running speed by subtracting the integer portion of the ratio of wrun/wo from the ratio itself, and ordering the program to adjust the sample speed slightly and resample if this number is too high. (Sample speed for each machine is saved in program variables, so resampling is usually only necessary the first time the program is run).

Once the program has convinced itself that the data is prepared for a DFT, the DFT algorithm is run, as shown in Fig. 13. Three filters are run- the one that is expected to hold running speed, and the filters immediately above and below (this is done in case some error causes the program to miss the proper placement of the filter that is supposed to catch the running speed signal. With the two side filters in place, there is at least some chance of the program catching the correct vibration signal). After the algorithm is finished, the arrays Total(N) and Phase(N) (where N is the counter for

the three filters) contain the 1x vibration and phase angle. The program takes the largest of the three values of N and assumes that is the 1x magnitude, and the corresponding Phase(N) is taken as the 1x phase angle. If the phase angle is less than zero or greater than 360, the program sets it to be -999 degrees, which the evaluation and display modules interpret as an error condition.

As shown in Fig. 12c, after the DFT is run, the overall vibration level and and gap volts are found by simply taking the maximum and average values of the data and applying multipliers from the configuration file to convert them into vibration levels and volts, respectively.

Once the DFT is run, the program writes the acquired data to the output file EVAL.AQU, performs an FFT one of the two probe (if required), then moves onto the next pair of planar vibration probes, if the machine is so equipped. If none are available, the program moves onto the next form of probe, as shown in Fig. 12b.

3.6 IMTS Evaluation module

The purpose of the evaluation module, which is written in the Prolog language, is to read the raw data files created by the acquisition module, judge the data against the standards (much as a human operator might do), and relay the results of these judgements to the display module. In addition, this module is responsible for resetting standards when indicated by the user. It does not have a display of any kind, as it is meant to be run in the background or in a "minimized" state under Windows.

The evaluation module is capable of analyzing vibration data, temperature data, flow rate data, and any other miscellaneous data collected and converted to digital form. This task is complicated by the fact that the form of data and

the types of evaluations to be performed may vary widely from one installation to another. The evaluation module is designed to be versatile enough to handle a multitude of different installations without requiring extensive re-coding.

The evaluation module initializes itself by reading a database, ODD.SET, into a Prolog internal database. It then begins reading and evaluating the data file, EVAL.PLG, that was written by the acquisition module. The results of the evaluations, as well as the data itself, is written to the file EVAL.EVL as they are processed. When a complete set of data has been evaluated, this file is renamed EVAL.DSP and read by the display module, which displays it to the screen. This procedure is outlined in Fig. 5.

As is the case with any database-intensive program, its form is completely dependant on the format used by the database file, which in turn is dependent on the types of data being gathered. The format of the data file is discussed in more detail in Appendix B.

The operation of the evaluation program is shown in Fig. 14a-c. This program does not require any input from the operator, and runs completely autonomously. Thus, it has no menus, key presses, or other steps that require human intervention.

When activated, it first clears all internal databases and searches for a data file from the acquisition program. Then a "repeat" statement and a small predicate named "check_key" are used. The repeat will cause the program to cycle indefinitely. The "check_key" predicate is used for braking out of the loop and terminating the program. It monitors the keyboard buffer, terminating the program if the

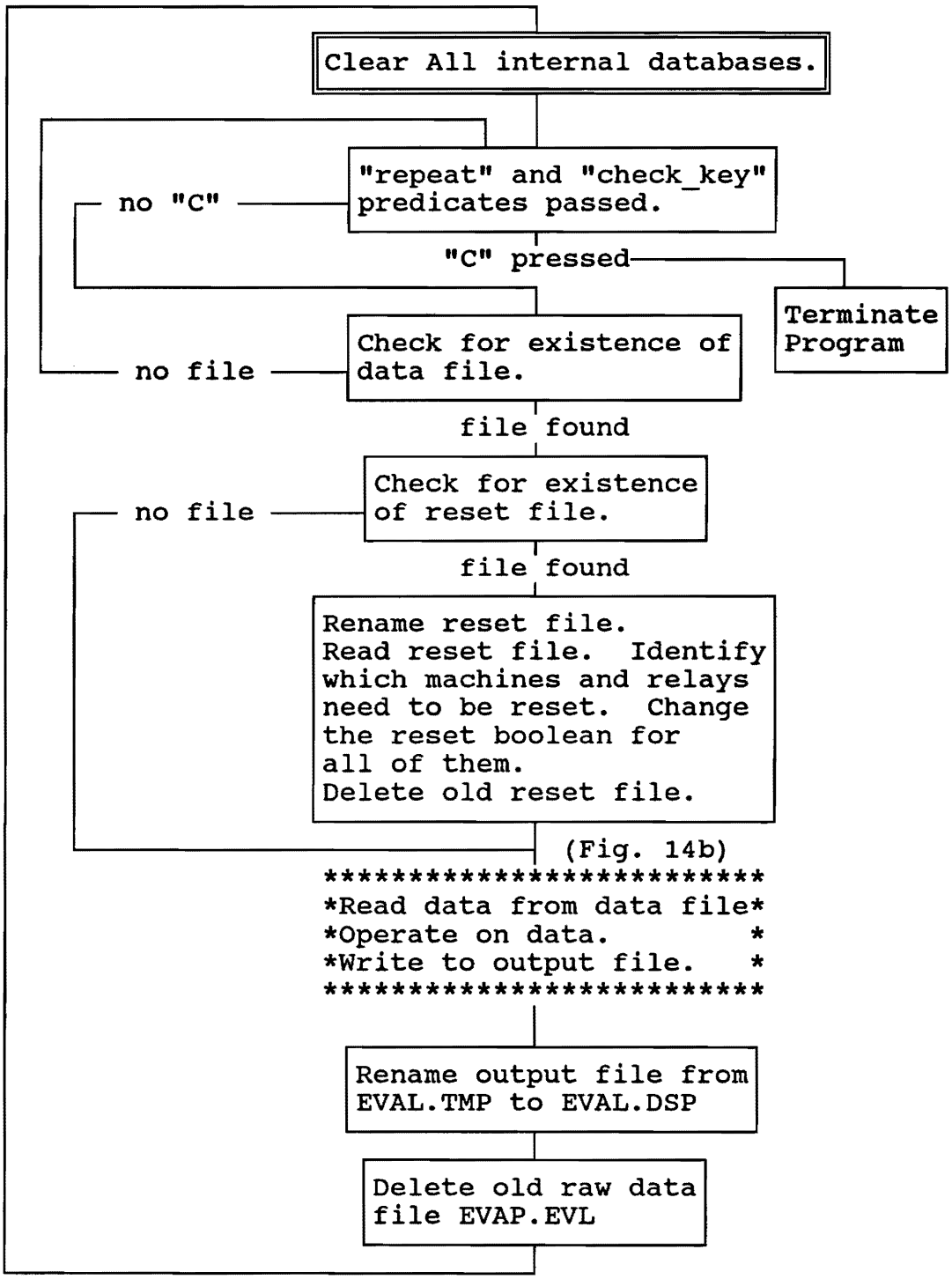


Figure 14a

Operation of the IMTS Evaluation Module

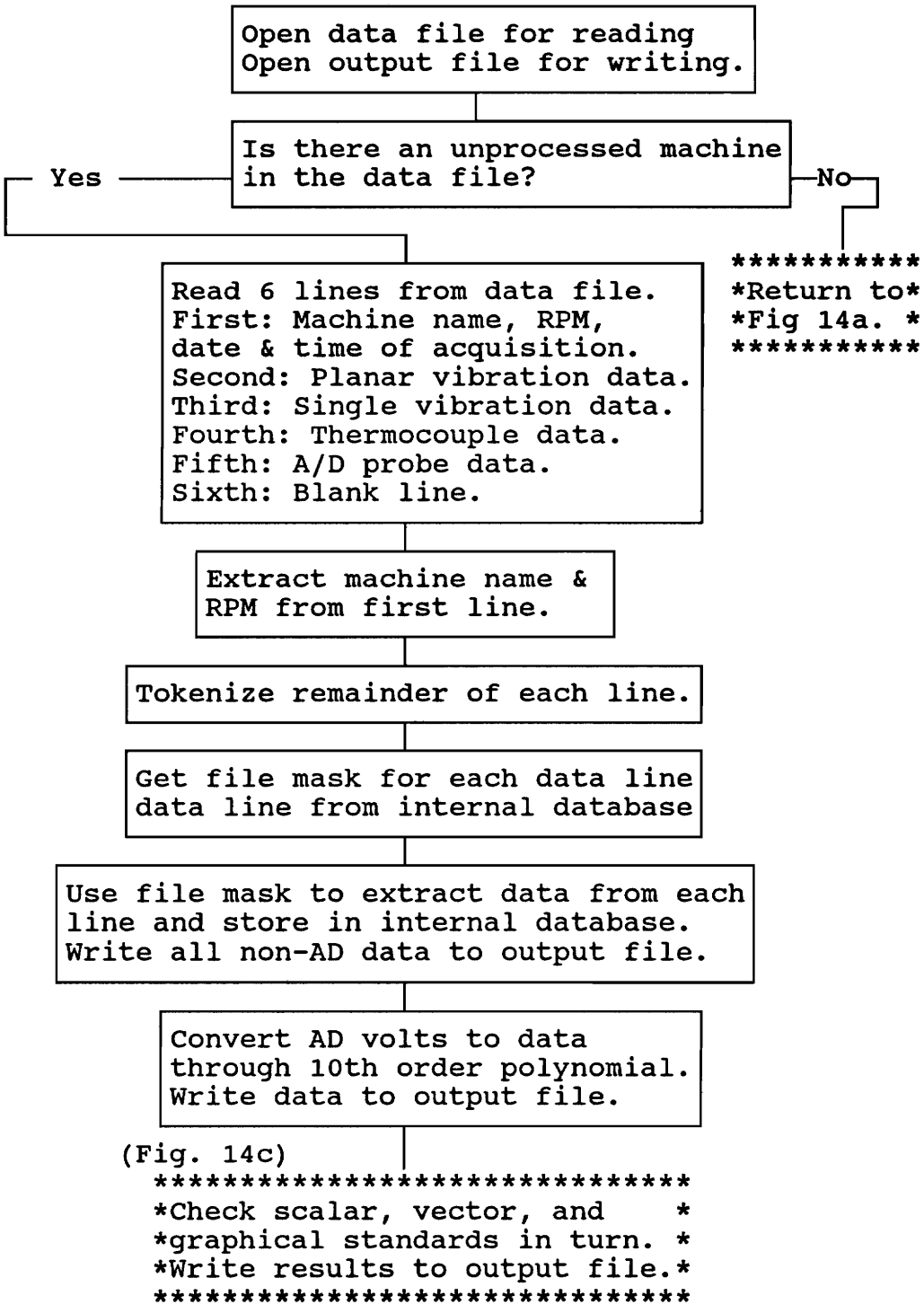


Figure 14b

IMTS Evaluation Module Reading and Operating on Data

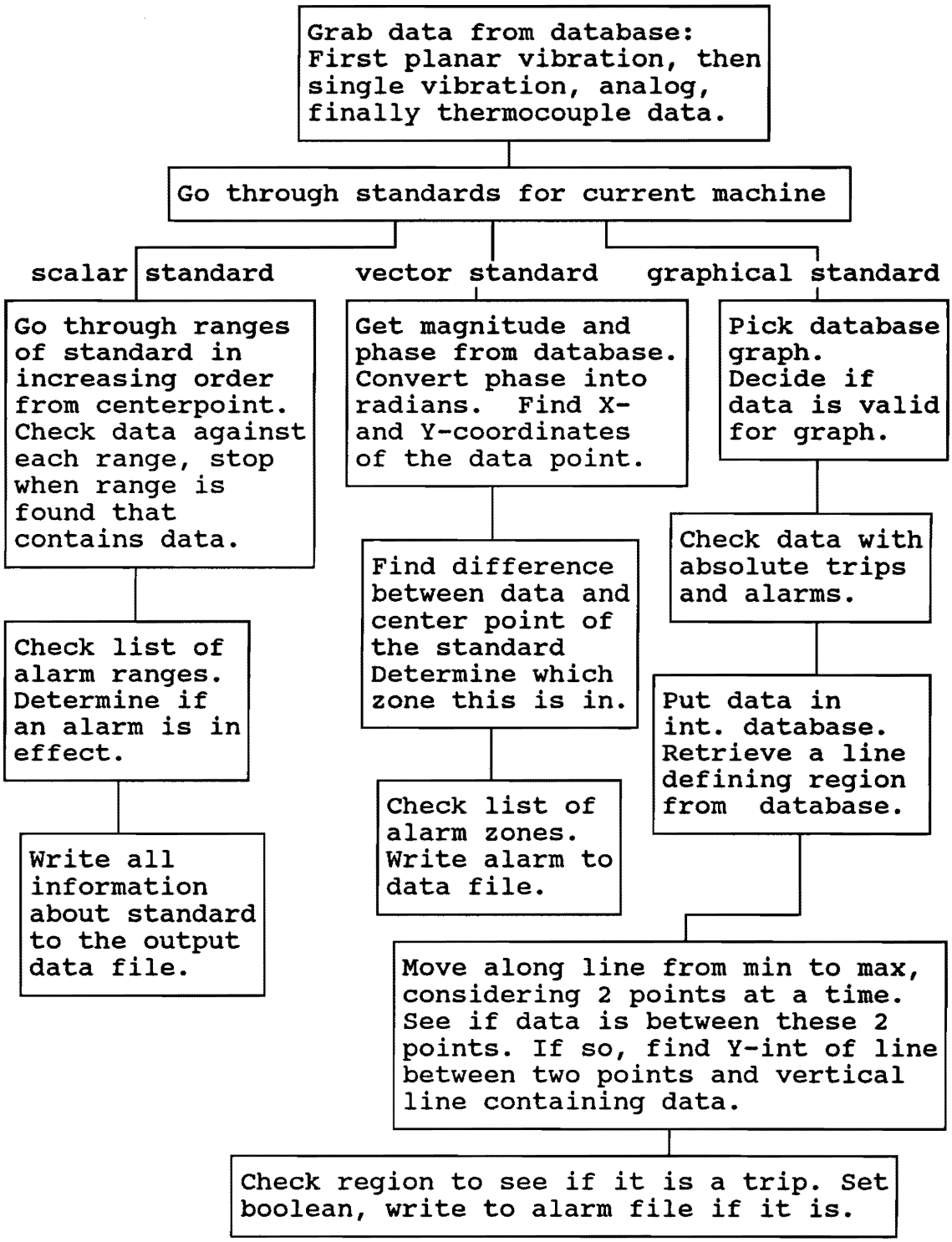


Figure 14c
Comparing Data to Assorted Standards

letter "C" is input while the window containing the evaluation program is the active window. This is the intended way to shut down the program (You can, of course, turn the computer off or reboot it).

Once a data file is found, the program looks for a "reset" file. Reset files contain the names of all probes and machines whose resetable standards need to be reset. A sample reset file is shown in Fig. 15, which contains the names of three machines with a total of ten probes between them that need to be reset.

The program will read the names of each of these machines and their associated probes and alter a boolean in the internal database predicates that store them. When the data for those particular probes are read from the raw data file, all resetable standards applied to them will be reset to the values in the raw data file.

After any reset files have been found and processed, the program proceeds to operate on the raw data file. As each machine's data is read, it is compared to the assorted standards and the evaluations are written to a temporary file, EVAL.TMP. After the last evaluation is made, this file is closed and the file is renamed to EVAL.DSP. This is the filename that the display program is searching for. The old raw data file is then deleted and the program begins to search for a new one.

The process by which the raw data file is read and evaluations performed is outlined in Fig. 14b-c. These figures outline the predicate "read_data_from_file", which contains all portions of the program responsible for reading and using the data from the input file. This predicate opens the appropriate files, and calls "read_dataset". This

```
01-TEST-GEN
OUTV,OUTH,INV,INH
01-TEST-THRUST
A-V,B-H
01-TEST-TURB
EXHV,EXHH,STMV,STMH
```

Figure 15

Sample Reset File for IMTS Acquisition Module

predicate begins reading the data for the first machine in the file line by line.

For any given machine, there will be six lines in the input file. A sample of the file is shown in Fig. 16. The first line contains the name of the machine who's data will follow, and then the time and date that the data was acquired by the acquisition module. After this is the machine's speed in RPM if the machine is equipped with a keyphasor (or the machine's fixed speed if not equipped if it is a fixed-speed machine with no keyphasor). Following this are four lines of data and a line that serves as a spacer between this machine and the next. The first two blank lines are for data acquired from vibration probes. The first line contains data from probes that have been paired off because they occupy a single X- and Y- plane. The second contains data from probes that are not paired into planes (it was decided early in the project to group planar vibration probes differently from single probes, in case it was later decided to give the program the ability to plot orbital paths of the shaft). For both lines, each point's data is listed in the order that the probes are named in the setup file. There can be up to four pieces of information from any probe: overall vibration, gap volts, 1x vibration, and the phase angle of the 1x vibration. For example, in Fig. 16, for the first data point, these values are 1.2 mils, -9.5 volts, 0.45 mils, and 345 degrees. The next probe has values of 1.3, -9.96, 0.50, and 350. Together, these probes make up a single X-Y vibration plane. There are no other planes in this data file. The next line contains only a zero, indicating that there are no non-planar vibration probes on this machine. If there were, their format would be identical to the previously discussed line.

```
01-GEN-TEST, 12:05:45,02-04-94, 3601
1.2, -9.5, 0.45, 345, 1.3, -9.96, 0.50, 350,
0
9.8
235.56, 297.06
0
```

Figure 16

Sample Input File for IMTS Evaluation Module

The fourth line contains one number, 9.8, which in this case is a voltage proportional to the generator's output in megawatts. This is the only piece of general A/D data being acquired from this particular machine. The final line contains two numbers that represent temperatures acquired from a pair of thermocouple probes. The final line, the spacer, always contains a zero unless this is the last machine in the data file, in which case the line is blank.

This file contains the name of the machine that the data are from. The identity of each individual probe is not defined in the set-up file. Rather, the file contains the names of each probe and the order in which each probe's data will appear in the data file. The program knows which data goes with which probe by the order in which the data appears in the file.

Once the data has been read and stored in internal databases, all but the analog data is written to the output file. At this stage, any analog data is still in the form of a raw voltage. The program uses the predicate "volts_to_variables" to convert the voltage into a measured quantity. This is done by a polynomial of up to 10th order whose terms are stored in the setup file. An upper limit of 10th order was chosen because this is the maximum length of polynomial needed to convert any of the common types of thermocouples into a temperature. The converted data is then written to the output file. If no polynomial is found in the setup file, the raw voltage is written in its place.

Data is then compared to the assorted standards (outlined in Fig. 14c). The program identifies which machine's data has been read by comparing the string containing the machine name to the internal database and determing the machine's ID number. Then the list of standards that have that ID number is checked. The various types of data are compared to the

standards in the following order: planar vibration data, single-probe vibration data, analog data, and thermocouple data. Each type is compared first against any fixed scalar standards, then deviation scalars, then vector standards (for vibration data only) and finally graphical standards.

Comparison to scalar standards is quite straightforward. As each scalar is pulled from the database, its ranges are considered in order from innermost to outermost. When a region is found that contains the data, the comparisons are stopped. Then the list of alarm regions is considered- if the found region is in the list of alarm ranges, then the standard is considered to be in an alarm condition.

Vector standards are slightly more complicated. The phase angle of the data is converted from degrees to radians by multiplication. Then it is converted into Cartesian coordinates by letting X = Magnitude * cos(phase angle) and Y = Magnitude * sin(phase). This same conversion is made for the centerpoint of the standard. The distance between the data point and standards centerpoint is found by taking square root of the sums of the squares of the differences between their X and Y components. This distance is compared against the radius of each of the vector standard's zones, starting with the innermost. When a radius is found that is less than this distance, the data is classified as being in that zone.

The last form of classification done is to graphical standards. Before any classification is done, all information about the graph is pulled from the database by the predicate "get_graph_info". The acquired data is checked by the predicate "test_limits" to see if it falls within the maximum and minimum bounds defined by the graph. (This step is not necessary for scalar and vector standards, since they both extend to infinity away from their centerpoint. On the other hand, a graph defined from 0 to 5000 R.P.M. is simply not

applicable if the rotor speed happens to be 6000 R.P.M.) If such an out-of-bounds data point is found, the data is written to output file with an evaluation of "invalid". Then the predicate "check_absolute_Htrip" compares the data on the graph's horizontal axis to any simple "horizontal axis standards" that have been created for this graph. These standards consist of a pair of real numbers, one a minimum, the other a maximum (this form of standard is functionally identical with a scalar standards. It exists primarily as a convenience for the operator, so that a scalar standards need not be created for a particular data point if a graphical standards already is being created). If the data violates one of these standards, a boolean is set to one.

After the simple horizontal trips are checked, the predicate "check_pt_condition" is used to find which region of the graph the data falls into. This is done by loading a list that contains each of the graph's points. Each point is a database predicate of the type pt(X,Y,LN) where X and Y are the coordinates of the line and LN is a string that names which line the point belongs to. The predicate loads the points two at a time, then passed them and the data on to the predicate "test_pt_and_lines". If the data point lies between these two points, the predicate locates the intersection of a vertical line through the data point and the line connecting the two graph points with the following simple interpolation:

```
ShowYint = ((Y2-Y1)*(ShowXco - X1)/(X2 - X1)) + Y1

where:ShowYint = Y-coordinate of the intersection
      X1,Y1 = coordinates of the leftmost point on the line
      X2,Y2 = coordinates of the rightmost point on the line
      ShowXco = X-coordinate of the data point
```

Then the relative height of the data point and the intersection point are found by:

Percent = 100*((ShowYco - Ymin)/(ShowYint - Ymin))

where:Percent = relative height of data to the graph line
Ymin = minimum Y-coordinate for the graph

The variable "Percent" will be greater than 100 if the data point is above the line, less than 100 if the data point is below the line. The predicate "find_region" checks the value of this variable. If it is less than 100 and the line currently being checked is lower than the last line that the data was found to be in (identified by the line's "condition number", which increases with the line's height) then the data is assumed to be in the region defined by this line. This information is asserted into the database predicate pointpos(Gnum,Linename,Cond) where Gnum is the graph's ID number, Linename is the name of the region, and Cond is the condition number of that line. Then the process is repeated with the next lower line, and finished when the last line is checked.

Once the region the data plots into is identified, the predicate check_trips takes the list of any alarm trips that have been set for this variable and checks them against the data. If the condition number of the region the data plots into is either equal to or lower than the condition number of the condition number of a minimum trip region, or greater than or equal to the number of a maximum trip region, a boolean is set that notes that an alarm trips has gone off. The final predicate output_point_condition writes the results of the trip comparisons to the output file.

For all the various kinds of standards, the end result of a trip is a line written to the output file, EVAL.DAT, a sample of which is shown in Fig. 17. This file is organized by machine: that is, it is divided into sections, each one of

```
01-GEN-TEST, 12:05:45,02-04-94, 3601
1.2, -9.5, 0.45, 345, 1.3, -9.96, 0.50, 350,
0
6.7
235.56, 297.06
0

P, 0, INV, 1x , Nominal, D, 0.01, 1x std
P, 0, INH, 1x , Nominal, D, -0.03, 1x std
P, 0, INV, Overall , Nominal, D, -0.098, overall vib. std.
P, 0, INH, Overall , Nominal, D, 0, overall vib. std.
P, 0, INH, Shaft Speed, Nominal, D, 1, speed
P, 0, INV, Overall , Nominal, F ,vib. spec #1
P, 0, INH, Overall , Nominal, F ,vib. spec #1
P, 0, INV, 1x , low, F ,vib. spec #2
P, 0, INH, 1x , low, F ,vib. spec #2
V, 0, IHV, Gen std #1, 1x, Phase, acceptable
V, 0, INH, Gen std #2, 1x, Phase, acceptable
G, 0, INV, Rotor Kit Standard, speed, Overall, acceptable
```

Figure 17

Sample Output File of IMTS Evaluation Module

which contains the data and evaluations for a given machine. Figure 17 contains only one machine, 01-GEN-TEST. The data appears at the beginning of each machine's section (the first six text lines in Fig. 17). The format of this data is identical to the format of the evaluation program's input file, EVAL.PLG. Indeed, in the event that the system is run with no evaluations for any of the data, EVAL.PLG and EVAL.DAT will be identical.

After the data is a blank line separator, then the lines that contain the results of evaluations against the standards. Each standard is on a sperate line. They appear in the same order that they were evaluated, since the program writes the results of the evaluations to the file as it makes them (this is done to conserve RAM. If the results of all standards were stored in memory and written all at once, there would be a limit to how many standards could be applied to a given set of data).

Thus, the first nine standard evaluations that appear in Fig. 17 are from scalar standards. An evaluated scalar standard has the following format: first is a letter, "P", "S", "A" or "T", that identifies if the standard applies to data from a planar vibration probe, a single vibration probe, an analog probe, or a thermocouple probe. (The machine 01-GEN-TEST was not equipped with any of the latter three kinds of probes). Next is a 1 or a 0, which indicates if this standard is in an alarm condition or not, then the name of the probe that was evaluated, the name of the data from that probe, the standard's region the data plotted too, and a letter, "D" or "F" that indicates if it is a fixed or deviation standard. If it is a deviation standard, the amount of the deviation follows the "D". Last is the name of the standard itself. In this particular file, there are five deviation standards (one for the 1x and overall vibration from

two probes, and one for shaft speed) followed by four fixed standards.

Vector standards are another possible format that the evaluation module can use. These appear in Fig. 17 immediately after the scalar standards. The format for these starts with a "V", identifying the standards as a vector standard. This is followed by a 1 or 0 to indicated the presence or absence of an alarm condition, the name of the relay where the magnitude of the vector came from, the name of the vector standard itself, the name of the data being used as the magnitude (this will usually be "1x", the name of the variable being used as the phase (usually this will simply be "phase", and the name of the area of the standard into which the data was plotted. Figure 17 has two vector standards, "Gen. std #1" and "Gen. std. #2".

The last form of standard that the system can consider is the graphical standards. Figure 17 contains the results of one graphical standard immediately after the vector standards. This particular evaluation is based on the "Rotor Kit Standard" shown in Fig. 1. The format of this line is similar to the others: a "G" to indicate that this is a graphical standard, a 0 or 1 to indicate the presence of absence of a trip, the name of the relay who's data is being compared, the name of the graphical standard, the name of the variables on the horizontal and vertical standards, and lastly the name of the region that the data plotted into.

Once the last form of standard has been compared and the results written to the output file, the read_dataset predicate is satisfied. The predicate is "machine oriented"- that is, it performs all necessary evaluations for a given machine before reading the data from the next one from the input file. After a given machine is finished, it checks the data file again. If anther set of six lines are present, it reads them

and the process starts over (as outlined in Fig. 14-c). When the last pieces of machine data are read from the input file, the predicate finishes for the last time. The predicate "read_data_from_file" closes the input file. At this point, nothing remains for the program to do except delete the old input file and rename the output file from EVAL.EVL to EVAL.DSP. The program then goes into a loop waiting for the next input file from the evaluation module to appear.

3.7 Graphical Standards Editor

As previously noted, most published standards for rotating machinery take the form of some parameter plotted against shaft speed. These plots typically contain lines or simple curves that divide the plot into different regions of acceptability. There are rarely more than three regions, and the lines tend to be straight, defined only by their intersections with the axes, which may be either linear or logarithmic. For machines that operate at a fixed speed, such as most power plant turbines and generator units, these sorts of standards can be easily simulated by scalar standards, since rotor speed is not a variable. However, the IMTS system needed to be able to handle variable-speed machines, which scalar standards could not judge effectively. There are many other types of evaluations that would require two-dimensional standards, such as comparing temperatures and flow rates to plant load. While it would be possible to create and edit two-dimensional graphical standards by means of tables of numbers, it would be difficult for any operator to visualize what was happening. Thus, a graphical editor had to be created for the system to use. In order to easily merge with the evaluation module, the IMTS program used to create and edit the graphs would have to be written in Prolog. Unfortunately, no suitable graphics package could be found for

Prolog. A package had to be written from scratch, using the BGI (Borland Graphics Interface) for basic graph functions. This package has proven quite useful in other parts of this project and also in other, more minor projects at the rotor lab.

The graphical standards editor, STAND.EXE, operates as shown in Fig. 18a-c. As shown in Fig. 18a, from the main menu the choices are to create a new standard, review or a previously created standard as a graph, or to review a standard in text mode. The text mode primarily exists in case the operator does not have monitor capable of dealing with VGA graphics, and is of little consequence. Figure 1 shows what a completed graph looks like, with the graphical standard menu below it. The operation of this menu is shown in Fig. 18c. This screen was originally created as part of the display program, when it was thought that the display program would be written in Prolog. Thus, it contains assorted capabilities that are of little actual use in the IMTS, although they have been left in. The cursor (a small yellow point) can be moved about slowly by the arrow keys or quickly with a combination shift-arrow key. The upper box on the lower left, marked "cursor", indicates current cursor position in terms of the units of the graph. The F1 key can be used to enter coordinates of a point which will be plotted on the graph, and the program will calculate which region of the graph the point lies under, as well as the percentage of the height the point is from the bottom of the graph to the current "active line". Which line is "active" is noted in the lowermost left-hand box. This line can be changed via the F2 key. This function uses the same routines that the evaluation module uses to calculate the position of the line.

Originally, it was intended that this graph would be used in the display module to plot points of data as they were

acquired. This plan was dropped, after it was realized that the final display program would have to be written in Pascal instead of Prolog. Thus, this screen has only one effective use, that is, to check the appearance of a graphical standard after it has been created.

Creating a graphical standard follows the procedure shown in Fig. 18a-c. First, the filename that will store the standard is requested. This data file uses standard ASCII text to store the graph and may contain many different standards. Then a series of questions are asked to define standard's parameters. These parameters include such information as the maximum and minimum value for each axis, whether the axes are logarithmic or linear, the axis labels, how the axis labels should appear, the graph title, and so forth. While the shown figure has only positive numbers on both axes, the program can work perfectly well with negative numbers. There is error checking when the axes are entered- if the axis labels would overlap each other (such as a horizontal axis going from 0 to 4000 rpm with a label at every 10 rpm) the program will notify the operator of the error and ask that new values be entered. The graph name is different than the graph title- the title is what appears on the top of the graph when it is shown, the name is what is used in the various menus to pick which graph is to be shown.

Once all the identifying graphical details are entered, the program jumps into graphical mode and the graph can be drawn. The regions of a standard are defined from lowest to highest. In the case of the example shown in Fig. 1, the first line to be created is the "runout" line, which is a straight line consisting only of two points at the beginning

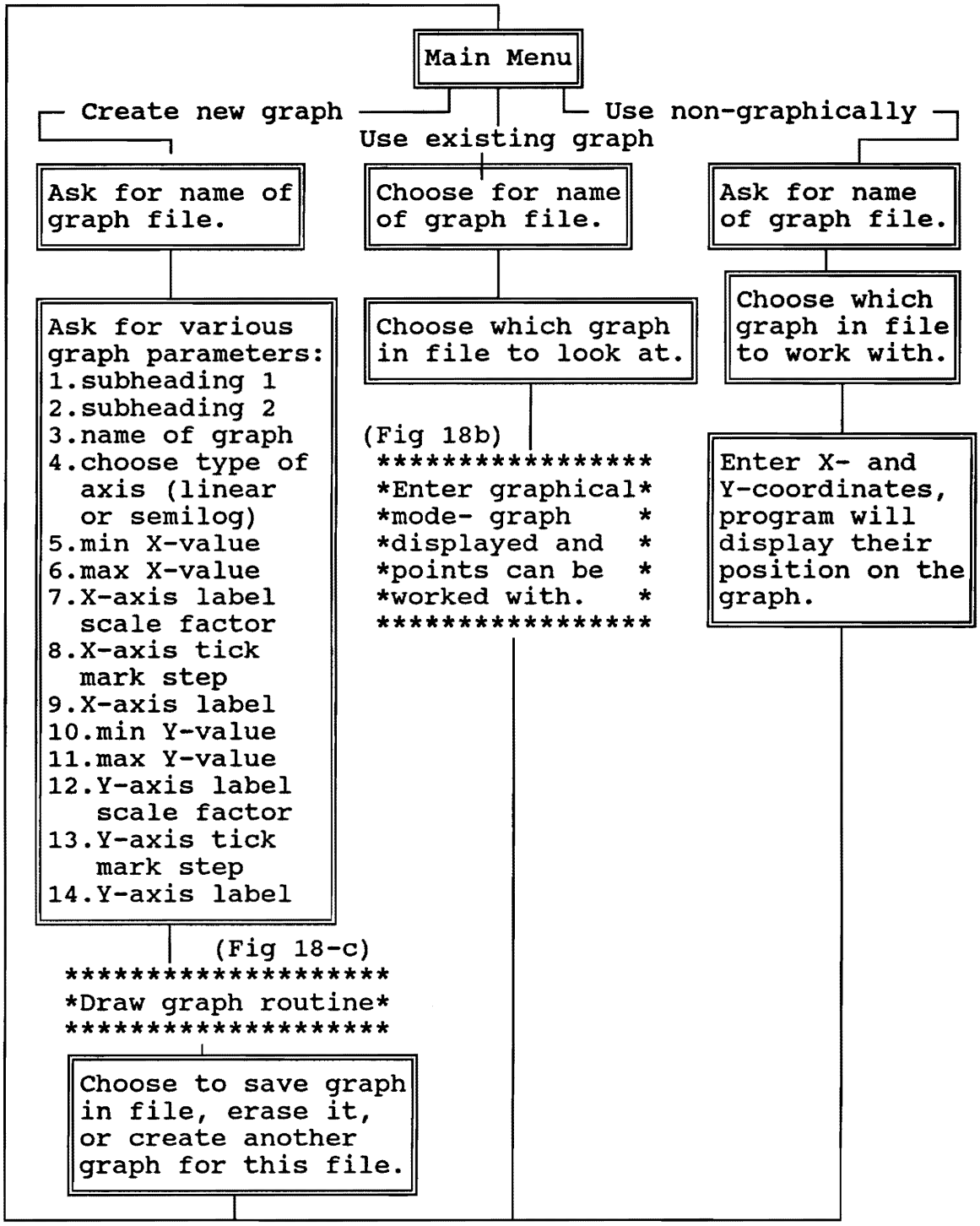


Figure 18a

Operation of Graphical Standards Editor

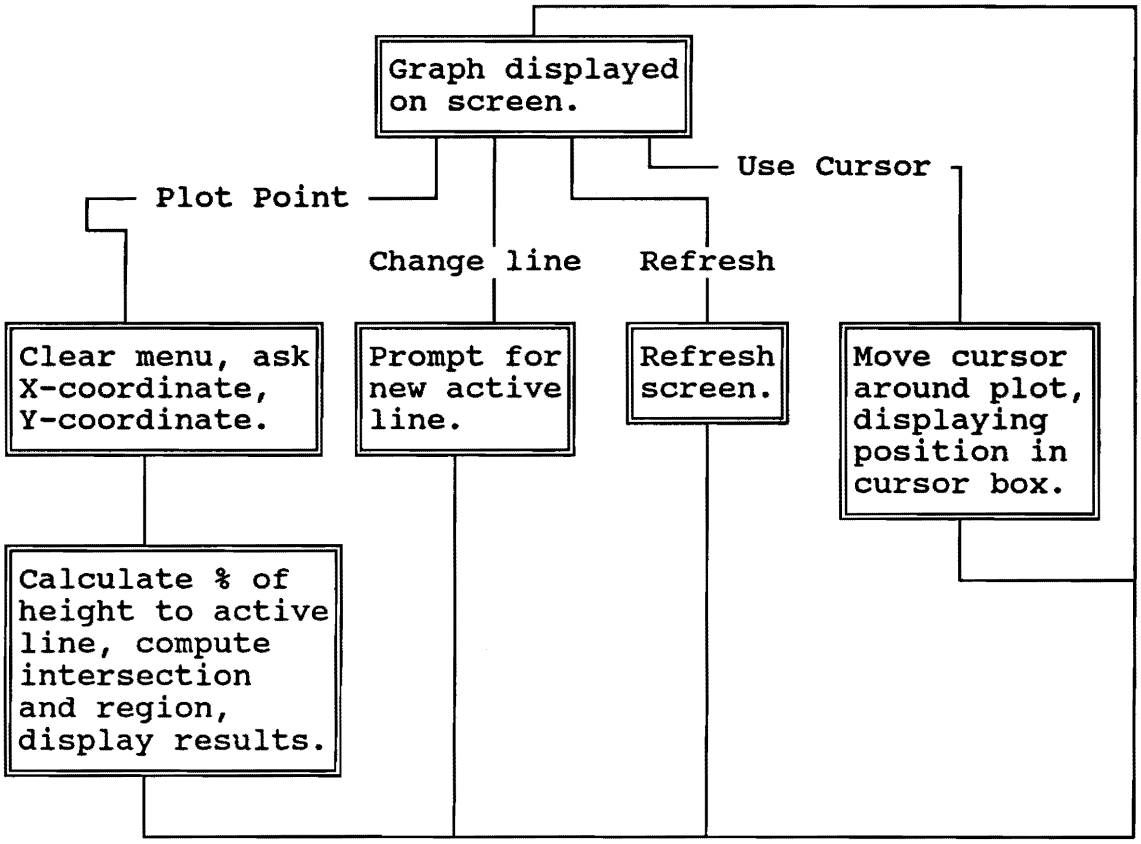


Figure 18b

Viewing a Graphical Standard

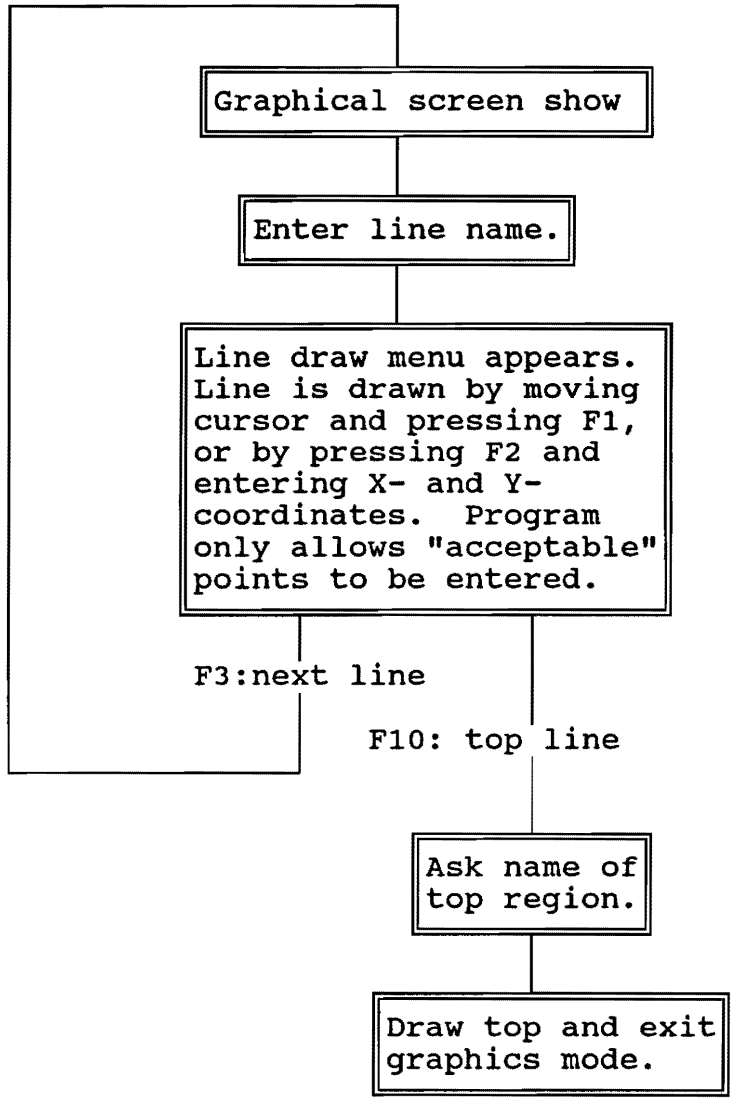


Figure 18c

Creating a Graphical Standard

and end of the range (0 and 5000 rpm). The next range is defined with a line labeled "acceptable", and contains five points, tracing a rough curve. The "marginal", "rough", and "alarm" regions were created in a similar manner. (This is an extremely simplistic graph for test purposes, points can actually be spaced as closely as desired.) There are two ways of drawing a line, which are outlined in Fig. 18c. The first is to move the cursor (a yellow dot) around the screen and press F1 when a desired point is reached. The box marked "cursor" on the lower right side of the screen will indicate the cursor's exact position. The other is to use the F2 function key and enter a point's X- and Y-coordinates. As each point is entered, the program will draw that point on the screen. The program checks every entered point before accepting it, by whichever method is used. Regions on the graph may not cross. When creating Fig. 1, for example, the line defining the "marginal" line could not be drawn such that it crossed the "acceptable" line. Any such points would be rejected and the program would ask that a different point be selected. Also, for any given line, each point must have a greater X-coordinate than the point that preceded it.

Once the last point on the last line is entered, the F10 key is used to finish the graph. This allows the "top" region of the graph (the portion between the topmost line and infinity) to be labeled. In this case, that region is "trip," and is drawn in as a line at the absolute top of the screen. Once the graph is finished, it can be stored on disk and other graphs created in the same graph file.

4.0 The Combined WOT/IMTS System

The on-line version of the WOT expert system is essentially the same program as the off-line version, run under Windows at the same time as the IMTS. The operator has

a choice of operating the expert system in either off-line or on-line mode. When operated in off-line mode, it is exactly the same as the previous off-line only version. When run in on-line mode, the expert system seeks out a "link file." This file contains a number of "links" that connect conditions in the database to data stored in the IMTS log files. Many different link files can be created for a single database. This allows the expert system to handle many machines individually without requiring numerous customized databases. For instance, the database on "compressors" can be left fairly generic, and a separate link file used for each compressor that the on-line system is to evaluate. If someone devises a new rule for the expert system, it only has be added to the one database, rather than all of them. The only things that needs to be changed are the link files for whichever compressors the new rule applies to. The link files will be explained in the following section.

4.1 The Linking Program

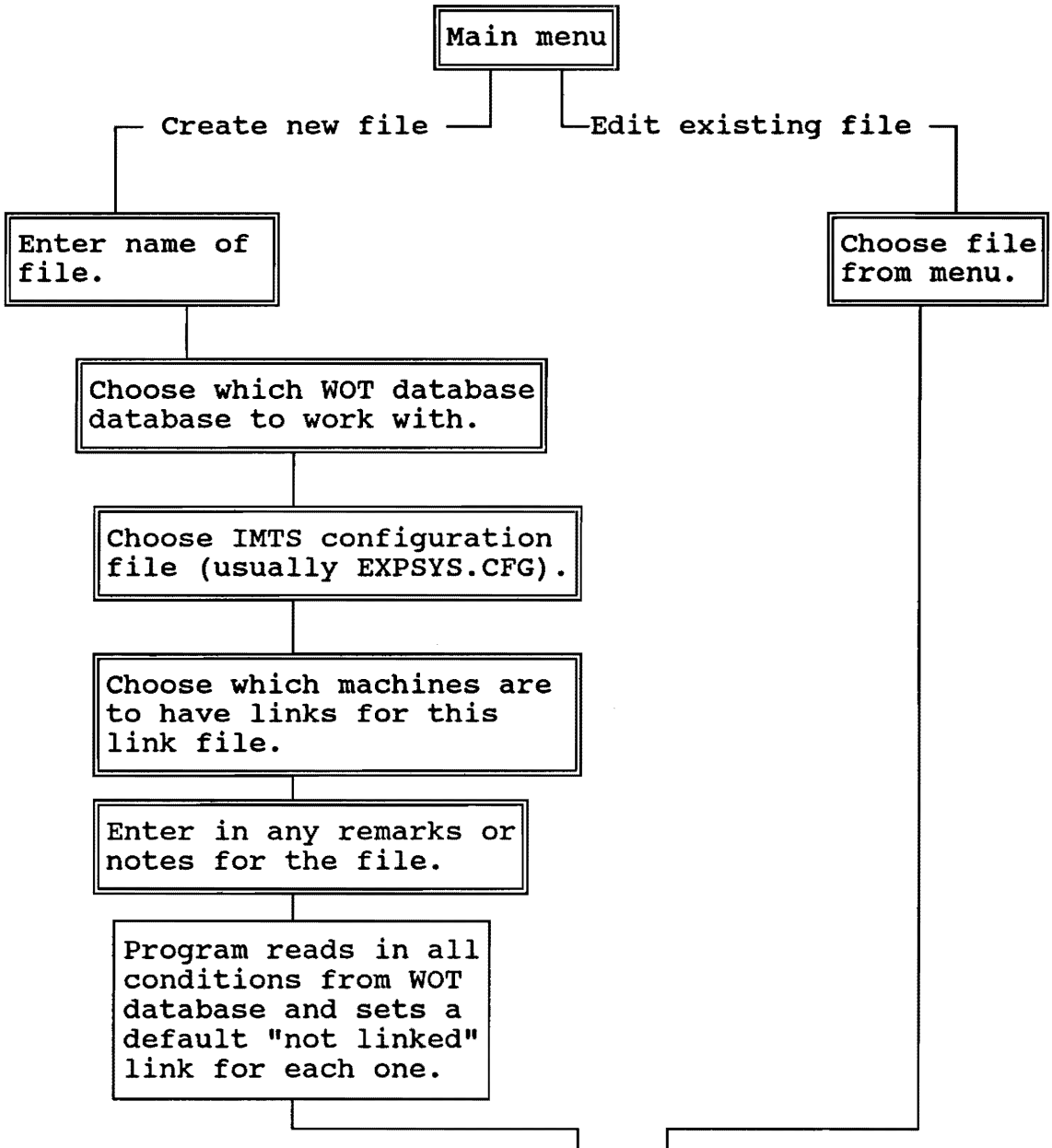
The linking program, WOTSETUP.EXE, is a DOS-based Prolog program that is used to create and modify the "datalink" files that are used to allow the WOT expert system to interpret the data files where the IMTS stores its information. The intent is that a separate datalink file can be created for every individual machine or model of machine that the system is used on.

There are two forms of links- the simplest and almost trivial kind are the "constant" links. These are used to enable a condition in the database to always be answered the same way. This is useful for conditions that would never change for a given machine, like "does the machine have oil seals" or "is the machine a centrifical compressor." These do not require the use of the IMTS data files. The more

complicated kinds of links are those that are answered by IMTS data. A simple example would be the condition "is the vibration high?". A single condition can be linked to as many pieces of data as desired. The various links can be connected by the boolean operators "OR" and "AND".

A separate link would be set between this condition and the vibration levels of each probe on the machine. Each link would be connected to the others by a logical AND, so the condition would be answered "Yes" if the vibration level of any of the probes exceeded whatever limit had been set.

Figure 19a-b shows the operation of the program. The first step is either choose to create a new datalink file or choose which existing one to work with. When creating a new one, the first steps are to give it a DOS filename and choose which WOT expert system database it is to work with. The program will then read the configuration file (EXPSYS.CFG unless told otherwise) to learn the configuration of the machines and probes that the system is expected to work with. A list is composed of every machine present in the configuration file and the user can choose which of those are to be involved in the link file. A text editor is called up that allows any remarks or notes about this particular link to be entered in (typically ones that would give an operator more information about the machines this link file will operate with). These remarks can be edited later, if desired. The program then reads in every condition in the selected WOT database and creates a "default" link for it- the default link being a note in the linkfile that the condition exists, but that no link has been created for it. The program then enters the main edit menu, which is contained in the predicate "datalink_main_menu". If an existing link file is selected,



(Fig 19b)

```
**************************************
* Main edit menu for editing remarks*
* or datalinks.                      *
**************************************
```

Figure 19a

Operation of WOTSETUP Program

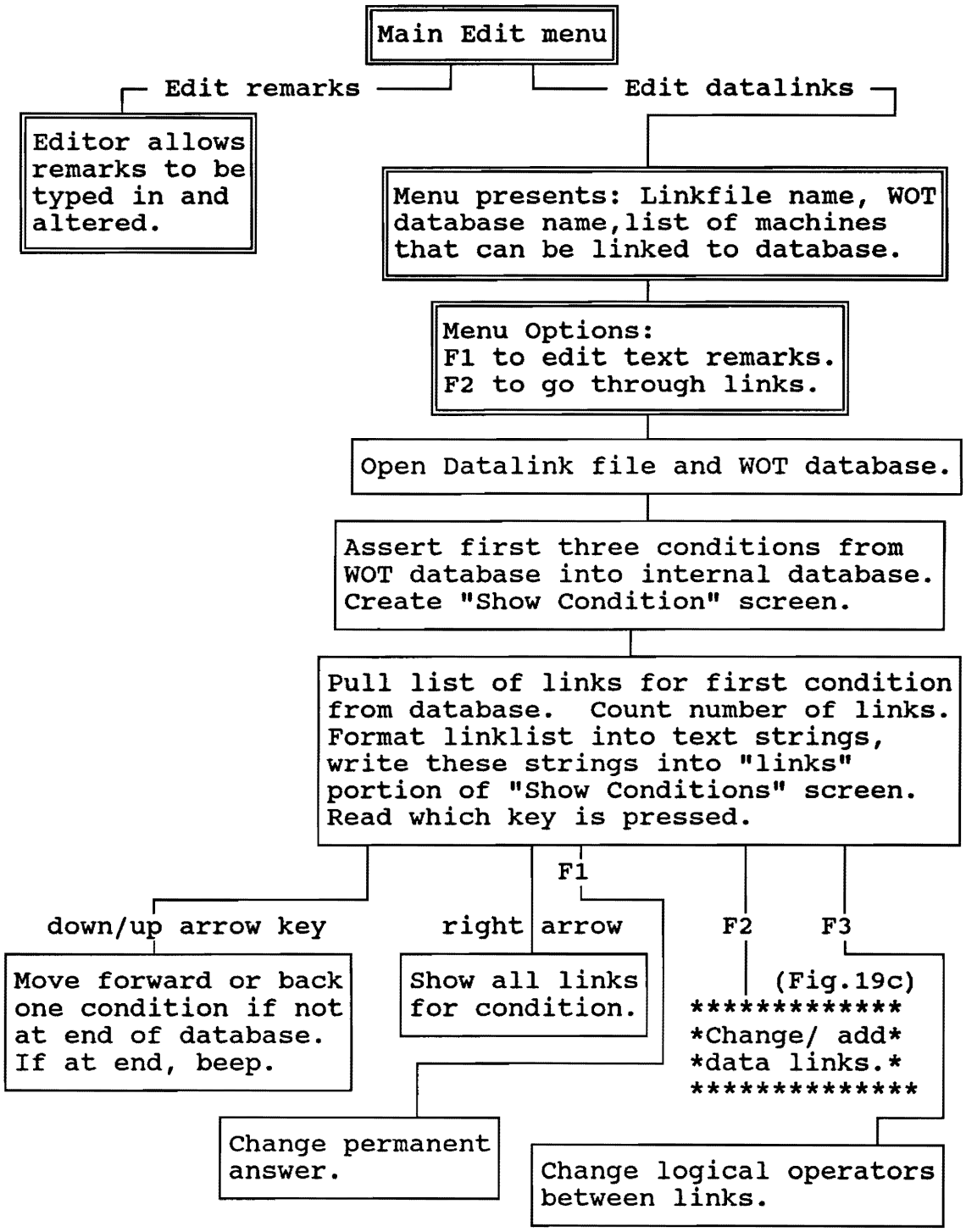


Figure 19b

Using WOTSETUP Program to Work with Conditions

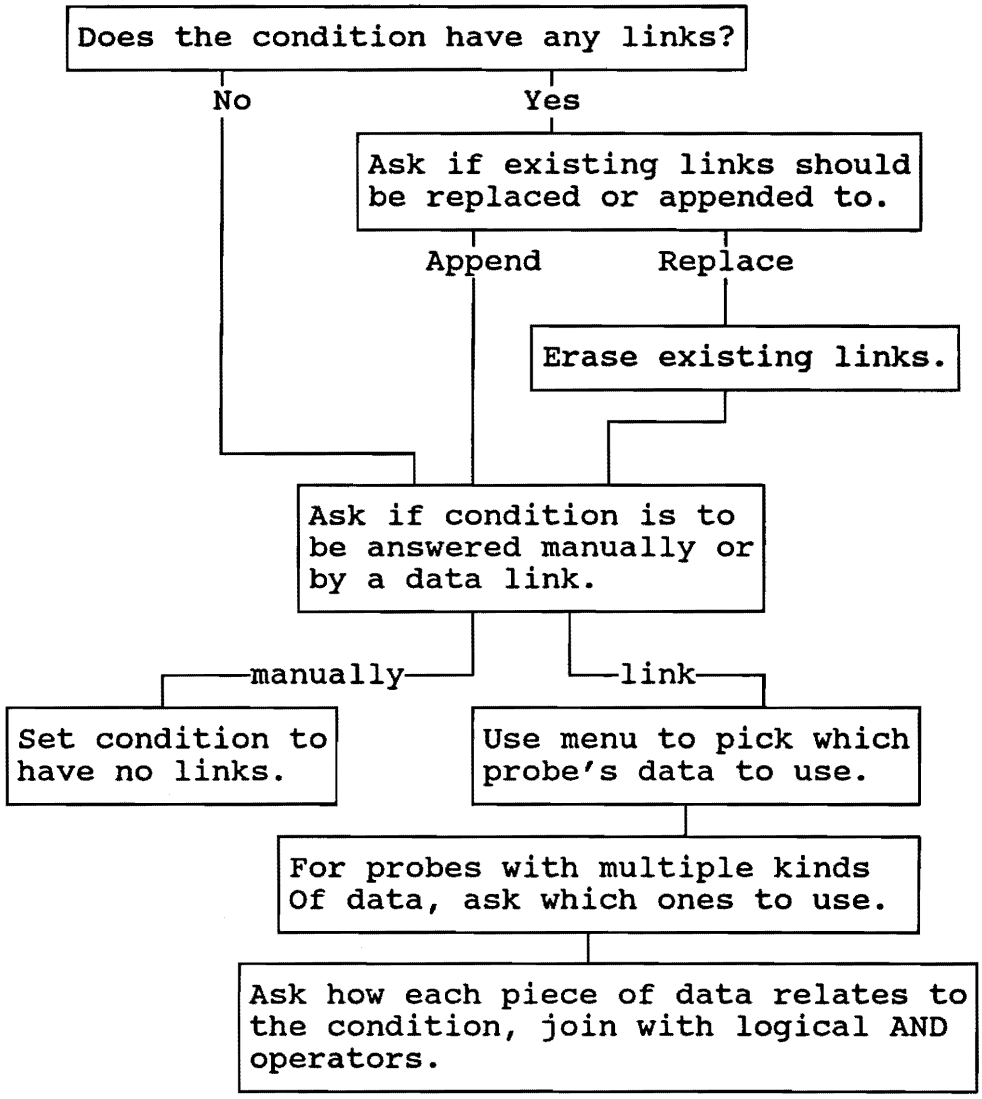


Figure 19c

Changing or adding links with the WOTSETUP Program

all this information is already known, so the program goes directly two the main edit menu. The possible operations from this menu is outlined in the first part of Fig. 19b. The main choices are either to edit the text remarks or to edit the datalinks themselves. If the second option is chosen, the WOT and Linkfile databases are opened, and the first few conditions from the WOT database asserted into an internal database. The next menu, the "Show Conditions" menu, then appears. A sample of this menu is shown in Fig. 20. This menu gives the names of the WOT database and the linkfile filename at the top. In this case, they are the turbogenerator database "turbgen.dba," and the turbogenerator link file, "turbgen.lnk." In a window below that is one condition, "The predominant vibration is at running speed," and a note about how many links this condition has (in this case, four).

Below the condition window is another window that lists the condition's links. If a condition has no links, the link window will read "Has no assignment in the link file. Please exit or scroll on to another condition." The link window can show up to two links and part of a third. More are not shown because there is not room on the screen for a window that can hold more than two- the portion of the third is shown as an additional visual cue that there are additional links. Below the link window is the commands menu. The up/down arrows can be used to "scroll" through the conditions in the database-- that is, change which condition is shown at the upper window and the links in the link window. If a condition has more than two links, the right arrow key will call up another window on top of the Show Conditions screen. This window is large enough to show up to six conditions (although it covers

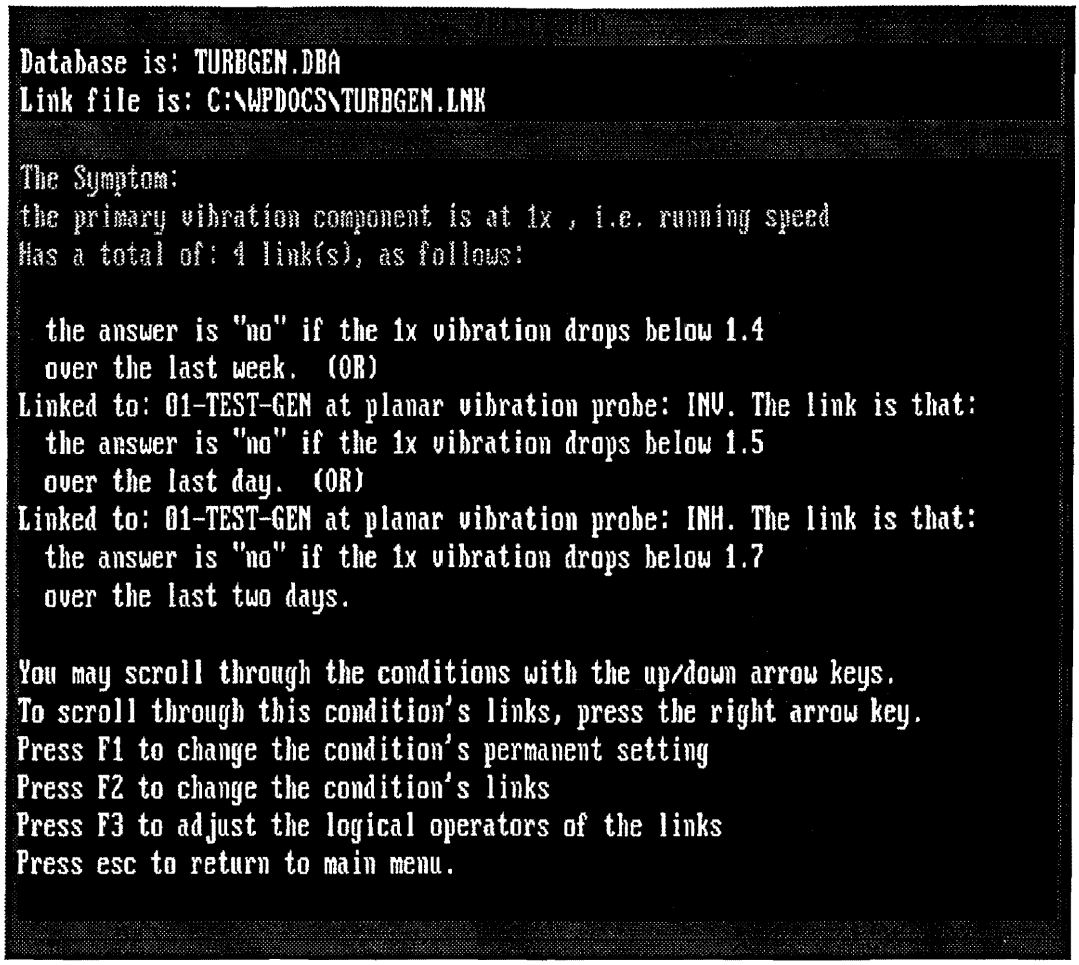


Figure 20

Sample Show Conditions Menu from WOTSETUP program

up most of the rest of the screen) and can be scrolled through with the up/down arrows if there are more than six.

The condition shown in Fig. 20 has four links: The 1x vibration at probe STMV drops below 1.8, the 1x vibration at probe STMH drops below 1.0 over the last two days, the 1x vibration at probe OUTV drops below 1.3 over the last day, and the 1x vibration at probe OUTH drops below 0.9, over the last week (the first two conditions are not shown in Fig. 20). These links are all connected by the logical OR operator. Multiple links can be linked by this operator or a logical AND operator, and they are resolved by the program using standard boolean logic.

Below the link window are the main commands for this screen. The first two are the previously mentioned arrow key commands that are used to scroll through either all the conditions in the database or all the links in a condition. There are also three function key commands that are used to set the condition's links. F1 is used to set a conditions permanent setting. This allows a condition to be set to always answer as "yes", "no", or "don't know." It is primarily meant to be used for conditions that will never vary in a particular installation, such as "Is the machine a turbine" or "Does the machine have a flexible shaft". (This option would normally only be used to set a condition to "yes" or "no", the "don't know" option is left in mainly for completeness' sake. The F3 key calls up a menu that allows the logical operators of the links to be changed between OR and a logical AND. The F2 key (as outlined in Fig. 19c) is used to set up links for unlinked conditions or to change links for ones that already have them. Using this when a linked condition is visible will call up another menu (not shown) that lists the links again and asks if the user wishes to replace them or to append new ones. This menu is skipped

if the condition is not linked. Once that is done, another short menu asks if the question is to be answered manually (that is, be unlinked and only answerable by an operator at run time) or by comparison to the data. If "compare to data" is chosen, the program presents a list of all probe locations available on the machines chosen for this link file. The operator picks from that list which ones have the data with which the condition is to be linked too. Once all locations are picked, if any of these locations have more than one kind of data (such as a vibration probe, which can have 1x, overall, phase angle, and gap volts) then the program asks about each location in turn, asking which sorts of data from each to use. If a location has only one kind of data (such as a temperature probe) then it does not ask.

Once the locations and types of data to be used in evaluations are decided upon, the program goes through the list of data and asks how each should be compared to the condition. There are three possible choices- a condition can be answered either "yes" or "no" based on the magnitude of a piece of data over a period of time, on how much the magnitude of the data changes over a period of time in absolute terms, and how much the magnitude changes in terms of percentages of its current value over a period of time. (Such as, "Is the 1x vibration steady?" can be answered "yes" if it does not vary by more than 0.2 mils over the last two weeks that data was collected). Actually setting these is done through a series of menus (not show here, they are very self-explanatory).

Once all the data selected has been gone through, the program returns to the "show conditions" screen, now with the new links. By default, each link is connected by a logical "AND". The "show conditions" menu can be used to change this to a logical "OR" at the operator's discretion. For example, if the operator wants the condition "Is the overall vibration

low?" to be answered "yes" only if the vibration is below a particular value at every probe, then the links can be set to something like: "The overall vibration at probe OUTV is below 1.3" AND "The overall vibration at probe OUTH is below 0.9," et cetera. If the operator wishes the condition "Is the overall vibration high?" to be answered "yes" if the vibration is high at any probe, then the links can be set to something like: "The overall vibration at probe IN-H is above 1.2" OR "The overall vibration at probe OUT-H is above 0.8," etc.

4.2 Operation of Combined System

The on-line WOT system is activated by shifting the IMTS display program off of full-screen mode and into a window so that the Program manager can be seen and double-clicking on the WOT icon. The display program will operate almost normally when in a window, the main problem is that the parts of the screen that are supposed to flash will not flash except when in full-screen mode. The WOT program may be left running as a minimized icon at all times, even when it is not being used, although this may slow collection of data by the IMTS if the computer does not have enough RAM to operate all the programs at once (at least 8 meg is required) or the processor is too slow to multitask between that many (a 386 is the minimum to run the system, but a 483 33 MHz minimum is recommended).

After activation, the WOT system will then display the same log-on screens as it does in off-line mode, and ask the same question about what the minimum confidence factor should be. Then a menu will appear, giving four options. The first, "Ask questions Manually," will cause the system to operate as if in off-line mode, that is, all conditions will be resolved by querying the operator. The second, "Use available automatic answers," will cause the system to operate in fully

automatic mode.

4.3 Installation at Virginia Tech Power Plant

In October of 1993, the process of installing the IMTS system at the Virginia Tech power plant began. This facility is a co-generation plant that provides most of the steam for the Virginia Tech campus. Power is generated by a single turbine and generator with a maximum capacity of 7.5 megawatts. The generator is equipped with four Bently-Nevada proximitor vibration probes that monitor two planes. The turbine also has four such probes. In addition, there is a keyphasor probe (with an overspeed trip) and a pair of thrust probes on the shaft. Various temperatures are measured by type J thermocouples and various RTD's, as well as total power output from the generator.

The purpose of this installation was to conduct long-term tests of the system during its final stages of development in an actual industrial setting. Feedback obtained from plant operators allowed the system's user interface to be fine-tuned.

An illustration of the installation at the VT power plant is given in Fig. 21. The turbine and generator are located on the ground-level floor of the power plant, with a main control panel in the same room. This control panel contains a number of analog and digital instruments. Power output from the generator is recorded by a paper strip-chart recorder. All other recording is done by manually noting the values of the various instruments at regular intervals.

As can be seen in Fig. 21, the data acquisition boards were installed in a small equipment locker immediately behind

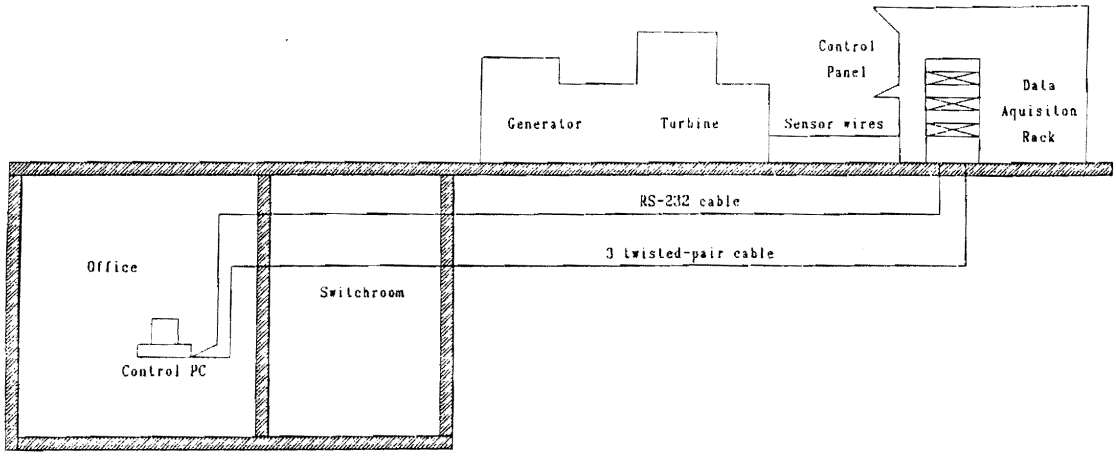


Figure 21
Installation of IMTS Hardware at the VT Power Plant

the main control panel. The IMTS computer was placed in the basement office of the plant's instrumentation supervisor. This location was chosen partially because that is where the supervisor wished to have access to the data, and partially because the office is normally locked (the turbine room is normally not locked, and several pieces of equipment left there have been stolen). The computer is connected to the data acquisition equipment by two cables, both approximately two hundred and fifty feet long. One is a standard shielded nine-wire serial cable, the other contains three individually shielded twisted-pair wires. The initial installation had to make do with temporary wires strung over the surface of the instrumentation console, held on by tape and adhesive clamps.

In this installation, the IMTS monitoring system would automatically record data from the generator and turbine much more frequently than the manual method used previously. It would allow the supervisor to monitor the turbine and generator from his office, and to access and interpret recorded data much quicker than was possible with the data recorded manually and on the strip-chart.

The conduit leading into the equipment locker was already nearly filled to capacity, so the IMTS system was initially connected to the computer with only the nine-wire serial cable, without the three twisted pair cables that are normally used. Theoretically, this cable should have been sufficient. A normal serial connection uses all nine wires, but only three are actually required to send data (one wire for each direction and a signal ground). The other wires transmit various "handshake" signals between the boards and computer when communications are initiated (these signals are essentially relics from the days when serial cables were used between electronic devices that were not computers). However, by connecting the transmitting and receiving handshake wires

together, it is normally possible to "fool" a computer and a remote device into thinking that the signals have been passed when, in fact, they have not, eliminating the need to pass these signals through wires. Thus, with only three of the serial cable's nine wires used to form a serial connection, the other six would be free to be used in place of the three twisted-pair wires. Since the serial wires are not shielded from each other (although the cable itself is shielded) it was expected that there would be crosstalk between the signal wires. The plan was to attempt to operate the system and investigate how serious the crosstalk was. The extra twisted-pair wires would be pulled through the conduit and used only if the crosstalk was unacceptably high. In any event, no matter how high the crosstalk, the complete system could at least be tested, even if the data acquired was too inaccurate to be useful in diagnosing the turbine.

Unfortunately, this plan did not work. The computer proved completely unable to communicate with the boards through a three-wire serial cable, despite many attempts and several consultations with the manufacturers of the board. The time and effort spent on these attempts were not entirely wasted, as they provided an excellent opportunity for the author to learn how to solder computer connectors and associated wiring.

Ultimately, the computer was connected to the board using a full, nine-wire serial cable, at which point commands could be sent and received normally. The baud rate was kept at 300 to minimize problems with noise signals in the cable (since relatively few commands pass between the computer and the boards, a low baud rate did not hamper the system's effectiveness). Unfortunately, since every wire in the cable was used for communications, another wire with three twisted pairs had to be pulled through the conduit in order to send

vibration signals. It was decided to use individually shielded twisted-pair wires, in the hope that the entire issue of crosstalk between the signals could be avoided entirely. Once this wire had been acquired, pulled, and connected, testing of the system could begin in earnest.

Even after the system was fully installed and wired up, there were extreme difficulties getting data to pass properly through any of the wires, due to the excessive noise that swamped the signals. Most of this noise was at 60 Hz, line frequency. Since this is also the frequency of the generator that was being monitored, the noise could not be eliminated by filtering the data. The source of the noise were identified, usually with a hand-held voltmeter and a short length of wire used an antenna.

The strongest noise source turned out to be a high-voltage electrical switchroom located behind the wall that the IMTS computer was adjacent too. Other noise sources included the coffee machine in the plant manager's office, other instrumentation that shared the space behind the control panel with the remote IMTS boards and the generator itself. A variety of techniques were used to reduce the noise. All slack was taken out of the between the instruments and the boards, as well as between the boards and the PC. A ground bus was installed on the remote board equipment rack, that was connected by heavy-gauge copper wire to a large copper bar that serves as the main ground for the other plant instruments. All grounds and wire shields were carefully connected to this ground bus, and disconnected at the PC end to avoid ground loops. A simple filter was used on the coffee machine cord, and a grounded foil-lined cardboard enclose was constructed around the PC (which proved very cumbersome to use and useless at eliminating noise, so it was dismantled). With all of these techniques in place, noise amplitude was reduced

to less than the signal generated by the weakest proximitor probe when the plant was in normal operation. This was considered acceptable for operating the system.

Over the 1994 year, the IMTS system was tested at the plant. Testing usually consisted of letting the system run until a problem was located, and then taking it off-line long enough to correct it. Extensive debugging, refining, and improvements were made during this period. Extensive modifications were made to the user interface, mostly as a result of suggestions offered by the plant's normal personnel.

The most extensive modifications were made to the system's graphical displays of trended data. The system was originally able to plot data only in terms of its magnitude. Data from different points could be compared by plotting them on different graphs, one on top of the other, and non-vibration data could not be compared to vibration data on one screen. Plant operators noted that what was actually wanted was the ability to compare relative magnitudes of data from many different sources at once on the same graph, and they considered it vital to be able to compare plant load to the vibration signals directly. This ability was added to the system, along with a new menu system to allow data to be picked and scaled properly. The plots were also modified to allow periods of less shorter than 24 hours to be shown. Unfortunately, the program's previous data management and graphics routines proved to be incompatible with these modifications, and had to be completely re-written.

Additional routines were added that allowed the program to eliminate data from the log files that had grown too old to be useful. The default age for dropping old data is sixty days, although that number can be adjusted while the program is in operation.

Numerous minor adjustments were made to the system to

make it simpler and easier for plant operators to use it. New switches were added to the setup module's screens. These allowed improvements such as the ability to vary the frequency of data logging for any given probe to be adjusted globally or individually, the ability to handle different kinds of probes in a less generic manner, and allowing new probes or machines to be entered into the system by copying the information off of existing machines and changing the relevant portions. The data files were also changed to allow machines that are not equipped with any vibration probes to be monitored.

The acquisition module's performance was improved, partially in response to changes made in the setup module. The FFT algorithm was slightly modified to increase its speed, and substantial improvements in speed were made with the algorithm for measuring temperatures, plant loads, and flow rates.

The installation itself was finalized. The temporary wiring to the various probes on the control console was removed and replaced by permanent wiring to the proper output terminals in the equipment locker. This caused a slight reduction in measured noise, presumably due to the shorter wire runs. This was done over the summer, when the power plant was not in operation.

At this point, the system has been used to continuously record data from thirty probes for periods of as long as two months. It would have been run for longer periods, but the system is periodically taken off-line to install one of the previously mentioned software upgrades and modifications.

5.0 Conclusions and Recommendations

This project has resulted in the creation of a computer-controlled data acquisition and analysis system for power plant and other similar industrial installations. This system

has proven itself capable of collecting large amounts of data for extended periods of time and has been tested in an actual industrial environment, namely the Virginia Tech Power Plant. The IMTS system piggybacks off of standard plant probes, and displays this data at a remote computer screen. It evaluates data through user-defined standards and alerts plant personnel when a predefined alarm defined by one or more of these standards is triggered. The WOT expert system can be used to analyze data acquired by the IMTS more completely, and can attempt to diagnose potential problems and recommend suggested fixes. This system is designed to be as generic as possible- it will work with any form of industrial machine equipped with voltage-variable data probes.

The assigned goals for this project have been accomplished with varying degrees of success. The extended time frame of the project resulted in unplanned tasks necessary to keep the system as near as possible to current computer operation and display technology. The major item that should be added to the system is an automated report summary capability that would give multiple levels of detail to satisfy either the operators or the plant supervisors.

As they currently exits, neither system can be considered ready for commercial release. Both still contain logical program errors and bugs that must be corrected before they can be considered anything but experimental. For example, the plot routines for trended data within the IMTS system does not always label its axes correctly, particularly when dealing with time periods of less than two days. While the system notes when more data points are plotted than there are pixels in the graph (through the word "compressed" that appears on the bottom of the screen) it should also indicate how many points of data are being plotted. The IMTS system needs to keep better track of the units used for certain of the

measurements and display them to the user whenever the data is displayed. Modifications should be done to the log file system- in particular, the system should allow the user to remove old data from the files and save this data in a different file for long-term storage. Consideration should be made for running the system on a network, so that logged data can be checked by different programs on different computers at the same time. An ability to store FFT spectra, or at least information about the peak magnitudes zones of within a spectra standard, might prove highly useful. The on-line WOT system is also in need of debugging and enhancements. The current system for creating links between data and conditions needs to be streamlined so that it is easier to define and verify the actual links to be used by the system.

6.0 References

[1] Durkin, John, "Expert Systems: A view of the field", *IEEE Expert Intelligent Systems and their Applications*, April 1996, pp. 56-63.

[2] Corsberg, Dan, "Alarm Filtering: Practical Control Room Upgrade using Expert System Concepts", *Intech*, April 1987, pp. 39

[3] Vale, Z.A, Mows, A.M, "An Expert System with Temporal Reasoning for Alarm Processing in Power System Control Centers", *IEEE Transactions on Power Systems*, V8, num 3, Aug. 1993, pp 1307-1313

[4] Kirk, R.G., J. Hoglund, and R.E. Mondy, "Development of a PC-Based Off-Line Expert System for Evaluation of Turbomachinery Response", *Proceedings of the 1st International Machinery Monitoring and Diagnostic Conference*, sponsored by Union College, Las Vegas, NV, Sept. 11-14, 1989, pp. 439-444.

[5] Kirk, R.G., J. Hoglund, and J. Keesee, "Application of Artificial Intelligence for Rapid Evaluation of Turbomachinery Dynamic Response and Stability:, *Proceedings of the JSME International Symposium on Advanced Computers for Dynamics and Design*, Tsuchiura, Japan, Sept. 6-8, 1989, pp. 149-154.

[6] Kirk, R.G., J. Hoglund, and R.E. Mondy, "Development of a Unique Off-Line PC-Based Turbomachinery Expert System," *Proceedings of IASTED Conference*, Zurich, Switzerland, June 24-26, 1989.

[7] Typrin, M.,E.C. Pawtowski, and R.G. Kirk, "A PC-Based Expert System for Rotating Machinery," *International Conference on Rotordynamics*, Venice, Italy, Apr. 1992.

[8] Hoglund, J.R., "An Expert System for Off-Line Analysis of Rotating Equipment," Mechanical Engineering Department, Virginia Polytechnic Institute and State University, August 1989.

[9] Typrin, M., IMTS (Intelligent Monitoring and Trending System) A PC based monitoring System for Rotating Machinery," Mechanical Engineering Department, Master of Science Thesis, Virginia Polytechnic Institute and State University, February 1992.

[10] Kirk, R.G., E.C. Pawtowski, and M. Typrin, "An Intelligent Filter for a PC-Based Expert System," *Proceedings of 45th Meeting of the Mechanical Failures Prevention Group*, Annapolis, MD, Apr. 9-11, 1991, pp. 109-116.

[11] Liddle, Ian, Reilly, Steven, "Diagnosing Vibration Problems with an Expert System", *Mechanical Engineering*, April 1993, pp. 54-55.

Appendix A. PROLOG PRIMER

It is difficult to understand the Prolog portions of the WOT system without a basic knowledge of the Prolog language. This section discusses some of these basics. Those familiar with Prolog need not bother reading it.

Prolog is a programming language based on Horn clauses, which are a subset of a formal logic called predicate logic. Solution to problems are inferred from previous solutions or known facts.

Prolog differs substantially from conventional procedure oriented languages such as FORTRAN and object-oriented languages such as C++. In any Prolog program, each statement is called a clause. All clauses can be classified as "facts" or "rules". A "fact" clause contains information of some kind. A "rule" clause is a relation that makes inferences from facts or other rule clauses. A series of clauses can be grouped together to form a predicate, which can be called by other clauses in a manner somewhat similar to a FORTRAN subroutine.

Predicates typically consist of a series of clauses, calls to databases, or other program operations. Unlike most forms of prolog, PDC prolog requires that programmer-defined predicates and their arguments be declared, typically near the beginning of the program. Arguments to a clause can be either a variable or a symbol. The name of a symbol begins with a numeral or a lowercase letter, such as the integer "2" or the name "bill". Of much more use are variables (sometimes known as "objects", since they share some of the properties of objects in object-oriented languages), which begin with uppercase letters. A variable is said to be "bound" when it is assigned to a piece of data, and "unbound" otherwise. Variables are bound by simply equating them to the data. For example, the clause:

MAL = "The machine has a loose bearing shell"

would bind the variable MAL to the text string "The machine has a loose bearing shell." If this fact clause was a part of a predicate, MAL could not be bound to a different text string within that particular predicate.

Simple variables are bound to single pieces of data, such as text strings or numbers. Compound variables may be bound to many pieces of data, to other variables, or even whole lists of data and/or variables. Variables are local, not global, in that they do not maintain the same value in different predicates unless they are passed, either explicitly or through a database.

PDC Prolog allows two forms of databases. The first are "internal databases", which consist entirely of "fact" clauses. They are stored in RAM and are typically used to access facts anywhere within the program.

An example of an clause that uses the internal database clause from the expert system editor is:

rule_using_cond(MAL)

This internal database clause consists of two parts. The first is the head of the clause, "rule_using_cond". The second part is the variable MAL. This clause could have two possible uses. If it is called while the variable MAL was unbound, this clause would bind MAL to a text string that had previously been stored in the internal database clause "rule_using_cond". If MAL was already bound, this clause would test to see if it was the same as any of the strings that had been stored in "rule_using_cond."

Storing such a string in the internal database would use the clause:

assert(rule_using_cond(MAL))

This clause consists of three parts. The first part is the built-in command "assert(<fact clause>)." This command

causes the fact clause between the parenthesis to be inserted into the internal database. Any clauses that are to be sorted into the internal database must be declared at the start of the program. The second part is the fact clause "rule_using_cond(<text string>), as described earlier. The third part is the variable MAL. This particular clause could not be called unless MAL had previously been bound to a text string.

The other form of database are "external databases", which reside in DOS files or extended memory. Like an internal database, external databases can access data sequentially or randomly. They take significantly longer to access than internal ones, and are somewhat more complicated to control. Their most significant advantage over internal databases is that they can be extremely large, since only the portions currently in use by the program are kept in RAM.

Prolog programs operate by considering each predicate in turn. All rule predicates are treated as a "goal" to be "resolved", or satisfied. A predicate is resolved when all of its sub-goals, which can be mathematical operations, calls to databases, built-in functions, or other rules, can be resolved.

For example, consider the simple predicate "stall" from the expert system editor:

```
stall :-
 makewindow (9,12,64,"",10,20,3,40),
 write ("      Please Wait !  I'll hurry.").
```

This predicate takes no arguments, and is defined by two built-in predicates, "makewindow" and "write", both of which can always be resolved immediately. "Makewindow" takes eight input arguments, all of which are used to define a small window in the center of the screen. "Write" takes a single string argument, and prints it in the current write device (in

this case, the just-created window). Thus, when "stall" is called upon, a small window is created on the screen, containing the message "Please Wait ! I'll hurry." This is typically done just before a lengthy database call is performed. The call to stall is always followed by a "removewindow" command, to eliminate the small window.

Prolog predicates that perform simple functions such as writes to the screen or straightforward mathematical operations on numerical arguments differ little from their counterparts in other languages, such as FORTRAN statements or subroutines. Predicates that perform more complex operations, such as backtracking or recursive looping, can differ substantially from equivalent statements in other languages.

One of the most useful abilities of prolog is its ability to easily handle lists of variables. For example, the expert system predicate "bref" is used to pull a list of references from the database and print it to the screen:

```
/*-------------------------------------------------------*/
    /*                        Predicate Bref
Recursive predicate takes as input list of reference ID
numbers.  List broken into PUB and REST.  Reference withdrawn
from external database "mydba".  Data withdraw is:
    AUTHOR....Name of author of reference
    TITLE1....First line of title.
    TITLE2....Second line of title.
    PUBLISH...Publisher.
    PDATE.....Date of publication.

    Data is printed to screen.  New reference printed after
each
pressed. */
/*-------------------------------------------------------*/

 bref ([PUB|REST]):-!,
  chain_terms (mydba,lib,dbdom,library (PUB,AUTHOR,TITLE1,
                                  TITLE2,PUBLISH,PDATE),_),
  writef ("\nPaper   : %\n          %\nAuthors : %\nPub. by :
       \nDate    : %\n",TITLE1,TITLE2,AUTHOR,PUBLISH,PDATE),
  readkey(_),!,
  bref (REST).
```

```
bref        ([]):-!.
```

This predicate has two instances (separated by a blank line), and takes a list of integers as an argument. Each integer identifies one reference in the database that is to be printed. When called with an list of integers, the first instance of this predicate will operate. The list is broken up into two parts by a "¦" symbol: the first integer is assigned to the variable "PUB." The rest of the list is assigned to the variable "REST." In order to be broken up by the "¦" symbol, the list must have at least one element. Were it to have only one, "PUB" would be set to that integer, and "REST" would be set to the "empty list," symbolized in Prolog as "[]." The first sub-goal in the predicate is "chain_terms", a built-in predicate that withdraws from a database file the information concerning a reference with the ID number identified by the integer assigned to "PUB". This information is stored in a six-part compound variable called "library", which contains one integer (PUB) and five text strings (Author, Title1, Title2, Publish, and Pdate). This variable can be passed from one predicate to another either with all six sub-parts identified, as shown above, or as a single variable.

The second sub-predicate in bref, "writef", is a built-in predicate that causes a formatted write to occur on screen. The next statement, "readkey", takes a single key from the keyboard. Since its input argument an underscore, (a "null" variable), any key can be pressed, and it will not be bound to any variable. This clause simply makes the program wait until a key is pressed before continuing. The explanation point is called a "cut," which is used to control memory use and backtracking. The predicate then calls itself, in an iterative loop. If "REST" is not an empty list, the first

instance of "bref" will repeat and print out the next reference. Eventually, all references will be printed, and "REST" will be an empty list. An empty list cannot be broken into two parts, so the first instance of "bref" will not be able to process it, and will "fail." The program will then seek out the next instance of "bref." This instance can only take an empty list as an input argument, so it will not fail. The second instance performs no operations, but serves merely to allow the initial call to "bref" to be satisfied. Once the second instance of "bref" has been called, the predicate that called "bref" in the first place will proceed with whatever its next operation is. If this second instance of "bref" were not present, then the program would start to backtrack at the point at which the original call to "bref" had been made.

This example demonstrates several key features of PDC Prolog. The first is that predicates are allowed to recursively call themselves. In fact, this is the preferred way of handling lists. The second is the extensive database system. Finally, it illustrates how the flow of a program can be controlled by allowing a predicate to recursively call itself until it fails. To contrast, a similar task would be accomplished in FORTRAN by the subroutine:

```
      SUBROUTINE BREF(NPUB,PUB,FILENAME)
*-------------------------------------------------------------
*      Input:
*               NPUB      - Number of published reference to
locate
*             PUB       - Array of reference numbers
*                  FILENAME - Name of file containing the
references
*-------------------------------------------------------------
*      Variables:
*             I         - Integer counter
*             AUTHOR    - Name of author of the Ith reference
*              TITLE1    - First line of the title of the Ith
reference
```

```
*              TITLE2    - Second line of the title of the Ith
reference
*              PUBLISH   - Publisher of the Ith reference
*                 PDATE      - Date of publication of the Ith
reference
*             JUNK      - Junk Character string for User input
*-------------------------------------------------------------*
*     Declare Variables
      INTEGER I
      INTEGER NPUB,PUB(100)
      CHARACTER*8 FILENAME,JUNK
      CHARACTER*40 AUTHOR,TITLE1,TITLE2,PUBLISH,PDATE

*     Loop through all reference numbers
      DO I = 1,NPUB

*          Get the data for the Ith reference number
           CALL
GETREF(FILENAME,PUB(I),AUTHOR,TITLE1,TITLE2,PUBLISH,PDATE)

*     End loop
 10   CONTINUE

*     Print the data
      WRITE(6,20)PUB,AUTHOR,TITLE1,TITLE2,PUBLISH,DPATE
      READ(5,*) JUNK

      RETURN
      END
```

Many differences exist between the subroutine and the predicate. The FORTRAN subroutine must be told the length of the list to be processed, and must dimension arrays to contain the list. The Prolog predicate need not know the length of the list. Indeed, there is no reason to determine the length of a list unless the user specifically requires it. FORTRAN routine also requires another user-defined or library subroutine, GETREF, to extract the data from the database.

The biggest difference between prolog and most other languages is the form of flow control called backtracking. When a predicate is called, the program will note if each of its subgoals have only one possible solution, or many

different solutions. For example, a write to the screen can only be resolved in one way, namely, by writing the input variables. A call to a database, such as the built-in "chain_terms" predicate described above, has as many possible solutions are there are entries in the database.

If a predicate is found whose subgoals can be resolved in more than one manner, the first available solution is used, and a "backtracking point" is left at that predicate. If a sub-goal in a later predicate cannot be resolved, the later predicate will "fail," and the program will begin to "back up." That is, it will begin at the predicate who's subgoal could not be satisfied, and begin looking at previously executed predicates in the reverse order from which they were originally executed. Any predicates that had only one possible solution (such as a write command, or a predicate with a fixed input) will be ignored. When a predicate is found that has at least one untried possible solution (i.e., one that has a backtracking point set) the original solution is undone, the backtracking point is cleared, and one of the untried alternative solutions is tried. The program continues from that point. If the predicate that was backtracked to still has an untried possible solution, a backtracking point is set again. If a program backtracks to the same predicate enough times to try all of the possible solutions, then no backtracking point is set, and the program backtracks past that predicate and looks for the next backtrack point.

The following expert system predicate illustrates the use of backtracking:

```
auto_answer(3):-
chain_terms(mydba,con2,dbdom,cond2(HEAD,_,_,_,Old_answer,
  _),_),
  not(Old_answer = 3),
  not(Old_answer = 0),
  not(Old_answer = 4),
```

```
  add_answer(HEAD,Old_answer),
  fail.

auto_answer(_):-!.
```

The predicate auto_answer was called by a previous predicate with one input argument, an integer. If the argument was the integer three, the first instance of auto_answer shown will be executed (Other instances of auto_answer exist in the expert system, but are not shown here, for simplicity). The first sub-goal is "chain_terms", which calls a compound variable, cond2, from the external database. This variable represents a condition for the expert system to ask. Since this is a database call, there always exists the possibility that the predicate could be called again and give a different answer (the predicate does not check if there are, in fact, any more conditions in the database). Thus, a backtracking point is left at the chain_terms predicate. Only two of the sub-variables in cond2 are assigned to variables, the text string HEAD, which is the text of the condition, and the integer Old_answer, which represents the answer given to this particular condition the last time this database was used. This variable is subjected to a series of "not" predicates, all of which succeed if the integer is not equal to 3, 0, or 4. If it is equal to any of these numbers, one of the predicates will fail. This will cause the program to back up until it finds the backtracking point left at chain_terms. It will then attempt to retrieve the next condition from the database and proceed as before. If a condition is found whose Old_answer value passes through the checks, the text of that condition (HEAD) as well as the Old_answer variable itself are sent to the predicate add_answer (not shown here), which inserts them into an internal database. Once the add_answer predicate is complete,

the program proceeds to the next predicate, "fail". This is a built-in special predicate that can never be satisfied, so it will always fail. Thus, the program will back up to the backtracking point left at chain_terms again. Eventually, all of the conditions will have been tried, and then chain_terms itself will fail. This will cause the program to backtrack even farther, to the backtracking point that was left at the beginning of auto_answer (because auto_answer contained more than once instance). The next instance of auto_answer will be tried. This one takes any input arguments, and always succeeds, so it will simply satisfy the original call to chain_terms. The exclamation point "!" in this predicate is a "cut". This prevents a backtracking point from being left after this instance of auto_answer is completed.

Appendix B. Acquisition Module Data File Format

The IMTS evaluation module stores all of it's data about the data from the acquisition module and the standards that are to be applied to that data in a database file called ODD.SET. A sample of this file is shown in Fig. B-1. This is an extremely small database, for demonstration purposes only. This database was reformatted into 80-character lines so that they could be printed. In the actual database, many of the lines would be hundreds of characters long. The labels "Line x:" are used to identify what the original lines would be normally. They are meant to be read only by a program, not by humans, so there is normally no need to keep them narrow enough to fit on one screen. This database contains essentially two kinds of information: information about the format of the raw data files, and information about the standards that to this data will be compared to.

The first internal database predicate, "target", contains the format information in a list. Each item in the list is a complex data variable, called "string_target", that completely specifies the format of one line in the acquisition module's output file.

Each string_target variable contains two integers and a text string. The first integer (from one through five) identifies which type of data the line contains (vibration plane, single vibration, thermocouple, or A/D). The second integer identifies which machine the variable refers to. Lines 7-9 show which corresponds to which machine. The text string acts as a template for the actual data line. Each "xxxx" represents where a real or integer number representing data would appear. If any words or ASCII symbols were to be present in the data file, they would also appear in the template. The words "mname," "hour," "minute," "second," "day," "month" and "year" are special keywords that represent

the places in the lines where information about the name of the machine the data was taken from, as well as the time and date at the point the data was acquired.

After each "target" line is a series of predicates that define the various standards that are to be applied to the data. The first is the predicate "graphs." This predicate contains a list of items of type "gr," each of which represents one of the graphs that is to be used by the program. Each "gr" contains the vital information about each graph. In order, these are the graph title, the identification integer, the horizontal axis label, the vertical axis label, the upper and lower bounds for the vertical axis, the upper and lower bounds for the horizontal axis, a list naming each of the graph's zones from highest to lowest, and a list of each of the points that make up the lines that form the graph. The zone list contains items of type "ln," each of which contains a name and an integer identifying each zone. The point list is made up of items of type "pt," each of which contains the x- and y- coordinates of each point, as well as the name of the region that each point belongs too (a zone is defined by the line that is at its upper bound). The lines are defined in order, starting with the bottommost and progressing to the uppermost. For any given line, points are identified from left to right. All lines defined by this program must be a proper function: that is, there can be only one possible y-coordinate for each x-coordinate. The end of each line is identified by a "dummy point" with coordinates 1,1 and the string "End of line."

In this case, there is only one graph, named "amp. vs. speed," with an ID number of 1, a horizontal axis label of

```
Line 1: target([string_target(1,1," mname, xxxx , hour :
minute : second,day - month - year"),string_target(2,1,"xxxx,
xxxx,xxxx,xxxx,xxxx,xxxx,xxxx,xxxx,xxxx,xxxx,xxxx,xxxx,xxxx,
xxxx,xxxx,xxxx,"),string_target(3,1,"0"),string_target(4,1,"0"
),string_target("0"),string_target(1,2," mname, xxxx , hour :
minute : second,day - month-year"),string_target(2,2,"xxxx,
xxxx,xxxx,xxxx,xxxx,xxxx,xxxx,xxxx,"),string_target(3,2,"0"),
string_target(4,2,"0"),string_target(5,2,"xxxx,"),string_tar
get(1,3," mname, xxxx , hour : minute : second, day - month-
year"),string_target(2,3,"xxxx,xxxx,xxxx,xxxx,xxxx,xxxx,xxxx,
xxxx,xxxx,xxxx,xxxx,xxxx,xxxx,xxxx,xxxx,xxxx,"),string_targe
t(3,3,"0"),string_target(4,3,"0"),string_target(5,3,"xxxx")])
Line 2: graphs([gr("amp. vs. speed",1, "Speed (rpm)","overall
vibration(milspk-pk)",0,25,0,5000,[ln("trip",60),ln("alarm",
50),ln("rough",40),ln("acceptable",20),ln("runout",10)],[pt(
0,1,"runout"),pt(5000,1,"runout"),pt(1,1,"End line"),pt(0,2.5,
"acceptable"),pt(772.162,9.993,"acceptable"),pt(2502.807,1.9
90,"acceptable"),pt(4988.3,1.990,"acceptable"),pt(1,1,"Endof
line"),pt(0,9.99,"rough"),pt(901.040,15.38,"rough"),pt(1895.
240,15.38,"rough"),pt(2447.57,11.5,"rough"),pt(3091.963,7.90,
"rough"),pt(3791.58,5.12,"rough"),pt(4988.308,5.12,"rough"),
pt(1,1,"End of line"),pt(0,15.038,"alarm"),pt(937.86,20.08,
"alarm"),pt(1840.07,20.083,"alarm"),pt(2963.085,13.124,"alar
m"),pt(3975.697,6.165,"alarm"),pt(4988.308,6.165,"alarm"),pt
(1,1,"End of line"),pt(0,25,"trip"),pt(5000,25,"trip")])])
Line 3:s_v_trip(trscale(1,9,"vib. spec #1","INV","Overall",
"planar",[range(3,10,"alarm","alarm"),range(4,8,"excessive",
"excessive"),range(5,7,"not runout","not runout")],"fixed",
almrngs(["alarm","excessive"]),0))
Line 4: s_v_trip(trscale(1,11,"1x std","INV","1x ","planar",
[range(0.22,3,"trip","trip"),range(0.22,2,"wide
variation","widevariation"),range(0.22,1,"off-setpoint","off
-setpoint")],"deviation",almrngs(["None chosen"]),0))
Line 5: s_v_trip(trscale(1,-1,"speed","OUTH","Shaft Speed",
"planar",[range(3599.97,1500,"trip","overspeed trip"),range(
3599.97,1000,"alarm","alarm"),range(3599.97,600,"very
slow","very   fast"),range(3599.97,200,"slowing","speeding
up")],"deviation",almrngs(["None chosen"]),0))
Line   6:s_v_trip(trgraph(3,"planar",1,-1,"Shaft
Sp","200.00",1,"Overa","200.00",[tri("Max_cond",40,"Trip")]))
Line 7:mach_name("01-TEST-GEN",1,"kphase")
Line 8:mach_name("01-TEST-THRUST",2,"kphase")
Line 9:mach_name("01-TEST-TURB",3,"kphase")
Line 10:ad_data(adrelay(2,"LOAD",2,0,1.44,0,0,0,0,0,0,0,0))
Line 11:ad_data(adrelay(3,"EXHBR",1,0,-30,0,0,0,0,0,0,0,0))
```

Figure B-1

Sample Configuration File for AQUMOD.EXE

"Speed (rpm)," a vertical label of "overall vibration (mils pk-pk)," a vertical axis from 0 to 25, a horizontal axis from 0 to 5000, and five regions, named "trip," "alarm," "acceptable," and "runout."

After the graph definitions are the scalar and vector standard. Each of these are defined by a "s_v_trip" predicate (the name is a contraction of "scalar or vector trips," which is what these standards were called in the early stages of this project). Each of these take up one line apiece.

The first one, located on the third line of the file shown in Fig. B-1, contains an internal database predicate of the of the type "trscale," identifying it as a scalar standard. The first item in this scalar standard is the integer 1, which identifies which machine the standard applies to. The sixth item identifies what kind of probe the data was gathered from. In this case the string is "planar," which means that the standard is meant to be applied to data gathered from one of the planar vibration probes on that machine. The second item, the integer 9, means that the standard should be applied to the ninth piece of data from the planar probes of the machine in question.

The third item, "vib spec #1," is a string that gives the name of this specific standard. The fourth item, "INV," is the name of the vibration probe where this data comes from. The fifth item, "Overall," identifies what variable the data from the probe is supposed to represent. After these identifiers is a list that defines the standard itself. Each element of the list is of type "range." Each "range" contains two real numbers (a high and a low value) and two strings, which is how the data should be judged if it falls within those high and low values, respectively. Following the list is a string, "Fixed," that identifies this standard as a fixed scalar standard (as opposed to a deviation standard) and a

list of "alarmrngs." This list contains the names of the ranges that are to be considered alarm conditions (in this case, an alarm will be triggered if the data falls into either the range named "alarm" or "excessive"). Finally, there is an integer with the value zero. This integer serves no function in the data file. It is used within the filter program to keep track of which standards need to be reset. It is present in the data file only as a placeholder.

There are a total of three scalar standards defined in Fig. B-1, on lines 3, 4 and 5. The first is fixed, the second and third are deviation. The deviation standards are defined in a manner very similar to the fixed standard. The exception is in the list of "range" elements. The first number in each range represents the midpoint of a data range, and the second represents the minimum deviation from that midpoint for the data to be contained within that range. Both kinds of standards are defined from broadest range to narrowest. Data that falls within the narrowest range is considered inside the default range named "nominal".

After the scalar standards are the vector standards. Figure B-1 contains one. Each are defined by a predicate named "trvector" that contains the necessary data. The first data item is a number identifying which machine the standard applies to. This is followed by a string identifying what kind of probe the data is gathered from (once again, it is from a planar probe). The third item is a string that names the standard. The fourth and seventh items are integers that identify which piece of data are to be treated as magnitude and phase angle. The fifth and eighth name the probe that the data is gathered from, and the sixth and ninth name the data itself.

After the vector standards is a single standard that applies to the graph, on line 6. This is an object of type

"trgraph", and contains the ID number of the machine it is to be applied to, 3, the type of probe it is to be applied to, "planar", the ID number of the graph that it is to employ, 1, then an ID number, name, and scale factor for the data to be plotted on the horizontal graph (-1, "Shaft Sp," and "200," respectively), followed by the same information for the vertical data point (1, "Overall," and "200"). After this is the list of any alarm trips that are set for this data. In this case, only one. The data will be in an alarm state if the data is in or above the graph region with the ID of 40, which is the "rough" region. The first string in the trip (take up by "Max_cond" here) is what tells the system to alarm if the data is in or above, in or below, or only in, the indicated region.

This format is intentionally redundant in several places and more complex than necessary to handle the task that it was designed for. For example, the integer that identifies line number could easily be dropped, and the position of the item in the list used in its place. The template string could be replaced by an integer that identifies the number of data points on this line, and the time and date information simply assumed to always be in the same place. The reason for this additional complexity is to allow the system to better handle changes in one or more parts without having to re-write the system from scratch. The first generation IMTS/WOT system suffered heavily from this problem, which is one of the reasons it was abandoned.

Appendix C. List of Programs

The following is a list of programs used by the IMTS and WOT system, along with a note of what language they were written in and a brief description of each program's purpose:

1. GENSTD.EXE: Prolog program. Used to create scalar and vector standards.
2. GRAPH.EXE: Prolog program. Used to create graphical standards.
3. FFTSTD.EXE: Prolog program. Used to create standards that FFT spectra are compared to.
4. WOTEDIT.EXE: Prolog program. Used to create or modify databases for the WOT expert system.
5. WOTEXP.EXE: Prolog program. The WOT expert system itself. Can be run in off-line or on-line mode.
6. MODIFY.EXE: Prolog program. Used to modify WOT databases created for the off-line-only version of the WOT expert system so that they can be used with the current version.
7. EXPSETUP.EXE: Pascal program. Used to create configuration file that contains information about the hardware of the machines that are to be analyzed.
8. EXPSYS.CFG: ASCII data file. This is the hardware configuration file created by EXPSETUP.EXE.
9. SETFILT.EXE: Prolog program. Used to select which scalar, vector, and graphical standards the IMTS evaluation module is to use to evaluate data, and to decide which standards constitute alarms.
10. ODD.SET: ASCII data file. Created by SETFILT.EXE, contains the standards that are to be used by the acquisition file, information on which pieces of data are applied to which standard, and what standards will trigger alarms.

11. AQUMOD.EXE: BASIC program. This is the IMTS acquisition module. Operates all data acquisition hardware and passes information along to the evaluation module, and compares FFT standards to FFT spectra.
12. FILDATA.EXE: Prolog program. This is the IMTS evaluation module. It compares acquired data to the standards in ODD.SET and passed the results on to the display module.
13. DISPPROG.EXE: Pascal program. This is the IMTS display program. Displays all acquired data to the user, and is how the user interacts with the IMTS system.
14. WOTSETUP.EXE: Prolog program. Used to create link files that connect conditions in a WOT database to data in the IMTS log files.
15. AUTOANS.EXE: Prolog program. This program is used by the on-line WOT system. It goes through a WOT database created by WOTEDIT.EXE, a link file created by WOTSETUP.EXE and data log files created by DISPPROG.EXE, and answers as many of the conditions as possible based on data.

Vita

Eric Pawtowski is originally from the Washington DC area. He graduated from Georgetown Preparatory School in 1985, and graduated with a B.S. in Mechanical Engineering from Virginia Tech in 1989. He has stayed on at this university to continue with this Master's Thesis in that department.

With the completion of his Master's thesis, he intends to find a job related to his field someplace in the Eastern portion of the country and marry his fiance', Yvonne Bennett, shortly afterwards.