Automated Classification of Power Signals

by

Ethan R. Proper

B.S. Electrical Engineering, University of Wisconsin – Madison, 1995

Submitted to the Department of Mechanical Engineering and Engineering Systems Division in Partial Fulfillment of the Requirements for the Degrees of

Naval Engineer
and
Master of Science in Engineering and Management
at the
Massachusetts Institute of Technology

June 2008

Signature of Author___ _____

Department of Mechanical Engineering and
System Design and Management Program
May 9, 2008

Certified by __ _____

Robert W. Cox
Assistant Professor of Electrical and Computer Engineering, UNC Charlotte
Thesis Supervisor

Certified by _____

Steven B. Leeb
Professor of Electrical Engineering and Computer Science & Mechanical Engineering
Thesis Supervisor

Accepted by _ ____

Pat Hale
Director, Systems Design and Management Fellows Program
Engineering Systems Division

Accepted by _____ _____

Lallit Anand
Chairman, Department Committee on Graduate Students
Department of Mechanical Engineering

Page Intentionally Left Blank

Automated Classification of Power Signals

by

Ethan R. Proper

[SOHM]

Submitted to the Department of Mechanical Engineering and the Engineering Systems Division on May 9, 2008 in Partial Fulfillment of the Requirements for the Degrees of

Naval Engineer

And

Master of Science in Engineering and Management

ABSTRACT

The Non-Intrusive Load Monitor (NILM) is a device that utilizes voltage and current measurements to monitor an entire system from a single reference point. The NILM and associated software convert the V/I signal to spectral power envelopes that can be searched to determine when a transient occurs. The identification of this signal can then be determined by an expert classifier and a series of these classifications can be used to diagnose system failures or improper operation. Current NILM research conducted at Massachusetts Institute of Technology's Laboratory for Electromagnetic and Electronic Systems (LEES) is exploring the application and expansion of NILM technology for the use of monitoring shipboard systems.

This thesis presents the *ginzu* application that implements a detect-classify-verify loop that locates the indexes of transients, identifies them using a decision-tree based expert classifier, and then generates a summary *event file* containing relevant information. The *ginzu* application provides a command-line interface between streaming preprocessed power data (PREP) and an included graphical user interface. This software was developed using thousands of hours of archived data from the Coast Guard Cutters ESCANABA (WMEC-907) and SENECA (WMEC-906). A validation of software effectiveness was conducted as the software was installed onboard ESCANABA.

Thesis Advisor: Robert W. Cox
Title: Assistant Professor of Electrical and Computer Engineering, UNC Charlotte

Thesis Advisor: Steven B. Leeb
Title: Professor of Electrical Engineering and Computer Science & Mechanical Eng.

The author would like to acknowledge the following organizations and individuals for their assistance. Without them this thesis would not have been possible.

- Dr. Steve Leeb who provided continuous enthusiasm, fabulous technical insight, lofty goals, and a complete unwillingness to settle for anything short of excellence.
- Dr. Robert Cox who provided the ideal direction and recommendations – and was invaluable in keeping me pointed in the right direction.
- Pat Hale who cordially greeted a significant number of 2008 Navy students into the MIT System Design and Management program – providing us with an incredible education outside of our traditional Naval Engineering path.
- Jim Paris who provided wizardly software support and was instrumental in the Escanaba deployment as he was able to quickly integrate the *ginzu* application with his 2006 data acquisition framework.
- LT Ashley Fuller and LT Perry Branch who single-handedly resolved all hardware related issues during the Escanaba NILM installation and follow-up visits.
- The Officers and crew of the USCG Cutter Escanaba. Specifically, LT Jennifer Haag who went far past expectations in allowing the MIT LEES research team access to the ship.
- The NILM authors and developers who created the various classification tools and diagnostics. The ginzu application was only possible through their hard work and knowledge.

Table of Contents

List of Figures

List of Tables

Page Intentionally Left Blank

1. Introduction

1.1 Background

The Non-Intrusive Load Monitor (NILM) is a device that measures voltage and current at a single node in an electrical distribution system. The voltage and current waveforms (Figure 1-1) are used to calculate power levels (Figure 1-2) and harmonic contents, which are then analyzed over time to determine the behavior of the monitored system. The electrical theory and architecture of the NILM hardware has been well documented in works such as [14].

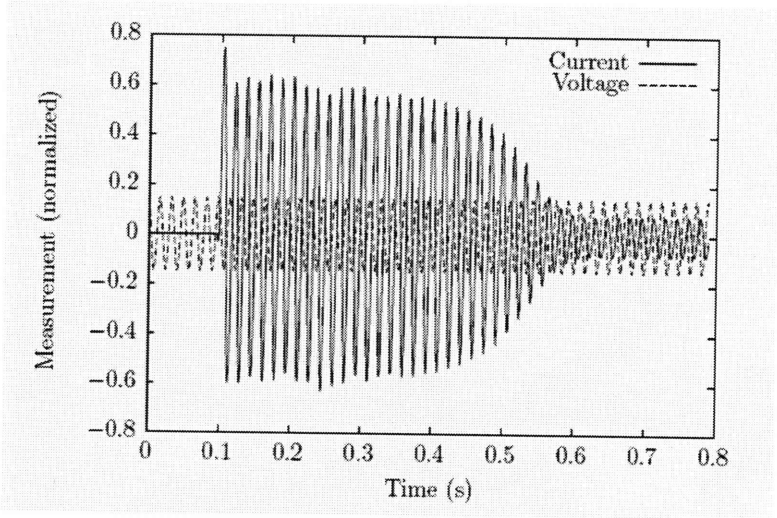


Figure 1-1 : Raw AC voltage and current measurement taken during motor start transient. [14]

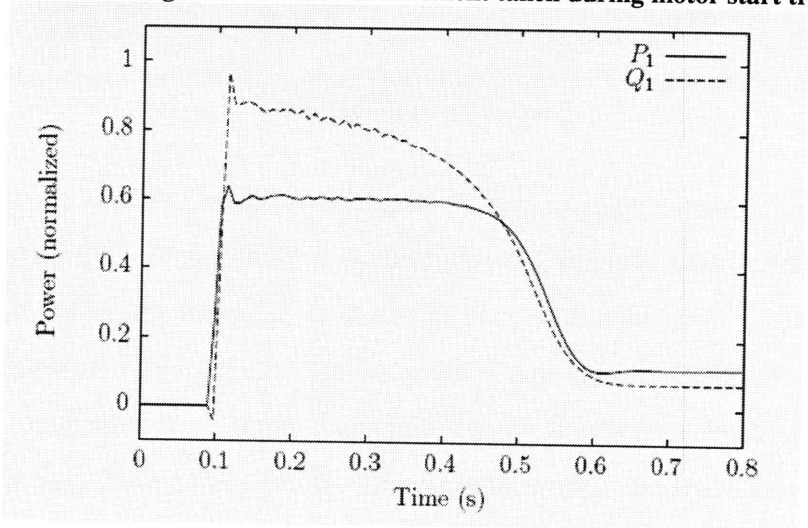


Figure 1-2 : Voltage and current signals shown in previous figure are converted (preprocessed) into a spectral power envelope. This representation provides a clearer representation of variable power usage over time. [14]

Over the last 20 years, the approach used to analyze NILM signatures has evolved and improved. Recent research has shown impressive results in non-intrusive load monitoring of specific systems such as shipboard waste disposal [12, 15] and the water purification systems [11]. This research has shown how the various characteristics of NILM signatures can be used to identify system problems and has indicated that the NILM could be used for realtime detection and classification. In parallel with these findings, other research [8, 14] has shown that intelligent automated methods such as machine based pattern recognition can provide the tools to enable realtime event classification.

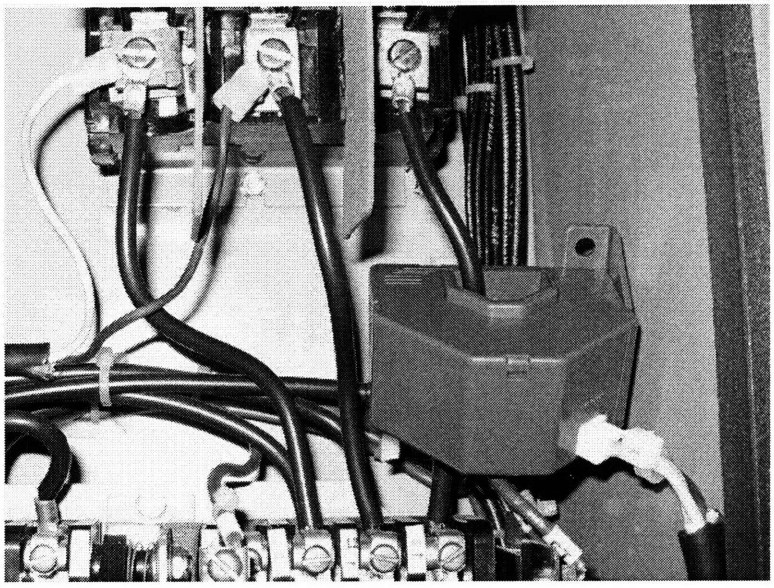


Figure 1-3 - Typical NILM hardware installation showing a single current transducer and two leads to measure voltage. [14]

Prior to this thesis work, a typical NILM installation involved the acquisition of voltage and current data through the insertion of transducers in a distribution panel or controller (Figure 1-3). This data is archived as a series of one-hour data files [14]. These file are retained on the hard drive of an attached PC workstation and are transferred to permanent archives[1] when convenient. In the case of seagoing vessels, the data is transferred when the ship returns to port. In this system, no significant realtime

[1] In practice, archived power data is permanently stored on 'bucket', a redundant array of independent drives (RAID) located at the MIT Laboratory for Electromagnetic and Electronic Systems (LEES).

information is passed to the system operators. The archived data is analyzed at a later date – typically many months after it is received. This analysis is often aided by ship's logs or crew comments regarding problems that occurred while underway. By reviewing the NILM signatures that correspond to a known problem, the identifying characteristics of the fault condition can be determined and used for future diagnostics. An example of a proposed NILM architecture that would acquire data and generate system diagnostics is shown in Figure 1-4.

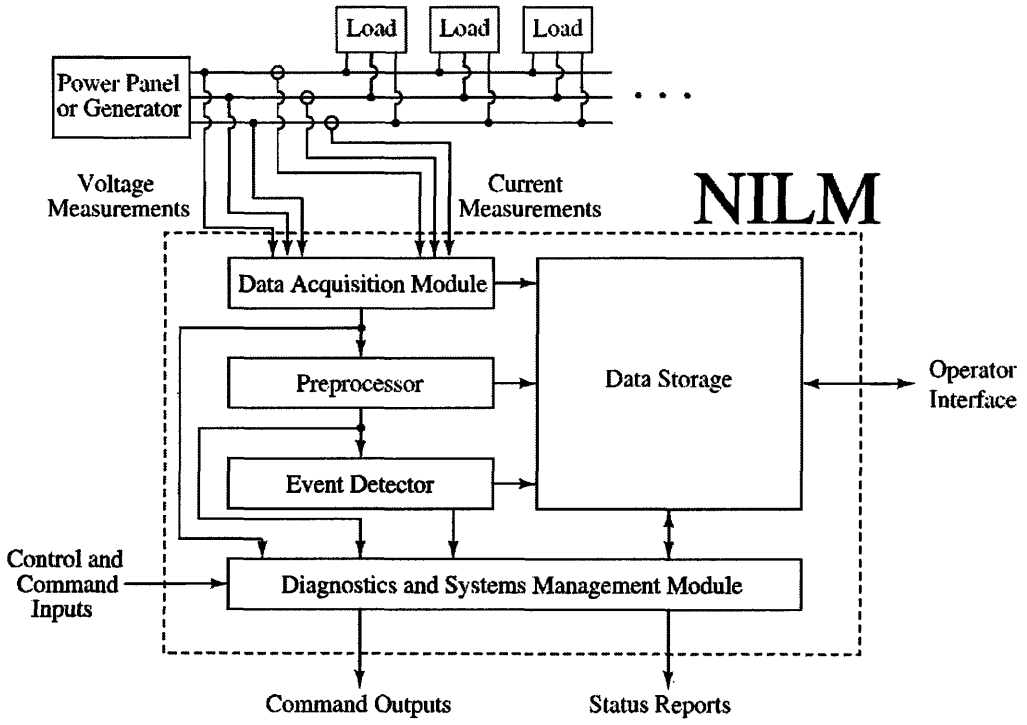


Figure 1-4 : Proposed Architecture of NILM [12]

The research presented in this thesis shows how the gap between the acquisition of data and the analysis and presentation of data can be bridged through an intelligent software solution. This interface software implements a decision-tree classifier that provides the ability to detect and classify system events within seconds of when they occur. Once the events are classified, they can be imported into a graphical user interface (GUI) that performs various system diagnostics. The thesis provides a summary of a diagnostic NILM deployment on a Famous Class USCG Cutter (*WMEC-907 Escanaba*).

The Escanaba deployment represents the first implementation of a fully-functional diagnostic NILM with user interface on realtime power data. The core software is named *Ginzu* - due to the function of 'slicing' streaming power data into small usable event data files. Additionally, it is shown that, while the decision-tree classifiers are accurate and effective, they require a significant degree of *a priori* knowledge and a time consuming test and validation process in order to tune internal threshold variables. Since this degree of effort may be undesirable for wide-spread NILM implementation, a second solution is introduced (5.2.1 *Ginzu2*) that attempts to attain performance similar to the decision tree classifiers without requiring the upfront costs of expert training. This second solution provides an unsupervised learning algorithm to cluster events with similar characteristics. After the clusters are formed, they can be labeled by an expert user – and then used for future classification through a Euclidean-distance calculation.

1.2 Motivation for Research

NILM research has shown that power signatures can provide an effective and cost-efficient [12] diagnostic tool in recognizing unusual and/or abnormal system performance. Specifically, NILM research has shown direct applicability to naval ship systems. In a typical ship condition based maintenance strategy, maintenance actions are either preventative or corrective. When preventative maintenance is performed, a cost is assumed in order to reduce the probability of future failure. If the cost of preventative maintenance exceeds the cost of component failure, the maintenance is ineffective. However, since the total cost of any component failure can be difficult to determine – it can be difficult to determine the value of certain preventative maintenance actions.

As the Navy searches for methods to minimize the cost, NILM provides additional data to aid the condition-based maintenance strategy. NILM provides the opportunity for early detection of system problems; allowing the ship's crew to take action before a component failure. In order to recognize these benefits, the significant gap between the acquisition of NILM data and the presentation of this data must be overcome.

1.3 Objective and Outline

The findings of virtually all recent NILM research have indicated that abnormal system behavior was evident in the system's power signatures; and in many cases, that the data could have been used to detect the abnormal behavior before it lead to a greater or prolonged failure. The objective of this research is to create, improve, and validate a framework that can be used to allow realtime detection and classification of NILM signals as well as deliver this information to the NILM user.

The thesis consists of 5 chapters. Chapter Two identifies a candidate system for the application of a signal classification and realtime diagnostic strategy – the shipboard vacuum assisted waste collection and disposal system (CHT). Chapter Two also provides the classification tools that are used to construct an event classifier and provides a generic classification framework that is applicable to shipboard systems. Chapter Three describes the logic contained in the *Ginzu* algorithm and the decision trees that are implemented to classify both the waste disposal (CHT) and reverse osmosis (RO) systems. This also includes a description of a *GinzUI* that brings NILM event data to the user and provides rudimentary system-level diagnostics. Chapter Three concludes by discussing the importance of minimizing tuning parameters and suggests methods to eliminate or intelligently select them. These parameters have significant impact on software robustness and limit portability. Chapter Four discussed the actual field implementation of the diagnostic NILM system on the USCGC Escanaba and a summary of the deployment results. Chapter Five presents observations, future work and conclusions.

2. NILM Signal Classification – Methods and Strategy

This chapter outlines the development and implementation of the *ginzu* event classification software and an associated diagnostic graphical user interface. An ensemble of previously developed classification techniques and strategies were used to create this software package and will be introduced prior to the explanation of the *ginzu* algorithm (Chapter 3) or discussion of results (Chapter 4).

2.1 Shipboard Waste Collection and Disposal System (CHT)

To establish a context for the application of the *ginzu* software package, the vacuum assisted sewage collection and disposal system (CHT) is given as a candidate system. This system is present on the Famous Class USCG Cutters and has been analyzed in detail in previous works such as Mosman [12], and Piber [15]. A detailed discussion of the operating characteristics of this system is provided from page 20-24 of [12] and a review is provided here for the benefit of the reader.

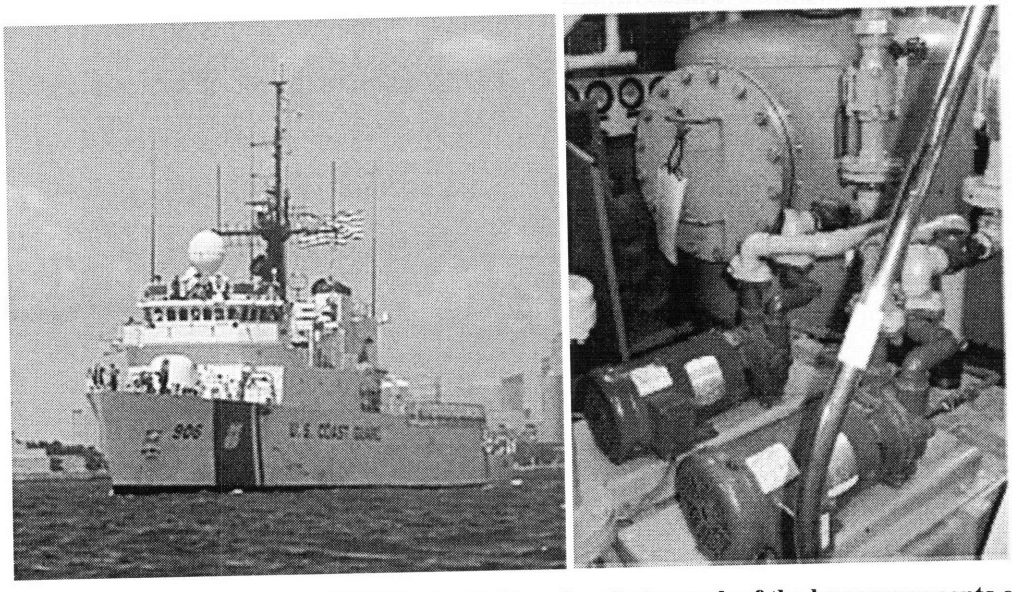


Figure 2-1 : A Famous-Class USCG Cutter (left) and a photograph of the key components of the shipboard vacuum assisted sewage collection and disposal system (CHT) (right)

The sewage collection and disposal system consists of a retention tank and pumps that are located in an auxiliary machinery space onboard the ship. The system receives the drains from eighteen vacuum toilets, two urinal lift valves, one urinal non-lift valve

and one galley garbage grinder. A 360 gallon collection tank stands upright with two 1.5 HP vacuum pumps connected to the top of the tank via piping and two check valves that function to retain the system vacuum pressure when the pumps are de-energized. The toilets, urinals and garbage disposer are zoned throughout the ship and lead into the top of the tank through isolation valves. A separate tank discharge system with two 2.0 HP pumps automatically drains the collection tank based on tank level. A controller interlock prevents the simultaneous operation of both discharge pumps. [12]

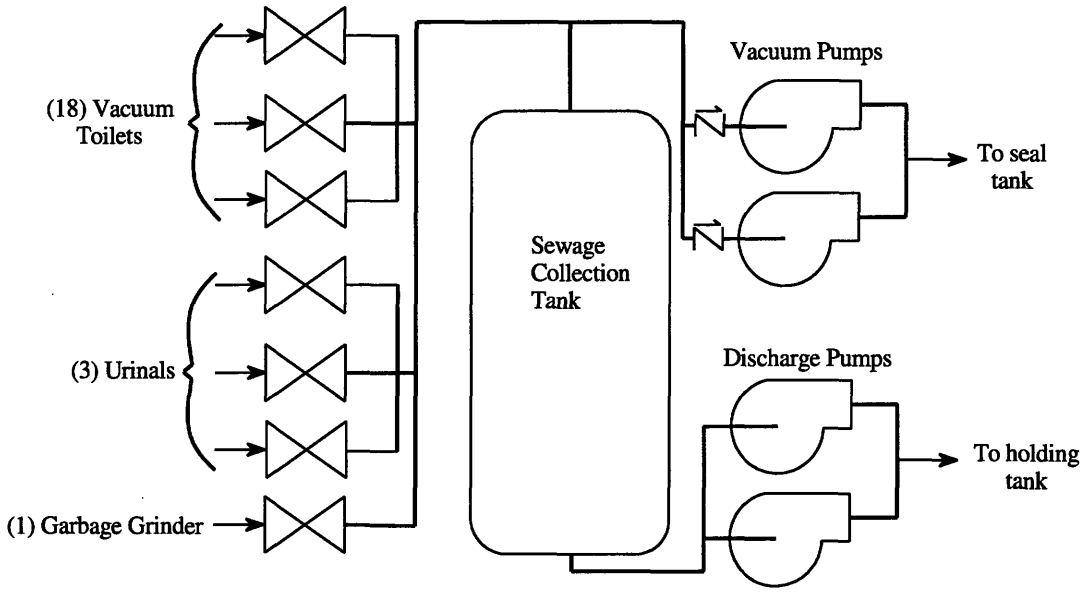


Figure 2-2 : One-line diagram of shipboard CHT system showing key components [12]

The vacuum pumps attempt to maintain vacuum in the system. When the system pressure drops to 14 in-Hg, one vacuum pump energizes. Consecutive starts alternate between pumps to equalize the wear. If the pressure drops to 12 in-Hg, the second vacuum pump starts to assist the already running pump. The pump(s) de-energize when the tank pressure reaches 18 in-Hg. [12]

An installed NILM monitors two phases of the 440 volt electrical power inside the pump controller and the current is measured on the third phase. Both the vacuum pumps and discharge pumps use the same power supply so their input voltages are the same. The current transducer was installed to measure the current passing to the four pumps collectively. That is, if both vacuum pumps were energized and one of the discharge pumps energized, the current sensed would be the sum of the currents to the

three individual loads. A typical power plot showing both a vacuum pump and the discharge pump is show in Figure 2-3. [12]

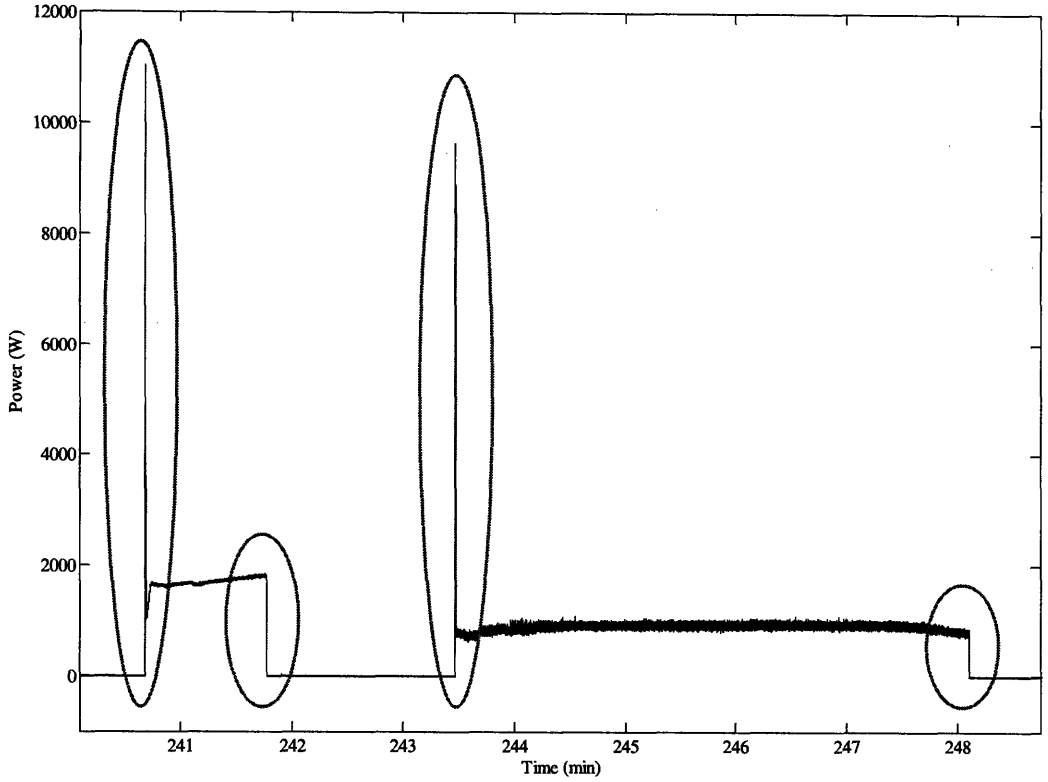


Figure 2-3 : Normal CHT system response showing the spectral power envelopes of the vacuum pump (left) and the discharge pump (right). System transients are circled. [12]

The nine minutes of data shown in Figure 2-3 contains four CHT system transients. In chronological order, these transients are the starting of vacuum pump induction motor, the off event of the vacuum pump motor, the starting of the discharge pump induction motor, and the off event of the discharge pump motor. In the past, diagnostic strategies involved the manual identification and classification of system events. Once events were located, the characteristics were examined to determine if system operation was normal or if the transient characteristics were unusual – representing symptoms of a system fault. If the identity of these transients can be determined in realtime (or near realtime) by an automated classifier, an appropriate set of diagnostics can then be applied to evaluate system health based on the characteristics and/or timing of the transients. Consequently, the ability to automate the classification of system power signals is a key prerequisite to realtime system diagnostics.

2.2 Classification Tools

Various methods have been applied to classify NILM signatures. In general, these methods attempt to identify system-specific events based on signal characteristics that make the event positively differentiable from other system-specific events. The methods implemented in the *ginzu* classification application compare the shape characteristics of a transient to shape characteristics of known events. Specifically, shape characteristics are defined as (1) the relative steady state power change across the transient event index and (2) the shape of the spectral envelope during the transient. These comparisons are aided by continuously tracking the finite state of the system (i.e. the running status of each known motor or electric component in the system) and limiting classification decisions to only those permitted by the associated finite state diagram.

2.2.1 Classification based on Shape (Steady State Power Change)

When a transient is detected, the change in real power (ΔP) and the change in reactive power (ΔQ) across the event can be calculated. As the loads in the monitored system will draw different levels of power during different modes of operation (e.g. on, off); the difference in power between these power levels may be compared to historical values to classify the specific event.

For example, Figure 2-3 shows the starting transient of a vacuum pump and a discharge pump. If these transients are assumed to be representative of typical vacuum pump and discharge pump transients, we can conclude that a vacuum pump should normally create a ΔP of approximately 1.8 KW and a discharge pump will normally result in a ΔP of roughly 0.9 KW. A similar calculation may be performed for the reactive power component and a range of expected ΔP and ΔQ can be determined for each system component. Since the vacuum pumps and discharge pumps are the only large electrical loads in the CHT system, the steady state power change across any system event may be compared to the historically expected values to determine event classification. Figure 2-4 illustrates this concept.

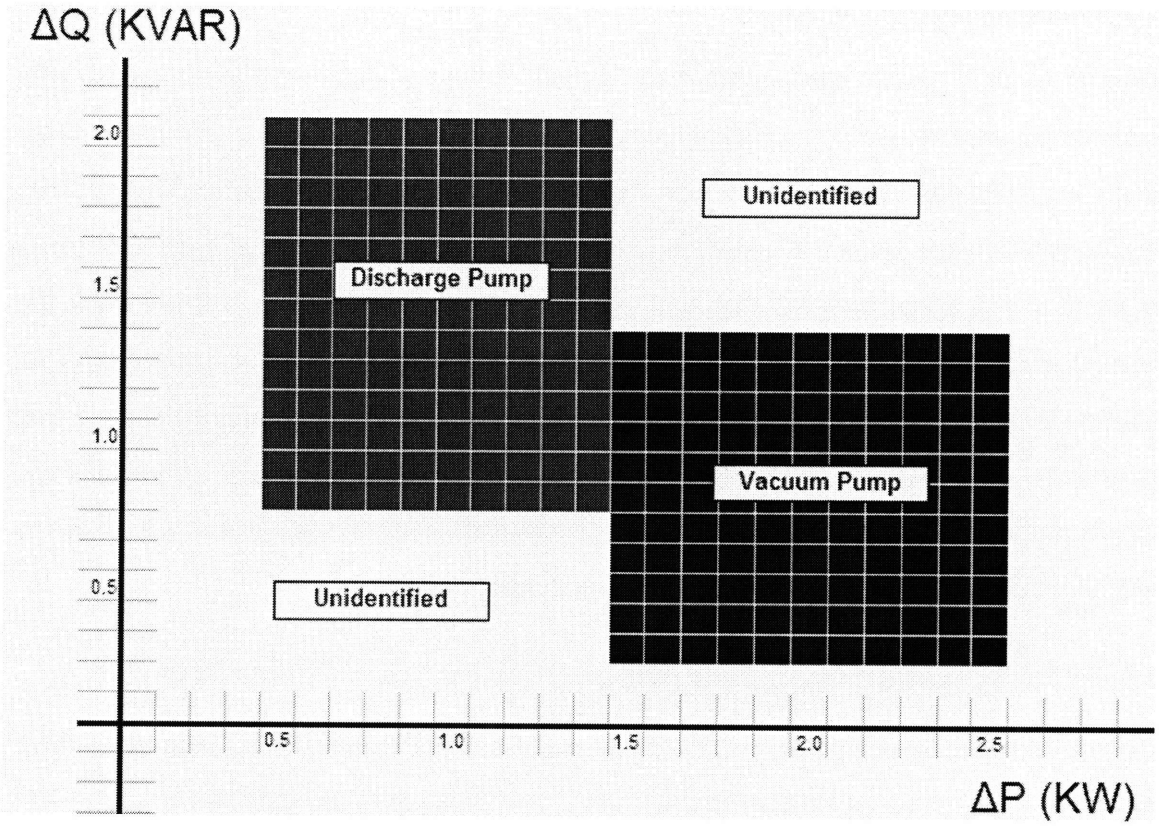


Figure 2-4 : Example of classification made based on the steady state ΔP and ΔQ across the transient event index. If the ΔP and ΔQ fall within the expected ranges, a classification is made. If the ΔP and ΔQ do not fall in expected ranges, the event is labeled as 'Unidentified'.

2.2.2 Classification based on Shape (Transient Pattern Matching)

Each system transient possesses a shape – although not necessarily a unique shape. Depending on the electrical component making the transient, the shape can vary from a simple step change in real power (such as an incandescent light) to a complex dynamic change in both real and reactive power (such as the starting transient of an induction motor).

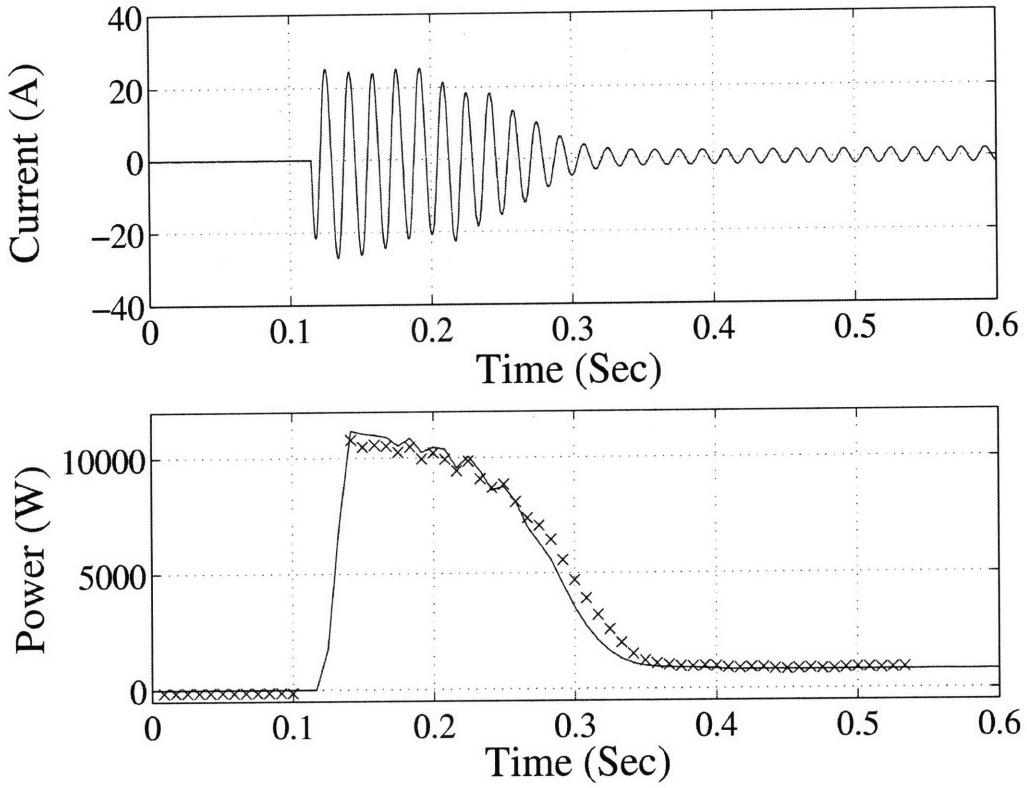


Figure 2-5 : NILM current signal converted to a spectral power envelope. The lower figure also shows a template shape that can be compared to the transient signal to identify a transient. [3]

By comparing the shape of a transient to the shapes of known system events, a numerical score (or correlation score) can be assigned based on the degree of similarity. This score can then be compared to a tuned threshold to determine a classification. This method is particularly effective in small systems where each piece of electrical equipment creates a unique power signature, which allows disparity across the events.

The method of least squares (LS) was applied to classify NILM signatures by Shaw in 2000 [16] and by Lee in 2003 [8]. LS was implemented in the *ginzu* classification algorithm to compare transient patterns to a library of historical shapes and the result of this comparison was then used to identify events. This method is described by Lee from page 52-57 in [8] and is reviewed here.

Method of Least Squares

Consider a vector **x** that contains data representing the current system power signal, and a vector **y** that represents an *event template* containing the power signature of a known

event – such as the starting transient of an induction motor. The goal of the calculation is to assign a value (correlation score) that indicates how similar the vectors are. The vector **y** may be multiplied by a gain factor and then shifted by a constant offset (**w**) to approximate **x**.

$$y = \alpha x + \beta y + w = Aa + w \tag{2.1}$$

where $A = [x \quad 1]$ and $a = [\alpha \quad \beta]^T$. The gain value (α) indicates the degree of scaling between the power data **x** and the known template **y**. To assign a correlation score, it is necessarily to identify the gain value that minimizes the error between **x** and **y** and the numerical value for the minimized error (**e**). Specifically, gain and offset are adjusted such that error (**e**) is minimized in:

$$\|e\|^2 = \|y - Aa\|^2 \quad . \tag{2.2}$$

In [17], Strang showed this equation is solved when:

$$a_{opt} = [\alpha_{opt} \ \beta_{opt}]^T = (A^T A)^{-1} A^T y \tag{2.3}$$

, where a_{opt} is the LS estimation of the error between **y** and **x**. It is comprised of the AC gain (α) and DC offset (β) that minimize error. Once these values are determined, this minimum residual error (γ) can be calculated and normalized by the size of the template vector (M).

$$\gamma = \frac{1}{M}\|y - Aa_{opt}\|^2 \quad \text{,where } a_{opt} = [\alpha_{opt} \ \beta_{opt}]^T \tag{2.4}$$

Since the resulting gain value (α_{opt}) may be greater than 1.0, it is normalized before being used as a measure of comparison.

$$\alpha_n = \begin{cases} 0, if \ \alpha < 0 \\ \alpha_{opt}, if \ 0 < \alpha \le 1.0 \\ 1/\alpha_{opt}, if \ \alpha > 1.0 \end{cases} \tag{2.5}$$

where α_n represents the normalized gain that minimizes the residual error between the power data **x** and the template **y**.

In practice, a short offset in time may exist between the template and the detection index, so the above calculation is repeated by shifting the incoming signal a nominal number of indexes in both directions and recording the minimized error during each shift. The *index of optimal shift* is then located by selecting the index where the lowest residual error (γ) is attained.

To arrive at a final correlation score, both the normalized gain and the residual error must be taken into account. In [8], Lee provides the *composite gain formula* (equation 3.5). This equation rescales the normalized residual error and multiplies this value by the normalized gain. The composite gain represents the correlation score that is used for determining if a template match is sufficient to classify an event.

$$correlation\ score = composite\ gain = \alpha_n (1 - \frac{(1-\alpha_{thresh})}{\gamma_{thresh}} \cdot \gamma) \tag{2.6}$$

where $0 \leq \gamma < \gamma_{thresh}$, and $\alpha_{thresh} < \alpha_n \leq 1.0$. Since these threshold values normalize the composite gain, they must be specifically tuned for every *event template*. The limits of the composite gain formula are important to recognize when tuning the residual threshold value (γ_{thresh}). If the residual threshold is set excessively high, the composite gain becomes insensitive to residual error and simply becomes the normalized gain value. As the residual threshold approaches zero, any amount of residual error will drive the composite gain negative – resulting in over-sensitive scoring.

Once the correlation score is calculated, it can be referenced to a tuned threshold to determine if the similarity between the power signal and the template is sufficient to classify the event. The *Ginzu* application provides a C-based implementation of LS. This is accomplished through the use of Numerical Recipes [13] for required matrix manipulations and the hard-coded entry of four induction motor *event templates* (i.e. CHT Vacuum Pump, CHT Discharge Pump, RO Low Pressure Pump, and RO High Pressure Pump).

2.2.3 System State

Many systems can be modeled as finite state machines (FSM) [6] where system state is the operating status of all system components at any given time. Once the possible states of a system are defined, the system operation can be reviewed to determine typical state

transitions. For example, a simple system such as a resistive heater could be defined by two states – on and off. A state transition could then be detected by recognizing a change in real power usage. This simple example is shown in Figure 2-6.

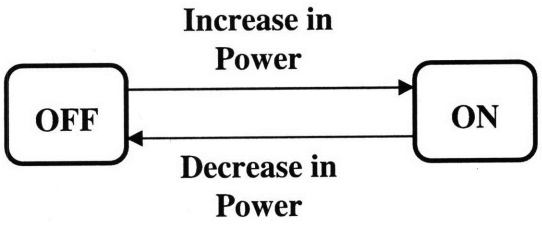


Figure 2-6 : Simple Finite State Machine for a resistive heater. State transitions are recognized as changes in power usage.

This approach can be applied to the CHT system. The CHT system consists of two vacuum pumps and two discharge pumps. If there were no operating restrictions placed on the CHT system, any of the pumps may be running at any given time. Figure 2-7 shows all possible CHT states with no restrictions.

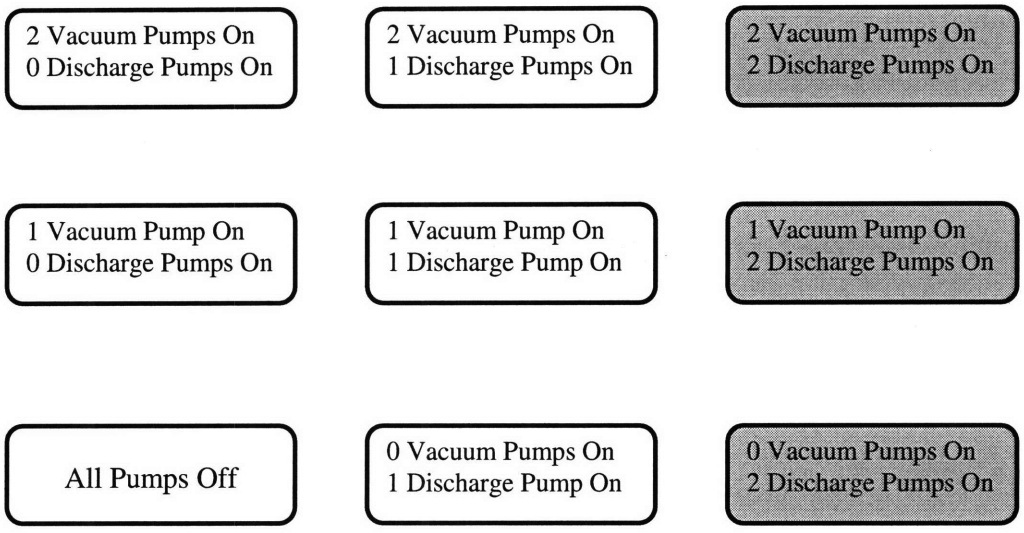


Figure 2-7 : Possible finite states of CHT System. The states with more than one running discharge pumps (shown in grey) are prevented by a controller interlock. Since the expert classifier understands these state are not allowed, the classifier may prevent classifications that would result in a 'not allowed' state.

An interlock in the CHT controller prohibits the simultaneous operation of both discharge pumps. With the aid of a system expert, a classifier can recognize that the CHT system controller prohibits states with two running discharge pumps and can use this information as either a classification tool or a system diagnostic. As a classification tool, the classifier may reject any classification that would place the system in a 'not allowed' state, which could prevent illogical classifications. Alternatively, this information can be diagnostic. If an event classification places the finite state machine in a 'not allowed' state, the operator can be informed that the system is not working correctly.

Once the allowable states are defined, a human expert can identify the transitions that occur during normal system operation. By identifying the most likely state transitions and tracking the current state of the system, the classifier can be tuned so that the most likely transition are given additional consideration. Figure 2-8 shows the possible CHT states but adds common transitions.

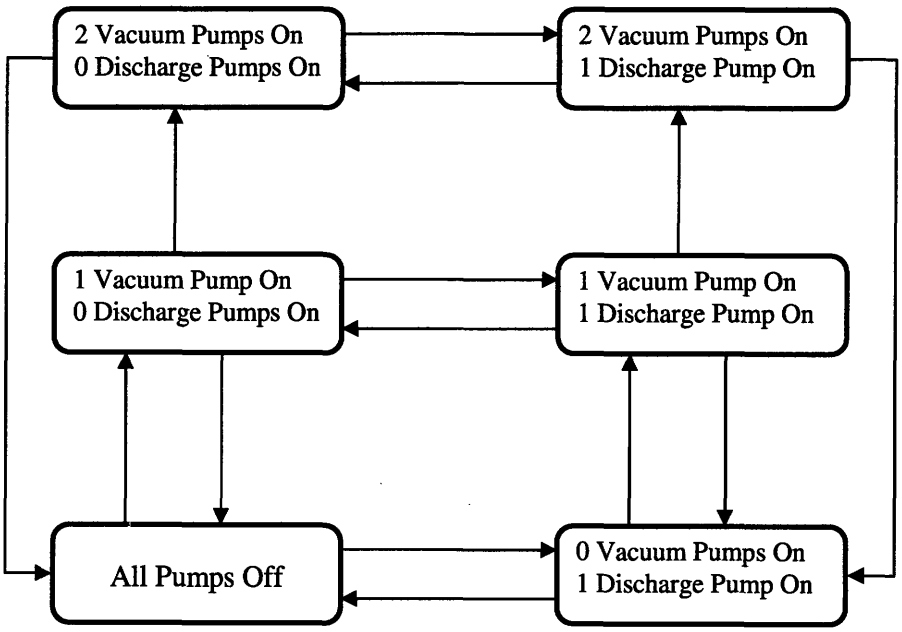


Figure 2-8 : FSM for Shipboard CHT System showing the most common state transitions.

Since the number of possible state changes in a small NILM system is often limited, *a priori* knowledge of the system's composition is a very valuable tool when predicting the classification of an event. For example, if an ON event is detected (based

on positive ΔP) while system state indicates that both vacuum pumps are already running, the event can not be a vacuum pump ON. In fact, the system diagram shows that the only event that can cause a positive ΔP is a discharge pump ON event. The state information can be combined with power change information to create accurate classifiers. Consider an event where post-event power is approximately zero. If the pre-event state was known, the event can immediately and accurately be classified as the OFF of each component that was ON before the event.

State Verification and Correction

While state information provides a valuable classification aid, any misclassification will place the classifier in an incorrect state. If this incorrect state is used to make consequent classification decisions, a single failure can cascade. In order to minimize the impact of a misclassification, state verification and correction is implemented so that erroneous states are corrected before they can affect later classifications. State verification and correction is accomplished by comparing current power levels to power levels that would be expected for the current state. If the power level does not match the expected power level, the state information is corrected – typically by guessing the state based on the current power level.

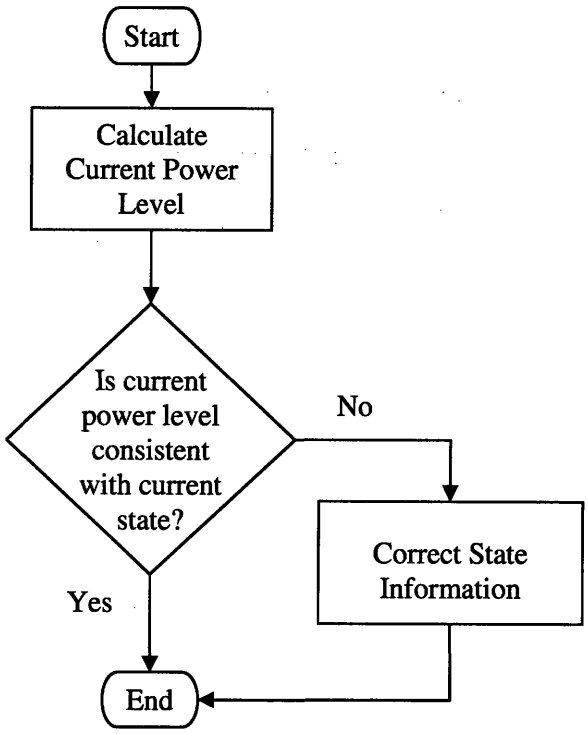


Figure 2-9 : State Verification is performed by comparing current power levels to power level that would be expected for the current state. If the power levels are inconsistent, the state information is corrected.

2.3 Classification Strategy

Once the classification methods are defined, they can be implemented in a decision tree framework (Figure 2-10). As a realtime stream of power data is received, it is searched for events that cause the signal to change rapidly as a function of time. If a transient is detected, a decision tree is invoked that asks a series of questions to classify the event. Once the classification decision is made, the information is passed to the user in a useful format.

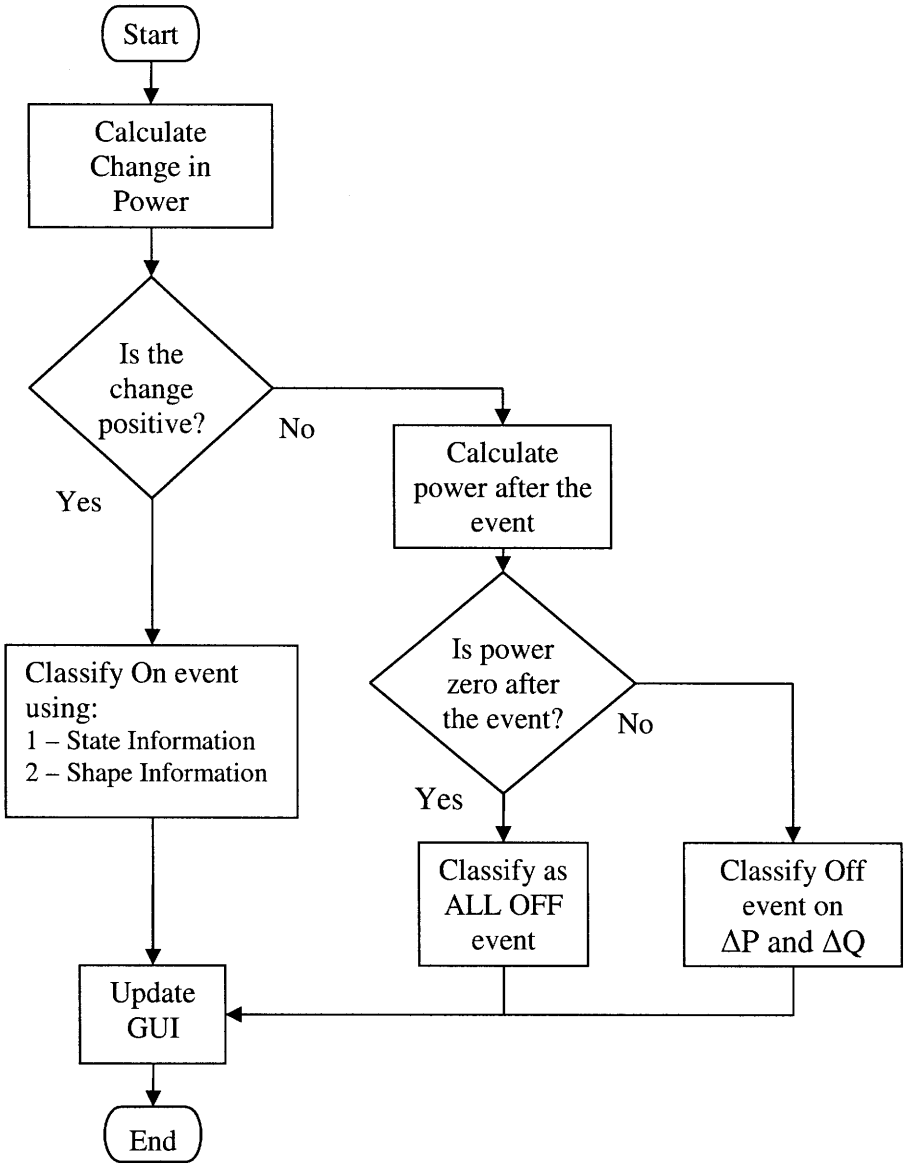


Figure 2-10 : Decision Tree Framework for event classification in shipboard systems. This sequence of decisions may be applied to classify any typical shipboard power transient.

A standard set of questions follows.

1. **Is the steady state change in power (ΔP) positive or negative?** This question determines the follow-up questions should be used in the classification. A positive **ΔP** (or up power event) will drive the classifier to search for ON events. A negative **ΔP** (or down power event) will focus the classifier on OFF events.
2. **If the ΔP is positive, do the shape of the event and the magnitude of the power change correspond to any events that would be expected based on the**

current system state? If the shape and magnitude match the shape and magnitude of a known event, classify the transient accordingly. If no match is found, label the event as 'unclassifiable' or guess based on what the most probable event should be.

3. **If the ΔP is negative, does the power after the event equal to zero?** If the post-event power is zero, the state information is checked and any loads that were running before the event are set to OFF.
4. **If the ΔP is negative and the post-event power does not go to zero, does the magnitude of the change correspond to any known event?** If the change matches the change of a known OFF event, classify it accordingly. If it does not, label the event as 'unclassifiable' or guess based on what the most probable event should be.

The classification framework illustrated in Figure 2-10 can be combined with the state verification and correction approach discussed in section 2.2.3 – resulting in a robust expert classifier that is capable of classify system events and recovering from misclassifications. This general approach was implemented in the CHT system classifier and the RO system classifier and is discussed in detail in Chapter 3.

3. Ginzu

Chapter Three provides a functional overview and code review of the NILM event classification software (*ginzu*). The application was built using the methods and strategy prescribed in Chapter two. *Ginzu* was developed in *Microsoft Visual C++ 2005 edition* within the *Windows OS*. It was then ported to *Ubuntu Linux 7.10* and compiled as a C application using the *gcc* compiler[2]. The *ginzu* application is an evolution of previously developed code written by various NILM authors (Leeb, Shaw, Lee, Cox, Paris, et al). The majority of the application and Matlab User Interface was developed between July and December of 2007. The version of *ginzu* described in this chapter is the final release version of *ginzu*. This final version has been modified slightly from the beta release that was deployed on the USCG Cutter Escanaba in January 2008. The modifications include minor bug fixes and improved classification logic (described in 4.3 Resulting Changes).

This chapter also describes the associated graphical user interface (*GinzUI*). This diagnostic user interface receives information regarding event classifications and times, and uses this information to execute a series of rudimentary diagnostics. Diagnostic graphical user interfaces were created for the CHT system and the Reverse Osmosis system. The CHT system GUI is described in this thesis. The Reverse Osmosis GUI is described by P. Branch in [1].

3.1 Ginzu Overview and Functions

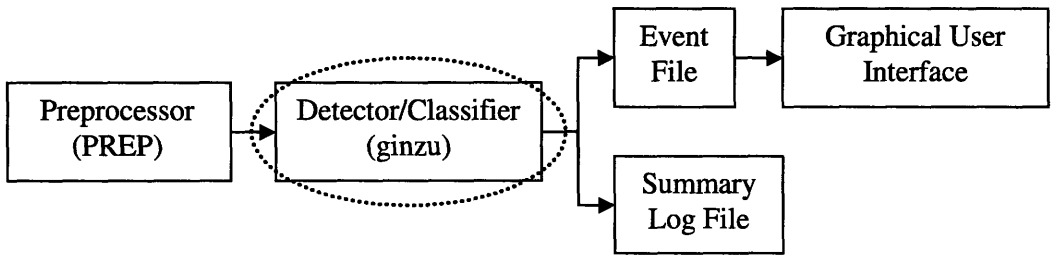


Figure 3-1 : *Ginzu* Classifier software showing software system interfaces. *Ginzu* processes an incoming stream of power data to generate *event files* and a verbose log file.

[2] Note ginzu requires the math libraries and must be compiled using the '-lm' extension.
(gcc ginzu.c –o ginzu –lm)

The *ginzu* application acts as an interface between the existing NILM preprocessing software (prep) and the diagnostic Graphical User Interface (Figure 3-1). The functions of *ginzu* are (1) to receive streaming preprocessed power data, (2) to search the incoming data for transients, and to (3) attempt to classify the transients as system-specific events. To provide feedback of a system event, *ginzu* generates *event files* that represent transients. A single *event file* contains relevant information such as a time of the event, classification of the event, relative index of the event within the dataset, and a ten second snapshot (1200 points) of power during the event. Figure 3-2 shows an example of an *event file.*

```
Thu Dec 06 12:28:12 2007                    (timestamp)
/home/nilm/snapshot.txt                      (filename)
109278                          (global index of event)
438              (relative index within event window)
Vacuum Pump Off                 (Classification string)
002                                 (Classification ID)
1200            (# of P, Q data points in event file)
1.783989                  (Start of P data - 1st point)
1.782211
…
0.002534
0.006264                (End of P data - 1200th point)
0.858268                  (Start of Q data - 1st point)
0.856385
…
0.007835
0.006428                (End of Q data - 1200th point)
```

Figure 3-2 : Sample *Event file* Format

This *event file* is created and then read by the GUI to inform the user that an event has occurred. Additionally, the data contained in the *event file* may be analyzed by the GUI to perform various system diagnostics. Since the *event files* are small (20kB) in comparison to large uncompressed raw datasets (40MB), they can be loaded by a supporting GUI very quickly and provide almost immediate feedback. G*inzu* also generates a verbose log file as it attempts to detect and classify. This log is printed to the terminal as the default but is generally redirected to a text file. This summary log file does not interface with the normal user but is a valuable tool for troubleshooting and development.

The *ginzu* executable may be invoked from the command line and is controlled with command line arguments as shown in Table 3-1. A user friendly *ginzu* user's guide is provided in Appendix [A]. The stable source code used in the Escanaba deployment is included in Appendix [D].

Argument	Title	Description
m [0-3]	Mode	Sets the Mode of *Ginzu*. (default 3) 0 – Detect and classify using CHT Classifier 1 – Detect and classify using RO Classifier 2 – Detect. Do not classify. 3 – Streaming window mode.
l [0,1]	Load	If [Load=0], *Ginzu* reads incoming PREP via piped input. Example: dd if=snapshot.txt | *ginzu* l 0 If [Load=1], *Ginzu* will attempt to load a specified file. The file must be specified with the 'f' parameter listed below. Ex: *./ginzu* l 1 f snapshot.txt
f file.txt	Filename	If [Load=0], the f argument defines the naming string of the resultant *event files*. If no file is specified with piped input, *event files* will be named snapshot-*. If [Load=1], *Ginzu* will attempt to load the specified file. The file that will be loaded is specified with the f argument.
v [0,1]	Verbose	If [Verbose=1], *Ginzu* will provide a detailed output log of its operation. This output should generally be re-directed to a file. (default is off)
e [1-10]	Window Period	Defines how often an *event file* will be created in streaming mode. This setting has no effect if not running in MODE 3 (streaming mode). (default is 2 seconds)
t float	Change-of-mean Threshold	Sets the threshold for the change-of-mean detector. This has no effect if not running in MODE 2 (detect only). (default is 0.4)
p float	P Scale	This setting scales the incoming real power data into Watts. The scaling is a function of NILM hardware (specifically, the choice of CT and measuring resistor sizing) (default is 0.62 based on existing CHT data)
q float	Q Scale	This setting scales the incoming reactive power data into VAR. (default is 0.62 based on existing CHT data)
d int	Data rows in the incoming dataset	This allows the user to specify the exact number of prep data rows in the incoming stream. *Ginzu* will exit after reading d rows of data. (Default is 432,000 rows that represents 1 hour of preprocessed data – 120 rows/sec x 60 sec x 60 min)
r int	Real_time	This parameter informs *ginzu* that it is being run on realtime or archived data that determines what time is used in *event file* creation. If [Real_time=1], timestamp are generated based on the computer clock. If [Real_time=0], timestamps are generated by parsing the filename of the archive.

Table 3-1 : Summary of *Ginzu* Command Line Parameters

3.2 Detect-Classify-Verify Program Flow

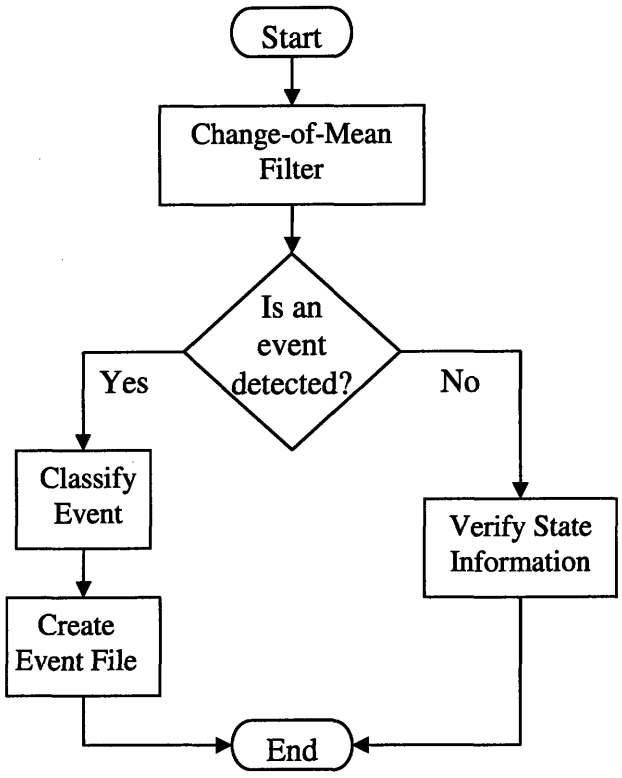


Figure 3-3 : Simplified Classifier Program Flow showing Detect-Classify-Verify loop. This sequence of events is implemented on every incoming power window.

Basic program flow consists of a Detect-Classify-Verify loop. The algorithm initializes by loading a 10 second data window consisting of real and reactive power. This window is passed to a detection algorithm that locates the indexes where a rapid change in power has occurred. These indexes represent system transients and are candidates for classification.

If an event is detected, the classifier may be called. The classifier implements a hierarchy of classification decisions to make a 'best guess' based on the relative power levels around the event, the state of the system prior to the event, and if possible, the correlation between the shape of the power signal during the event and the shape of a known library of events. If a rapid power change is not detected, a *state verification and correction* function is called. This function provides a number of features but specifically attempts to verify that the current power levels are consistent with what the *ginzu*

algorithm believes they should be for the current system state. The algorithm then reads an additional period of data from the input buffer. This data is inserted into the P/Q buffer and the older time is discarded. This new power window is then passed to the detect-classify-verify loop and the cycle is repeated.

3.3 Event Detection

In the default operating mode, the preprocessor samples the power at 120 Hz so the ten second data windows are represented by two 1200 index arrays – one containing real power and one containing reactive power. The 1200 index power array is passed to a detection algorithm that determines where rapid power changes are located. This is accomplished by passing the array through a change-of-mean filter that calculates the difference between the original power signal and the output of a low pass filter. The result is a processed signal that only contains rapid power changes. The filter constants are hard-coded in the *ginzu* header and do not change with application.

The filtered signal is then passed to a comparator that identifies any array index where the signal exceeds a preset detection threshold. Three 1200 index arrays are initialized to further process the signal: *Compare, Jump, and Detect.* The 1200 index *Compare array* is set to **1** wherever the output of the change-of-mean filter exceeds a tuned threshold.

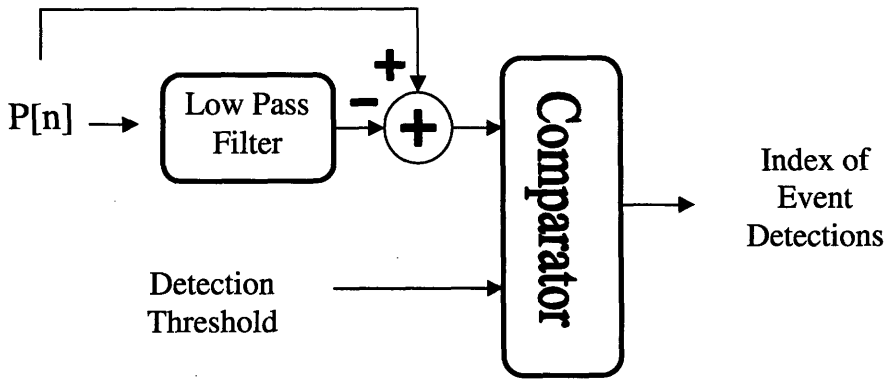


Figure 3-4 : Change-of-Mean filter

The *Jump array* is then calculated as the digital derivative of the *Compare array* and the absolute value is taken. Jump will contain a **1** at the indexes where Compare

changes from 0→1 or from 1→0. Since the rapid power changes associated with a single event tend to cause clusters of detections, the algorithm establishes a short duration lock-out window[3] that prevents redetections from being passed to the classifier as new events. The final array, *Detect*, contains the index of events that are considered for classification.

Since the incoming signal contains the real power (P) and reactive power (Q) components, either (or both) may be useful for transient detection. Practical application has shown that real power provides the best indication of the actual work that the system is doing – but tends to contain more noise and variation than reactive power. Conversely, reactive power tends to be very 'clean' but may contain less useful data. In addition to locating the index where a transient has occurred, the detect algorithm calls load_index() to determine if the detection requires classification. The load_index() function provides critical updates by determining if the current detections need to be added to the **global** event data. *Ginzu* maintains the following key global variables as global event data:

Type	Name	**Function**
int	event_index[n]	Provides the global index of the n^{th} event in the incoming data stream. This number typically ranges from 0 to 432,000 corresponding to the number of potential lines in the incoming dataset.
Boolean	event_class_status[n]	This Boolean corresponds to the classification status of the nth event. It is set to **1** when the event is classified.
char	*Class[n]	This string is the text classification of the n^{th} event.
int	Class_ID[n]	This integer represents the classification code of the n^{th} event.
Boolean	EVC[n]	The 'Event file created?' Boolean is set to **1** if the event has had an event file created for it.
int	event_count	This integer is the current number of events in the global event data.
int	local_index[m]	This integer represents the index of the m^{th} event in the current detection window. This number typically ranges from 1 to 1200 corresponding to the 1200 points in a detection window.
int	local_detect	This integer represents the number of events in the current window that require classification.

Table 3-2 : Global Event Data

[3] This duration is set as 153 indexes (or approximately 1.25 seconds) in the ginzu application. This is defined as one second (120 indexes) plus the length of the filter vector (33 indexes).

The load_index() function is not trivial as detections may be any of the following.

- New events that have not been previously detected as they have just entered the detection window.
- Events that have been previously detected but were not classified as insufficient data was available to determine the change in power.
- Events that have been previously detected and classified, but will continue to be detected until they exit the detection window.
- Events that are located in a lock-out window.

During each window, the classifier determines the number of events that require classification (local_detect). If these events exist, the window is either passed to a classifier or, if no classifier is specified, passed directly to the *event file* creation function (create_evt_file).

3.4 Event files and Naming Conventions

When a positive classification is made, the window and the classification ID are sent to create_evt_file() and an *event file* is created in the directory where *ginzu* was invoked. *Event file*s are created with a specific format (Figure 3-2) and standard naming convention.

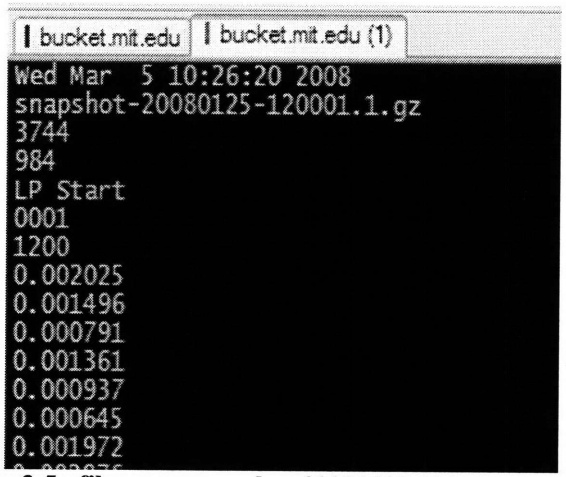


Figure 3-5 : filename: snapshot-20080125-120001-0014-0001

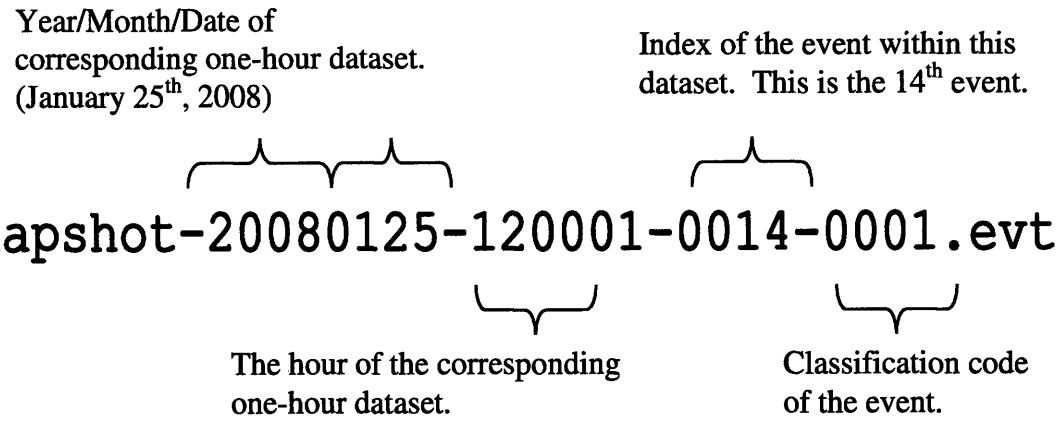


Figure 3-6 : Standard event file format showing information contained in event file title. This example is shown for the scenario where ginzu is run on an archived dataset. The Year/Month/Date/Hour corresponds to the date of the archived dataset. In the realtime scenario, the Year/Month/Date/Hour correspond to the time that *ginzu* was invoked.

Depending on whether the *event file* was (1) generated while running *ginzu* on realtime data (i.e. a realtime pipe of streaming preprocessed power data) or (2) generated by running *ginzu* on an existing archived dataset, the filename will have a slightly different meaning (Figure 3-6). In both cases, the name begins with the Year/Month/Date of reference and is followed by the Time of reference. If the *event file* is created on a realtime stream, this data corresponds to the date-time that the specific instance of *ginzu* was invoked. If *ginzu* is run on archived data, this string is generated by parsing the archive filename, which contains the same date-hour information. The archive name must be properly formatted per the naming convention outlined in [6].

As shown in Figure 3-2, the actual time of the event is contained inside the *event file*. In realtime applications, this is the time that the *event file* was created, which is the actual time that the event occurred shifted by a short processing time[4]. In archived applications, the actual event time is determined based on the archive filename (which contains the hour) and an offset that is calculated by the index of the event within the archive file (assuming a 120 sample/second data rate). The fourth column of data represents the index of the event within the incoming dataset. This number begins at zero and as incremented with each *event file*. The final column is the classification

[4] In the CHT system, the data is delayed by six seconds to calculate post-event power levels. In the RO system and in 'detect only' mode, the data is approximately two seconds late.

identification code, which varies depending on the system classifier that is specified. Figure 3-7 provides an example of a Linux directory containing multiple *event files*.

```
proper@bucket:~/ginzu$ ls snap*20080130-140*
snapshot-20080130-140001-0000-0001.evt  snapshot-20080130-140001-0013-0005.evt
snapshot-20080130-140001-0001-0007.evt  snapshot-20080130-140001-0014-0002.evt
snapshot-20080130-140001-0002-0002.evt  snapshot-20080130-140001-0015-0005.evt
snapshot-20080130-140001-0003-0006.evt  snapshot-20080130-140001-0016-0002.evt
snapshot-20080130-140001-0004-0004.evt  snapshot-20080130-140001-0017-0005.evt
snapshot-20080130-140001-0005-0005.evt  snapshot-20080130-140001-0018-0002.evt
snapshot-20080130-140001-0006-0002.evt  snapshot-20080130-140001-0019-0005.evt
snapshot-20080130-140001-0007-0005.evt  snapshot-20080130-140001-0020-0002.evt
snapshot-20080130-140001-0008-0002.evt  snapshot-20080130-140001-0021-0005.evt
snapshot-20080130-140001-0009-0005.evt  snapshot-20080130-140001-0022-0002.evt
snapshot-20080130-140001-0010-0002.evt  snapshot-20080130-140001-0023-0005.evt
snapshot-20080130-140001-0011-0005.evt  snapshot-20080130-140001-0024-0002.evt
snapshot-20080130-140001-0012-0002.evt  snapshot-20080130-140001-0025-0005.evt
proper@bucket:~/ginzu$
```

Figure 3-7 : Example directory contents showing naming convention

3.5 NILM Classification of Waste Disposal (CHT) System Events

The CHT system represents a common shipboard auxiliary system used to transfer sewage from installed heads to a sanitary tank where it then pumped overboard. The system operation and performance has been detailed in 2.1 Shipboard Waste Collection and Disposal System (CHT) and in [12, 15]. NILM has been monitoring CHT systems on USCG vessels since 2003 and various problems have been detected and corrected through the application of NILM signal analysis. As the CHT system is relatively simple and tends to have recurring failure modes, it was an attractive candidate for realtime NILM diagnostics.

3.5.1 CHT Event Detection

CHT event detection is performed using the methods described in 3.5.1 CHT Event Detection. Event windows requiring classification are passed to the CHT Classifier. Windows that do no require classification are passed to the CHT State Verification algorithm.

3.5.2 CHT Classifier Implementation and Outputs

The CHT Classifier is invoked from the *Ginzu* shell if mode 0 is specified via command line arguments (e.g. **dd if=sample_file.txt | ./ginzu m 0**). During normal CHT system operation, the majority of events will be classified as one of the following:

Description	ID CODE
Vacuum pump ON	(001)
Discharge pump ON	(003)
Vacuum pump OFF	(002)
Discharge pump OFF	(004)
ALL pumps OFF	(009)

Table 3-3 : CHT Classification Codes

In addition to these normal classifications, the classifier may also generate the following outputs:

Description	ID CODE
Vacuum pump ON, but indicates low running power	(005)
Vacuum pump with low running power has turned OFF	(006)
Unclassifiable event (probable noise)	(000)
Unidentified ON event	(007)
Unidentified OFF event	(008)
Vacuum pump cycling [5]	(021)
Discharge pump cycling	(043)
Cycle of Unidentified Load	(087)
2 Vacuum pumps ON	(011)
Vacuum pump ON and Discharge pump ON	(013)

Table 3-4 : Non-Standard CHT Classification Codes

[5] A 'Cycling' load is a load which turns on but immediately turns off during the same event window. This occurs when a template match indicates a specific type of pump has started – but no relative power change occurs across the event.

3.5.3 CHT Classifier Logic

When a 10 second event window is received, the classifier immediately determines if the event is positioned in the window such that at least five seconds of post-event power are available. This abnormal delay is a unique aspect of the CHT classifier that is specifically required for the classification of the 'clogged' vacuum pump event – an event where the pump fails to draw vacuum due to a clog in the priming line. This event is characterized by a 'normal' shaped vacuum pump start followed by the pump failing to maintain some expected running power level [15]. Figure 3-8 provides an example of a CHT event shown by the *GinzUI* ViewerUI. Note the window contains ten seconds of power data. The first arrow indicates the location of the event. The second arrow shows the end of the lock-out window (3.3 Event Detection). The Viewer also shows the time and classification of the event, which are available in the *event file*.

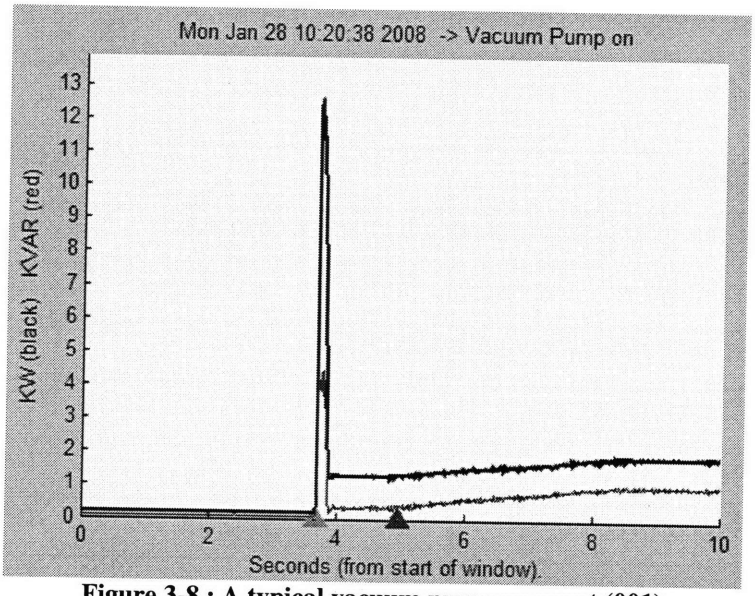


Figure 3-8 : A typical vacuum pump on event (001)

If the event is located too early in the event window, no classification is performed. Since the sliding window advances two seconds during each detect cycle, the event will be re-detected and returned to the classifier. When the event is correctly positioned in the window, the classification resumes with a calculation of the change in ΔP and ΔQ. This value determines if the classifier should be considering ON events, OFF events, or if there was no 'significant' power change across the event. It is accomplished through a single static window that calculates the difference between the

subsequent 0.25 seconds of real power data and the preceding 0.25 seconds of real power data (while neglecting the 0.04 seconds around the event).

Ultimately, this also requires the definition of a Noise Threshold tuning parameter that defines the value that is considered to be a significant change. In the CHT classifier, a value of 0.25 KW provided adequate to identify noise detections. Since the change in power calculation rejects the power data within 40 milliseconds of the event index, it is impossible to distinguish 'noise' from very short-term cycling loads (i.e. a vacuum pump or discharge pump that starts and secures within ~80 milliseconds).

Classifying CHT ON Events

The CHT system is composed of few major electrical components and these components demonstrate unique starting transients [12]. This makes the system ideally suited for template matching. However, before the correlation scores are calculated, the window is checked for additional (multiple) events. Multiple event windows are capable of causing undesirable results in the template match algorithm as the starting transients can interfere and distort.

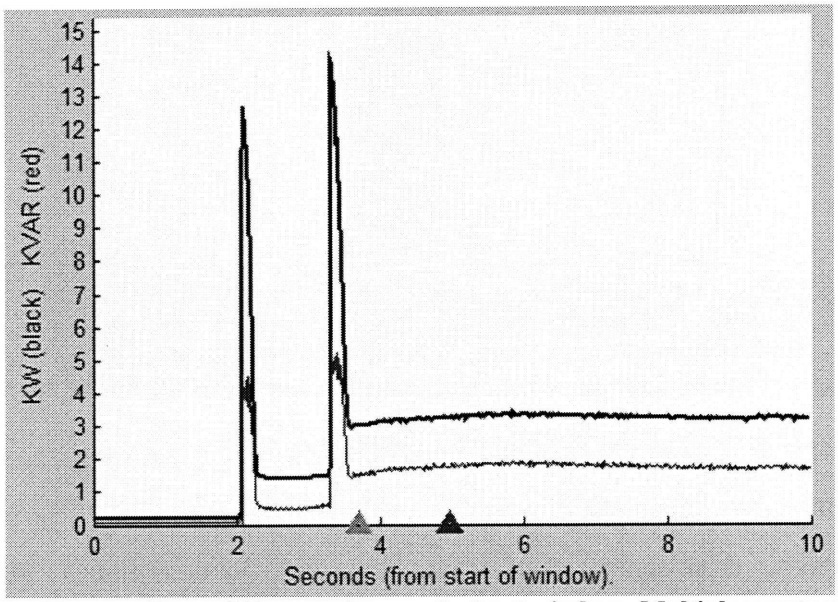


Figure 3-9 : Two Vacuum Pump ON events in same lockout window. Multiple events can complicate shape matching algorithms as shape can combine and interfere.

Figure 3-9 shows an example of a detection window where multiple events occur. In this specific example, the second pump start is 'locked' within the lockout window of the first pump start, so they are not identified in different *event files*. Lee [8] provided a method

to classify multiple events. This is accomplished through the calculation of the numerical derivative of the windowed data – and a determination of the number of times that the numerical derivative exceeds a tuned threshold. If this threshold is exceeded, indicating multiple events, the correlation score is not calculated; instead the event is 'arbitrated' by calculating the ΔP and ΔQ across the lockout window. These values are compared to the expected change for various pump start combinations and may result in the classification of the multiple ON events (011, 013). *Ginzu* implements this approach in the functions search_lockout_window() and arbitrate_event_overlap().

Once the double events are eliminated from consideration, the remaining event should match a template. The CHT classifier uses two 30 index templates – one representing a 'normal' vacuum pump start and one representing a 'normal' discharge pump start (Figure 3-10). These pump starts were chosen based on personal observation as being representative of the expected starting response. An alternative approach could be the use of some statistically averaged shape of 'well-shaped' pump runs.

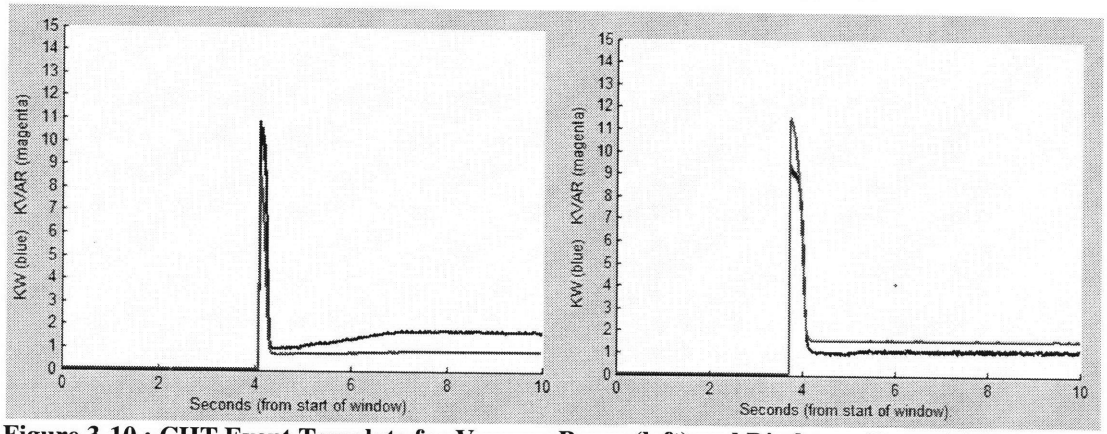


Figure 3-10 : CHT Event Template for Vacuum Pump (left) and Discharge Pump (right). These event templates are used for shape matching using the Least Squared Fit (LS) approach.

The CHT *Ginzu* Classifier performs the LS fit (see 2.2.2 Classification based on Shape) with Residual Threshold (γ_T) = 2.0 and Gain threshold factor (α_T) = 0.60. The correlation score is then compared to a tuned threshold to determine if the degree of similarity is sufficient to assign a classification. In order to arrive at an acceptable value for this threshold, the *Ginzu* classifier was alpha tested on existing CHT datasets.[6] A final value of 0.45 was set for both vacuum pump and discharge pump thresholds. If neither

[6] Seneca 2003, Seneca 2006, Escanaba 2007

correlation score achieves the 0.45 required to classify, the event is labeled as an 'unknown ON (007)'.

If the event is classified as a vacuum pump, the power level at five seconds (600 indexes) after the detection index is checked to determine if the vacuum pump has attained a normal value for running power. Observations of vacuum pump runs indicate that ~1.6 KW is drawn continuously if the pump is operating properly. If the running power is less than 50% of normal running power, the vacuum pump is considered to be clogged.

Finally, the ΔP across the window is calculated. If the final change in power is negligible (tuned to less than 0.10 KW[7]), the classified load is labeled a 'cycling' load and the classification is changed to indicate that the pump motor also turned OFF. In this case,

- A vacuum pump ON event (001) ➔ cycled vacuum pump event (021).
- A discharge pump ON event (003) ➔ cycled discharge pump event (043).
- An unknown ON event (007) ➔ cycled unknown event (087).

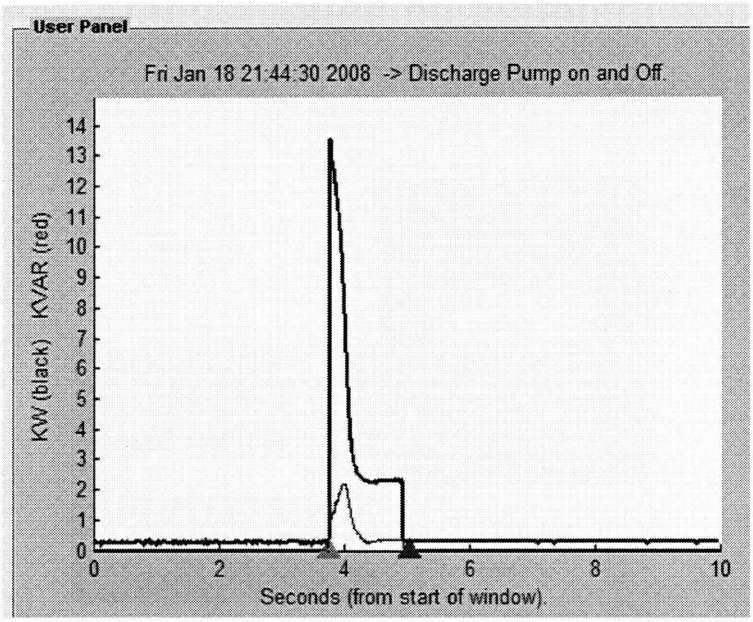


Figure 3-11 : Example of a cycling Discharge Pump event (Escanaba 18JAN2008). Pump motor demonstrates an expected start shape but secures within two seconds.

The classifier then calls the *event file* creation function and exits.

[7] This is 0.10 KW as referenced to the normal zero level. In Figure 3-11, the normal zero level is ~ 0.25 KW, so the zero determination is actually < 0.35 KW.

Classifying CHT OFF Events

The classification of OFF events is often simple as it common for only one CHT load to be operating at a given time. As a result, many OFF events results in a zero (or approximately zero) power condition, which allows the classifier to immediately base its classification on pre-event state. Typically, the ALL OFF event (009) is a very common CHT classification.

One of the most difficult (and inaccurate) CHT classifications tends to be the OFF event where power does not go to zero. Since the step-shaped change provides no useful shape information (except magnitude) and the system does not restrict any state change, the classification is completely based on relative power change. These classifications tend to be inaccurate as the normal power change for a discharge pump OFF and a vacuum pump OFF tend to vary greatly with system operation. If the bounds of this classification are relaxed, they tend to overlap – and if they are restricted, the power changes tend to fall out of the required bounds. This can result in misclassification or no classification. Figure 3-12 shows the tuned boundaries for OFF event detection.

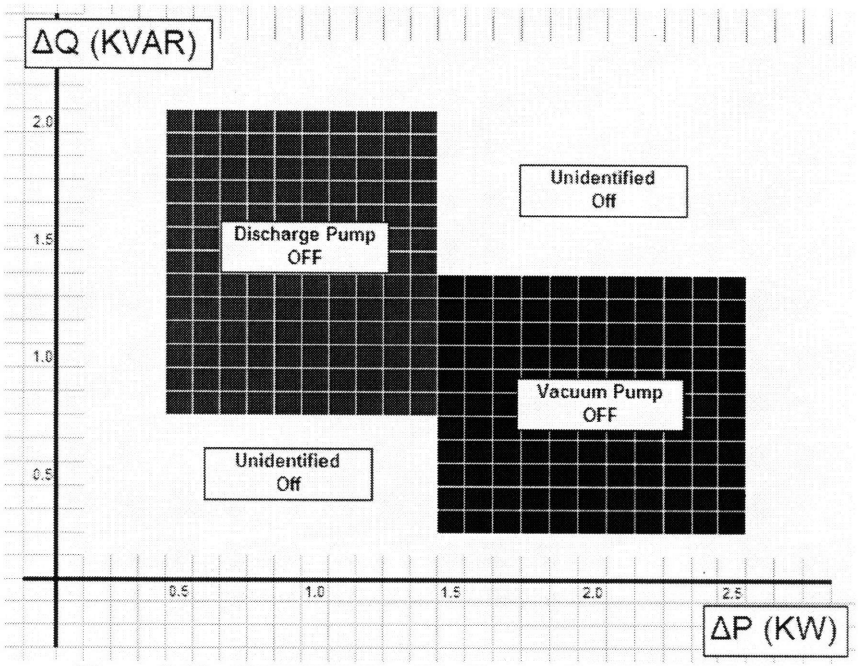


Figure 3-12 : CHT OFF Classification as a function of ΔP, ΔQ

Once an OFF classification is assigned, the classifier calls the *event file* creation function and exits.

CHT State Verification

If no transients are detected in a window, the classifier does not need to be called – so *Ginzu* takes the opportunity to verify the state. This is accomplished by calculating:

- Power Level (μ) - the average power level over the most recent 10 seconds.
- Power Deviation (σ) - the standard deviation of the power over the most recent 10 second window. This is treated as a measure of steady state.

These values are used in the state verification function, CHT_IC(), in the following checks.

- If level is ~ 0 KW ($\mu < 0.25$ KW), and steady ($\sigma < 0.1$ KW), and state information is not ALL OFF, the state is reset to ALL OFF.
- If level is non-zero ($\mu > 0.50$ KW), and steady ($\sigma < 0.1$ KW), and state information is ALL OFF, the state is corrected by 'guessing' that a vacuum pump is running as it is the most common running state.
- If state information indicates that we are in an invalid state (such as negative pumps running or more pumps running than are installed), the state is corrected to the most likely state.

3.6 NILM Classification of Reverse Osmosis (RO) System Events

USCG Cutters utilize the Village Marine Tec RC7000+ Reverse Osmosis Unit to convert ingested seawater into potable water. The original RO NILM installations were completed on the USCG Cutter Seneca in 2005 and on the Escanaba in 2006. DeNucci [3], Mitchell [11], and Branch [1] provide a detailed description of RO NILM observations as well as an operational overview. A summary of Mitchell's 2007 description is provided here.

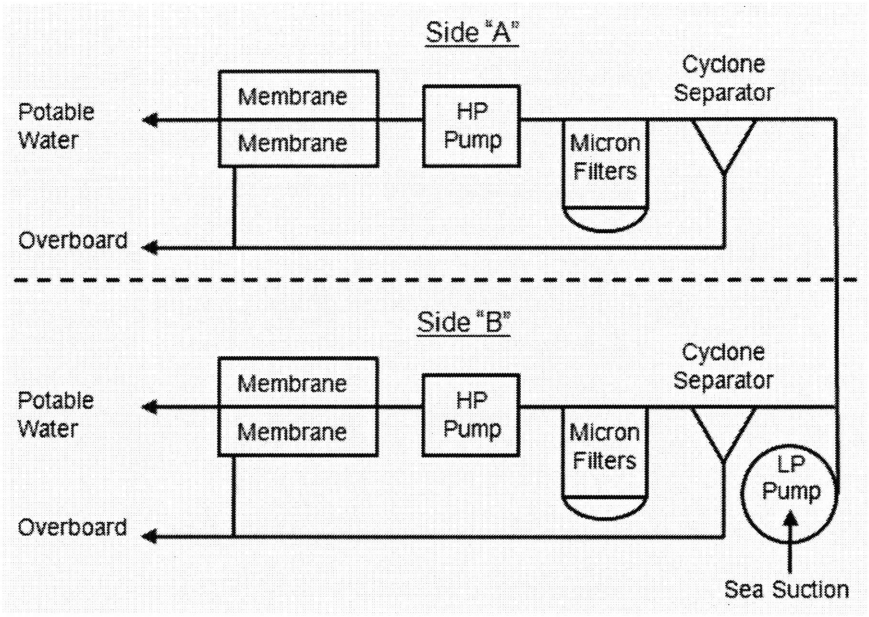


Figure 3-13: Simplified system diagram for Village Marine Tec RC7000+ Reverse Osmosis Unit. This system consists of a low pressure pump followed by two symmetric high pressure sides capable of producing freshwater from incoming seawater. [11]

The Village Marine Tec RC7000+ Reverse Osmosis Unit (Figure 3-13) consists of a low-pressure centrifugal pump, two micron filters, a high-pressure positive displacement pump, semi-permeable membranes, and a high-pressure regulating valve. Raw seawater from the sea-suction strainer is forced through the cyclone separator and micron filters by a 5 Hp centrifugal pump, known as the low-pressure (LP) pump. The cyclone separator discharges large suspended solids from the raw seawater while the filters trap the remaining smaller debris. The high-pressure (HP) positive displacement pump increases the pretreated raw seawater (i.e. feed water) pressure from 40 psi to over 800 psi. The pressurized feed water then flows directly into the membrane array. The membrane array is a fixed arrangement of two fiberglass pressure vessels that each contain two Model SW6040 RO membrane elements that are 6" in diameter and 40" in length. Reverse osmosis occurs as the semi-permeable membranes separate the pressurized feed water into two streams; the high purity product stream (i.e. purified water) and the concentrated reject stream (i.e. brine). The brine is piped directly overboard while the permeate is sent to the potable water storage tanks. [11, 18]

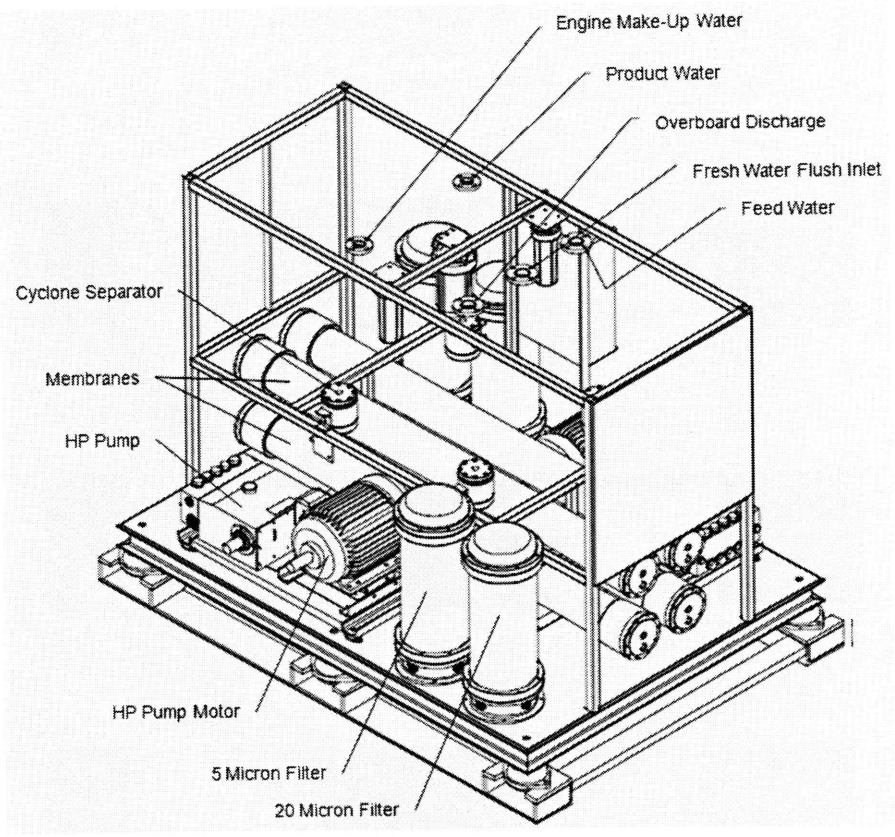


Figure 3-14: Village Marine RC7000+ Reverse Osmosis Distilling Unit [18]

There are several key valves used during normal RO operations. The HP Regulating Valve (V6 A/B) sets the pressure inside the membranes. Due to the high operating membrane pressure, the HP pump normally starts in an unloaded condition by using the HP Bypass Valve (V7 A/B). This valve diverts the discharge from the HP pump overboard without going through the HP Regulating Valve, thus reducing the membrane pressure. Once the HP pump is running smoothly, it is gradually loaded by manually closing HP Bypass Valve. The Product Water Solenoid Valve (V10 A/B) automatically directs permeate to the potable water storage tanks (valve open) or overboard (valve closed) based on the product salinity measured by the Water Quality Monitor (component number MON A/B). [11, 18] A normal RO system start cycle [11] is provided in Figure 3-15.

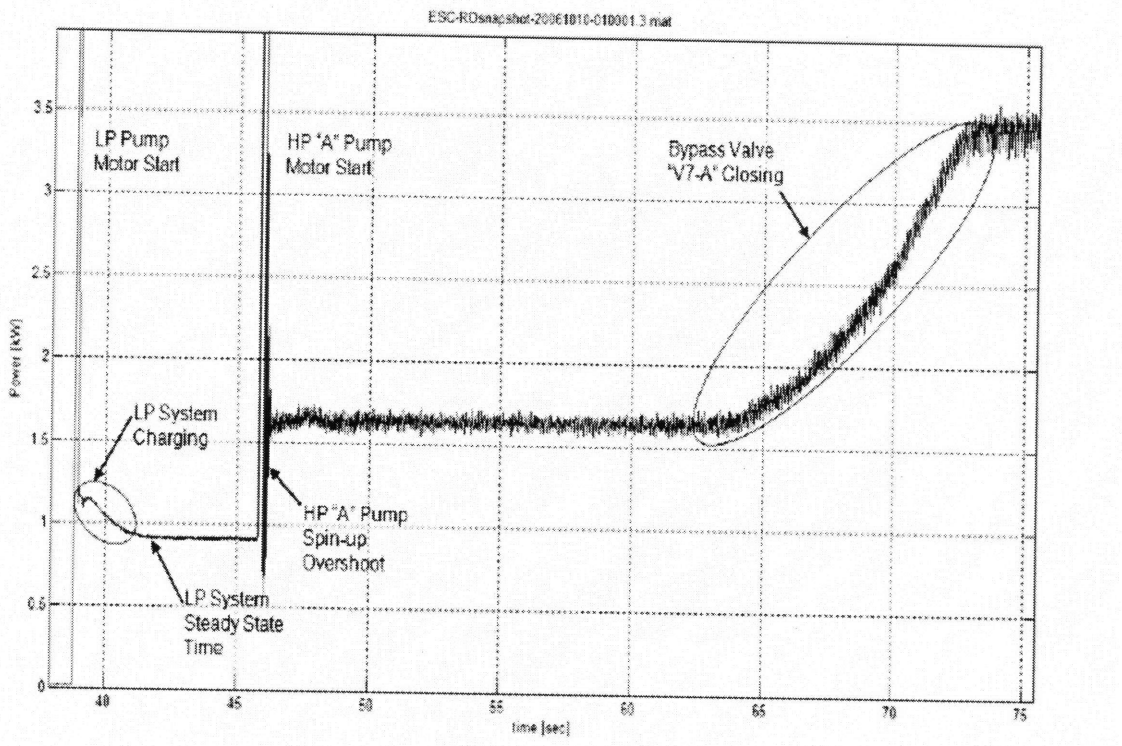


Figure 3-15 : Real Power Signal for typical RO start cycle [11]

While the CHT and RO systems are relatively simple systems and contain few large loads, the system operations are distinctly different, which ultimately affects the classification strategy and requirements. Specifically,

- The loads of the RO system normally cycle during startup and shutdown but normal system operation results in a constant system power. As discussed earlier, the CHT system continuously generates cycling load events during normal operation.
- The ON/OFF events of the RO system are pushbutton-triggered operator actions. The sequence of these actions is designated by procedure and is relatively predictable. The CHT pumps cycle automatically based on sensor set points or tank levels. The CHT system is normally set for automatic operation and minimal (or no) operator action is required. Since system load is variable, the distribution of pump starts tends to follow a Poisson distribution.
- All CHT classifications are associated with changes in motor operation (i.e. pump starts and stops). In the RO system, the adjustment of the bypass valve creates a

significant change in real power [11]. Depending on the speed of this adjustment, this event may or may not cause detections in the change-of-mean filter.

- The ability to determine when the LP pump has reached steady state is a desirable classifier output. This event is not detectable via the normal change-of-mean approach so additional classification logic is required.
- Noise in RO real power data is significantly higher than CHT noise. While this variation could be filtered, a simpler approach is the use of reactive power for event detection. Reactive power is a much cleaner signal and is already provided by the preprocessor.

3.6.1 RO Event Detection

The RO detection algorithm is identical to the CHT detection algorithm – implementing the change of mean filter described in 3.3 Event Detection. Event windows that require classification are passed to the RO Classifier. Windows that contain no events or contain redetections are passed to the RO State Verification algorithm, RO_IC().

3.6.2 RO Classifier Implementation and Outputs

The second expert classifier included in *Ginzu* is the RO event classifier. This classifier is also called from the *Ginzu* shell program. As the RO classifier receives event windows containing unclassified events, a decision tree approach is then applied to determine class. The use of the classifier is specified by setting mode 1 via command line arguments (e.g. **dd if=power_file.txt | ./ginzu m 1**). Table 3-5 provides a summary of possible event classifications in the RO classifier.

Description	ID CODE
Low Pressure Pump ON	(001)
High Pressure Pump ON	(002)
Low Pressure Pump OFF	(003)
High Pressure Pump OFF	(004)
ALL OFF	(005)
P/Q mismatch has cleared.	(006)
Power Stable after LP Start	(007)
Unclassifiable Cycling Load	(008)
Noise	(009)

Table 3-5 – Reverse Osmosis Classification Codes

3.6.3 RO Classifier Logic and FSM

When a 10 second event window is received, the classifier determines if the event is located in the window such that at least 0.5 seconds of post-event power are available. Unlike the CHT classifier, no 'long-term' change is power is calculated, so classification decisions are made much earlier than in the CHT classifier – and *event file*s are created with less delay. This is desirable as RO system events are normally driven by operator action. If an excessive delay existed between the operator action and the NILM response, the operator may think the RO unit and/or NILM are not operating correctly.

If the event is located too early in the event window, no classification is performed. During the next advance of the sliding window, the event will be re-detected and correctly positioned in the window. At this point, the classification resumes. When the classifier has a well-positioned event, the ΔP and ΔQ are calculated. Like CHT, this is accomplished through a single static window that calculates the difference between the subsequent 0.25 seconds of power data and the preceding 0.25 seconds of power data (while neglecting the 0.04 seconds around the event). The ΔQ is then compared to the noise threshold to determine if the event is an ON event, OFF event, or noise event. In the RO classifier, 0.5 KVAR value is used as the noise threshold. (Note the use of reactive power in the RO classifier.)

If a power change does not meet the noise threshold, the event is either (1) system noise or (2) a short-duration cycling load. The RO classifier attempts to distinguish these events by calculating the standard deviation (σ) of the reactive power around the event

index. If the σ is large, the event is considered an unclassifiable cycling load. If it is not, the event is labeled as noise.

Events that exceed the noise threshold are ON or OFF events. As the normal sequence of events in the RO system tends to be predictable, the RO classifier places a much higher value on state information. The RO FSM is shown in Figure 3-16.

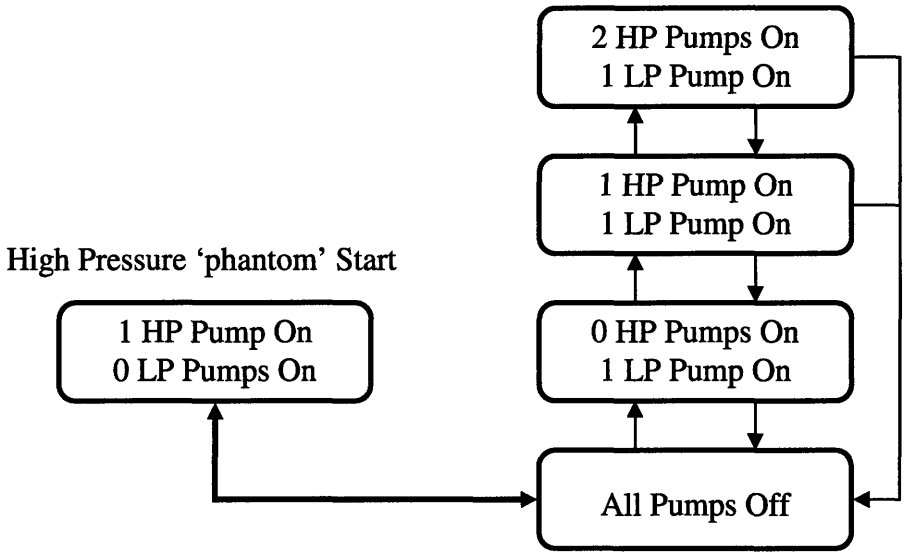


Figure 3-16 : RO State Transition Diagram. The 'phantom start' state was not included in the initial Escanaba deployment as this state was thought to be invalid. Field results indicated that this abnormal state can occur and must be considered by the expert classifier.

The addition of the 'phantom start' state is notable. A phantom start is describes by Mitchell in [11] as a start of HP pump without a running LP pump. This is a potentially damaging event and is normally prohibited by procedure. In the early versions of the RO classifier, the state consisting of zero running LP pumps and one running HP pump was considered to be invalid. Consequently, the classification of a HP pump start from an ALL OFF state was not considered. This constraint was proven to be invalid when a number of 'phantom HP pump' starts occurred during normal operation – resulting in misclassification as HP pump starts were assumed to be LP pump starts. This is described in Chapter Five and by P. Branch in [1].

Classifying RO ON Events

The detection of an ON event will immediately be followed by a correlation score calculation for both LP and HP pump templates.

If the initial system state is ALL OFF, the most likely event is the start of a low pressure pump (001). The classifier immediately checks the LP correlation score and classifies the ON event as an LP pump on if the score is greater than the tuned threshold (0.5). If the LP pump correlation score is weak (< 0.5), the event will still labeled as a LP pump start, but the classification string will be modified to indicate that the correlations was weak.

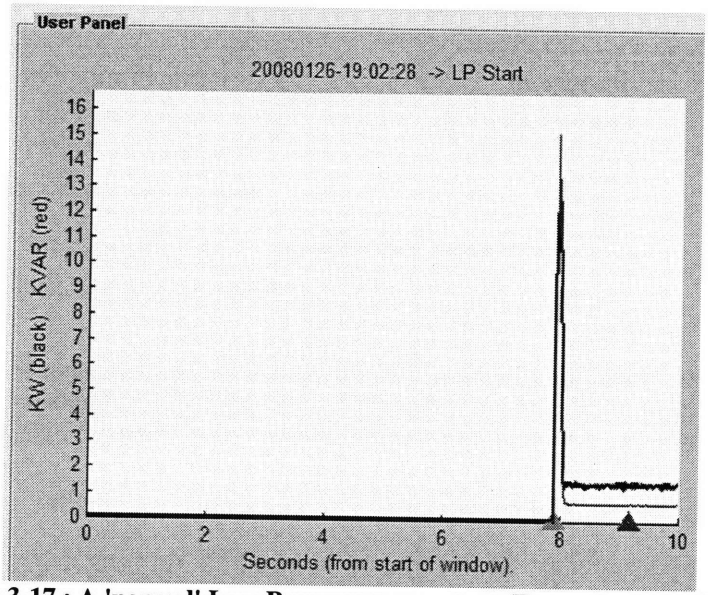


Figure 3-17 : A 'normal' Low Pressure pump start (Escanaba 26JAN2008)

An event that causes in increase in power will always be classified as a LP pump start unless the event can clearly be defined as a 'phantom' HP start. The event will be classified as a 'phantom' HP pump start if and only if the following conditions are met:

- The initial state is ALL OFF.
- The steady state ΔQ is greater than 10.0 KVAR.
- The HP pump correlation score is relatively strong (> 0).

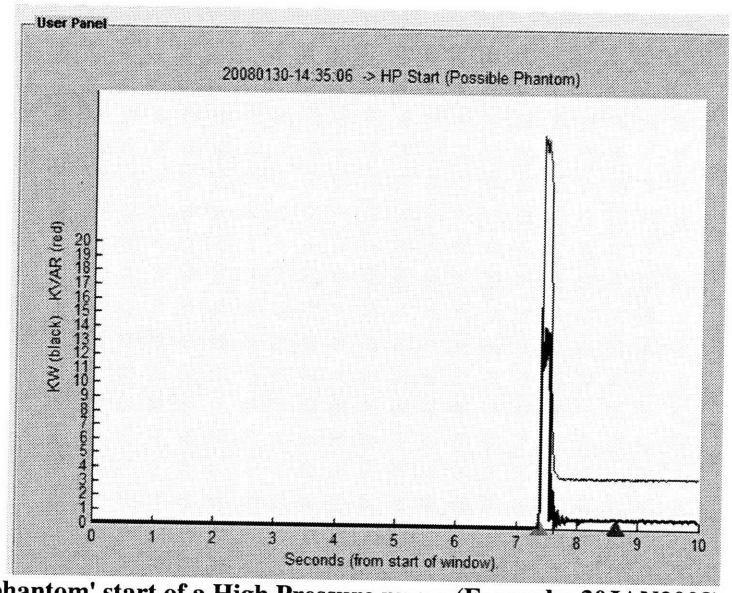


Figure 3-18 : A 'phantom' start of a High Pressure pump (Escanaba 30JAN2008). The LP pump is not running prior to the starting event of the HP pump.

An up power event during a period where the low pressure pump is operating will always be classified as a HP pump start. If the HP pump correlation score is weak (<0) or the ΔQ is less then 10 KVAR, the event will be labeled as an HP pump start – but the classification string will be modified to indicate that the correlation was weak.

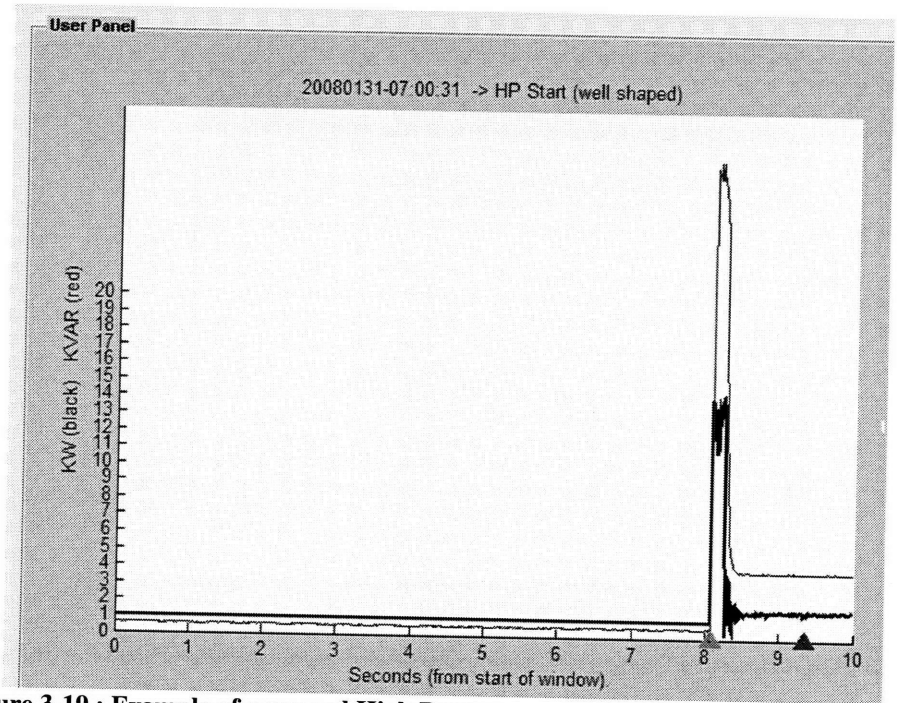


Figure 3-19 : Example of a normal High Pressure pump start (Escanaba 31JAN2008)

As shown in Figure 3-16, an up power from an ALL ON state is not an allowed transition. If this scenario is detected, the classifier considers the following options.

- If the HP pump correlation score is sufficiently strong (> 0) and the ΔQ is greater than 10 KVAR, the event is labeled as an HP pump start and the classifier comments that pre-event state was likely incorrect.
- If the event does not meet the HP pump criteria, the only possibility is an increase in power due to a bypass valve adjustment. This transient is normally not detected by the change of mean detector due to its slow rate of change; however a rapid manual valve operation could trip the detector.

Classifying RO OFF Events

In the OFF event scenario, post-event power is calculated. If this post-event power is approximately zero (< 0.1 KVAR), the event is classified as ALL OFF (005) and the state is reset to 'All Pumps Off'. If the post-event power is not zero, the event is classified based on the most likely transition (as indicated in the FSM diagram).

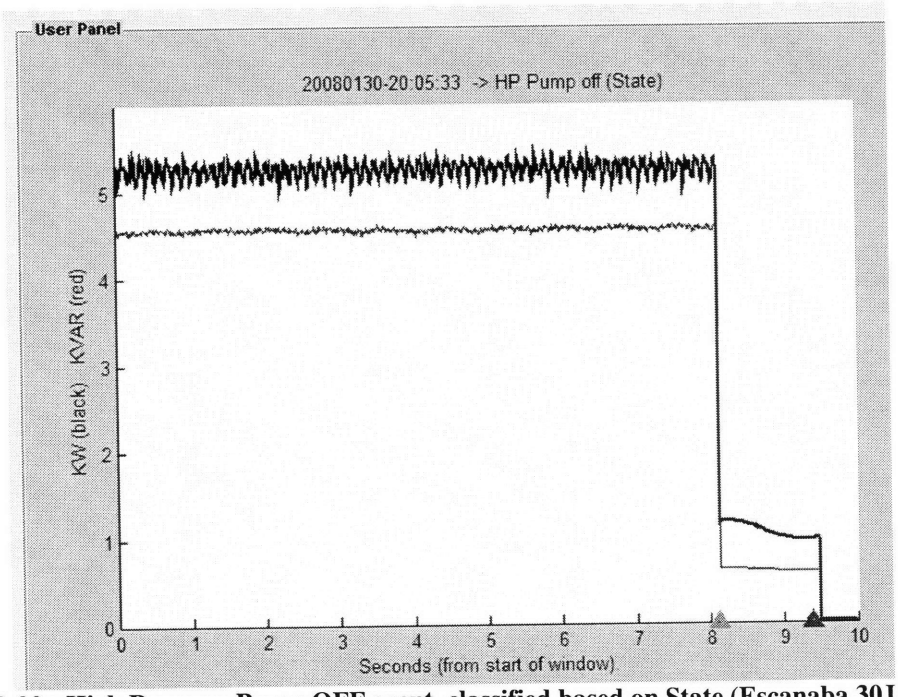


Figure 3-20 : High Pressure Pump OFF event, classified based on State (Escanaba 30JAN2008)

A down power during an ALL OFF state, where post-event power does not go to zero, is labeled as an 'Unknown OFF event' and is given an event code of (009).

RO Condition Checking and State Verification

As in the CHT classifier, *Ginzu* attempts to verify state information during windows where no events require classification. This is accomplished by calculating the Level (μ) and Standard Deviation (σ) over the most recent three seconds of data – for both real and reactive power. The following state verification is then performed:

- If level is ~ 0 ($\mu_{real} < 0.10$), and steady ($\sigma_{reactive} < 0.10$), and state information is not ALL OFF, we reset the state to ALL OFF.
- If level is non-zero ($\mu_{reactive} > 0.50$), and steady ($\sigma_{reactive} < 0.10$), and state information is ALL OFF, we correct the state information based on the current reactive power level. A low system power level ($Q < 1.5$ KVAR) will reset state information to 'One LP Pump On, Zero HP Pumps On'. An intermediate level ($Q < 10.0$ KVAR) will reset state to 'One LP Pump On, One HP Pump On'. Reactive power in excess of 10.0 KVAR will set 'All Pumps On'.

In addition to state verification, the function provides two additional classifications – (1) the detection of steady state following a LP pump start, and (2) the detection of a bypass valve adjustment. This is accomplished through the Boolean variable *stable*.

Steady State following Low Pressure Pump Start

Any detected event (such as an LP or HP start) will set *stable* to 0 – indicating that the system is not currently steady state. A 'Steady state following LP pump start' event (007) is created when the following conditions are met:

- System indicates that it is not stable. (i.e. stable = 0)
- The current system state indicates one LP pump is running.
- No classifiable events exist in the current window.
- Both $\sigma_{reactive}$ and σ_{real} are less than 0.10 indicating that power is constant.

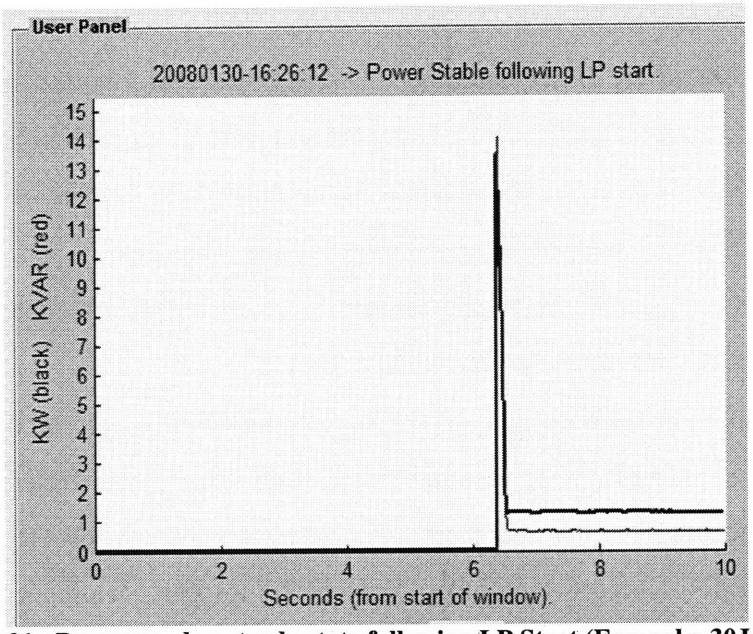


Figure 3-21 : Power reaches steady state following LP Start (Escanaba 30JAN2008)

Bypass Valve Adjustment

The operating instructions of the RC7000+ RO unit state the following: *"When flow through the reject discharge flow meter appears to be free of air bubbles, slowly close High Pressure Bypass Valve (V7 A/B)."* In 2006, Mitchell discussed the prognostic impact of improper operation of the bypass valve and observed that the operation of the bypass valve was evident in the monitored power signal. Specifically, Mitchell stated the *"As the valve closes the internal membrane pressure increases causing the HP pump to demand more power. In some cases watchstanders start the HP pump with valve number V7 closed placing a tremendous amount of friction and stress on the ceramic plungers. Additionally, the sudden increase in membrane pressure could damage the seals, leading to leaks and failures. ... there is no interlock to prevent incorrect operation of the bypass valve, but looking at the real power demand difference between the open and closed valve positions it is clear that NILM could detect this condition."*

The condition of the bypass valve is determined by examining the ratio between P and Q levels. Observation of RO system data has shown that P will not exceed 90% of Q until the bypass is closed. If the condition checker is called while a high pressure pump is running and Q is constant, the algorithm calculates the ratio of P/Q. If P is less than

90% of Q, the algorithm sets the global variable MISMATCH = 1 indicating that a mismatch currently exists. A 'P/Q mismatch has cleared' event (006) is created when:

- A P/Q mismatch exists. (i.e. MISMATCH =1)
- At least one HP pump is operating and the LP pump is operating.
- $\sigma_{reactive}$ is less than 0.10 indicating that reactive power is constant.
- P/Q > 0.90

Figure 3-22 provides an example where NILM history could be used to review operator performance. Since the rate of power change is proportional to the rate of V-7 operation, the NILM provides a historical log of how well operating procedures were followed.

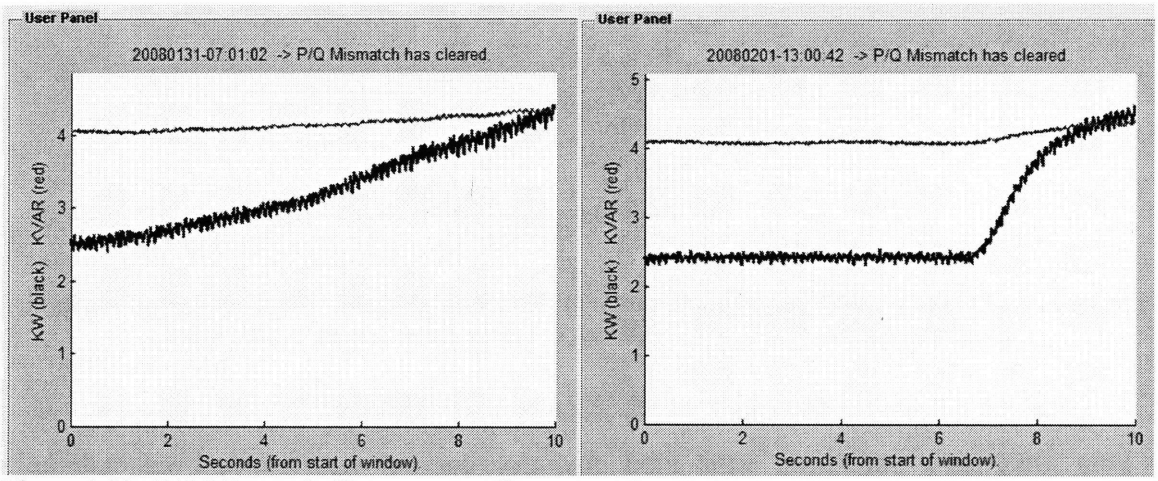


Figure 3-22 : P/Q Mismatch Clear events – Slow operation (Left), Fast Operation (Right). Procedure states that the valve should be operated slowly. Since the rate of power change is proportional to the rate of V-7 operation, the NILM provides a historical log of how well operating procedures are being followed.

3.7 Detection Only Mode

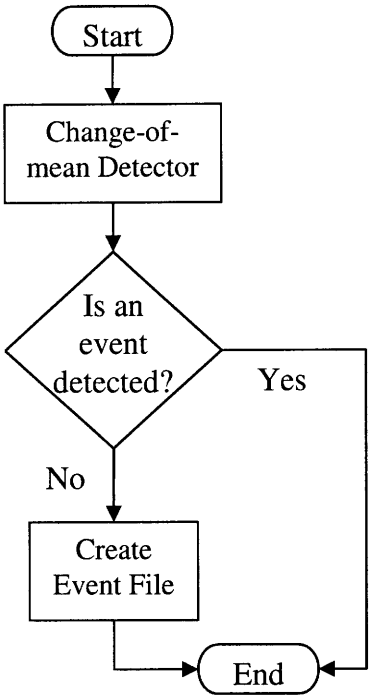


Figure 3-23 : Flowchart showing *ginzu* operation in 'Detect only' mode. Ten second event windows are created when transients are detected, but no classification is attempted.

Ginzu may be set to automatically generate *event files* whenever a transient is detected. No classification is attempted in this mode and all *event files* are generated with a default label of 'unclassified' and a default code of (000). This mode is useful when processing data from a new system that will not have an associated expert classifier or if it is intended to conduct the classification using an external classifier.

In this mode, *Ginzu* receives power data and located transients using the previously discussed detection algorithm. When a transient is detected, the 10 second window is immediately passed to the *event file* creation function that labels the event as 'unclassified', timestamps the event, and generates the *event file*. The threshold for the change-of-mean detector is set to a default value of 0.5 KW but may be modified by a command line argument.

3.8 Streaming Mode

In streaming mode, *Ginzu* generates a continuous realtime stream of *event files* – independent of transient detection. In streaming mode, the preprocessed power data is received by *ginzu,* which then loads a default 10 second buffer with data. Once an initial buffer is loaded, the *event file* is created. As two new seconds of data are received, the two oldest seconds are discarded and a new *event file* is created. This two-second 'Window Period' may be modified via a command line argument. This mode is typically used for setup or troubleshooting to determine if the NILM hardware is providing the anticipated signal at the appropriate power levels. This mode can be used to (1) set power scaling coefficients, (2) set variable measuring resistor dip switch settings inside the NIB, and/or (3) verify that the GUI is able to properly receive *event files*.

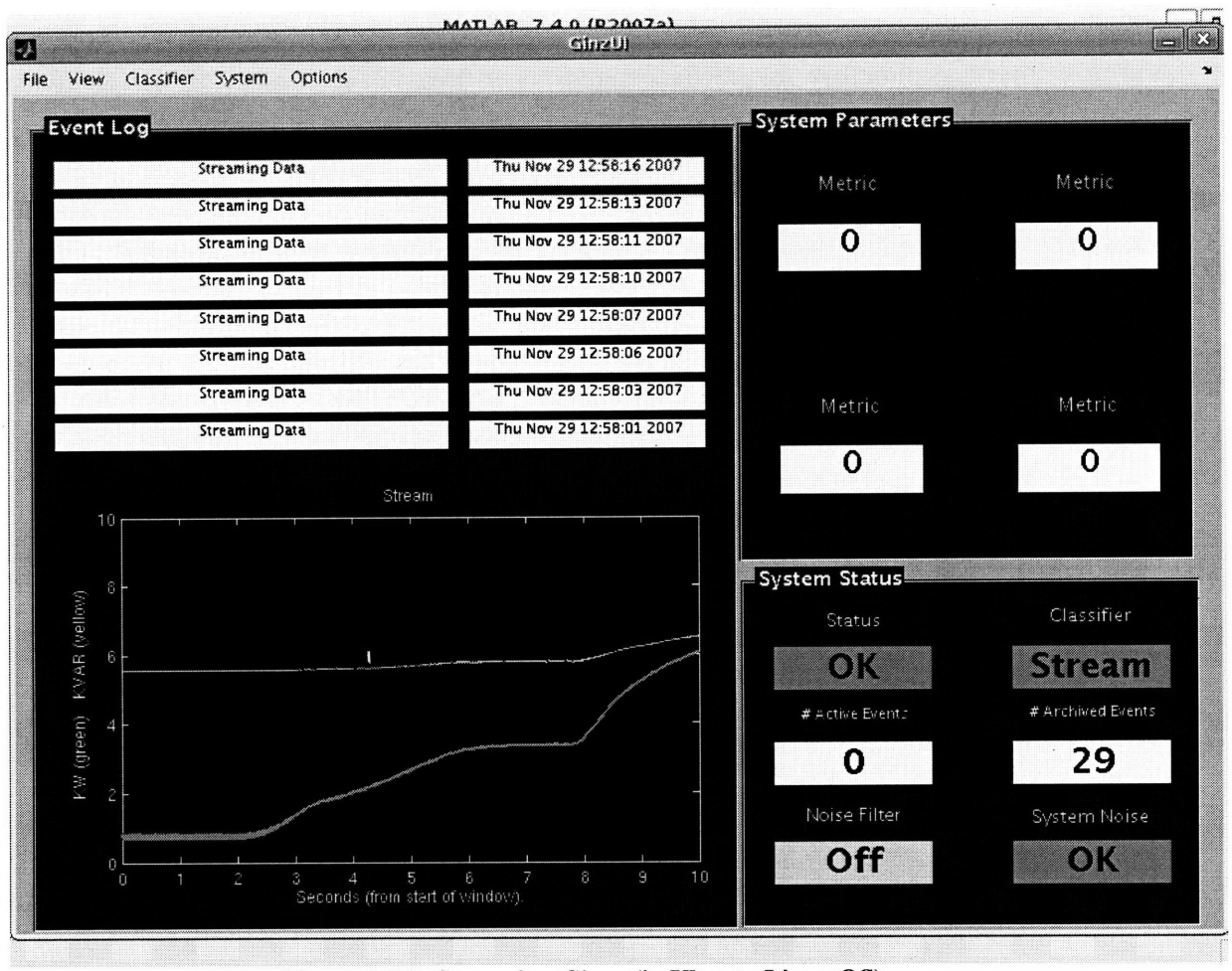


Figure 3-24 : Streaming *Ginzu* (in Ubuntu Linux OS)

3.9 Ginzu Graphical User Interface (GinzUI) and Diagnostics

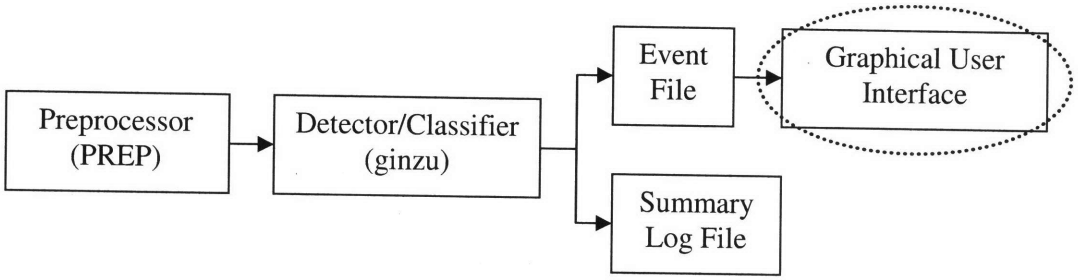


Figure 3-25 : *Ginzu* Graphical User Interface processes event files generated by the *ginzu* classifier and displays them to the user.

3.9.1 Functional Overview

The *Ginzu Graphical User Interface (GinzUI)* provides the interface between the *event file* and the NILM user. The primary functions of *GinzUI* are (1) to continuously check the user interface directory for newly created *event files*, (2) to read *event file* contents and move the *event files* to an archive directory, (3) to perform diagnostics on *event file* data and alert the user if a diagnostic has failed, and (4) to allow the user to graphically view *event file* contents. Figure 3-26 shows the *GinzUI* functional block diagram.

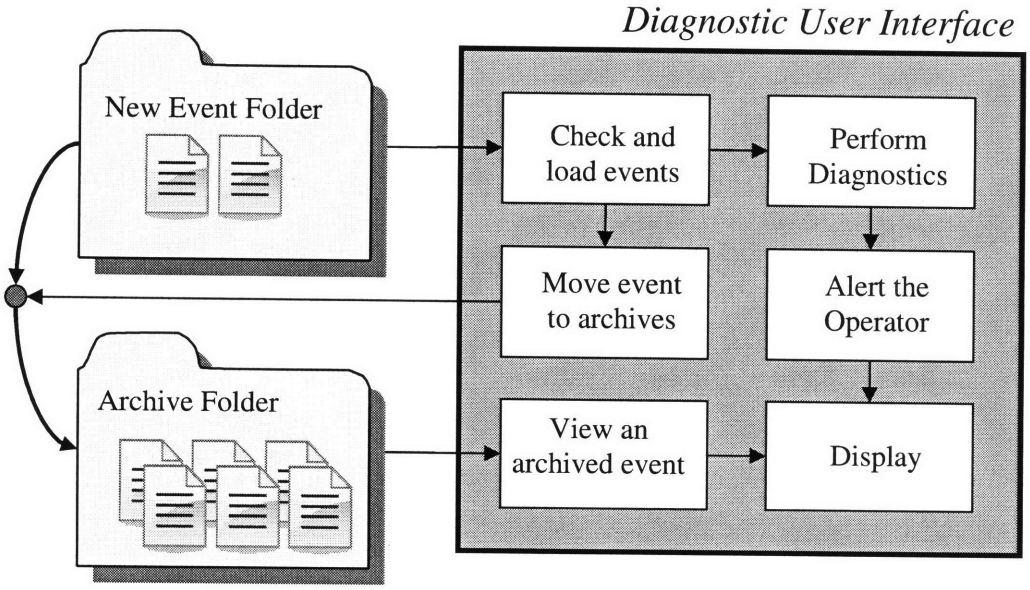


Figure 3-26 : Diagnostic User Interface (*GinzUI*) functional Diagram

The *GinzUI* was generated in Matlab using the graphical user interface development environment (GUIDE). A stand-alone binary executable was generated using the Matlab C compiler (mcc)[8]. By creating a stand-alone executable, the UI could be invoked from the Linux OS without the Matlab shell. This allows a user to run the UI with no background in Matlab. The *GinzUI* was specifically designed for the CHT system and provides the first three functions via a single front-end display (Figure 3-27). This display presents the user with an active log of system events, a simplified diagram of the CHT system showing running pumps, a numeric display of average pump runtimes and downtimes, indication of current system status, and a review of hourly diagnostics.

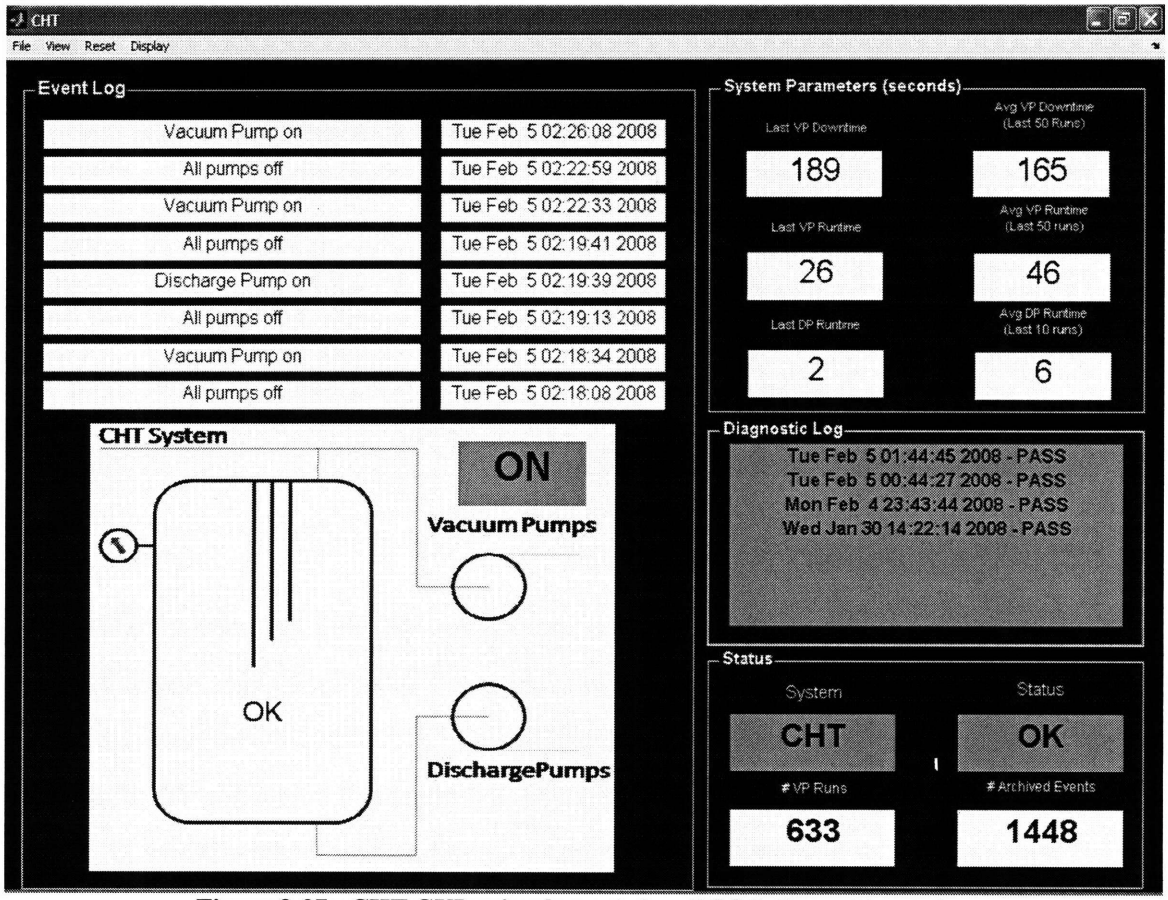


Figure 3-27 : CHT GUI as implemented on USCG Cutter Escanaba

The basic operation of the UI is controlled by a simple Matlab timer object (Figure 3-28). The timer object creates a background thread that invokes timerfcn(),

[8] The Matlab command 'mcc –m *UI_filename*' will create a stand-alone executable of the *UI_filename.m* source code and all supporting .m files.

which loads new *event file* and moves the file to an archive directory. In the CHT implementation, timerfcn() is invoked every two seconds.

```
% The default timer is TimerFcn.  It is implemented every 2 seconds.

handles.timer = timer('period',2);
set(handles.timer,'ExecutionMode','fixedrate','StartDelay',2,'BusyMode','drop');
set(handles.timer,'timerfcn',{@TimerFcn, hObject});
start(handles.timer);
```

Figure 3-28 – Matlab Timer Object as implemented in Matlab code. The timer creates a background thread that updates the GUI and checks for new *event files* every two seconds.

Additionally, timerfcn() does the following:

- Updates the event log.
- Illuminates a green textbox on the system diagram to indicate an ON event or disables the textbox to indicate an OFF event.
- Updates the pump runtime fields. Updates the pump downtime fields.
- Ensures the diagnostic package is run once per hour.
- Illuminates red and yellow warning blocks to indicate a diagnostic failure has occurred.
- Tracks the number of events in the archive folder and tracks the number of vacuum pump runs since the last system reset.

3.9.2 CHT Diagnostics

A rudimentary CHT diagnostic package was included in the *GinzUI* to provide realtime detection of CHT system faults that had been observed in ([12], [15]). These failures include clogs in gauge lines and/or priming orifices, tank level indication probe failures, and system leaks. When a system fault is detected, a comment is printed to the diagnostic log and the log turns yellow to indicate an abnormal condition. Additionally, a line is printed to a text file (errorlog.txt) that contains the description of the failure and the time of the detection.

Vacuum Pump Clog Detection

In [15], Piber describes a fault where the CHT vacuum pump fails to achieve normal running power level due to a system clog – this can either be due to a suction line blockage or a clog in the priming line. In reference to the clog in the priming line, Piber stated that "*Upon start up, seal water is automatically sucked into the pump housing through a small line connected to an orifice on the pump body. If this line or orifice becomes clogged the pump cannot prime and little or no vacuum is drawn. Without doing any work the power drawn by the pump decreases significantly.*" With regard to the suction line blockage, he stated "*This pump housing orifice clog occurs due to the introduction of sewage into the seal water tank via the vacuum pump air intake. This can occur when the sewage level inside the VCT reaches a high enough level for the sewage itself to be sucked up into the vacuum pumps.*"

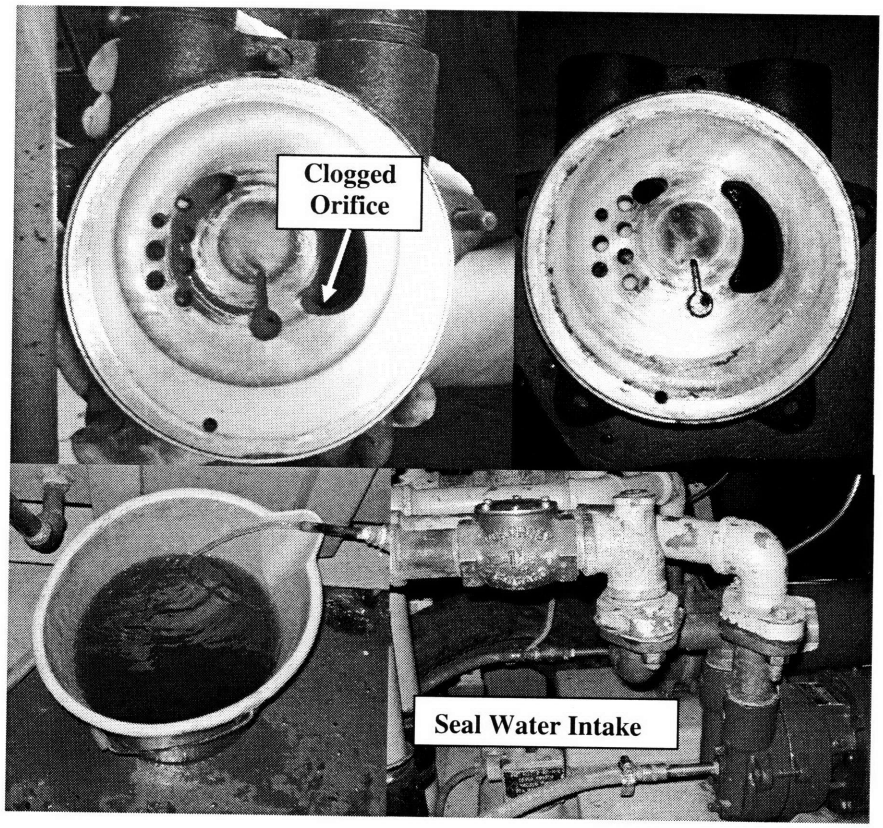


Figure 3-29 : Pictures from the loss of prime casualty repair onboard the USCGC *Seneca*. Pump housing prior to cleaning (above left) and after cleaning (above right). Sewage contaminated seal water (bottom left). Intact pump with seal water line entering body of pump housing (bottom right). [Piber 2007]

In either case, as the clogged pump does not draw a normal vacuum, the vacuum levels in the tank will continue to fall and the second pump will start to maintain pressure. Vacuum then rises until both pumps turn off at the high vacuum set point. This is illustrated in Figure 3-30 as the initial vacuum start is accompanied by a normal vacuum pump run that successfully draws normal system vacuum.

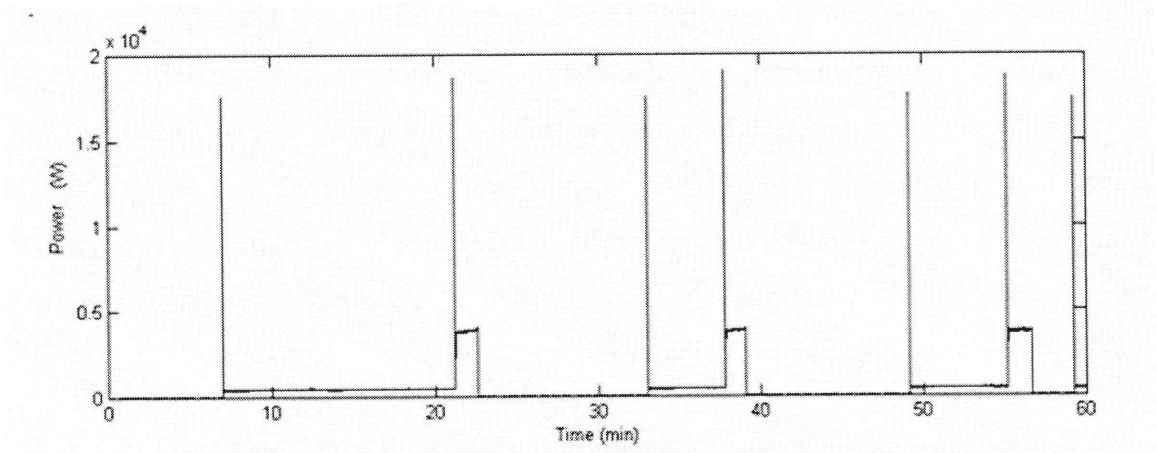


Figure 3-30 : Clogged Vacuum Pump Runs. First pump does not reach normal running power levels. Second pump starts to pull vacuum. Then both pumps secure. [15]

The VP ON with low running power (005) event is a potential output of *Ginzu* (see 3.6.3 RO Classifier Logic). To prevent the UI from alarming when a single 005 event is received, *GinzUI* examines the most recent ten VP starting events in a ten-element Boolean stack-type data structure. A zero is pushed onto the stack when a normal Vacuum start (001) is detected. A one is pushed onto the stack when *ginzu* classifies a 'low power' start (005). When the 11th event is pushed onto the stack, the oldest event is discarded. If the sum of the stack reaches five, more than half of the recent ten runs are faulty and there is adequate indication of a clog. The alarm indication will clear when the stack sum drops less than three – indicating most (8/10) recent runs were normal.

Tank Level Indication (TLI) Probe Failure Detection

Piber also described the operation of the CHT discharge pump control circuit and the effect of failed/fouled TLI probes. He states that "*These probes consist of a set of insulated metal probes that use the contents of the tank as the medium to complete the control circuit. Only the tips of each probe are exposed to the sewage. The circuit is completed when the tips of the probes make contact with the liquid contents allowing the*

electrical the signal to the common ground. The control circuits are broken when the sewage level clears the probes during a discharge pump run. The discharge pump cycles the sewage level inside the tank between the 33% and 5% full levels. The typical discharge pump run lasts about 3 to 4 minutes as 120 gallons of sewage is transferred to a larger holding tank."

In 2007, Piber had observed a casualty where the discharged pumps failed to run the normal "3 to 4 minutes" due to fouling of the TLI probes (Figure 3-31) and/or improper operation of the TLI control circuitry. Since the operation of the discharge pumps is controlled by the output of the TLI circuit, improper operation of the TLI probes (or circuitry) leads directly to abnormal pump operation. This is illustrated in Figure 3-32 where a failure in the TLI circuit was shown to create unusually short 'burst' discharge pump runs.

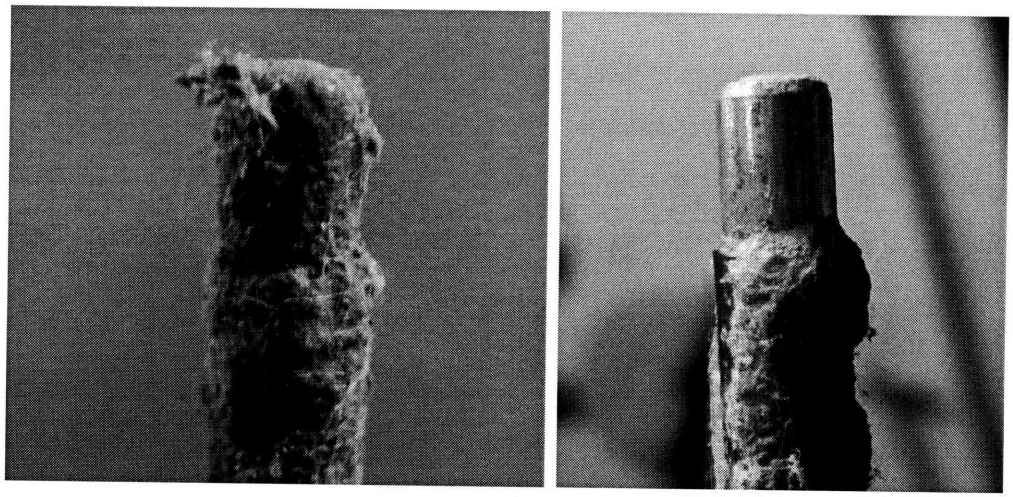


Figure 3-31: TLI Probe Fouling. This type of fouling results in erroneous level signals sent to the CHT controller. Since the discharge pumps are controlled by level, these erroneous signals cause abnormal pump operation. [Piber 2007]

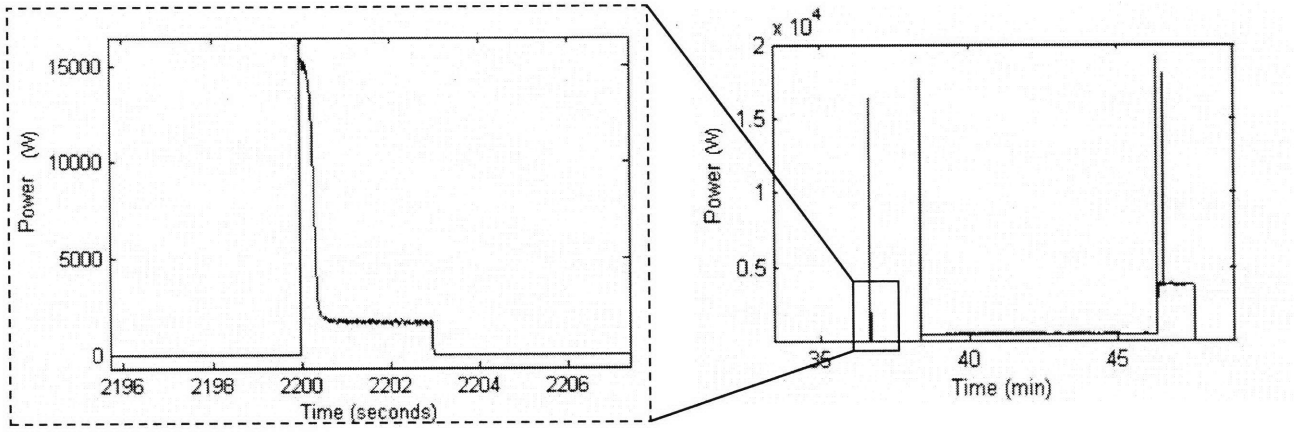


Figure 3-32 : USCG Cutter Seneca Short Duration Discharge Pump Runs Characteristic of Probe Failures. [15]

To detect a TLI probe failure, the *GinzUI* tracks the average of the last ten discharge pump runtimes. During normal operation, the average of this array should be between 180 to 240 seconds. A series of simultaneous short runs will drive the average down and will eventually cause a fault condition if the average drops below ten seconds. The GUI will alert the user by changing the color of the diagnostic log to yellow and showing a red 'PROBE FAILURE' block on the CHT diagram (Figure 3-33). The fault condition will not clear until the running average rises above 60 seconds – so three 'normal' pump runs will generally to clear the alarm.

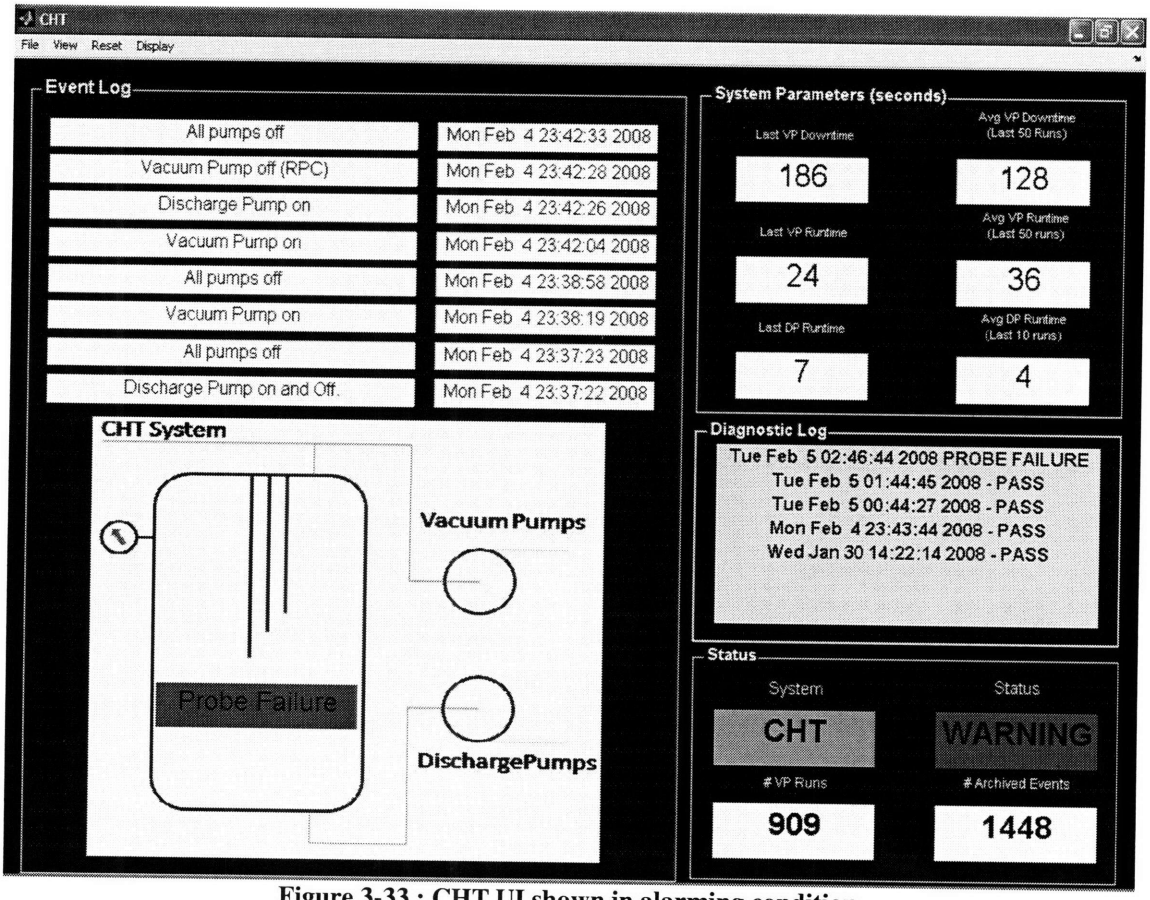


Figure 3-33 : CHT UI shown in alarming condition

Excess Vacuum Pump Run Duration

The operation of the vacuum pumps is controlled by two pressure switches. According to the operator's manual for the CHT system [7], "*The vacuum pumps are controlled by two pressure switches mounted onto the side of the VCT. One switch controls the normal operation of the vacuum pumps by cutting on a single lead pump when the vacuum reaches 14" Hg and cutting off the same pump when system vacuum pressure reaches 18" Hg. The second vacuum switch operates the backup vacuum pump and is set to cut in at 12" Hg to aid the leading pump under unusual instances of heavy loads. Upon reaching 18" Hg both pumps de-energize in unison. Both switches can be manually adjusted by altering the range and differential settings using dials inside the switch. The differential adjustments adjust the vacuum difference between the cut-in and cut-out points. The range adjustment raises or lowers the cut-in points.*"

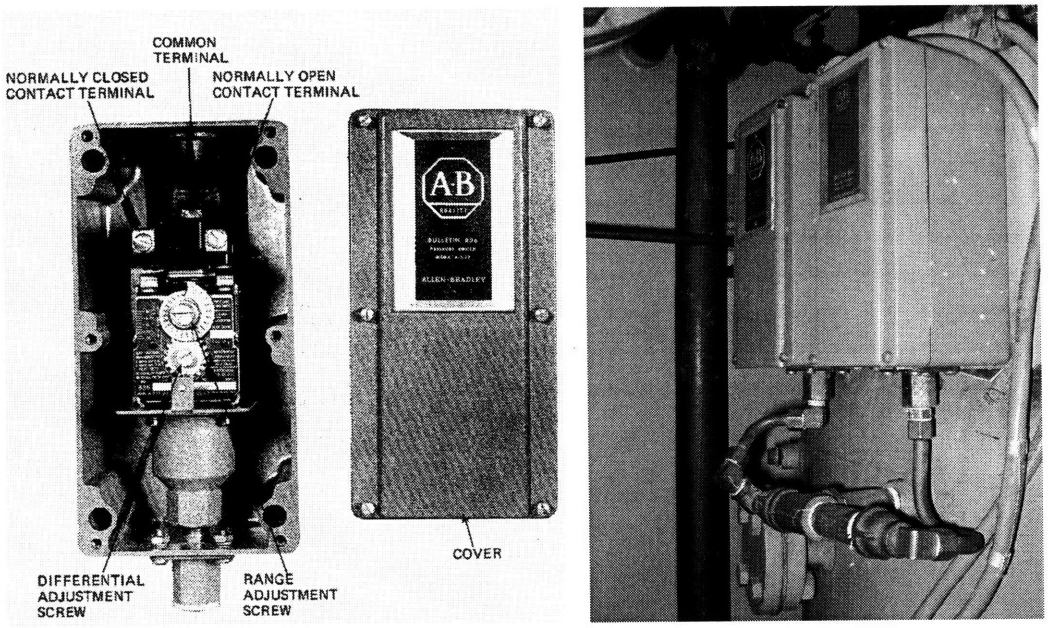


Figure 3-34: Pressure switches mounted on side of sewage retention tank. Both switches share a common vacuum sendsing line and are subject to improper operation due to blockage or clogging. [Piber 2007]

Piber showed that this signal from the pressure switches has a significant effect on system operation as (1) incorrect switch settings can cause the pumps to attempt to maintain high vacuum (increasing run duration and system wear), and (2) blockage in the vacuum sensing line can introduce a delay in system operation (increasing run duration and system wear). Figure 3-35 and Figure 3-36 show running power during normal operation and then during a period of excessive runtime.

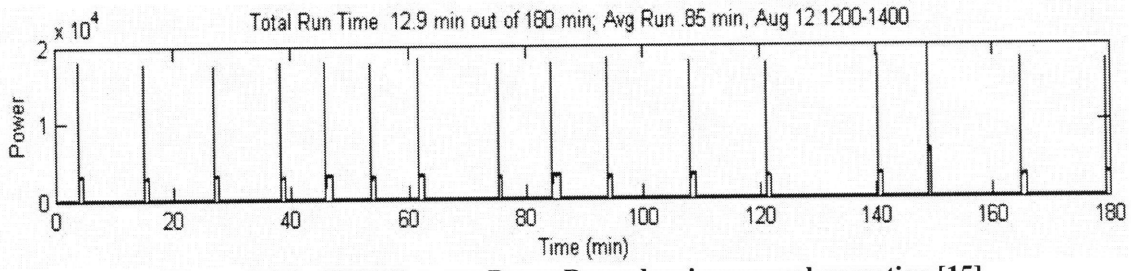


Figure 3-35 : CHT Vacuum Pump Runs showing normal operation [15]

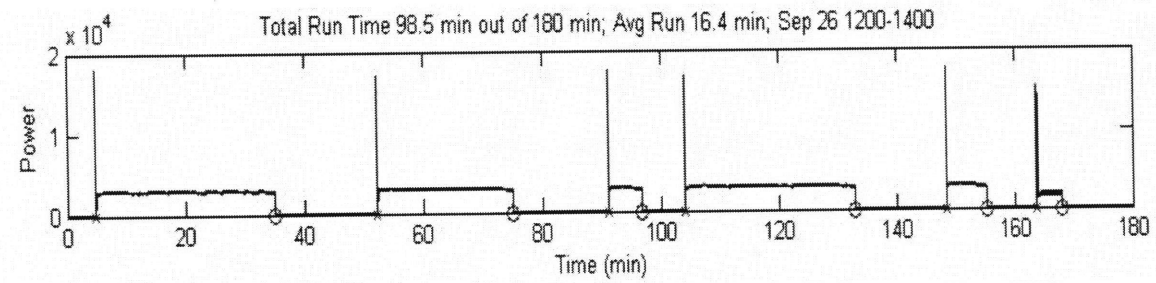


Figure 3-36 : CHT Vacuum Pump Operation showing unusually long run times [15]

The CHT UI continuously displays the most recent vacuum pump run duration and the average of the runtimes over the last 50 runs. During normal operation, this value should be between 30 and 60 seconds [12]. The runtimes are calculated and retained in an array as Vacuum pump OFF (004) or ALL OFF (009) events are received. The UI alerts the user if the average runtime exceeds 180 seconds. Since a clogged vacuum pump combined with improperly set or faulty sensor could result in similar system behavior, this diagnostic is disabled if a Vacuum Pump Clog has already been detected.

Leak Detection[9]

A statistical method to detect leaks in a cycling vacuum system was developed and tested in [12] and [2]. A complete discussion of vacuum leak detection is provided in [2] and briefly summarized here.

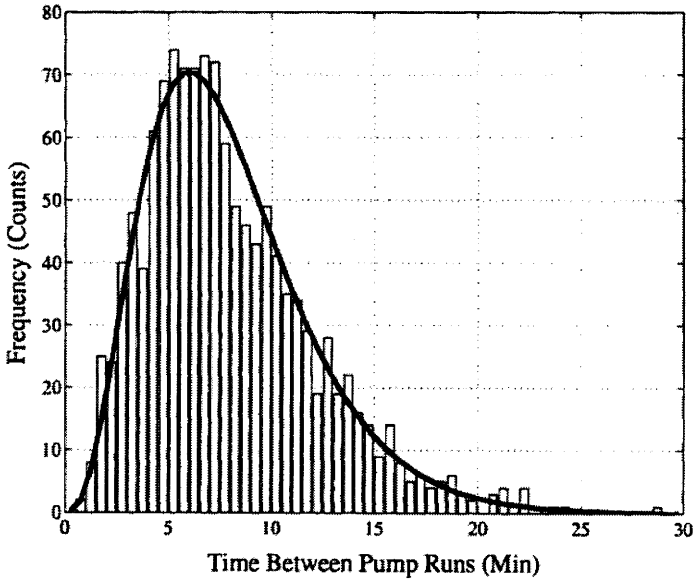


Figure 3-37 : Histogram of 'time between vacuum pump runs' for a simulated vacuum system [9]

Vacuum leak detection methods review the statistical distribution of 'time between vacuum pump runs' to determine if a change in system loading has occurred. A histogram (such as Figure 3-37) tracks the number of pump runs as a function of time

[9] The leak detection diagnostic was disabled in the *Escanaba* prior to the January underway. This was done due to early observations that the existing probe casualty and improperly set vacuum pump switches would cause erroneous leak detections.

between pump runs. The distribution tends to be Poisson where the k and λ parameters are a function of the variable system load.

$$f(k;\lambda) = \frac{\lambda^k e^{-\lambda}}{k!} \tag{3.1}$$

Theoretically, a large leak would create a constant drain on the system and reveal itself as a sharp peak in the histogram. The location of the peak could then be used to approximate the leak rate. In reality, Cox had shown that various real-world factors caused this peak to be 'smeared' over the histogram for more typical 'small' leaks. Leak detection is accomplished by two methods. The large leak diagnostic searches the histogram for 'edges' or spikes in the distribution. If the edge is significant (in excess of some tuned threshold), the diagnostic reports that a large leak has been detected. A small leak diagnostic reviews gradual changes in the histogram (defined by k and λ) to determine if a change is occurring. If the change exceeds some tuned threshold, a small leak is diagnosed.

3.9.3 Event Viewer (ViewerUI)

The ability to view files is provided by the ViewerUI, which can be called by a pull-down menu or can also act as a stand-alone UI. The ViewerUI accesses the archive directory and allows the user to select any *event file* in a list box. The viewer also has the ability to simultaneously display a current event and either the vacuum pump or discharge pump template. This allows the user to compare events. The ViewerUI is shown in Figure 3-38.

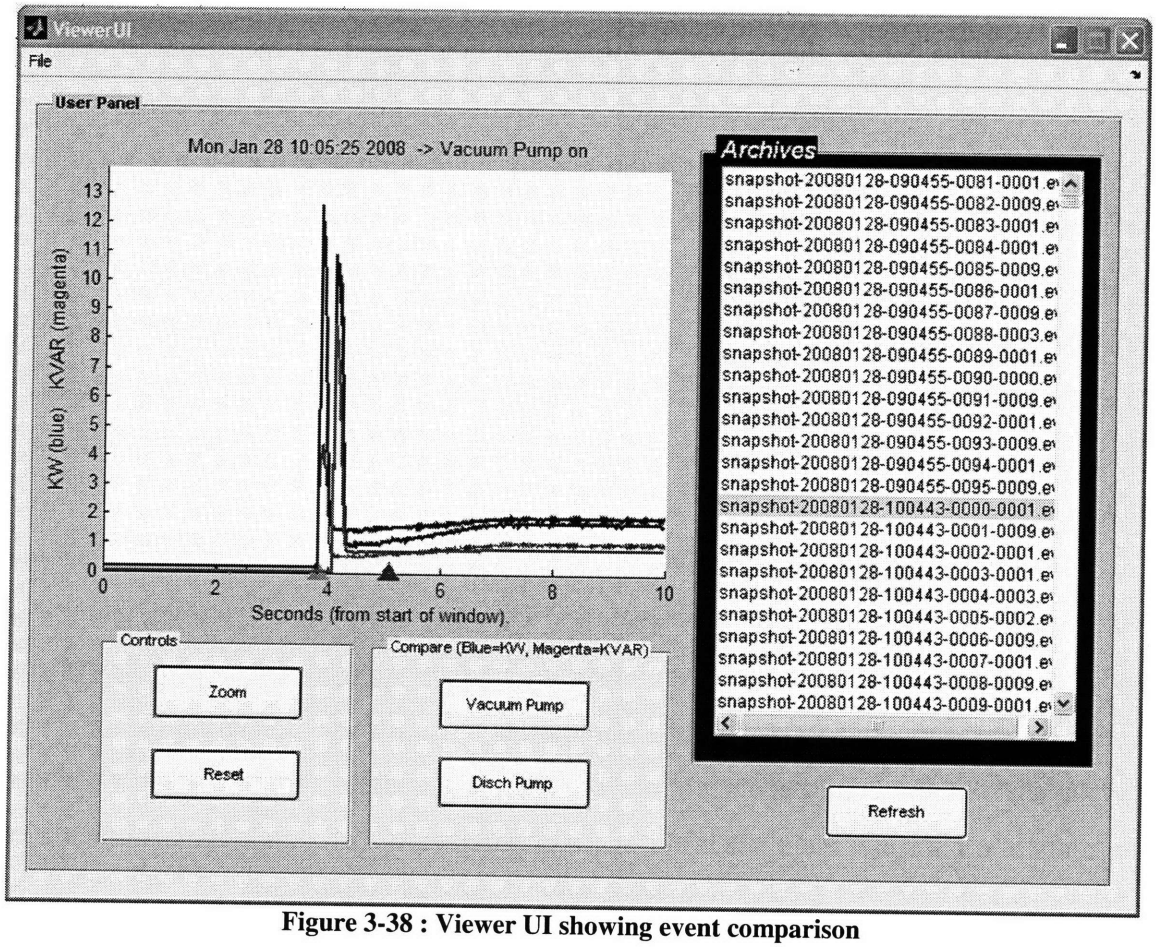


Figure 3-38 : Viewer UI showing event comparison

3.10 Software Tuning Parameters

The *ginzu* classifier makes decisions by comparing various characteristics of the power signal to known or expected values. These measures of comparison require numerical thresholds called tuning parameters, which are system-specific and typically adjusted to maximize classifier accuracy. Each tuning parameter has an affect on classifier performance. As tuning parameters are defined, the classifier becomes more tuned to a specific dataset or application. For example, correlation scores are compared to tuned values, to determine if a transient shape represents a template shape. Obviously, this value affects classifier performance. If the value is increased, fewer events will be classified as vacuum pumps but the events that are classified will possess very similar shapes.

From an application standpoint, tuning parameters are undesirable. The application becomes much less portable as each new implementation requires a time-consuming setup process where the operator has to adjust and test until his system works

properly. Additionally, the software robustness is reduced as the decision-making process becomes very sensitive to changes in system operation. While this could be seen as a positive as changes in system operation may indicate system faults, excess tuning parameters make it more probably that normal variations in system operation will result in misclassifications. Tuning parameters also reduce robustness as modifications to the parameters can cause the system to become unstable. Consider the example where the classifier checks post-event power. Since the power is stored in an array, an increase in this value will eventually cause an out of bounds array reference – causing an exception fault.

As the *ginzu* application evolves, additional classifier modules could be generated (e.g., Auxiliary Seawater, Low Pressure Air Compressor). Every tuning parameter represents another branch in the decision tree and increases the complexity of the system. In that sense, the number of tuning parameters is a direct indication of the code complexity. As the code becomes more complex, it becomes more difficult to debug. The reduction and intelligent selection of these parameters is a continuous challenge in decision-tree classifiers. A review of *ginzu* was conducted to determine the effect of each significant tuning parameter. In each case, the following determinations were made.

- What is potential for instability if this parameter is changed? Can the value cause *ginzu* to crash if it is improperly set? Should this parameter be adjustable by the user or hidden in the source code?
- What is the effect of this parameter on system operation? How does changing this value change the classifier performance?
- Can this parameter be defined as a function of an existing tuning parameter? Can this parameter be learned?

This analysis was performed for the CHT and RO system classifiers and is summarized in table format in Appendix [B] and Appendix [C]. This review showed that the *ginzu* classifier is defined by 36 key tuning parameters. Seven of these parameters were identified to be invariant as changes could cause system instability. The remaining 29 tuning parameters were then reviewed to determine the parameters that could be combined or identified. This review indicated that these 29 parameters could be reduced to the 17 parameters shown in Table 3-6.

Detection Threshold	The required output of the change of mean filter required to generate an event. This value can be set by command line parameter.
Zero power level	If power is below this level, all system loads are OFF.
Vacuum Pump P and Q	Expected Steady State P and Q following a Vacuum Pump (2 parameters)
Discharge Pump P and Q	Expected Steady State P and Q following a Discharge Pump (2 parameters)
RO Low Pressure Pump P and Q	Expected Steady State P and Q following a RO Low Pressure Pump (2 parameters)
RO High Pressure Pump P and Q	Expected Steady State P and Q following a RO High Pressure Pump (2 parameters)
AC Gain Threshold (α_T)	Used in correlation score calculation. See 2.2.2 Classification based on Shape
Residual Threshold (γ_T)	Used in correlation score calculation. See 2.2.2 Classification based on Shape
LP Score Threshold	Minimum correlation score to classify an event as a LP Pump start
HP Score Threshold	Minimum correlation score to classify an event as a HP Pump start
VP Score Threshold	Minimum correlation score to classify an event as a Vacuum Pump start
DP Score Threshold	Minimum correlation score to classify an event as a Discharge Pump start
P/Q Mismatch (V-7 Bypass Adjustment)	See 3.6.3 RO Classifier Logic, Bypass Valve Adjustment

Table 3-6 - Summary of Final Tuning Parameters

These modifiable values were then moved out of the body of the source code and into the function headers to allow for easier modification. In a future version, each of these values could be individually modifiable by specific command line parameters or moved to an external header file that could be read when *ginzu* was invoked. The most comprehensive method to eliminate tuning parameters is the construction of a decision-tree classifier that learns parameters as it operates. In this sense, the parameters exist but are hidden to the user as they are learned by the classifier – not manually set by trial and

error. This *Ginzu2* classifier (discussed in 5.2.1 *Ginzu2*) is an application of a learning classifier on NILM data.

3.11 Software Development and Lab Testing

The *ginzu* application and the *GinzUI* were beta tested on three 'Red Team' datasets of archived CHT data: Seneca 2006, Seneca 2003, and Escanaba 2007. These datasets included over 6,000 hours of historical system data and in excess of 400,000 CHT transients. The data also included at least one occurrence of each known CHT system fault. RO data sets are slightly different as transients occur in sequence during startup or shutdown – but steady state operation causes very few events. RO classifier performance was tested (and improved) by a partial review of the Seneca 2007 and Escanaba 2007 patrol data.

3.11.1 Classifier

A clean dataset, such as Seneca 2006, consists of nearly all well-shaped events that cause expected power changes. There is significant duration between most event indexes – so few events become trapped in lock-out windows. As classification accuracy remains high, state information remains valid that further improves accuracy. Since this dataset was mostly 'normal' events, the classifier performed nearly perfect.

On the other side, datasets such as Escanaba 2007 are a very difficult test for the classifier. This data set is plagued by numerous system faults (leaks, probe failures, clogs), which generate events at much higher frequencies. Events overlap more often and detections are lost in lock-out windows. Cycling pump starts tend to be poorly shaped and provide poor correlation scores. Additionally, the Escanaba NILM had been monitoring a phase that also contained control and indication power. This resulted in a significant 'standing' system power, which led the classifier to never meet the ALL OFF condition as power never decreased to zero. The inability to reach an ALL OFF condition prevented the frequent state corrections that help to maintain valid state data, which consequently lead to cascading state-based classification errors.

In each dataset, a random sampling of *event files* was reviewed to confirm that *ginzu* classification was accurate and consistent. If recurring classification problems

were found, the classifier was modified or tuning variable were adjusted. *Ginzu* constantly evolved during this testing and various features were added to deal with recurring problems. This process led to a stable Escanaba release version of the software and ultimately to the final *ginzu* classifier described in chapter 3 of this thesis.

3.11.2 Diagnostics

Pre-deployment diagnostic testing was executed by sequentially running the complete red team data sets through the *ginzu* classifier and then directing the generated *event files* to *GinzUI* for diagnostics. Many of system faults in the red team datasets had been documented in earlier research ([12, 15, 11]); those results could be compared with the GUI diagnostics. This extensive beta testing covered thousands of hours of archived data and a multitude of revisions to the diagnostic software.

4. *Ginzu* Field Testing and Performance

Chapter Four describes the deployment of a complete diagnostic NILM system on the USCGC Escanaba (Figure 4-1). The system was deployed in January of 2008 until late March of 2008. No major bugs or application crashes were encountered during this period.

Figure 4-1 : USCGC ESCANABA (WMEC-907)

4.1 USCGC Escanaba (WMEC-907) Implementation

The installed system consisted of the *ginzu* event classification software and the *GinzUI* diagnostic GUI running on an IBM T60 tablet-style laptop under the Ubuntu 7.10 operating system. These installations included a hardware install of the newly designed Ethernet-ready NILM sensor box (designed and built by J. Rodriguez) and an IBM T60 ThinkPad laptop with a full NILM software front end. The complete diagnostic NILM system is shown in Figure 4-2.

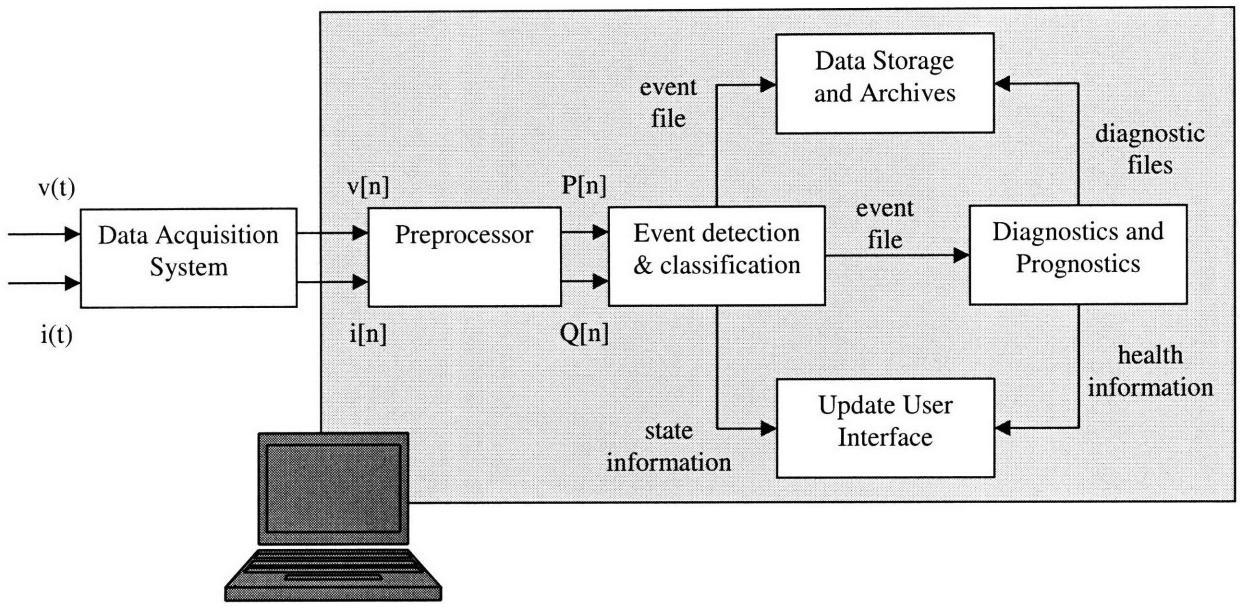


Figure 4-2 : Diagnostic NILM System showing data acquisition, signal processing and diagnostics as applied to a shipboard waste disposal system. Raw voltage and current are acquired by the NILM sensor hardware and sampled at 8 KHz. This data is then passed via Ethernet to the NILM computer and preprocessed in preparation for classification and diagnostics.

The data acquisition occurs in the NILM sensor box (Figure 4-3). Raw voltage and current is sampled at the input of the CHT controller at an 8 kHz sample frequency. This signal is then directed to the attach NILM computer (Figure 4-4) that provides the required signal processing, event classification, diagnostics and data storage.

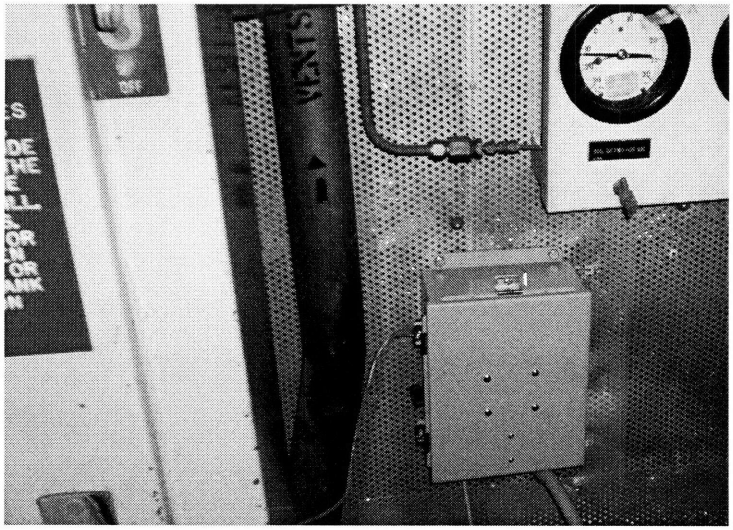


Figure 4-3: NILM Sensor Box. This hardware interface receives raw voltage and current signals (v(t), i(t)) from the CHT controller, samples them at 8 kHz and sends the resulting signal to the NILM computer via Ethernet.

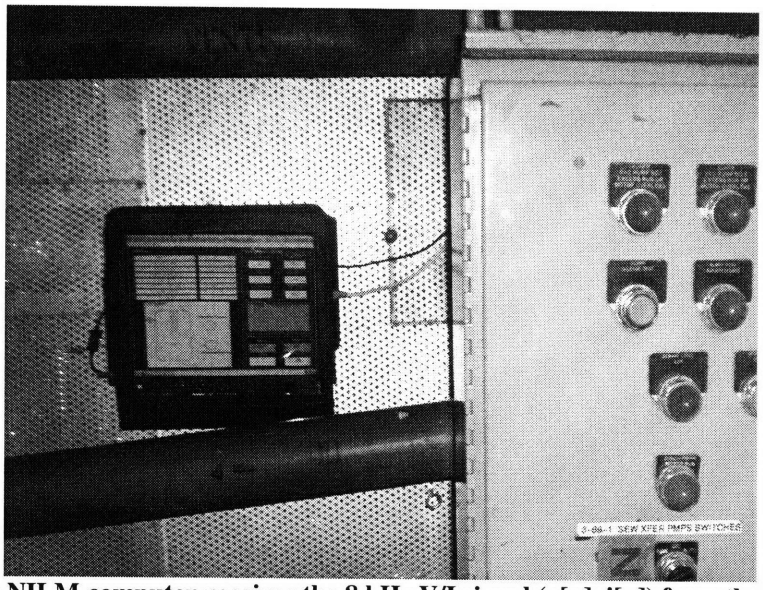


Figure 4-4: The NILM computer receives the 8 kHz V/I signal (v[n], i[n]) from the sensor box and performed required signal processing and diagnostics.

The actual Escanaba implementation was driven by two overarching requirements. The software would be provided with limited (or no) training - so virtually all aspects of data acquisition and normal operation needed to be automatic. Despite having limited training, the crew would be asked to observe the software, to attempt to use the software, and to provide a commentary of its usefulness. So, the package needed to be (1) user friendly, (2) stable, and (3) provide an automatic restart function to recover from any scenario where the program execution was stopped or the UI was closed. The power data generated from the NILM needed to be recorded in one-hour long data files. Arguable, this should not be necessary as *ginzu* extracts the useful information by creating individual *event files*. However, since this was the initial field implementation of the classifier, the source data needed to be retained for validation.

The data acquisition and restart functionality was implemented by J. Paris in various Linux shell scripts and drivers. These scripts provide an interface between the new TCP/IP data from the NILM Interface box, the existing NILM data acquisition software [14] and the *ginzu* application. These files are included in the NILM Installation package and highlights are discussed here.

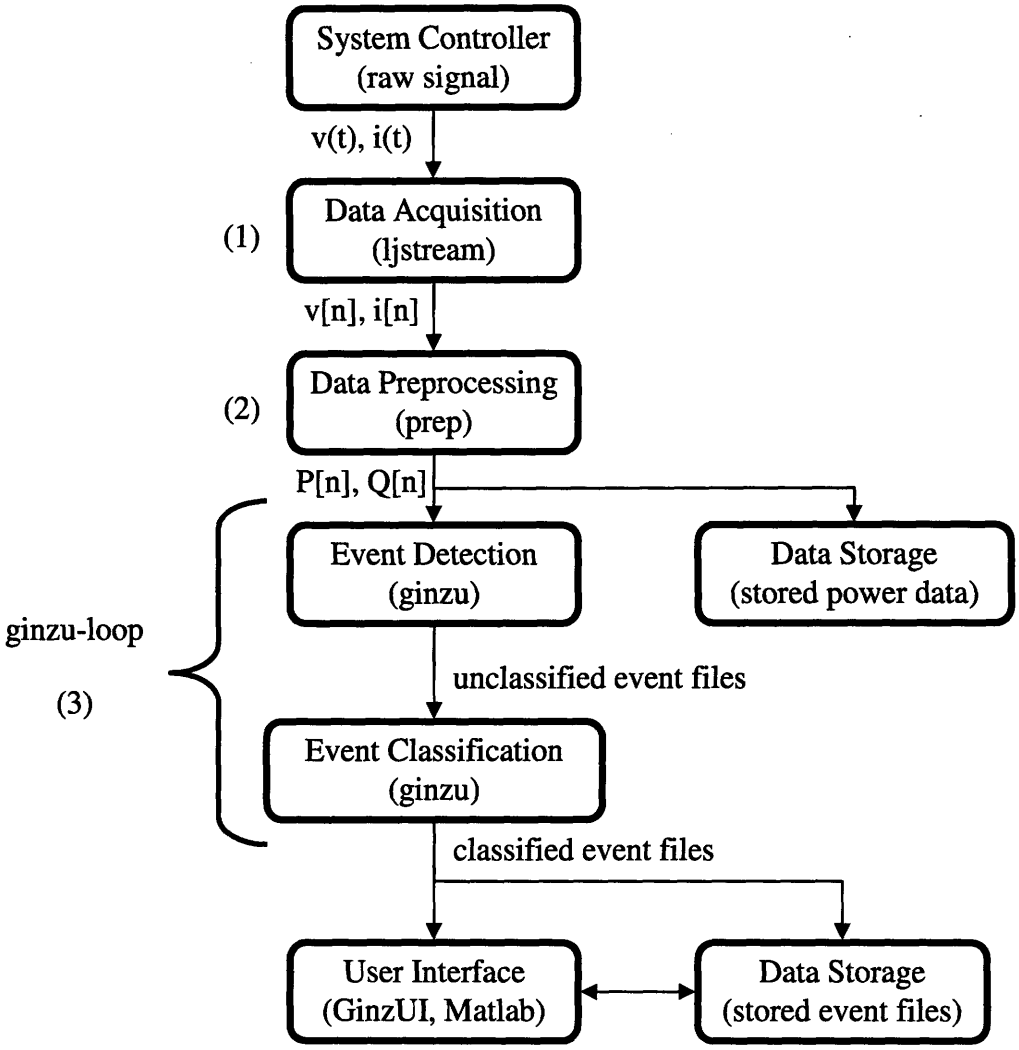


Figure 4-5 : NILM Software Architecture. The numbers in parentheses are referenced to the discussion below.

(1) ljstream

The command-line executable ljstream reads raw current and voltage data from the Ethernet port and directs this data to the terminal (stdout). Command line arguments allow the user to set specific attributes such as sampling rate, zero-level, and number of channels.

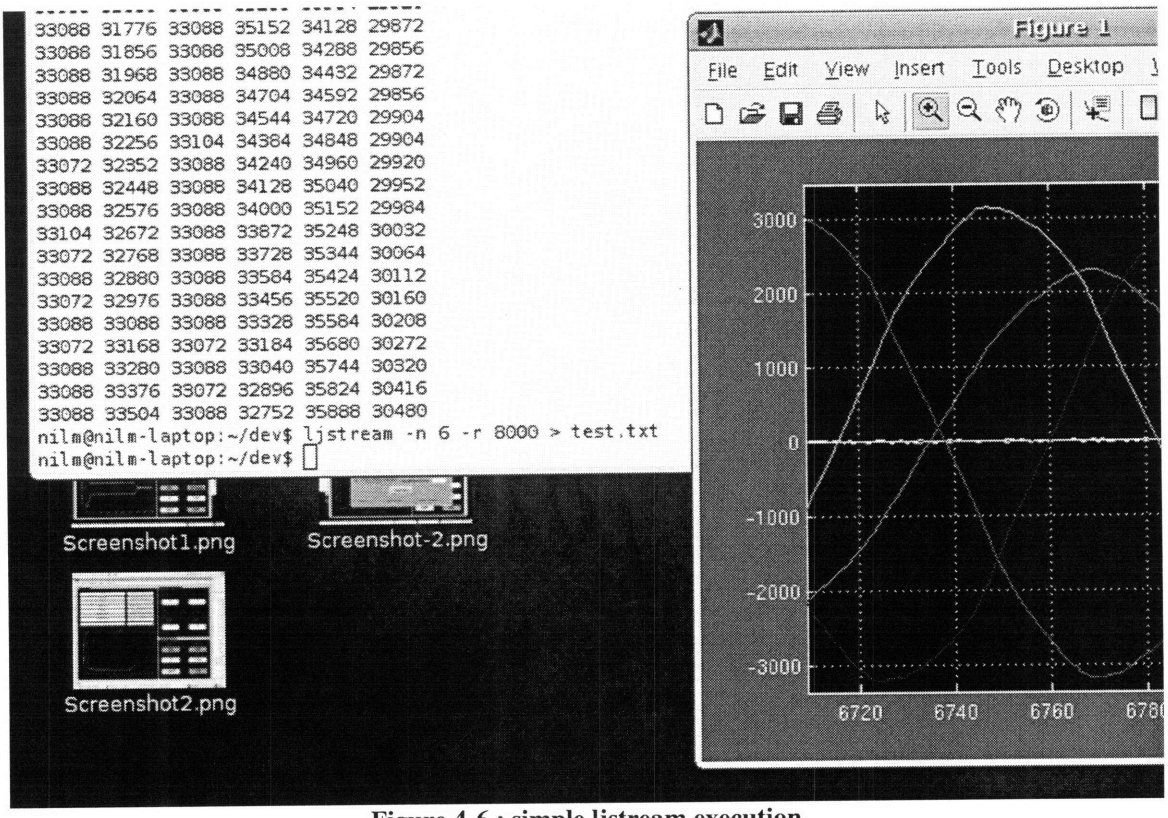


Figure 4-6 : simple ljstream execution

Figure 4-6 shows a simple execution of ljstream set to receive six channels with an 8 kHz sampling rate. The partial figure shows three phases of voltage and a single current (in magenta).

$ ljstream –n 6 –r 8000 | awk '{print $4/16 " " $2/16}'| prep | grep –v '#' | ./ginzu m 3 I 0

The Linux command line above shows a more standard *ginzu* application. Since a standard NILM will provide three channels (i.e. three phases) of voltage in addition to at least one channel of current, the output of ljstream is manipulated to send the correct two channels to prep. This is accomplished with the Linux awk function, which in this example, extracts the 4th and 2nd column and pipes them to prep. Some conditions cause prep to insert comments into the output. As *ginzu* strictly requires incoming data to be numeric and eight-column, grep is used to extract these comments. The pipe is then directed to *ginzu*.

(2) prep

The executable *prep* transforms streaming voltage and current data into the associated real and reactive power components. The implementation and use of prep is beyond the scope of this document and is described in [5].

(3) ginzu-loop

In the default mode, *ginzu* exits after reading 432,000 rows of power (one hour of data at 120 samples per second). The *ginzu*-loop script simply restarts *ginzu* every time it exits so that realtime streaming power data is continuously read. This script is executed at every startup and every four hours by the restart-everything script. Note that state information is not retained when *ginzu* closes, so state resets to ALL OFF at the end of every data set. This is normally not a significant problem as the state correction function quickly corrects the state information, but this feature could be implemented in future versions.

restart-everything

The restart-everything script provides automatic crash recovery. Since a bug or inadvertent operator action could stop data acquisition or data processing, restart-everything automatically kills all NILM processes and restarts them every four hours. These processes include ljstream, prep, *ginzu*, and the *GinzUI*. Restart-everything is invoked by the Linux cron daemon and timing can be adjusted with crontab.

4.2 Performance Results - Escanaba 2008

The *Ginzu* classifier automatically identified over 100,000 CHT and RO transients. Additionally, the associated GUI performed realtime diagnostics that identified the occurrence of:

- A continuous fault in the CHT tank probe level indication circuit causing over one thousand cycling discharge pump events (See 3.9.2 CHT Diagnostics – Tank Level Indication).

- Two situations where the vacuum pumps were not reaching normal power level due to probable clogging (See 3.9.2 CHT Diagnostics – Vacuum Pump Clog Detection).

These results represent the first instance that NILM data has been automatically processed in real time to recognize a system fault and provide user feedback.

4.2.1 CHT Classifier Performance

CHT Classifier accuracy was assessed by randomly sampling and manually classifying 1500 CHT events from the Escanaba dataset. In these 1500 events, 62 events were determined to be misclassifications[10]. This equates to 95.9% accuracy. The 62 events provide a representative sample of common classifier errors.

Classification Errors

Missed Double Up power events – In situations where more than one event occurred in the same window, the classifier would internally classify the event but would fail to create an *event file*; consequently the data would not get passed to the UI. This was caused by caused by an omission in the source code and has been corrected.

[10] Events which were classified as 'Unidentified Down power' or "Unidentified Up power' were considered to be correct classifications if the event was actually a change in power that was not a 001, 002, 003, 004, or 009 events.

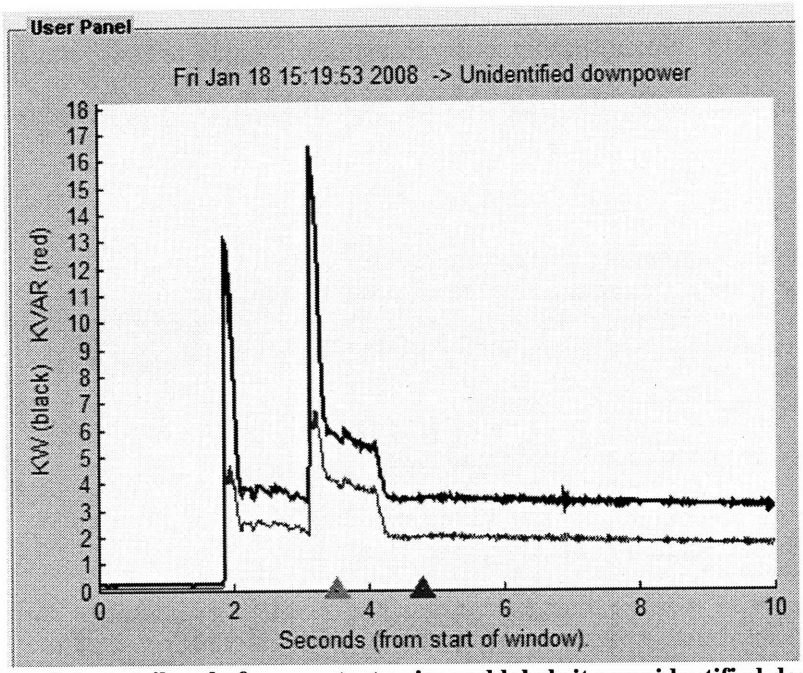


Figure 4-7 : *Ginzu* detects tail-end of pump start noise and labels it as unidentified down power. The preceding double vacuum pump ON event (011) was detected and classified but *event file* was not created due to a code problem.

Unclassified Pump On events – The classifier failed to identify 44 ON events and labeled them as unidentified load events (007 or 087). This failure is caused when the gamma threshold (γ_T) is improperly set – resulting in overly sensitive scoring (2.2.2 Classification based on Shape).

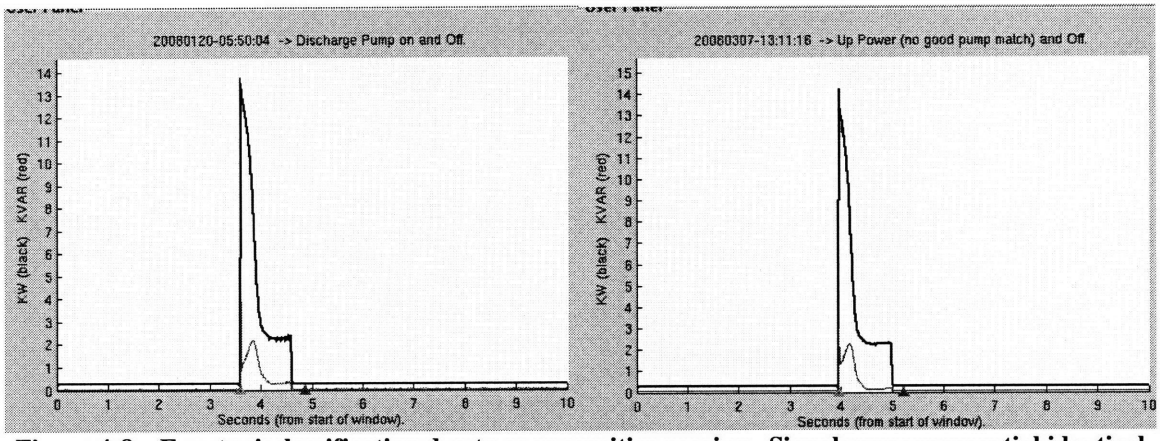


Figure 4-8 – Event misclassification due to over-sensitive scoring. Signals appear essential identical but the resulting correlation scores are significantly different. This is caused by improperly set threshold values in the LS matching algorithm.

Figure 4-8 shows the importance of tuning the threshold for the shape matching algorithm. In this example, both pump starts appear to be similarly shaped discharge pump events. The *ginzu* log (Figure 4-9) shows the first event is scored as a 0.591. The

second event is 0.243. Since the detection threshold is tuned to 0.45, the first pump start was correctly labeled while the second start was labeled as an unknown cycling event (087).

```
Classifying event at <360552> - DeltaP_event =[10.658]
 -> Checking for double ON event -> No double event detected.
 -> Classifying ON -> Running transient classifer - > VP template Comp Gain = [-0.216]
 -> Not vacuum, Checking Discharge., DP template Comp Gain = [0.591]
360552  Discharge  +1
Probable Cycling load - no power change.
360552 Dis -1
Writing snapshot-20080120-050001-0035-0043.evt

                                    ------ break -------

Classifying event at <81234> - DeltaP_event =[10.501]
 -> Checking for double ON event -> No double event detected.
 -> Classifying ON -> Running transient classifer - > VP template Comp Gain = [0.074]
 -> Not vacuum, Checking Discharge., DP template Comp Gain = [0.243]
 Probable ON event, no match
Probable Cycling load - no power change.
81234 Load Cycled.
Writing snapshot-20080307-130001-0012-0087.evt
```

Figure 4-9 - Excerpt from *ginzu* log file showing correlation scores for above pump start examples

Missed events or Misclassification due to closely spaced events – The remainder of the failures were exclusively related to the inability of *ginzu* to successfully classify closely spaced events.

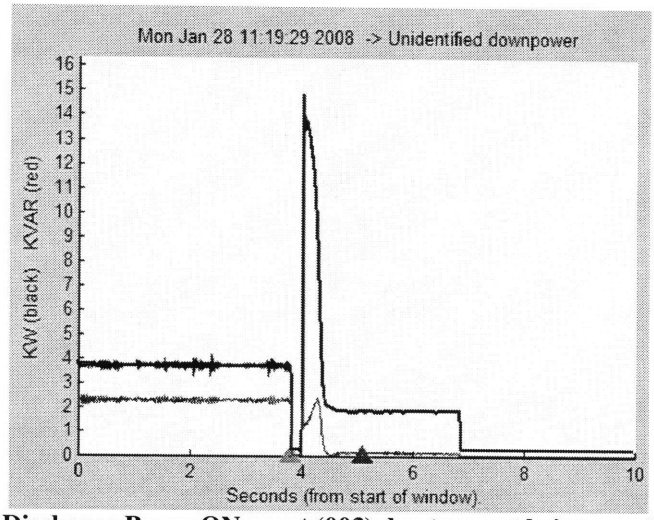


Figure 4-10 - Missed Discharge Pump ON event (003) due to event being caught in lockout window.

Unclassifiable Events

Additionally, the classifier generated many 'non-classifications' of events (i.e., transients where the shape and power change did not match any expected system event). In almost

all cases, this occurs when the unusual noise after a vacuum pump start is detected as system event. Figure 4-11 shows an expected vacuum pump start shape and an initial starting transient followed by a slow rise as the pump pulls a steady vacuum. The pump reaches a normal running power of ~1.8 KW.

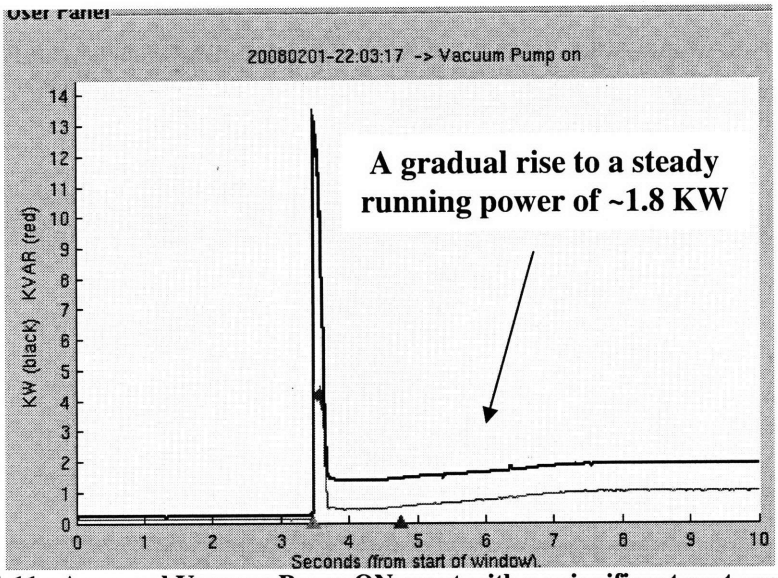


Figure 4-11 : A normal Vacuum Pump ON event with no significant post-event noise.

Figure 4-12 shows two examples of events that were correctly classified as vacuum pump starts based on shape changes. The lock-out window is shown by the location of the green arrows. Note that the pump shape templates only consider the 30 data points after the event (1/4 of a second after the event index) so the noise does not affect the correlation score. Since the lockout window essentially disables the detection algorithm, the noise disruption inside the event window does not cause any additional event.

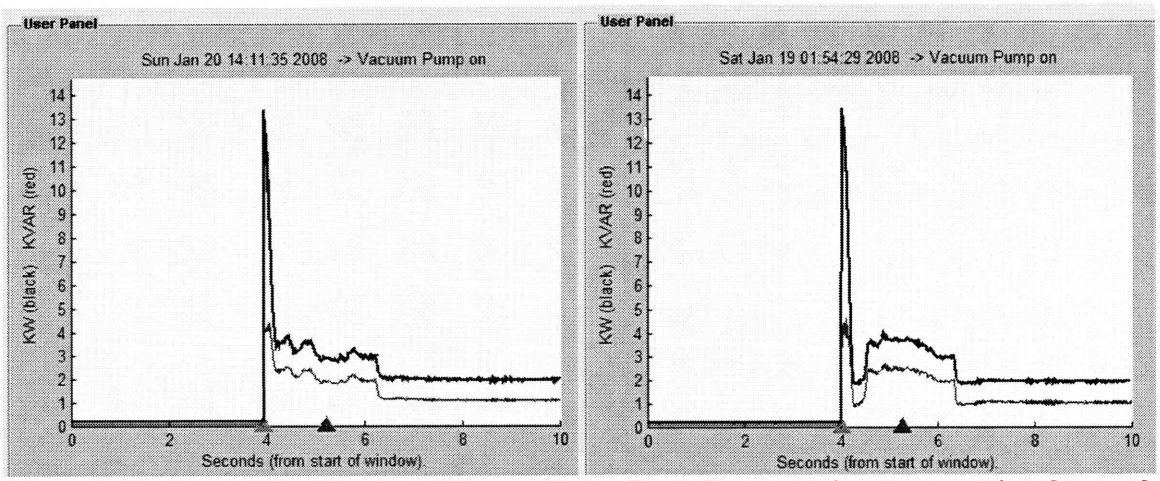


Figure 4-12 : Correct Classification of Vacuum Pump Start. Post-start signature contains abnormal noise that does not affect correlation score.

As the lockout window closes and the detection algorithm restarts, the remaining noise is detected as a new transient. Since this transient does not have the shape of any pump and the power change does not fall into a known category, the event is unidentified. Figure 4-13 show two examples of this behavior. This classification does not adversely affect the performance of diagnostics associated with the user interface as unidentified events are not considered by the diagnostic modules.

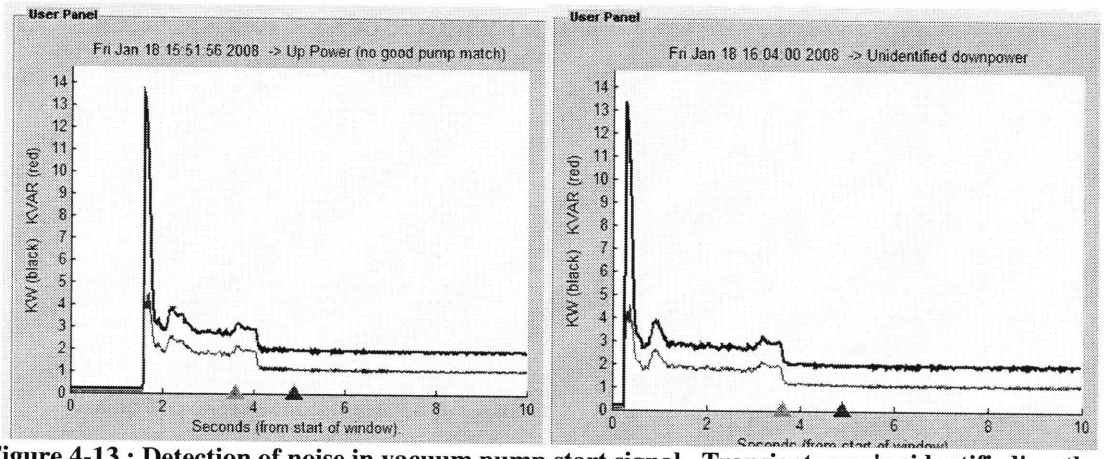


Figure 4-13 : Detection of noise in vacuum pump start signal. Transients are 'unidentified' as they do not meet shape match or power change requirements of known events.

4.2.2 CHT Diagnostics – Field Results

Automatic Diagnosis of Probe Failure

The Escanaba's CHT NILM was installed at 1400 on January 18, 2008. At 1914, the NILM diagnostics module alarmed that a probe failure had been detected due to excessive cycling discharge pump runs. This fault condition remained throughout the Escanaba's three month patrol.

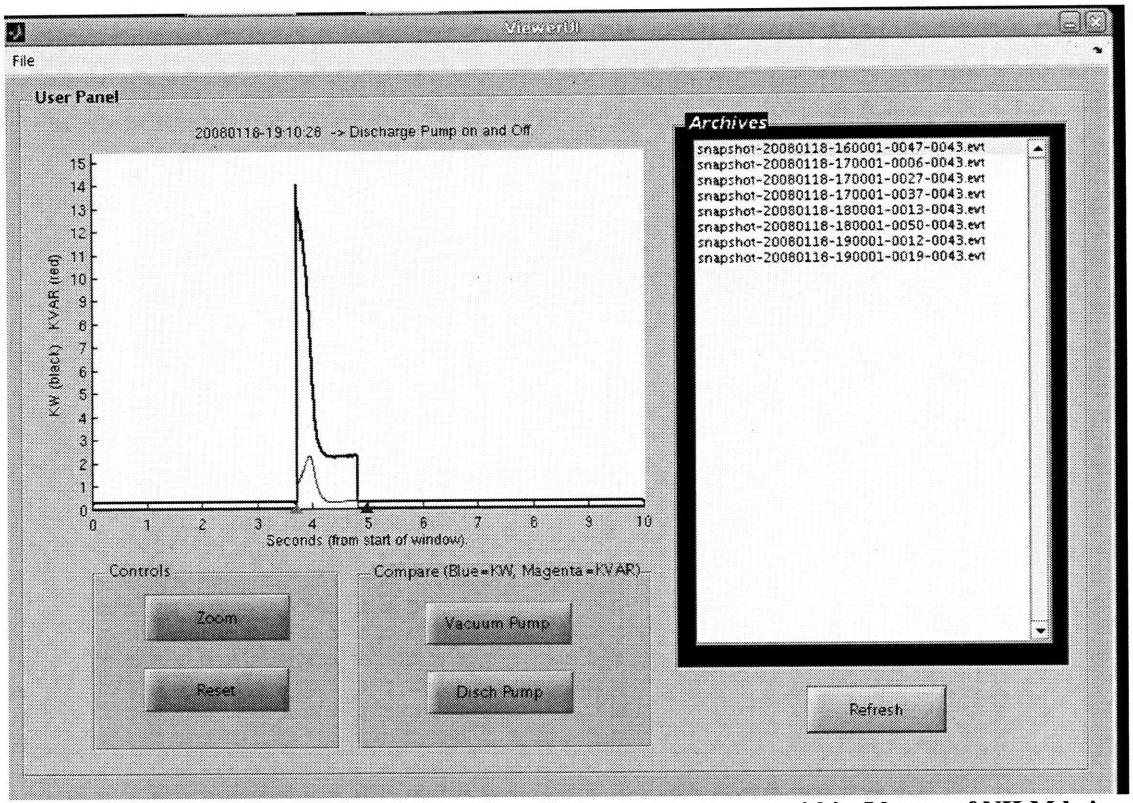


Figure 4-14 : ViewerUI showing eight cycling discharge pump runs within 5 hours of NILM being installed. These short 'burst' runs are abnormal and a clear indication of a system fault.

This was an automatic diagnosis of a problem that had been documented by Piber in [15]. In his 2007 thesis, Piber discussed the temporary repair of Escanaba tank probes and the unsuccessful restoration of the system. In his description, he showed that the crew had repaired the TLI probe and placed the system back online. He then said that "*Over the following days the NILM was used to check the operating schedule of the pump to determine if the temporary correction had indeed fixed the problem. Unfortunately for the Escanaba the discharge pump problems continued indicating that there were larger problems with the circuitry of the panel itself.*" When it was discovered that the probe repair had failed to correct the problem, Piber conducted a review of the CHT electrical controller where he manually filled the tank with water and observed the results. This lead to a May 2007 discovery that the CHT controller's wiring had been modified from the design specification. The fact that this condition remained in January 2008 indicates that repair efforts between May 2007 and January 2008 had not been successful.

Automatic Diagnosis of Vacuum Pump Clogging

The GUI diagnosed two vacuum pump clogging events during the three month patrol.

- 1707, 02/11/08 → 1806, 02/12/08 – detected 177 clogged vacuum pump events
- 0107, 02/20/08 → 1130, 02/21/08 – detected 167 clogged vacuum pump events

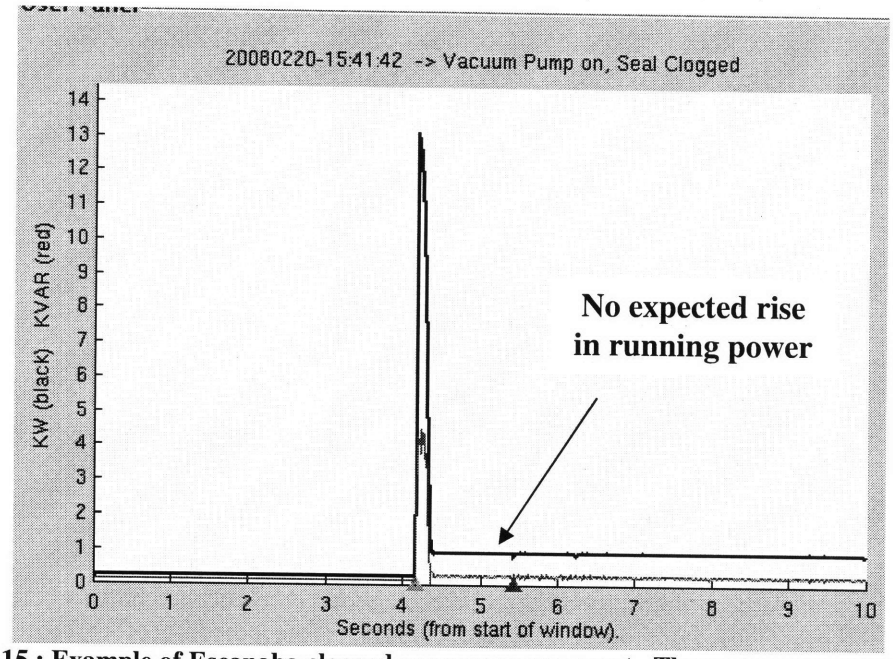


Figure 4-15 : Example of Escanaba clogged vacuum pump event. The motor generates a normal starting transient but the steady state power does not ramp up to a typical value. This occurs due to a blockage in the pump suction or priming line.

As described in 3.9.2 CHT Diagnostics, a clogged vacuum pump occurs when the running pump fails to pull adequate system pressure. This normally occurs due to a clog in the pump suction or priming line. As header vacuum continues to fall, the second pump starts to aid the first pump. After the second pump starts, both pumps work to pull vacuum to the high setting (normally 18"). Figure 4-16 shows an hour of Escanaba CHT data during a clogged pump period. The figure shows the characteristic 'double start' sequence and it can be observed that the first pump does not attain the normal running power that we see in Figure 4-11. In both cases, the clogged condition eventually cleared with no operator action and the system returned to normal operation.

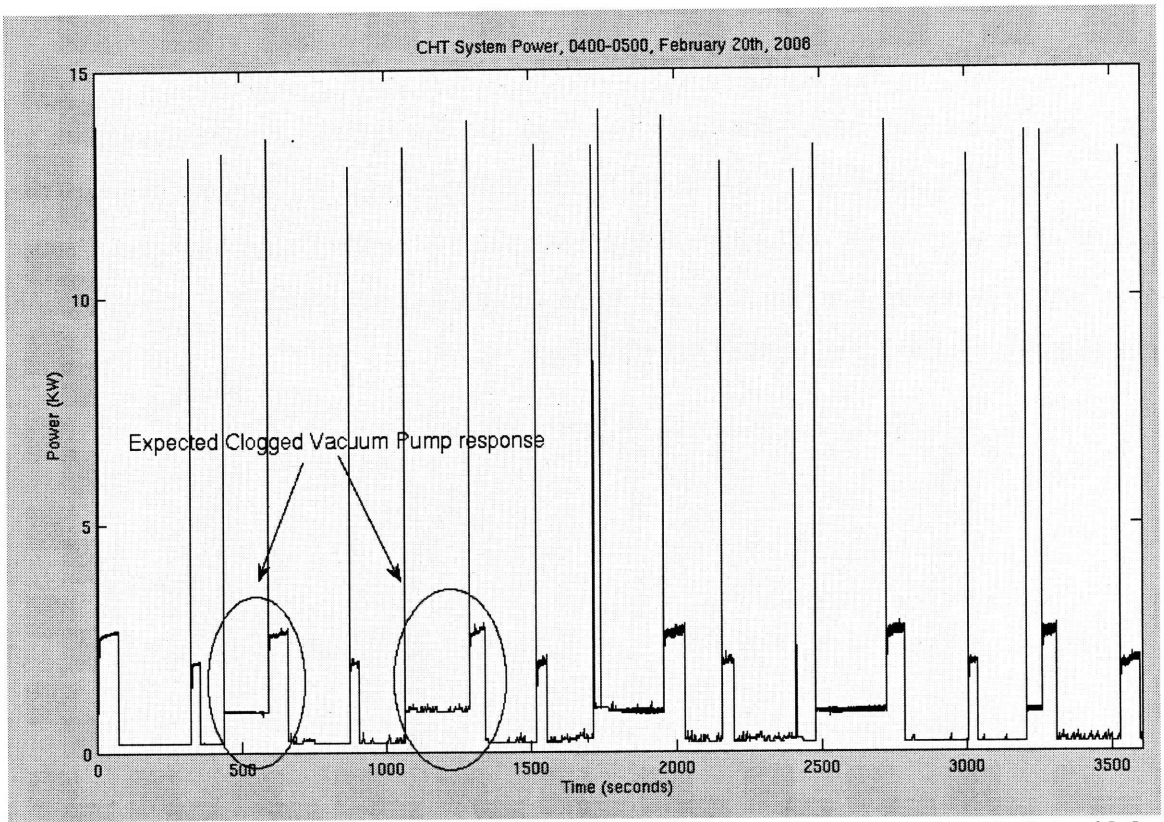


Figure 4-16 : Escanaba Vacuum Pump clogging problem. The second vacuum pump starts to aid the first pump as it is unable to pull vacuum due. This system failure was automatically diagnosed by *GinzUI* diagnostics module on two separate occasions during the Escanaba 2008 patrol.

4.2.3 RO Classifier and Diagnostics

A detailed discussion of RO classifier results is provided by P. Branch in [1]. In that thesis, Escanaba RO classifier was assessed to be a 92.1% accurate. This was primarily the result of misclassifications of HP 'phantom' starts. These results drove changes in RO classifier logic (i.e. added shape change logic and considerations for phantom start transition discussed in 4.3.2). When the archived Escanaba dataset was re-classified using the improved logic and an associated user interface, the classifier accuracy was 99.9%.

4.3 Resulting Changes

The results of the Escanaba deployment resulted in modifications and improvements to the classification and diagnostic software. The notable changes are summarized here.

Reverse Osmosis Template Matching

The initial *ginzu* software deployed on Escanaba did not include template matching as part of the RO decision logic. Since the RO FSM was considered very predictable, the classification was based completely on change in relative power change and initial state. However, classification robustness was improved through the addition of correlation templates for the RO LP Pump and RO HP Pump.

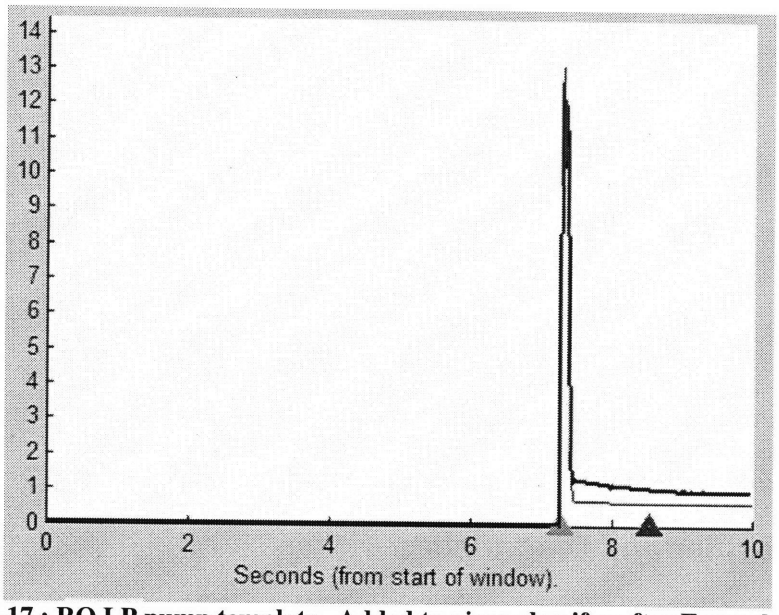


Figure 4-17 : RO LP pump template. Added to *ginzu* classifer after Escanaba patrol.

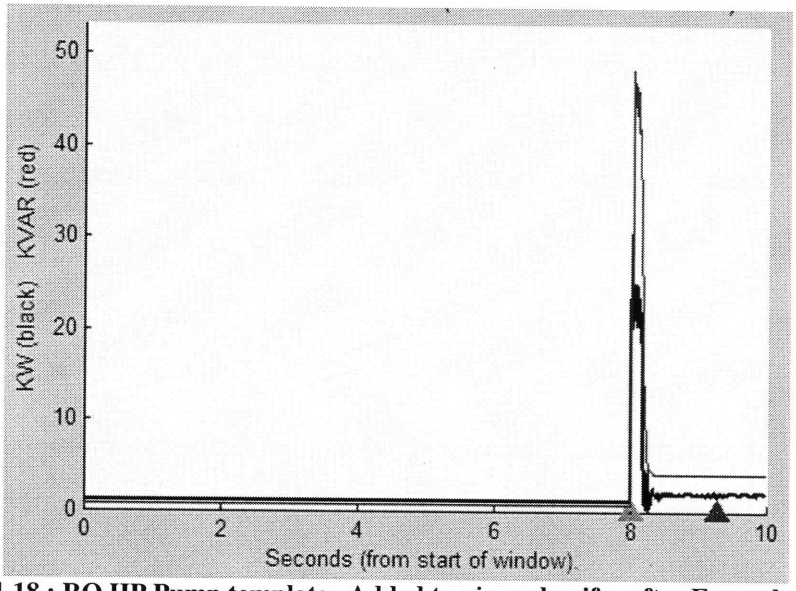


Figure 4-18 : RO HP Pump template. Added to *ginzu* classifer after Escanaba patrol.

Consideration of HP 'Phantom' Start Logic

During the Escanaba deployment, an LP start was considered to be the only valid state change from an ALL OFF condition. Consequently numerous HP pump 'phantom' starts were incorrectly classified as an LP pump starts. This logic was corrected through minor modifications to the decision tree, which included an option to classify as a 'phantom' start if there was a sufficiently strong correlation score and sufficiently high change in power (described in 3.6.3 RO Classifier Logic).

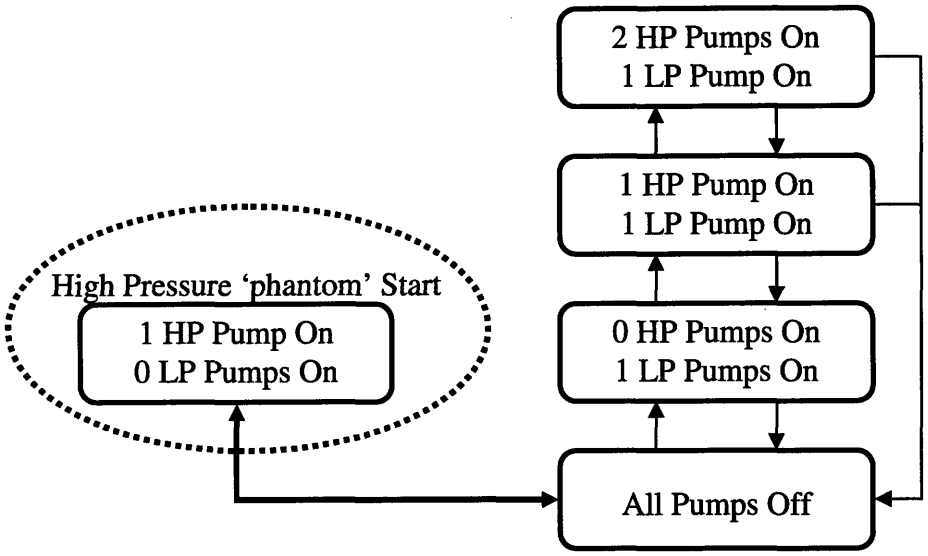


Figure 4-19 – Modified RO FSM including possibility for HP 'phantom' start. This FSM includes the possibility for a transition to a state where only a HP pump is running. This option was not present in the initial deployment of the classifier as this state was considered to be invalid.

Corrected failure to create *event files* on Double CHT Events

As discussed in the 4.2.1 CHT Classifier Performance, *ginzu* would fail to create *event files* if double events were detected. This was due to a set of code in the arbitrate_event_overlap() function that had been erroneously commented out.

Reduction and Relocation of Tuning Parameters in *Ginzu* Source Code

As discussed in 3.10 Software Tuning Parameters, key tuning parameters were moved to more convenient locations in the *ginzu* source code in order to improve future tuning efforts.

5. Observations, Future Work, and Conclusions

5.1 Observations

The performance of *Ginzu* on the USCGC Escanaba has shown that NILM is a maturing and capable technology. Both CHT and RO classifiers performed adequately and validate the use of decision trees for simple shipboard systems. Crew feedback was positive overall, specifically on the Reverse Osmosis system as the crew felt the automatic recording aspect of NILM would be beneficial while discussing unusual system operation with the vendor.

Unfortunately, the development and testing of these classifiers is very time-consuming – requiring trained programmers and system experts. A wide-scale implementation of NILM requires some additional degree of generality. New NILM classifiers need to be portable not only across different ships but also across ship systems.

5.2 Future Work

5.2.1 *Ginzu2*

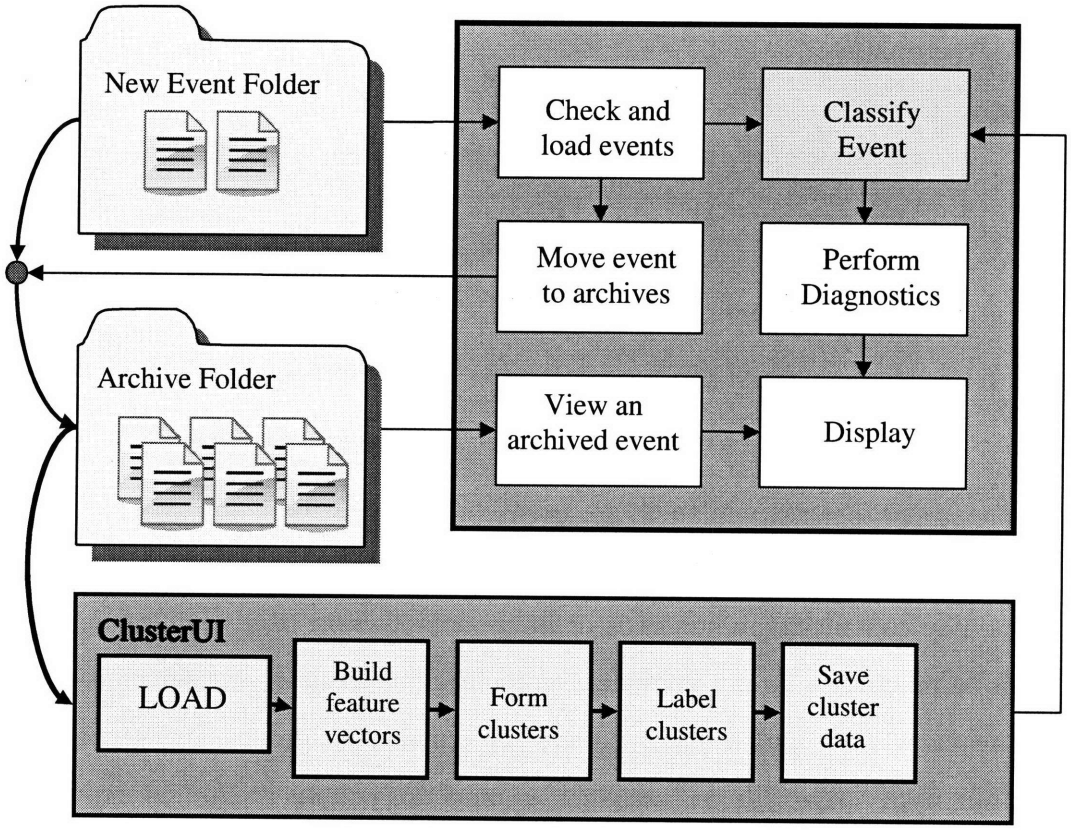


Figure 5-1 - *Ginzu2* Functional Diagram

The *Ginzu2* application consists of two programs. The first, *GinzUI*, is an evolution of the existing *GinzUI* software modified to support unclassified *event files*. The second, ClusterUI, provides the user with the ability to form Euclidean distance-based classification rules within the UI environment. The discussion below details these programs (Figure 5-1) and a hypothetical implementation on the Escanaba CHT system.

The *Ginzu2* clustering application utilizes the *ginzu* classifier in detect mode (mode 2). The NILM will be installed on a simple shipboard system and the detection threshold will be set based on historical power levels. As the system operates and transients are detected, *ginzu* will generate unclassified *event files*. Once generated, these files are read by the *GinzUI* and moved to an archive directory. No diagnostics are able to be performed at this point as the event classifications are unknown.

Initialization and Learning

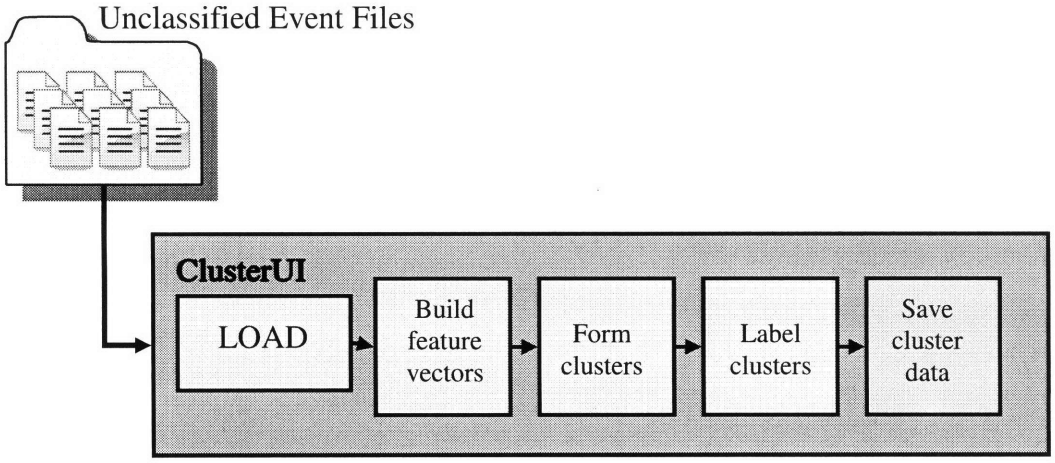


Figure 5-2 - *Ginzu*2 (during Initialization and Learning phase)

Over time, *event file*s accumulate in the archives. Once a sufficient number of *event file*s are generated, the ClusterUI is called and the events are manually labeled. In reality, the archives may contain many hundreds or thousands of *event file*s – and manual labeling will not be practical. To accommodate mass labeling, k-means clustering is utilized to force the *event file*s into a small number of classifiable groups. To accomplish this clustering, each *event file* is assigned a numeric feature vector to represent key classification data. This strategy is shown in Figure 5-2 and a screenshot of the clustering UI is provided in Figure 5-3.

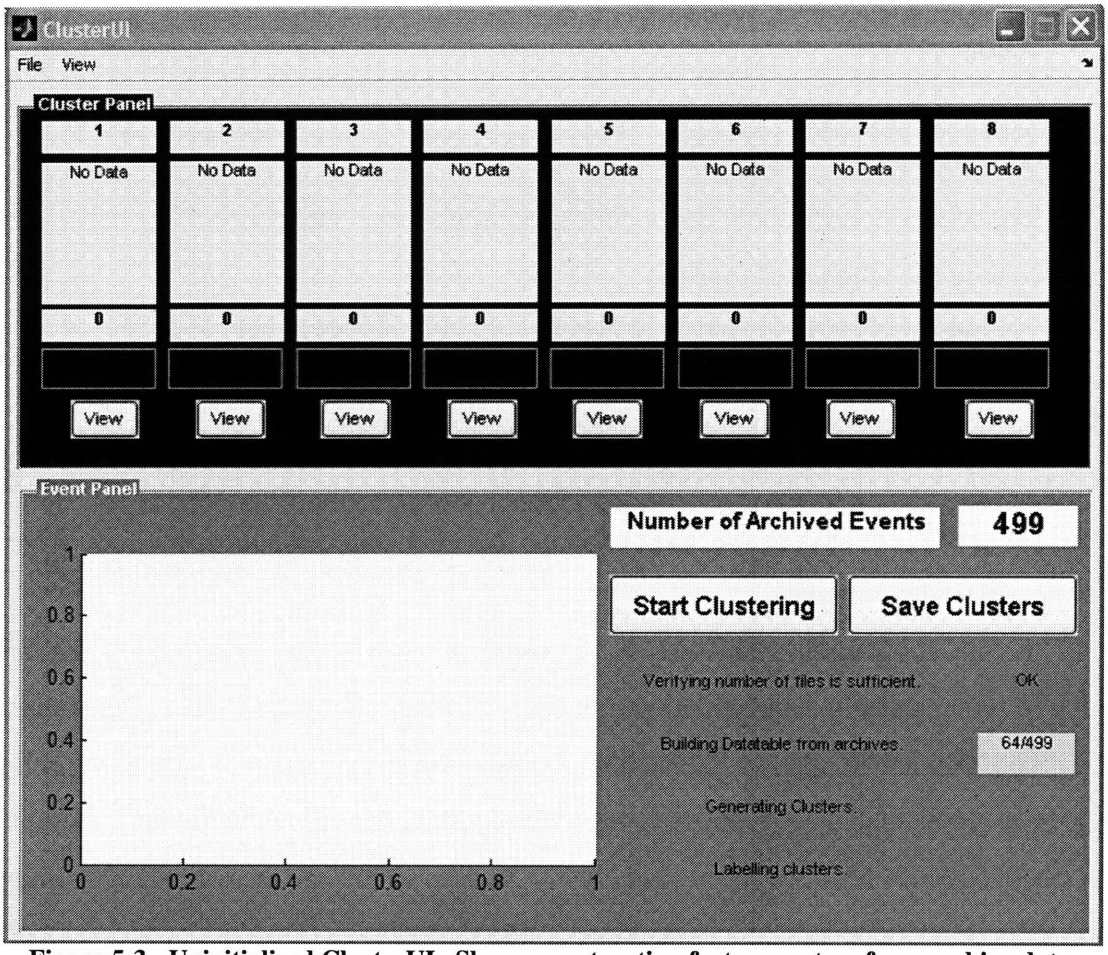


Figure 5-3 - Uninitialized ClusterUI. Shown constructing feature vectors from archive data.

The contents of the feature vector ultimately determine how distinct the clusters will become. It is tempting to build the feature vectors to include all information available to the decision tree classifiers – i.e. power change, system state, and correlation score.

ΔP and ΔQ are immediate and obvious choices for the feature vector as the event index and power levels are all immediately available in the *event file* – so change in power can easily be generated. State, on the other hand, is not useful at this point. State information implies that the classifier understands a FSM that models the system. In this case, the CHT system is treated as a new system and no FSM is defined. Similarly, shape correlation requires a library of known events for comparison.

A first pass could assume no a priori knowledge is available – and build the vector only based on ΔP and ΔQ. However, it is reasonable to assume that templates could be quickly and automatically generated with minimal operator aid.

For example, a simple NILM function could perform the following sequence of events.

1. While operating *ginzu* in detect only mode, request that the operator start a known piece of equipment. Allow the equipment to operate for a short period (~10 seconds) and then secure.
2. Repeat this process three times, thus generating three 'unclassified' *event files* that contain the pump start transient.
3. Compute and record a statistical average of the three ON events.
4. Generate a 30 index power template based on this statistical average.

Alternatively, if the operator is confident they can classify an event based on seeing the shape, the operator could simply choose from the archived *event files* that would then generate a 30 index template.

In this application, templates were assumed to exist for a vacuum pump and discharge pump starting event. Correlation scores (composite_gains) were calculated by the method provide in section 2.2.2 Classification based on Shape [11]. Relative power changes were calculated for both short-term (within 0.25 seconds of the event) and long-term (the difference between averaged power 0.50 seconds before the event and averaged power 1.0 seconds after the event) windows. The first 499 CHT events of the Escanaba 2008 patrol were assigned a feature vector, FV, as follows.

```
FV=[FV;  deltaP_event deltaQ_event deltaP_window deltaQ_window composite_gains];
```

The feature vectors comprised a [499x6] matrix that was then directed to the Matlab kmeans clustering algorithm12 to form eight clusters.

```
[IDX,C,sumd,D]= kmeans(FV,num_clusters,'EmptyAction','drop','Display','final');
```

The number of clusters was chosen arbitrarily, but should typically be a function of system size and number of possible classifications.

[11] The Matlab implementation of template matching is credited to R. Cox and K.D. Lee.
[12] K-means is implemented with the Matlab *kmeans* command (part of the Statistics Toolbox).

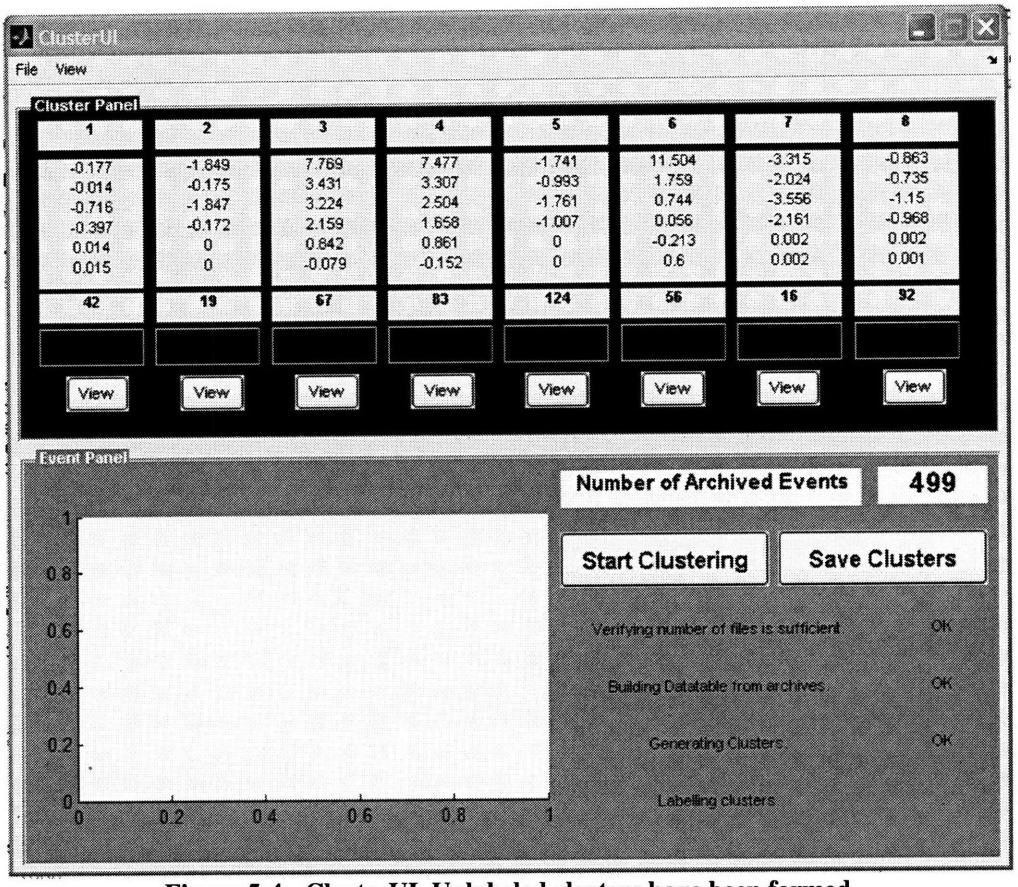


Figure 5-4 - ClusterUI. Unlabeled clusters have been formed.

The clusters are formed and displayed in the cluster panel (Figure 5-4). Each cluster centroid is displayed along with the number of *event file*s that are members of that cluster.

```
>> load clusters.mat
>> C , labels

C =

   -0.1773   -0.0140   -0.7159   -0.3968    0.0138    0.0153
   -1.8486   -0.1751   -1.8470   -0.1720    0.0000         0
    7.7688    3.4306    3.2240    2.1588    0.8425   -0.0793
    7.4773    3.3074    2.5045    1.6583    0.8612   -0.1524
   -1.7415   -0.9931   -1.7606   -1.0075    0.0000    0.0003
   11.5036    1.7594    0.7439    0.0557   -0.2135    0.5996
   -3.3154   -2.0244   -3.5565   -2.1606    0.0016    0.0019
   -0.8634   -0.7353   -1.1505   -0.9677    0.0015    0.0007


labels =

    'Noise'
    'DP Off'
    'VP On'
    'VP On'
    'VP Off'
    'DP On'
    'DP Off'
    'Noise'
```

After the clusters are formed, the user clicks the VIEW pushbutton and cycles through the events contained in that cluster. As the events share common features, they are normally the same type of event. The user then manually labels the eight clusters by typing in the label textbox. Once all clusters are labeled, the Save Clusters button is depressed (Figure 5-5), which saves the clustering centroids (C) and labels (labels) to a file, *cluster.mat*. ClusterUI can then be closed.

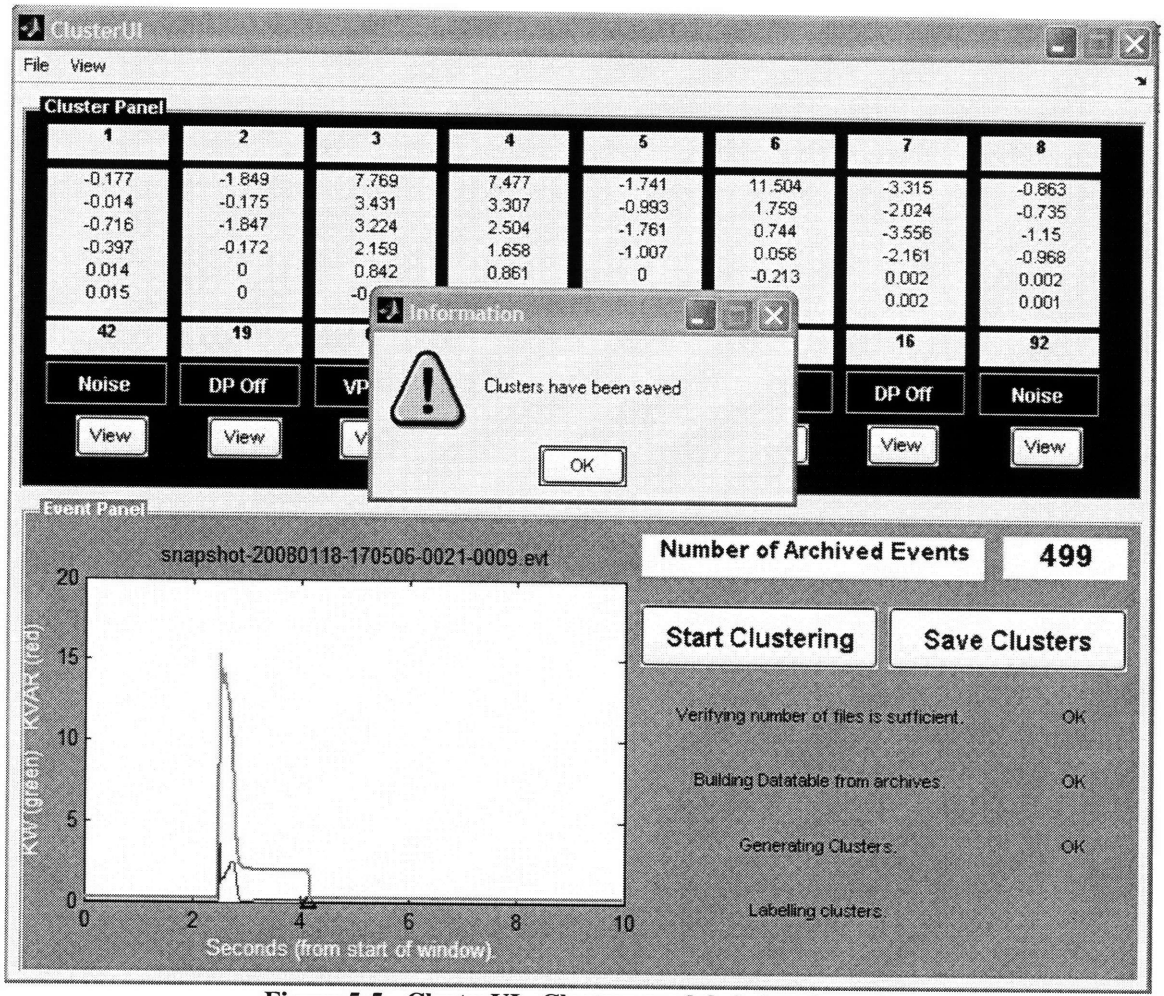


Figure 5-5 - ClusterUI. Clusters are labeled and saved.

Classification

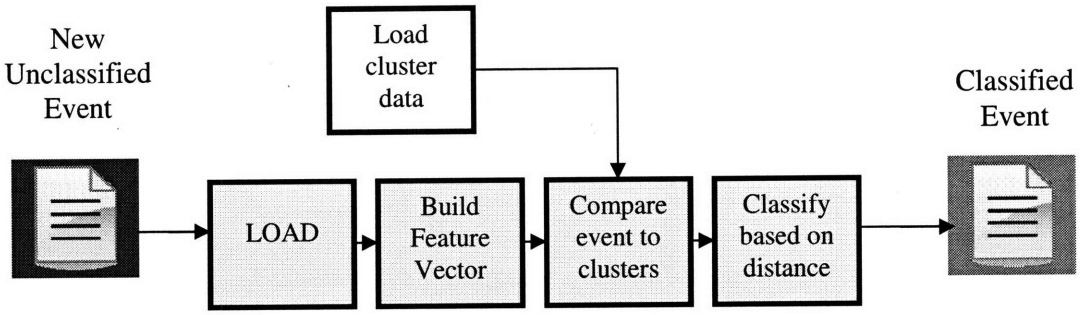


Figure 5-6 - *Ginzu*2 (during Classification phase)

When the user returns to *GinzUI*, she is informed that the Classifier is now OK, which indicates that automatic classification can be performed. This is accomplished by immediately calculating a feature vector for every incoming *event file*. A Euclidean distance is then calculated between the incoming feature vector and each cluster centroid. The event is classified with the label of the centroid with the minimum distance (Figure 5-6).

```
% This next line loads a 'feature vector' for distance
% comparison to the K-means clusters.

fv=[deltaP_event deltaQ_event deltaP_window deltaQ_window composite_gains];

for row_index=1:8
    distances(row_index)=sqrt(sum((handles.C(row_index,:)-fv).^2));
end
% Now we calculate the distance to the clusters.
[minval,minindex]=min(distances);

% Now we label the event based on the closest cluster.
% set(handles.class1,'String',labels(minindex));
event_class=labels(minindex);
```

Since the classifier is initialized and ready for automated classification, subsequent Escanaba 2008 *event file*s are now directed to *Ginzu*2, which classifies them accordingly (Figure 5-7).

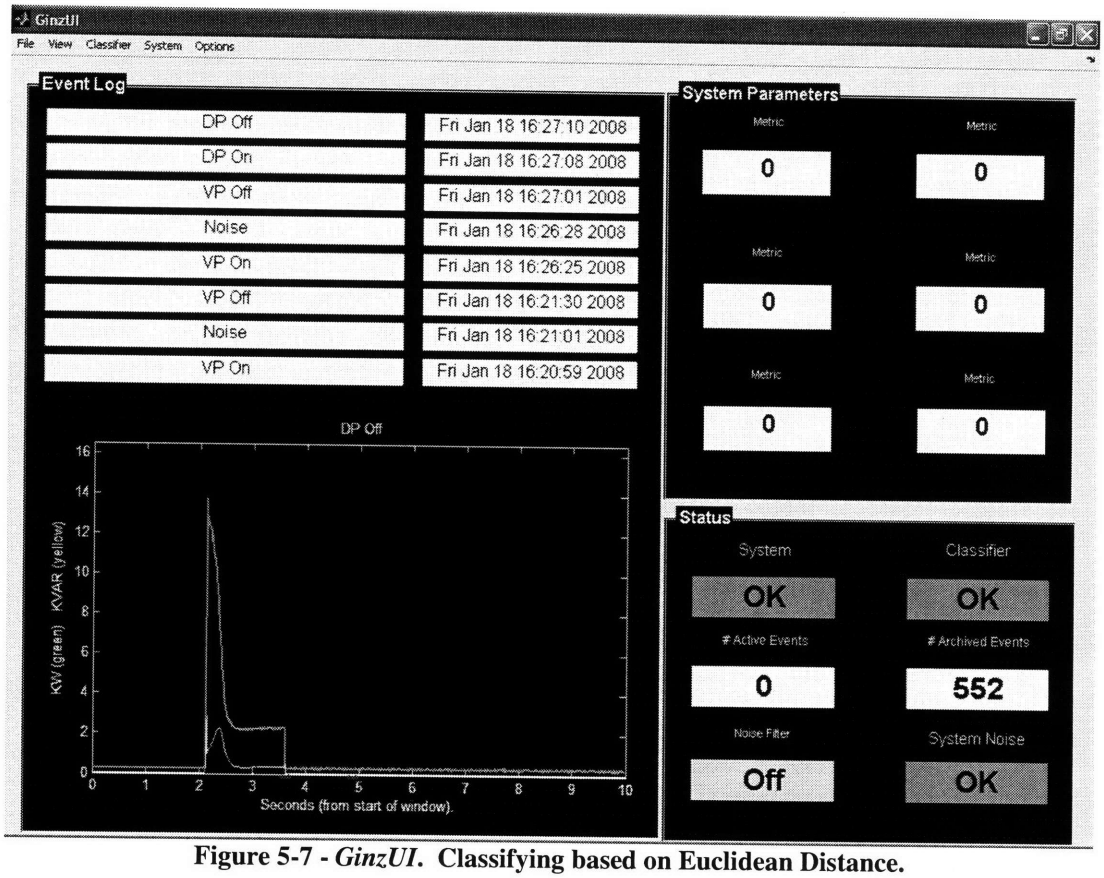


Figure 5-7 - *GinzUI*. Classifying based on Euclidean Distance.

The beauty of this process is its minimal dependency on expert knowledge. While an operator must be knowledgeable enough to label clusters, this is information that can usually be learned by starting equipment and observing patterns. The evolution of *Ginzu*2 would require a detailed analysis of failure modes and significant improvement in robustness. The feature vector could also be expanded to include spectral content or other distinguishing features. K-means clustering provides a very simple method of unsupervised learning. Many machine learning methods could be utilized to improve classifier accuracy.

5.2.2 DataLoadUI

User-friendly and simple interfaces are a key aspect of shipboard diagnostics. The average shipboard mechanic doesn't normally operate at a Linux command line and it is not reasonable to ask crew members to spend significant time learning a complicated interface. This is also true of the steep learning curve in NILM research. It is not unusual for students to require three to four months to be able to competently manipulate

NILM data. While the software discussed in this thesis has closed the gap somewhat, significant work remains.

The DataLoadUI (Figure 5-8) provides a windows-based environment that allows a novice user to conduct data processing without understanding the Linux command line. DataLoadUI provides the following functions:

- Initiates data acquisition and selects the source of the data to be processed. Data may be read from the Ethernet port (using ljstream) or piped from an existing data file.
- Provides the option to prep the incoming data. This is required as some data is archived as voltage and current. Ethernet data is automatically processed by prep.
- Allows for manual setting of *Ginzu* Mode, Change-of-Mean threshold, Window Period, and Scaling.
- Ensures that the power data is piped to grep before *ginzu* to ensure comments are removed.

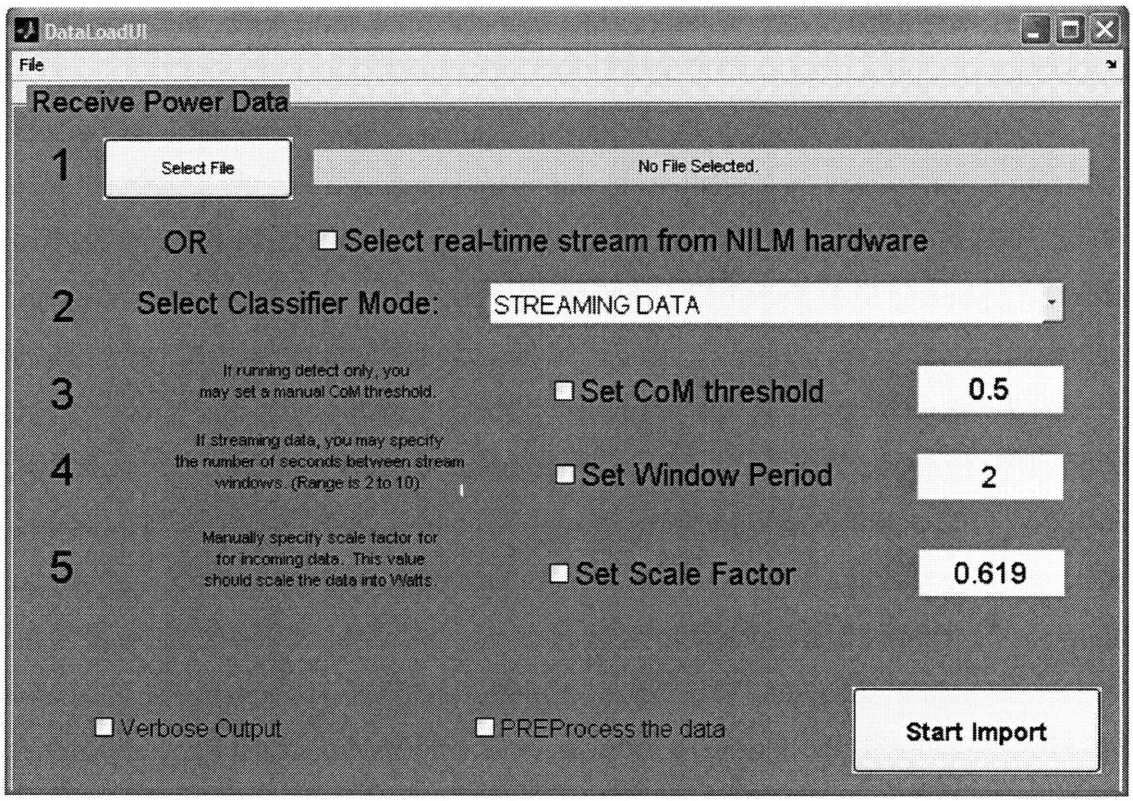


Figure 5-8 - DataLoadUI

The DataLoadUI uses the Matlab 'system' command to build command-line Linux statements that are a function of UI input boxes. This program is left as future work as it requires significant testing and debugging. Additionally, it must be brought up to date with the existing *ginzu* software.

5.3 Conclusion

The deployment and validation on USCGC Escanaba shows that NILM is a maturing technology capable of providing a non-invasive diagnostic tool to aid the condition-based maintenance strategies employed by the Navy. Specifically, this research shows the gap between data acquisition and realtime user information can be bridged by a capable software application.

Chapter 2 introduces the shipboard waste disposal system as a candidate for the application of a realtime automated event classification strategy. A review of this system's operation is given as well as a review of the most effective transient classification methods. This chapter also provides a generic classification framework that is applicable to any shipboard system. The *ginzu* software application implements this framework and applies it to the CHT system and water purification system (RO).

Chapter 3 shows how the decision-tree classifier *ginzu* can be used for realtime identification of transients in a small-scale shipboard system such as the distillation plant (RO) or CHT. Additionally, this applied research shows that the classified event data can quickly be processed by a diagnostic graphical user interface to inform crewmembers that an abnormal condition is present. Chapter 3 concludes with an optimization strategy of the 'tuning variables' used in the classifier. These variables define system-specific thresholds that are used during classification. As these variables limit the portability and robustness of the classifier, every effort is made to eliminate or reduce them. The process used to develop the *ginzu* software package is provided. Significant pre-deployment testing of the entire data processing sequence was aided by the existence of three historical 'red team' data sets. By verifying the detect-classify-diagnose sequence on these existing datasets, the Escanaba software was able to run with high classification accuracy (over 95%) and provide accurate diagnostic information.

Chapter 4 describes the method that was used to implement this detect-classify-diagnose process on the USCG Cutter Escanaba. This framework ensures that the process can automatically recover from errors and that the acquired power signals are archived. A review of classifier performance identified various categorical errors. This review motivated improvements that should further improve accuracy.

During the Escanaba patrol, the *ginzu* diagnostics package was able to provide realtime diagnosis of a tank level indication probe failure as well as two situations where the CHT vacuum pumps were running excessively due to an inability to draw required vacuum levels. This represents the first instance that a shipboard NILM has been able to identify a fault condition in realtime and immediately provide that information to an operator.

Finally, Chapter 5 discusses a possible next step for the evolution of the *ginzu* framework. A machine learning classifier is provided that implements a simple k-means clustering algorithm to allow a user to build accurate classifiers without the need of an 'expert'-coded decision tree classifier. This application is defined as future work as effort will be required to implement and test the effectiveness of the approach.

List of References

1. P. Branch. *Development of Real Time Non-Intrusive Load Monitor for Shipboard Fluid Systems*, Massachusetts Institute of Technology, NE/S.M. EM, June 2008
2. Cox, R. 2006. *Minimally intrusive strategies for fault detection and energy monitoring.* Ph.D. Diss., Massachusetts Institute of Technology, Cambridge.
3. Cox, R,. M. Piber, G. Mitchell, P. Bennett, J. Paris, W. Wichakool, S. Leeb, 2007. *Improving Shipboard Maintenance Practices Using Non-Intrusive Load Monitoring*. In *Proc. of ASNE Intelligent Ships Symposium VII*, Philadelphia, PA, May 2007
4. T.W. DeNucci. *Diagnostic Indicators for Shipboard Systems using Non-Intrusive Load Monitoring*. Massachusetts Institute of Technology S.M. NAME/S.M. ME Thesis, June 2005.
5. A. Fuller, *Spectral Approaches to Non-Intrusive Load Monitoring*, Massachusetts Institute of Technology NE/S.M EM thesis, June 2008
6. G. Hart. "Nonintrusive Appliance Load Monitoring". *Proceeding of the IEEE,* V 80, #12, 1992
7. Instruction Manual for the Envirovac Vacuum Sewage Collection System, United States Coast Guard 270' B Class WMEC. Technical Publication No. 2839
8. Lee, K. 2003. *Electric load information system based on non-intrusive power monitoring*. Ph.D. diss., Massachusetts Institute of Technology, Cambridge.
9. S.B. Leeb. A *Conjoint Pattern Recognition Approach to Non-Intrusive Monitoring* Ph.D. Dissertation, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1993.
10. Leeb, S., S. Shaw, and J. Kirtley. 1995. Transient event detection in spectral envelope estimates for nonintrusive load monitoring. *IEEE Trans. on Power Delivery* 10: 1200–1210.

11. G. Mitchell 2007, *Shipboard Fluid System Diagnostics using Non-Intrusive Load Monitoring.* Massachusetts Institute of Technology NE/S.M thesis, June 2007

12. J.P.Mosman, Evaluation *of Non-Intrusive Load Monitoring for Shipboard Cycling System Diagnostics*, Massachusetts Institute of Technology NE/S.M thesis, June 2006

13. *Numerical Recipes: The Art of Scientific Computing, Second Edition* (1992) Cambridge University Press ISBN-10: 0521880688.

14. J. Paris. *A Framework for Non-Intrusive Load Monitoring and Diagnostics.* Massachusetts Institute of Technology S.M. EECS Thesis, February 2006.

15. M. Piber 2007, *Improving Shipboard Maintenance Practices Using Non-Intrusive Load Monitoring,* Massachusetts Institute of Technology, SM ME Thesis, June 200

16. S.R. Shaw. "System Identification and Modeling for Nonintrusive Load Diagnostics," Ph.D. Dissertation, Massachusetts Institute of Technology, Cambridge, MA, 2000.

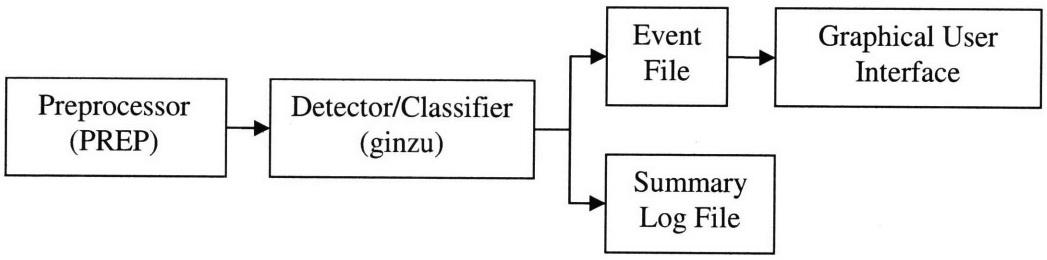
17. G. Strang. *Introduction to Applied Mathematics.* Wellesley-Cambridge Press. 1986

18. Village Marine Tec. *Model RC7000 Plus Reverse Osmosis Desalination Plant Operations and Maintenance Manual.* Gardena, CA : Village Marine Tec, 2004.

Ginzu User's Manual

Overview

The function of *ginzu* is to extract transients from 'prep' power data and output them in small (~20kB) *event files*. These files are generated in the same directory where *ginzu* is run. *Ginzu* also generates a running log as it operates. This log is directed to standard output (the monitor) and is normal re-directed to a text file as shown below in basic *ginzu* usage.



The *ginzu* application is a command-line program written in C and compiled with the gcc compiler. A complete discussion of *ginzu* operation and usage is provided in *Automated Classification of Power Signals* (Proper 2008). Throughout this guide, important issues are noted. A user should be aware of these issues when using *ginzu*.

User Note #1: *Ginzu* must receive prep data – not raw data.

Basic *ginzu* Setup

The basic procedure for implementing *ginzu* on a Linux computer is provided.

1. Create a directory for *ginzu*. This will be the directory that the *ginzu* executable runs in and will also be the directory that *event files* are created in. In this example, I create the directory /nilm in my home directory.

```
proper@bucket:~$
proper@bucket:~$ mkdir nilm
proper@bucket:~$ cd nilm
proper@bucket:~/nilm$ cp /home/proper/ginzu/ginzu.c ginzu.c
proper@bucket:~/nilm$ ls
ginzu.c
proper@bucket:~/nilm$ gcc -o ginzu ginzu.c -lm
proper@bucket:~/nilm$ ls
ginzu  ginzu.c
proper@bucket:~/nilm$
```

2. Move *ginzu* source code (*ginzu*.c) from the NILM Software CD (or wherever you have the source code) to your newly created directory.
3. Compile *ginzu* source code into an executable (i.e. gcc –o *ginzu ginzu*.c -lm).
4. You should now have a working copy of *ginzu*.

User Note #2: When compiling with gcc, *ginzu* MUST be compiled using the '-lm' extension. This tells gcc to include the math libraries while compiling.

You can test that *ginzu* is available by typing '*./ginzu* –help' at the command line. This will give a basic summary of command line arguments. Now you are ready to create *event files*.

There are two ways to use *ginzu*. The first (and simplest) is using *ginzu* to process existing (archived) power data. The second is to pipe realtime power data directly into *ginzu* as the data is received from the NILM hardware (via Ethernet or PCI 1710 connection).

Command-Line Arguments

Ginzu accepts a number of command line parameters that allow the user to modify the default modes of operation. A complete list of command-line arguments is provided in *Automated Classification of Power Signals* (Proper 2008). A shorter list is provided by typing '*./ginzu* –help' .

Basic Usage

Assume you have an uncommented prep data file called snapshot.txt in your *ginzu* directory. A simple example of a *ginzu* implementation is:

$ dd if=snapshot.txt | *./ginzu* m 2

This pipes (or directs) the contents of snapshot.txt to *ginzu* that runs in 'detect only' mode and creates ten second *event files* where any power transients exist.

Advanced Usage

A more typical (and complicated) line would be:

$ dd if=snapshot.txt | *./ginzu* r 0 m 1 v 1 p 0.2 q 0.2 l 0 f filename.txt > summary.out

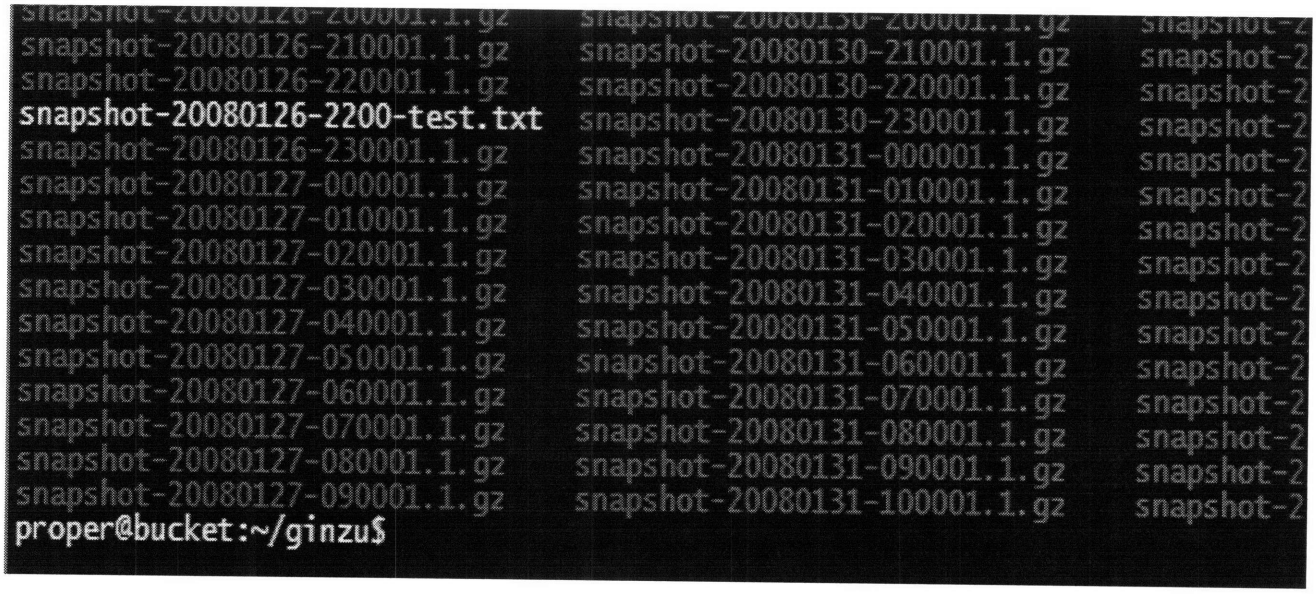
This command pipes the contents of snapshot.txt to *ginzu* that does the following:

- [m 1] – Tells *ginzu* that the data came from the Reverse Osmosis system and that you want *ginzu* to classify the events using the built in RO classifier.
- [v 1] – Tells *ginzu* to create a verbose log file.

- [p 0.2 q 0.2] – Tells *ginzu* to scale the incoming prep data by 0.2 so that it is converted to KW and KVAR.
- [l 0 f filename.txt] – Tells *ginzu* that it is receiving piped data. Since *ginzu* has no idea what the name of the incoming data file is, you can tell it the name. *Ginzu* then uses this filename when it names to the output *event files*.
- [r 0] – Tells *ginzu* to parse the filename to determine the date-hour that the file was taken. It then builds the event timestamps using the date-hour as a reference.
- [>> summary.out] – Linux argument telling the OS to direct the output of *ginzu* to a file called summary.out (this is a text file).

Using *Ginzu* on Archived Data

When a NILM is installed on a system, the data collection software records the NILM data to one-hour files and stores these files in the /data directory. These files are stored in the data directory until they can be retrieved for analysis. For example, on the USCG Cutters, this data is retrieved when the ship returns to port.



The data directory contents are similar to that shown above. Note that you can not tell if this data is stored in raw format (voltage and current @ 8 kHz) or prep format (power @ 120 Hz). The files will also typically be zipped.

User Note #3: *Ginzu* MUST receive uncommented 8-column PREP data.

Regardless of the method that you use, the data must be presented to *ginzu* in strict uncommented 8-column prep format. There are a number of ways to do this but the following method is provided as an example.

```
proper@bucket:~/ginzu$
proper@bucket:~/ginzu$ gunzip -c /home/pbranch/RO/Escanaba/gz/Feb2008/snapshot-20080204-130001.1.gz
# nilm-laptop reopened 20080204-120001-0500 r5783 "/usr/local/bin/prep-1 < /tmp/run-prep/input-1 |"
4.26726e+04  -4.78282e+04  7.93482e+01  -3.69842e+02  -3.28262e+02  -3.87411e+02  -6.46288e+02  2.42861e+02
4.24504e+04  -4.77737e+04  2.62218e+02  -3.25435e+02  -3.98315e+02  -3.62157e+02  -6.00548e+02  2.04058e+02
4.27288e+04  -4.79664e+04  -9.85964e+00  -2.38140e+02  -3.22049e+02  -4.53656e+02  -6.08696e+02  2.99326e+02
4.29097e+04  -4.77854e+04  3.37821e+01  -3.67767e+02  -3.09680e+02  -4.24592e+02  -6.19585e+02  3.09918e+02
4.28135e+04  -4.72263e+04  7.78473e+01  -2.35271e+02  -2.98227e+02  -4.51334e+02  -6.26242e+02  3.44340e+02
4.28133e+04  -4.69882e+04  -4.95127e+01  -2.69868e+02  -2.46172e+02  -4.55701e+02  -6.55674e+02  3.74368e+02
```

The gunzip command with the '-c' extension unzips a file and directs its contents to stdout. You can see here that the file contains 8-column prep data. If the data was raw, it would be two-column and be integer values (normally sinusoidal around 2048).

Note that the file contains comments; these comments are NOT allowed by *ginzu* so they must be extracted. This is done with grep. For example:

$ gunzip –c {filename} | grep –v '#'

This will direct the contents of the file to stdout while extracting all comments. So, if the data was stored in raw format, you could type:

$ gunzip –c {filename} | prep | grep –v '#'

This would do the same thing but it would prep the data first. Since you want the data to be directed to *ginzu*, the following is typical.

$ gunzip –c {filename} | grep –v '#' | *./ginzu* [insert arguments here]

Example

```
proper@bucket:~/nilm$ ls
ginzu  ginzu.c  snapshot-20080125-120001.1.gz
proper@bucket:~/nilm$
```

In this example, I moved a zipped RO archive file to the /nilm directory. You don't actually have to move the file. You could just direct the output – but I moved it here to make the example easier to understand. I know that this file is zipped prep data that contains comments. I'm going to unzip and uncomment the file and move it to a temporary text file. Again, you don't have to do this in practice.

```
proper@bucket:~/nilm$ gunzip -c snapshot-20080125-120001.1.gz | grep -v '#' >> power_data.prep
proper@bucket:~/nilm$ ls
ginzu  ginzu.c  power_data.prep  snapshot-20080125-120001.1.gz
proper@bucket:~/nilm$
```

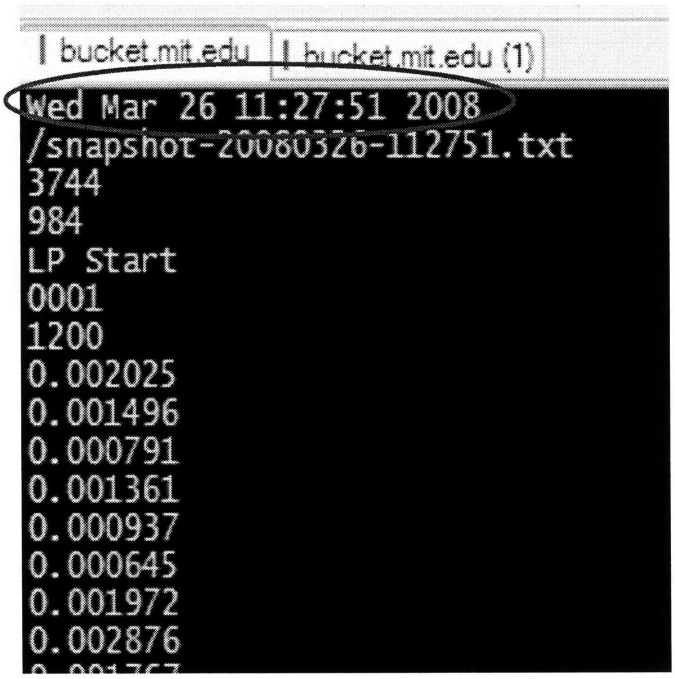
So there is now a file called power_data.prep. Now I pipe this file to *ginzu* (using the dd command) and specify that I want RO classification. I also set the scaling factors.

User Note #4: PREP data is NOT scaled. So if you don't set the scaling factors in *ginzu*, *ginzu* doesn't get watts and vars. SO, YOU NEED TO SET THE SCALE FACTORS.

User Note #5: *Ginzu* exits after reading 432,000 lines of data. This corresponds to one-hour of data at 120 samples per second. Since the prep data rate is not exactly 120 samples per second, the hour data files do not contain exactly 432,000 lines of data.

```
proper@bucket:~/nilm$ ls
ginzu  ginzu.c  power_data.prep  snapshot-20080125-120001.1.gz
proper@bucket:~/nilm$ dd if=power_data.prep | ./ginzu m 1 p 0.2 q 0.2 >> summary.out
19348+1 records in
19348+1 records out
9906595 bytes (9.9 MB) copied, 0.922465 seconds, 10.7 MB/s
proper@bucket:~/nilm$ ls
ginzu            snapshot-20080125-120001.1.gz          snapshot-20080326-112751-0002-0001.evt  summary.out
ginzu.c          snapshot-20080326-112751-0000-0001.evt  snapshot-20080326-112751-0003-0007.evt
power_data.prep  snapshot-20080326-112751-0001-0005.evt  snapshot-20080326-112751-0004-0005.evt
proper@bucket:~/nilm$
```

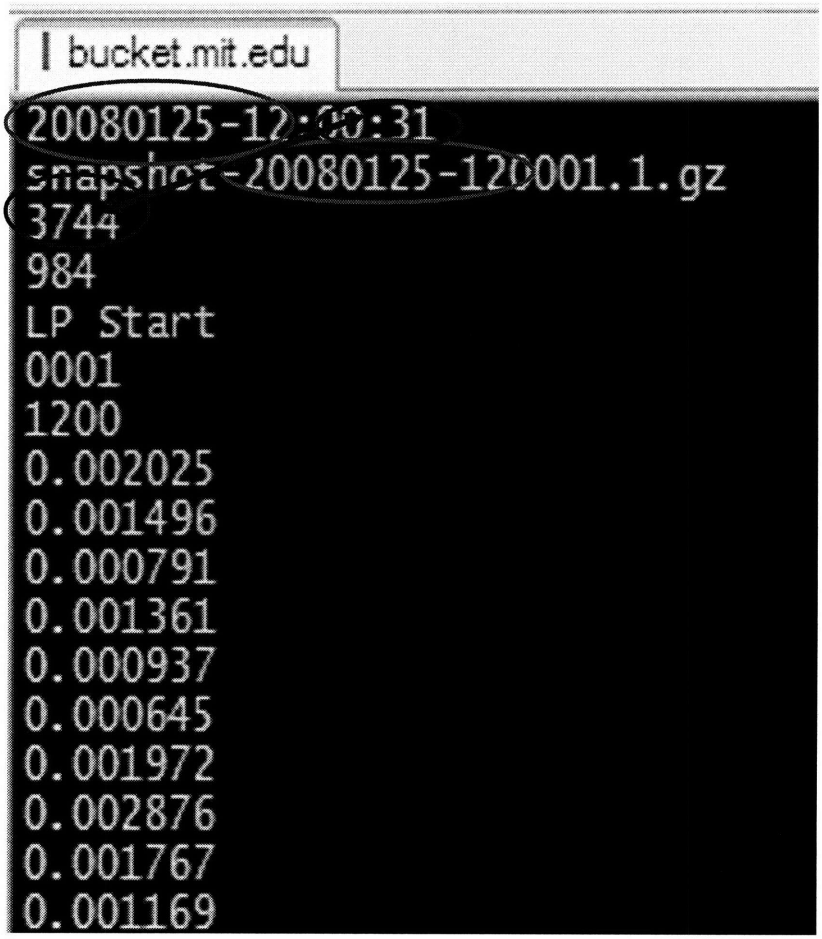
So *ginzu* processes the incoming data and generates *event files*. *Ginzu* has no idea what time and date that the actual data was collected. All it received was a large stream of 8-column text. So you see the *event files* are named using the current date-time as reference and time-stamped using current time. I ran this example on March 26th at 11:27.



In practice, we actually want to know the time that the event occurred so a better way to do this is to specify the name of the file and to tell *ginzu* not to create realtime timestamps, which forces it to build the timestamp using the index of the event in the stream. Note the 'r 0' command line argument.

```
proper@bucket:~/nilm$ gunzip -c snapshot-20080125-120001.1.gz | grep -v '#' | ./ginzu m 1 p 0.2 q 0.2 r 0 l 0 f snapshot-20080125-12
0001.1.gz >> summary.out
proper@bucket:~/nilm$ ls
ginzu             snapshot-20080125-120001-0000-0001.evt  snapshot-20080125-120001-0003-0007.evt  summary.out
ginzu.c           snapshot-20080125-120001-0001-0005.evt  snapshot-20080125-120001-0004-0005.evt
power_data.prep   snapshot-20080125-120001-0002-0001.evt  snapshot-20080125-120001.1.gz
proper@bucket:~/nilm$
```

Now we see the *event files* are named based on the filename that we told *ginzu*. Also, we see that the timestamps are built based on the date in the filename and the index of the event divided by 120.



User Note #5: If you specify a filename, it MUST be formatted in the 'standard' format. That means filename-YEARMODA-HOURXX.*. *Ginzu* uses this data to reconstruct the actual event time. This is the format the NILM has used since at least 2003.

User Note #6: Note that the format of the timestamp is different if using realtime timestamps or non-realtime stamps.

Scripting with *Ginzu*

The manual method shown above allows you to use *ginzu* on individual data files. In practice, it is much more useful to conduct mass processing of large data sets. For example, to *ginzu* all the one-hour files for a given period. This is accomplished with scripts.

| bucket.mit.edu | bucket.mit.edu (1) | bucket.mit.edu (2)

```
#!/bin/sh


SOURCE_PATH_NAME="/home/pbranch/RO/Escanaba/gz/Feb2008"
DEST_PATH_NAME="/home/proper/ginzu"

cd $SOURCE_PATH_NAME
FILE_LIST="`ls *.gz`"

cd $DEST_PATH_NAME

for file in ${FILE_LIST}
do

        echo $file IP

        # This runs ginzu on the contents of every GZ file in the source directory
        # Ginzu is running (NON_VERBOSE), (RO CLASSIFY), (TAKING PIPED INPUT), (DIRECTING OUTPUT TO summary.out)
        gunzip -c ${SOURCE_PATH_NAME}/${file} | grep -v '#' | ./ginzu r 0 v 1 l 0 m 1 f $file p -0.2 q 0.2 >> summary.out

        # Remove the comments here if you want a tar archive created for EVERY hour of data.
        # tar -cf ${DEST_PATH_NAME}/${file}_event.tar *.evt
        # rm *.evt
        # If these lines are left in their commented state, all EVT files will be created in the DEST_PATH_NAME directory.
        # and NOT TARRED.
done
```

This shows a simple #bash script that can be used to mass *ginzu* a large list of prep files. Basically, the user sets the source directory that contains the one-hour files (SOURCE_PATH_NAME) and the directory that contains *ginzu* (DEST_PATH_NAME). The script changes the active directory to the source directory and builds a list of all the *.gz files. The script then changes back to the *ginzu* directory and runs *ginzu* on every file on the list. The *event files* are created in the *ginzu* directory.

User Note #7: *Ginzu* does not care about the sign (positive, negative) of the incoming data. Since *ginzu* only considers magnitude of change, the absolute value of the data is taken as soon as it is imported. So the user does not need to be concerned about whether the data is signed.

It may be useful to archive (tar) the *event files* during each hour into a single tar file. This is commented out in this case, but is simple to implement.

Using *Ginzu* on Realtime Streaming Data

Ginzu provides a useful tool for analysis of existing archive data. However, the primary function of *ginzu* is to provide realtime analysis of power data as it is generated by a NILM. Ultimately, this is accomplished in the same method as described above. The data must be received by *ginzu* in uncommented 8-column prep format. The key to realtime data acquisition is the ljstream program (J. Paris 2007). The NILM hardware generates raw voltage/current data and sends this data to the NILM

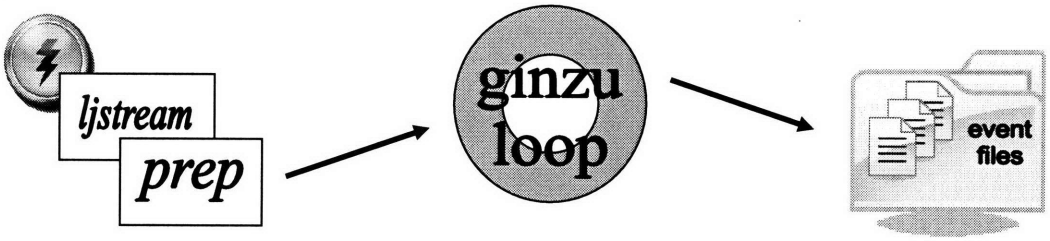
PC via Ethernet. The ljstream program receives this data and sends it to stdout. A typical ljstream command would be:

$ ljstream –n 2 –r 8000

This would generate six channels of data – 3 columns of current data followed by 3 columns of voltage. To actually use the data, we need to extract the correct 2 columns, scale them appropriately, send them to prep in the correct order, and remove the comments. This command line is an example.

$ ljstream –n 6 –r 8000 | awk '{print $4/16 " " $2/16}' | prep | grep –v '#' | *./ginzu* m 2 p 0.34 q 0.34

Since the data is received by *ginzu* as soon as it is created, the *event file*s are generated as they occur (with a small processing delay). Also, remember that *ginzu* exits after reading 432,000 lines of power data. If more than one hour of data is being received, *ginzu* must be set to restart every time it exits (i.e. loop).



Event file Handling

*Event file*s are generated in the directory that *ginzu* is in. If no action is taken the files will just accumulate in this directory. In a more useful application, a GUI (or handler program) will be running while *ginzu* is creating *event file*s. This GUI will check the directory at a regular interval and do some additional processing with the information in the *event file*. This is discussed in detail in *Automated Classification of Power Signals* (Proper 2008) and related works by P. Branch, 2008 and R. Jones, 2008.

Appendix [B] - CHT Tuning Parameters

Value	Description	Stable?	Can be eliminated?	Can be combined?	Effect?	Function of:
153	Lockout Window	No	No	No	This is the number of indexes that is 'locked out' to prevent multiple detections of the same event.	**Lockout Window**
0.4 KW	Detection Threshold (CoM Filter)	Yes	No	Yes	If the Change of Mean filter generates a response greater than this value, an event will be generated.	Detection Threshold
0.25 KW	Noise Threshold	Yes	No	Yes	If the change in power across the event is less than this value, the event will be labelled as noise (or cycling load).	Detection Threshold
0.45	Threshold for VP Score	Yes	No	No	This is the minimum correlation score to classify and event as a VP On.	**Threshold for VP Score**
0.45	Threshold for DP Score	Yes	No	No	This is the minimum correlation score to classify and event as a DP On.	**Threshold for DP Score**
(60, 7, 3)	Derivate detector (lockout section, skip after detect, magnitude of change)	No	No	Yes	This value sets the number of indexes after the event to skip before searching for an additional event. The second value sets the magnitude of change required for an additional event. The last value is the number of indexes that are skipped if an additional event is detected.	**Derivate detector (lockout section, skip after detect, magnitude of change)**
0.1 to 0.8	Change in power that indicates a VP is clogged.	Yes	No	Yes	If the system power is between these bounds after a VP start, the VP is labelled as clogged.	Steady VP Running Power

0.3 KW	Zero Power Level	Yes	No	Yes	If post-event power is less than this value (on a downpower), the event is an ALL OFF.	**Zero Power Level**
600-650 indexes	Time after VP on to recognize an event as a clogged pump	No	No	Possible	This is the number of indexes after a VP On that is checked to determine post-start power levels.	**Time after VP on to recognize an event as a clogged pump**
Note 1	Ranges for double event classfication (P,Q)	Yes	Yes	Yes	These values determine the change in P and Q required to classify a '2 VP On' or a '1 VP On and 1 DP On' event.	Steady VP Running Power, Steady DP Running Power
Variable	Time windows for 'change in power' calculations	No	No	No	These values determine the indexes that are used to calculate average power before and after the index that an event was found at.	**Time windows for 'change in power' calculations**
0.1 KW, 0.1 KVAR	Maximum Change in power to cause a cycling load classfication	Yes	Yes	Yes	When a strong shape match is found, the power after the event is checked. If the power is less than this value, the pump turned of and is labelled as 'cycling'.	Steady VP Running Power
Note 2	Magnitude of downpower to classifify VP or DP Off	Yes	No	Yes	Sets required change in P and Q to classify a VP Off or DP Off event (if power does not go to zero).	Steady VP Running Power, Steady DP Running Power
0.6 (Note 3)	alpha threshold	Yes	No	No	See discussion in Chapter 2.	**alpha threshold**
2 (Note 3)	gamma threshold	Yes	No	No	See discussion in Chapter 2.	**gamma threshold**

0.25 KW	Maximum Steady State Power that indicates a ALL OFF condition.	Yes	No	Yes	If power is less than this value, state information must be ALL OFF.	Zero Power Level
0.5 KW	Minimum Steady State Power that indicates a pump is running.	Yes	No	Yes	If power is greater than this value, state information must NOT be ALL OFF.	Zero Power Level
0.1	Maximum Sigma for Steady State	Yes	No	Yes	The variation in power during the current window must have a standard deviation less than this value, or the window is not considered to be steady state.	Detection Threshold

Note 1: See arbitrate_event_overlap() function.

Note 2: See section 3.5.4 regarding 'Classifying Off events'

Note 3: See section 2.2.3 regarding 'Classification based on Shape Correlation'

Appendix [C] - RO Tuning Parameters

Value	Description	Stable?	Can be eliminated?	Can be combined?	Effect?	Function of:
153	Lockout Window	No	No	No	This is the number of indexes that is 'locked out' to prevent multiple detections of the same event.	**Lockout Window**
0.5 KVAR	Detection Threshold (CoM Filter)	Yes	No	Yes	If the Change of Mean filter generates a response greater than this value, an event will be generated.	Detection Threshold
0.5 KVAR	Noise Threshold	Yes	No	Yes	If the change in power across the event is less than this value, the event will be labelled as noise (or cycling load).	Detection Threshold
0.5	Threshold for LP Pump Score	Yes	No	No	This is the minimum correlation score to classify and event as a LP On Event.	**Threshold for LP Pump Score**
3	Number of recent seconds that are checked to determine steady state power.	No	No	Yes	The mean and standard deviation of the last 3 seconds of power is determine to check steady state power levels.	**Window Size**
0	Threshold for HP Pump Score	Yes	No	No	This is the minimum correlation score to classify and event as a HP On Event.	**Threshold for HP Pump Score**
0.1 KW or KVAR	Zero Power Level	Yes	No	Yes	If post-event power is less than this value (on a downpower), the event is an ALL OFF.	Zero Power Level
10 KVAR	Minimum Change in Q to Classify as HP Pump	Yes	No	Yes	If Q is greater than this number, the change is significant enough for the classifier to favor a HP event.	Steady HP Pump Running Power
Variable	Time windows for 'change in power' calculations	No	No	No	These values determine the indexes that are used to calculate average power before and after the index that an event was found at.	**Time windows for 'change in power' calculations**

0.6 (Note 3)	alpha threshold	Yes	No	No	See discussion in Chapter 2.	**alpha threshold**
2 (Note 3)	gamma threshold	Yes	No	No	See discussion in Chapter 2.	**gamma threshold**
0.1 KVAR	Maximum Steady State Power during state verification that indicates a ALL OFF condition.	Yes	No	Yes	If power is less than this value, state information must be ALL OFF.	Zero Power Level
0.5 KVAR	Minimum Steady State Power that indicates a pump is running.	Yes	No	Yes	If power is greater than this value, state information must NOT be ALL OFF.	Steady LP Pump Running Power
0.1	Maximum Sigma for Steady State	Yes	No	Yes	The variation in power during the current window must have a standard deviation less than this value, or the window is not considered to be steady state.	Detection Threshold
0.1	Maximum standard deviation of power over last 3 seconds to call event as noise.	Yes	No	Yes	If sigma is very high, the power has a large degree of variation. This normally indicates large spikes or cycling loads.	Detection Threshold
90%	P/Q ratio checked to determine if bypass valve has adjusted.	Yes	No	No	After a bypass is shut, real power is low until the bypass is shut. When P rises such that this ratio is exceeded, the bypass is said to be shut.	**P/Q ratio checked to determine if bypass valve has adjusted.**
1.5 KW	Threshold (during steady state correction) to distinguish 1 LP pump from more than 1 LP pump.	Yes	No	Yes	If steady state is reached and power is over this value but less than the following value, system will correct to 1 LP on, 1 HP on [1 1]	Steady LP Pump Running Power
10 KW	Threshold (during steady state correction) to distinguish less than 2 HP pumps from more than 2 HP pumps.	Yes	No	Yes	If steady state is reached and power is over this value, the classifier will correct to ALL ON [1 2].	Steady HP Pump Running Power

Appendix [D] – *Ginzu* Source Code (as deployed on USCGC Escanaba)

```
////////////////////////////////////////////////////////////////////////////////
// GINZU.C                                                                    //
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
//  Examples of implementation
//  Ex (1) - dd if=snapshot.txt | ./ginzu l 0  (piped input, default CHT Classify)
//     Ex (2) - ./ginzu l 1 f C:/NILM/mysnapshot.txt (designated mysnapshot.txt as loadfile,
//                  .... will CHT Classify as default)
//     Ex (3) - dd if=snapshot.txt | ./ginzu l 0 m 3 v 1 d 431231 e 5
//              (piped input, streaming event files, read in 431231 lines, create eventfiles
//                  ... every 5 seconds with default windowsize of 1200, in verbose mode)
//  Ex (4) - dd if=snapshot.txt | ./ginzu l 0 m 2 t 1.0 v 1 >> results.out
//              (piped input, detect only using 1.0KW threshold, no classifier used, verbose mode,
//                  ... redirect output of ginzu to results.out file which will be created
//                  ...    in same directory as ginzu (if results.out already exists, text will be
//                  ... appended to it.)
////////////////////////////////////////////////////////////////////////////////


#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <time.h>
#ifdef _WIN32
#include "stdafx.h"  // This header is req'd by Visual C, Delete in gcc
#else
#include <signal.h>
#endif


#define BUFFER  1200       // Size of search window
// Window_time = BUFFER/120 seconds
// Default BUFFER = 1200 indexes = 10 seconds
// Minimum BUFFER is 700 + 120 + ADVANCE
// This is due to the fact that the CHT classifier verifies the event is not within 700 indexes of a
// boundary as the CHT classifier may use data at (t+ 6) seconds.  If the BUFFER is < 820 + ADVANCE,
// events will be continuously skipped.
#define ADVANCE 120        // ADVANCE = the # of indexes skipped forward during each window
// Default ADVANCE = 120 indexes = 1 second
#define float double       // Forces all floats to double precision
#define MAX_EVENTS 500  // This is the maximum number of events that can be detected before ginzu
// must be restart.
#define FIR_SIZE 33        // Number of elements in low pass filter (B)
#define A_SIZE BUFFER+2*FIR_SIZE // A_SIZE established that size of the P and Q data arrays.
// An additional buffer is placed at the front and back of the
// P/Q arrays to prevent edge effects during the filtering.
```

```
// Required by Numerical Recipies routines (matrix, vector implementation)
#define NR_END 1
#define FREE_ARG char*
#define TINY 1.0e-20

//  Boolean definitions
#define TRUE  1
#define FALSE 0

////////////////////////
//      GLOBALS       //
////////////////////////

// This is set by 'l' parameter.
int LOADFILE = 0;
// If LOADFILE == 1, Ginzu will attempt to load the designated filename (which must
// also be set via command line argument.)  If the user sets L=1 and does not designate
// a filename, Ginzu will attempt to load the filename below (which will likely not
// exist).  [ i.e.  ginzu l 1 f myprepfile.txt ]

// If LOADFILE == 0, Ginzu will recieve prep data from stdin (which is typically piped in)
// [ i.e. dd if=myprepfile.txt | ginzu l 0 ]

// This is the default filename for .evt file AND the file that will be loaded if 'l' = 1.
// This should be set to "snapshot.txt" as default.
char filename[256];

// This is set by 'v' parameter. (Default 1)
int VERBOSE= 1      ;      // VERBOSE = 1 provides a verbose output of ginzu execution.  (Default = 0)

// This is set by 'm' parameter.  (Default 3)
int MODE = 0 ;   // MODE is set at command-line to indicate what system is being classified.
// if MODE == 0, CHT
// if MODE == 1, RO
// if MODE == 2, No classifier is called. Only P/Q stubs are created.
// if MODE == 3, No classifier is called. Data just streams out.

// This is set by 'e' parameter.  (Default 2)
int STREAM_INC = 2; // The sets the default delay between stream window generation
// With STREAM_INC=2 and default 10s windows, there is 8 seconds of overlap.

// This is set by 't' parameter.
double GETEVENTS_THRESHOLD = 0.40;
// This is the CoM threshold for event detection.

// These are set by 'p' and 'q' parameters.
double P_SCALE = (-.61946);             // P_SCALE scales incoming P to KW.
double Q_SCALE = (0.61946);             // Q_SCALE scales incoming Q to KVAR.
```

```
// This allows the user to specify the exact number lines that will be read
// Set by 'd' parameter
int NUM_PREP_LINES = 432000;

// Set by 'r' parameter
// If REAL_TIMESTAMP = 1, the event files will be timestamped with the actual event time
// If REAL_TIMESTAMP = 0, Ginzu will attempt to stamp the file by parsing the filename of the
// incoming piped file (or the designated file by the 'f' argument).
int REAL_TIMESTAMP = 1;


int window=0;       // Indicates the index of the current window.  (0 is first window)
int line_number=0;
double P[A_SIZE];   // Real Power data from input file.
double Q[A_SIZE];   // Reactive Power data from input file.

////////////////////////////////////////////////////////////////////////////////////////
// System Specific GLOBAL State Variables for CHT Classifier
int dis=0;                  // The # of discharge pumps currently running (CHT)
int vac=0;                  // The # of vacuum pumps currently running  (CHT)
int clogged_vac=0;  // The # of clogged vacuum pumps currently running (CHT)

// System Specific GLOBAL State Variables for RO Classifier
int LP=0;                   // The # of low pressure pumps currently running (RO)
int HP=0;                   // The # of HP pumps currently running (RO)
int stable=1;       // Variable.  If stable == 1, power has been essentially constant
// over the last n seconds of data.  This variable is tracked to
// indicate when Ginzu should do Steady State Checks.
int LP_STEADY=1;    // Variable indicating if the system has reached STST after an LP start.
// This is set to FALSE after an LP start which tells the RO_IC to check
// for a STST condition and create an event file if reached.
int MISMATCH=0;     // Variable to indicate if a P/Q mismatch has occured.
// This follows a HP Start when bypass valve needs to be positioned.
int ST_WIN_Index=0;                     // This index tracks the streaming window index.
// It is also used during RO Classification to identify the index of the RO Steady state window.
// is periodically given for diagnostics.  ST_WIN_Index tracks
// the number of steady state files that have been generated.
// if ST_WIN_Index > 9999, it will be reset to 0
////////////////////////////////////////////////////////////////////////////////////////


// EVENT TABLE DATA
int event_index[MAX_EVENTS];            // This contains the index of the event referenced to the input datastream.
            // For example, a 1-hour input datafile contains ~430000 lines of prep data
            // So the event_index will be between 0 and 430000.  If the data is streaming,
            // This number will continue to grow and must be reset at some point.
            // The use of 'int' means the event_index value is limited.  This may need to be
            // changed to 'unsigned long' at some point.
int event_class_status[MAX_EVENTS];     // Boolean to indicate whether an event has been classified.
int loc_index[MAX_EVENTS];              // This is the index of an event within a specific window.
char *Class[MAX_EVENTS];                // String text label to indicate type of event.
```

```
int Class_ID[MAX_EVENTS];               // Classification Code to indicate the type of event.
int EVC[MAX_EVENTS];                          // Boolean to indicate whether an event has had an .evt file created
int local_det=0;                              // Number of events found in current window.
int event_count=0;                            // current index of event within global dataset
// event_count is incremented when a new event is found
// event_count is reset to zero if it gets within 10 of MAX_EVENTS
//  ... and the global data table is output and then cleared



//////////////////
//// NILMCONST  ///
//////////////////

int          GETEVENTS_SKIP = 120+FIR_SIZE;  // This is the LOCKOUT window size.
// Do NOT set GETEVENTS_SKIP to be > than 1/4 of BUFFER

// The B array provides the convolution values for the LP filter
double B[FIR_SIZE] =  {-0.00152884475502,
       -0.00190413765156,
       -0.00251201020054,
       -0.00315172285924,
       -0.00338157483003,
       -0.00256899869677,
       0.00000000000000,
       0.00497037076557,
       0.01275585329868,
       0.02339963105131,
       0.03649018015257,
       0.05115537256927,
       0.06614310773965,
       0.07997947219925,
       0.09117927260132,
       0.09847221964249,
       0.10100361794610,
       0.09847221964249,
       0.09117927260132,
       0.07997947219925,
       0.06614310773965,
       0.05115537256927,
       0.03649018015257,
       0.02339963105131,
       0.01275585329868,
       0.00497037076557,
       0.00000000000000,
       -0.00256899869677,
       -0.00338157483003,
       -0.00315172285924,
       -0.00251201020054,
       -0.00190413765156,
       -0.00152884475502
```

```
};

// imP_P1 is the template array for the correlation
// This correlation is done during classification to
// differentiate Vacuum pump starts from discharge pump starts.
// *IMPORTANT* imP_P1 scaled to KW already.

// This is the vacuum pump template.
double imP_P1[30]={
       -.0034810531470,
       -.0028567315217,
       -.0016764414181,
       .0884024851346,
       5.1078472517000,
       10.3169307863333,
       10.5831333970000,
       10.3843073856667,
       10.4604803170000,
       10.1704078476667,
       10.2200265936667,
       10.1167006656667,
       9.6590436176667,
       9.9572826346667,
       9.3273537606667,
       9.0469345430000,
       9.1635582123333,
       8.3367752746667,
       7.8404742470000,
       7.6113772903333,
       6.8113983868667,
       5.9300812881000,
       5.2264831336000,
       4.4387612786667,
       3.5793142152000,
       2.8205490179667,
       2.2178433461000,
       1.7456919665333,
       1.3948040117000,
       1.1701186426667
};


// This is the discharge pump template.
double imP_P2[30]={
       0.0011,
       0.0012,
       0.0052,
       0.0053,
       0.0000,
       1.7817,
       6.6638,
```

```
        9.4152,
        8.7898,
        8.8359,
        8.7713,
        8.7091,
        8.7912,
        8.6292,
        8.7735,
        8.5845,
        8.7284,
        8.6070,
        8.5947,
        8.6914,
        8.4573,
        8.6716,
        8.4488,
        8.4973,
        8.5434,
        8.3263,
        8.4322,
        8.3316,
        8.2208,
        8.2588
};

double imP_LP[30]={
0.000630,
0.001387,
0.001923,
0.000892,
0.001213,
8.770360,
13.553180,
11.172280,
10.815480,
11.589280,
11.075360,
10.480340,
11.826680,
9.601280,
10.522200,
10.503940,
8.498480,
8.304160,
7.376840,
5.843600,
4.552880,
3.273700,
2.285400,
1.782732,
1.568252,
```

```
1.463954,
1.450996,
1.492080,
1.556496,
1.614886
};

double imP_HP[30]={
0,
0,
0,
0,
0,
6.2,
10,
10,
10,
10,
10,
10,
10,
10,
10,
10,
10,
10,
10,
10,
10,
10,
10,
10,
10,
6,
1.5,
0,
0
};


int signal_ignore = 0;
int signal_received = 0;

//////////////////////
//  PROTOTYPES      //
//////////////////////
void handle_signal(int signal);

void init_event_table(void);
void display_and_exit(void);
```

```
// Functions used in CoM detector to find event locations and load them into global data
void filterslow(double *B, double *PQ, int LEN_B, int LEN_PQ, double *FILT_PQ);
void load_index(int *x);

// Classifier functions and Condition Check functions
void CHT_CLASSIFY(void);
void CHT_IC(void);
void RO_CLASSIFY(void);
void RO_IC(void);
void slice(void);
void stream(void);

// Processing functions used in CHT classifere
double  *exemplar_fit(double *Pact,  int index, int pump_type);
double  *transient_classifier(double *Pact, int ev_loc,  int pump_type);

// Double Event detection algorithms
int          search_lockout_window(int i, int ev, double *deriv);
void   arbitrate_event_overlap(int i, int ev, double *Pact, double *Qact, int CGI);

// Processing functions used in both classifiers
double *mean_compare(double *A, int ref, int fis, int fie, int bis, int bie);
double *check_stable_power(double *P,int t);

// Functions used to create .evt files or provide output data to the user
void create_evt_file(double *Pact,double *Qact ,int index,int local_position);
void show_event_data(void);
void print_help(void);


// Simple function to convert an integer to a string (atoi)
char *int2str(int val, int base);
// Convert Global Index to an array[2] of min, sec
int *index_to_time(int global_index);
char *filename_to_date(void);

//////////////////////////////////////////
//     Matrix and vector recipes        //
//  Credit to Numerical Recipes in C    //
//      Second Edition, 1992            //
//////////////////////////////////////////

void nrerror(char error_text[]);
double **dmatrix(long nrl, long nrh, long ncl, long nch);
void ludcmp(double **a, int n, int *indx,float *d);
double *dvector(long nl, long nh);
void free_dvector(double *v,long nl,long nh);
void ludcmp(double **a, int n, int *indx, float *d);
void lubksb(float **a, int n, int *indx,float b[]);
double **invert(double **a,int N);
```

```
double **transpose(double **A, long Arow, long Acol);
double **mat_mult(double **A, long Arow, long Acol, double **B,long Brow, long Bcol);
void show_matrix(double **A, long row, long col);


// This handler allows the UI to send a kill signal to
// ginzu to stop operation.  The handler has an ignore capability that is
// set by the create_evt_file() function.  This prevents ginzu from being
// killed when an event file is being written.
void handle_signal(int signal)
{
	signal_received = 1;
	if (!signal_ignore)
		exit(2);
}


////////////////////////////////////
////   MAIN PROGRAM              /////
////////////////////////////////////

main(int argc, char *argv[])
{

	FILE *fp;
	double D;  // Dummy placeholder for reading in higher harmonics
	double R[A_SIZE];
	double PQf[A_SIZE];
	int Compare[A_SIZE];
	int Jump[A_SIZE];
	int Detect[BUFFER];
	int i=0;
	int j=0;
	time_t curtime;
	struct tm *loctime;

	// Create filename based on our start time
	curtime = time(NULL);
	loctime = localtime(&curtime);
	strftime(filename, sizeof(filename),"/snapshot-%Y%m%d-%H%M%S.txt",loctime);

	// set up signal handler
#ifndef _WIN32
	signal(SIGTERM, handle_signal);
#endif

	// READ IN COMMAND LINE ARGUMENTS.
	if (argc>1)
	{
		for (i=1;i<argc;i=i+2)
		{
```

```
            if (strcmp(argv[i],"v")==0)
            {VERBOSE = atoi(argv[i+1]);       }
            else if (strcmp(argv[i],"l")==0)
            {LOADFILE = atoi(argv[i+1]);}
            else if (strcmp(argv[i],"m")==0)
            {MODE = atoi (argv[i+1]);}
            else if (strcmp(argv[i],"f")==0)
            {strcpy(filename,argv[i+1]);      }
            else if (strcmp(argv[i],"t")==0)
            {GETEVENTS_THRESHOLD = atof(argv[i+1]);}
            else if (strcmp(argv[i],"e")==0)
            {
                  STREAM_INC = atoi(argv[i+1]);
                  if ((STREAM_INC<1) || (STREAM_INC>10))
                  {
                        printf("\nInvalid command line argument - [%s]\n",argv[i]);
                        print_help();
                        exit(0);
                  }
            }
            else if (strcmp(argv[i],"p")==0)
            {P_SCALE = atof(argv[i+1]);}
            else if (strcmp(argv[i],"q")==0)
            {Q_SCALE = atof(argv[i+1]);}
            else if (strcmp(argv[i],"r")==0)
            {REAL_TIMESTAMP = atoi(argv[i+1]);}
            else if (strcmp(argv[i],"d")==0)
            {NUM_PREP_LINES=atoi(argv[i+1]);}
            else
            {
                  printf("\nInvalid command line argument - [%s]\n",argv[i]);
                  print_help();
                  exit(0);
            }
      }
}

printf("[MODE=%d] ",              MODE);
printf("[VERBOSE=%d] ",           VERBOSE);
printf("[LOAD_FILE=%d] ",  LOADFILE);
printf("[FILENAME=%s] ",   filename);
printf("[P_SCALE=%3.5lf] ",       P_SCALE);
printf("[Q_SCALE=%3.5lf] ",       Q_SCALE);
printf("[INC_LINES=%d] ",  NUM_PREP_LINES);
printf("[REAL_TS=%d] ",           REAL_TIMESTAMP);


// This converts the scaling factor so that Ginzu sees KW.
// Note that Ginzu takes the absolute value of the incoming data so it
// is not nesc to consider the correct sign.
P_SCALE = P_SCALE/1000;
```

```
Q_SCALE = Q_SCALE/1000;

// IF MODE IS RO, SET THRESHOLD AND LOCKOUT WINDOW SIZES.
switch(MODE)
{
case 0:
       // CHT Settings
       GETEVENTS_THRESHOLD = 0.4;
       printf("[THRESH=%3.5lf KW] ",GETEVENTS_THRESHOLD);
       GETEVENTS_SKIP = 120+FIR_SIZE;
       break;
case 1:
       // RO Settings
       GETEVENTS_THRESHOLD = 0.5;
       printf("[THRESH=%3.5lf KW] ",GETEVENTS_THRESHOLD);
       GETEVENTS_SKIP = 120+FIR_SIZE;    // Set RO skip window
       break;
case 2:
       // Detect Only settings
       printf("[THRESH=%3.5lf KW] ",GETEVENTS_THRESHOLD);
       GETEVENTS_SKIP = 240 +FIR_SIZE;
       break;
case 3:
       // Streaming Data mode.  No threshold needed.  Data is not sent to CoM detector.
       // No skip window needed.  Events are not found in this mode.
       printf("[STREAM_PERIOD=%d seconds] ",STREAM_INC);
       break;
default:
       // These are the defaults (set above in the headers)
       printf("\n Error: Ginzu called with 'm' parameter which is not valid.  See help.");
       print_help();
       exit(0);
       break;
}

printf("\n");

init_event_table();


////////////////////////////////////////////////////
// IF LOADFILE IS ON, WE CHECK TO SEE IF OUR FILE EXISTS
////////////////////////////////////////////////////

if (LOADFILE)
{
       if ((fp=fopen(filename,"r"))==NULL)
       {
              printf("\nError opening %s.",filename);
              exit(1);
       }
```

```
        else printf("\nSuccess. %s is open.",filename);
}


while (1)
{
        if (VERBOSE)
        {
                //if(VERBOSE) printf("\n\n============================================================");
                //if(VERBOSE) printf("\nReceiving Window %d - Data [%d - %d]",window,window*ADVANCE,window*ADVANCE+BUFFER);
        }
        if (window==0)
        {
                //if(VERBOSE) printf("\nInitializing and loading first window.");
                //LOADS THE FRONT END WITH ZEROS ON THE FIRST WINDOW
                //Default is the first 33 elements

                for (i=0;i<FIR_SIZE;i++)
                {
                        P[i]=0;
                        Q[i]=0;
                }

                // Load elements FIR_SIZE->BUFFER + FIR_SIZE*2-1 with data

                for (i=FIR_SIZE;i<A_SIZE;i++)
                {
                        if (LOADFILE)
                        {
                                if (fscanf(fp,"%lf %lf %lf %lf %lf %lf %lf %lf",&Q[i],&P[i],&D,&D,&D,&D,&D,&D) < 2)
                                        display_and_exit();

                                line_number++;
                                if (line_number>NUM_PREP_LINES)
                                {
                                        display_and_exit();
                                }

                                Q[i]=fabs(Q_SCALE*Q[i]);
                                P[i]=fabs(P_SCALE*P[i]);

                        }
                        else
                        {
                                if (fscanf(stdin,"%lf %lf %lf %lf %lf %lf %lf %lf",&Q[i],&P[i],&D,&D,&D,&D,&D,&D) < 2)
                                        display_and_exit();
                                line_number++;
                                if (line_number>NUM_PREP_LINES)
                                {
                                        display_and_exit();
                                }
```

```
                        Q[i]=fabs(Q_SCALE*Q[i]);
                        P[i]=fabs(P_SCALE*P[i]);

                        //printf("\n%d - [%.32lf , %.32lf]",window,P[i],Q[i]);
                }

                //if (i%(BUFFER/10)==0)if(VERBOSE) printf("-");
        }
        //if(VERBOSE) printf("]");
}
else  // NOT THE FIRST WINDOW
{
        j=0;
        // This slides the last sections of the current window to the right.
        for (i=ADVANCE;i<A_SIZE;i++)
        {
                P[j]=P[i];
                Q[j]=Q[i];
                j++;
        }
        // This loads the new data.
        for (i=0;i<ADVANCE;i++)
        {
                if (LOADFILE)
                {
                        if (fscanf(fp,"%lf %lf %lf %lf %lf %lf %lf %lf",&Q[j],&P[j],&D,&D,&D,&D,&D,&D) < 2)
                                display_and_exit();
                }
                else
                {
                        if (fscanf(stdin,"%lf %lf %lf %lf %lf %lf %lf %lf",&Q[j],&P[j],&D,&D,&D,&D,&D,&D) < 2)
                                display_and_exit();
                }
                line_number++;
                if (line_number>=NUM_PREP_LINES)
                {
                        display_and_exit();
                }
                P[j]=fabs(P_SCALE*P[j]);
                Q[j]=fabs(Q_SCALE*Q[j]);
                j++;
        }
}

// Pass the power data through a LP filter
// For RO, we use Q (as it is cleaner and a better indication of system changes)
// For all others, we use real P.

switch (MODE)
{
```

```
	//CHT System
case 0:
	filterslow(B,P,FIR_SIZE,A_SIZE,R);
	for (i=0;i<A_SIZE;i++)
	{
		PQf[i]=P[i]-R[i];
	}
	break;
	// RO System
case 1:
	filterslow(B,Q,FIR_SIZE,A_SIZE,R);
	for (i=0;i<A_SIZE;i++)
	{
		PQf[i]=Q[i]-R[i];
	}
	break;
	// Detect System
case 2:
	filterslow(B,P,FIR_SIZE,A_SIZE,R);
	for (i=0;i<A_SIZE;i++)
	{
		PQf[i]=P[i]-R[i];
	}
	break;
	// Streaming EVT files
case 3:
	// We don't need to do any CoM filtering because we are
	// just going to display the window
	break;
default:
	break;

}

// If we are NOT streaming (MODE 3), we process results of CoM.
if (MODE!=3)
{

	// Compare the filtered output to the stated threshold.  If the threshold is
	// exceeded, a '1' is put in the 'Compare' array.  If not, a '0' is recorded.
	// So the 'Compare' array contains a '1' at any index that the filtered output
	// exceeds the threshold.
	for (i=0;i<A_SIZE;i++)
	{
		Compare[i]= (fabs(PQf[i])<GETEVENTS_THRESHOLD) ? 0 : 1 ;
	}
	// The 'Jump' array is the absolute derivative of the 'Compare' array so a value of '1'
	// is recorded at any index that the 'Compare' array changes from 0->1 or 1->0.
	Jump[0]=0;
	for (i=1;i<A_SIZE;i++)
	{
```

```
                Jump[i]=Compare[i]-Compare[i-1];
        }
        for (i=0;i<A_SIZE;i++)
        {
                Jump[i]=abs(Jump[i]);
        }
        // Since the 'Jump' array includes a buffer on each end, the indexes of the actual
        // P (or Q) data are offset.  To prevent us from having to correct for this offset,
        // the 'Jump' data is reloaded into a 'Detect' array so that the event indexes match
        // the real index of the event.
        for (i=0;i<BUFFER;i++)
        {
                Detect[i]=Jump[i+FIR_SIZE];
        }
        // The 'Detect' array is passed to the load_index routine and the event tables are
        // updated based on the detections in the current window.
        load_index(Detect);

}

// If classification is required, Ginzu calls the classifier corresponding to the
// correct system.  If no events are detected, Ginzu calls the IC check functions
// and attempts to double check that the state information is accurate.

switch (MODE)
{
case 0:
        if (local_det>0) CHT_CLASSIFY();
        else CHT_IC();
        break;
case 1:
        if (local_det>0) RO_CLASSIFY();
        else RO_IC();
        break;
case 2:
        if (local_det>0)
        {
                slice();
        }
        break;
case 3:
        // Checks the modulus of the window and the stream increment
        // If the modulus is 0, we give a window.
        if ((window%STREAM_INC)==0)
        {
                stream();
        }
        break;
}

// Check the gloval event table to determine if we are
```

```
            // close to the MAX_EVENTS limit.  If we are, write a summary table
            // and then clean the global event table.  Also, need to make sure
            // all global events are classified before we wipe it.  Should also
            // experiment to determine if the global event table can hold more
            // MAX_EVENTS.  An example of a bad day could have 120x24 events (>2500)
            // events, this would blow up the event table.

            if ((MAX_EVENTS-event_count)<10)
            {
                  show_event_data();
                  init_event_table();
            }

            // The ST_WIN_Index index simply tracks the number of steady state window we have given.
            // We reset it when it rises over 999.
            if (ST_WIN_Index>9999)
                  ST_WIN_Index=0;

            window++;
      }
}

char *filename_to_date(void)
{
      int length=0;
      int index;
      int end;
      static char buf[11];
      // This is the index of the last character in the filename.
      end=(int)strlen(filename);
      while (end>0)
      {
            end--;
            if (filename[end]=='-')
            {
                  // we found the place where the time is
                  // load the char array
                  for (index=0;index<11;index++)
                  {
                        buf[index]=filename[end-8+index];
                  }

                  return (buf);
            }
      }
      // we exited and no '-' was found so we load a default
      return ("YEARMODA-HR");
}

int *index_to_time(int global_index)
{
```

```
int *ev_time =(int*)calloc(2,sizeof(int));
ev_time[0]=global_index/7200;
ev_time[1]=(int)(global_index%7200)*60/7200;
return (ev_time);
}

void print_help(void)
{
      printf("\n Usage: ginzu [arg1 #] [arg2 #] [...]");
      printf("\n   example:  dd if=snapshot.txt | ./ginzu    -  This calls default setttings.");
      printf("\n   example:  dd if=snapshot.txt | ./ginzu v 1 l 0 > ss.out");
      printf("\n  ");
      printf("\n  Argument Options");
      printf("\n  v #                 -  Verbose (0=non-verbose) (1=verbose)");
      printf("\n  l #                -  Load Specified File (Always set to 0 if you are piping in PREP)");
      printf("\n  m 0-3             -  System (0=CHT) (1=RO) (2=Detect Only) (3=Stream Only)");
      printf("\n  t #                    -  CoM Filter Threshold (set in KW)");
      printf("\n  e #                    -  Delay between window generation in streaming mode (default is 2 seconds) (Range: 2-
10)");
      printf("\n  f filename.txt - (If l=0) designates the filename that events will have.");
      printf("\n  f filename.txt - (If l=1) designates the filename that will be loaded.");
      printf("\n  p #                    -  Manually scale PREP input into WATTS.");
      printf("\n  q #                    -  Manually scale PREP input into VAR.");
      printf("\n  d #                    -  Designate number of lines in incoming prep data.");
      printf("\n                -  This is used to determine when to stop processing data.");
      printf("\n  r 0 or 1        -  Real-time timestamps (0=parse filename) (1=use actual time).");
      printf("\n\n\n");
}


void init_event_table(void)
{
      int i;
      // CLEAN THE EVENT TEXT DATA.
      for (i=0;i<MAX_EVENTS;i++)
      {
            Class[i]="Empty.";
            Class_ID[i]=0;
            EVC[i]=FALSE;
            event_class_status[i]=FALSE;

      }
      event_count=0;
}


void display_and_exit(void)
// Unless the user has supressed the output table (via the command line),
// a summary table is displayed.
{
      show_event_data();
```

```
	printf("\n");
	printf("\n--> Last line read is %d.",line_number);
	printf("\n");
	exit(0);
}

void filterslow(double *B, double *PQ, int LEN_B, int LEN_PQ, double *FILT_PQ)
// B is the array of filter constants.  PQ is the array to be filtered.
// LEN_B is the length of the filter.  LEN_PQ is the length of the array to be filtered.
// FILT_PQ is the result of the convolution. It is an array of length = Nx.
{
	int i,j,min;

	for (i=0;i<LEN_PQ;i++)
	{
		FILT_PQ[i]=PQ[i]*B[0];
	}
	min =(LEN_B<LEN_PQ) ? LEN_B : LEN_PQ;
	if (LEN_B>1)
	{
		for (i=1;i<(min-1);i++)
		{
			for (j=i;j<LEN_PQ;j++)
			{
				FILT_PQ[j]+=PQ[j-i]*B[i];
			}
		}
	}
}

void load_index(int *x)
// load_index takes the Detect array which has [1] where there is an actual
// transient. It loads the global event_index and increments the event_count.
// It also updates the 'local_index' based on the events that need to be
// classified within the current window.

{
	extern int window;
	//extern int found_recent;
	int i,j,already_found,in_lock_out,classified;

	in_lock_out=0;
	local_det=0;

	// for each entry in the Detect array, which is x
	// Note: the index starts at 1 instead of 0.  This prevents the detector
	// from finding a detecting the front edge if their is a positive power
	// at startup - their are better ways to do this but this works for now.
	// In the future, this check should be moved to a conditional inside the
	// loop below.
	for (i=1;i<BUFFER;i++)
```

```
	{
		// if an event occurred and you are not in a lock-out window
		// if found_recent is negative, then you are not in a lock-out window
		if (x[i]==1)
		{
			// Check to see if we already detected this event or if the event is in a lockout window
			already_found=	FALSE;
			in_lock_out= FALSE;
			classified=	FALSE;
			for (j=0;j<MAX_EVENTS;j++)
			{
				if (((window*ADVANCE+i)==event_index[j]))
				{
					already_found=	TRUE;
					if (event_class_status[j]==TRUE)
					{
						classified=  TRUE;
						//if (VERBOSE) printf("\n Found event @ %d that is already classified and in the global
event table.",window*ADVANCE+i);
					}
				}
				if (((window*ADVANCE+i)>event_index[j]) && (window*ADVANCE+i)<=(event_index[j]+GETEVENTS_SKIP))
				{
					in_lock_out= TRUE;
					//if (VERBOSE) printf("\n Found event @ %d that is in a lockout window.",window*ADVANCE+i);
				}
			}
			if ((!already_found) && (!in_lock_out))
			{
				//if(VERBOSE) printf("\n* %d: TRANSIENT DETECT, locking out %d-
%d",window*ADVANCE+i,window*ADVANCE+i+1,window*ADVANCE+i+GETEVENTS_SKIP);
				event_index[event_count]= window*ADVANCE+i;  // load the global event index
				//if(SHOWREADS) printf("\n Event detected @ %d.",window*ADVANCE+i);
				event_count++; // increment the number of global events
				loc_index[local_det]=i;
				local_det++;
				//found_recent=GETEVENTS_SKIP;  // set the lock-out window at max
			}

			if ((already_found) && (!classified))
			{
				//if(VERBOSE) printf("\n  * %d: TRANSIENT REDETECT OF UNCLASSIFIED EVENT.",window*ADVANCE+i);
				loc_index[local_det]=i;
				local_det++;
			}
		}

	}
	//if (local_det==0) if(VERBOSE) printf("\n -> No new events detected in this window.");
}
```

```
void show_event_data(void)
// Displays the final result table.
{
	extern int event_count,event_index[MAX_EVENTS];
	int i;
	printf("\n=== Final Event Data ====");
	printf("\n[#]\tIndex\tEVC\tClassification",event_count);
	printf("\n___\t_____\t____\t_____________________");
	if (event_count==0) printf("\nNo Detections in this dataset.");
	for (i=0;i<event_count;i++)
	{
		printf("\n[%.3d]\t%.2d:%.2d\t%d\t%s
",i,index_to_time(event_index[i])[0],index_to_time(event_index[i])[1],EVC[i],Class[i]);
	}
}


double *exemplar_fit(double *Pact, int index, int pump_type)

// exemplar_fit() performs a correlation of the event to a given template and returns a gain factor corresponding
// to the strength of the match.  The algorithm actually performs 10 correlations by shifting the template 5
// indexes in both directions.  The point of maximum correlation is then returned.

// Index is the index of the event in the Pact array.
// pump_type is given to designate which template to attempt to match. (1=VacPump, 2=DisPump)

// Returns a double R[4] where R[0]=minJ; R[1]=optimal_shift; R[2]=b[1]; R[3]=b[2];
// N is length of template vector (30).

// Start is a value stored in the exemplar file - tells match where the exemplar starts relative to
// the location of the massive change.


{
	int N=30;
	int i,j,k;
	int shift = 5;
	double **event_window,minJ,optimal_shift,maxGain;
	int start = 4;
	double *ex;
	double **A,**At,**b,**Ab,**ew_Ab,**J;
	double *R;
	R=(double*)calloc(4,sizeof(double));

	if (pump_type==1)
		// do VP match
		ex=imP_P1;
	else if (pump_type==2)
		// do DP match
		ex=imP_P2;
	else if (pump_type==3)
```

```
        // do LP match
        ex=imP_LP;
else if (pump_type==4)
        // do HP match
        ex=imP_HP;


event_window=dmatrix(1,N,1,1);
A=dmatrix(1,N,1,2);
J=dmatrix(1,4,1,shift*2+1);
ew_Ab=dmatrix(1,N,1,1);

for (i=1;i<=N;i++)
{
        A[i][1] = 1.0;
        A[i][2] = ex[i-1];
}
At=transpose(A,N,2);


////////////////////////////////////////////
// Make the assumption that the transient could
// actually be a few indexes to the right or left
// of the original detect.  So we examine the fit
// over that entire range and take the ideal match.
////////////////////////////////////////////////
for (i=-shift;i<=shift;i++)
{
        ////////////////////////////
        // LOAD THE EVENT WINDOW    //
        ////////////////////////////
        for (j=1;j<=N;j++)
        {
                event_window[j][1] = Pact[index-start+i+j];
        }

        ////////////////////////////////
        // Do the correlation           //
        ////////////////////////////////
        b=mat_mult(invert(mat_mult(At,2,N,A,N,2),2),2,2,mat_mult(At,2,N,event_window,N,1),2,1);
        Ab=mat_mult(A,N,2,b,2,1);
        for (k=1;k<=N;k++)
                ew_Ab[k][1]=event_window[k][1]-Ab[k][1];  // W, DIFFERENCE BETWEEN WINDOW AND
                                                                                        // OPTIMALLY SCALED TEMPLATE.


        ////////////////////////////////////////////////////////
        // Load the result matrix with the shift, the residual value,
        // and the gain factor for the correlation.
        ////////////////////////////////////////////////////////
        J[1][i+shift+1]=i;  // INDEX OF SHIFT RANGING FROM -5 to +5
```

```
            J[2][i+shift+1]=mat_mult(transpose(ew_Ab,N,1),1,30,ew_Ab,N,1)[1][1];
            J[2][i+shift+1]=J[2][i+shift+1]/N;  // SUMMED SQUARED ERROR NORMALIZED FOR ARRAY SIZE
            J[3][i+shift+1]=b[1][1];  // DC SHIFT TO MINIMIZE SQUARED ERROR
            J[4][i+shift+1]=b[2][1];  // SIGNAL GAIN TO MINIMIZE SQUARED ERROR
      }

      ////////////////////////////////////////////////////////////////
      // Search the J matrix for the point at which gain is maximum and
      // return array R[] with the associated values.
      ////////////////////////////////////////////////////////////////

      maxGain=J[4][1];
      optimal_shift=J[1][1];

      for (i=2;i<=2*shift+1;i++)

            if (J[4][i]>maxGain)
            {
                  maxGain=J[4][i];
                  minJ=J[2][i];
                  optimal_shift=J[1][i];
                  R[0]=minJ;  // THIS IS THE LOWEST SUM SQUARED ERROR.
                  R[1]=optimal_shift;  // THIS IS THE INDEX OF OPTIMAL SHIFT.
                  R[2]=J[3][i];  // THIS IS THE DC SHIFT AT "BEST LOCATION"
                  R[3]=J[4][i];  // THIS IS THE AC GAIN AT "OPTIMAL LOCATION"
            }

            return(R);

            // CURRENTLY, THIS ALGORITHM RETURNS THE SHIFT DATA COLUMN THAT CORRESPONDS TO
            // THE POINT WHERE THE HIGHEST AMOUNT OF AC GAIN WAS USED.. THAT'S SCREWED UP.
            // IT SHOULD SEARCH THE J MATRIX FOR THE LOWEST J (I THINK).

            // THIS NEEDS A REWRITE TO MOVE TC FUNCTION TO THIS FUNCTION
            // THIS SHOULD JUST RETURN COMPOSITE GAIN FOR THE PROVIDED TEMPLATE
            // AND PACT INDEX.
}



double *transient_classifier(double *Pact, int ev_loc,  int pump_type)
//transient_classifier() is a shell for the actual correlator (exemplar_fit()),
//that ensures that the gain values are scaled properly, normalized and within
//expected bounds.

//Given an actual Power array and the event location, returns a double[3] with..
//TC_return[0]=match_val_norm;
//TC_return[1]=match_val_gain;
//TC_return[2]=composite_gains;
{
      double *ex,*R,*TC_return;
```

```
	double match_val_norm, match_val_gain, composite_gains;

	// Classification parameters - See K.D. Lee 2003
	double a_thresh = 0.6;  // Normalized gain threshold.  Optimized AC gain should be above this
                                        // value in all cases.  An optimal AC gain below this value means there is
                                        // likely a very weak match.

	double gam_thresh = 2;

	ex=imP_P1;
	TC_return=(double*)calloc(3,sizeof(double));

	// NOTE: DOES THIS DEAL WITH A PUMP START IF THE PRE_EVENT
	// POWER WAS NON-ZERO.. IT DOESN'T APPEAR TO.

	R=exemplar_fit(Pact,ev_loc,pump_type);
	// Returns a double R[4] where R[0]=minJ; R[1]=optimal_shift; R[2]=b[1]; R[3]=b[2];

	match_val_norm = R[0];
	////  % This is the minimum value of J from the event classifier. See exemplar_fit for an explanation.

	match_val_gain = R[3];
	// This is the AC gain factor from the fit.

	if (match_val_gain > 1)
		// If the gain is greater than 1, normalize so that the value is less than 1.
		match_val_gain = 1/match_val_gain;


	if (match_val_gain < 0)
	{
		//  % If the gain is less than zero, make it zero.
		match_val_gain = 0;
	}
	composite_gains =  match_val_gain*(1-(1-a_thresh)/gam_thresh*match_val_norm);

	TC_return[0]=match_val_norm;
	TC_return[1]=match_val_gain;
	TC_return[2]=composite_gains;

	return(TC_return);
}



int search_lockout_window(int i, int ev, double *deriv)
{

	// search_lockout_window() examines the N=GETEVENTS_SKIP elements that follow
	// the transient index for an additional (or multiple additional) events.
	// This can occur if, for example 2 pumps start within a fraction of a second
```

```
	// of eachother.  If this occurs, the algorithm returns a non-zero value to indicate
	// that the classifcation must be done in a manner other that templated correlation


	int d = 0;
	int j;

	// Jump to 60 points after the detected event.
	// This forces the .5 seconds from the detection to be ignored
	// modified from original code
	j = ev+60;
	while (j < (ev+GETEVENTS_SKIP))
	{
		//		% This loop uses the derivative to search the lockout window
		//		% for additional events.  If deriv(j) > 3, it is decided that
		//		% another event has occured in the lockout window.  If so, j is incremented
		//		% by 7.  This value may need to be larger.  j is incrmemented
		//		% because it is true that the derivative will remain somewhat
		//		% large for a few points after the new event.

		// This was increased to 3 based on testing.
		if (deriv[j] > 3)
		{
			d++;
			j += 7;
		}
		else {
			j ++;
		}
	}
	return(d);
}


void arbitrate_event_overlap(int i, int ev, double *Pact, double *Qact,int CGI)
{
	// This is the sister function to search_lockout_window().  If a double
	// event is detected, this algorithm classifies a double event based on
	// the change in P and Q during the transient.

	double delta_P, delta_Q;

	// Two events happened, let's use s.s. info to determine what happened
	// We won't use the examplar, we"ll classify based on state space.

	delta_P = mean_compare(Pact,ev,600,610,-12,-60)[0];
	delta_Q = mean_compare(Qact,ev,600,610,-12,-60)[0];

	if(VERBOSE) fprintf(stdout,"\nDouble Event @ %d :",ev);

	if (delta_P > 2.35 && delta_P < 3.35 && delta_Q > 2.25 && delta_Q < 2.7)
```

```
	{
		dis ++;
		vac ++;
		if(VERBOSE) fprintf(stdout, "\n%d Discharge +1",ev+window*ADVANCE);
		if(VERBOSE) fprintf(stdout, "\n%d Vacuum +1",ev+window*ADVANCE);
		Class[CGI]="DP On and VP On (RPC)";

		Class_ID[CGI]=13;
		create_evt_file(Pact,Qact,CGI,ev);
	}
	else if (delta_P > 3.3 && delta_P < 3.8 && delta_Q > 1.5 && delta_Q < 1.9)
	{
		vac = vac+2;
		if(VERBOSE) fprintf(stdout, "\n%d Vacuum +1",ev+window*ADVANCE);
		if(VERBOSE) fprintf(stdout, "\n%d Vacuum +1",ev+window*ADVANCE);
		Class[CGI]="2 VP on (RPC)";
		Class_ID[CGI]=11;
		create_evt_file(Pact,Qact,CGI,ev);
	}

	else
	{
		if(VERBOSE) fprintf(stdout, "\n%dUnknown multiple events",ev+window*ADVANCE);
		Class[CGI]="Unknown multiple events";
		Class_ID[CGI]=000;
		create_evt_file(Pact,Qact,CGI,ev);
	}


}

double *mean_compare(double *A, int ref, int fis, int fie, int bis, int bie)
// This takes an array of doubles (A[]) and compares the mean of the values from
// [fis--> fie] to the mean of values [bis-->bie].it also verifies that
// these references will not be out of array bounds.  If the references are out of bounds
// it will shorten the windows.  This may not be the right way to do this.
// It should probably be changed to return an error if the values are OOB.
// It returns the difference mean(A[fis-->fie] - A[bis-->bie])

// IMPORTANT:  THIS ASSUMES A IS OF SIZE = BUFFER
{
	int i;
	double after=0,before=0;
	double *R =(double*)calloc(3,sizeof(double));

	if ((bie+ref)<=0)
	{
		if(VERBOSE) printf("\n Far before range in mean comparison is out of bounds. Setting to max.");
		bie=ref;
	}
	if ((bis+ref)<=0)
```

```
	{
		if(VERBOSE) printf("\n Close before range in mean comparision is out of bounds. Setting max.");
		bis=ref;
	}
	if ((fie+ref)>=BUFFER)
	{
		if(VERBOSE) printf("\n Far after range in mean comparision is out of bounds. Setting max.");
		fie=ref;
	}
	if ((fis+ref)>=BUFFER)
	{
		if(VERBOSE) printf("\n Close after range in mean comparision is out of bounds. Setting max.");
		fis=ref;
	}

	for (i=fis;i<=fie;i++)
		after+=(A[ref+i]);

	after=(after/(fie-fis+1));

	for (i=bis;i>=bie;i--)
		before+=(A[ref+i]);

	before=(before/(bis-bie+1));
	R[0]=(after-before);
	R[1]=before;
	R[2]=after;
	return (R);
}


void CHT_CLASSIFY(void)

// CHT_CLASSIFY is the classifier shell.  CHT_CLASSIFY should be called after every window
// that has a detect.  CHT_CLASSIFY recieves a buffer which contains local_detect elements
// and the elements of loc_index correspond to the true location of the transient in the
// ACTUAL P or Q ARRAYS (Pact, Qact) (NOT THE BUFFERED ARRAY).

// Note: The global P,Q arrays are buffered on each end to prevent edge effects during filtering.
// To simplify references to the P/Q arrays, CHT_CLASSIFY loads the Pact and Qact arrays which simply
// fixes the offset issue that buffering causes.
{
	double Pact[BUFFER], Qact[BUFFER];  // ACTUAL P,Q arrays (UNBUFFERED)
	int i,j;

	int VPi, DPi, CVPi;		// # of pumps running prior to event

	int vsd=0, dsd=0;		// used in 'ON' classifier to record the type of pump that was detected

	double *deltaP_event,*deltaQ_event;		// change in P across the event index.
	double *deltaP_window, *deltaQ_window;  // change in Q across the lockout window.
```

```
double gain[2];                                          // The Correlation Gain (event vs. templates)


double P_deriv[BUFFER];    // The derivative of the Pact array.  Used to check for double events.

int dbl_detect,last_event=0,t_to_next=0;
// dbl_detect is incremented if multiple events appear in the same lockout window
// last_event is flagged to 1 if the classifier is checking the final event in the window
// if the event is not the last_event, t_to_next indicates how many indexes until the subsequent event

double *TC_return;  // The return from the transient classifer.

double change ,P_5seconds_later, P_final;
// change = delta between post-transient & pre-transient power
// P_5seconds_later is used by vacuum pump classifier to check if power is lower than
//                  expected 5 seconds after the VP On is detected.  This can indicate a
//                  clogged pump has been located.
// P_final is the average power after an event used to determine if P went to zero

int CGI;            // Current Global Index.  This is the index of the event in the dataset.
// CGI = (WINDOW_NUMBER * ADVANCE) + LOCAL_INDEX

char tempstr[100];  // Temporary string storage used if concatenating onto 'pump on' events


// Load the P and Q arrays (as discussed above)
for (i=0;i<BUFFER;i++)
{
      Pact[i]=P[i+FIR_SIZE];
      Qact[i]=Q[i+FIR_SIZE];
}



//////////////////////////////////////////
//   LOAD DERIVATIVE POWER ARRAY FOR     //
// FUTURE USE TO CHECK FOR DBL EVENTS    //
//////////////////////////////////////////

// Now, we try to see if there are any other events stuck inside the lockout
// window.  This part of the script computes a numerical approximation to
// the derivative.  It then searches a moving 10-point window after the event.

P_deriv[0]=0;
for (i=1;i<BUFFER;i++)
{
      P_deriv[i]=Pact[i]-Pact[i-1];
}
for (i=0;i<BUFFER;i++)
{
      P_deriv[i]=fabs(P_deriv[i]);
```

```
}

////////////////////////////
//  BEGIN CLASSIFICATION  //
////////////////////////////

// for each event
for (i=0;i<local_det;i++)
{
	//if(VERBOSE) printf("\n\nClassifying event at <%d>", loc_index[i]+window*ADVANCE);
	// check if it is the last event in the window. If not, get how long until next event

	if (((loc_index[i]-120)<0) || ((loc_index[i]+700)>BUFFER))
	{
		//if(VERBOSE) printf("\n\nClassifying event at <%d> - OOB ", loc_index[i]+window*ADVANCE);
		break;
	}
	else
	{
		// Since it is not unclassifiable, we are going to classify it now.
		// So we search the global event table and set the status to TRUE for the
		// appropriate index (assuming that it will be classified).
		for (j=0;j<MAX_EVENTS;j++)
		{
			if (loc_index[i]+window*ADVANCE==event_index[j])
			{
				event_class_status[j]=TRUE;
				CGI=j;
			}
		}
	}

	if (i==local_det-1)
		last_event =1;
	else
		t_to_next = loc_index[i+1]-loc_index[i];


	// Record the pumps that were running pre-event
	VPi=vac;
	DPi=dis;
	CVPi=clogged_vac;

	// Calculate the strength of correlation with VP and DP templates
	gain[0]=transient_classifier(Pact,loc_index[i],1)[2];
	gain[1]=transient_classifier(Pact,loc_index[i],2)[2];

	// Calculate the relevant average power levels
	deltaP_window = mean_compare(Pact,loc_index[i],GETEVENTS_SKIP,GETEVENTS_SKIP+60,-2,-60);
	deltaQ_window = mean_compare(Qact,loc_index[i],GETEVENTS_SKIP,GETEVENTS_SKIP+60,-2,-60);
	deltaP_event = mean_compare(Pact,loc_index[i],5,30,-5,-30);
```

```
deltaQ_event = mean_compare(Qact,loc_index[i],5,30,-5,-30);

change = deltaP_event[0];
if(VERBOSE) printf("\n\nClassifying event at <%d>", loc_index[i]+window*ADVANCE);
if(VERBOSE) printf(" - DeltaP_event =[%.3lf]", change);

if (fabs(change)<.25)
{
      if(VERBOSE) printf("\nChange in power is negligible across event - possible noise.");
      Class[CGI]="Unclassifiable (No P Change)";
      Class_ID[CGI]=000;
      //create_evt_file(Pact,Qact,CGI,loc_index[i]);
      break;
}


////////////////////////////////
// CHECK FOR A MULTIPLE EVENT  //
////////////////////////////////

if (change>0)
{
      if(VERBOSE) printf("\n -> Checking for double ON event");

      // Now, we try to see if there are any other events stuck inside the lockout
      // window.  This part of the script computes a numerical approximation to
      // the derivative.

      dbl_detect = search_lockout_window(i,loc_index[i],P_deriv);

      // If they did, we deal with them outside the normal classifer via
      // arbitrate_event_overlap().

      if (dbl_detect > 0)
      {
            if(VERBOSE) printf(" -> Arbitrating event overlap");
            arbitrate_event_overlap(i,loc_index[i],Pact,Qact,CGI);
            create_evt_file(Pact,Qact,CGI,loc_index[i]);
            break;
      }
      else if(VERBOSE) printf(" -> No double event detected.");
}

//////////////////////////////
//  CLASSIFY THE ON EVENT   //
//////////////////////////////

if (change>0)
{
      ////////////////////////////////////////
      // DO THE CORRELATION WITH A VAC PUMP  //
```

```
////////////////////////////////////////

if(VERBOSE) printf("\n -> Classifying ON -> Running transient classifer");
TC_return=transient_classifier(Pact,loc_index[i],1);
if(VERBOSE) printf(" - > VP template Comp Gain = [%.3lf]",TC_return[2]);
gain[0]=TC_return[2];

////////////////////////////////////
// IF IT LOOKS LIKE A VACUUM PUMP //
////////////////////////////////////

if (TC_return[2] > 0.45)
{  // START Vacuum PUMP CLASS
       if(VERBOSE) printf("\n -> Classified as Vacuum Pump Event.   ");
       vac ++;
       vsd =1;

       ////////////////////////////////////////////////////////////////////////////////
       //  Now check the DeltaP (P_5seconds_later) to determine if the VP is loaded .
       //  If the Power change is < .8 KW,  it is clogged.
       ////////////////////////////////////////////////////////////////////////////////

       if ((last_event) || ((!last_event) && (t_to_next>600)))
       {
              // This first condition should always be met
              // because of the sliding windows approach.  The event
              // that you are classifying will almost alwasy be the 'last'
              // event in the current window (because the windows are relatively small).

              P_5seconds_later = mean_compare(Pact,loc_index[i],600,650,-12,-60)[0];

              // Magic numbers.  This means Power went up but not enough to
              // match a VP start... so we're probably clogged seal.
              // The low bound is in because if P<0.1 at 5 seconds after, the
              // VP probably just turned off.
              if ((P_5seconds_later < 0.8) && (P_5seconds_later > 0.1))
              {
                     if(VERBOSE) fprintf(stdout,"\n%d  Clogged Vacuum  +1",loc_index[i]+window*ADVANCE);
                     clogged_vac = 1;
                     Class[CGI]="Vacuum Pump on, Seal Clogged";
                     Class_ID[CGI]=5;
              }
              else
              {

                     if(VERBOSE) fprintf(stdout,"\n%d  Vacuum  +1",loc_index[i]+window*ADVANCE);
                     Class[CGI]="Vacuum Pump on";
                     Class_ID[CGI]=1;
              }

       }
```

```
        // Basically, if this code is hit, it means that an event occured within 5 seconds
        // of the VP On event so we can't really check for clogging based on P change.  So
        // we assume it is a normal loaded run (i.e. no clogging).
        else
        {
              Class[CGI]="Vacuum Pump on";
              Class_ID[CGI]=1;
              if(VERBOSE) printf("\nNext transient was too close to the start.");
              if(VERBOSE) fprintf(stdout,"\n%d  Vac (could be clogged)  +1",loc_index[i]+window*ADVANCE);
        }



}  // END VACUUM PUMP CLASS

////////////////////////////////////
// NO VACUUM, CHECK IF IT IS A    //
//       DISCHARGE PUMP                    //
////////////////////////////////////
else
{  // START DISCHARGE PUMP ON
        if(VERBOSE) printf("\n -> Not vacuum, Checking Discharge.");


        ////////////////////////////////////////////
        // DO THE CORRELATION WITH A DISCH PUMP  //
        ////////////////////////////////////////////

        TC_return=transient_classifier(Pact,loc_index[i],2);

        if(VERBOSE) printf(", DP template Comp Gain = [%.3lf]", TC_return[2]);

        gain[1]=TC_return[2];

        if (TC_return[2] > 0.45)
        {
              Class[CGI]="Discharge Pump on";
              Class_ID[CGI]=3;
              if(VERBOSE) fprintf(stdout,"\n%d  Discharge  +1",loc_index[i]+window*ADVANCE);
              dis ++;
              dsd =1;
        }

        ////////////////////////////////////////////////////////////////////////
        // THE SHAPE IS NOT CORRELATED BUT THE DELTAP ACROSS THE EVENT IS POSITIVE
        ////////////////////////////////////////////////////////////////////////
        else
        {
              Class[CGI]="Up Power (no good pump match)";
              Class_ID[CGI]=7;
              if(VERBOSE) fprintf(stdout,"\n Probable ON event, no match",loc_index[i]+window*ADVANCE);
```

```
            // May be a good idea to guess at a pump here

      }

} // end else (discharge pump classifier)

////////////////////////////////////////////////////////////////////////////
// NOW CHECK FOR SPECIAL EVENTS BASED ON THE POWER CHANGE DURING THE EVENT
// THIS CAPTURES EVENTS SUCH AS LOAD CYCLES AND SOME OTHER MULTIPLE EVENTS.
////////////////////////////////////////////////////////////////////////////

//if(VERBOSE) printf("\nNow Check special cases.");
// Changed from original classifer, indexes moved away from event

// This should never trip based on earlier check
if ((loc_index[i]+GETEVENTS_SKIP+2) >= BUFFER)
{
      if (VERBOSE) printf("\n *Event window is rolling forward out of buffer.");
      if (VERBOSE) printf("\n *Not checking special cases for this event.");
      //create_evt_file(Pact,Qact,CGI,loc_index[i]);
}
else
{ // BEGIN SPECIAL CLASSIFY
      //if(VERBOSE) printf("\n   [delP=%.5lf] [delQ=%.5lf]",deltaP_window[0],deltaQ_window[0]);
      if ((deltaP_window[0] > 0.1) && (deltaQ_window[0] < - 0.1))
            // Real Power Increased and Reactive Decreased across the window.
            // This indicates a Discharge Pump turned off during the ON event.
      {
            strcpy(tempstr,Class[CGI]);
            strcat(tempstr,", Disch Pump off during on");
            Class[CGI]=(char *)malloc(strlen(tempstr)+1);
            strcpy(Class[CGI],tempstr);
            Class_ID[CGI]+=40;
            //create_evt_file(Pact,Qact,CGI,loc_index[i]);
            dis--;
            if(VERBOSE) printf("\nSpecial Case: Discharge pump turned off during on.");
            if(VERBOSE) fprintf(stdout,"\n%d Dis -1",loc_index[i]);
      }
      else if ((deltaP_window[0] < -0.1) && (deltaQ_window[0] > 0.1))
            // Real Power Decreased while Reactive Increased across the window.
            // This indicates a Discharge Pump turned off during the ON event.
      {
            vac --;
            if (clogged_vac != 0)
            {
                  strcpy(tempstr,Class[CGI]);
                  strcat(tempstr,", VP off with bad seal during on.");
                  Class[CGI]=(char *)malloc(strlen(tempstr)+1);
                  strcpy(Class[CGI],tempstr);
                  Class_ID[CGI]+=60;
                  //create_evt_file(Pact,Qact,CGI,loc_index[i]);
```

```
                if(VERBOSE) printf("\nSpecial Case: Clogged Vac off during on.");
                if(VERBOSE) fprintf(stdout,"\n%d Clogged Vac -1",loc_index[i]+window*ADVANCE);
                clogged_vac = 0;
        }
        else
        {
                strcpy(tempstr,Class[CGI]);
                strcat(tempstr,",  VP off during on");
                Class[CGI]=(char *)malloc(strlen(tempstr)+1);
                strcpy(Class[CGI],tempstr);
                Class_ID[CGI]+=20;
                //create_evt_file(Pact,Qact,CGI,loc_index[i]);
                if(VERBOSE) printf("\nSpecial Case: Vac off during on.");
                if(VERBOSE) fprintf(stdout,"\n%d Vac -1",loc_index[i]+window*ADVANCE);
        }
}
else if ((fabs(deltaP_window[0]) < 0.1) && (fabs(deltaQ_window[0]) < 0.1))
        // Both Real and Reactive power did not change across the window.
        // This indicates that whatever turned on, immediately turned off.
{
        strcpy(tempstr,Class[CGI]);
        strcat(tempstr," and Off.");
        Class[CGI]=(char *)malloc(strlen(tempstr)+1);
        strcpy(Class[CGI],tempstr);

        if(VERBOSE) printf("\nProbable Cycling load - no power change.");
        if (dsd)
        {
                if(VERBOSE) fprintf(stdout,"\n%d Dis -1",loc_index[i]+window*ADVANCE);
                Class_ID[CGI]+=40;
                //create_evt_file(Pact,Qact,CGI,loc_index[i]);
                dis--;
        }
        else if (vsd)
        {
                if(VERBOSE) fprintf(stdout,"\n%d Vac -1",loc_index[i]+window*ADVANCE);
                Class_ID[CGI]+=20;
                //create_evt_file(Pact,Qact,CGI,loc_index[i]);
                vac--;
        }
        else
        {
                if(VERBOSE) fprintf(stdout,"\n%d Load Cycled.",loc_index[i]+window*ADVANCE);
                Class_ID[CGI]+=80;
                //create_evt_file(Pact,Qact,CGI,loc_index[i]);
        }
}
else
{
        // No double event occured.
        // create_evt_file(Pact,Qact,CGI,loc_index[i]);
```

```
            }

        } // END SPECIAL CLASSIFY

        dsd=0;
        vsd=0;

} // end of ON event analysis

////////////////////////////
//  CLASSIFY THE OFF EVENT  //
////////////////////////////

else
{
        if(VERBOSE) printf("\n -> Classifying OFF");

        ////////////////////////////////
        // CHECK TO SEE IF THE POWER      //
        // GOES TO ZERO AFTER THE EVENT  //
        ////////////////////////////////

        P_final = deltaP_event[2];

        if(VERBOSE) printf("\n -> P post-off is %lf",P_final);

        ////////////////////////////////////////////////////////////////////////
        // IF IT DID, CLASSIFY THE EVENT BASED ON WHAT PUMPS WERE
        // RUNNING PRIOR TO THE EVENT.  IF P=0, THEN THOSE PUMPS TURNED OFF.
        ////////////////////////////////////////////////////////////////////////

        if (fabs(P_final) < 0.3)
        {
                if(VERBOSE) printf("\n -> Power_returns_to_zero");
                if(VERBOSE) fprintf(stdout, "\n All Off Event @ %d",loc_index[i]+window*ADVANCE);
                clogged_vac = 0;
                vac = 0;
                dis = 0;
                Class[CGI]="All pumps off";
                Class_ID[CGI]=9;

        }

        ////////////////////////////////////////////////////////////////////////
        // IF P is (+), CLASSIFY BASED ON RELATIVE CHANGE AND
        // KNOWLEDGE OF HOW MUCH POWER WOULD CHANGE FOR SPECIFIC PUMP OFF EVENTS.
        ////////////////////////////////////////////////////////////////////////
        else
        {
                if(VERBOSE) printf("\n -> OFF event but P is (+)");
                {
```

```
                                    if ((fabs(deltaP_event[0]) > 1.5) && (fabs(deltaP_event[0]) < 2.5) && (fabs(deltaQ_event[0]) >
0.3) && (fabs(deltaQ_event[0]) < 1.3) && vac>0 )
                                    {
                                          // single vac off
                                          vac = vac-1;
                                          if(VERBOSE) fprintf(stdout,"\n%d Vac  -1",loc_index[i]+window*ADVANCE);
                                          Class[CGI]="Vacuum Pump off (RPC)";
                                          Class_ID[CGI]=02;
                                          //create_evt_file(Pact,Qact,CGI,loc_index[i]);
                                    }
                                    else if ((fabs(deltaP_event[0]) > 0.5) && (fabs(deltaP_event[0]) < 1.5) && (fabs(deltaQ_event[0])
> .8) && (fabs(deltaQ_event[0]) < 2.0) && dis>0)
                                    {
                                          // single disch off
                                          dis = dis-1 ;
                                          if(VERBOSE) fprintf(stdout,"\n%d Dis  -1",loc_index[i]+window*ADVANCE);
                                          Class[CGI]="Discharge Pump off (RPC)";
                                          Class_ID[CGI]=04;
                                          //create_evt_file(Pact,Qact,CGI,loc_index[i]);
                                    }

                                    else
                                    {
                                          if(VERBOSE) fprintf(stdout,"\n%d Unidentified downpower",loc_index[i]+window*ADVANCE);
                                          Class[CGI]="Unidentified downpower";
                                          Class_ID[CGI]=0;
                                          //create_evt_file(Pact,Qact,CGI,loc_index[i]);
                                    }
                              } // end function
                        }
            }    // END OFF CLASSIFY


            create_evt_file(Pact,Qact,CGI,loc_index[i]);

      }
} // END CHT_CLASSIFY


void CHT_IC(void)
// CHT_IC() is a system state verification check (i.e. the # and types of pumps which are
// currently thought to be running) .  The stability of the system is recorded
// through a global variable ('stable').  If the system is not stable, CHT_IC() determine if
// it has returned to stability.  If it has returned to stability, CHT_IC() performs a number
// of checks to see if the classifier state is consistent with current power levels.
// If the state is not consistent, CHT_IC() attempts to fix the state.

// Example: if the system is not stable, CHT_IC() gets called and calculates power
// and ? over the current window.  If the sigma is low and P~0, CHT_IC() looks at system
// state to see if any pumps are running.  If ginzu thinks a pump is running but P~0 and
```

```
// stable, an inconsistency exists (i.e. a running pump would cause some system Power to exist).
// CHT_IC() will then correct the state (i.e. set the pumps to off).

// Similar checks exist for other types of state verification.

{
	double *stats;
	int total_running_pumps,i;
	double Pact[BUFFER], Qact[BUFFER];  // ACTUAL P,Q arrays (UNBUFFERED)
	for (i=0;i<BUFFER;i++)
	{
		Pact[i]=P[i+FIR_SIZE];
		Qact[i]=Q[i+FIR_SIZE];
	}

	stats = check_stable_power(Pact,BUFFER/120);
	total_running_pumps= vac+dis+clogged_vac;
	// stats[0] == AVERAGE
	// stats[1] == SD


	// IF (POWER ~ 0) AND (STEADY AT 0) AND (WE THINK A PUMP IS RUNNING),
	// WE HAVE AN INCONSISTENCY.  SO WE FIX THE STATE.
	if ((stats[0]<.25) && (total_running_pumps>0) && (stats[1]<.1))
	{
		if (VERBOSE) printf("\nP = [%.3lf,%.3lf], State = [V-%d, CV-%d, D-%d]",stats[0],stats[1],vac, clogged_vac,dis);

		if (VERBOSE) printf("\n ****[EC1] System Power is 0, but State = [V-%d, CV-%d, D-%d]",vac, clogged_vac,dis);
		if (VERBOSE) printf("\n Assuming everything is off. ");
		// Need to add code here to inform the UI of the specific pump.
		vac=0; dis=0; clogged_vac=0;
	}

	// IF (POWER > 0) AND (STEADY AT >0) AND (WE THINK NO PUMPS ARE RUNNING),
	// WE HAVE AN INCONSISTENCY.  SO WE GUESS VACUUM PUMP AND THEN FIX THE STATE.
	else if ((stats[0]>.5) && (total_running_pumps==0) && (stats[1]<.1))
	{
		if (VERBOSE) printf("\nP = [%.3lf,%.3lf], State = [V-%d, CV-%d, D-%d]",stats[0],stats[1],vac, clogged_vac,dis);

		if (VERBOSE) printf("\n ****[EC2] System Power is (+), but State = [V-%d, CV-%d, D-%d]",vac, clogged_vac,dis);
		if (VERBOSE) printf("\n Guessing a vacuum pump is on. ");
		vac=1;
		// Need to add code here to inform the UI of the vacuum pump assumption.
	}

	// IF THE CLASSIFIER EVER PUTS US INTO A STATE WHERE A NEGATIVE # OF PUMPS EXISTS,
	// OR A # OF PUMPS WHICH CAN'T EXIST, WE KNOW THIS IS INCORRECT SO WE FIX IT.
	if ((vac<0) || (dis<0) || (clogged_vac<0))
	{
		if (VERBOSE) printf("\nP = [%.3lf,%.3lf], State = [V-%d, CV-%d, D-%d]",stats[0],stats[1],vac, clogged_vac,dis);
		if (VERBOSE) printf("\n ****[EC3] Bad State, State = [V-%d, CV-%d, D-%d]",vac, clogged_vac,dis);
```

```
		vac=0;dis=0;clogged_vac=0;
	}

	if (vac>2)
	{
		if (VERBOSE) printf("\nP = [%.3lf,%.3lf], State = [V-%d, CV-%d, D-%d]",stats[0],stats[1],vac, clogged_vac,dis);
		if (VERBOSE) printf("\n ****[EC4] Bad State, State = [V-%d, CV-%d, D-%d]",vac, clogged_vac,dis);
		vac=2;
	}

	if (dis>2)
	{
		if (VERBOSE) printf("\nP = [%.3lf,%.3lf], State = [V-%d, CV-%d, D-%d]",stats[0],stats[1],vac, clogged_vac,dis);
		if (VERBOSE) printf("\n ****[EC4] Bad State, State = [V-%d, CV-%d, D-%d]",vac, clogged_vac,dis);
		dis=2;
	}

	if (clogged_vac>2)
	{
		if (VERBOSE) printf("\nP = [%.3lf,%.3lf], State = [V-%d, CV-%d, D-%d]",stats[0],stats[1],vac, clogged_vac,dis);
		if (VERBOSE) printf("\n ****[EC4] Bad State, State = [V-%d, CV-%d, D-%d]",vac, clogged_vac,dis);
		clogged_vac=2;

	}

}

void RO_CLASSIFY(void)
{
	double change,*mean_sigma;
	double Pact[BUFFER], Qact[BUFFER], Pcompare[BUFFER];  // ACTUAL P,Q arrays (UNBUFFERED)
	double *deltaQ_event, *deltaP_event, *deltaP_window, *deltaQ_window;
	double SS_THRESHOLD = .25;
	int i,j;
	int LPi, HPi;  // Pre-event LP or HP pumps running
	int CGI; // This is the global event index
	double gain[2];

	// LOAD THE ACTUAL POWER ARRAYS
	for (i=0;i<BUFFER;i++)
	{
		Pact[i]=P[i+FIR_SIZE];
		Pcompare[i]=P[i+FIR_SIZE];
		Qact[i]=Q[i+FIR_SIZE];
	}

	// Signal that the system is no longer steady state.
	stable=0;

	// If we were in a P/Q mismatch, set it to false so we stop looking for it.
	MISMATCH=FALSE;
```

```
// Record the average power and std dev over the most recent 3 seconds of data
mean_sigma = check_stable_power(Qact,3);

// for each event in the current window
for (i=0;i<local_det;i++)
{
	//if(VERBOSE) printf("\n\nTransient Detect @ [%d]", loc_index[i]+window*ADVANCE);
	////////////////////////////////////////////////////////////////
	// IF WE'RE TOO CLOSE TO THE FRONT OF THE WINDOW, WE WON'T HAVE
	// DELTAP INFO, SO WE WAIT UNTIL NEXT WINDOW.
	////////////////////////////////////////////////////////////////

	if (((loc_index[i]-100)<0) || ((loc_index[i]+GETEVENTS_SKIP+60)>=BUFFER))
	{
		//if(VERBOSE) printf(" - Saving until next window.");
		break;
	}
	else

		////////////////////////////
		//           BEGIN CLASSIFY
		////////////////////////////
	{
		// Now that we've determined the event is classifiable based on position
		// in window.  We assume that it will be classified in this function and
		// mark it as such.

		for (j=0;j<MAX_EVENTS;j++)
		{
			if (loc_index[i]+window*ADVANCE==event_index[j])
			{
				event_class_status[j]=TRUE;
				CGI=j;
			}
		}
	}


	if (VERBOSE) printf("\nPre-event  state: LP [%d] HP [%d]",LP,HP);


	LPi=LP;
	HPi=HP;



	deltaP_event = mean_compare(Pact,loc_index[i],5,30,-5,-30);
	deltaQ_event = mean_compare(Qact,loc_index[i],5,30,-5,-30);
	deltaP_window = mean_compare(Pact,loc_index[i],GETEVENTS_SKIP,GETEVENTS_SKIP+60,-2,-60);
	deltaQ_window = mean_compare(Qact,loc_index[i],GETEVENTS_SKIP,GETEVENTS_SKIP+60,-2,-60);
```

```
change = deltaQ_event[0];

if(VERBOSE) printf("\n\nClassifying event at <%d>", loc_index[i]+window*ADVANCE);
if(VERBOSE) printf(" - DeltaQ_event =[%.3lf]", change);

////////////////////////////
//  CLASSIFY THE ON EVENT   //
////////////////////////////

if (change>=.5)
{  // START CLASSIFY ON

      ////////////////////////////////////////////////////////////////
      // WE ASSUME THAT ONLY AN LP PUMP CAN START IF NO PUMPS ARE RUNNING
      // WE VERIFY THE DELTA IS IN A REASONABLE RANGE - BUT LP ON IS ASSUMED.
      // Note: The 'Phantom' start is not captured by this code.
      ////////////////////////////////////////////////////////////////

      if (VERBOSE) printf(" -> ON ");
      gain[0]=transient_classifier(Pcompare,loc_index[i],3)[2];
      gain[1]=transient_classifier(Pcompare,loc_index[i],4)[2];
      if (VERBOSE) printf("\n *** -> MATCH (LP,HP) = %.3lf,%.3lf",gain[0], gain[1]);


      if ((LP==0) && (HP==0))
      {
            if (gain[0]>0.5)
            {
                  if(VERBOSE) fprintf(stdout,"\n%d - Detected LP Start (template
match=%.3lf)",loc_index[i]+window*ADVANCE,gain[0]);
                  LP++;
                  Class[CGI]="LP Start";
                  Class_ID[CGI]=1;create_evt_file(Pact,Qact,CGI,loc_index[i]);
                  LP_STEADY=FALSE;
            }

            else if ((gain[1]>0) && (change>10))
            {
                  if (VERBOSE) printf("\n%d(?) - Probable HP pump start (template match [%.3lf] and high power
change)",loc_index[i]+window*ADVANCE,gain[1]);
                  Class[CGI]="HP Start (Possible Phantom)";
                  Class_ID[CGI]=2;create_evt_file(Pact,Qact,CGI,loc_index[i]);
                  HP++;


            }

            else if ( (change>10))
            {
```

```
                                if (VERBOSE) printf("\n%d(?) - Probable HP pump start (template match [%.3lf] and high power
change)",loc_index[i]+window*ADVANCE,gain[1]);
                                Class[CGI]="HP Start (Possible Phantom)";
                                Class_ID[CGI]=2;create_evt_file(Pact,Qact,CGI,loc_index[i]);
                                HP++;
                        }

                        else
                        {
                                if (VERBOSE) printf("\n%d(?) - Probable LP pump start (no good template
match).",loc_index[i]+window*ADVANCE);
                                Class[CGI]="LP Start (no good template match)";
                                Class_ID[CGI]=1;create_evt_file(Pact,Qact,CGI,loc_index[i]);
                                LP++;
                                LP_STEADY=FALSE;

                        }

                }

                ////////////////////////////////////////////////////////////////////////////
                // IF AN LP PUMP WAS RUNNING AND ANOTHER ON EVENT OCCURRED, IT HAS TO BE AN HP ON
                // WE VERIFY THE DELTA IS IN A REASONABLE RANGE - BUT HP ON IS ASSUMED.
                ////////////////////////////////////////////////////////////////////////////

                else if ((LP==1) && (HP==0))
                {
                        if ((change>10) && (gain[1]>0))
                        {
                                if(VERBOSE) fprintf(stdout,"\n%d - Detected HP Start (template match and power
change)",loc_index[i]+window*ADVANCE);
                                HP++;
                                Class[CGI]="HP Start (well shaped)";
                                Class_ID[CGI]=2;create_evt_file(Pact,Qact,CGI,loc_index[i]);
                        }
                        else
                        {
                                if (VERBOSE) printf("\n%d(?) -  Probable HP pump start but match is
weak.",loc_index[i]+window*ADVANCE);
                                Class[CGI]="HP Start (poorly shaped)";
                                Class_ID[CGI]=2;create_evt_file(Pact,Qact,CGI,loc_index[i]);
                                HP++;
                        }
                }

                ////////////////////////////////////////////////////////////
                // WE HAVE AN ON EVENT WITH AT LEAST 2 PUMPS ALREADY ON
                // SO EITHER THE SECOND HP PUMP STARTED, OR THE PRE-EVENT STATE
                // INFORMATION WAS WRONG, OR THE BYPASS VALVE WAS ADJUSTED.
                ////////////////////////////////////////////////////////////
```

```
                else  // multiple pumps already on and another started
                {
                        if ((LP==1) && (HP==2))
                        {

                                if (gain[1]>0.0)
                                {
                                        Class[CGI]="HP Start (well shaped)";
                                        Class_ID[CGI]=2;create_evt_file(Pact,Qact,CGI,loc_index[i]);
                                        if (VERBOSE) printf("\n%d(?) -  Power up but system indicates all pumps already
on.",loc_index[i]+window*ADVANCE);
                                        if (VERBOSE) printf("\n%d(?) -  Classify as HP Start based on high
sigma.",loc_index[i]+window*ADVANCE);

                                }

                                else
                                {
                                        Class[CGI]="Possible V-7 adjustment(?)";
                                        Class_ID[CGI]=6;create_evt_file(Pact,Qact,CGI,loc_index[i]);
                                        if (VERBOSE) printf("\n%d(?) -  Power up but system indicates all pumps already
on.",loc_index[i]+window*ADVANCE);
                                        if (VERBOSE) printf("\n%d(?) -  Classify as V-7 adjustment based on low
sigma.",loc_index[i]+window*ADVANCE);
                                }
                        }

                        // THIS IS THE 1LP / 1HP CONDITION

                        else if ((change>10) && (gain[1]>0))
                        {
                                if(VERBOSE) fprintf(stdout,"\n%d - Detected 2nd HP Start (Template match & Rel Power
Change)",loc_index[i]+window*ADVANCE);
                                HP++;
                                Class[CGI]="2nd HP Start";
                                Class_ID[CGI]=2;create_evt_file(Pact,Qact,CGI,loc_index[i]);
                        }
                        else
                        {
                                if(VERBOSE) fprintf(stdout,"\n%d - Detected 2nd HP Start (Init
State)",loc_index[i]+window*ADVANCE);
                                HP++;
                                Class[CGI]="2nd HP Start (? Power OOB)";
                                Class_ID[CGI]=2;create_evt_file(Pact,Qact,CGI,loc_index[i]);
                        }

                }

        }       // end of ON event analysis

        ////////////////////////////
```

```
//  CLASSIFY THE OFF EVENT  //
//////////////////////////////

else if (change < -0.5)
{
       if (VERBOSE) printf(" -> OFF ");

       ////////////////////////////////////////////////////
       // IF IT DID, CLASSIFY THE EVENT BASED ON WHAT PUMPS WERE
       // RUNNING PRIOR TO THE EVENT.  IF P=0, THEN THOSE PUMPS TURNED OFF.
       ////////////////////////////////////////////////////

       if (deltaQ_event[2]<.1)
       {
              if (VERBOSE) printf("\nAll pumps turned off.");
              Class[CGI]="All Off (State & P=0)";
              Class_ID[CGI]=5;create_evt_file(Pact,Qact,CGI,loc_index[i]);
              HP=0;
              LP=0;
       }

       ////////////////////////////////////////////////////
       // IF POWER WENT DOWN BUT NOT TO ZERO, CLASSIFY THE EVENT BASED ON WHAT PUMPS WERE
       // RUNNING PRIOR TO THE EVENT ASSUMING THAT THE LP PUMP WILL NOT
       // TURN OFF WITH THE HP PUMPS RUNNING.
       ////////////////////////////////////////////////////

       else
       {
              // Need to classify off based on pre-state
              if ((LP==1) && (HP==0))
              {
                     LP--;
                     if (VERBOSE) printf("\nGuessing LP off (State)");
                     Class[CGI]="LP off (State).";
                     Class_ID[CGI]=3;create_evt_file(Pact,Qact,CGI,loc_index[i]);
              }
              else if ((LP==1) && (HP>0))
              {
                     HP--;
                     if (VERBOSE) printf("\nGuessing HP turned off (State)");
                     Class[CGI]="HP Pump off (State)";
                     Class_ID[CGI]=4;create_evt_file(Pact,Qact,CGI,loc_index[i]);
              }
              else  // they were all zero and now off event with a +power
              {

                     if (VERBOSE) printf("\nDetected OFF but no pumps were running(?)");
                     Class[CGI]="Off with no running pumps (?)";
                     Class_ID[CGI]=9;create_evt_file(Pact,Qact,CGI,loc_index[i]);
              }
```

```
                }


        }       // END OFF CLASSIFY


        ////////////////////////////////////////////////////////////////////////////
        // IF THE CHANGE IN Q WAS < 0.5 KVAR, IT IS NOT SUFFICIENT TO MAKE A STRONG GUESS.
        // THE EVENT IS EITHER A CYCLING LOAD OR NOISE.
        ////////////////////////////////////////////////////////////////////////////

        else
        {

                if (mean_sigma[1]>GETEVENTS_THRESHOLD)
                {
                        Class[CGI]="Cycling Load Transient";
                        Class_ID[CGI]=8;create_evt_file(Pact,Qact,CGI,loc_index[i]);
                        if (VERBOSE) printf("\n%d(?) - No change in power - cycled load (Based on High
Sigma).",loc_index[i]+window*ADVANCE);

                }
                else
                {
                        GETEVENTS_THRESHOLD = GETEVENTS_THRESHOLD*1.1;
                        SS_THRESHOLD = SS_THRESHOLD* 1.1;
                        Class[CGI]="Noise.";Class_ID[CGI]=9;
                        create_evt_file(Pact,Qact,CGI,loc_index[i]);
                        if (VERBOSE) printf("\n%d(?) - No delta P - Noise (bumping detect threshold to
[%.2lf]).",loc_index[i]+window*ADVANCE,GETEVENTS_THRESHOLD);

                }
        }

        if (VERBOSE) printf("\nPost-event state: LP [%d] HP [%d]",LP,HP);
    }

}


void RO_IC(void)
// RO_IC() is a system state verification check (i.e. the # and types of pumps which are
// currently thought to be running) .  The stability of the system is recorded
// through a global variable ('stable').  If the system is not stable, RO_IC() determine if
// it has returned to stability.  If it has returned to stability, RO_IC() performs a number
// of checks to see if the classifier state is consistent with current power levels.
// If the state is not consistent, RO_IC() attempts to fix the state.

{
    double *Qstats,*Pstats;
```

```
	int i;
	double Pact[BUFFER], Qact[BUFFER];  // ACTUAL P,Q arrays (UNBUFFERED)
	double SS_THRESHOLD = .1;


	for (i=0;i<BUFFER;i++)
	{
		Pact[i]=P[i+FIR_SIZE];
		Qact[i]=Q[i+FIR_SIZE];
	}

	Qstats = check_stable_power(Qact,3);
	Pstats = check_stable_power(Pact,3);

	////////////////////////////////////////////////////////////////////////
	// IF THE Q IS RELATIVELY CONSTANT OVER THE LAST 3 SECONDS, WE ASSUME IT HAS
	// RETURNED TO STEADY STATE.
	////////////////////////////////////////////////////////////////////////

	if (Qstats[1]<SS_THRESHOLD)
	{
		// if (VERBOSE)printf("[%d] System is now steady state.",window*ADVANCE + BUFFER);
		if ((LP==1) && (HP==0) && (Pstats[1]<SS_THRESHOLD) && (!LP_STEADY)  )

			////////////////////////////////////////////////////////////////////////
			//  IF WE REACH STEADY STATE AND ONLY AN LP PUMP IS RUNNING, WE INFORM THE
			//  OPERATOR THAT THE SYSTEM IS STEADY STATE (SO HE CAN START HP PUMP).
			////////////////////////////////////////////////////////////////////////
		{
			if (VERBOSE)printf("\n - Operator Note: LP pump is running, System power has been steady over last 3
seconds.");
			event_index[event_count]= window*ADVANCE+BUFFER;  // load the global event index
			Class[event_count]="Power Stable following LP start.";
			Class_ID[event_count]=007;
			event_class_status[event_count]=TRUE;
			create_evt_file(Pact, Qact, event_count,window*ADVANCE+BUFFER);
			event_count++; // increment the number of global events
			LP_STEADY=TRUE;
		}

		if ((LP==1) && (HP>0))
			////////////////////////////////////////////////////////////////////////
			// IF WE REACH STEADY STATE AND THE LP PUMP AND AT LEAST ONE HP PUMP IS ON,
			// WE CHECK P vs Q.  IF A 10% MISMATCH EXISTS, WE INFORM THE OPERATOR THAT
			// THE BYPASS NEEDS TO BE POSITIONED.
			////////////////////////////////////////////////////////////////////////
		{
			if ((Pstats[0]/Qstats[0])<0.9)
			{
				if (VERBOSE)printf("\n - Operator Note: Bypass valve is likely open or partially open with HP pump
running. (%.2lf)",Pstats[0]/Qstats[0]);
```

```
                        MISMATCH=TRUE;
                }
                else if (MISMATCH)
                {
                        if (VERBOSE)printf("\n - P/Q Mismatch has cleared.  Creating event for last 10 seconds of data.");
                        event_index[event_count]= window*ADVANCE+BUFFER;  // load the global event index
                        Class[event_count]="P/Q Mismatch has cleared.";
                        event_class_status[event_count]=TRUE;
                        Class_ID[event_count]=006;
                        create_evt_file(Pact, Qact,event_count ,window*ADVANCE+BUFFER);
                        event_count++; // increment the number of global events
                        MISMATCH=FALSE;
                }

        }


        if (!stable)

                ////////////////////////////////////////////////////////
                // WE JUST DETECTED THAT WE HAVE RETURNED TO STEADY STATE
                ////////////////////////////////////////////////////////
        {

                //if (VERBOSE)printf("\n[t %.3lf min] Q has been steady @ [%.1lf] KVAR for 3
seconds.",trans_t(window*ADVANCE+BUFFER)/60,Pstats[0]);


                if (Pstats[0]<.1)

                        ////////////////////////////////////////////////////////////////
                        // P~0 and STEADY, so we make sure State info indicates no pumps running.
                        ////////////////////////////////////////////////////////////////
                {
                        //if (VERBOSE) printf("\nStable zero power condition detected.");

                        if ((!LP) && (!HP))
                        {
                                //if (VERBOSE) printf("\nConfirmed the 'ALL OFF' condition.");
                        }
                        else
                        {
                                if (VERBOSE) printf("\n[%d]   [EC1] Reseting pump lineup to 'No pumps running'",window*ADVANCE +
BUFFER);
                                LP=0;
                                HP=0;
                                event_index[event_count]= window*ADVANCE+BUFFER;
                                Class[event_count]="[EC1] All Off (Return to SS)";
                                Class_ID[event_count]=005;
                                //create_evt_file(Pact,Qact,event_count,window*ADVANCE+BUFFER);
                                event_count++;
```

```
                }
        }


if ((Pstats[0]>=.5) && ((LP==0) && (HP==0)))
        {
                if (VERBOSE) printf("\nReseting pump lineup.");
                event_index[event_count]= window*ADVANCE+BUFFER;

                if (Pstats[0]<1.5)
                {
                        LP=1;
                        Class[event_count]="[IC2] LP on (Fixing Bad State)";
                        //create_evt_file(Pact,Qact,event_count,window*ADVANCE + BUFFER);
                        if (VERBOSE) printf("\n      Guessing a LP pump is on based on current power level.");

                }
                else if (Pstats[0]<10)
                {
                        LP=1;
                        HP=1;
                        Class[event_count]="[IC2] LP On and HP On (Fixing Bad State)";
                        //create_evt_file(Pact,Qact,event_count,window*ADVANCE+BUFFER);
                        if (VERBOSE) printf("\n      Guessing a LP and HP pump are on based on current power level.");

                }
                else // High power condition
                {
                        LP=1;
                        HP=2;
                        Class[event_count]="[IC3] All pumps on (Fixing Bad State)";
                        //create_evt_file(Pact,Qact,event_count,window*ADVANCE+BUFFER);
                        if (VERBOSE) printf("\n      Guessing all are on based on current power level.");

                }
                event_count++;
        }
}

////////////////////////////////////////////////////////////////////
// WE WERE ALREADY STABLE, AND WE JUST DETECTED THAT WE ARE STILL STABLE
// SO WE CAN DOUBLE CHECK STATE INFORMATION OR PROVIDE THE OPERATOR WITH
// INFORMATION ABOUT THE CURRENT STEADY STATE WINDOWS
////////////////////////////////////////////////////////////////////

else
{

        ////////////////////////////////////////////////////////////////////
        // Check to see if power is stable @ zero and state info indicates
        // a running pump (which would be inaccurate).  If so, reset the
```

```
                    // pumps to zero.
                    ////////////////////////////////////////////////////////////////////

                    if ((Pstats[0]<.1) && ((LP==1) || (HP>0)))
                    {
                          if (VERBOSE) printf("\n[%d]  [IC1] Stable zero power condition detected but system thinks a pump is
running.",window*ADVANCE + BUFFER);
                          if (VERBOSE) printf("\n      Reseting pump lineup to 'No pumps running'");
                          LP=0;
                          HP=0;
                          event_index[event_count]= window*ADVANCE+BUFFER;
                          Class[event_count]="[IC1] All Off (Fixing Bad State)";
                          //create_evt_file(Pact,Qact,event_count,window*ADVANCE+BUFFER);
                          event_count++;
                    }

                    ////////////////////////////////////////////////////////////////////
                    // Check to see if power is stable '+' and state info indicates
                    // nothing is running (which would be inaccurate).  If so, make a
                    // best guess regarding running pumps.
                    ////////////////////////////////////////////////////////////////////


                    if ((Pstats[0]>=.5) && ((LP==0) && (HP==0)))
                    {
                          if (VERBOSE) printf("\n[%d]  [IC2] Stable (+) power condition detected but system shows no running
pumps.",window*ADVANCE + BUFFER);
                          if (VERBOSE) printf("\nReseting pump lineup to 'No pumps running'");
                          event_index[event_count]= window*ADVANCE+BUFFER;

                          if (Pstats[0]<1.5)
                          {
                                LP=1;
                                Class[event_count]="[IC2] LP on (Fixing Bad State)";
                                //create_evt_file(Pact,Qact,event_count,window*ADVANCE + BUFFER);
                                if (VERBOSE) printf("\n      Guessing a LP pump is on based on current power level.");

                          }
                          else if (Pstats[0]<10)
                          {
                                LP=1;
                                HP=1;
                                Class[event_count]="[IC2] LP On and HP On (Fixing Bad State)";
                                //create_evt_file(Pact,Qact,event_count,window*ADVANCE+BUFFER);
                                if (VERBOSE) printf("\n      Guessing a LP and HP pump are on based on current power level.");

                          }
                          else // High power condition
                          {
                                LP=1;
                                HP=2;
```

```
                        Class[event_count]="[IC3] All pumps on (Fixing Bad State)";
                        //create_evt_file(Pact,Qact,event_count,window*ADVANCE+BUFFER);
                        if (VERBOSE) printf("\n      Guessing all are on based on current power level.");

                    }
                    event_count++;
                }



        }


        // SET STABLE TO 1 TO INDICATE THAT WE ARE NOW STEADY STATE.
        stable = 1;
    }


    ////////////////////////////////////////////////////////////////////////
    // OTHERWISE, WE WERE NOT STEADY STATE AND WE'RE STILL NOT STEADY STATE.
    ////////////////////////////////////////////////////////////////////////

    else
    {
        // if (VERBOSE) printf("System power is NOT steady state.");
        stable=0;
    }

}
void slice(void)
{
    double Pact[BUFFER], Qact[BUFFER];  // ACTUAL P,Q arrays (UNBUFFERED)
    //double *deltaQ_event, *deltaP_event, *deltaP_window, *deltaQ_window;
    int i,j;
    int CGI; // This is the global event index
    //double gain[2];



    // LOAD THE ACTUAL POWER ARRAYS
    for (i=0;i<BUFFER;i++)
    {
        Pact[i]=P[i+FIR_SIZE];
        Qact[i]=Q[i+FIR_SIZE];
    }

    // for each event in the current window
    for (i=0;i<local_det;i++)
    {

        //if(VERBOSE) printf("\n\nTransient Detect @ [%d]", loc_index[i]+window*ADVANCE);
```

```
            ////////////////////////////////////////////////////////////////
            // IF WE'RE TOO CLOSE TO THE FRONT OF THE WINDOW, WE WON'T HAVE
            // DELTAP INFO, SO WE WAIT UNTIL NEXT WINDOW.
            ////////////////////////////////////////////////////////////////

            //     Check if the local events are within a reasonable range of the BUFFER boundaries
            //     For every event in the local event table, check to see if it has been EVC'd.

            if (((loc_index[i]-60)<0) || ((loc_index[i]+GETEVENTS_SKIP+60)>=BUFFER))
            {
                  //if(VERBOSE) printf(" - Saving until next window.");

            }

            else

                  //  If the event is within boundaries and not EVC'd, create an event file.

                  ////////////////////////////////
                  //            BEGIN CLASSIFY
                  ////////////////////////////////
            {
                  // Now that we've determined the event is classifiable based on position
                  // in window.  We assume that it will be classified in this function and
                  // mark it as such.

                  for (j=0;j<MAX_EVENTS;j++)
                  {
                        if (loc_index[i]+window*ADVANCE==event_index[j])
                        {
                              event_class_status[j]=TRUE;
                              CGI=j;
                              Class[CGI]="Unclassified.";
                              Class_ID[CGI]=0;
                        }
                  }

                  if(VERBOSE) printf("\n\nTransient Detect @ [%d]", loc_index[i]+window*ADVANCE);
                  create_evt_file(Pact,Qact,CGI,loc_index[i]);
            }

      }
}


void stream(void)
{
      double Pact[BUFFER], Qact[BUFFER];  // ACTUAL P,Q arrays (UNBUFFERED)

      int i;
```

```
	// LOAD THE ACTUAL POWER ARRAYS
	for (i=0;i<BUFFER;i++)
	{
		Pact[i]=P[i+FIR_SIZE];
		Qact[i]=Q[i+FIR_SIZE];
	}
	event_index[event_count]= window*ADVANCE+BUFFER;  // load the global event index
	Class[event_count]="Stream.";
	Class_ID[event_count]=000;
	create_evt_file(Pact,Qact,event_count,1);
	event_count++;
	ST_WIN_Index++;
}



double *check_stable_power(double *P,int t)
// This routine takes an actual power array and checks to see if the most recent t seconds of data indicate constant power.
// It returns a 2 elements array. The first element is average, second is StdDev.
{
	double average=0, variance=0, SD=0;
	int i;
	double *R;
	R=(double*)calloc(2,sizeof(double));

	for (i=1;i<=120*t;i++)
		average+=P[BUFFER-i];
	average = average/(120*t);
	R[0]=average;

	for (i=1;i<=120*t;i++)
		variance += (P[BUFFER-i]-average)*(P[BUFFER-i]-average);
	variance = variance/(120*t);

	SD=sqrt(variance);
	R[1]=SD;

	//if (VERBOSE) printf("\nLast %d second(s) - average [%.3lf], StdDev [%.3lf]",t,average,SD);
	return (R);
}

void create_evt_file(double *Pact,double *Qact, int index,  int local_position)
{
	// create_evt_file generates an event file when called.  The name of the file must be
	// specified from the command line (for example   ./ginzu f snapshot.txt will force all
	// event files to be named "snapshot-xxxx-yyyy.evt").

	// .EVT file naming convention (for "snapshot-xxxx-yyyy.evt")
	//                         - xxx ==> 3 digit # corresponding to the # of the event (starting at 000)
```

```
//                          - yyy ==> 3 digit # corresponding to the classification of the event

FILE *evt_fp;
int i=0,b=BUFFER;
char evt_filename[80]="";
char str2[80];
char str3[80];
int j;
time_t curtime;
struct tm *loctime;

curtime = time(NULL);
loctime = localtime(&curtime);

while ((filename[i]!='\0') &&  (filename[i]!='.'))
{
       str2[i]=filename[i];
       i++;
}
str2[i]='\0';
j=(int)strlen(str2);

while (j>=0)
{
       j--;
       if ((str2[j]=='/') || (str2[j]=='\\'))
       {
              break;
       }
}

i=0;

while (str2[j+1+i]!='\0')
{
       str3[i]=str2[j+1+i];
       i++;
}

str3[i]='\0';

strcat(evt_filename,str3);
strcat(evt_filename,"-");
strcat(evt_filename,int2str(index,10));
strcat(evt_filename,"-");
strcat(evt_filename,int2str(Class_ID[index],10));
strcat(evt_filename,".evt");

signal_ignore = 1;

if((evt_fp=fopen(evt_filename,"w+")) == NULL)
```

```
	{
		fprintf(stderr, "\nError creating %s file.\n",evt_filename);
		exit(1);
	}
	if (VERBOSE) printf("\nWriting %s",evt_filename);

	if (REAL_TIMESTAMP)
		fprintf(evt_fp, "%s",asctime(loctime));
	else
		fprintf(evt_fp,
"%s:%.2d:%.2d\n",filename_to_date(),index_to_time(event_index[index])[0],index_to_time(event_index[index])[1]);

	fprintf(evt_fp, "%s",filename);
	fprintf(evt_fp, "\n%d",event_index[index]);
	fprintf(evt_fp, "\n%d",local_position);
	fprintf(evt_fp, "\n%s",Class[index]);
	fprintf(evt_fp, "\n%s",int2str(Class_ID[index],10));
	fprintf(evt_fp, "\n%d",b);

	for (i=0;i<BUFFER;i++)
		fprintf(evt_fp, "\n%lf",Pact[i]);

	for (i=0;i<BUFFER;i++)
		fprintf(evt_fp, "\n%lf",Qact[i]);

	EVC[index]++;

	fclose(evt_fp);

	signal_ignore = 0;

	if (signal_received)
		exit(2);
}


char *int2str(int val, int base)
// int2str is itoa() which is non-standard C.  The function takes an integer value
// which must be < 10000 and converts it to a char string.   Credit is given to Stuart Lowe
// (http://www.jb.man.ac.uk/~slowe/cpp/itoa.html) for the code.
{

	static char buf[32] = {0};
	int ival=val;
	int i = 5;
	if ((val==0) || (val>9999))
		return "0000";
	for(; val && i ; --i, val /= base)
		buf[i] = "0123456789abcdef"[val % base];

	if (ival<10)
```

```
	{
		// put 3 zeros at the front
		buf[i]='0';
		buf[i-1]='0';
		buf[i-2]='0';
		i=i-3;
	}

	else if ((ival>9) && (ival<100))
	{
		// put 2 zero at the front
		buf[i]='0';
		buf[i-1]='0';
		i=i-2;
	}

	else if ((ival>99) && (ival<1000))
	{
		// put 1 zero at the front
		buf[i]='0';
		i=i-1;
	}
	return &buf[i+1];
}



//////////////////////////////////////////
//     Matrix and vector recipes        //
//  Credit to Numerical Recipes in C    //
//      Second Edition, 1992            //
//////////////////////////////////////////

//  The following are either taken directly from the above reference
//  or simply coded (transpose, matrix multiple, show_matrix) for use
//     by the linear algebra functions needed in the classification.
//     Please use the above reference for additional infomation.



void show_matrix(double **A, long row, long col)
{
	int i,j;
	if(VERBOSE) printf("\n --- Start ---");
	for (i=1;i<=row;i++)
	{
		if(VERBOSE) printf("\n[");
		for (j=1;j<=col;j++)
			if(VERBOSE) printf(" %.2lf ",A[i][j]);
		if(VERBOSE) printf("]");
	}
```

```
        if(VERBOSE) printf("\n --- stop ---");
}

double **dmatrix(long nrl,long nrh, long ncl, long nch)
{
        long i, nrow=nrh-nrl+1,ncol=nch-ncl+1;
        double **m;

        m=(double**) malloc((size_t)((nrow+NR_END)*sizeof(double*)));
        if (!m) nrerror("allocation failure 2 in matrix()");
        m += NR_END;
        m -= nrl;

        m[nrl]=(double*) malloc((size_t)((nrow*ncol+NR_END)*sizeof(double)));
        if (!m[nrl]) nrerror("allocation failure 2 in matrix()");
        m[nrl] += NR_END;
        m[nrl] -= ncl;

        for (i=nrl+1;i<=nrh;i++) m[i]=m[i-1]+ncol;

        return m;
}

void nrerror(char error_text[])
{
        fprintf(stderr,"\nNumerical Recipes run-t error...");
        fprintf(stderr,"\n%s",error_text);
        fprintf(stderr,"\n...now exiting to system..");
        exit(1);
}

void ludcmp(double **a, int n, int *indx, float *d)
{
        int i, imax,j,k;
        float big,dum,sum,temp;
        float *vv;

        vv=dvector(1,n);
        *d=1.0;
        for(i=1;i<=n;i++) {
                big =0.0;
                for(j=1;j<=n;j++)
                        if ((temp=fabs(a[i][j])) > big) big=temp;
                if (big ==0.0) nrerror("Singular matrix in routine ludcmp");
                vv[i]=1.0/big;
        }
        for (j=1;j<=n;j++) {
                for (i=1;i<j;i++) {
                        sum=a[i][j];
                        for (k=1;k<i;k++) sum -= a[i][k]*a[k][j];
                        a[i][j]=sum;
```

```
            }
            big=0.0;
            for (i=j;i<=n;i++)
            {
                  sum=a[i][j];
                  for(k=1;k<j;k++)
                        sum -= a[i][k]*a[k][j];
                  a[i][j]=sum;
                  if ((dum=vv[i]*fabs(sum)) >= big)
                  {
                        big=dum;
                        imax=i;
                  }
            }
            if (j!= imax)
            {
                  for (k=1;k<=n;k++)
                  {
                        dum=a[imax][k];
                        a[imax][k]=a[j][k];
                        a[j][k]=dum;
                  }
                  *d=-(*d);
                  vv[imax]=vv[j];
            }
            indx[j]=imax;
            if (a[j][j]==0.0) a[j][j]=TINY;
            if (j!=n)
            {
                  dum=1.0/(a[j][j]);
                  for (i=j+1;i<=n;i++) a[i][j] *= dum;
            }
      }
      free_dvector(vv,1,n);
}

void lubksb(float **a, int n, int *indx,float b[])
{
      int i,ii=0,ip,j;
      float sum;

      for (i=1;i<=n;i++)
      {
            ip=indx[i];
            sum=b[ip];
            b[ip]=b[i];
            if (ii)
                  for (j=ii;j<=i-1;j++) sum -= a[i][j]*b[j];
            else if (sum) ii=i;
            b[i] = sum;
      }
```

```
	for (i=n;i>=1;i--)
	{
		sum=b[i];
		for (j=i+1;j<=n;j++) sum -= a[i][j]*b[j];
		b[i]=sum/a[i][i];
	}
}


double *dvector(long nl, long nh)
{
	double *v;

	v=(double*)malloc((size_t)((nh-nl+1+NR_END)*sizeof(double)));
	if (!v) nrerror("allocation failure in dvector()");
	return (v-nl+NR_END);
}

void free_dvector(double *v,long nl,long nh)
{
	free((FREE_ARG)(v+nl-NR_END));
}


double **invert(double **a,int N)
{
	double **y,d,*col;
	int i,j,*indx;
	y=dmatrix(1,N,1,N);
	col=(double*)calloc(N,sizeof(double));
	indx=(int*)calloc(N,sizeof(int));

	ludcmp(a,N,indx,&d);
	for (j=1;j<=N;j++)
	{
		for (i=1;i<=N;i++) col[i]=0.0;
		col[j]=1.0;
		lubksb(a,N,indx,col);
		for (i=1;i<=N;i++) y[i][j]=col[i];
	}
	return (y);
}

double **mat_mult(double **A, long Arow, long Acol, double **B,long Brow, long Bcol)
{
	int i, j, k;

	double **C;
	C=dmatrix(1,Arow,1,Bcol);

	if (Acol!=Brow)
```

```
	{
		nrerror("allocation failure 3 in matrix() - inner dimensions don't match");
	}

	for( i = 1; i <= Arow; i++ )
		for( j = 1; j <= Bcol; j++ )
			C[i][j]=0.0;

	for( i = 1; i <= Arow; i++ )
		for( j = 1; j <= Bcol; j++ )
			for( k = 1; k <= Acol; k++ )
				C[i][j] += A[i][k] * B[k][j];

	return (C);
}


double **transpose(double **A, long Arow, long Acol)
{

	int i,j;

	double **B;

	B=dmatrix(1,Acol, 1, Arow);
	for (i=1;i<=Acol;i++)
	{
		for (j=1;j<=Arow;j++)
		{
			B[i][j]=A[j][i];
		}
	}
	return (B);
}
```