A Framework for Non-Intrusive Load Monitoring and Diagnostics

by

James Paris

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2006

Author ....
Department of Electrical Engineering and Computer Science
uary 26, 2006

Certified by ....
Steven B. Leeb
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by ....
Robert W. Cox
Candidate
Supervisor

Accepted by ... .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

A Framework for Non-Intrusive Load Monitoring and Diagnostics

by

James Paris

Submitted to the Department of Electrical Engineering and Computer Science
on January 26, 2006, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The widespread use of electrical and electromechanical systems places increasing demands on monitoring and diagnostic techniques. The non-intrusive load monitor (NILM) provides a low-cost, low-maintenance way to perform this monitoring and diagnostics from a centralized location. This work critically evaluates the current state of the NILM hardware and software in order to develop new techniques and a new hardware and software framework in which to better apply the NILM to real-world systems. New diagnostic indicators are developed on the *USCGC SENECA* using an improved hardware and software platform. A database-driven framework with the flexibility to create and implement these and future diagnostic indicators is presented.

Thesis Supervisor: Steven B. Leeb
Title: Professor of Electrical Engineering and Computer Science

Thesis Supervisor: Robert W. Cox
Title: Doctoral Candidate

Acknowledgments

I would like to thank Professor Leeb for his continued guidance and support, and acknowledge and thank Steve Shaw, Rob Cox, Chris Laughman, John Rodriguez, Tom DeNucci, Bill Greene, Jip Mosman, Duncan McKay, and Mariano Alvira for their contributions and assistance. I also extend my appreciation to all those who contributed to the many open-source software packages that made this work possible.

This research was generously supported by The Grainger Foundation, the ONR Electric Ship Research and Development Consortium, and NEMOmetrics Corporation, with additional valuable support from Mr. Ron Koo and Dr. Emmanuel Landsman.

Contents

1 Introduction **19**

1.1 The NILM . . . 19

1.1.1 NILM Theory . . . 21

1.1.2 Power Envelope Generation . . . 22

1.1.3 Steady-State Analysis Approach . . . 25

1.1.4 Transient Analysis Approach . . . 25

1.1.5 NILM system example . . . 27

1.2 Contributions and Organization . . . 29

2 Diagnostic Indicators **31**

2.1 Cycling Systems . . . 34

2.1.1 *SENECA* Data . . . 36

2.1.2 Simulation . . . 41

2.1.3 Analysis . . . 50

2.1.4 Indicator . . . 59

2.1.5 Conclusions . . . 61

2.2 Mechanical Coupling Failure . . . 62

2.2.1 *SENECA* Data . . . 62

2.2.2 Simulation and Analysis . . . 65

2.2.3 Indicator . . . 66

2.2.4 Conclusions . . . 67

3 Data Acquisition **69**

3.1 PCI-1710 card 69

3.2 Custom USB Module 71

3.2.1 Hardware 71

3.2.2 Firmware 73

3.2.3 Driver Implementation 75

3.2.4 Driver Usage 79

4 Support Software **81**

4.1 Preprocessor 81

4.1.1 Consolidated Text Input 82

4.1.2 Build System Improvements 82

4.1.3 Improved Command-Line Options 83

4.2 Data Logging 83

4.2.1 Overview 84

4.2.2 Data Acquisition and Storage 85

4.2.3 Snapshots 86

4.2.4 CD and DVD Burning 87

4.3 Graphical Demonstration 88

4.3.1 Implementation 90

4.3.2 Developed Systems 94

5 NILM Software Framework **97**

5.1 Design 97

5.1.1 Metadata 99

5.1.2 Records 101

5.2 Implementation 102

5.2.1 Database Storage 103

5.2.2 Stream Interface Library 103

5.2.3 Client Programs 104

5.3 Example Modules 105

5.3.1 Insert 106
5.3.2 Extract 106
5.3.3 Dump and Remove 108
5.3.4 Filter 108
5.4 Usage and Applications 110
5.4.1 System Control and Graphical Interfaces 110
5.4.2 Compatibility with Existing Hardware and Software 110
5.4.3 Interfacing with Matlab/Octave 112

6 Conclusions 113
6.1 NILM Applications on the *SENECA* 113
6.2 NILM Framework Improvements 114
6.3 Future Work 115

A Code Listings 117
A.1 Cycling Systems 117
A.1.1 `time-between.pl` 117
A.1.2 `sim.m` 118
A.1.3 `pumpleak.m` 119
A.1.4 `sim_between.m` 120
A.2 USBADC 121
A.2.1 `adc.asm` 121
A.2.2 `ftdi-adc` 132
A.2.3 `convert` 166
A.3 Support Software 168
A.3.1 `menu-system` components 168
A.3.2 `nilmgui` 180
A.4 NILM Software Framework 217
A.4.1 Database 217
A.4.2 Library and Object Interface, `libstream` 219
A.4.3 Client Programs 239

A.4.4 Matlab/Octave Functions 254

B Hardware Design **257**

B.1 USB ADC Board Layout 257

B.2 USB ADC Schematic 258

List of Figures

1-1 Typical system layout for non-intrusive monitoring of several loads. Data acquisition and processing occurs at a central location. 20

1-2 Non-intrusive monitoring of multiple loads at a distribution panel. The installation consists of a single current transducer to measure current, and two leads to measure voltage. 21

1-3 Block diagram of typical NILM data processing stages. Fault diagnostics often uses results from many of the earlier stages. 22

1-4 Raw AC voltage and current measurements taken during a motor startup transient. 24

1-5 Result of applying the preprocessor to the data in Figure 1-4. These spectral envelopes provide a much clearer view of the electrical power usage. 24

1-6 Plot of measured ΔP and ΔQ values for various loads, used in steady-state analysis. From [13]. 25

1-7 Plot of the power envelopes generated by turning an AC fan on and off, showing two characteristic transient events. 26

1-8 Startup transient of a single device, consisting of three smaller transients with variable spacing. From [11]. 27

1-9 Photograph of a single-board NILM computer system, left, monitoring and identifying various satellite loads, right. 28

1-10 Graphical user interface for the satellite load monitoring system. The power waveforms are shown in the top half of the window. Identified transients are listed at the bottom as they occur. 28

2-1 System identification results for a ventilation fan. The numeric tags on the bottom contain estimated parameter information. The thicker line in the plot is the result of a simulation with the estimated parameters. From [13]. 33

2-2 Variable speed drive exhibiting large, slow oscillations after approximately 100 seconds. From [10]. 34

2-3 Diagram of a typical pneumatic cycling system. Solid lines are air, and dotted lines are electrical connections. 35

2-4 Operation of a cycling air-supply system. When pressure reaches H, the compressor turns off and the system discharges as loads are used. When pressure falls to L, the compressor runs and the system charges again. 35

2-5 Photograph of the twin pumps used to recharge the vacuum reservoir on the *USCGC SENECA*. The reservoir is in the rear. 36

2-6 Power usage of the *SENECA* vacuum pump during a single charging cycle. The transient is clear and easy to match. 38

2-7 Power usage of the *SENECA* vacuum pump during multiple charge/discharge cycles over the course of an hour. 38

2-8 Histogram showing the distribution of the time between pump runs over a two-week period during which no leaks were known to exist. From [7]. 39

2-9 Histogram showing the distribution of the time between pump runs over a two-week period during which the system had a large leak in a faulty check valve. From [7]. 40

2-10 Three histograms corresponding to three different manually-induced leaks in the *SENECA*. The peak of the histogram shifts to the left as the leak rate is increased. 40

2-11 Cumulative distribution function (CDF) of the random variable T representing the time between flushes in the cycling system simulation. . 45

2-12 Histogram of the pump cycling data generated by the simulator with no leak. The parameters for this simulation are as shown in Table 2.2 where $R_{\text{leak}} = 0$ in. Hg / hr. 48

2-13 Histogram of the pump cycling data generated by the simulator with a small leak. The parameters for this simulation are as shown in Table 2.2 where $R_{\text{leak}} = 4.5$ in. Hg / hr. 48

2-14 Histogram of the pump cycling data generated by the simulator with a medium-sized leak. The parameters for this simulation are as shown in Table 2.2 where $R_{\text{leak}} = 7$ in. Hg / hr. 49

2-15 Histogram of the pump cycling data generated by the simulator with a large leak. The parameters for this simulation are as shown in Table 2.2 where $R_{\text{leak}} = 25$ in. Hg / hr. 49

2-16 Plot of the Erlang distribution for $\eta = 4$ and varying λ. This distribution, from Equation 2.34, corresponds to the sum of multiple Poisson arrival times, and models the baseline distribution of crew usage with no leak. 51

2-17 Plot of the Erlang cumulative distribution function from Equation 2.36 for $\lambda = 30$/hr and varying η. 53

2-18 Plot of the CDF for the *SENECA* vacuum pump in the presence of both crew usage and leaks, for $\lambda = 10$/hr and $\tau = 5$ min. The η for the pump system is 4. When t is a multiple of τ, the pump CDF "jumps" to follow the next Erlang CDF. 56

2-19 Calculated pump PDF plotted against simulated data. The system and leak parameters are the same as in Figure 2-12. 57

2-20 Calculated pump PDF plotted against simulated data. The system and leak parameters are the same as in Figure 2-13. 57

2-21 Calculated pump PDF plotted against simulated data. The system and leak parameters are the same as in Figure 2-14. 58

2-22 Calculated pump PDF plotted against simulated data. The system and leak parameters are the same as in Figure 2-15. 58

2-23 Plot of the the two leak cases from Figure 2-10, with the peak values indicated at $\tau_{50} = 8$ and $\tau_{30} = 12$. 60
2-24 Data recorded on the *SENECA* during working and evening hours, when crew usage was high. The peak corresponding to the leak still shows up in nearly the same place despite the additional usage, as expected. The τ_{50} line from Figure 2-23 is shown for comparison. . . . 61
2-25 Diagram of the *SENECA* auxiliary seawater (ASW) system including the two primary pumps. From [6]. 62
2-26 Photograph of one ASW motor and pump. A flexible rubber coupling connects the two in the middle. From [6]. 63
2-27 Power usage measured at the motor during the startup of one of the ASW pumps. A characteristic high-frequency oscillation is visible in region R near the start of the transient. 63
2-28 Plots of the spectral content of the high-frequency oscillation in five ASW pump starts. The magnitude increases around 44 Hz as the coupling progressively fails. From [7]. 64
2-29 Photographs of the degrading ASW pump coupling during failure. The letters correspond to the pump starts shown in the spectral plots in Figure 2-28. From [7]. 65
2-30 Model of the ASW motor, coupling, and pump system, as presented in [7]. 66

3-1 Photograph of the Advantech PCI-1710 data acquisition card, used to record voltage and current data into a desktop PC. 70
3-2 Photograph of the custom USB data acquisition module developed for use with the NILM. The hardware was manufactured and provided by Professor Steven R. Shaw of the Montana State University. 72
3-3 Data format used by the USBADC firmware to transfer data from the ADC to the PC. 75

4-1 Screenshot of the main `menu-system` interface. 84

4-2 Sequence of screenshots demonstrating control of the `run-prep.pl` data collection process through the interactive menus. The prompts are designed to easily guide inexperienced users who may need to control systems installed in the field. 86

4-3 Sequence of screenshots demonstrating the automated burning of snapshot data to CD through the `burn-cd` script. 88

4-4 Main window of the `nilmgui` graphical demonstration program. . . . 89

4-5 Screenshot of the "Exemplars" dialog in the `nilmgui` software. Five transients have been recorded, and guided training of a sixth is in progress. 90

4-6 Photograph of a demonstration platform built around the `nilmgui` software. Here, the connected power strip allows the system to monitor a lamp, a fan, and a heat gun. 94

5-1 Conceptual layout of a stream in the NILM software framework. . . . 98

5-2 Example of stream slicing in the NILM framework. Streams 1, 2, and 3 are recorded data from the same source. A request for data between a and b is assembled and returned as Stream A, while a request for a stream between c and d cannot be satisfied due to missing data. . . . 99

5-3 Layout of metadata within a stream in the NILM software framework. 100

5-4 Examples of potential data record formats in a NILM stream. 101

5-5 Diagram of the software layering in the NILM framework. 102

5-6 Example of retrieving, manipulating, and storing data in the NILM database from within Matlab/Octave. 112

List of Tables

2.1 Parameters for the simulation model of the *SENECA* cycling system. 42

2.2 Simulation parameters used in Figures 2-12 through 2-15. 47

3.1 Format of the command sent to the USB ADC hardware to configure and start conversion. 74

3.2 List of the command-line options supported by the `ftdi-adc` program. 79

4.1 List of the command-line options available to the `prep` program. . . . 83

4.2 List of the possible event identifiers in the comments inserted into recorded data streams by `run-prep.pl`. 85

5.1 List of the command-line options common to all C++ client modules. 105

5.2 List of specific command-line options for the `insert` module. 107

5.3 List of specific command-line options for the `extract` module. 107

5.4 List of command-line options for the `filter.pl` script. 109

Chapter 1

Introduction

Electric systems can be found everywhere. Systems ranging from satellites to home appliances rely on actuators, controls, and power sources that are often electrical or electromechanical. As such systems grow more complicated, the task of monitoring and detecting problems becomes difficult. One key to managing this complexity is the observation that physically different events can be associated with different power usage patterns, and that by recording and analyzing these power signatures, inferences can be made about the state of the system. The non-intrusive load monitoring (NILM) system provides a straightforward and inexpensive means for this measurement and analysis.

This work examines the current state of the NILM system as presented in [11, 13], and extends the non-intrusive approach with respect to load-monitoring techniques, load identification and scheduling, and system diagnostics. Real-world systems and applications are a particular focus, as is the definition and implementation of a generalized software and hardware framework to facilitate practical usage.

1.1 The NILM

Fundamentally, the NILM disaggregates the operation of individual loads from measurements of the combined power usage for multiple loads. The typical system layout, depicted in Figure 1-1, allows for simultaneous measurements of all connected loads

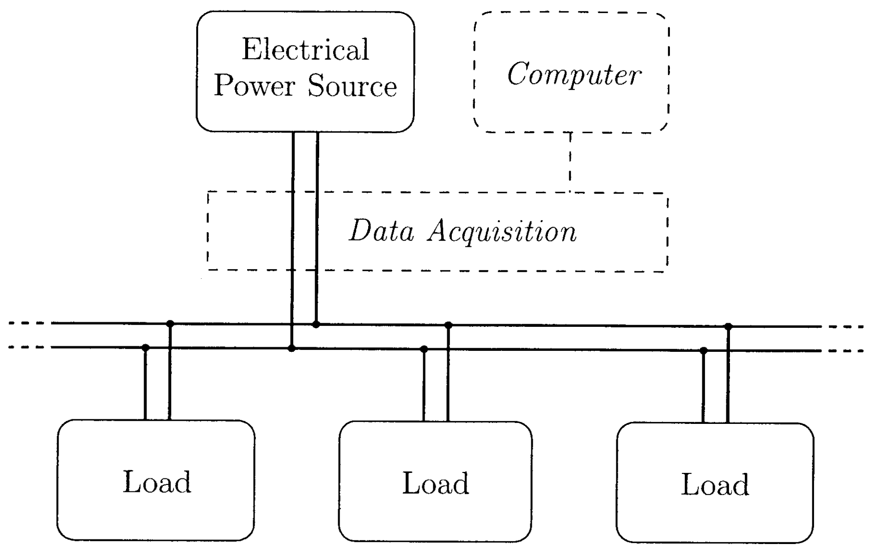


Figure 1-1: Typical system layout for non-intrusive monitoring of several loads. Data acquisition and processing occurs at a central location.

from one central point. This is *non-intrusive* in that no modifications or measurements need to be made at the individual loads. For a single-phase system, the installation involves just one voltage measurement and one current transducer, as shown in Figure 1-2. The collected aggregate power data can be used as-is to determine the overall state of the system, including total real power consumption, reactive power consumption, and harmonic production, but it is typically disaggregated to allow for specific analysis of particular loads. The methods for disaggregation vary. For events that are temporally disjoint, this may be as straightforward as separating the data in time. Other situations may involve subtracting out constant or predictable patterns, or grouping the data by harmonic content. The ability of NILM to distinguish between different loads in the collected data is discussed in [8].

Once separated, the power signatures for loads may be analyzed individually. Since virtually every aspect of a system's physical behavior involves the use of power in some form, the collected electrical power data conveys useful information about nearly all monitored systems. Measured characteristics of the power usage can be used to apply models and make deductions about the state of the loads. One typical use of the NILM system is for load identification and scheduling, where the uniqueness

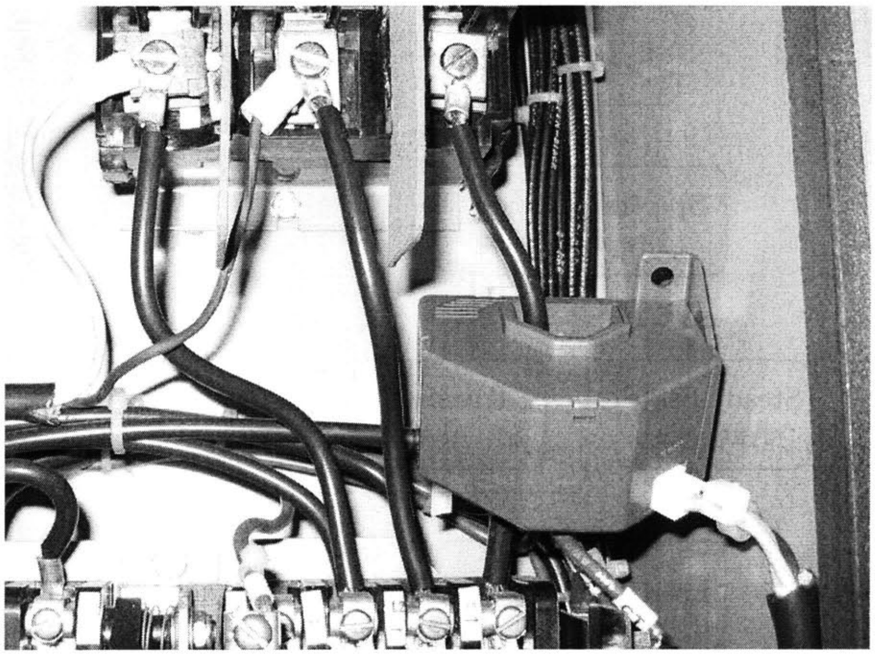


Figure 1-2: Non-intrusive monitoring of multiple loads at a distribution panel. The installation consists of a single current transducer to measure current, and two leads to measure voltage.

of startup transients or steady-state usage can be used to determine what loads are on and at what times they run. The disaggregated data is also very useful for diagnostics that determine faults or failures; some types and methods of diagnostics are explained and developed further in Chapter 2.

1.1.1 NILM Theory

There are several approaches to performing diagnostics and identification through non-intrusive monitoring. A diagram of a typical NILM signal processing workflow is shown in Figure 1-3. In general, data is first preprocessed to create power spectral envelopes, as described in §1.1.2. Then, load-specific methods are used to analyze systems. One scheme, used in the non-intrusive appliance load monitor (NALM) in [3, 13] and described in §1.1.3, is to classify the power consumption in steady-state. The non-intrusive transient classifier (NITC) in [13], on the other hand, focuses on the shapes and characteristics of transient power usage, using methods described

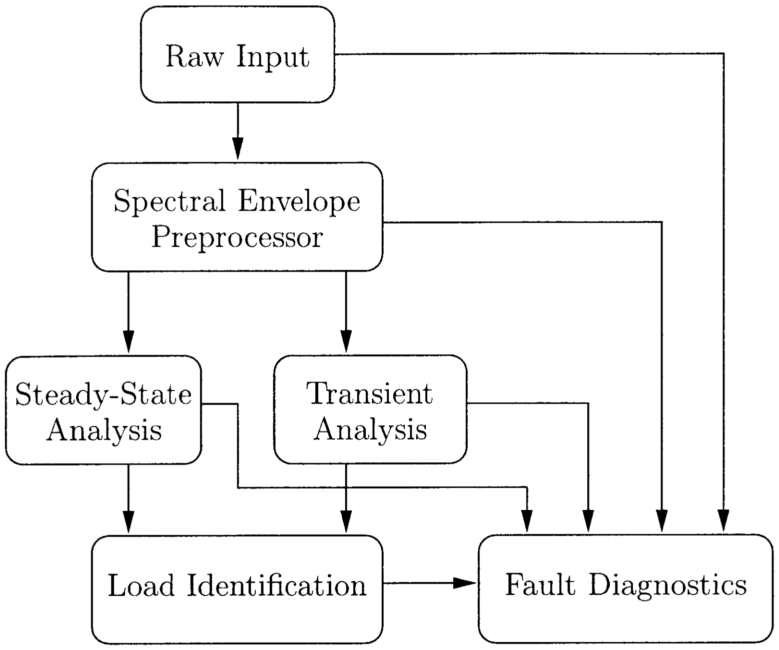


Figure 1-3: Block diagram of typical NILM data processing stages. Fault diagnostics often uses results from many of the earlier stages.

in §1.1.4. Such methods are not exclusive and may be combined or modified to create a particular diagnostic indicator.

1.1.2 Power Envelope Generation

The NILM hardware usually measures raw current and voltage. For load monitoring and diagnostics, the primary interest lies in power usage. For a direct-current load, the transformation from one to the other is straightforward, as the power is proportional to the measured current. Alternating-current loads, on the other hand, often exhibit behaviors that are synchronous with the AC line frequency, and it is usually desirable to extract out the power envelopes from this for analysis. This envelope generation is done by the *preprocessor*, so named because it is almost always the first step before classification and other processing on the recorded data.

The goal of the preprocessor is to extract spectral envelopes, which are short-term averages of harmonic content present at each of the harmonics of the incoming line frequency. As reviewed in [13], the in-phase spectral envelopes a_k of an input current signal $x(t)$ are

$$a_k(t) = \frac{2}{T}\int_{t-T}^{t} x(\tau)\sin(k\omega\tau)d\tau \tag{1.1}$$

where k is the harmonic index, and the quadrature spectral envelopes are

$$b_k(t) = \frac{2}{T}\int_{t-T}^{t} x(\tau)\cos(k\omega\tau)d\tau \tag{1.2}$$

For NILM purposes, the time t is referenced such that the term $\sin(\omega\tau)$ in Equation 1.1 is phase-locked to the voltage measurement, which can be achieved using a Kalman filter [14]. The averaging interval T is one or more periods of the line frequency. Under these conditions, the spectral envelopes are computed as $P_k = a_k$ and $Q_k = -b_k$, and these are the outputs of the preprocessor. The outputs are defined in this way so that the values P_1 and Q_1 correspond to the conventional definitions of real and reactive power.

Figures 1-4 and 1-5 demonstrate the utility of using the preprocessor for AC loads. The first plot is the raw input from a data acquisition board connected to a computer. The second shows P_1 and Q_1 as computed by the preprocessor. The preprocessed output provides a much clearer view of the electrical power usage. For example, a glance at Figure 1-5 shows that this load draws more reactive than real power, a fact that was not immediately apparent in the raw data.

The implementation of the preprocessor has changed over the years. Because it is a relatively complex but fixed operation, hardware implementations have been used to efficiently compute the spectral envelopes from the incoming analog signals [11]. Newer hardware implementations have included digital filters and implementation on dedicated digital signal processors [13]. Today, the availability of inexpensive and fast general-purpose computers makes a purely software implementation attractive. Writing the preprocessor as a software component also allows for greater flexibility in controlling the parameters of the envelope extraction. The current NILM system uses the preprocessor developed in [13], with improvements as described in §4.1.

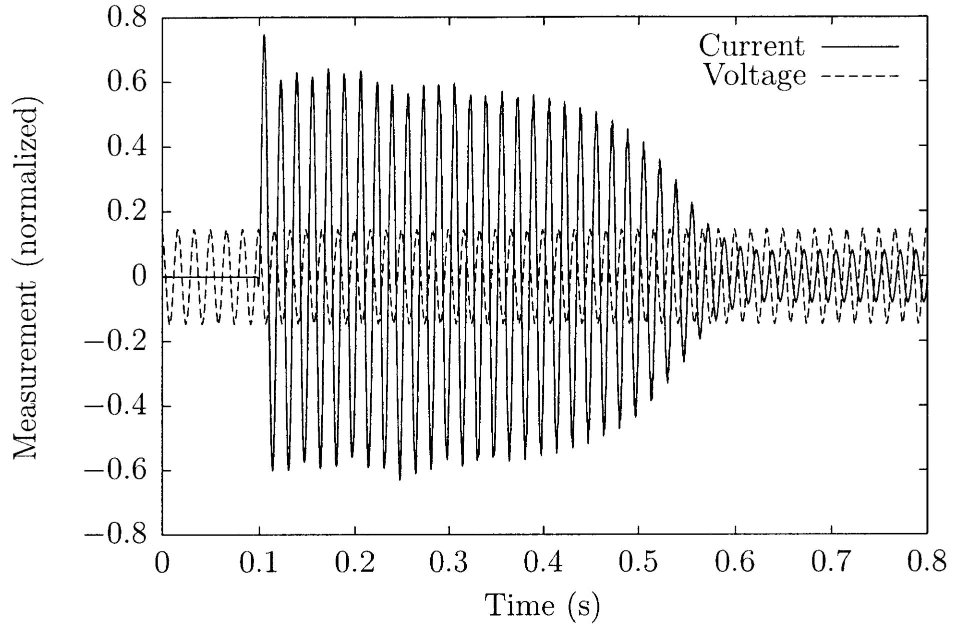


Figure 1-4: Raw AC voltage and current measurements taken during a motor startup transient.

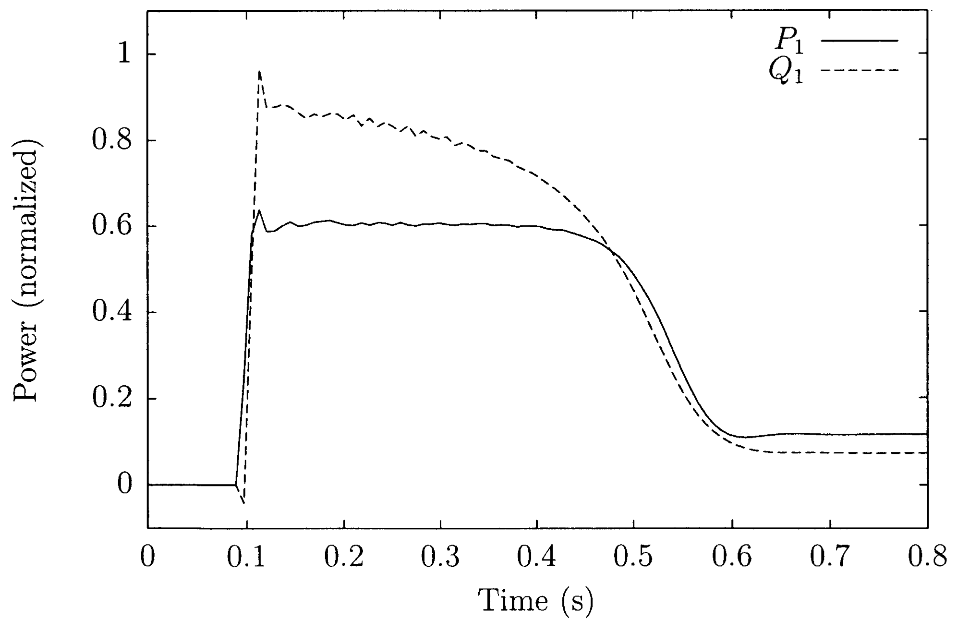


Figure 1-5: Result of applying the preprocessor to the data in Figure 1-4. These spectral envelopes provide a much clearer view of the electrical power usage.

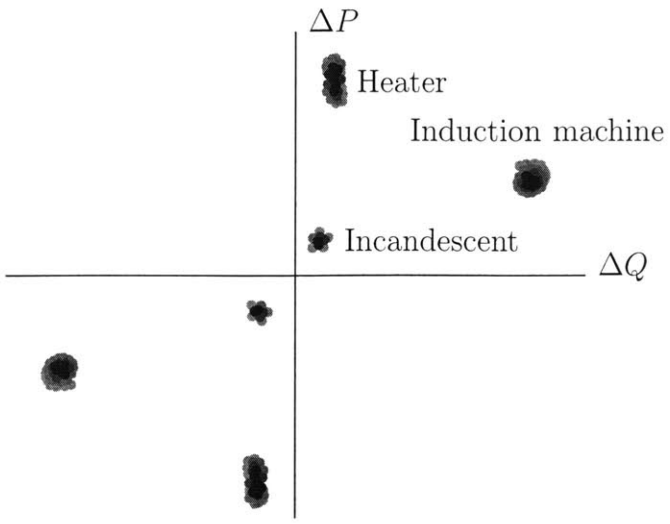


Figure 1-6: Plot of measured ΔP and ΔQ values for various loads, used in steady-state analysis. From [13].

1.1.3 Steady-State Analysis Approach

Once power spectral envelopes are computed, approaches such as steady-state analysis can be applied to identification and diagnostics. Steady-state analysis looks at the long-term step changes in P and Q that occur when a particular load turns off or on. An example plot in ΔP and ΔQ space is shown in Figure 1-6. As demonstrated, these step changes form clusters based on the loads. Since this method looks only at changes from settled values, these clusters can be formed and differentiated even when many loads are present and drawing power. Steady-state analysis is particularly good for identification of loads. In addition, deviation from the expected clusters can indicate faults, but more complicated diagnostics are often difficult with this method because of the limited state space.

1.1.4 Transient Analysis Approach

Another approach is to use the short-term transient power usage as shown in Figure 1-7 to identify loads and perform diagnostics. Transients are associated with any event in which the power usage of the system changes quickly. Typically, this occurs when a load turns on and off, and may also occur with certain types of failures such as

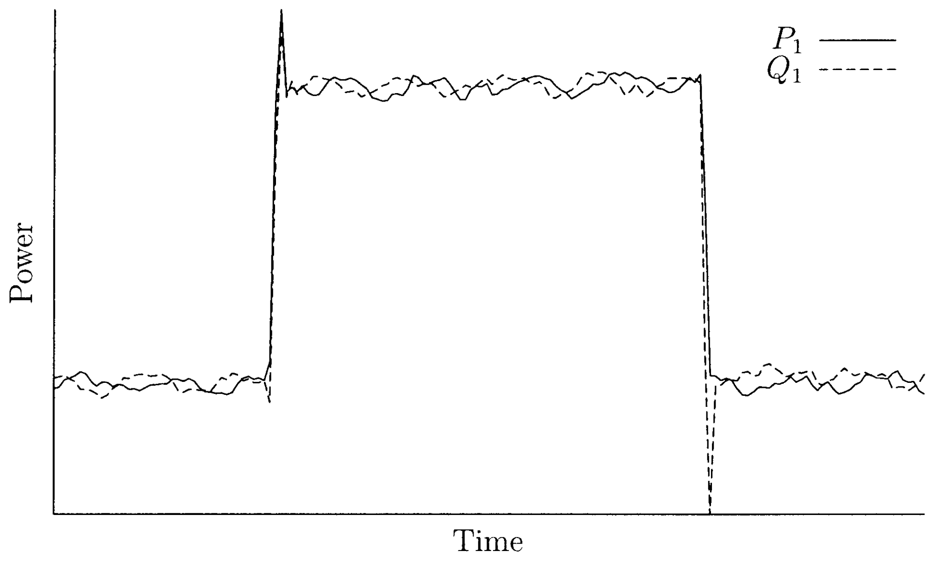


Figure 1-7: Plot of the power envelopes generated by turning an AC fan on and off, showing two characteristic transient events.

the physical breaking of a link. Transients can often convey more information than steady-state data, because it is more likely that they will differ between types of loads. Furthermore, if more detail is desired on a particular transient event, a higher sampling rate or resolution can be used when acquiring data. As a result, even very slight and brief changes in the shape of a transient can be analyzed. Transients are particularly good for diagnostics, as many mechanical systems use the most power during startup, and mechanical faults can show up more easily there. The diagnostic developed in §2.2 is based on characteristics found in the transient.

For basic identification and load scheduling, statistical and heuristic based approaches can be used to match transients to exemplars. The most straightforward approach is to create exemplars manually by selecting them from an envelope waveform, as was done for the NILM software in [13], or by having the computer detect a manually-triggered transient as does the software in §4.3. Once transient waveforms are stored, a least-squares or similar fit can be applied to determine when the transient shape is seen again. For more complicated transients, like the one shown in Figure 1-8, there may be multiple areas of interest with variable spacing in time. Software like

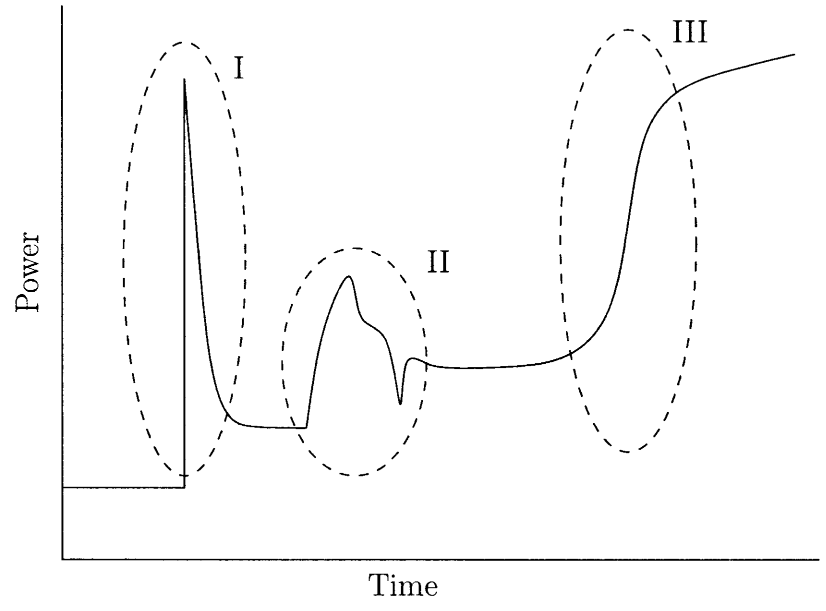


Figure 1-8: Startup transient of a single device, consisting of three smaller transients with variable spacing. From [11].

the system presented by [13] can be configured to match cases like this by matching the smaller sections and combining them into a larger match when appropriate.

Additional matching strategies can include a classifier-based approach in which the software automatically recognizes previously untrained transients as being generated by the same source. The software framework in Chapter 5 facilitates this sort of learning by providing a generalized database backend for storage of classifier data as well as the transients themselves. The manual training step would then be reduced to the task of merely naming the detected events.

1.1.5 NILM system example

Using the above components, a basic NILM system can be built. The satellite load monitoring system pictured in Figure 1-9 is one example that was constructed as part of this work. This particular system, which performs identification of transients based on manual training, uses a number of new and unconventional hardware components that makes it an useful platform for designing and testing the flexibility of the NILM.

The system consists of a number of low-power DC loads fed by a single source.

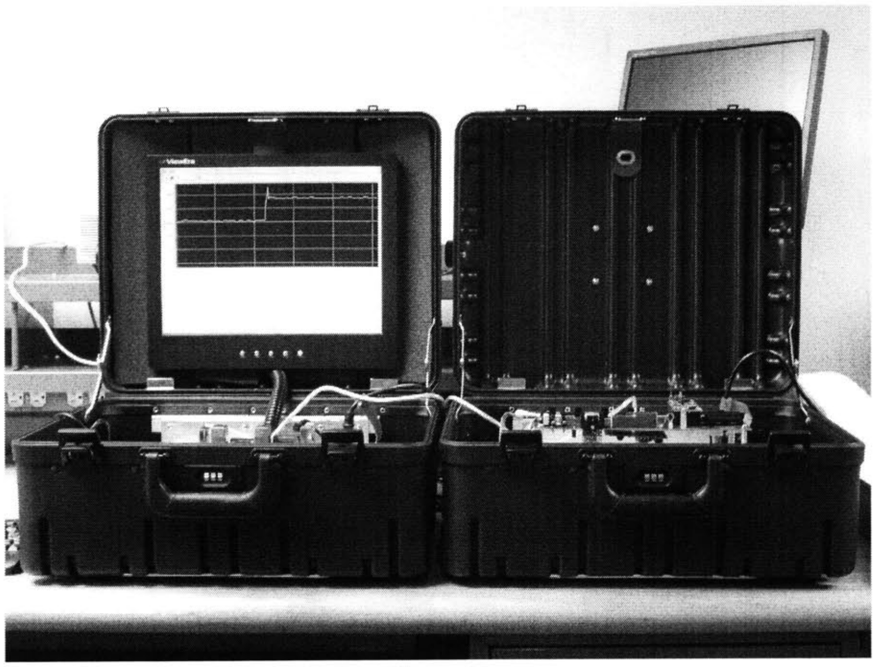
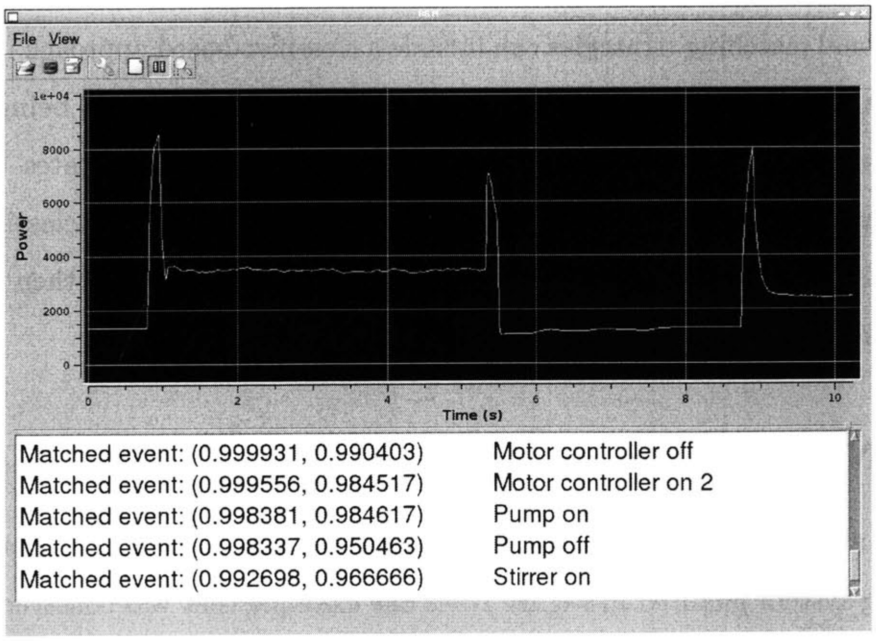


Figure 1-9: Photograph of a single-board NILM computer system, left, monitoring and identifying various satellite loads, right.



```
Matched event: (0.999931, 0.990403)      Motor controller off
Matched event: (0.999556, 0.984517)      Motor controller on 2
Matched event: (0.998381, 0.984617)      Pump on
Matched event: (0.998337, 0.950463)      Pump off
Matched event: (0.992698, 0.966666)      Stirrer on
```

Figure 1-10: Graphical user interface for the satellite load monitoring system. The power waveforms are shown in the top half of the window. Identified transients are listed at the bottom as they occur.

The current from the source is measured by the USB-based data acquisition module developed in §3.2, which connects to a single-board 266 MHz CM-i686M Computer-on-Module. The user interface software, pictured in Figure 1-10 and further described in §4.3, receives this data and shows the running plot of power usage versus time along the top. The system was previously trained to recognize particular transients, and these transients are matched with a least-squares fit and identified on the bottom of the screen as they occur. This particular system is DC, but can easily be adapted for AC loads by using a preprocessor to compute spectral envelopes.

1.2 Contributions and Organization

The wide variety of application-specific methods that can be profitably applied in a NILM system necessitates a flexible framework for signal processing. In order to develop the software and hardware for that framework, we begin with a survey of indicators and methods used in practice. The requirements of the NILM are motivated through the development of new diagnostic methods, with a particular focus on cycling systems and mechanical coupling failures in Chapter 2. The data-acquisition hardware improvements that were developed to facilitate data collection are described in Chapter 3. Support software for diagnostic development, including improvements to the preprocessor, a data recording and retrieval system, and a basic graphical interface, are presented in Chapter 4.

This work serves to demonstrate what is needed in the NILM framework. The relative disconnect between the data acquisition, storage, processing, and reporting stages becomes particularly apparent. The NILM software framework, presented in Chapter 5, unifies this. It provides for database-driven storage and tagging of data, with a generalized facility for processing this data through filters. Chapter 5 concludes with example client modules that demonstrate functionality.

Chapter 2

Diagnostic Indicators

One of the more useful features of any monitoring system is the ability to detect and report on performance characteristics that indicate operational failures or shortcomings. Such diagnostic abilities are central to systems like the Navy's Integrated Conditional Assessment System (ICAS) [9] and are the motivation for many NILM installations. In many cases, purely electrical monitoring of electromechanical systems has demonstrated the ability to both detect subtle problems and predict failures before they occur [4, 7, 9, 12, 13]. As a result, support for a wide variety of diagnostic indicators is a primary goal of the NILM software.

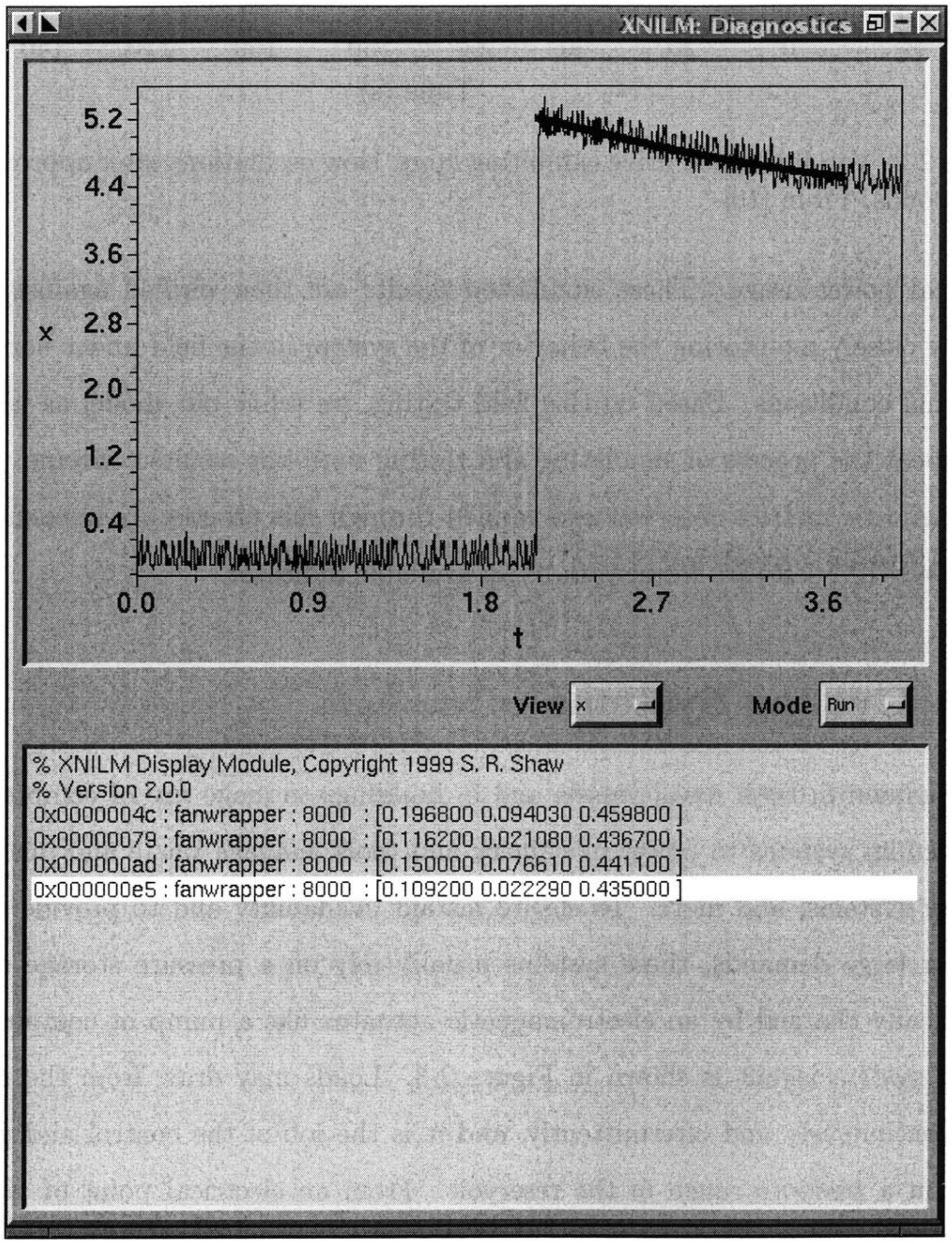
The flexibility of electrical monitoring has allowed for the development of a number of diagnostic methods. Transient classifiers as presented in [11, 13] can provide straightforward diagnostics like "motor B is off" or "the pump ran four times" by recognizing the power transients associated with on/off events. The satellite load monitoring system shown in Figure 1-10 is one example of identification-based diagnostics. With sufficient data and training, a transient classifier can also recognize anomalous events like a mechanical jam in an electromechanical actuator. However, it is generally limited to providing indicators based on easily distinguishable transient events, which is a category that many failure modes do not fit.

Another method for electrically determining the physical state of a system using the NILM is through modeling and parameter estimation as developed in [13]. This method involves creating a mathematical model for measurable quantities like current and voltage, based on the physical characteristics and parameters of the mechanical system. The model predicts the shape of power transients, and optimization methods can be used to infer original mechanical parameters based on observed transient shapes. The high sampling rate and resolution of the NILM hardware provides the ability to apply such modeling in the non-intrusive context, and this has been demonstrated with the determination of load characteristics of fans and pumps, as shown in Figure 2-1.

The NILM framework in [13] provides both of these identification and diagnostic techniques. However, field testing has shown that transient analysis and parameter estimation are not sufficient to cover all detectable conditions. For example, variable speed drive (VSD) controls on fans exhibit long-term steady-state behaviors that reflect physical conditions but do not involve individual on/off transients, as shown in Figure 2-2 [10]. The cycling systems in [7, 9] and §2.1 utilize a transient classifier to identify pump starts, but require further statistical analysis on a large set of pump starts to develop an indicator. Furthermore, complex systems like those in [7] and §2.2 can be difficult to model for parameter estimation.

To improve on the variety of diagnostic indicators that can be handled by the NILM software, a closer examination of real-world diagnostics and systems was undertaken in order to evaluate and understand the methods and tools required to compute such indicators. This analysis included a wide variety of systems used in satellites, automobiles, and building services, with a particular focus on diagnostics in the context of naval systems. The work with cycling systems in §2.1 and mechanical coupling failure in §2.2 demonstrated particular functionality and features that the NILM software needs to have. In the process, we also reveal some exciting new techniques for creating diagnostics for systems that are relatively disjoint from their electrical input.

The approaches developed here suggest a common process for finding and developing such diagnostic metrics. First, we examine the system and create a simulation model based on the underlying mechanics and physics. Then, we simulate the physical problems we are trying to diagnose and observe the effects they have on the



```
% XNILM Display Module, Copyright 1999 S. R. Shaw
% Version 2.0.0
0x0000004c : fanwrapper : 8000  : [0.196800 0.094030 0.459800 ]
0x00000079 : fanwrapper : 8000  : [0.116200 0.021080 0.436700 ]
0x000000ad : fanwrapper : 8000  : [0.150000 0.076610 0.441100 ]
0x000000e5 : fanwrapper : 8000  : [0.109200 0.022290 0.435000 ]
```

Figure 2-1: System identification results for a ventilation fan. The numeric tags on the bottom contain estimated parameter information. The thicker line in the plot is the result of a simulation with the estimated parameters. From [13].

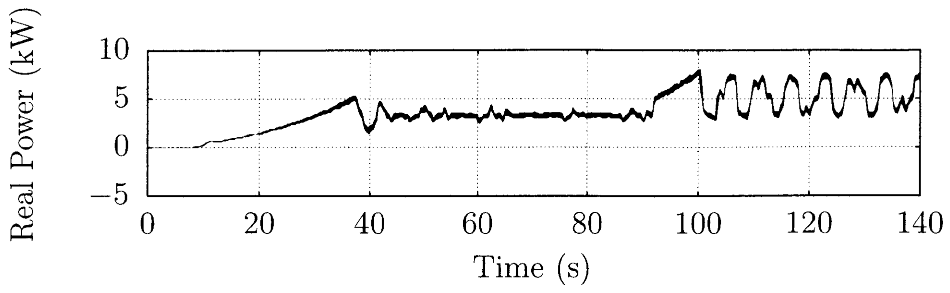


Figure 2-2: Variable speed drive exhibiting large, slow oscillations after approximately 100 seconds. From [10].

electrical power usage. These simulation results are then verified against real-life data by closely monitoring the behavior of the system in the field under normal and abnormal conditions. Based on this field testing, we refine our model as necessary, and repeat the process of simulating and testing until our results converge. Finally, the diagnostic metrics observed and verified through this process are implemented in the new NILM software framework as an available indicator.

2.1 Cycling Systems

It is common in both naval vessels and in buildings to make use of compressed air and vacuum systems to power machinery and tools, manage waste and dust, create delivery systems, and more. To ensure instant availability and to provide for temporarily large demands, these systems usually rely on a pressure storage reservoir periodically charged by an electromagnetic actuator like a pump or compressor. A typical system layout is shown in Figure 2-3. Loads may draw from the reservoir both continuously and intermittently, and it is the job of the control and pump to maintain a pressure range in the reservoir. From an electrical point of view, this behavior creates a *cycling system*, where the power usage follows a regular cycle of charging and discharging based on pressure. A diagram of one such cycle is shown in Figure 2-4.

Cycling systems create unique and interesting problems for the non-intrusive monitoring system because the loads under consideration are only indirectly powered by the electric charging unit. In many cases, only a small fraction of total time is spent

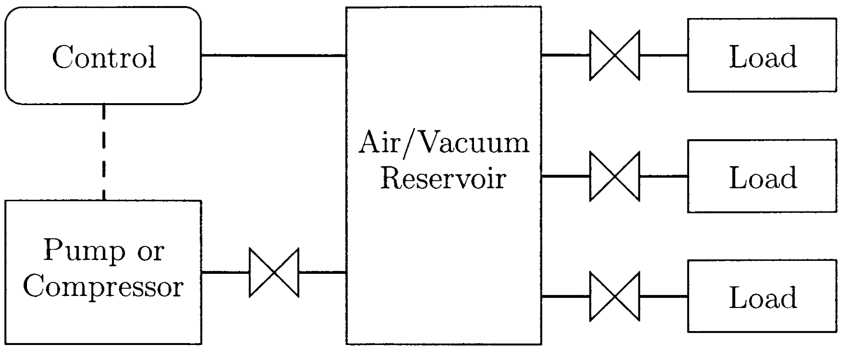


Figure 2-3: Diagram of a typical pneumatic cycling system. Solid lines are air, and dotted lines are electrical connections.

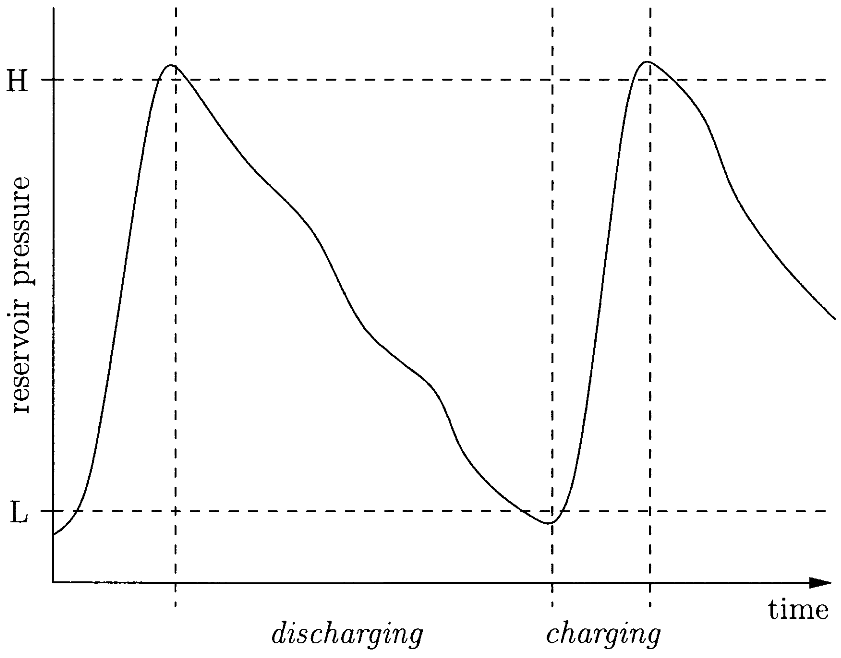


Figure 2-4: Operation of a cycling air-supply system. When pressure reaches H, the compressor turns off and the system discharges as loads are used. When pressure falls to L, the compressor runs and the system charges again.

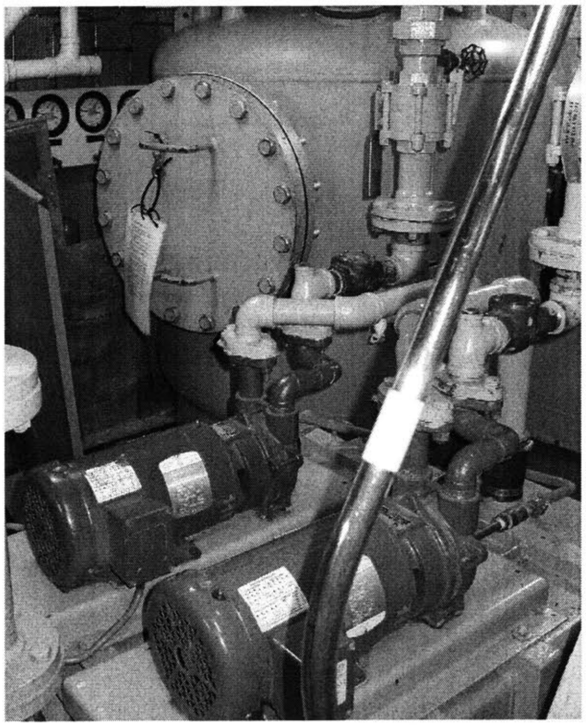


Figure 2-5: Photograph of the twin pumps used to recharge the vacuum reservoir on the *USCGC SENECA*. The reservoir is in the rear.

actively recharging the system, and so the electrical system is likely to be completely idle when some particular failure, leak, or other event occurs, making transient classification and transient parameter estimation alone ineffective. Instead, we show that the NILM system can use statistical techniques to still create a highly detailed view of the system loads and their state. The ability to perform load monitoring and diagnostics on cycling systems demonstrates an intriguing new application of electrical monitoring on these primarily non-electrical systems.

2.1.1 *SENECA* Data

Our primary testbed for the development of cycling system diagnostics is the *USCGC SENECA*, a 270-foot U.S. Coast Guard cutter. The *SENECA* uses a vacuum air wastewater disposal system. The vacuum reservoir is evacuated by two pumps, shown in Figure 2-5, which can run individually or simultaneously depending on the measured and desired vacuum pressure. Valves are located in bathrooms and sinks throughout the ship, and upon opening, allow waste to be drawn through the system and into a collection unit. Like many pneumatic systems, the *SENECA*'s vacuum system is plagued by air leaks, which results from breaks in the plumbing and improperly sealed valves. On several occasions, high leak rates have caused the evacuating pumps to overload and run too frequently, resulting in system failure [7].

The NILM was applied to this situation with the goal of detecting leaks early, so that they may be identified and fixed before becoming a serious problem. The primary job of this cycling system diagnostic, therefore, is to differentiate between leaks and legitimate usage patterns. This is a difficult problem, as usage can be expected to vary by hour, day, crew size, and crew behavior. A metric based strictly on overall flow is inconclusive. Furthermore, due to the disconnect between the electrical and pneumatic system, the NILM cannot directly monitor individual load usage. Instead, we rely on statistical approaches.

To gather baseline data for analysis and verification, a load monitoring system based on the hardware in §3.1 and the software in §4.2 was installed on the *SENECA*. A flow meter was installed on a vacuum collection tank gauge line, and the throttle valve on the flow meter was used to introduce controlled fixed leaks [7]. The recorded AC waveform data was converted to power spectral envelopes by the preprocessor in §4.1. From this, transients corresponding to the individual pump runs are readily apparent. A single pump run is shown in Figure 2-6, and a collection of pump runs over the course of an hour is shown in Figure 2-7. The lengths of the discharge times between pump runs is computed by the script `time-between.pl` in Appendix A.1.1. The statistical distribution of these discharge times, which are measured between a pump shutdown and the subsequent pump restart, are the basis of our analysis.

Initial data was collected while the *SENECA* was at sea for a month, with no artificial leaks inserted through the flow meter. The resulting histogram of the time between pump runs, shown in Figure 2-8, covers a two-week period of that data, and follows the shape demonstrated to be indicative of system behavior when no leaks are present [6, 7].

The data presented in Figure 2-9, on the other hand, corresponds to a two-week period during which the system had an unplanned large leak due to a faulty check

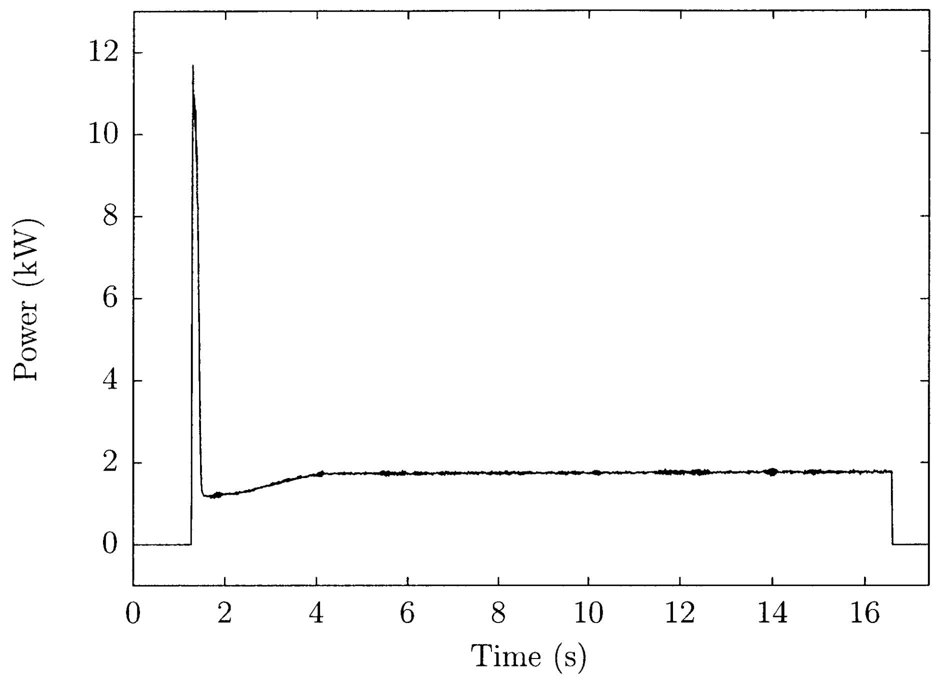


Figure 2-6: Power usage of the *SENECA* vacuum pump during a single charging cycle. The transient is clear and easy to match.

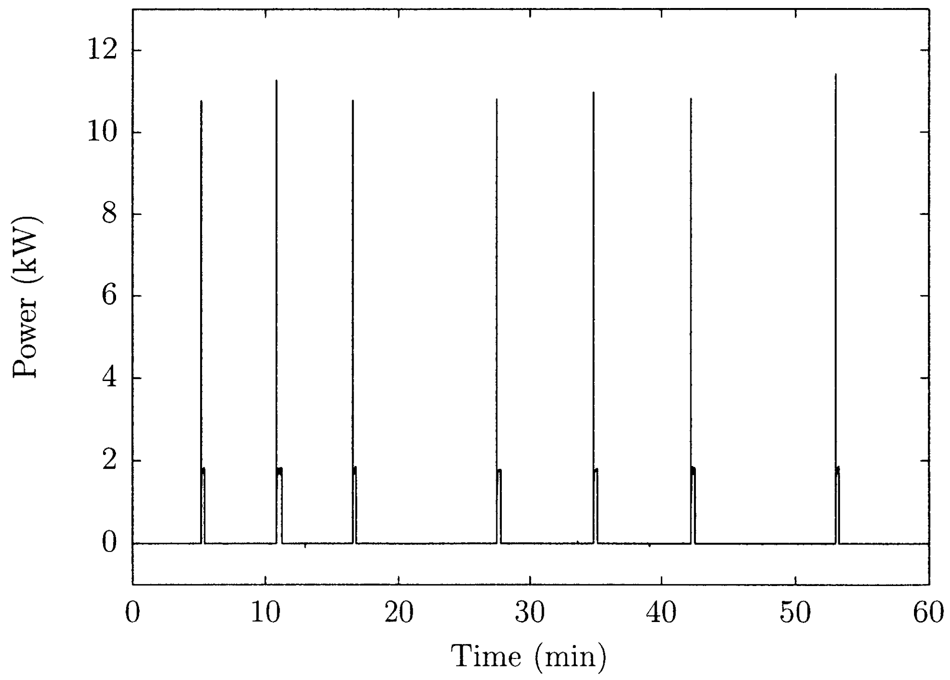


Figure 2-7: Power usage of the *SENECA* vacuum pump during multiple charge/discharge cycles over the course of an hour.

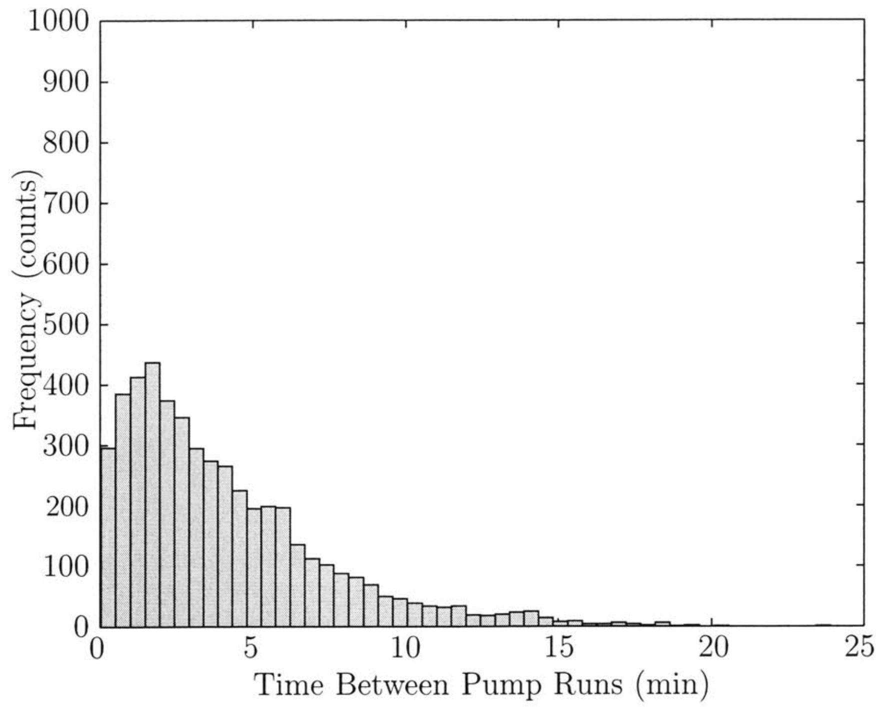


Figure 2-8: Histogram showing the distribution of the time between pump runs over a two-week period during which no leaks were known to exist. From [7].

valve. It is clear from the difference between the two histograms that such leaks have a large effect on the distribution of pump times; in particular, nearly all runs occur within two minutes of the previous cycle in the high-leak case.

Further tests with variable leak rates were undertaken to determine how distinguishable different leaks might be. These tests used the flow meter with a variable throttling valve to set three different leak rates: no leak, 30 standard cubic feet per hour (SCFH), and 50 SCFH. For each leak, data was collected for four days during the hours of 2200-0600 local time. This experiment was carried out while the ship was in port and at night in order to minimize influence by crew usage. The resulting histograms, presented in Figure 2-10, shows a shifting effect similar to the underway data in Figures 2-8 and 2-9, where the pump runs more frequently and with less discharge time as the magnitude of the leak increases.

These tests and data show that the effects of a leak can be detected and quantified based on electrical monitoring alone. What they do not demonstrate, however, is whether the NILM is also capable of distinguishing between the predictable loss of a

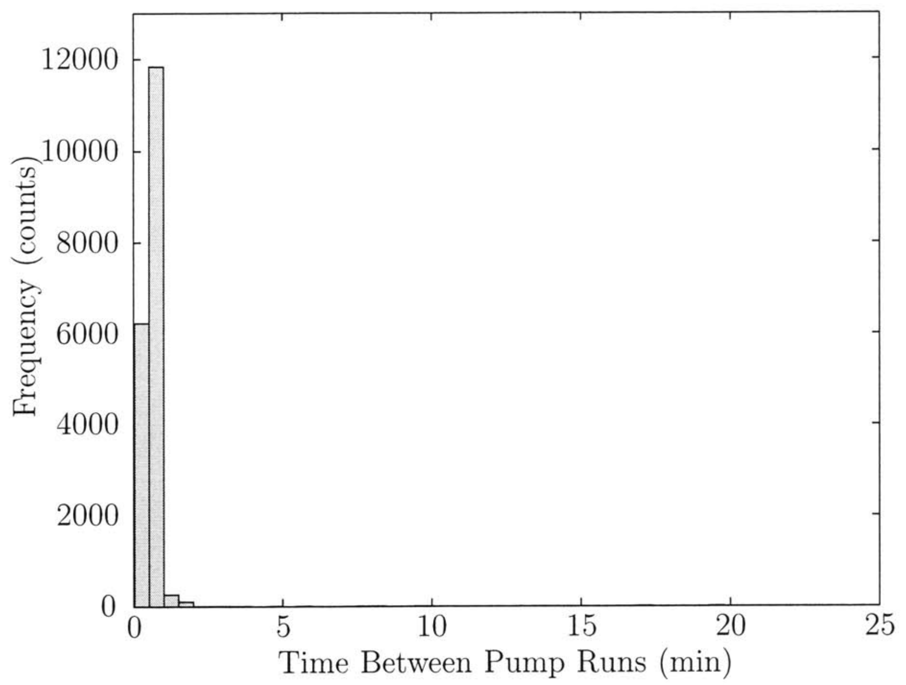


Figure 2-9: Histogram showing the distribution of the time between pump runs over a two-week period during which the system had a large leak in a faulty check valve. From [7].

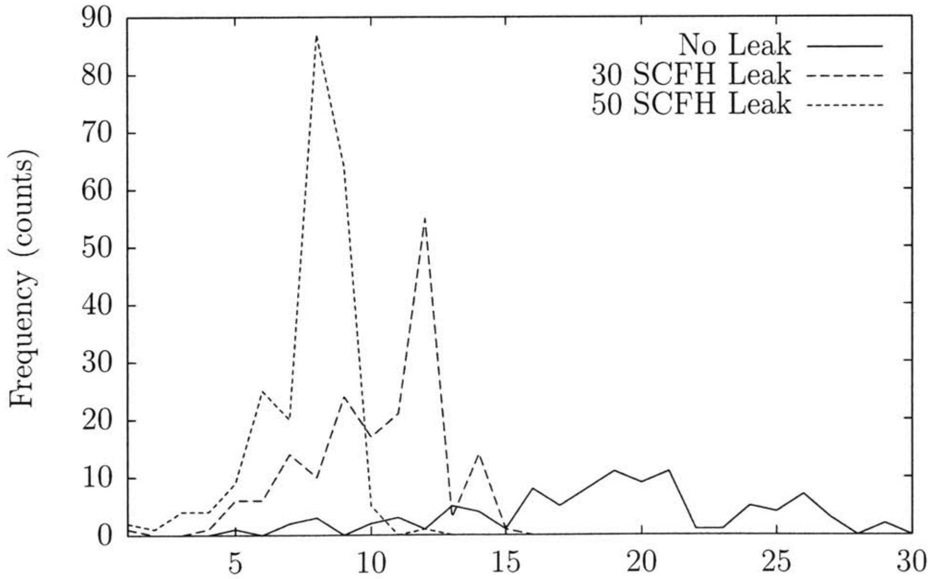


Figure 2-10: Three histograms corresponding to three different manually-induced leaks in the *SENECA*. The peak of the histogram shifts to the left as the leak rate is increased.

steady leak and the randomized aspect of crew usage. To explore the creation of such a metric, a simulation model of the system was created.

2.1.2 Simulation

The experimental setup on the *SENECA* provides for a manually-induced overall leak rate. There are a number of other important parameters like crew usage that have a large effect on the system, but, since the *SENECA* is an active vessel, these variables cannot be controlled directly. Instead, the observed baseline data was used to create and verify a simulation model of the vacuum system.

The developed software, listed in Appendix A.1, simulates three primary aspects of the sewage system, described in detail in the following section:

1. Pump characteristics and control system behavior
2. Vacuum loss due to constant leaks in the system
3. Vacuum loss due to crew usage, modeled as discrete "flush" events

Design

The pump characteristics, pump control, and leak behavior is designed to match the behavior of the physical twin pumping system. There are six configurable parameters for the pumps and constant leak, which are included in the parameter list in Table 2.1. The pressures are measured as the difference between atmospheric and reservoir pressures, so a higher "pressure" in the simulation represents a higher vacuum. The pumping rates and leak rate are assumed to be constant over the expected pressure ranges. Measurements of system pressure over time have verified that this is a reasonable assumption.

In normal operation, only one pump is used, and it turns on when P_{low} is passed, and turns off when P_{high} is reached. If the leak rate is very high, or a large number of flush events arrive at once, then the action of one pump is insufficient, and so the vacuum pressure of the system may continue to drop. If the pressure falls below P_{lower}, the second pump starts, and both pumps turn off together when P_{high} is reached.

Name	Description	Units
T	Simulation time	hours
Pump characteristics and control:		
P_{low}	Low pressure point at which one pump turns on	in. Hg
P_{lower}	Low pressure point at which both pumps turn on	in. Hg
P_{high}	High pressure point at which running pumps turn off	in. Hg
R_1	Pumping rate of the first pump	in. Hg / hr
R_2	Pumping rate of the second pump	in. Hg / hr
Loss due to fixed leak:		
R_{leak}	Constant leak rate	in. Hg / hr
Loss due to flush events:		
P_{flush}	Pressure drop caused by a single flush event	in. Hg
λ	Average expected rate of flushes	flushes / hr

Table 2.1: Parameters for the simulation model of the *SENECA* cycling system.

Constant leaks in the system are approximated as a single constant R_{leak}. Note that if R_{leak} is higher than the combined rate of both pumps $R_1 + R_2$, then the system cannot maintain pressure and the simulator generates an error.

The vacuum loss due to the crew is modeled as discrete flush events. The occurrence of these events is approximated as an M/D/∞ queue [17], which is equivalent to a Poisson arrival process. The Poisson process is used to describe events that are randomly spaced in time, and is often used to model natural arrival-type stochastic processes [2]. The work in [6, 7] confirms experimentally that the Poisson process is a close match for the crew usage aboard the *SENECA*. Table 2.1 includes the variables used in simulating crew flush events.

Simulating the Poisson arrival process

The simulation, written as a Matlab/Octave script and provided in Appendix A.1.2, is built around the primary function of simulating the Poisson arrival process of the crew usage. Let X be a Poisson random variable that represents the number of events per unit time, with an average rate of λ. By definition [2], the probability that X

equals some constant k for a Poisson variable is

$$P(X = k) = \lambda^k \frac{e^{-\lambda}}{k!} \tag{2.1}$$

To compute the probability of the number of events X_t per non-unit time t, we can scale the rate λ by multiplying it by t to get

$$P(X_t = k) = (\lambda t)^k \frac{e^{-\lambda t}}{k!} \tag{2.2}$$

For simulation purposes, we need to model the time between flush events. Let the random variable T represent the time before the next flush. The probability that T is greater than some value t is equal to the probability that we get no events in that time, so

$$P(T > t) = P(X_t = 0) \tag{2.3}$$

Substitute Equation 2.2 and solve to get

$$\begin{aligned} P(T > t) &= (\lambda t)^0 \frac{e^{-\lambda t}}{0!} &(2.4)\\ &= e^{-\lambda t} &(2.5) \end{aligned}$$

We can find $F(t)$, the cumulative distribution function (CDF), using the total probability theorem:

$$\begin{aligned} F(t) &= P(T \leq t) &(2.6)\\ &= 1 - P(T > t) &(2.7) \end{aligned}$$

Substitute Equation 2.5 into this to get

$$F(t) = 1 - e^{-\lambda t} \tag{2.8}$$

The probability distribution function (PDF), $f(t)$, can be found by taking the derivative of the CDF from Equation 2.8:

$$f(t) = P(T = t) = \frac{dF(t)}{dt} \tag{2.9}$$
$$= \frac{d}{dt}(1 - e^{-\lambda t}) \tag{2.10}$$
$$f(t) = \lambda e^{-\lambda t} \tag{2.11}$$

This $f(t)$ tells us the distribution of the time between flush events. We can compute the expected value $E(t)$ of this distribution by integrating over all time:

$$E(t) = \int_0^\infty f(t)dt \tag{2.12}$$
$$= \int_0^\infty e^{-\lambda t}dt \tag{2.13}$$
$$= \left[-\frac{1}{\lambda}e^{-\lambda t}\right]_0^\infty \tag{2.14}$$
$$= -\frac{1}{\lambda}(e^{-\infty} - e^0) \tag{2.15}$$
$$= \frac{1}{\lambda} \tag{2.16}$$

This result verifies that the variable λ represents the expected number of flushes per unit time.

The CDF of the random variable T, $F(T)$, is graphed in Figure 2-11. In order to synthesize values of T for the simulation, we can choose a random value R uniformly from $[0, 1]$ and return the inverse CDF, $F^{-1}(t)$. Graphically, this is equivalent to choosing a point on the vertical axis of Figure 2-11 and returning the corresponding value of t from the horizontal axis.

To prove that these synthesized values have the correct distribution, first assume that we create a new random variable T_s as described:

$$T_s = F^{-1}(R) \tag{2.17}$$

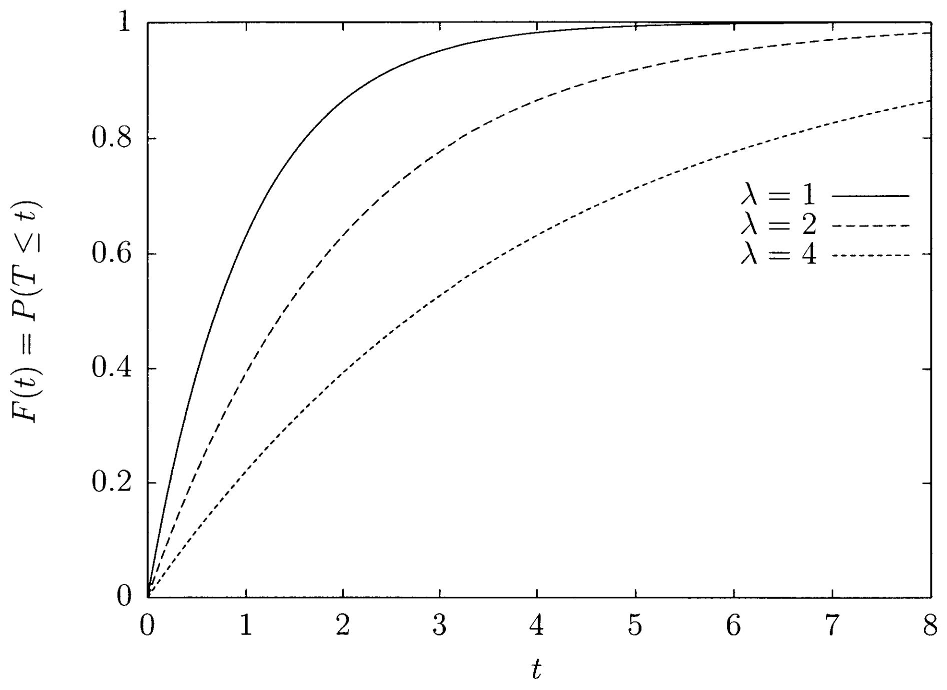


Figure 2-11: Cumulative distribution function (CDF) of the random variable T representing the time between flushes in the cycling system simulation.

The CDF of this new variable is

$$P(T_s \leq t) = P(F^{-1}(R) \leq t) \tag{2.18}$$

Apply F to the inequality on the right hand side to obtain

$$P(T_s \leq t) = P(R \leq F(t)) \tag{2.19}$$

Since R is uniformly distributed on $[0, 1]$,

$$P(R \leq k) = k \tag{2.20}$$

Substitute Equations 2.20 and 2.6 into Equation 2.19 to obtain

$$P(T_s \leq t) = F(t) \tag{2.21}$$

$$= P(T \leq t) \tag{2.22}$$

which indicates that our synthesized T_s does follow the same distribution as the original random variable T.

To compute values with the same distribution as T directly, we can therefore use Equation 2.17. To find T_s, we first invert the CDF $F(t)$:

$$R = F(t) \tag{2.23}$$

$$R = 1 - e^{-\lambda t} \tag{2.24}$$

$$t = -\ln(1 - R)/\lambda \tag{2.25}$$

If we define $R' = 1 - R$, this can be simplified to

$$t = -\ln(R')/\lambda \tag{2.26}$$

The new variable R' has the same distribution as R and so can be also chosen uniformly from $[0, 1]$. The result in Equation 2.26, combined with a uniform pseudorandom number generator, is used by the simulation to synthesize the crew flushes.

Implementation and Simulation Results

Flush events are the basis of the simulator. The main simulation loop, listed in Appendix A.1.2 as `sim.m`, generates the new time t_f at which the next flush event occurs. The supplemental script `pumpleak.m` is then used to calculate the effects of the twin pumps and the loss due to the constant leak, as described in §2.1.2, until the time t_f passes. At this point, the pressure drop P_{flush} is applied to the reservoir pressure, and the simulation continues in the same fashion until the next flush event.

The output of the simulator is the two vectors `pump_on` and `pump_off`, which

Name	Value
T	120 hr
P_{low}	14 in. Hg
P_{lower}	10 in. Hg
P_{high}	18 in. Hg
R_1	1100 in. Hg / hr
R_2	1000 in. Hg / hr
R_{leak}	*varied*
P_{flush}	1 in. Hg
λ	40 flushes / hr

Table 2.2: Simulation parameters used in Figures 2-12 through 2-15.

contain the times at which either pump turned on and the times at which both pumps turned off, respectively. The script `sim_between.m` in Appendix A.1.4 can be used to transform this output to the a single vector matching the format of `time-between.pl`. A histogram can then be computed and plotted. An example of the output of the simulator when no leaks are present is shown in Figure 2-12.

The simulator was run multiple times with the parameters shown in Table 2.2 and varying leak rates. The histograms corresponding to four such runs are shown in Figures 2-12 through 2-15. These four figures are representative of the behavior of the leaks. The simulated no-leak case in Figure 2-12 is similar to the real-world *SENECA* data from in Figure 2-8 where the system was healthy. The high-leak case in Figure 2-15, on the other hand, is approaching the form of the *SENECA* data in Figure 2-9 where a large leak was present.

The simulation results show a characteristic asymmetric distribution that is relative unchanged as the leak rate varies. Experiments with changing the parameters in Table 2.2 shows that this underlying distribution does vary with crew usage. The introduction of leaks, on the other hand, shows sharp "spikes" in the histogram. This can be seen in Figure 2-13, for example, at around 13 minutes. These spikes grow in magnitude and number and shift to the left as the leak rate is increased.

In order to understand how this distribution and these spikes arise, and whether they can be used to differentiate between crew usage and leaks, we can analyze the behavior of the simulator in order to explain their source.

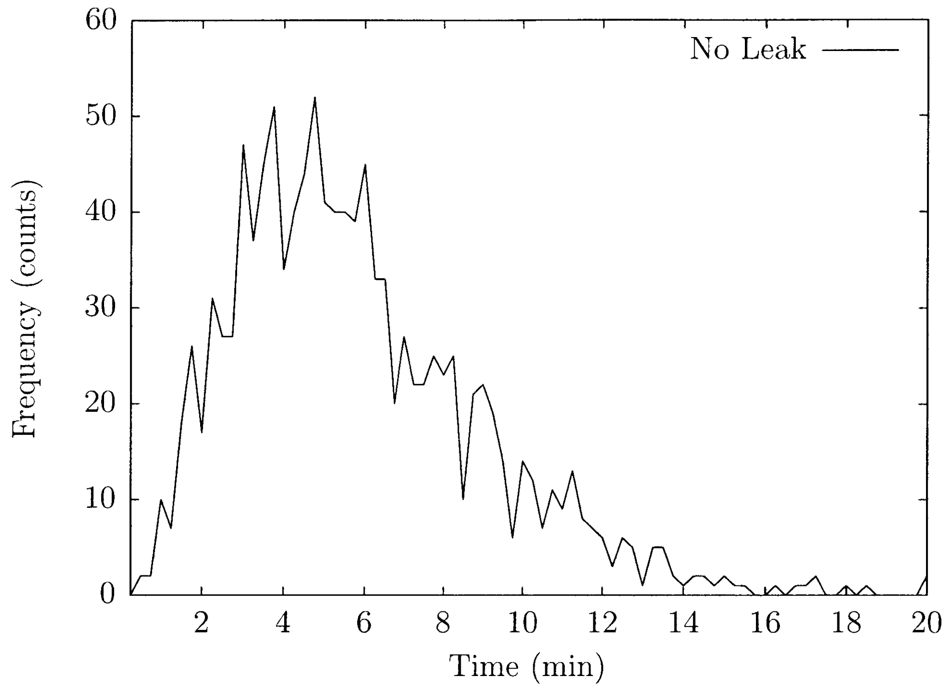


Figure 2-12: Histogram of the pump cycling data generated by the simulator with no leak. The parameters for this simulation are as shown in Table 2.2 where $R_{\text{leak}} = 0$ in. Hg / hr.

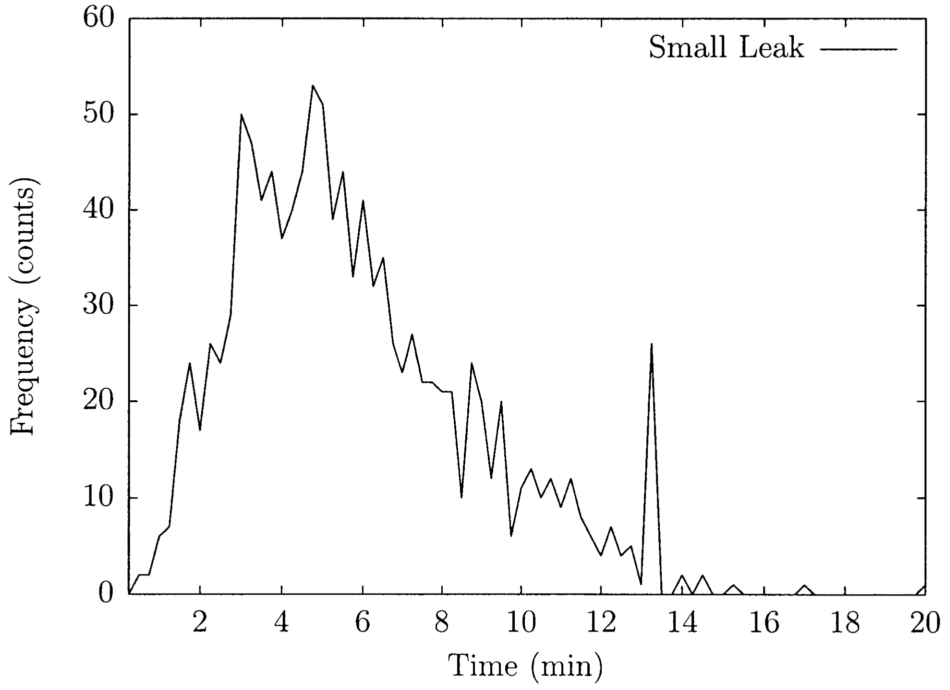


Figure 2-13: Histogram of the pump cycling data generated by the simulator with a small leak. The parameters for this simulation are as shown in Table 2.2 where $R_{\text{leak}} = 4.5$ in. Hg / hr.

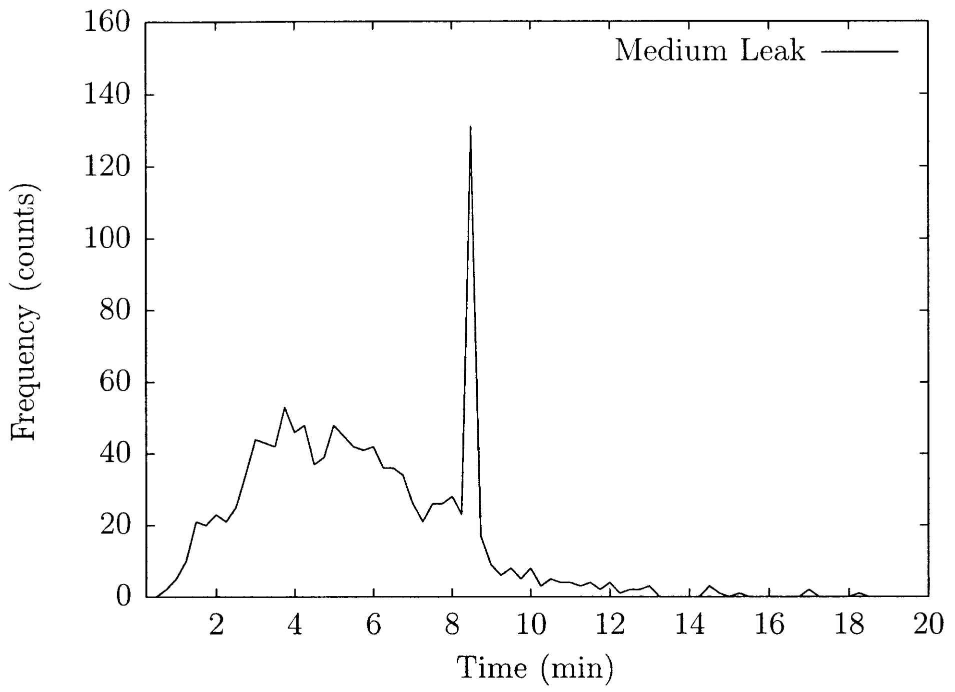


Figure 2-14: Histogram of the pump cycling data generated by the simulator with a medium-sized leak. The parameters for this simulation are as shown in Table 2.2 where $R_{\text{leak}} = 7$ in. Hg / hr.

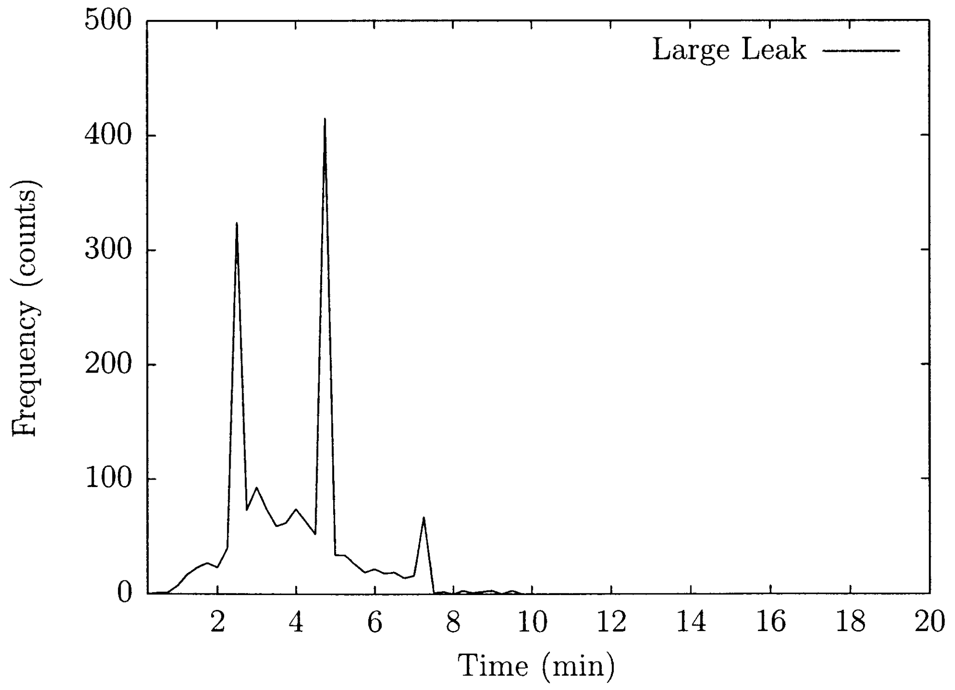


Figure 2-15: Histogram of the pump cycling data generated by the simulator with a large leak. The parameters for this simulation are as shown in Table 2.2 where $R_{\text{leak}} = 25$ in. Hg / hr.

2.1.3 Analysis

Crew Usage Alone

First, let us consider the case where there are no leaks, in order to determine the nature of the baseline distribution shown in Figure 2-12. Since the pumps turn on after a fixed pressure point is passed, and each flush causes the loss of a fixed amount of vacuum pressure, the number of flushes that must occur between pump runs is a constant. Assume for simplicity that this constant is an integer, and call it η. Then, the time T_p from the end of one pump run until the beginning of the next is the same as the time it takes for η flushes to occur:

$$T_p = T_1 + T_2 + \cdots + T_\eta \tag{2.27}$$

From Equation 2.11, we know that the PDF of each flush time is equal to

$$f(t) = \lambda e^{-\lambda t} \tag{2.28}$$

The PDF $f_p(t)$ of the summed variable T_p is equal to the convolution of the PDFs for the individual flushes [2]:

$$f_p(t) = \overbrace{f(t) * f(t) * \cdots * f(t)}^{\eta} \tag{2.29}$$

To solve for $f_p(t)$, first substitute Equation 2.28 in Equation 2.29 and apply the unilateral Laplace transform to both sides:

$$f_p(t) = (\lambda e^{-\lambda t}) * (\lambda e^{-\lambda t}) * \cdots * (\lambda e^{-\lambda t}) \tag{2.30}$$

$$\mathcal{L}[f_p(t)] = \mathcal{L}[(\lambda e^{-\lambda t}) * (\lambda e^{-\lambda t}) * \cdots * (\lambda e^{-\lambda t})] \tag{2.31}$$

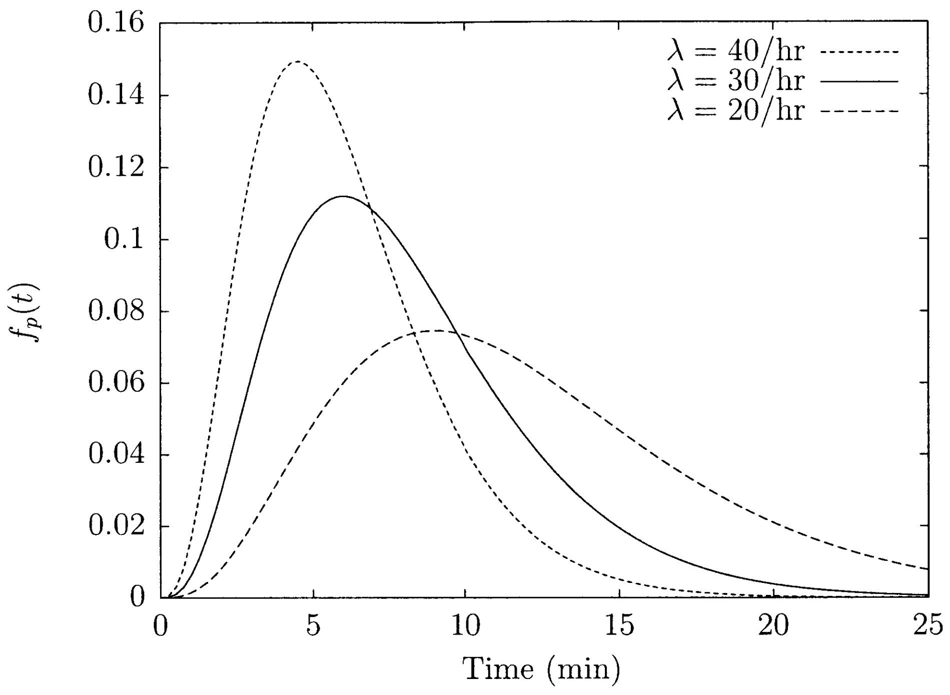


Figure 2-16: Plot of the Erlang distribution for $\eta = 4$ and varying λ. This distribution, from Equation 2.34, corresponds to the sum of multiple Poisson arrival times, and models the baseline distribution of crew usage with no leak.

which, using standard Laplace transformations and properties, simplifies to

$$\mathcal{L}[f_p(t)] = \frac{\lambda^\eta}{(s+\lambda)^\eta} \tag{2.32}$$

Then apply the inverse Laplace transform:

$$\mathcal{L}^{-1}[\mathcal{L}[f_p(t)]] = \mathcal{L}^{-1}\left[\frac{\lambda^\eta}{(s+\lambda)^\eta}\right] \tag{2.33}$$

and simplify to find $f_p(t)$:

$$f_p(t,\eta,\lambda) = \lambda^\eta \frac{t^{(\eta-1)}}{(\eta-1)!} e^{-\lambda t} \tag{2.34}$$

This distribution, shown in Figure 2-16 for fixed η and varying λ, is known as the Erlang distribution. The Erlang is a form of the more general Gamma distribution, which allows for non-integer η [15].

The CDF of the Erlang distribution is computed as the integral of the PDF:

$$F_p(t, \eta, \lambda) = \int_0^t \lambda^\eta \frac{\tau^{(\eta-1)}}{(\eta-1)!} e^{-\lambda\tau} d\tau \tag{2.35}$$

This integral has the transcendental solution [16]

$$F_p(t, \eta, \lambda) = \frac{\gamma(\eta, \lambda t)}{(\eta-1)!} \tag{2.36}$$

where γ is the incomplete gamma function, defined[1] as

$$\gamma(x, a) = \int_0^x e^{-t} t^{a-1} dt \tag{2.37}$$

The function $F_p(t, \eta, \lambda)$ from Equation 2.36, which represents the cumulative distribution of the time between pump runs due to crew usage with no leaks, is shown in Figure 2-17 for fixed λ and varying η.

Effect of Leaks

In the previous section, we assumed that each flush causes a fixed drop P_{flush} in vacuum pressure, and that some integer number of flushes η are necessary to cause the pump to run. That is, the total drop P_0 in vacuum pressure that occurs before the pump will run is

$$P_0 = \eta \cdot P_{\text{flush}} \tag{2.38}$$

Now consider the pressure drop of the system at time t when a leak is present. Let N represent the number of flushes that have occurred so far since the last pump run, and let R_{leak} represent the leak rate. The pressure drop at time t is

$$P = N \cdot P_{\text{flush}} + t \cdot R_{\text{leak}} \tag{2.39}$$

[1]There are conflicting definitions of the incomplete gamma function. Equation 2.37 is used by Mathematica and Maple. Matlab, Octave, Gnuplot, and many other mathematical programs use the definition $\gamma(x, a) = (1/\Gamma(a)) \int_0^x e^{-t} t^{a-1} dt$, where $\Gamma(a)$ is the complete gamma function.

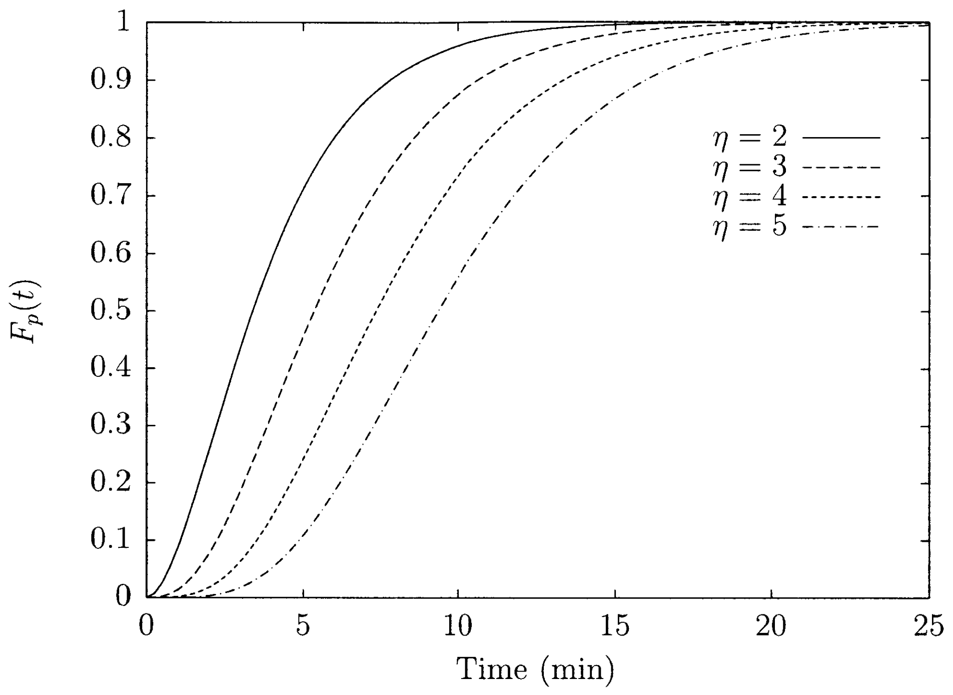


Figure 2-17: Plot of the Erlang cumulative distribution function from Equation 2.36 for $\lambda = 30/\text{hr}$ and varying η.

As before, the pump will run when the drop equals or exceeds P_0:

$$P \geq P_0 \tag{2.40}$$

and so, substituting Equations 2.38 and 2.39, the pump will run when

$$N \cdot P_{\text{flush}} + t \cdot R_{\text{leak}} \geq \eta \cdot P_{\text{flush}} \tag{2.41}$$

We can rearrange this to separate the flushing from the leaking:

$$\underbrace{N}_{\text{flushes}} + \underbrace{\frac{t \cdot R_{\text{leak}}}{P_{\text{flush}}}}_{\text{leak term}} \geq \eta \tag{2.42}$$

This equation, which indicates when the pump will run, describes an interesting behavior. Since N and η are both integers, the fractional portion of the leak term has no effect on the inequality. In other words, the leak will only affect the pump runs at the times when the leak term is a positive integer:

$$\frac{t \cdot R_{\text{leak}}}{P_{\text{flush}}} = \{1, 2, 3, \cdots\} \tag{2.43}$$

Solve for these times t to get

$$t = \{\tau, 2\tau, 3\tau, \cdots\} \tag{2.44}$$

where

$$\tau = \frac{P_{\text{flush}}}{R_{\text{leak}}} \tag{2.45}$$

Intuitively, this makes sense. In effect, the combined pressure drop from the steady leak after time τ has added up to equal a single flush. Until that happens, leaks have no effect, because we need an integer number of flushes to cause a pump run. For times before τ, we still require all η flushes to cause the pump to run. For times after τ, only $\eta - 1$ actual flushes are required. After 2τ, only $\eta - 2$ actual flushes are required, etc.

Now let us consider how this affects the complete probability distribution $f_c(t)$ of the time between pump runs. For $t < \tau$, the leak has no effect, and so the pump will behave according to the Erlang distribution derived earlier in Equation 2.34, with η required flushes:

$$f_c(t) = f_p(t, \eta, \lambda) \tag{2.46}$$

$$f_c(t) = \lambda^{\eta} \frac{t^{(\eta-1)}}{(\eta-1)!} e^{-\lambda t} \qquad \text{for } t < \tau \tag{2.47}$$

At time $t = \tau$, the effect from the leak will have added up to the amount of single extra flush. Thus, if $\eta - 1$ flushes have already occurred before $t = \tau$, the pump will run immediately at τ. That means that there is an additional probability of the pump running at τ, and this probability is equal to the accumulated probability that we had at least $\eta - 1$ but not η pump runs during $t < \tau$. Using the CDF from Equation 2.36, we can write this extra probability as

$$P(N = (\eta - 1))|_{t=\tau} = F_p(\tau, \eta - 1, \lambda) - F_p(\tau, \eta, \lambda) \tag{2.48}$$

$$= \frac{\gamma(\eta - 1, \lambda\tau)}{(\eta - 2)!} - \frac{\gamma(\eta, \lambda\tau)}{(\eta - 1)!} \tag{2.49}$$

The total probability of the pump running at $t = \tau$ is then the base probability from Equation 2.47 plus the extra probability from Equation 2.49:

$$f_c(t) = \lambda^{\eta} \frac{t^{(\eta-1)}}{(\eta - 1)!} e^{-\lambda t} + \frac{\gamma(\eta - 1, \lambda\tau)}{(\eta - 2)!} - \frac{\gamma(\eta, \lambda\tau)}{(\eta - 1)!} \qquad \text{for } t = \tau \tag{2.50}$$

For $\tau < t < 2\tau$, we only need $\eta - 1$ actual flushes to cause a pump run. Like $t < \tau$, this will follow the Erlang distribution, but now with the parameter $\eta - 1$:

$$f_c(t) = f_p(t, \eta - 1, \lambda) \tag{2.51}$$

$$f_c(t) = \lambda^{(\eta-1)} \frac{t^{(\eta-2)}}{(\eta - 2)!} e^{-\lambda t} \qquad \text{for } \tau < t < 2\tau \tag{2.52}$$

At this point, it is useful to consider the CDF of the complete pump distribution, which is

$$F_c(t) = \int_0^t f_c(t')dt' \tag{2.53}$$

For time $t < \tau$, the $f_c(t)$ is just an Erlang PDF, so $F_c(t)$ is the Erlang CDF with the same parameter η. At $t = \tau$, the added probability from Equation 2.49 corresponds exactly to the difference between $F_p(t, \eta - 1, \lambda)$ and $F_p(t, \eta, \lambda)$. Effectively, this means that on $F_c(t)$, there is a corresponding "jump" from the η curve to the $\eta - 1$ curve. For $t > \tau$, $f_c(t)$ is also an Erlang PDF, and $F_c(t)$ will therefore continue to follow the CDF of the Erlang with parameter $\eta - 1$.

This behavior is shown graphically in Figure 2-18. $F_c(t)$ is a piecewise function that begins on the Erlang CDF corresponding to the initial number of flushes needed for a pump to run, which in this example is $\eta = 4$. At every multiple of τ, the system

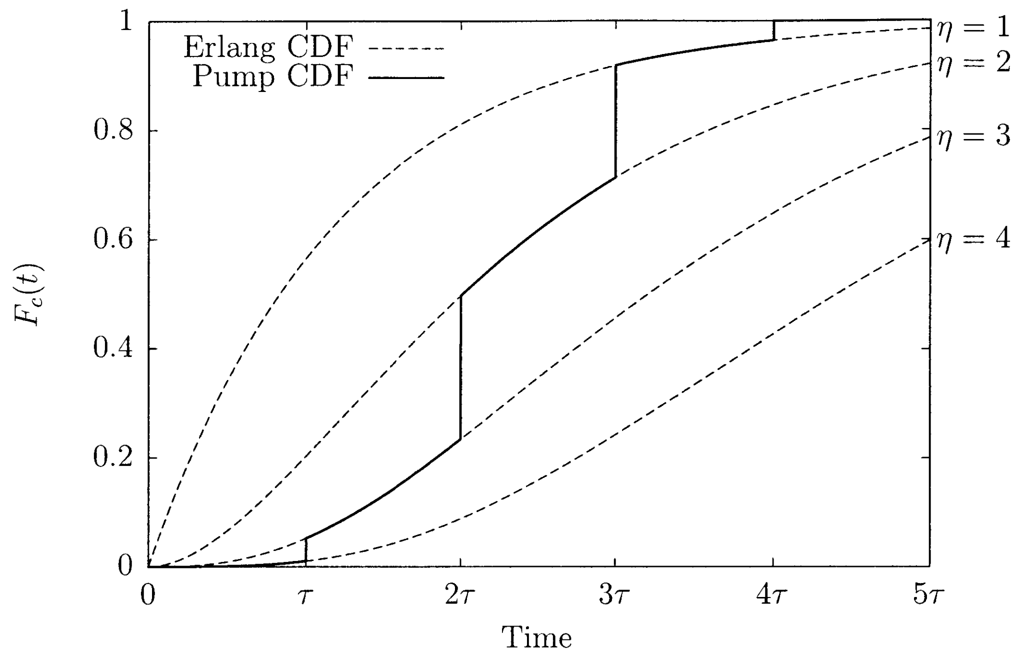


Figure 2-18: Plot of the CDF for the *SENECA* vacuum pump in the presence of both crew usage and leaks, for $\lambda = 10/\text{hr}$ and $\tau = 5$ min. The η for the pump system is 4. When t is a multiple of τ, the pump CDF "jumps" to follow the next Erlang CDF.

now behaves as if one fewer actual flush had been required all along, and so it follows the Erlang CDF corresponding to the next lower η parameter.

Following the form of Equations 2.47, 2.50, and 2.52, we can write the complete pump PDF as follows, where k takes integer values from 1 to η inclusive:

$$f_c(t,\eta,\lambda) = \begin{cases} \lambda^{(\eta-k)} \frac{t^{(\eta-k)}}{(\eta-k)!} e^{-\lambda t} & \text{for } (k-1)\tau < t < k\tau \\ \lambda^{(\eta-k)} \frac{t^{(\eta-k)}}{(\eta-k)!} e^{-\lambda t} + \frac{\gamma(\eta-k,\lambda\tau)}{((\eta-k)-1)!} - \frac{\gamma((\eta-k)+1,\lambda\tau)}{(\eta-k)!} & \text{for } t = k\tau, k < \eta \\ e^{-\lambda t} + 1 - \gamma(1,\lambda\tau) & \text{for } t = \eta\tau \\ 0 & \text{otherwise} \end{cases} \tag{2.54}$$

This function is plotted in Figures 2-19 through 2-22, using the same parameters as the simulation data from before. The zero-width impulses at $t = k\tau$ have been widened to match the bucket size used in the simulation histograms, but their areas are unchanged. The simulation data is also included for reference.

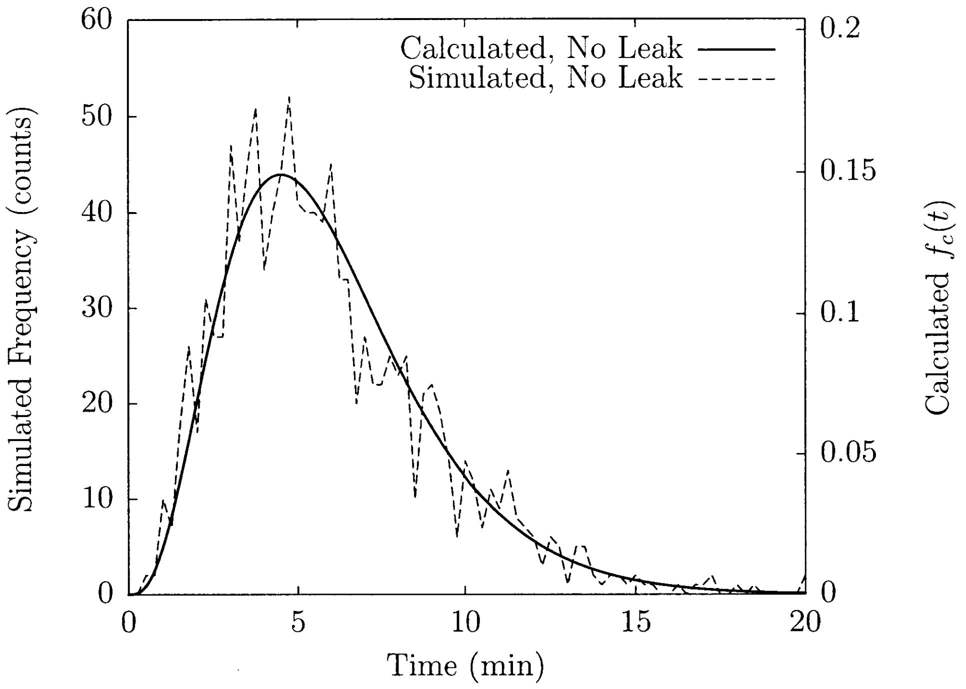


Figure 2-19: Calculated pump PDF plotted against simulated data. The system and leak parameters are the same as in Figure 2-12.

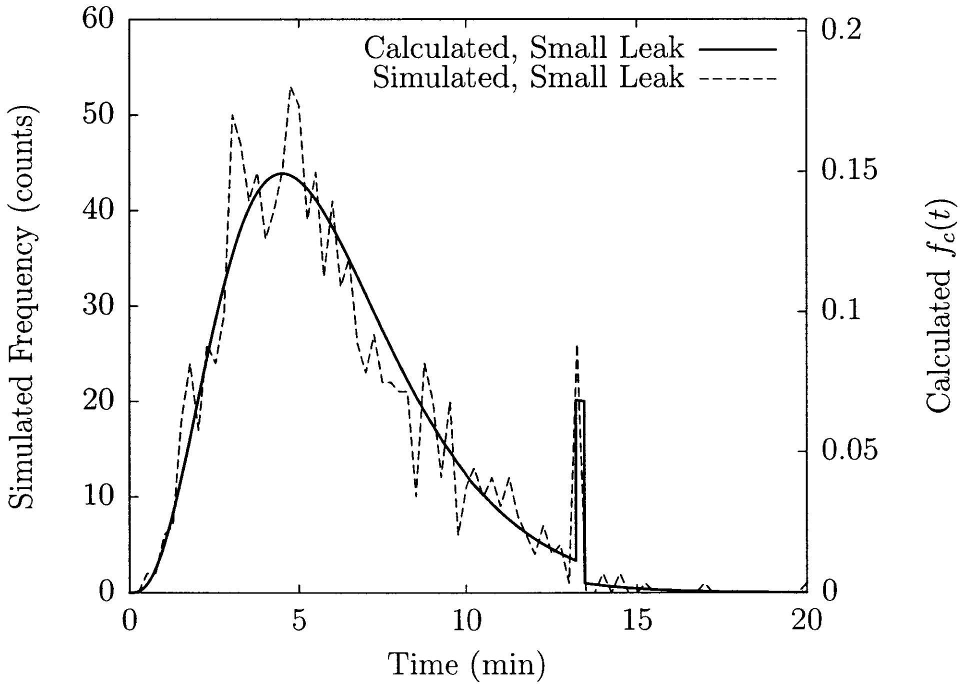


Figure 2-20: Calculated pump PDF plotted against simulated data. The system and leak parameters are the same as in Figure 2-13.

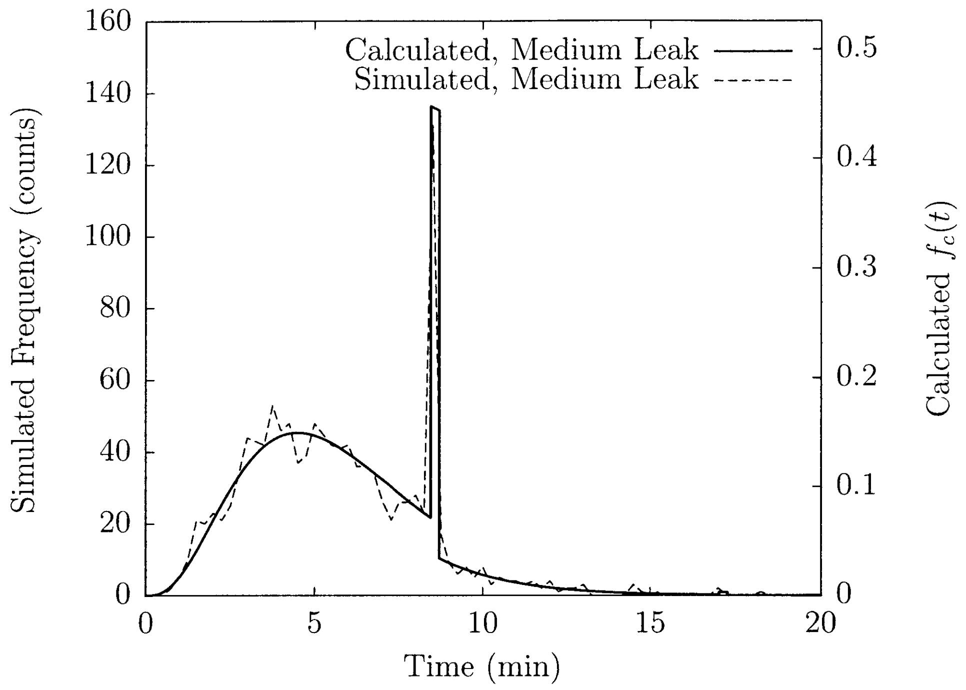


Figure 2-21: Calculated pump PDF plotted against simulated data. The system and leak parameters are the same as in Figure 2-14.

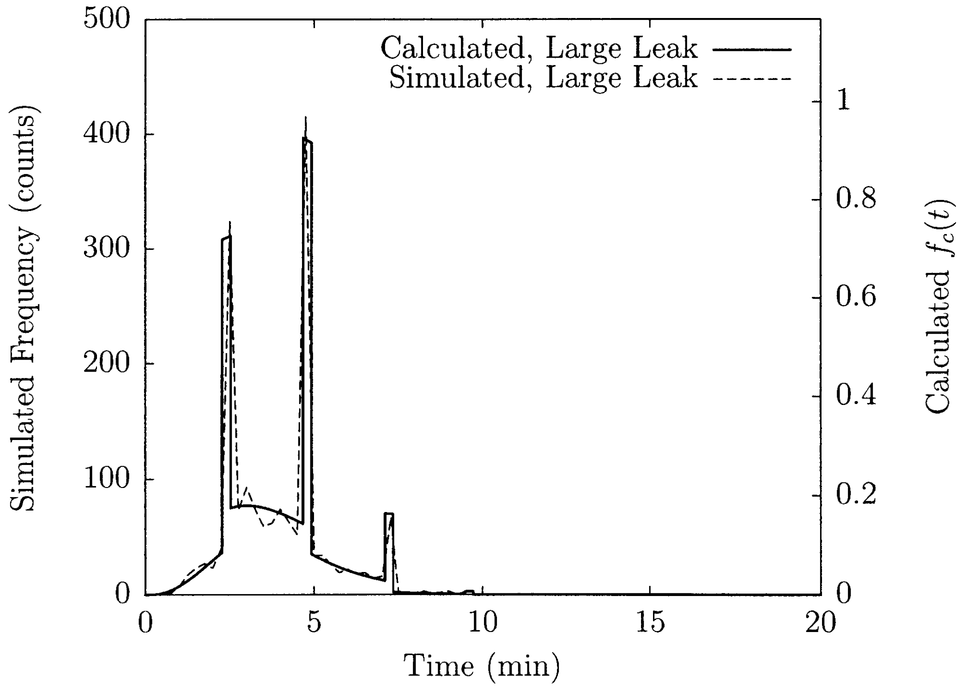


Figure 2-22: Calculated pump PDF plotted against simulated data. The system and leak parameters are the same as in Figure 2-15.

2.1.4 Indicator

As shown in §2.1.3, the simulation of the *SENECA* cycling system in §2.1.2 can be modeled using a piecewise function consisting of Erlang distributions and impulses. One method of using this as a diagnostic indicator is to perform a fit from the model $f_c(t)$ to the histogram of the recorded data in order to determine the τ, η, and λ parameters. The quality of such a fit relies on having low noise in the histogram, which can require collecting data for many days taken while the crew behavior is relatively constant.

A simpler method is to locate the peaks in the histogram caused by the leak. They will be located at approximately τ, 2τ, etc, and the first will typically be the largest and easiest to locate for leaks that are relatively low compared to crew usage. We can solve Equation 2.45 for R_{leak}:

$$R_{\text{leak}} = \frac{P_{\text{flush}}}{\tau} \tag{2.55}$$

Since the pressure drop per flush P_{flush} will be relatively constant, this indicates that the overall leak rate will be proportional to $1/\tau$. Furthermore, we can rearrange this equation to find the relationship between two different leak rates R_{leak1} and R_{leak2}, assuming a constant flush drop:

$$R_{\text{leak1}} \cdot \tau_1 = P_{\text{flush}} \tag{2.56}$$

$$R_{\text{leak2}} \cdot \tau_2 = P_{\text{flush}} \tag{2.57}$$

$$\frac{R_{\text{leak1}} \cdot \tau_1}{R_{\text{leak2}} \cdot \tau_2} = 1 \tag{2.58}$$

This method can be used to cross-verify the leak data that was presented in Figure 2-10, without necessarily knowing the P_{flush} or the system constant η. The locations of the peaks for the two leak cases are noted in Figure 2-23. For a leak rate of 30 SCFH, the corresponding peak is at $\tau_{30} = 12$. For a larger leak of 50 SCFH, the

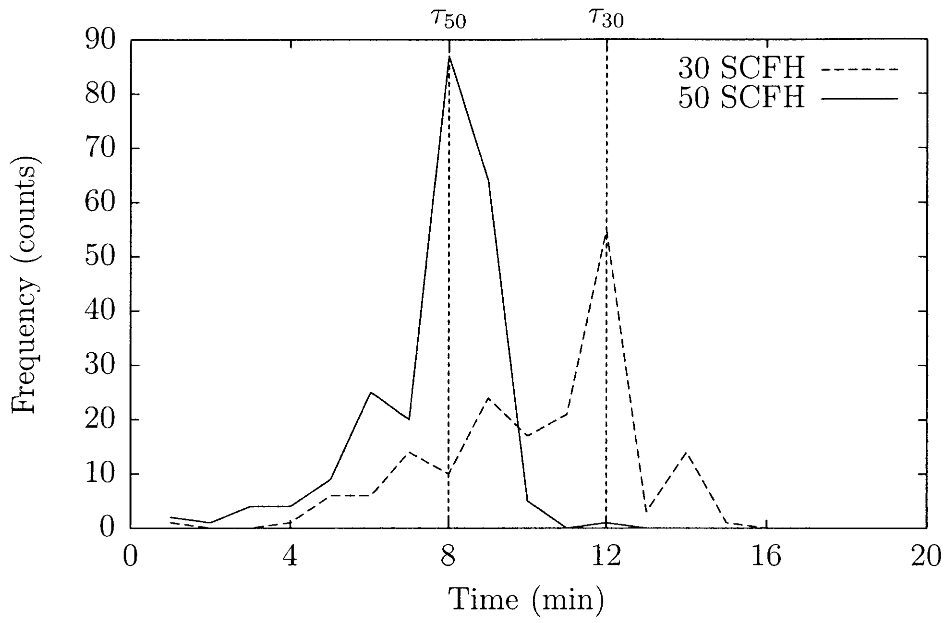


Figure 2-23: Plot of the the two leak cases from Figure 2-10, with the peak values indicated at $\tau_{50} = 8$ and $\tau_{30} = 12$.

peak moves to $\tau_{50} = 8$. The ratio from Equation 2.58 is therefore

$$\frac{30 \cdot \tau_{30}}{50 \cdot \tau_{50}} = \frac{30 \cdot 12}{50 \cdot 8} = 0.9$$

which indicates that our measured values of τ are relatively consistent with the measured leak rates. The small discrepancy can be explained by the presence of a pre-existing leak in the system while we conducted our tests. An existing leak of just 10 SCFH would explain this particular discrepancy.

The data in Figure 2-23 was taken during the night, when crew usage was minimal. The developed model for the system indicates that crew usage should not affect the location of the peak. To demonstrate this, we collected pump data from the *SENECA* during working and evening hours with a variety of introduced leaks. Figure 2-24 shows such data collected over approximately 48 hours with an added leak of 50 SCFH. While the general shape of the distribution has changed compared to the nighttime data, the location of the peak corresponding to the new τ has changed only slightly, as predicted.

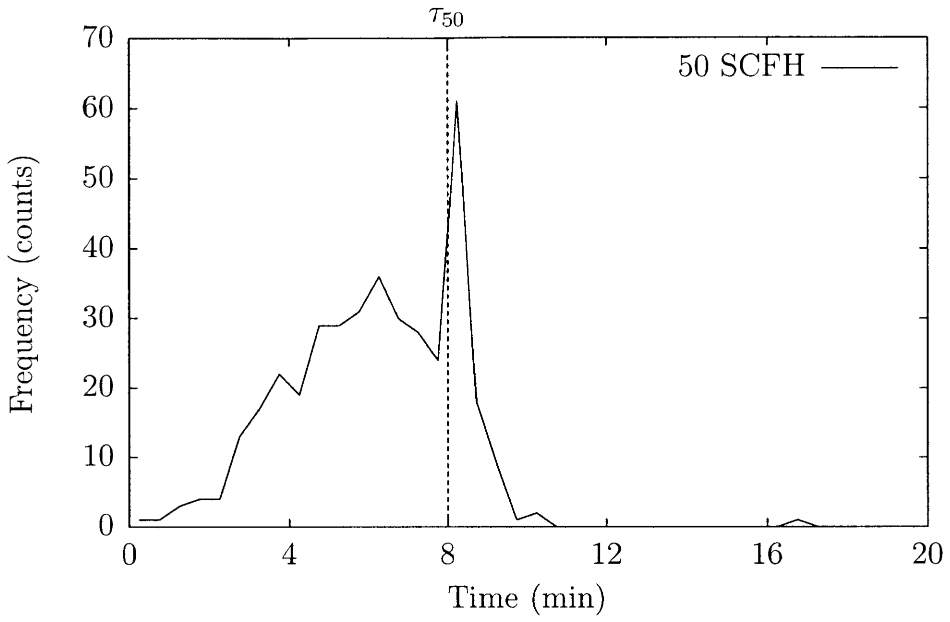


Figure 2-24: Data recorded on the *SENECA* during working and evening hours, when crew usage was high. The peak corresponding to the leak still shows up in nearly the same place despite the additional usage, as expected. The τ_{50} line from Figure 2-23 is shown for comparison.

2.1.5 Conclusions

The diagnostic indicator developed here for the *SENECA*'s vacuum system leads to two primary conclusions. The first is that electrical non-intrusive load monitoring and diagnostics can be applied to pneumatic cycling systems with excellent results. With statistical methods like those developed here, usage due to expected loads and unexpected leaks can be detected and differentiated without the need for additional instrumentation on the air system.

Second, this work demonstrates a number of necessary capabilities of the NILM framework. For example, the large amounts of test data collected from the *SENECA* required a disconnected system capable of unattended recording, storage, and transfer of data, which is presented in §4.2. The diagnostic itself, which applies new statistical methods on top of previously developed techniques like AC preprocessing and transient identification, demonstrates how new forms of analysis can be developed through the flexible use of existing tools. The software framework in Chapter 5 has been created with this in mind.

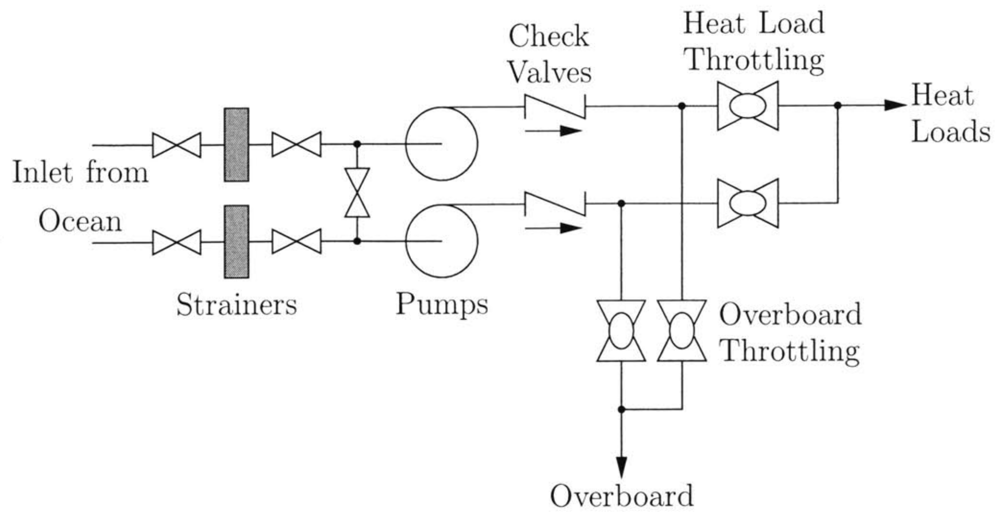


Figure 2-25: Diagram of the *SENECA* auxiliary seawater (ASW) system including the two primary pumps. From [6].

2.2 Mechanical Coupling Failure

The Auxiliary Seawater (ASW) system on the *USCGC SENECA* provides another informative example of the types of diagnostic indicators the NILM can create. The ASW pumps were experiencing premature failure of a motor coupling that was proving to be a maintenance burden [6], and so the NILM was employed to detect this coupling failure before it occurred. The methods developed for this diagnostic demonstrate a new class of frequency-based analysis for the NILM framework to support.

A diagram of the ASW system is shown in Figure 2-25. It has a pair of pumps, each driven by an induction motor in the configuration shown in Figure 2-26. The motors are connected to the pumps through a flexible rubber coupling to provide controlled failure in the case of blockage or other mechanical problems. These couplings were observed to tear and fail more often than expected, after only 8-13 pump starts [6]. To investigate these failures, the NILM was used to record voltage and current, and power spectral envelopes were computed using the preprocessor.

2.2.1 *SENECA* Data

Figure 2-27 shows the power transient of one pump start. It was observed that this startup transient contains a brief region of high-frequency oscillation as denoted in

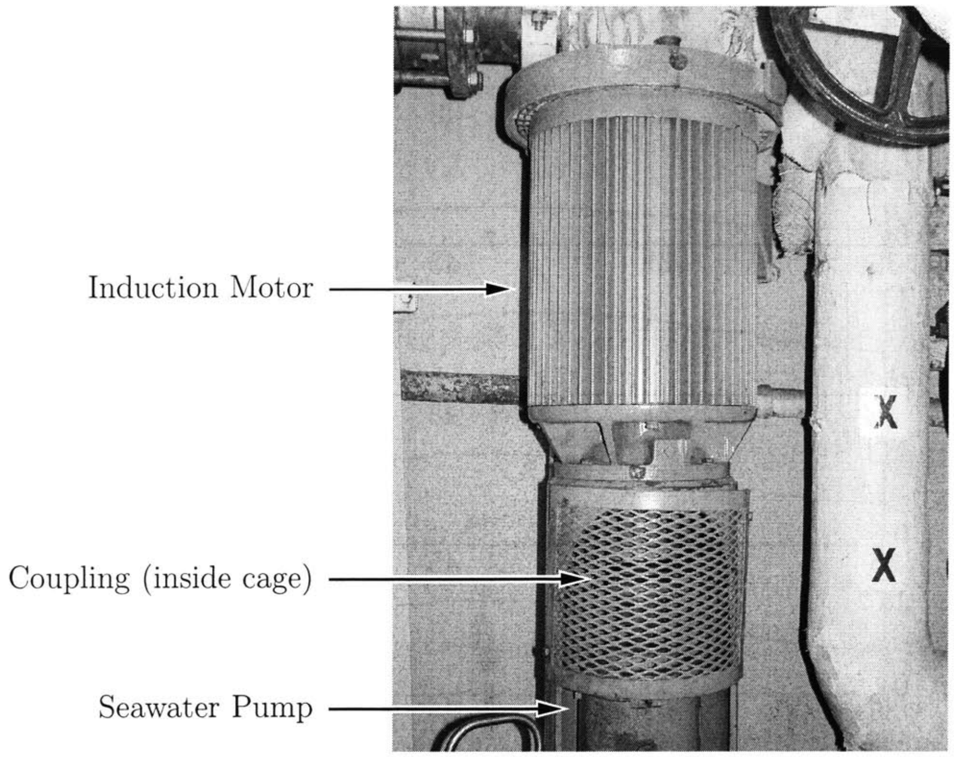


Figure 2-26: Photograph of one ASW motor and pump. A flexible rubber coupling connects the two in the middle. From [6].

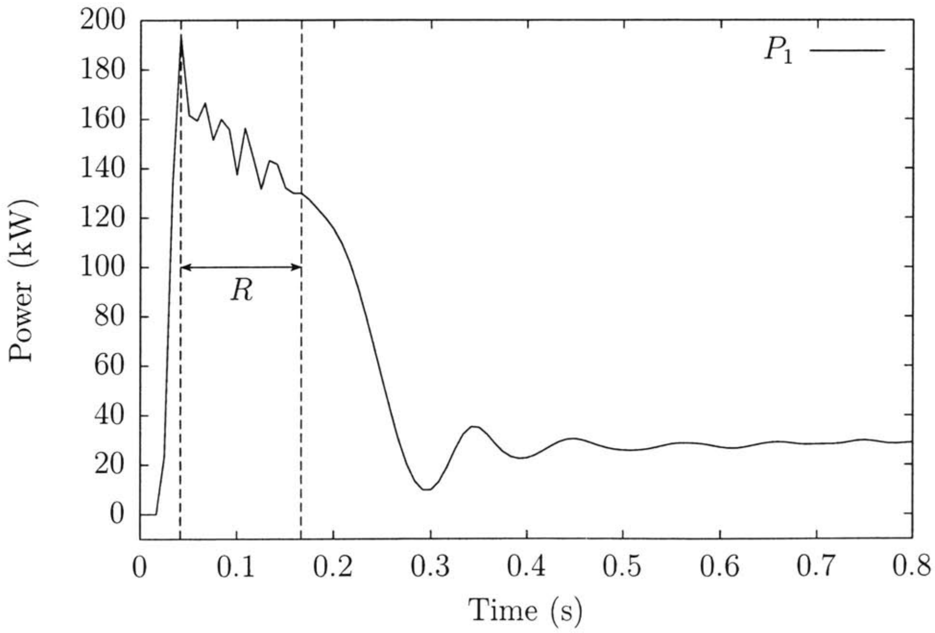


Figure 2-27: Power usage measured at the motor during the startup of one of the ASW pumps. A characteristic high-frequency oscillation is visible in region R near the start of the transient.

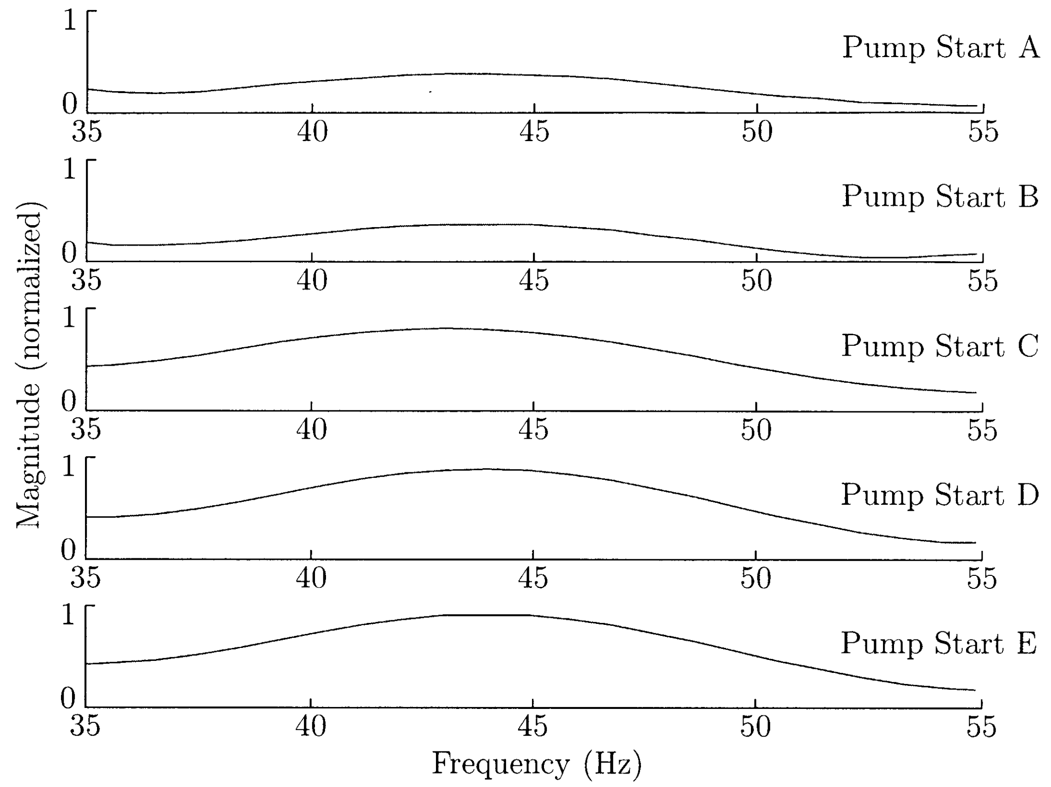


Figure 2-28: Plots of the spectral content of the high-frequency oscillation in five ASW pump starts. The magnitude increases around 44 Hz as the coupling progressively fails. From [7].

region R of the figure. Furthermore, empirical testing indicated that the magnitude of this "ripple" changed as the coupling degraded [7]. In order to better visualize this change, a 16-point data vector corresponding to the region R was extracted from a fixed offset in each startup transient. As detailed in [6] and [7], a 128-point windowed discrete Fourier transform (DFT) was then used to extract the frequency spectrum. This showed that the observed spectral content was concentrated around approximately 44 Hz.

Figure 2-28 shows the relevant portion of the frequency spectrum corresponding to five pump starts as the coupling progressively fails. Pictures of the physical coupling failure during these tests are provided in Figure 2-29. These results verify that the magnitude of the observed high-frequency oscillation does increase as the coupling fails. Furthermore, as the plots show, the frequency of the oscillation does not significantly change, which simplifies tracking of this trend.

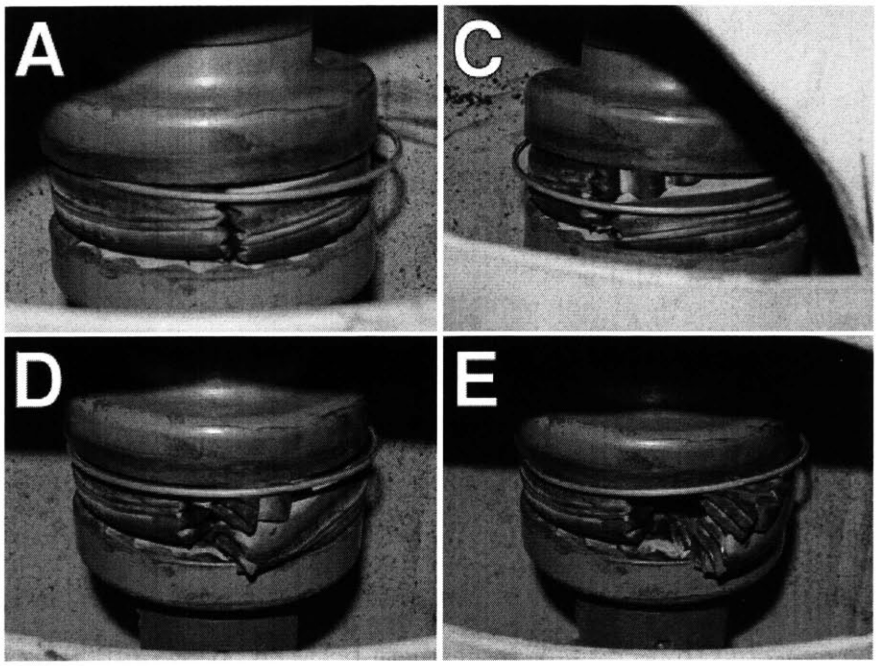


Figure 2-29: Photographs of the degrading ASW pump coupling during failure. The letters correspond to the pump starts shown in the spectral plots in Figure 2-28. From [7].

2.2.2 Simulation and Analysis

The spectral content around 44 Hz is a measurable quantity that appears to be strongly related to the physical state of the coupling between the induction motor and the pump. For this to be used as a reliable diagnostic indicator, it is important to verify that this effect is still present when the coupling fails in different ways and to determine the dependence on the motor and other system parameters. As with the cycling system in §2.1, this verification was performed through simulation.

The model used for the mechanical simulation is shown in Figure 2-30. The coupling is modeled as a damper and spring between two independent inertias. The derivation of the state equations based on this and an electrical model of an induction motor is detailed in [7], as are the particular parameters used for this simulation.

The simulation showed that the system behaves as predicted by earlier analysis. See [6] and [7] for complete results. Specifically, decreasing either the damping coefficient or the spring stiffness, which are effects that would be expected as the coupling fails, increases the magnitude of the peak seen in the spectral content of the

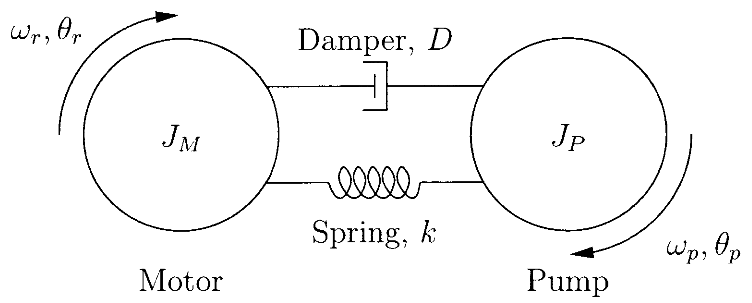


Figure 2-30: Model of the ASW motor, coupling, and pump system, as presented in [7].

power envelopes. Adjusting motor or pump parameters does cause this peak to move slightly, but it remains at an effectively constant frequency for any given motor and pump combination. Furthermore, it was determined from the simulation that the peak can be expected to be present somewhere between 30 Hz and 60 Hz on a typical system [6, 7].

2.2.3 Indicator

The final process for computing the indicator for the state of the mechanical coupling is as follows [7]:

1. Find the spectral envelopes of real power usage using the AC preprocessor as described in §1.1.2.
2. Locate the transients corresponding to pump starts using techniques as in §1.1.4.
3. Use the DFT to compute the spectral content $X[k]$ of a 16-point region following the start of the transient.
4. In $X[k]$, determine the frequency ω_p of peak spectral content between 30 Hz and 60 Hz.
5. Approximate the energy E contained in $X[k]$ in a small frequency band centered at ω_p.

This energy E is the diagnostic indicator, and larger values are an indication of higher coupling degradation.

2.2.4 Conclusions

The ASW coupling failure can be detected by a diagnostic indicator that, like the indicator for pneumatic cycling systems, utilizes interesting new techniques. In this case, we apply the Fourier transform to find the spectral content of the power usage, which itself is already the spectral envelope of the recorded signal. For the data recorded on the *USCGC SENECA*, this leads to a highly accurate indicator that can reliably predict coupling failure [7].

Furthermore, this "transform of a transform" technique may be useful for other systems exhibiting oscillations, such as variable speed drive control of fans. The process outlined for computing it also demonstrates some of the required functionality for the NILM framework, and suggests that a general-purpose language for specifying these indicators would be useful. The modules presented in §5.3 provide this ability by allowing for any external program, such as a Matlab or Octave script, to be used to process data.

Unlike the cycling system indicator, which requires many hours of recorded data and long-term statistical output, this indicator is computed from a single transient and can be returned immediately after a pump starts. The framework defined in Chapter 5 supports both of these cases equally well by not placing any particular restrictions on the amount of data processed or the rate of output of any particular indicator.

Chapter 3

Data Acquisition

Data acquisition is the first step of any monitoring and diagnostic system. In a typical NILM installation for AC data that will be passed to the preprocessor, we require a single voltage measurement and one current measurement per line phase. For software-based classification and diagnostics, a sampling rate of approximately 8 KHz per measurement is typical, and a resolution of at least 12 bits is desired. This resolution translates to approximately 80 mV per bit when sampling voltage from 120 volt AC service.

The hardware used for data acquisition has evolved with the computers that do the processing. Early work with software-based load monitoring and diagnostics at MIT used an Advantech PCL-818 acquisition card interfaced via the Industry Standard Architecture (ISA) bus of standard personal computers. The Peripheral Component Interconnect (PCI)-based Advantech PCI-1710, described below, was in use when this work was started. To support future computer hardware, a new Universal Serial Bus (USB) module was developed for this thesis and is described in §3.2 below.

3.1 PCI-1710 card

The Advantech PCI-1710 is a PCI card for a standard PC. It provides 16 single-ended channels or 8 differential channels of 12-bit analog input at an overall acquisition rate of 100 kilosamples per second. The card connects to an ADAM-3968 interface board,

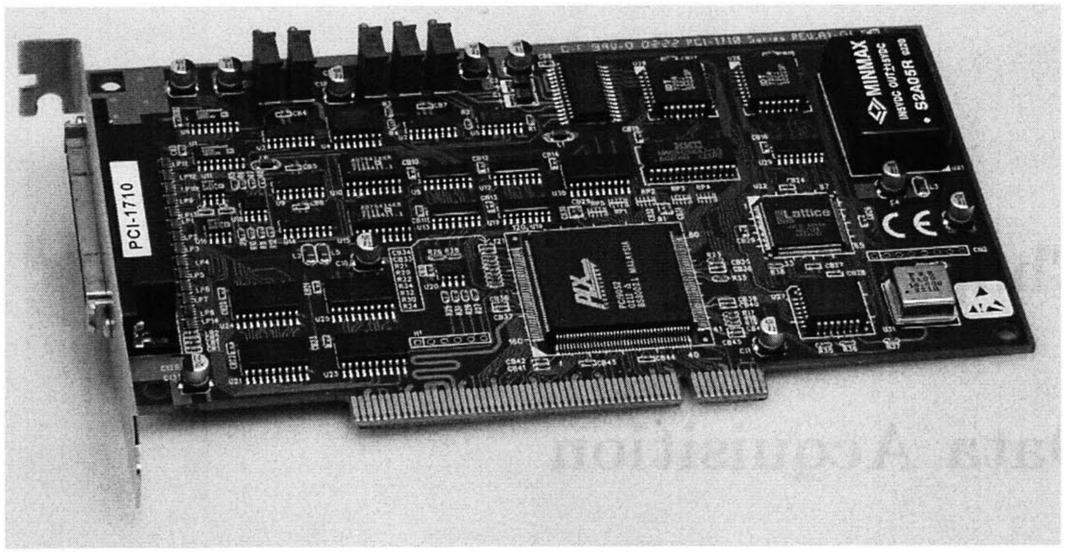


Figure 3-1: Photograph of the Advantech PCI-1710 data acquisition card, used to record voltage and current data into a desktop PC.

which provides terminal connections for each channel, through a 68-pin SCSI-II cable.

The software driver for the PCI-1710 provides a configuration and data interface for the Linux operating system through the `/dev/pci1710` character device node. The output from this device is the recorded data expressed as 16-bit signed integers. Typically, the program `pci2asc` is used to convert this raw data into a stream of ASCII numbers formed into columns, where each column contains numbers from 0 to 4095 representing the measured data. Minor additions and changes to the software were made as part of the development of the data recording system in §4.2 to improve driver support for the Linux 2.4 kernel.

Using this PCI card demonstrated a number of practical drawbacks. The most significant is that it requires a computer with an available PCI slot. This makes it difficult to use the card in small and application-specific computers that do not provide the same expansion capabilities as a desktop PC. It also precludes the use of a laptop, which would add convenience when doing initial testing in remote locations.

Other drawbacks of the PCI-1710 include the high hardware cost, the limited length of the SCSI connection cable, and the difficulty in using more than one acquisition card per computer. There is also a need to continuously update the drivers to track changes in the Linux kernel, and this has not been done for the latest kernel series, Linux 2.6. To address these issues, a custom USB-based module was developed as a replacement.

3.2 Custom USB Module

A custom data acquisition module, the USBADC, was developed in conjunction with Professor Steven R. Shaw at the Montana State University as a replacement for existing cards like the PCI-1710. This new module utilizes the common Universal Serial Bus (USB) interface, which is available on nearly every computer system currently available. New USB standards have remained backwards compatible with previous ones, indicating that USB is a stable interface for use with future systems as well.

Full-speed USB provides a bidirectional interface with a theoretical bandwidth of up to 12 Mbps. It supports up to 127 devices connected to one host, and provides up to 2.5 W of power to each device. USB cables are thin and flexible with a maximum length of 5 m, or 30 m with repeaters [5]. These features satisfy the requirements for NILM data acquisition well.

3.2.1 Hardware

Our USBADC hardware is shown in Figure 3-2. It consists of a Parallax (formerly Ubicom, Scenix) microcontroller, an FTDI USB interface module, and an Analog Devices ADC. A detailed board layout and circuit schematic are presented in Appendices B.1 and B.2.

The analog input is digitized by the Analog Devices AD7856, a 14-bit single supply analog to digital converter with a total throughput of 285 kilosamples per second. It supports eight single-ended analog inputs and communicates with the microcontroller through a serial interface. In this module layout, we expose the inputs of the ADC directly to the external header on the board, which means that the proper input range of the system matches the AD7856 range of 0-4.096 V. To allow for the highest signal quality, no additional protection circuitry was added to these lines, and so it

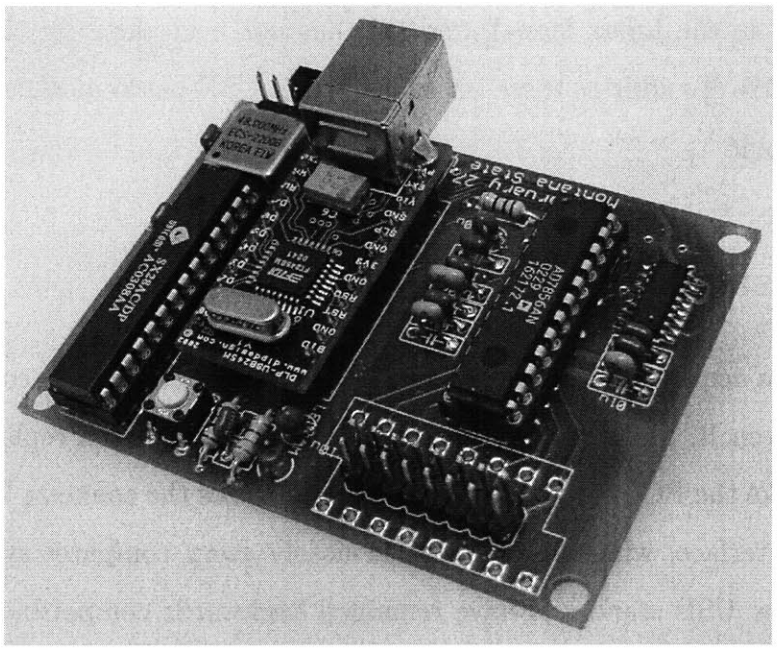


Figure 3-2: Photograph of the custom USB data acquisition module developed for use with the NILM. The hardware was manufactured and provided by Professor Steven R. Shaw of the Montana State University.

is important that applied voltages to the module do not go outside this range.

The Parallax SX28AC provides the logic and control for the board. This series of microcontrollers is based on the Microchip PIC architecture and supports clock speeds of up to 100 MHz. The SX28 has 20 bidirectional digital I/O lines and is programmed using an external interface connected to the clock input. Our module uses a 50 MHz oscillator and provides a connector to allow quick firmware reprogramming both during development and in the field. The firmware, described in §3.2.2, synchronizes and controls the sampling of the ADC and handles the data transfers between the ADC and the USB module.

USB communication is performed through the DLP-USB245M module from Future Technology Devices International (FTDI). This convenient interface is built around the FT245BM chip and implements much of the low-level USB protocol by providing a fixed function first-in, first-out (FIFO) interface for transferring simple streams of bytes. The physical DIP layout of the module abstracts away connector, layout, and noise issues associated with the USB physical interface. Using this module also avoids the cost and inconvenience of needing to apply for a custom USB vendor identifier (VID) from the USB Implementers Forum, as it comes preconfigured to use the FTDI VID. The DLP-USB245M communicates using an 8-bit parallel interface to the SX28 microcontroller and automatically buffers and packages outgoing data to conform with the USB Bulk Transfer specification. Signals are available to the microcontroller to indicate when the 384-byte transmit and receive buffers in the FT245BM are full or empty. The module also provides for a persistent identification tag to be set and queried through USB Control Transfers, which is useful for differentiating between multiple USBADC boards connected to a single machine.

3.2.2 Firmware

The firmware, listed in Appendix A.2.1 as `adc.asm`, receives commands from the computer via USB. Because the DLP-USB245M abstracts away the USB interface and provides a simple FIFO, the microcontroller sees a simple command stream consisting of sequential bytes of data. There are three primary states in which the firmware can be running.

Stop

This state is the default on power up and when the reset button is pressed. It is also entered on reception of the character `S` from the PC. The microcontroller will do nothing and wait for another command.

Benchmark

The character `B` begins a benchmark. Benchmarking is useful to test and verify the capabilities of the firmware, software driver, computer, and any hubs and repeaters, particularly when using multiple devices. In this mode, the microcontroller continuously sends a repeating string of 27 characters to the USB FIFO as quickly as it will accept them.

Command: `A` x y y	
`A`	Literal ASCII character 'A'.
x	Number of channels, ASCII '1' through '8'.
yy	Sampling divisor, as a two-character hexadecimal number.

Table 3.1: Format of the command sent to the USB ADC hardware to configure and start conversion.

Run

This is the primary mode of the device. The command to enter this mode is a four-byte string of ASCII characters following the format shown in Table 3.1. x indicates the number of ADC channels that will be read and returned. yy represents the sampling rate divisor, which must be nonzero. The actual per-channel sampling rate is computed as:

$$R = \frac{200 \text{ KHz}}{x \cdot yy} \tag{3.1}$$

The sampling rate can vary, therefore, from a maximum of $R = 200 \text{ KHz}/1 = 200 \text{ KHz}$ for a single channel, to a minimum of $R = 200 \text{ KHz}/(8 \cdot 255) \approx 98$ Hz per channel when sampling from all eight.

Upon entering the Run state, the firmware first sets up a timer-based interrupt to occur at rate R. The on-board LED is lit to indicate that acquisition has started. At each interrupt, the next analog channel from 1 through x is captured and converted. While conversion is in progress, the firmware avoids changing any outputs on the SX28 I/O lines in order to minimize potential interference and noise on the board. After conversion completes, the result is read out sequentially through the serial interface of the ADC.

Each 14-bit channel measurement is packed into two 8-bit bytes as shown in Figure 3-3. The highest bit is used to denote the pair corresponding to the first channel, for synchronization purposes on the computer. Each pair of bytes is then sent to the USB FIFO for transfer to the PC.

If the USB transmit buffer is full when the firmware has data to send, it will wait until the buffer empties. If the buffer is still full when an interrupt occurs, indicating

Byte 1

C	0	D_{13}	D_{12}	D_{11}	D_{10}	D_9	D_8

Byte 2

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0

C:	1 for the first channel, 0 otherwise
$D_{13} \dots D_0$:	14-bit data from the ADC

Figure 3-3: Data format used by the USBADC firmware to transfer data from the ADC to the PC.

that another conversion should start, the firmware will turn off the on-board LED to indicate that data has been lost. This condition indicates that the computer did not request to receive the data from the USB FIFO quickly enough. The driver software, described below, tries very hard to ensure that this does not happen.

3.2.3 Driver Implementation

One of the initial benefits expected from using the DLP-USB245M module was that software drivers already exist for a number of operating systems including Windows and Linux, as the FT245BM is a relatively common part. The drivers provide a virtual serial port that provides a transparent FIFO interface to the computer similar to that at the microcontroller.

Unfortunately, the existing drivers in Linux 2.4 and 2.5 turned out to be insufficient for our needs. In particular, the `ftdi_sio` driver was written with relatively low-speed uses in mind, and is not particularly careful regarding throughput and latency. Extensive benchmarking showed that the transfer performance varied greatly between versions of Linux and the model of USB host controller in the computer. A small selection of high-end external USB 2.0 hubs were found to sometimes improve the situation by splitting large USB high-speed requests from the computer and submitting multiple smaller full-speed requests to the module, but even this technique did not work on all tested systems.

To improve the stability and performance, a new custom software driver called `ftdi-adc` was written specifically for the USBADC. An in-kernel replacement for the existing `ftdi_sio` was considered, but due to the rapid development of Linux, it was apparent that the maintenance of this driver could pose difficulties for future systems. Instead, the open-source *libusb* library is used. Libusb provides a stable cross-platform interface that exposes all of the details of each USB transfer to programs running in userspace outside of the kernel, providing a large amount of flexibility in the processing that is done.

The main components of the `ftdi-adc` source code are listed in Appendix A.2.2. Particular implementation details are explained in the following sections.

USB Transfers

All USB transfers are initiated by the host. If a device like the USBADC has data to send, it must wait for a USB Request Block (URB) from the PC that requests data on the proper endpoint. Requests and transfers on a USB endpoint occur in discrete frames at regular intervals. For a full-speed USB chip like the FT245BM using bulk transfers, the frame interval is 10 ms and the data payload consists of up to 64 bytes per transfer per frame. Multiple transfers may occur in one frame.

When the host wants to request more than 64 bytes of data, the host controller interface (HCI) driver layer will divide the request into multiple transfers as necessary. For example, a request from the host for 640 bytes may be translated into ten 64-byte requests and sent in a single frame. Upon receipt of the data, the HCI will then combine the responses before returning them to the requesting program.

Even when sufficient bandwidth is available, the USBADC has a potential data loss condition that can occur between two requests, which is a major issue for the software driver. Recall from §3.2.1 that the transmit buffer of the FT245BM chip is 384 bytes in size. Consider the case where the command `A104` has been sent to the firmware, indicating that the module should begin sampling and returning one channel of data at 100 KHz. Each sample consists of two bytes, which means that the amount of time until the transmit buffer fills is

$$\frac{384 \text{ bytes}}{\text{buffer}} \cdot \frac{1 \text{ sample}}{2 \text{ bytes}} \cdot \frac{1 \text{ second}}{100 \text{ kilosamples}} = 1.92 \text{ ms / buffer} \tag{3.2}$$

In other words, the transmit buffer will fill, and data will be lost, if only 1.92 milliseconds pass between two requests from the computer for data. This places two conditions on the PC driver for reliable transfer:

1. There must be multiple URBs presented to the bus during each USB frame, since the fixed frame interval indicates that the next may not occur for up to 10 ms.

2. There can be little to no delay between the time when all existing URBs are fulfilled and new URBs are submitted, or an entire USB frame opportunity may be “missed”.

These goals are met by the driver through asynchronous URB submission and scheduling delay minimization, described below.

Multiple URB Submission

The functions for submitting URBs through libusb are synchronous, meaning that the program blocks while waiting for a response. This prevents the userspace driver from having multiple URBs reach the HCI at once, since it must wait for a previous response before sending a new one. Instead, the `ftdi-adc` program bypasses libusb for URB submission, and uses the underlying Linux `usbdevfs` driver directly. This virtual filesystem provides a stable interface to the HCI in much the same fashion as libusb, but supports asynchronous operations as well.

Up to 16 URBs are submitted by the driver at any given time. Each URB requests 4 kB of data, which gets further subdivided by the HCI to fit in the 64 byte maximum bulk transfer size. This large number of pending requests ensures that the host controller will be able to immediately respond to a completed transfer by beginning another. URBs are recycled and resubmitted immediately upon notification of their completion.

Delay Minimization

Even with multiple asynchronous URB submissions queued at the HCI, they will still eventually run out if they are not resubmitted by `ftdi-adc`. Most of the time, this is not a problem. However, all user-space programs are subject to process scheduling and preemption in the Linux kernel, which means that the kernel can switch to another kernel or user process at any time. Worse, there are no guarantees of when a stopped process will be resumed. Factors such as I/O load and dynamic metrics of process behavior are considered by the kernel, and scheduling delays can easily range from microseconds to seconds or longer. For a userspace program like the USBADC driver, this problem cannot be fully solved, but there are a two main techniques that are used to minimize the delays and the subsequent USBADC data loss they might incur.

The first is to increase the process priority significantly. Linux provides the ability to assign processes to one of a number of scheduling queues in the kernel. The driver utilizes `SCHED_FIFO`, which provides the most aggressive scheduling available. Placing a process in this group ensures that it will always preempt all other user processes when it needs to run.

Secondly, the USB URB processing is fully separated from the data output. The driver runs a second thread at normal priority that handles all output to disk or other programs. The incoming USB data is sent to this thread using shared memory segments protected by mutual exclusion locks. The default size of this memory buffer is 4 MB, which is sufficient to handle even long I/O delays.

Results

Using these techniques, `ftdi-adc` driver works very well. Due to the use of the `usbdevfs` driver and aggressive scheduling policies, only Linux systems are supported, but this does not pose a problem with the systems currently in use. It has been tested

Option	**Description**
`--id` *string*	ID of which USBADC to use. If not specified, the first available is chosen.
`--set-id` *string*	When only one USBADC is connected, set its ID to the given *string* and exit.
`--list`	List all detected and available USBADC devices and exit.
`--command` *string*	Initial command to send the device. Overrides the `Axyy` default.
`--channels` *1 - 8*	Number of channels of data to sample and receive.
`--divisor` *1 - 255*	Sampling rate divisor.
`--retry`	Keep retrying if the device is disconnected. This allows the USBADC to be temporarily unplugged if necessary.
`--verbose`	Print additional diagnostic and informative messages to `stderr`.

Table 3.2: List of the command-line options supported by the `ftdi-adc` program.

to work on Linux 2.4 and 2.6 series kernels, and is expected to work without change on future kernels. It handles multiple devices through the use of persistent, configurable device ID strings, and works under moderately heavy system load without losing data. If necessary, the buffer sizes and submitted URB count can be increased to provide even more protection against data loss.

3.2.4 Driver Usage

The `ftdi-adc` program uses the command-line arguments shown in Table 3.2 to control its behavior. Output is sent directly to the `stdout` stream in the 16-bit format shown in Figure 3-3. An associated program named `convert`, listed in Appendix A.2.3, is typically used as a filter to convert this binary format into a columnated ASCII form similar to that of `pci2asc` for the PCI-1710. The driver will continue to receive and output data until the process is terminated by a keyboard interrupt or other signal, or until the USB hardware is disconnected.

An example typical invocation of the program is:

```
ftdi-adc --channels 2 --divisor 10 --verbose | convert > raw.txt
```

This will record two channels of data at a rate of 10 KHz each, convert the result into a two-column list of ASCII numbers, and write the output to the file `raw.txt`.

Chapter 4

Support Software

A number of support software components were developed as part of this work to facilitate data collection, processing, and storage, as well as to demonstrate the potential of the NILM system. Along with the driver software written for the data acquisition hardware discussed in Chapter 3, this software provided the primary environment for the majority of the work presented in this thesis.

4.1 Preprocessor

The preprocessor, `prep`, is the program that computes spectral power envelopes from the raw current and voltage measurements as described in §1.1.2. The basis of the preprocessor is the code written by Professor Steven R. Shaw of the Montana State University and presented in [13].

The architecture of the existing `prep` contained a number of issues that made it inconvenient to use. In particular, the program relied on compiled binary modules dynamically linked at run-time to perform input and data processing. While this allowed for the input type and processing to be changed easily, it also complicated the build process, reduced portability, and required that multiple binary objects be kept together and installed wherever the package was used. Another design drawback of the program is that certain mathematical constants in the spectral envelope computation were hard-coded and required recompilation from source to change.

Specific changes were made to the preprocessor to address these issues and to match observed typical usage. The major areas that were modified are outlined in the following three sections.

4.1.1 Consolidated Text Input

The existing hardware-specific input methods were removed from the preprocessor. Instead, a single input method was created that takes ASCII-formatted numerical data. Each text-based line of input contains two floating-point values representing the instantaneous measured voltage and current. A fixed offset is accepted and the zero point can be specified as a command-line option. Both the PCI-1710 and USBADC data acquisition hardware drivers in Chapter 3 can be configured to present output in this form.

Besides simplifying the code in the preprocessor, text input also provides a more convenient interface for processing data generated by other programs like the ASW pump simulation in §2.2.2, and allows the preprocessor to fit more easily into the framework of Chapter 5. For backwards compatibility with existing installations, an option to handle the binary data format of the PCI-1710 data acquisition card is provided.

4.1.2 Build System Improvements

The requirement that the input and processing code be provided in a separate dynamically linked object was removed. With the consolidated input method, only one function is needed to process input data, and so `prep` is now built as a single binary containing the frontend, input, and processing components. This does limit the flexibility of any particular build of `prep` with regards to future preprocessing methods. However, the built binary is very small on a typical Linux system, and multiple versions can be developed for specific tasks if this is found to be necessary. This provides similar functionality to the previous implementation, is more straightforward, and allows for better control of algorithm-specific configuration options.

Option	Description
`--harm` n	Number of odd harmonics for which to compute spectral envelopes.
`--basis` n	Number of points in basis (internal parameter).
`--inc` n	Number of points to increment (internal parameter).
`--rows` n	Number of rows in matrix (internal parameter).
`--lines` n	Process n lines of output, then exit.
`--annotate`	Annotate the output with the line numbers of the corresponding input.
`--pci1710`	Use the old PCI-1710 binary data format.
`--zero` v	Zero point for text input format.

Table 4.1: List of the command-line options available to the `prep` program.

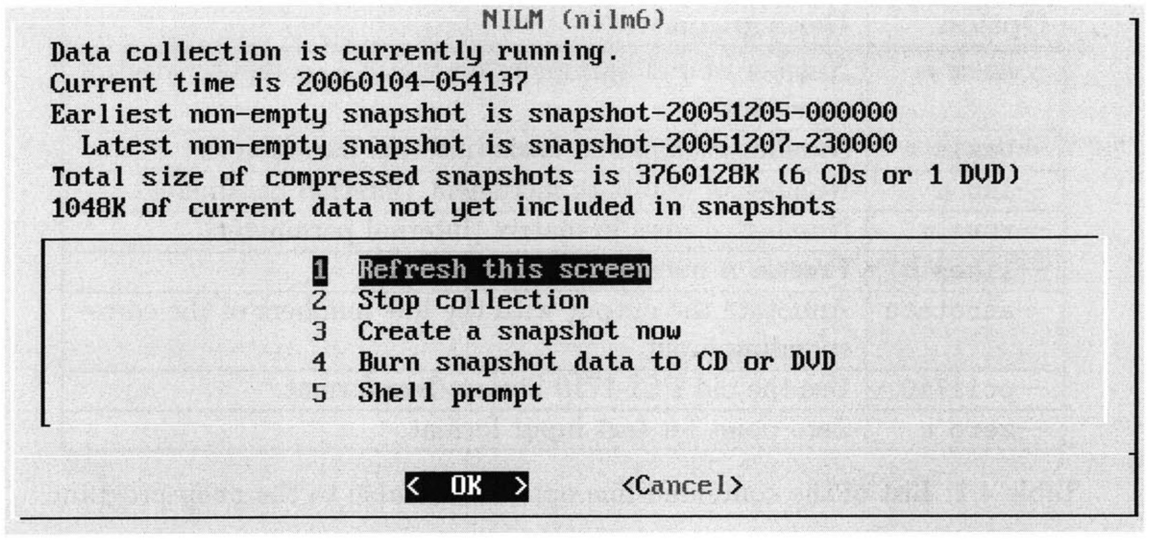
The custom build system was also replaced with a system based on the GNU Autotools (Autoconf, Automake, and Libtool). This change allows for easier integration with existing packaging systems, provides features to allow better customization of where and how the program is built and installed, and improves portability. For example, this new build system will allow `prep` to be more easily cross-compiled for a Windows system from a Linux host.

4.1.3 Improved Command-Line Options

Several constants that were previously hard-coded in the binary are now exposed as command-line options. These include the number of harmonic indices for which to compute spectral envelopes and other tunable internal parameters. It is expected that future configurable aspects of the preprocessor will be exposed through command-line options rather than requiring changes to the source code, when possible, in order to better facilitate flexible preprocessing by the NILM framework. The current command-line options are shown in Table 4.1.

4.2 Data Logging

Development of NILM diagnostics and concepts often requires large amounts of real-world data for analysis and cross-verification. An early problem identified as part of



```
                         NILM (nilm6)
Data collection is currently running.
Current time is 20060104-054137
Earliest non-empty snapshot is snapshot-20051205-000000
  Latest non-empty snapshot is snapshot-20051207-230000
Total size of compressed snapshots is 3760128K (6 CDs or 1 DVD)
1048K of current data not yet included in snapshots

                  1  Refresh this screen
                  2  Stop collection
                  3  Create a snapshot now
                  4  Burn snapshot data to CD or DVD
                  5  Shell prompt

                 <  OK  >        <Cancel>
```

Figure 4-1: Screenshot of the main menu-system interface.

this work was that such long-term data collection was often difficult to perform due to the wide variety of systems and conditions in which they were being monitored. Indicators like those developed in Chapter 2 on the *USCGC SENECA* may require months of reliable unattended data collection, while experiments on system models in the lab may require more interactive control of the collection process. To this end, a menu-driven system was created to automate and simplify data acquisition while providing a suitable level of interactive control.

4.2.1 Overview

The `menu-system` software package, listed in Appendix A.3.1, is a set of scripts and modifications to a standard Linux system to facilitate data collection, processing, storage, and retrieval. The software utilizes the preprocessor in §4.1 and the PCI-1710 data acquisition hardware in §3.1.

The user interface is based around the full-screen menu shown in Figure 4-1. This menu is created by the shell script `menu-system` and is based on interface components provided by the open-source software project *zenity*. The menu is loaded automatically at boot, and allows the user to both monitor and interact with the system via the keyboard. Specific features are detailed in the following sections.

Identifier	Description
`start`	Started recording preprocessed data.
`startraw`	Started recording raw data.
`stop`	Stopped recording data.
`reopening`	About to close this file and reopen it.
`reopened`	Reopened this output file. Continuing to record.

Table 4.2: List of the possible event identifiers in the comments inserted into recorded data streams by `run-prep.pl`.

4.2.2 Data Acquisition and Storage

The user may choose to record raw voltage and current data, or to pass the incoming data through the preprocessor and store the spectral envelope data instead. Recording preprocessed data is generally preferred when applicable because the storage requirements are an order of magnitude lower.

Data recording and processing is performed as a background process controlled by the Perl script `run-prep.pl`. This script collects data from the PCI-1710 card, optionally executes the preprocessor, and stores the output to a standard ASCII text file. The process can be controlled from the `menu-system` script through signals that cause it to start and stop collection and switch output to a new file.

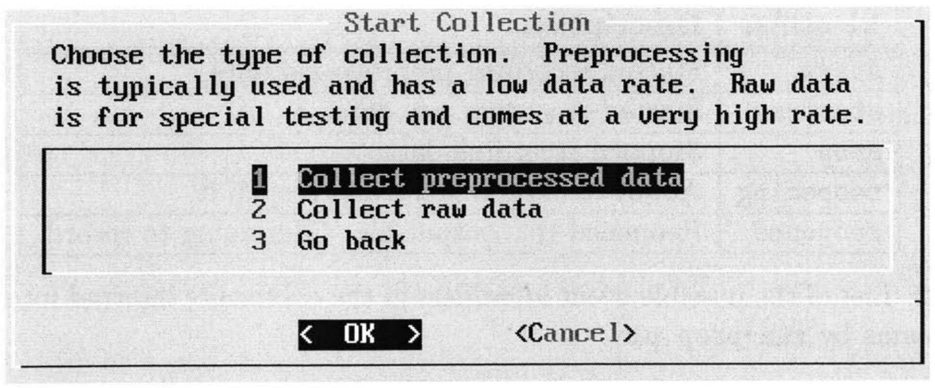
The generated data file is annotated with timestamps and comments indicating the source of data as well as any changes that may affect continuity. Each comment is formatted as follows:

```
# <hostname> <event> = <timestamp>
```

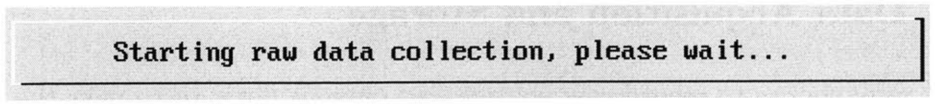
where *<hostname>* is the name of the computer that recorded this file, *<timestamp>* is the date and time that this comment was added, and *<event>* is one of the identifiers described in Table 4.2.

The primary purpose of these comments is to clarify later analysis by reducing misidentification of data. They also provide a complete record of every time collection was stopped or started, which can assist greatly in determining how data may have been affected by unexpected power outages and other temporary failures.

The `reopening` and `reopened` events in Table 4.2 are caused by an external signal.



```
                    Start Collection
Choose the type of collection.  Preprocessing
is typically used and has a low data rate.  Raw data
is for special testing and comes at a very high rate.

           1  Collect preprocessed data
           2  Collect raw data
           3  Go back

           < OK >        <Cancel>
```



```
Starting raw data collection, please wait...
```

Figure 4-2: Sequence of screenshots demonstrating control of the `run-prep.pl` data collection process through the interactive menus. The prompts are designed to easily guide inexperienced users who may need to control systems installed in the field.

This signal indicates that `run-prep.pl` should close the current data file and reopen it, without pausing or stopping data collection, which allows for accurate archival. The snapshot script described in §4.2.3 makes use of this functionality.

Figure 4-2 shows a typical sequence of starting data collection interactively. The first line of the main menu in Figure 4-1 indicates the current state of data collection.

4.2.3 Snapshots

A "snapshot" is an archive containing of all of the data stored by `run-prep.pl` for a given length of time. These snapshots are the files that are periodically retrieved from the system for later processing. They are an important part of the collection process because they ensure that the current recorded data file does not grow unmanageably large. Furthermore, snapshots are individually compressed to reduce storage requirements by a factor of 3 or more.

Snapshots are generated by the `snapshot-data` script. This script renames the current data file to the form `snapshot-`*`YYYYMMDD`*`-`*`HHMMSS`* denoting the date and time at which the snapshot was taken. Then, `run-prep.pl` is signalled to indicate that it should close and reopen its output, causing it to stop writing to this file and begin writing to a new one. Finally, the snapshot data is compressed with `gzip` or `bzip2` and stored in the `data` subdirectory for later retrieval.

By default, snapshots are generated at the top of every hour by a scheduled `cron` job. They can also be created at any time by interactive request from the menu. The comments embedded in the recorded data, described in §4.2.2, resolve any ambiguity that may arise from sporadic or inconsistent snapshots.

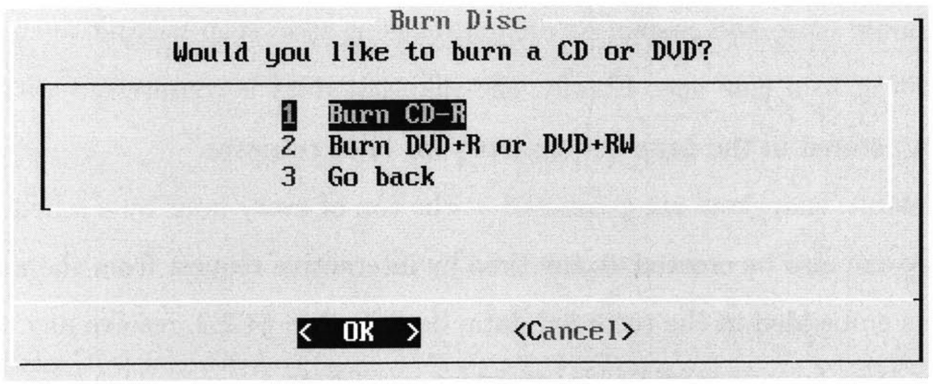
The main menu in Figure 4-1 displays information pertaining to snapshots. This includes the current date and time, the approximate date range of snapshot data stored on this computer, the total amount of data stored in snapshots, and any data that may have been recorded but not yet placed in a snapshot. It also provides an estimation of the number of optical media disks required to retrieve the snapshot data.

4.2.4 CD and DVD Burning

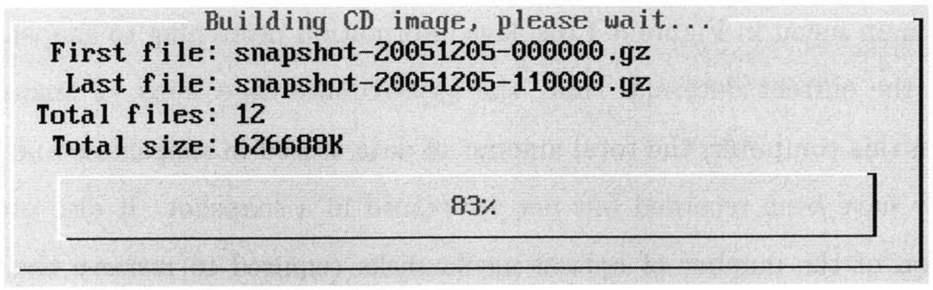
A primary feature of the menu system is to facilitate and simplify data retrieval. The monitoring systems we have installed include a CD-R or DVD+R(W) drive to which snapshot data can be copied. In some cases, this is done to provide rapid access to the data for analysis, but a more typical use is to account for storage limitations on the PC, which may only have the space to hold a few weeks' or months' worth of data.

The script `burn-cd` handles the selection of an appropriate number of snapshot files, the building of a disk image, and the process of burning the image to disk. To simplify operation, the data to burn is automatically chosen starting with the oldest snapshot data. Screenshots of the process of burning a CD through the interactive menus are shown in Figure 4-3.

Once a disc has been burned, an option is provided to burn additional copies of the same data. Then, the recorded snapshots are removed from the data directory and placed in a temporary queue that is automatically purged when disk space runs low. This temporary queue has been found to be useful for manual data recovery in



```
Burn Disc
Would you like to burn a CD or DVD?

1  Burn CD-R
2  Burn DVD+R or DVD+RW
3  Go back

< OK >        <Cancel>
```



```
Building CD image, please wait...
 First file: snapshot-20051205-000000.gz
  Last file: snapshot-20051205-110000.gz
Total files: 12
 Total size: 626688K

                 83%
```

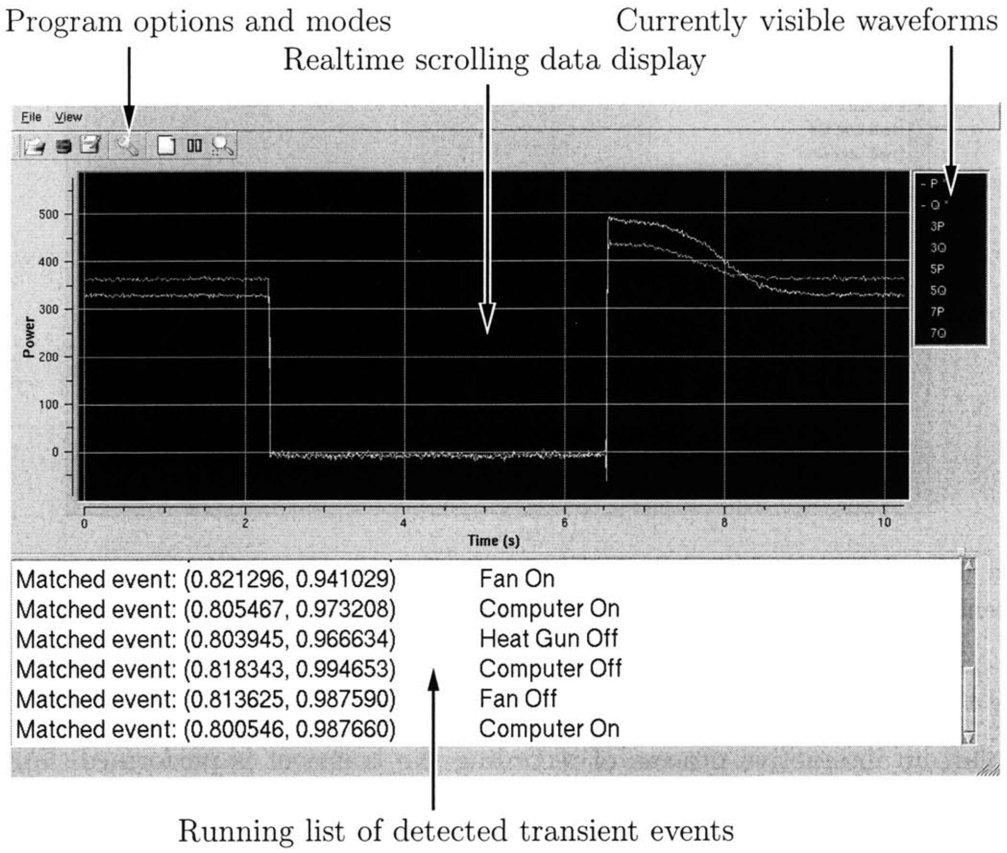
Figure 4-3: Sequence of screenshots demonstrating the automated burning of snapshot data to CD through the `burn-cd` script.

the case of lost CDs or other unexpected situations.

In practice, the CD and DVD burning abilities have proven to be very useful. Data collected from the *USCGC SENECA* while underway, for example, was periodically written to CD by crew members and returned via mail for analysis. The software has been successfully used to collect hundreds of gigabytes of preprocessed and raw data from various systems.

4.3 Graphical Demonstration

The graphical demonstration software first introduced in §1.1.5 is a tool for visualizing the process of capturing data and identifying loads. The primary goal of this software is to introduce NILM concepts and features within a clean interactive interface that automates and simplifies the transient-based training and identification described in §1.1.4. Incoming data from a hardware acquisition card is classified based on trained exemplars, and the software provides indication of any recognized events.


Program options and modes
Realtime scrolling data display
Currently visible waveforms
File View
P
Q
3P
3Q
5P
5Q
7P
7Q
Power
Time (s)
Running list of detected transient events

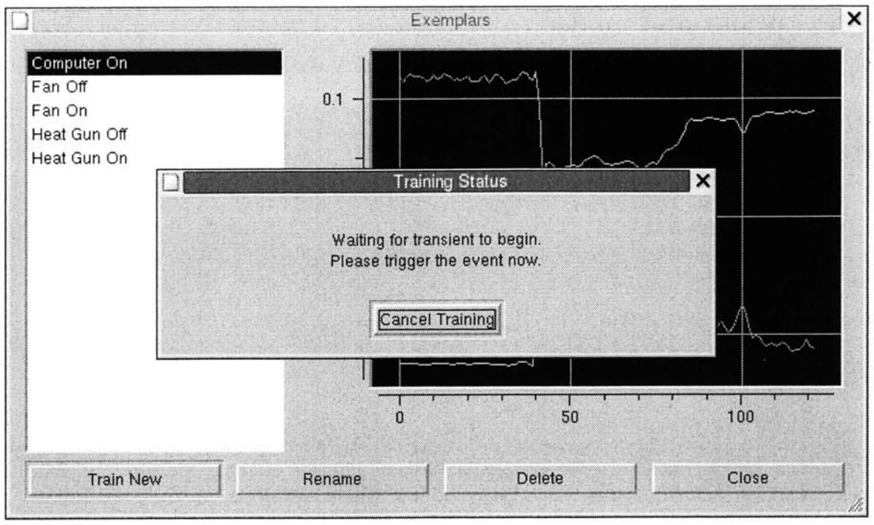
```
Matched event: (0.821296, 0.941029)      Fan On
Matched event: (0.805467, 0.973208)      Computer On
Matched event: (0.803945, 0.966634)      Heat Gun Off
Matched event: (0.818343, 0.994653)      Computer Off
Matched event: (0.813625, 0.987590)      Fan Off
Matched event: (0.800546, 0.987660)      Computer On
```

Figure 4-4: Main window of the `nilmgui` graphical demonstration program.

These events typically correspond to the turning on or off of some particular load, but can more generally be associated with any unique rapid change in the power consumption of some device.

Figure 4-4 highlights the features of the main `nilmgui` window. The software performs capture, preprocessing, and real-time display of power envelopes in the top half of the screen. These waveforms scroll to the left as new data is added to the right, similar in fashion to a continuous strip chart recorder. In the lower half of the screen, a text display provides a running log of detected matches between the incoming data and previously recorded transient exemplars. Matches are determined using the method described in §4.3.1, and matching is performed simultaneously against all existing exemplars.

These recorded exemplars can be viewed, created, and modified from within the "Exemplars" dialog, shown in Figure 4-5. This dialog provides the user with the



Waiting for transient to begin.
Please trigger the event now.

Figure 4-5: Screenshot of the "Exemplars" dialog in the `nilmgui` software. Five transients have been recorded, and guided training of a sixth is in progress.

ability to view, rename, and delete existing exemplar waveforms. To create a new exemplar, an interactive process of recording the transient is performed. First, the user selects the "train new" option and provides a name. The `nilmgui` software waits for the incoming data signal to settle, and the user is prompted to trigger the desired event by, for example, physically turning the device on or off. Then, when a transient is detected, the software again waits for the signal to settle. Finally, the software automatically extracts a small region around the detected power transient and stores it as a new exemplar. Once the system has been trained in this fashion, the recorded exemplars can be saved to disk for later use.

4.3.1 Implementation

The code for the NILM graphical interface is provided in Appendix A.3.2. It is written in C++ and utilizes the Trolltech *Qt* toolkit library. The scrolling plot that displays the current power waveforms is based on functions provided by the Qt Widgets for Technical Applications library, *Qwt*.

Input Plugins

The `nilmgui` software utilizes a plugin architecture for data input. Each plugin, which is a shared object module linked by `nilmgui` at run-time, provides a structured interface for configuration and data transfer. The current systems utilizing this software are based on the `cmdline` plugin, which generates data using any existing command-line program as the source. Through this plugin, the data acquisition hardware and software described in Chapter 3 can be used. The command to execute is provided through the plugin configuration interface, allowing it to be changed without recompilation.

Transient Matching

As data arrives from the input plugin, the P_1 and Q_1 waveforms are individually matched against each of the transient exemplars by computing metrics based on the least-squares error. The incoming waveform is first used to create a data vector $\vec{x}$ of the same length n as the corresponding exemplar vector $\vec{e}$. The least-squares error E between the two is defined as

$$E = \sum_{i=1}^{n} (x_i - e_i)^2 \tag{4.1}$$

This expression can be expanded and then separated:

$$E = \sum_{i=1}^{n} (x_i^2 - 2x_i e_i + e_i^2) \tag{4.2}$$

$$= \sum_{i=1}^{n} x_i^2 - 2\sum_{i=1}^{n} x_i e_i + \sum_{i=1}^{n} e_i^2 \tag{4.3}$$

The norm of a vector $\vec{x}$ is defined as

$$|\vec{x}| = \sqrt{\sum_{i=1}^{n} x_i^2} \tag{4.4}$$

and the dot product between vectors $\vec{x}$ and $\vec{e}$ is defined as

$$\vec{x} \cdot \vec{e} = \sum_{i=1}^{n} x_i e_i \tag{4.5}$$

Substitute Equations 4.4 and 4.5 into 4.3 to obtain:

$$E = |\vec{x}|^2 + |\vec{e}|^2 - 2\vec{x} \cdot \vec{e} \tag{4.6}$$

Instead of computing this error term directly, we can first normalize the vectors $\vec{x}$ and $\vec{e}$:

$$\hat{x} = \frac{\vec{x}}{|\vec{x}|}, \qquad \hat{e} = \frac{\vec{e}}{|\vec{e}|} \tag{4.7}$$

These new vectors $\hat{x}$ and $\hat{e}$ have magnitude 1, and so we can compute the error term for the normalized vectors as

$$\begin{aligned} E &= |\hat{x}|^2 + |\hat{e}|^2 - 2\hat{x} \cdot \hat{e} & (4.8) \\ &= 1 + 1 - 2\hat{x} \cdot \hat{e} & (4.9) \\ &= 2 - 2\hat{x} \cdot \hat{e} & (4.10) \end{aligned}$$

Define the "score" s as

$$s = \hat{x} \cdot \hat{e} \tag{4.11}$$

Substitute s into Equation 4.10 to obtain

$$E = 2 - 2s \tag{4.12}$$

This indicates that the error E decreases as s increases; that is, larger scores indicate a closer correlation between the two input data vectors. The maximum value of s is equal to the maximum dot product between two unit vectors, which is $s = 1$, and this corresponds to a perfect match in shape between the incoming data and the exemplar.

The score s is computed using normalized vectors. This metric is therefore independent of magnitude differences, and is effectively comparing only the shape of the exemplar and the incoming data. To account for magnitude, we also consider the ratio r of the magnitudes of the original vectors:

$$r = \begin{cases} |\vec{x}|/|\vec{e}| & \text{if } |\vec{x}| \leq |\vec{e}| \\ |\vec{e}|/|\vec{x}| & \text{if } |\vec{x}| > |\vec{e}| \end{cases} \tag{4.13}$$

As with s, larger values of r indicate a better match, with equal magnitudes at $r = 1$.

The matching algorithm in `nilmgui` performs this computation of score and ratio for every exemplar at every data point of the input. Each metric is computed twice, once for each of the P_1 and Q_1 waveforms. The scores s_P and s_Q are then combined to the single weighted score s_w based on the relative magnitudes of P_1 and Q_1. Similarly, the weighted ratio r_w is calculated from r_P and r_Q.

Using a heuristic, these weighted values s_w and r_w are then compared to a configurable threshold. An exemplar match is considered to have occurred if both values are above the threshold. The heuristic also allows for one value to be slightly below the threshold as long as the other is slightly above it. For example, with a threshold of 0.95, matches may be considered to occur if any of the following three conditions are met:

(1)	(2)	(3)
$s_w > 0.97$	$s_w > 0.95$	$s_w > 0.93$
$r_w > 0.93$	$r_w > 0.95$	$r_w > 0.97$

Further details of this heuristic can be found in `classifier.cpp` in Appendix A.3.2.

Once suitable exemplar matches have been found, the match with the highest weighted score s_w and ratio r_w is displayed on the main NILM window along with the name of the matching exemplar. Additional matches are then inhibited for a short time to prevent spurious matches against the same data.

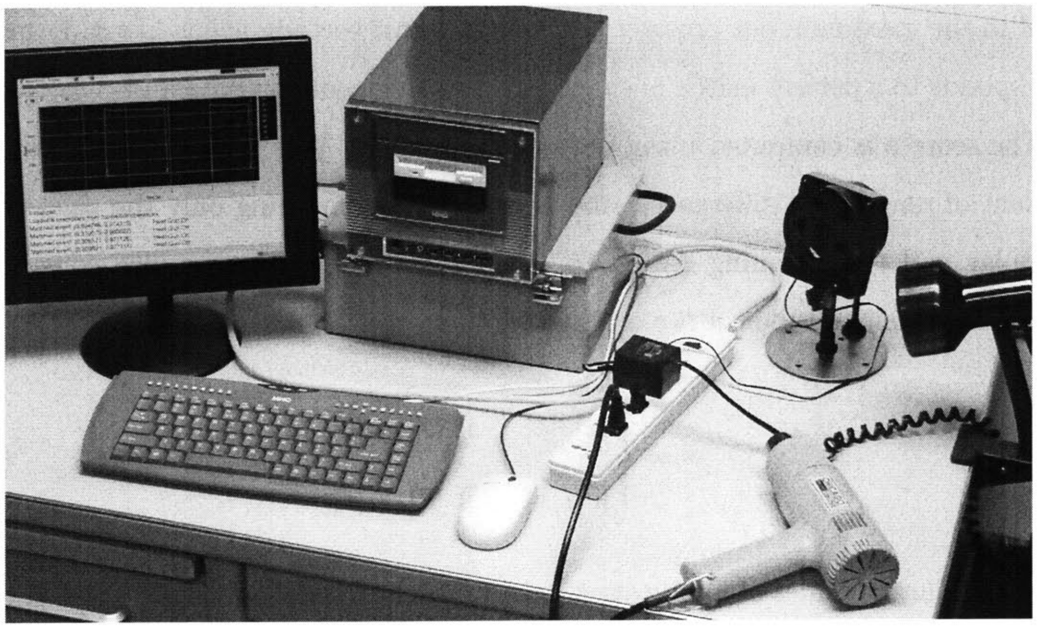


Figure 4-6: Photograph of a demonstration platform built around the `nilmgui` software. Here, the connected power strip allows the system to monitor a lamp, a fan, and a heat gun.

4.3.2 Developed Systems

The `nilmgui` software was created and used to build two similar but independent systems in order to demonstrate the performance and capabilities of the NILM in different situations. One, the "satellite" demo, was introduced in §1.1.5. In a satellite environment, there is often a strong desire to reduce weight and power consumption, and the processor available to run the NILM algorithms was therefore very limited. Furthermore, the components being monitored are small DC loads with power usage that is quite dissimilar to typical NILM diagnostics like those explored in Chapter 2. A tuned version of `nilmgui` was developed that incorporates optimizations for low-speed CPUs, reduced memory requirements for embedded systems, and additional filtering for DC power waveforms with a low signal-to-noise ratio. The resulting software successfully demonstrated the applicability of NILM techniques to specialized environments.

Figure 4-6 shows another demonstration system built around the `nilmgui` software. This particular setup, referred to as the "scotland" demo, consists of a standard

PC in the MicroATX form factor, an interface box containing power supplies and the data acquisition hardware, and a universal power strip that accepts many standard plug types. It can be powered from either 110 or 220 volts, and monitors all devices connected through the included power strip. Because the software runs with one common set of training and matching parameters, this system provides an excellent example of how generalized matching algorithms can automatically adapt to a wide variety of loads and power systems.

Chapter 5

NILM Software Framework

The NILM software framework is a collection of programs, libraries, and databases designed to annotate, store, retrieve, and manipulate the data measured by non-intrusive load monitoring systems. It is a modular and extensible package that that defines and implements common data formats and interfaces to support sharing and interchange between various NILM hardware and software components.

This chapter introduces the concepts and design behind the NILM framework in §5.1. The implementation of the framework is presented in §5.2 and §5.3, and the usage of this implementation is discussed and demonstrated in §5.4.

5.1 Design

The primary concept of the framework is that everything is represented as a *stream*, which represents either measured data or the results of some particular computation over a period of time. The raw voltage and current data from a data acquisition card, for example, would be stored in one stream, and the computed spectral envelopes from the AC preprocessor would be another. A list of detected power transients might be a third.

Streams are annotated lists of *records* and their associated *metadata*, as depicted in Figure 5-1. The records are the primary data storage element of the stream, and can contain any amount of data, but are typically used to store just one set of samples or

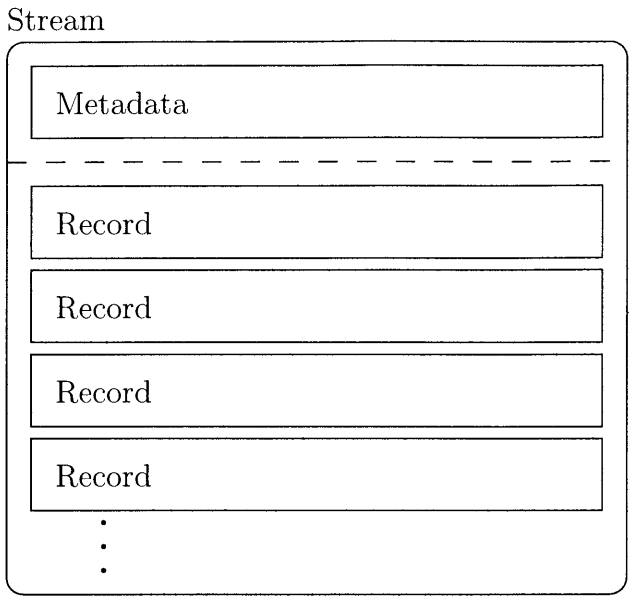


Figure 5-1: Conceptual layout of a stream in the NILM software framework.

one data point. The metadata notes the origin, type, and timespan of data contained within the stream's records, and facilitates general-purpose tagging and indexing of streams.

Streams contain information for exactly one contiguous block of time. This block may be absolute ("today, noon to 3 pm") or relative ("first three seconds of a training pattern"), but data with time gaps must be represented as multiple streams. The software and libraries understand that two streams with matching metadata represent different time-slices of the same information, and can split and join them as required to fulfill a request. For example, if one process stores streams consisting of adjacent ten-minute blocks of recorded data, another process can request an entire hour and have the smaller streams transparently assembled. This process of *stream slicing*, depicted in figure 5-2, allows flexibility in analysis that is independent of how data was recorded or the behavior of any earlier processing steps. Stream slicing also allows the underlying data storage methods to choose an optimal way to store data without exposing such details to the rest of the framework.

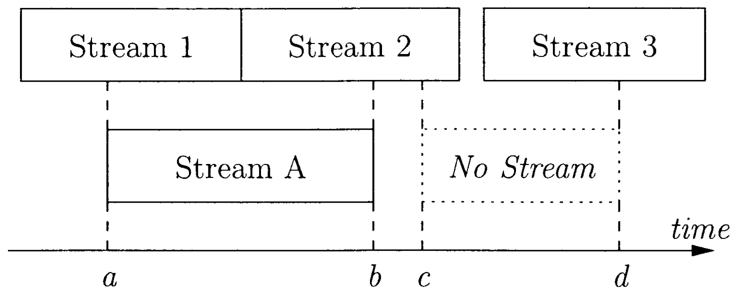


Figure 5-2: Example of stream slicing in the NILM framework. Streams 1, 2, and 3 are recorded data from the same source. A request for data between a and b is assembled and returned as Stream A, while a request for a stream between c and d cannot be satisfied due to missing data.

5.1.1 Metadata

Metadata associated with a stream serves to uniquely identify the source, type, and timing of the data records within the stream. The metadata format is designed to provide a minimum required amount of information to the library and storage layers to allow for flexible storage and retrieval, while also supporting additional custom annotations to be used by software built on the NILM framework. The general metadata layout is shown in figure 5-3. It consists of the following fields:

Tag

The tag is a free-form string that provides a unique identifier for the stream. Streams that are from the same source but representing different periods of time would typically be given the same tag. No two streams with identical tags and types may overlap in time. Example tags include "`Building entry, phase A raw`", "`Pump #3 transients`", and "`Training exemplar #1401`".

Type and Flags

The type and flags fields serve to identify the format of records stored in the stream. The *type* is an integer or predefined string used by high-level programs, modules, and filters to denote the type of data in the stream. Different stream types might represent measured voltage and current, computed spectral envelopes, or a list of when some particular event occurred. These type identifiers

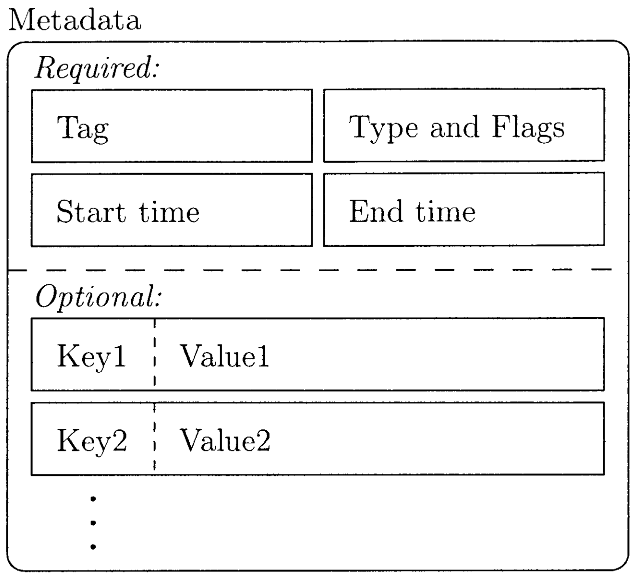


Figure 5-3: Layout of metadata within a stream in the NILM software framework.

can be defined in an application-specific way and do not affect the underlying storage or retrieval. The *flags*, on the other hand, indicate fundamental features of the stream that must be commonly defined throughout the framework. For example, one flag marks whether slicing may be performed on this stream.

Start and End Time

The time fields indicate the period of time over which the records in this stream are defined. These fields may represent either absolute times and dates, times relative to some reference, or simply the length of data stored in the stream. The correct interpretation of the timestamps is designated in the stream flags.

Key/Value pairs

Depending on the stream type, programs that utilize the NILM framework may wish to add additional annotations to a stream. The framework supports this through arbitrary <*key, value*> pairs. Spectral envelopes from the AC preprocessor, for example, can be annotated with the tag of the stream from which they were computed. Keys provide a textual description of the additional information, while values may contain either text-based or binary data.

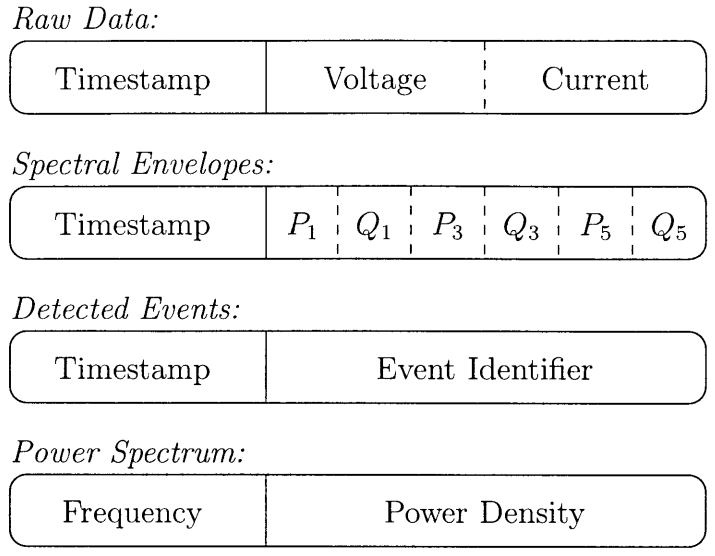


Figure 5-4: Examples of potential data record formats in a NILM stream.

5.1.2 Records

Each record consists of a timestamp and associated data. The format of this associated record data is flexible and can be interpreted based on the stream type and stream flags. The record data may be strings, integers, floating point numbers, lists of these, or any application-specific binary data. The timestamp field is used by the framework for stream slicing, and may alternatively be used to hold other monotonically increasing unique identifiers for streams where slicing is not supported and true timestamps are not applicable.

Representative examples of the record formats corresponding to various stream types are shown in Figure 5-4 and described below. These formats are by convention, and neither the format nor the length of the record data is enforced by the storage or library layers of the framework.

Raw data: Raw voltage and current data, as measured by the data acquisition hardware. The scaling is arbitrary and may be noted in the stream metadata.

Spectral envelopes: Spectral envelopes could contain any number of harmonics. The particular stream type or custom metadata within the stream could be used to indicate the number of elements per record.

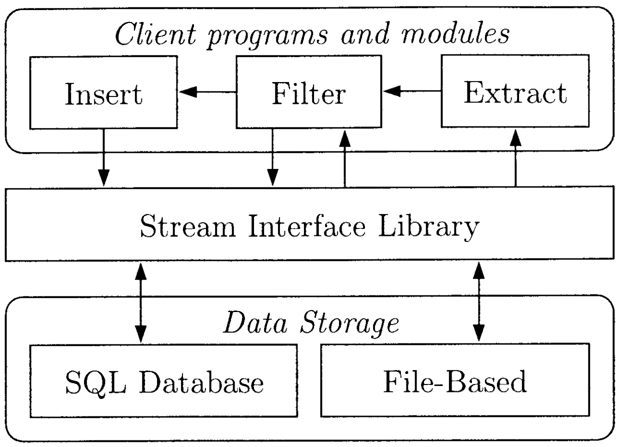


Figure 5-5: Diagram of the software layering in the NILM framework.

Detected events: A transient event detector can generate a new stream with records that note the time and identification of any matches.

Power Spectrum: For some data sets, such as a power spectrum computed over a length of time, the timestamp can be replaced with a more appropriate entry like frequency. A stream containing records like this would not support automatic stream slicing.

5.2 Implementation

The implementation of the NILM software framework uses logically separate layers to facilitate future changes and alternate configurations. This layering is shown graphically in Figure 5-5. The core software component is the Stream Interface Library (SIL), which provides a stable interface through which individual modules and client programs can access the underlying data storage.

A reference implementation that provides database-backed storage, an object-oriented SIL, and command-line client programs is included in Appendix A.4. Specific details of these components are described in the following sections.

5.2.1 Database Storage

The stream metadata format is designed for straightforward implementation in a relational database such as MySQL, following closely the specifications in section 5.1.1. This allows complex queries to be performed on the streams in a standard and well-known language. Stream metadata, records, and key/value pairs, are stored in the database in separate tables, with primary and foreign keys linking them together. A MySQL database schema for such an implementation is listed in Appendix A.4.1.

Depending on the amount of data present and the required processing speed, the stream records may be stored either in or out of the database. When storing in the database, different record types may be assigned to different tables that are optimized for their particular structure. This is primarily a decision that can be made at the interface library and database layers, although such specialized tables would necessarily constrain the format of data provided by high-level modules.

Very large datasets may be more optimally stored outside the primary database. In this case, the database entry for the stream might contain the names of external files or pointers to additional databases that contain the record data, and the library layer would handle the details of retrieving the data from the external source.

5.2.2 Stream Interface Library

The Stream Interface Library provides a number of library functions designed to build abstractions for database access, supply tools for manipulation of stream, metadata, and record objects, and create a facility for selecting and querying streams based on metadata and key/value pairs. Client programs are provided with an object-oriented view of the data and streams, and it is the job of the library to convert these objects to and from their respective database representations.

An implementation of the SIL as a C++ library, `libstream`, is presented in Appendix A.4.2. This library uses the *MySQL++* API to connect to and perform queries and operations against the MySQL database backend. It supports flexible connection options that include authentication and accessing a database over the network.

Record transfer between the library and the clients can efficiently handle large data sets through the use of producer and consumer callback functions, and future storage methods can be added that use the same client interface. The client interface is further stabilized by ensuring that all queries and operations are performed through the stream objects and that internal SQL statements are not exposed outside the library.

The features presented to client programs by the SIL include:

- A "stream" object containing stream metadata and methods to manipulate the stream data.
- A "database" object that initializes and accesses the database.
- Methods to insert and extract streams and associated data from the database, with automatic stream slicing when applicable.
- Methods to compare streams and query the database for streams with particular parameters.
- Enforcement of specifications, such as the rule that no two streams with the same tag and type can overlap in time.

5.2.3 Client Programs

Client programs, or modules, are the tools that provide and use the information in the streams. They include data input methods, filters, classifiers, and diagnostic indicators. The focus in client programs is the implemented algorithms and functionality, and they build upon the library interface to provide access to streams and records with a minimal amount of code. These modules may also build upon each other to provide additional levels of abstraction. For example, the "`filter`" program in §5.3.4 reuses the "`insert`" and "`extract`" programs rather than interfacing directly through the SIL.

The intention is that each client program performs one primary operation, and that separate clients are written for each processing task. This keeps the code simple,

Option	Description
`--db` *name*	Name of the NILM database, default "nilmdb".
`--host` *host*	Hostname of the database server, default "localhost".
`--user` *user*	Database authentication username, default "nilmdb".
`--pass` *pass*	Database authentication password, default empty.
`--version`	Display program name and version info.
`--help`	Display summary of options and usage.

Table 5.1: List of the command-line options common to all C++ client modules.

and provides more flexibility in allowing clients to be reused in different situations. Methods of controlling individual clients and how they link together are discussed in §5.4.1.

For most simpler clients, like those presented in the following section, command-line arguments are the preferred method of selecting and specifying streams and options. This design fits in particularly well with existing NILM hardware and software, as detailed in §5.4.2. Alternatively, programs might utilize a more robust configuration file or XML specification, depending on complexity and intended use.

5.3 Example Modules

Several C++ and script-based clients have been written and are presented in Appendix A.4.3. These clients build on the Stream Interface Library and each other to provide the core functionality necessary for data collection and diagnostics within the NILM software framework.

Parameter common to all of the C++ modules are shown in Table 5.1. These parameters primarily control database access, and their parsing is handled by the shared library `libopt`. In addition, each module shares a common format and routines for interpreting timestamps. The simplest specification is an arbitrary floating-point number, while streams dealing with absolute dates can more easily be specified with a string of the form "`Sun Dec 4 16:22:17 2005`".

5.3.1 Insert

The `insert` program in Appendix A.4.3 creates streams and data records in the database. Command-line arguments are used to specify the new stream tag, type, flags, and metadata, and the data records are supplied as text input on `stdin`. The SIL checks the supplied parameters and dates to ensure that the new stream is well-formed and does not conflict with any existing streams in the database.

Because the record specification requires that all data include a corresponding timestamp, `insert` provides four options for controlling how this timestamp gets generated. The available options are:

1. Provide a fixed time step Δt. The timestamp is increased by Δt for each record.
2. Provide a total number of records that will be supplied. This is used to compute a Δt such that records are evenly distributed over the stream interval.
3. Specify that the data is being supplied in real-time and that timestamps should be generated by `insert` based on when the data arrives.
4. Annotate the input with a timestamp on each line. This provides the most flexibility for the program supplying the data.

A table of command-line arguments that control this behavior and specify the stream metadata is presented in Table 5.2.

5.3.2 Extract

The `extract` program in Appendix A.4.3 is the logical opposite of `insert`, and it provides the ability to retrieve and print the records in the database corresponding to a specified stream. Output data follows the same format in which it was originally supplied to `insert`. Timestamps from the database are generally used for stream slicing and then discarded, but may also be optionally included in the output.

Like `insert`, this program makes use of command-line arguments to specify the stream parameters. In this case, however, many of these arguments are optional and

Option	Description
`--tag` *string*	Tag to use for the new stream.
`--start` *datespec*	Starting time for this stream.
`--end` *datespec*	Ending time for this stream.
`--type` *typespec*	Type of data in the stream.
`--metadata` *key=val*	Extra metadata to add to the stream.
`--noslice`	Mark this stream so it cannot be sliced.
`--delete`	Delete the partial stream if an error occurs.
`--step` *time*	Δt for each line of input.
`--lines` *count*	Total lines of input that will be supplied.
`--realtime`	Data is supplied in real-time.
`--annotate`	Incoming data is annotated with timestamps.

Table 5.2: List of specific command-line options for the `insert` module.

Option	Description
`--tag` *string*	Tag of the stream to extract.
`--type` *typespec*	Type of data in the stream.
`--start` *datespec*	Starting time of the data (optional).
`--end` *datespec*	Ending time of the data (optional).
`--metadata` *key=val*	Extra metadata to match (optional).
`--annotate`	Annotate output data with timestamps.
`--quiet`	Match the stream only, no data output.

Table 5.3: List of specific command-line options for the `extract` module.

are used to narrow the selection of streams from the database. If more than one stream matches the given specifications, an error message is printed. The available arguments are listed in Table 5.3.

This program demonstrates the use of the automatic stream slicing feature of the Stream Interface Library. If an exact match for the specified metadata and time interval is not available in the database, the SIL searches all streams with matching metadata and attempts to assemble streams that cover the requested time interval. If this is possible, a new sliced stream is dynamically created in memory and returned to the `extract` client. The client does check and report whether the stream was sliced for informative purposes, but otherwise treats it identically to any other stream.

5.3.3 Dump and Remove

The dump and remove programs in Appendix A.4.3 are supplementary clients provided for stream identification and removal. The first, dump, requests and displays the metadata for all streams currently in the database. Each dumped stream is displayed with a database-specific unique identifier, which can then be supplied to the remove utility to fully delete the stream and any associated data from the database.

Both clients accept the standard arguments in Table 5.1, and remove requires one additional argument to specify which stream to remove. Because they are relatively simple programs, these client modules provide a good example of the ease of performing stream manipulation through the Stream Interface Library.

5.3.4 Filter

An important feature of the NILM software framework is the ability to process data. Data streams can be selected from the database, modified in any way, and placed back into the database with a new tag. There are a number of ways that this can be done. Like the other modules, a data processing filter could utilize the Stream Interface Library directly. It could also invoke programs like extract and insert to manage stream storage. A third method is for this invocation to be done instead by an external process, which allows the filter to focus entirely on algorithms and methods. This is the method supported by the Perl script filter.pl in Appendix A.4.3.

This script is supplied three sets of arguments: the parameters for stream extraction, the parameters for subsequent re-insertion, and a pointer to a filtering application that reads data records on stdin and writes the processed data to stdout. The filtering application is responsible for calculating spectral envelopes, performing transient identification, computing diagnostic indicators, or applying any other desired operations to the data.

While some filters can perform processing based entirely on the supplied data, others may need to read and control the stream timestamps. For example, a median filter can be applied to a stream independent of timestamps, while an event identifier

Option	**Description**
`--in` *string*	Tag, type, flags, and metadata of the input stream, in the same form as arguments to `extract`.
`--out` *string*	Tag, type, flags, and metadata of the output stream, in the same form as arguments to `insert`.
`--filter` *command*	Command to execute as the filter.
`--start` *datespec*	Starting time of the input and output streams.
`--end` *datespec*	Ending time of the input and output streams.
`--annotate`	Use timestamp annotations in input and output.
`--tempfile`	Use temporary files for data storage.

Table 5.4: List of command-line options for the `filter.pl` script.

would want to report the times at which detected events occur. The `filter.pl` script supports two methods of dealing with timestamps:

1. The filtering application deals with raw data only, and `filter.pl` generates equally-spaced timestamps for the output data based on the stream's time interval and the number of data records created by the filter.

2. The filtering application understands and expects timestamps on input, and generates them on output. In this case, `filter.pl` passes the existing timestamps from `extract` to the filter, and the new timestamps from the filter to `insert`.

The former provides better support for existing filter applications like the preprocessor in §4.1, while the latter is more robust and extensible for new development.

Some filters, like those that invoke a Matlab or Octave function, may prefer to transfer data in temporary files rather than through the `stdin` and `stdout` file handles. If requested, `filter.pl` will use temporary files for storage and pass their names as arguments to the specified filter command. The option to control this behavior is included in the full list of command-line arguments shown in Table 5.4.

5.4 Usage and Applications

Practical and flexible usage is one of the design goals of the NILM software framework. The software provides functionality without specifying policy or enforcing any particular usage model. The following sections discuss particular aspects and examples of working within this framework.

5.4.1 System Control and Graphical Interfaces

The clients presented here take the form of separate programs that can be chained together to apply a full suite of processing and analysis to an incoming data stream. The manner in which these tools are invoked and combined can be specified manually by the user in a command shell, or overall control of the system can be performed by other applications. The majority of programming languages include the ability to spawn external programs, making the NILM framework clients easily accessible to nearly all applications in the system.

The level of control and configuration of the framework can vary with intended use. For systems that collect and process a single datastream in a fixed way, a simple shell script or Perl script may suffice, while more complicated dynamic setups may utilize a custom control program with scheduled jobs that manage the various NILM processes. Individual client programs may also be controlled via graphical interfaces, which may be desirable for realtime analysis and reporting. Such an interface could range from a web-based reporting and configuration tool to an integrated KDE/Gnome application that utilizes both external clients and the Stream Interface Library to provide advanced features.

5.4.2 Compatibility with Existing Hardware and Software

All of the hardware and software components presented in Chapters 3 and 4 are still usable within the new NILM framework. For data acquisition, the existing PCI-1710 and USBADC drivers already support creating output in an ASCII text format that matches the expected data format of the `insert` module. The "`--realtime`" option of `insert` is especially appropriate for data acquisition, as it will automatically generate timestamps based on when the data arrives. Alternatively, the "`--step`" option can be used to set a known sampling interval, which helps avoid timing inaccuracies that may arise through buffering and other transport delays.

Similarly, the new consolidated text input feature described in §4.1.1 for the AC preprocessor matches the format required by the `insert` and `extract` tools. The preprocessor can therefore be used in conjunction with the `filter.pl` script to easily convert a stream of raw data to corresponding spectral envelopes.

There are several ways that the `menu-system` data logging software in §4.2 can work with the framework. For existing installations where the primary use is recording data and burning it to CD, the software can remain unchanged, and the resulting data snapshots can be manually transferred and inserted directly into a NILM database elsewhere for processing. Another option is to modify the `snapshot-data` script to place the data directly within a local NILM database rather than creating individual snapshot files on the hard drive. This would allow more precise selection of what data is stored and transferred from the recording system. Finally, if a network connection is available, this modified `snapshot-data` script could transparently transfer data to a remote machine by utilizing the "`--host`" option of `insert`, connecting directly to a centralized NILM database and precluding the need to physically transfer CDs to retrieve data.

The plugin architecture of the `nilmgui` graphical demonstration described in §4.3 allows for a similarly straightforward interface with the NILM framework. The main difference between `nilmgui` and other applications it that it requires data to be supplied in real-time for proper interactive graphical display. This can be achieved through an additional wrapper script or a modification of the `extract` program that would delay the output of data records based on the stored timestamps. Periodic polling of new streams would support the continuous display of incoming data as it is stored by a data acquisition process.

```
data = extract(['--tag "Voltage" --type Raw ' ...
               '--start "Sun Dec 4 16:22:17 2005" ' ...
               '--end "Sun Dec 4 17:22:17 2005" ']);
data = -data;
insert(data, ['--tag "Inverted Voltage" --type Raw ' ...
              '--start "Sun Dec 4 16:22:17 2005" ' ...
              '--end "Sun Dec 4 17:22:17 2005" ' ...
              '--lines ' num2str(size(data,1))]);
```

Figure 5-6: Example of retrieving, manipulating, and storing data in the NILM database from within Matlab/Octave.

5.4.3 Interfacing with Matlab/Octave

Matlab and Octave provide an excellent language for the implementation of diagnostic indicators and other processing algorithms. They deliver a wide variety of tools and functions that simplify and assist calculations, and the script-based nature of the language allows for rapid development and testing. The majority of analysis for the diagnostic indicators developed in Chapter 2, for example, was carried out with Matlab, and extending the NILM framework to this environment creates a powerful tool for future work.

The most straightforward way to access the NILM database from within a Matlab script is to utilize the existing `insert` and `extract` clients by executing them directly using the "`system`" function. The results from `extract` can be stored to a file and subsequently loaded into Matlab for manipulation, and the input to `insert` can similarly be provided by first saving Matlab variables to a temporary file.

Two scripts which automate these tasks, `insert.m` and `extract.m`, are listed in Appendix A.4.4. They each provide a simple Matlab function that manages spawning the appropriate client and transferring data to and from Matlab variables. All of the flexibility and features of the clients are available, including the automatic stream slicing. The arguments used to specify streams match the command-line arguments shown in Tables 5.2 and 5.3.

As an example of using these functions, Figure 5-6 shows the Matlab commands used to retrieve a particular stream from the database, negate all of the values, and store the data in a new stream.

Chapter 6

Conclusions

The non-intrusive load monitor has widespread applications. It provides the ability to accurately identify and track the behavior and state of multiple electromechanical systems without requiring physical modification or custom instrumentation of each load. Any system that uses electric power can potentially benefit from this cost-effective and robust monitoring, and this work has demonstrated that benefit.

6.1 NILM Applications on the *SENECA*

The successful development of NILM diagnostics techniques for systems aboard the *USCGC SENECA* in Chapter 2 indicates that this non-intrusive approach is not merely academic. Real systems with real problems were located, their electrical power usage was monitored, and specific diagnostic indicators were created to explore and explain the observed faults.

For the leaks in the *SENECA* vacuum wastewater disposal system, Chapter 2 presented measurements and simulation results that verified the applicability of the NILM to the problem of pneumatic failures in cycling systems. Subsequent analysis demonstrated that new statistical interpretations of this data can be accurately applied to not only detect the leaks, but also to distinguish them from normal crew usage.

The analysis of the pump coupling failures on the *SENECA* further demonstrated the flexibility of the NILM. Using the same monitoring hardware, a metric was created that provides an accurate indication of the coupling health after observing as few as one pump start. Although the presented frequency-based approach is considerably different from the statistical approach used with cycling systems, the NILM scales well to both.

6.2 NILM Framework Improvements

This work has reevaluated and improved upon every aspect of the NILM, from initial data acquisition and storage to retrieval, processing, and presentation. The results include new hardware and software tools as well as a general framework for the application and development of future components that will unify and simplify the practical usage of non-intrusive load monitoring in real-world systems.

Chapter 3 described the development of the USB data acquisition hardware and associated driver software. This new device reduces the cost and increases the accuracy of power monitoring hardware. More importantly, it extends the NILM platform to a wide variety of computers and devices that did not support the previous PCI-based hardware, including laptops and embedded systems. The utility of the USBADC was successfully demonstrated with its application to the satellite load monitoring system, and it will continue to be a key component in the development of future NILM systems.

The supporting software tools presented in Chapter 4 proved to be highly useful. Improvements to the AC preprocessor allowed this fundamental component to continue to be applied alongside new hardware and analysis techniques. The menu-driven data storage and retrieval system supported the development of diagnostic indicators aboard the *SENECA* and elsewhere, and allowed for both long-term storage and timely retrieval of the recorded data. The creation of a graphical user interface for transient event training and identification explores, demonstrates, and suggests methods for future reporting and control.

All of this work is brought together by the software framework introduced in

Chapter 5. This new data storage and manipulation framework creates a centralized database for use by all other NILM components. It unifies different hardware, storage requirements, and processing techniques while increasing flexibility and compatibility with existing and upcoming tools. Clients and scripts such as the Matlab/Octave interface have been provided which make working with streams in the database as easy as loading data from a file, facilitating practical use. The system was designed to accommodate all of the the techniques and requirements demonstrated by earlier diagnostic work, and future development of diagnostic indicators can take place entirely within the new framework.

6.3 Future Work

Future work will involve extending and building functionality on top of the NILM hardware and software framework, including the development of a unified interface through which diagnostic indicators can be applied and results can be analyzed. This interface will provide the ability and means to install a system that will automatically monitor and report the state of electrical loads. Such a system could find immediate usage on the *USCGC SENECA* and other locations for which fault metrics have already been created.

The NILM will always benefit from the creation of new identification and diagnostic methods, including both generalized and system-specific techniques. Future utilization of improvements such as the increased resolution of the USBADC hardware should help expose new ways of applying non-intrusive monitoring to a growing variety of loads. The stream-based software framework will facilitate this exploration by allowing more efficient reuse and extension of existing algorithms and routines.

The work on the *SENECA* systems shows high potential for extension and additional exploration. For example, the cycling system diagnostic opens new possibilities for using non-intrusive electrical load monitoring to investigate pneumatic faults. Further verification and improvement of the models presented here is currently taking place, and similar statistical analysis and simulation is being used to apply these concepts to other cycling systems.

Finally, the USB data acquisition hardware and satellite load monitoring system mark the first among what is expected to be many developments in the miniaturization and specialization of NILM hardware for embedded use. Single-board computers can increase in prevalence as algorithms are optimized for these platforms. Using the flexibility of the software framework, specific tools can be recreated in hardware using a field-programmable gate array (FPGA) or other reconfigurable device. This can lead to the creation of a single module or even single-chip solution that includes every aspect of NILM diagnostics from acquiring data to reporting results.

Appendix A

Code Listings

A.1 Cycling Systems

A.1.1 `time-between.pl`

Computes the discharge times between pump runs from preprocessed input.

```
#!/usr/bin/perl
# $Id: time-between.pl 2412 2005-12-06 22:09:59Z jim $

# Computes time between pump runs for data given on stdin.

$edge_time = -1;
$last_val = 0;
$val = 0;
$last_rawval = 0;
$lines = 0;
while(<>)
{
    ($junk, $raw) = split(/\s+/);

    # $val should be 1 if the pump is on,
    #                0 if the pump is off.
    # On/off is determined by whether the power draw is above/below 1000
    # but only if it stays there for 2 seconds or longer

    if($raw > -1000) {
        $rawval = 0;
    } else {
        $rawval = 1;
    }
    if($rawval != $last_rawval) {
        # immediate change; reset the count
        $count = 0;
        $last_rawval = $rawval;
    } elsif($rawval != $val) {
        # raw value staying constant, increment count
        $count++;
        if($count > 240) {
            # stable, keep it
            $val = $rawval;
        }
    }

    if($val == 1 && $last_val == 0) {
        if($edge_time >= 0) {
            $diff = ($lines - $edge_time);
            print "$diff\n";
        }
    }
    if($val == 0 && $last_val == 1) {
        $edge_time = $lines;
    }

    $last_val = $val;
    $lines++;
}
```

A.1.2 sim.m

Main loop of the cycling system simulator. Generates flush events and runs the pumping/leaking simulation between them.

```
%% $Id: sim.m 2413 2005-12-06 22:26:58Z jim $
%% Parameters
global pump1_rate pump2_rate pump_low pump_lower pump_high leak;
T = 120;                    % simulation time (hours)
flush_drop = 1;             % pressure drop of single flush
pump1_rate = 19 * 60;       % rate for first pump, hg/hour
pump2_rate = 17 * 60;       % rate for second pump, hg/hour
pump_low = 10;              % one pump turns on
pump_lower = 7.5;           % both pumps turn on
pump_high = 15;             % both pumps turn off
leak = 15;                  % leak rate, hg/hour
lambda = 150;               % average flushes per hour

%% Initialization
pressure = pump_high;
pump_on = [];               % times that either pump turned on
pump_off = [];              % times that either pump turned off

%% Simulate flushes for the entire time period
t = 0;
lastt = 0;
```

```
while (t < T),
  %% Run the pumping/leaking simulation to get us caught up
  [ pressure, tmpon, tmpoff ] = pumpleak(pressure, t - lastt);
  pump_on = [ pump_on, tmpon + lastt ];
  pump_off = [ pump_off, tmpoff + lastt ];

  %% Now we're caught up with pumping, so flush the toilet.
  pressure = pressure - flush_drop;

  % Jump forward to next flush
  lastt = t;
  t = t - log(rand) / lambda;
end

% Results are in pump_on and pump_off
```

A.1.3 pumpleak.m

Computes the effect of of pumps and leaks between flush events for the cycling system simulator.

```
function [ pressure, turnons, turnoffs ] = pumpleak(pressure, T)
  %% Simulate the pumping / leaking for 't' hours
  %% $Id: pumpleak.m 2413 2005-12-06 22:26:58Z jim $
  global pump1_rate pump2_rate pump_low pump_lower pump_high leak;

  persistent pumps_running;
  if (T == 0)
    pumps_running = 0;
  end

  if (pressure < 0)
    error('Pressure is negative!  Too much flushing?\n');
  end

  t = 0;
  turnons = [];
  turnoffs = [];
  while(t < T),
    left = T - t;

    %% Figure out the current pressure rate
    if (pressure <= pump_lower | pumps_running == 2)
      rate = pump1_rate + pump2_rate - leak;
      if(pumps_running != 2)
        turnons = [ turnons, t ];
        pumps_running = 2;
      end
    elseif (pressure <= pump_low | pumps_running == 1)
      rate = pump1_rate - leak;
      if(pumps_running != 1)
        turnons = [ turnons, t ];
        pumps_running = 1;
      end
    else
      rate = -leak;
      if(pumps_running != 0)
        turnoffs = [ turnoffs, t ];
        pumps_running = 0;
      end
    end

    %% Now jump forward until we run out of time or something may change
    if (rate == 0)
      t = T;
    else
      if (pumps_running == 2 && rate < 0)
        error('Both_pumps_running_and_still_losing_pressure_due_to_leak!');
      elseif (pumps_running == 1 && rate < 0)
        %% Run until we need two
        stop = pump_lower;
      elseif (rate < 0)
        %% Run until we need one
        stop = pump_low;
      elseif (rate > 0)
        %% Run until we need none
        stop = pump_high;
      end

      turnoff = (stop - pressure) / rate;
      if (t + turnoff < T)
        t = t + turnoff;
        pressure = stop;
        if(pumps_running != 0)
          turnoffs = [ turnoffs, t ];
          pumps_running = 0;
        end
      else
        t = T;
        pressure = pressure + rate * left ;
      end
    end
  end
end
```

A.1.4 `sim_between.m`

Converts the output of `sim.m` to match the output of `time-between.pl`.

```
function [ intervals ] = sim_between(pump_on, pump_off)
  %% $Id: sim_between.m 2413 2005-12-06 22:26:58Z jim $
  %% pump_on is a sorted array of times that either or both pumps turn on
  %% pump_off is a sorted array of times that all running pumps turn off
  %%
  %% Compute and return the lengths of each interval between the time
  %% the pumps turn off and either pump turns back on (that is,
  %% the lengths of each interval where no pump is running)
  %%
  %% For example, if
  %% pump_on = [ 1, 3, 5, 10 ]
  %% pump_off = [ 2, 7, 15 ]
  %% this returns
  %% intervals = [ 1, 3 ]
  %% because the pumps are off from 2-3 and 7-10

  len_on = length(pump_on);
  len_off = length(pump_off);
  i_on = 1;
  intervals = [];

  for i_off = 1 : len_off

    %% Find the next turn-on after this off
    while(i_on <= len_on)
      if(pump_on(i_on) > pump_off(i_off))
        break;
      end
      i_on = i_on + 1;
    end

    %% If there was none, we're done
    if(i_on > len_on)
      break;
    end

    %% Otherwise, record it and go to the next turn off
    intervals = [ intervals , pump_on(i_on) - pump_off(i_off) ];

  end

end
```

A.2 USBADC

A.2.1 `adc.asm`

Firmware for the SX28AC microcontroller on the USBADC.

```
;;; USB ADC firmware
;;; Written by Jim Paris <jim@jtan.com>
;;; Based on code by S. R. Shaw <sshaw@alum.mit.edu>
;;; Compiles with gpasm

;;; The actual sampling rate for each channel is
;;; 200 kHz / (num_channels * sample_div)
;;; Sample_div must be at least 3 for reliable operation.

;;; Three states, switchable by sending a command via USB:
;;;  Axyy = begin ADC conversion,
;;;         x = num_channels (ascii digit, 1 through 8)
;;;         yy = sample_div (ascii hex, 01 through FF)
;;;         The ADC stream starts immediately.
;;;  B = begin bandwidth test
;;;         Response is stream of 'abcdefghijklmnopqrstuvwxyz!\n'
;;;  S = stop stream
;;;         No further data is sent; idle (default state)
;;; On error, 0xE? 0xFF is sent, where ? depends on the error.

;;; Note that the error response is not valid ADC output, since
;;; you will never get two bytes in a row with bits 7 and 6 set
;;; for a 14-bit ADC.

;;; LED indicates status.  It turns on when benchmarking or capturing
;;; ADC data, and will turn off on stop.  If it turns off at any other
;;; point, that means we overflowed (ADC is sending too fast for the USB)

        processor sx28
        include "sxdefs.inc"
        radix   dec
        ;; This code still fits into 1 page, so use pages1banks1
        ;; and set vec_reset = 0x1ff to speed programming.
device  equ     pins28+oschs2+turbo+pages1banks1+bor26+stackoptx
        id      'U', 'S', 'B', ' ', 'A', 'D', 'C', ' '

vec_interrupt   equ     0x000
vec_reset       equ     0x1ff   ; 0x3ff for pages2, 0x7ff for pages4

clock_freq      equ     50000000
int_clocks      equ     (clock_freq/200000)

        ;; base of data memory
data_global     equ     0x08    ; accessible in all banks
data_bank0      equ     0x10
data_bank1      equ     0x30

interrupt_entry:
        org     vec_interrupt
        page    interrupt
        goto    interrupt
reset_page0:
        page    start
        goto    start

;;; Pin definitions
```

```
#define usb_rxf      porta,3 ;  input
#define usb_txe      porta,2 ;  input
#define usb_wr       porta,1 ;  output
#define usb_rd       porta,0 ;  output
#define usb_data     portc
#define usb_sleep    portb,6

#define adc_sclk     portb,0 ;  communications clock for AD7856 (SX output)
#define adc_dout     portb,1 ;  adc data out line              (SX input)
#define adc_din      portb,2 ;  adc data in line               (SX output)
#define adc_sync     portb,3 ;  sync for serial comm           (SX output)
#define adc_busy     portb,4 ;  adc busy status line           (SX input)
#define adc_cnvst    portb,5 ;  adc conversion start           (SX output)

#define LED          portb,7

;;; Variables
        cblock data_global

        ADC_lo                  ; data to/from ADC
        ADC_hi
        ADC_ch                  ; channel code
        byte_wr
        byte_rd
        temp
        irq_count
        irq_stat

        endc

        cblock data_bank0

        string
        num_channels            ; Number of channels to scan, 1-8
        sample_div              ; Sampling rate divisor.
        hexc

        endc

;;; ------------------
;;; Interrupt handler
;;; ------------------
interrupt:
        decfsz irq_count, f     ; execute every sample_div times
        goto   skip
        movf   sample_div, w
        movwf  irq_count

        ;; if irq_stat was zero, start a new conversion
        movf   irq_stat , f
        btfsc  status,zf
        bcf    adc_cnvst

        ;; Now
        ;; – we are doing a conversion, so delay 100ns
        ;; – no conversion, so turn off LED
        ;; Also increment irq_stat while we're waiting.
        btfss  status,zf        ; 20ns
        bcf    LED              ; 20ns
        nop                     ; 20ns
        nop                     ; 20ns
        incf   irq_stat , f     ; 20ns
        bsf    adc_cnvst

skip:   movlw  −int_clocks      ; interrupt in int_clocks (since last)
        retiw

;;; ————————————
;;; Static data
;;; ————————————
_test   de     "abcdefghijklmnopqrstuvwxyz!",10,0

;;; ————————————————
;;; Main entry point
;;; ————————————————
start:
        ;; initialize pin values and directions
        mode  0x0f
        movlw b'00000001'       ; set usb_wr low, usb_rd high
        movwf porta
        movlw b'11111100'       ; and make them outputs
        tris  porta

        movlw b'00101101'       ; set initial ADC state
        movwf portb
        movlw b'01010010'       ; set ADC i/o directions
        tris  portb

        movlw b'00000000'       ; set usb pins all low
        movwf portc
        movlw b'11111111'       ; and all inputs
        tris  portc

        ;; Wait for ADC calibration
calib:  btfsc  adc_busy
        goto   calib

        ;; num_channels and sample_div will be set and the
        ;; timer interrupt will be enabled when we get the
        ;; appropriate command via USB.

;;; ——————————
;;; Main loop
;;; ——————————
mainloop:
        ;; Wait for irq_stat to increment.
        movf   irq_stat , f
        btfss  status,zf        ; irq_stat zero?
        goto    readadc          ; not zero, go read the ADC

        ;; Check for incoming USB data
        btfss   usb_rxf          ; usb data available ?
        goto    readusb          ; yes, go read it

        goto    mainloop

readadc:
        ;; Setup new control registers while waiting for conversion
        movlw b'11100000'
        movwf ADC_hi
        btfsc   ADC_ch,0
        bsf     ADC_hi,4
        btfsc   ADC_ch,2
        bsf     ADC_hi,3
        btfsc   ADC_ch,1
        bsf     ADC_hi,2
        clrf    ADC_lo

        ;; Wait for conversion to finish
adbusy: btfsc   adc_busy
        goto    adbusy

        ;; Get the result and set up the next conversion
        call    adc_io

        ;; Reset irq_stat ; turn LED off if overflow
        decfsz  irq_stat , f
        bcf     LED
        clrf     irq_stat

        ;; Mark if this was channel 0, then increment, wrapping as necessary
        movf    ADC_ch, f
        btfsc   status, zf
        bsf     ADC_hi, 7        ; Set channel mark flag

        incf    ADC_ch, f        ; increment channel
        movf    num_channels, w
        xorwf   ADC_ch, w        ; xor with num_channels
        btfsc   status, zf       ; if zero, we hit num_channels
        clrf    ADC_ch           ; so reset adc_ch to zero

        ;; Send data out via USB (little-endian)
        movf    ADC_lo, w
        movwf byte_wr
        call    send_byte
        movf    ADC_hi, w
        movwf byte_wr
        call    send_byte

        goto    mainloop

;;; -----------
;;; Subroutines
;;; -----------

;;; -----------
;;; Send "byte_wr" to PC via USB.
send_byte:
        btfss   usb_rxf         ; can we read?
        goto    readusb         ; yes, go do it.
                                ; It's OK to jump out of the call frame like this,
                                ; since the stack doesn't overflow.
        btfsc   usb_txe         ; can we transmit?
        goto    send_byte       ; no, wait

        mode    0x0F            ; port C = all output
        movlw   b'00000000'
        tris    portc

        bsf     usb_wr
        nop
        movf    byte_wr, w
        movwf   portc           ; set output
        nop
        bcf     usb_wr          ; latch on falling edge
        return

;;; -----------
;;; Send "hexc" to PC via USB in ASCII; for debugging.
send_byte_hex:
        swapf   hexc, w
        call    send_byte_hex_lookup
        movwf   byte_wr
        call    send_byte
        movf    hexc, w
        call    send_byte_hex_lookup
        movwf   byte_wr
        call    send_byte
        return
send_byte_hex_lookup:
        andlw   0x0F
        incf    wfile, f
        addwf   pcl, f
        dt "0123456789ABCDEF"

;;; -----------
;;; Get "byte_rd" from PC via USB and return with CF=0,
;;; or return with CF=1 if no byte is available within
;;; a short amount of time. The reason for this function
;;; is because even if two bytes are immediately available,
;;; there's a short period of time between reading the
;;; two where usb_rxf will remain high.
get_byte_tmout:
        bcf     status, cf
        clrf    temp
get_byte_tmout_loop:
        btfss   usb_rxf
        goto    get_byte_inner
        decfsz  temp, f
        goto    get_byte_tmout_loop
        bsf     status, cf
        return

;;; ------------
;;; Get "byte_rd" from PC via USB
get_byte:
        btfsc   usb_rxf         ; data available?
        goto    get_byte        ; nope, wait for it

get_byte_inner:
        mode    0x0F            ; port C = all input
        movlw   b'11111111'
        tris    portc

        bcf     usb_rd
        nop
        nop
        movf    portc, w
        movwf   byte_rd
        bsf     usb_rd
        return

;;; ------------
;;; Send null-terminated data at mem address 000:w to PC via USB
send_string:
        movwf   string          ; save copy of w
        mode    0x00            ; high address bits = 0
        iread                   ; read m:w into m:w
        mode    0x0f            ; reset mode to normal value

        xorlw   0               ; is it null?
        btfsc   status, zf
        return                  ; yes, we're done

        movwf   byte_wr
        call    send_byte
        incf    string, w
        goto    send_string

;;; ------------
;;; Convert hexc from ascii hex and return in hexc with CF=0
;;; or return with CF=1 on error
from_hex:
        movlw   '0'             ; convert from ascii
        subwf   hexc, f
        btfss   status, cf      ; is it negative?
        goto    from_hex_error  ; yes, error

        movlw   0x0a
        subwf   hexc, w         ; is it > 9?
        btfss  status, cf
        goto   from_hex_done   ; no, we're done

        bcf    hexc, 5         ; convert to uppercase
        movlw  'A'-'0'         ; convert to decimal
        subwf  hexc, f
        btfss  status, cf      ; is it negative?
        goto   from_hex_error  ; yes, error

        movlw  0x06            ; is it greater than 6 ('F')
        subwf  hexc, w
        btfsc  status, cf
        goto   from_hex_error  ; yes, error

        movlw  0x0a
        addwf  hexc, f
from_hex_done:
        bcf    status, cf
        return

from_hex_error:
        bsf    status, cf
        return

;;; -----------
;;; Perform ADC I/O.
;;; Control data in [ADC_hi ADC_lo] is written to the device,
;;; and the ADC result is stored in the same registers.
;;;
;;; Total time at 50MHz, including call/return:
;;; 180ns/bit * 16 bits + 100ns extra + 120ns call/return = 3100 ns
adc_io:
        bcf    adc_sync
        nop                    ; adc_sync low to first sclk low = 80ns
        nop
        nop

        ;; Macro to do a single bit of I/O.
        ;; This is just about optimal:
        ;; one bit = 180ns
        ;; lo sclk = 100ns = 55%
        ;; hi sclk = 80ns = 44%
        ;; Fsclk = 5.55 MHz
dobit   macro  reg,bit
        bcf    adc_sclk        ; SCLK fall
        bcf    adc_din         ; assume din = 0
        btfsc  reg, bit        ; wrong?
        bsf    adc_din         ; din = 1
        nop                    ; meet Tsetup for din
        bsf    adc_sclk        ; SCLK rise
        bcf    reg, bit        ; assume output = 0
        btfsc  adc_dout        ; wrong?
        bsf    reg, bit        ; then set output = 1
        endm
```

```
dobyte macro reg, bit          ; do all bits from 'bit' down to zero
       dobit  reg, bit
       if (bit > 0)
              dobyte reg, bit-1
       endif
       endm

       ;; Send/receive all 16 bits, unrolled for speed
       dobyte ADC_hi, 7
       dobyte ADC_lo, 7

       bsf    adc_sync

       return

;;; Receive command from USB and parse it.
readusb:
       ;; See top for the command format

       ;; Turn off interrupts
       movlw b'11111111'      ; disable interrupts
       option
       ;; (re-) initialize ADC
       movlw b'11100000'      ; control reg, single-ended, ch0, no power down
       movwf ADC_hi
       movlw b'00000000'
       movwf ADC_lo
       clrf   ADC_ch
       clrf   irq_stat
       call   adc_io
       bcf    LED             ; turn off LED

       call   get_byte_tmout  ; read a byte if possible
       movlw 0xE0
       btfsc  status, cf      ; timed out?
       goto   readusb_error   ; yes: error

       movlw 'A'              ; is it A?
       subwf  byte_rd, w
       btfsc  status, zf
       goto   readusb_a

       movlw 'B'              ; is it B?
       subwf  byte_rd, w
       btfsc  status, zf
       goto   readusb_b

       movlw 'S'              ; is it S?
       subwf  byte_rd, w
       btfsc  status, zf
       goto   readusb_s

       movlw 0xE1             ; fall through
```

```
readusb_error:
        ;; Error code 0xE0-0xEF should already be in W
        movwf byte_wr
        call    send_byte
        movlw 0xFF
        movwf byte_wr
        call    send_byte
        ;; fall through: stop after error

;;; stop
readusb_s:
        goto    mainloop          ; main loop will do nothing but check USB

;;; bandwidth test
readusb_b:
        bsf     LED
bwtest:
        btfss   usb_rxf           ; usb data available?
        goto    readusb           ; yes, go read it
        movlw _test
        call    send_string       ; send test string
        goto    bwtest

;;; ADC start
readusb_a:
        ;; Read num_channels and verify range
        call    get_byte_tmout  ; read a byte if possible
        movlw 0xE2
        btfsc   status, cf        ; timed out?
        goto    readusb_error     ; yes: error
        movf    byte_rd, w
        movwf num_channels

        movlw '0'                 ; convert to decimal
        subwf num_channels, f
        movlw 0xE3

        btfsc   status, zf        ; is it zero?
        goto    readusb_error

        movlw 0x09                ; is it 9 or higher?
        subwf num_channels, w
        movlw 0xE4
        btfsc   status, cf
        goto    readusb_error     ; yes: error

        ;; Read sample_div and ensure nonzero
        ;; High nibble:
        call    get_byte_tmout  ; read a byte if possible
        movlw 0xE5
        btfsc   status, cf        ; timed out?
        goto    readusb_error     ; yes: error
        movf    byte_rd, w
        movwf hexc
        call    from_hex          ; convert to decimal
        movlw 0xE6
        btfsc   status, cf        ; did it work?
        goto    readusb_error
        movf    hexc, w
        movwf sample_div
        swapf   sample_div, f

        ;; Low nibble:
        call    get_byte_tmout  ; read a byte if possible
        movlw 0xE7
        btfsc   status, cf        ; timed out?
        goto    readusb_error     ; yes: error
        movf    byte_rd, w

        movwf hexc
        call    from_hex          ; convert to decimal
        movlw 0xE8
        btfsc   status, cf        ; did it work?
        goto    readusb_error
        movf    hexc, w
        iorwf   sample_div, f

        ;; Ensure nonzero
        movlw 0xE9
        btfsc   status, zf        ; is it zero?
        goto    readusb_error

        ;; If we want to send a header to start the ADC stream,
        ;; this is the place to do it, via send_byte

        ;; Set up and enable interrupts, and let's go
        movf    sample_div, w
        movwf irq_count
        bsf     LED
        movlw b'10011111'         ; enable timer interrupt
        option

        goto    mainloop          ; start

;;; -----------
reset_entry:
        org     vec_reset
        errorlevel -306           ; don't complain about not setting page,
        goto    reset_page0       ; since we must assume zero on reset

end
```

A.2.2 `ftdi-adc`

The following code listings are the primary components of the `ftdi-adc` USBADC driver software.

`ftdi.c`

Main application which initializes and controls the capture process.

```
/*
 * ftdi-adc
 * Copyright (c) 2003 Jim Paris <jim@jtan.com>
 *
 * This is free software; you can redistribute it and/or modify it and
 * it is provided under the terms of version 2 of the GNU General Public
 * License as published by the Free Software Foundation; see COPYING.
 */

#define _GNU_SOURCE /* for asprintf */
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/time.h>
#include <time.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <pthread.h>
#include <usb.h>
#include <err.h>
#include <sched.h>
#include "ftdiusb.h"
#include "ftdirom.h"
#include "opt.h"
#include "debug.h"
#include "version.h"
#include "buffer.h"

struct options opt[] = {
        { 'i', "id", "string", "chip id to use (default: first avail)" },
        { 's', "set-id", "string", "set chip id (if single device found)" },
        { 'l', "list", NULL, "list all available devices" },
        { 'k', "command", "string", "initial command to send to device" },
        { 'c', "channels", "1-8", "number of ADC channels to sample (default: 2)" },
        { 'd', "divisor", "3-255", "sampling rate divisor (see below; default: 5)" },
        { 'r', "retry", NULL, "retry indefinitely if device disappears" },
        { 'h', "help", NULL, "this help" },
        { 'v', "verbose", NULL, "be verbose" },
        { 'V', "version", NULL, "show version number and exit" },
        { 0, NULL, NULL, NULL }
};
```

```
int verb_count = 0;

int do_setid(chipid_t id);
int do_list (FILE *out);
int do_adc(char *command, chipid_t id, int retry);

void ftdi_realtime(void);
void ftdi_normal(void);

#define NCHARS (4096)

int main(int argc, char *argv[])
{
        int optind;
        char *optarg;
        char c;
        FILE *help = stderr;
        chipid_t id;
        int setid  = 0;
        int gotid  = 0;
        int retry  = 0;
        int list   = 0;
        int channels = 1;
        int divisor  = 5;
        int setparms = 0;

        char *command = NULL;

        id[0]=0;

        opt_init(&optind);
        while((c=opt_parse(argc,argv,&optind,&optarg,opt))!=0) {
                switch(c) {
                case 'v':
                        verb_count++;
                        break;
                case 'l':
                        list++;
                        break;
                case 's':
                        setid++;
                        gotid--;
                        /* fall through */
                case 'i':
                        gotid++;
                        if(strlen(optarg)>sizeof(chipid_t)) {
                                fprintf(stderr,"id_too_long:_%s\n",optarg);
                                goto printhelp;
                        }
                        strcpy(id,optarg);
                        break;
                case 'k':
                        if(command) free(command);
        command = strdup(optarg);
        break;
    case 'c':
        channels = strtol(optarg, NULL, 0);
        if(channels < 1 || channels > 8) {
            fprintf(stderr,"channels␣must␣be␣between␣"
                "1␣and␣8␣inclusive\n");
            goto printhelp;
        }
        setparms++;
        break;
    case 'd':
        divisor = strtol(optarg, NULL, 0);
        if(divisor < 1 || channels > 255) {
            fprintf(stderr,"divisor␣must␣be␣between␣"
                "1␣and␣255␣inclusive\n");
            goto printhelp;
        }
        if(divisor < 3)
            fprintf(stderr,"warning:␣divisor␣is␣probably␣"
                "too␣low␣to␣work␣properly\n");
        setparms++;
        break;
    case 'r':
        retry++;
        break;
    case 'V':
        printf("ftdi-adc␣" VERSION "\n");
        printf("Written␣by␣Jim␣Paris␣<jim@jtan.com>\n");
        printf("This␣program␣comes␣with␣no␣warranty␣and␣is␣"
               "provided␣under␣the␣GPLv2.\n");
        return 0;
        break;
    case 'h':
        help=stdout;
    default:
    printhelp:
        fprintf(help,"Usage:␣%s␣[options]\n",*argv);
        opt_help(opt,help);
        fprintf(help,"Read␣data␣from␣the␣specified␣USB␣ADC␣"
               "board␣via␣libusb.␣␣See␣README␣for␣details.\n"
               "Actual␣per-channel␣sampling␣rate:␣"
               "200␣kHz␣/␣(channels␣*␣divisor)\n");
        return (help==stdout)?0:1;
    }
}

if(optind<argc) {
    fprintf(stderr,"Error:␣too␣many␣arguments␣(%s)\n\n",
        argv[optind]);
    goto printhelp;
}

if((list + setid + gotid)>1) {
```

```
                fprintf(stderr,"Error:␣-l,␣-s,␣and␣-i␣are␣exclusive\n\n");
                goto printhelp;
        }

        if((setid || list) && (setparms || command)) {
                fprintf(stderr,"Error:␣command/channel/divisor␣may␣not␣be␣used␣with␣"
                        "-l␣or␣-s\n\n");
                goto printhelp;
        }

        if(setparms && command) {
                fprintf(stderr,"Error:␣Specifiy␣either␣channel/divisor␣or␣"
                        "command,␣not␣both\n\n");
                goto printhelp;
        }

        ftdi_init();

        if(setid)
                return do_setid(id);
        if(list)
                return do_list(stdout);

        if(command == NULL)
                asprintf(&command,"A%d%02x", channels, divisor);

        return do_adc(command, id, retry);
}

int do_list(FILE *out)
{
        chipid_t chipid[128];
        int n, i;

        verb("Scanning␣for␣chips\n");
        n = ftdi_scan(chipid, 128);
        verb("%d␣found\n",n);

        if(n) {
                fprintf(out,"Detected␣device␣IDs:\n");
                for(i=0;i<n;i++)
                        fprintf(out,"␣␣'%s'\n",chipid[i]);
        } else fprintf(out,"No␣devices␣detected.\n");
        return 0;
}

int do_setid(chipid_t id)
{
        chipid_t chipids[128];
        usb_dev_handle *h;
        uint8_t rom[FTDI_ROM_SIZE];
        ftdi_rom r;
        int ret;
```

```
verb("Attempting␣to␣set␣id␣to␣'%s'\n",id);
if((ret=ftdi_scan(chipids,128))!=1) {
        fprintf(stderr,"Error:␣can␣only␣set␣ID␣when␣exactly␣one␣"
                "device␣is␣present.\n"
                "The␣following␣devices␣are␣currently␣available:\n");
        do_list(stderr);
        return 1;
}
verb("found␣device␣with␣chip␣id␣'%s'\n",chipids[0]);

if((h=ftdi_open_chipid(chipids[0]))==NULL) {
        fprintf(stderr,"Error␣opening␣USB␣device\n");
        return 1;
}

verb("opened␣successfully\n");

if(ftdi_read_eeprom(h,rom)<0) {
        fprintf(stderr,"Error␣reading␣EEPROM␣from␣device\n");
        ftdi_close(h);
        return 1;
}

verb("EEPROM␣loaded\n");

ftdi_rom_from_buffer(&r, rom);

if(strcmp(r.serial_string,chipids[0])!=0) {
        verb("Serial␣string␣doesn't␣match␣the␣one␣found␣in\n");
        verb("EEPROM␣('%s');␣corrupt␣data?\n",r.serial_string);
        verb("Resetting␣entire␣EEPROM␣to␣defaults.\n");
        ftdi_rom_init(&r);
} else {
        verb("Serial␣string␣in␣EEPROM␣matches.\n");
}

strcpy(r.serial_string,id);
ftdi_rom_to_buffer(&r, rom);

verb("Writing␣new␣EEPROM\n");

if(ftdi_write_eeprom(h,rom)<0) {
        fprintf(stderr,"Error␣writing␣EEPROM␣to␣device.\n");
        ftdi_close(h);
        return 1;
}

verb("Resetting␣device\n");
if(usb_reset(h)<0)
        verb("Reset␣failed;␣ignoring\n");

/* h is no longer valid */

printf("ID␣successfully␣changed␣from␣'%s'␣to␣'%s'\n",chipids[0],id);
```

```
        return 0;
}

pthread_mutex_t writer_mutex = PTHREAD_MUTEX_INITIALIZER;
int writer_can_write = 0;

void *writer_thread(void *arg)
{
        struct buffer_pool *bp = (struct buffer_pool *)arg;
        uint8_t *buffer;
        int buflen;
        int saved, i, len;

        for(;;) {
                pthread_testcancel();
                buffer = buffer_get_filled(bp, &buflen);
                pthread_testcancel();

                /* Decode FTDI format in-place */
                saved = 0;
                for(i = 0; i < (buflen - 2); i += 64) {
                        len = buflen - i - 2;
                        if(len > 62) len = 62;
                        memmove(&buffer[saved],
                                &buffer[i+2],
                                len);
                        saved += len;
                }
                LOCK(writer_mutex,{
                        if(writer_can_write) {
                                fwrite(buffer, saved, 1, stdout);
                                fflush(stdout);
                        }
                        else
                                debug("writer_thread_discarding_%d_bytes\n",saved);
                });

                buffer_mark_empty(bp, buffer);
        }
}

/* Read data until it stops.  Return <0 if data doesn't seem to stop. */
#define timediff(a,b) (((a).tv_sec-(b).tv_sec)*1000000+(a).tv_usec-(b).tv_usec)

int read_until_stop(usb_dev_handle *h)
{
        /* Consider successfully stopped if:
           1) We receive nothing for more than 0.2 seconds
           Give up if:
           1) We get more than 32k of data
           2) We receive for more than 1 second
        */
        struct timeval start, empty, now;
        uint8_t buf[4096];
        int ret;
        int total = 0;
        int gotdata = 1;

        gettimeofday(&start, NULL);
        for(;;) {
                ret = ftdi_read(h, buf, 4096);
                if(ret > 0) {
                        /* Got data.  Too much? */
                        gotdata = 1;
                        total += ret;
                        if(total > 32768)
                                return -1;
                        /* Too long? */
                        gettimeofday(&now, NULL);
                        if(timediff(now,start) > 1000000)
                                return -1;
                } else if(ret==-ETIMEDOUT || ret==0) {
                        /* No data: has it been more than 0.2 seconds? */
                        if(gotdata) {
                                gettimeofday(&empty, NULL);
                                gotdata = 0;
                        } else {
                                gettimeofday(&now, NULL);
                                if(timediff(now,empty) > 200000)
                                        return 0;
                        }
                } else
                        /* Error */
                        return -1;
        }
}

int do_adc(char *command, chipid_t id, int retry)
{
        usb_dev_handle *h = NULL;
        struct ftdi_async *ah = NULL;
        int printed = 0;
        int ret;
        uint8_t *buf;
        uint8_t tmp[4096];
        int gotany = 0;
        struct buffer_pool bp;
        pthread_t writer;

        /* Initialize buffer pool and start writer thread.
           Note that the writer stays at normal priority. */
        if((buffer_init(&bp, 4096) != 0) ||
           (pthread_create(&writer, NULL, writer_thread, (void *)&bp) != 0))
        {
                fprintf(stderr,"Can't initialize write_thread\n");
                return 1;
        }
```

```
for (;;) {
        verb("attempting␣to␣open␣chipid␣'%s'\n",id);
        h = ftdi_open_chipid(id);
        if(h==NULL && !retry) {
                fprintf(stderr,
                        "Can't␣find␣chip␣'%s';␣aborting\n",
                        id[0]?id:"(any)");
                ret = 1;
                goto out;
        }
        if(h==NULL) {
                if(!printed) {
                        fprintf(stderr, "Can't␣find␣chip␣'%s';␣"
                                "sleeping\n",id[0]?id:"(any)");
                        printed=1;
                }
                sleep(1);
                continue;
        }
        printed=0;
        fprintf(stderr,"Device␣opened.\n");
        /* If there was no ID specified, set it, so that
           we try to always get the same device if we need to
           reopen it. */
        if(!id[0]) {
                if( ftdi_get_chipid(h,id)<0)
                        id[0]=0;
                verb("going␣to␣use␣id␣'%s'␣from␣now␣on\n",id);
        }

        /* Tiny sleep to ensure all of the previous USB stuff
           has finished fully. Just an attempt to avoid races */
        usleep(1000);


        /***** Now do the ADC-specific stuff. */

        verb("sending␣ADC␣stop␣command\n");
        /* The very first packet sent after open is sometimes
           lost, if the FTDI's transmit buffer is full, so
           just send enough S(top) commands to ensure we use
           two packets. */
        memset(tmp, 'S', 65);
        ret = ftdi_write(h, tmp, 65);
        if(ret != 65) {
                fprintf(stderr, "Failed␣to␣send␣stop␣command\n");
                ftdi_close(h);
                h = NULL;
                if(!retry) { ret=1; goto out; }
                continue;
        }

        verb("waiting␣for␣ADC␣to␣stop\n");
        if( read_until_stop(h) < 0) {
            fprintf(stderr, "Failed to get ADC to stop\n");
            ftdi_close(h);
            h = NULL;
            if(!retry) { ret=1; goto out; }
            continue;
    }

    verb("queueing up async receives\n");
    if((ah=ftdi_async_start(h, &bp))==NULL) {
            fprintf(stderr,"Failed to start transfers\n");
            ftdi_close(h);
            if(!retry) { ret=1; goto out; }
            continue;
    }

    /* Finish printing all text and enter realtime, then
       start sending data. */
    verb("sending start command '%s', and reading data\n",
         command);
    ftdi_realtime();
    if(ftdi_async_send(h, command, strlen(command)) < 0) {
            ftdi_normal();
            fprintf(stderr,"Failed to send command\n");
            ftdi_async_stop(ah);
            ftdi_close(h);
            if(!retry) { ret=1; goto out; }
            continue;
    }

    /* Read the data now. The reaped URB from that command
       will get ignored. */

    LOCK(writer_mutex,{ writer_can_write = 1; });
    while((ret = ftdi_async_next(ah, &buf)) >= 0) {
            buffer_mark_filled(&bp, buf, ret);
    }
    LOCK(writer_mutex,{ writer_can_write = 0; });

    if(ret==-ENOMEM) {
            fprintf(stderr,"Ran out of buffers.  "
                    "Try increasing BUFFER_POOL.\n");
            /* We probably shouldn't retry in this case. */
            retry = 0;
    }
    else
            verb("read failed: %d\n", ret);

    if((ret = ftdi_async_stop(ah)) < 0)
            verb("ftdi_async_stop returned %d\n",ret);

    ftdi_normal();

    if(!retry) {
            fprintf(stderr,"Read failed; exiting\n");
```

```
                        ftdi_close (h);
                        ret = !gotany;
                        goto out;
                }

                fprintf (stderr ,"Read_failed;_sleeping\n");
                ftdi_close (h);
                h = NULL;
                sleep (1);
        }
out:
        pthread_cancel(writer);
        pthread_join(writer, NULL);
        buffer_free (&bp);
        return ret;
}

void ftdi_realtime (void)
{
        volatile char tmp[16384];
        struct sched_param sp;

        verb("***_realtime:_start\n");

        /* Disable paging and ensure stack allocation */
        mlockall(MCL_CURRENT|MCL_FUTURE);
        memset((char *)tmp,0,sizeof(tmp));

        /* Give ourselves priority over all user processes */
        sp.sched_priority = 50;
        if(pthread_setschedparam(pthread_self(), SCHED_FIFO, &sp) != 0) {
                fprintf (stderr ,"Warning:_can't_set_SCHED_FIFO\n"
                        "You_will_likely_lose_data.__"
                        "Try_running_this_as_root.\n");
        }
}

void ftdi_normal(void)
{
        struct sched_param sp;
        int policy;

        /* Return to normal scheduling. */
        sp.sched_priority = 0;
        if(pthread_setschedparam(pthread_self(), SCHED_OTHER, &sp) != 0) {
                fprintf (stderr ,"Warning:_can't_set_SCHED_OTHER\n");
                /* If we're SCHED_FIFO, bail out. Otherwise, we could
                   hang the system by not resetting to SCHED_OTHER. */
                if(pthread_getschedparam(pthread_self(), &policy, &sp) != 0 ||
                   policy != SCHED_OTHER) {
                        fprintf (stderr ,"But_we're_still_SCHED_FIFO:_abort\n");
                        abort();
                }
        }
        /* Reenable paging */
        munlockall();

        verb("***␣realtime:␣stop\n");
}
```

asyncusb.h

Asyncronous USB layer on top of `libusb` and the Linux `usbdevfs` driver, header file.

```
/*
 * ftdi-adc
 * Copyright (c) 2003 Jim Paris <jim@jtan.com>
 *
 * This is free software; you can redistribute it and/or modify it and
 * it is provided under the terms of version 2 of the GNU General Public
 * License as published by the Free Software Foundation; see COPYING.
 */

#ifndef ASYNC_USB_H
#define ASYNC_USB_H

#include <usb.h>
#include <inttypes.h>
#ifndef __user
#define __user
#endif
#include <linux/usbdevice_fs.h>

/* Gross hack to convince 2.4 UHCI driver to queue bulk transfers.
   We attempt to submit the URB with this flag, and resubmit
   without it if that fails. */
#ifndef USBDEVFS_URB_QUEUE_BULK_HACK
#define USBDEVFS_URB_QUEUE_BULK_HACK 0x10
#endif

/* Mirror the definition of usb_dev_handle in libusb */
struct async_usb_dev_handle {
        int fd;
        /* more libusb-dependent fields that we don't need */
};

int async_usb_fill_bulk(int ep, uint8_t *bytes, int size,
                        struct usbdevfs_urb *uurb);
int async_usb_set_buffer(struct usbdevfs_urb *uurb, uint8_t *buffer);
uint8_t *async_usb_get_buffer(struct usbdevfs_urb *uurb);
int async_usb_endpoint(struct usbdevfs_urb *uurb);
int async_usb_bulk_getdata(struct usbdevfs_urb *uurb, uint8_t **bytes);
int async_usb_submit(usb_dev_handle *dev, struct usbdevfs_urb *uurb);
int async_usb_resubmit(usb_dev_handle *dev, struct usbdevfs_urb *uurb);
int async_usb_discard(usb_dev_handle *dev, struct usbdevfs_urb *uurb);
int async_usb_reap(usb_dev_handle *dev, struct usbdevfs_urb **uurb);

const char *async_usb_strerror(int error);

#endif
```

asyncusb.c

Asyncronous USB layer on top of `libusb` and the Linux `usbdevfs` driver, implementation.

```
/*
 * ftdi-adc
 * Copyright (c) 2003 Jim Paris <jim@jtan.com>
 *
 * This is free software; you can redistribute it and/or modify it and
 * it is provided under the terms of version 2 of the GNU General Public
 * License as published by the Free Software Foundation; see COPYING.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/ioctl.h>
#include <err.h>
#include "debug.h"
#include "asyncusb.h"

/* Fill a uurb for bulk transfer. */
int async_usb_fill_bulk (int ep, uint8_t *bytes, int size,
                         struct usbdevfs_urb *uurb)
{
        if(size < 0 || size > 4096)
                return -EINVAL;
        if (!bytes)
                return -EINVAL;

        memset(uurb, 0, sizeof(struct usbdevfs_urb));
        uurb->type = USBDEVFS_URB_TYPE_BULK;
        uurb->endpoint = ep;
        uurb->flags = 0;
        uurb->buffer = bytes;
        uurb->buffer_length = size;
        uurb->status = 0;

        return 0;
}

/* Fill only the buffer; assume same len as before */
```

```
int async_usb_set_buffer(struct usbdevfs_urb *uurb, uint8_t *buffer)
{
        uurb->buffer = buffer;
        return 0;
}

uint8_t *async_usb_get_buffer(struct usbdevfs_urb *uurb)
{
        return uurb->buffer;
}

int async_usb_endpoint(struct usbdevfs_urb *uurb)
{
        return uurb->endpoint;
}

/* Extract data from reaped uurb.  Return number of bytes, or <0 on error. */
int async_usb_bulk_getdata(struct usbdevfs_urb *uurb, uint8_t **bytes)
{
        if(uurb==NULL)
                return -EPROTO;

        if(uurb->status!=0) {
                debug("URB reports error: %s\n",
                      async_usb_strerror(uurb->status));
                return uurb->status;
        }

        if(uurb->actual_length > uurb->buffer_length) {
                debug("URB too fat?!  actual=%d, buffer=%d\n",
                      uurb->actual_length, uurb->buffer_length);
                return -EOVERFLOW;
        }

        *bytes = uurb->buffer;
        return uurb->actual_length;
}

/* Re-submit an already-filled uurb; non-blocking. */
int async_usb_resubmit(usb_dev_handle *dev, struct usbdevfs_urb *uurb)
{
        int ret;

        if((ret = ioctl(((struct async_usb_dev_handle *)dev)->fd,
                        USBDEVFS_SUBMITURB, uurb)) < 0)
                return ret;

        return 0;
}

/* Submit an already-filled uurb; non-blocking. This differs
   from _resubmit in that it attempts to deal with _QUEUE_BULK_HACK. */
int async_usb_submit(usb_dev_handle *dev, struct usbdevfs_urb *uurb)
{
        /* 2.4 kernels with UHCI driver need the QUEUE_BULK flag set.
           2.6 kernels choke on it.  Try it, and if it fails , clear it. */
        static int explained = 0;

        uurb->flags |= USBDEVFS_URB_QUEUE_BULK_HACK;
        if(async_usb_resubmit(dev, uurb) >= 0) return 0;
        if(!explained) {
                debug("bulk_queuing_rejected,_trying_normal\n");
                explained=1;
        }
        uurb->flags &= ~USBDEVFS_URB_QUEUE_BULK_HACK;
        return async_usb_resubmit(dev, uurb);
}

/* Discard (cancel) URB */
int async_usb_discard(usb_dev_handle *dev, struct usbdevfs_urb *uurb)
{
        int ret;

        if((ret = ioctl(((struct async_usb_dev_handle *)dev)->fd,
                        USBDEVFS_DISCARDURB, uurb)) < 0) {
                /* debug("Error discarding urb: %s\n",strerror(errno)); */
                return ret;
        }

        return 0;
}

/* Block and return the next completed URB */
int async_usb_reap(usb_dev_handle *dev, struct usbdevfs_urb **uurb)
{
        int ret;

retry:
        if((ret = ioctl(((struct async_usb_dev_handle *)dev)->fd,
                        USBDEVFS_REAPURB, uurb)) < 0) {
                if(errno==EINTR) goto retry;
                debug("Error_reaping_urb:_%s\n",strerror(errno));
                return ret;
        }

        return 0;
}

/* Nonblocking reap.  Returns -EAGAIN if nothing is available. */
int async_usb_reap_nonblock(usb_dev_handle *dev, struct usbdevfs_urb **uurb)
{
        int ret;
        ret = ioctl(((struct async_usb_dev_handle *)dev)->fd,
                    USBDEVFS_REAPURBNDELAY, uurb);
        if(ret < 0 || ret != -EAGAIN)
                debug("Error_reaping_urb:_%s\n",strerror(errno));
        return ret;
}
```

```
const char *async_usb_strerror(int error)
{
        /* Codes for uurb->status */
        switch(error) {
        case 0:               return "Transfer completed successfully";
        case -ENOENT:         return "URB cancelled by usb_unlink_urb";
        case -EINPROGRESS:    return "URB still pending";
        case -EPROTO:         return "Internal USB error";
        case -EILSEQ:         return "CRC mismatch";
        case -EPIPE:          return "Endpoint stalled";
        case -ECOMM:          return "Buffer overrun during read";
        case -ENOSR:          return "Buffer underrun during write";
        case -EOVERFLOW:      return "Babble on endpoint";
        case -EREMOTEIO:      return "Short packet detected";
        case -ETIMEDOUT:      return "Timed out";
        case -ENODEV:         return "Device was removed";
        case -EXDEV:          return "ISO not complete";
        case -EINVAL:         return "ISO madness: log off and go home";
        case -ECONNRESET:     return "URB being unlinked asynchronously";
        default:              return strerror(-error);
        }
}
```

ftdiusb.h

USB routines specific to the FT245BM chip, header file.

```
/*
 * ftdi-adc
 * Copyright (c) 2003 Jim Paris <jim@jtan.com>
 *
 * This is free software; you can redistribute it and/or modify it and
 * it is provided under the terms of version 2 of the GNU General Public
 * License as published by the Free Software Foundation; see COPYING.
 */

#ifndef FTDI_USB_H
#define FTDI_USB_H

#include <usb.h>
#include <inttypes.h>
#include <unistd.h>
#include "buffer.h"
#include "asyncusb.h"

#define FTDI_VID 0x0403
#define FTDI_PID 0x6001

typedef char chipid_t[64];
```

```
void ftdi_init (void);
void ftdi_rescan_bus(void);
void ftdi_close (usb_dev_handle *h);
usb_dev_handle *ftdi_open_chipid( chipid_t chipid );
usb_dev_handle *ftdi_open_dev(struct usb_device *dev);
int ftdi_get_chipid (usb_dev_handle *ftdi, chipid_t chipid);
int ftdi_scan ( chipid_t *chipids, int n);
int ftdi_read_eeprom(usb_dev_handle *h, uint8_t *b);
int ftdi_write_eeprom(usb_dev_handle *h, uint8_t *b);
ssize_t ftdi_read (usb_dev_handle *h, void *buf, size_t count);
ssize_t ftdi_write (usb_dev_handle *h, void *buf, size_t count);

/* Async stuff */
#define FTDI_QUEUE 16
struct ftdi_async {
        usb_dev_handle *dev;
        struct buffer_pool *bp;
        struct usbdevfs_urb uurb[FTDI_QUEUE];
} ftdi_async_t ;

struct ftdi_async * ftdi_async_start (usb_dev_handle *h, struct buffer_pool *bp);
int ftdi_async_next(struct ftdi_async *ah, uint8_t ** filled );
int ftdi_async_stop (struct ftdi_async *ah);
int ftdi_async_send(usb_dev_handle *h, char *buf, int len);

#endif
```

ftdiusb.c

USB routines specific to the FT245BM chip, implementation.

```
/*
 * ftdi-adc
 * Copyright (c) 2003 Jim Paris <jim@jtan.com>
 *
 * This is free software; you can redistribute it and/or modify it and
 * it is provided under the terms of version 2 of the GNU General Public
 * License as published by the Free Software Foundation; see COPYING.
 */

#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <string.h>
#include <usb.h>
#include <err.h>
#include <errno.h>
#include "debug.h"
#include "ftdiusb.h"
#include "ftdirom.h"
```

```
#define FTDI_VID 0x0403
#define FTDI_PID 0x6001

int ftdi_busses;
int ftdi_devices;

void ftdi_init(void)
{
        ftdi_busses = 0;
        ftdi_devices = 0;

        usb_init();
}

void ftdi_rescan_bus(void)
{
        ftdi_busses += usb_find_busses();
        ftdi_devices += usb_find_devices();

        /* Older libusb always returns zero, so don't bother
           with this warning. */
        if(0) {
                if(ftdi_busses==0)
                        warnx("no USB busses found");
                if(ftdi_devices==0)
                        warnx("no USB devices found");
        }
}

void ftdi_close(usb_dev_handle *h)
{
        if(h) {
                usb_release_interface(h, 0);
                usb_close(h);
        }
}

/* Set up the chip.  Returns -1 on error, 0 on success.
   Called by open_chipid */
int ftdi_setup(usb_dev_handle *h)
{
        /* Send reset command */
        if(usb_control_msg(h, 0x40, 0, 0, 0, NULL, 0, 100) != 0) {
                warnx("error resetting chip");
                return -1;
        }

        /* Set baudrate to 921600 baud */
        if(usb_control_msg(h, 0x40, 3, 0x8003, 0, NULL, 0, 100) != 0) {
                warnx("error setting baudrate");
                return -1;
        }

        return 0;
}

usb_dev_handle *ftdi_open_dev(struct usb_device *dev)
{
        usb_dev_handle *h;

        if((h=usb_open(dev))==NULL) {
                warnx("can't open device");
                return NULL;
        }

        if(usb_claim_interface(h, 0)) {
                /* Already open and claimed (or we have no permission) */
                usb_close(h);
                return NULL;
        }
        return h;
}

/* Open and return a handle to the first  available  device
   with the given chipid, or just the  first  available
   device  if  chipid[0]=0.
   Returns NULL if no matching chips found.
*/
usb_dev_handle *ftdi_open_chipid(chipid_t  chipid)
{
        struct usb_bus *bus;
        struct usb_device *dev;
        usb_dev_handle *h;
        int found=0;
        chipid_t  id;

        ftdi_rescan_bus ();

        for (bus = usb_busses; bus; bus = bus->next) {
                for (dev = bus->devices; dev; dev = dev->next) {
                        if(dev->descriptor.idVendor == FTDI_VID &&
                           dev->descriptor.idProduct == FTDI_PID) {
                                found++;
                                if((h = ftdi_open_dev(dev))!=NULL)
                                {
                                        if( ftdi_get_chipid (h,id)>=0)
                                                if(chipid[0]==0 ||
                                                   strcmp(chipid,id)==0)
                                                        if( ftdi_setup (h)>=0)
                                                                return h;
                                        ftdi_close (h);
                                }
                        }
                }
        }
        return NULL;
}
```

```
/* Read the chipid stored as ascii text in the "serial"
   string of the device.  Return -1 if it cannot be read.  */
int ftdi_get_chipid (usb_dev_handle *h, chipid_t chipid)
{
        if(usb_get_string_simple(h, 3, chipid, sizeof(chipid_t))>=0)
                return 0;
        return -1;
}

/* Scan the USB bus, return up to n chipids of available devices.
   Returns the number of detected chips
*/
int ftdi_scan(chipid_t *chipids, int n)
{
        struct usb_bus *bus;
        struct usb_device *dev;
        usb_dev_handle *h;
        int found=0;
        int i=0;

        ftdi_rescan_bus();

        for (bus = usb_busses; bus; bus = bus->next) {
                for (dev = bus->devices; dev; dev = dev->next) {
                        if(dev->descriptor.idVendor == FTDI_VID &&
                           dev->descriptor.idProduct == FTDI_PID) {
                                found++;
                                if(((h = ftdi_open_dev(dev))!=NULL) && i<n)
                                {
                                        if(ftdi_get_chipid(h,chipids[i])>=0)
                                                i++;
                                        ftdi_close(h);
                                }
                        }
                }
        }
        if(found>i) {
                found-=i;
                warnx("found %d device%s that could not be opened",
                      found,(found==1)?"":"s");
        }
        return i;
}

/* Read the EEPROM from the device into b, which must have size
   FTDI_ROM_SIZE. Returns -1 on error, 0 otherwise. */
int ftdi_read_eeprom(usb_dev_handle *h, uint8_t *b)
{
        unsigned int i;
        int r;

        for(i=0;i<FTDI_ROM_SIZE/2;i++) {
                if((r=usb_control_msg(h, 0xC0,0x90,0, i, (char *)&b[i<<1], 2, 100))!=2) {
                        verb("read_eeprom: control returned %d\n",r);
                return -1;
            }
        }
        return 0;
}

/* Write the EEPROM from b, which must have size FTDI_ROM_SIZE,
   into the device.  Returns -1 on error, 0 otherwise. */
int ftdi_write_eeprom(usb_dev_handle *h, uint8_t *b)
{
        int i;
        int r;
        uint16_t *w = (uint16_t *)b;

        for(i=0;i<FTDI_ROM_SIZE/2;i++) {
                if((r=usb_control_msg(h, 0x40, 0x91, LE(w[i]), i, 0, 0, 100))!=0) {
                        verb("write_eeprom: control returned %d\n",r);
                        return -1;
                }
        }
        return 0;
}

/* Write up to count bytes.  Returns number of bytes actually
   written, or <0 on error. */
ssize_t ftdi_write(usb_dev_handle *h, void *buf, size_t count)
{
        /* Break writes up into 64-byte segments; there's some
           maximum size before the FTDI chokes, and 64 bytes is
           reasonable since we don't care about write throughput. */
        size_t written = 0;
        size_t len;
        int ret;

        do {
                len = count - written;
                if(len > 64)
                        len = 64;
                ret = usb_bulk_write(h, 2, buf+written, len, 100);
                if(ret<0)
                        return written?:ret;
                written += ret;
        } while(written<count);

        return written;
}

/* Read up to count bytes.  Returns number of bytes actually
   read, or <0 on error/timeout. */
ssize_t ftdi_read(usb_dev_handle *h, void *buf, size_t count)
{
        /* Try to read the requested number of actual data bytes,
           rounded such that we're asking the FTDI for a multiple of
           64 bytes.  This is ugly because of the necessary
```

```
     stripping of the 2-byte status header from each 64-byte
     packet. */
    static uint8_t b[4096];
    uint8_t *dest = (uint8_t *)buf;
    int ret, saved=0, req, len, i;
    int count_blocks;

    count_blocks = (count - saved) / 62;
    if(count_blocks==0) {
            fprintf(stderr,"code_bug: must read at least 62 bytes\n");
            return -1;
    }
    do {
            req = count_blocks * 64;
            /* Kernel requests for more than 4096 bytes may be broken up */
            if(req > 4096)
                    req = 4096;

            ret = usb_bulk_read(h, 1, (char *)b, req, 100);
            /* Error, or timed out */
            if(ret < 0)
                    return saved ? : ret;
            /* We always expect at least the status bytes */
            if(ret < 2)
                    return saved ? : -EINVAL;

            for(i = 0; i < (ret - 2); i += 64) {
                    len = ret - i - 2;
                    if(len > 62) len = 62;
                    memcpy(&dest[saved],
                           &b[i + 2],
                           len);
                    saved += len;
            }
            count_blocks = (count - saved) / 62;
    } while(ret == req && count_blocks > 0);

    return saved;
}

/****************/
/* Async stuff: */

/* Start receiving data. Return pointer to struct ftdi_async, or NULL
   on error. */
struct ftdi_async * ftdi_async_start (usb_dev_handle *h, struct buffer_pool *obp)
{
    int i;
    struct ftdi_async *ah;

    if((ah=(struct ftdi_async *)malloc(sizeof(struct ftdi_async)))==NULL) {
            verb("Out of memory\n");
            return NULL;
    }
```

```
        if(h==NULL || obp==NULL) {
                verb("params␣are␣NULL\n");
                goto out;
        }

        ah->bp = obp;

        /* The device doesn't like requests that aren't multiples of 64 */
        if((ah->bp->buflen%64)!=0) {
                verb("ftdi_async_start:␣buffer␣pool␣buflen%%64␣must␣be␣0\n");
                goto out;
        }

        ah->dev=h;

        /* Allocate and fill uurbs */
        for(i=0;i<FTDI_QUEUE;i++) {
                uint8_t *buf = buffer_get_empty(ah->bp);
                if(buf==NULL) {
                        verb("Can't␣get␣buffer␣%d\n",i);
                        goto out;
                }
                if( async_usb_fill_bulk (1 | USB_ENDPOINT_IN,
                                            buf, ah->bp->buflen,
                                            &ah->uurb[i])<0) {
                        verb("Error␣filling␣URB␣%d\n",i);
                        goto out;
                }
        }

        /* Submit them */
        for(i=0;i<FTDI_QUEUE;i++) {
                if(async_usb_submit(ah->dev, &ah->uurb[i])<0) {
                        verb("Error␣submitting␣URB␣%d\n",i);
                        goto outdiscard;
                }
        }

        /* All done */
        return ah;

outdiscard:
        for(i--;i>=0;i--)
                async_usb_discard(ah->dev, &ah->uurb[i]);
out:
        free(ah);
        return NULL;
}

/* Get the next block of data from the device. Blocks until data is
   available. Puts filled (raw) buffer in **filled, and returns the
   actual number of bytes read, or negative on error.
   Requests a new buffer via buffer_get_empty and resubmits. */
```

```
int ftdi_async_next(struct ftdi_async *ah, uint8_t ** filled )
{
        int ret;
        static struct usbdevfs_urb *uurb;
        uint8_t *buf;
        int len;

        if(ah==NULL || ah->dev==NULL || ah->bp==NULL)
                return -EINVAL;

        /* Reap URB. Blocks. */
another:
        if((ret=async_usb_reap(ah->dev, &uurb)) < 0)
                return ret;

        /* If it wasn't data coming in, free it and reap another instead.
           This is to catch the outgoing command. */
        if (!( async_usb_endpoint(uurb) & USB_ENDPOINT_IN)) {
                free(uurb);
                goto another;
        }

        /* Get the data */
        if((len=async_usb_bulk_getdata(uurb, filled))<0)
                return len;

        /* Request a new buffer */
        if((buf = buffer_get_empty(ah->bp)) == NULL)
                return -ENOMEM;
        async_usb_set_buffer(uurb, buf);

        /* All done.  Resubmit the URB and return. */
        if((ret=async_usb_resubmit(ah->dev, uurb)) < 0) {
                verb("Error resubmitting URB.\n");
                return ret;
        }

        return len;
}

int ftdi_async_stop(struct ftdi_async *ah)
{
        int i;
        int ret;
        static int printed;;

        if(ah==NULL || ah->dev==NULL || ah->bp==NULL)
                return -EINVAL;

        /* Wait a little bit.  If we have pending URBs, it's better
           to let them complete on their own than kill them here
           (causes occasional kernel oops on 2.4.20) */
        usleep(100000);
        printed = 0;
        for(i=0;i<FTDI_QUEUE;i++) {
                buffer_mark_empty(ah->bp, async_usb_get_buffer(&ah->uurb[i]));
                if((ret = async_usb_discard(ah->dev, &ah->uurb[i])) < 0) {
                        if(!printed) {
                                debug("Error discarding URBs:");
                                printed=1;
                        }
                        else debug(" %d",i);
                }
        }
        if(printed) debug("\n");
        free(ah);

        return 0;
}

int ftdi_async_send(usb_dev_handle *h, char *buf, int len)
{
        static struct usbdevfs_urb *uurb;

        if((uurb=(struct usbdevfs_urb *)
            malloc(sizeof(struct usbdevfs_urb)))==NULL) {
                verb("Error allocating send URB\n");
                return -1;
        }

        if( async_usb_fill_bulk (2, (unsigned char *)buf, len, uurb)<0 ||
           async_usb_submit(h, uurb)<0) {
                verb("Error submitting send URB\n");
                free(uurb);
                return -1;
        }

        return 0;
}
```

buffer.h

Thead-safe buffer pool management, header file.

```
/*
 * ftdi-adc
 * Copyright (c) 2003 Jim Paris <jim@jtan.com>
 *
 * This is free software; you can redistribute it and/or modify it and
 * it is provided under the terms of version 2 of the GNU General Public
 * License as published by the Free Software Foundation; see COPYING.
 */

#ifndef FTDI_BUFFER_H
```

```
#define FTDI_BUFFER_H

#include <pthread.h>
#include <inttypes.h>

#define LOCK(mutex,x) do { \
    int r; \
    if((r=pthread_mutex_lock(&(mutex)))!=0) \
    { verb("Cannot␣lock␣" #mutex ":␣%d\n",r); return 0; } \
    x; \
    if((r=pthread_mutex_unlock(&(mutex)))!=0) \
    { verb("Cannot␣unlock␣" #mutex ":%d\n",r); return 0; } \
} while(0)

/* How many data buffers available (must be > FTDI_QUEUE) */
/* They are 4k in size, so 1024 = 4 megs */
#define BUFFER_POOL 1024

/* Four operations:
   - get empty buffer (return next available buffer, NULL if not available)
   - mark buffer as filled (move to filled queue, send signal)
   - get filled buffer (return next filled buffer, blocks until ready)
   - mark buffer as empty (remove from filled, push on empty)
   Implemented with two queues for simplicity; between them we always
   have between 0 and BUFFER_POOL items.

   Filled needs also to keep track of length, so keep track separately.

*/

struct buffer_pool {
        int buflen;

        /* Empty buffers, ready to be put in URB */
        uint8_t *empty[BUFFER_POOL];
        int empty_offset, empty_len;
        int min_empty;

        /* Filled with data, awaiting output */
        uint8_t * filled [BUFFER_POOL];
        int  filled_buflen [BUFFER_POOL];
        int  filled_offset ,  filled_len ;

        /* Mutex for entire structure */
        pthread_mutex_t mutex;


        /* Signal when filled gets data */
        pthread_cond_t cond;
} out_pool_t;

int  buffer_init (struct buffer_pool *bp, int nbuflen);
int  buffer_free (struct buffer_pool *bp);
```

```
uint8_t *buffer_get_empty(struct buffer_pool *bp);
int buffer_mark_filled (struct buffer_pool *bp, uint8_t *buffer, int buflen);
uint8_t * buffer_get_filled (struct buffer_pool *bp, int *buflen);
int buffer_mark_empty(struct buffer_pool *bp, uint8_t *buffer);

#endif
```

---

buffer.c

Thead-safe buffer pool management, implementation.

```
/*
 * ftdi-adc
 * Cbpyright (c) 2003 Jim Paris <jim@jtan.com>
 *
 * This is free software; you can redistribute it and/or modify it and
 * it is provided under the terms of version 2 of the GNU General Public
 * License as published by the Free Software Foundation; see CBPYING.
 */

#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <inttypes.h>
#include "buffer.h"
#include "debug.h"

int buffer_init (struct buffer_pool *bp, int nbuflen)
{
        int i;

        bp->buflen = nbuflen;

        /* Allocate empty buffers */
        for(i = 0; i < BUFFER_POOL; i++) {
                if((bp->empty[i] = (uint8_t *)malloc(bp->buflen * sizeof(uint8_t)))
                   == NULL) {
                        verb("Out_of_memory_allocating_buffer_%d\n",i);
                        goto out_freei;
                }
        }
        bp->empty_offset = 0;
        bp->empty_len = BUFFER_POOL;
        bp->min_empty = BUFFER_POOL;

        /* No filled buffers */
        bp->filled_offset = 0;
        bp->filled_len = 0;

        pthread_mutex_init(&bp->mutex, NULL);
        pthread_cond_init(&bp->cond, NULL);
        debug("pool:␣%d␣buffers␣initialized\n", BUFFER_POOL);
        return 0;

        i = BUFFER_POOL;
out_freei:
        for(i--; i >= 0; i--)
                free(bp->empty[i]);
        return -1;
}

#define WRAP(x) ((x) % BUFFER_POOL)

int buffer_free(struct buffer_pool *bp)
{
        int i;
        pthread_cond_destroy(&bp->cond);
        pthread_mutex_destroy(&bp->mutex);

        /* We can only free buffers that have been placed back in
           either the empty or filled lists. */
        for(i = 0; i < bp->empty_len; i++)
                free(bp->empty[WRAP(i + bp->empty_offset)]);
        for(i = 0; i < bp->filled_len; i++)
                free(bp->filled[WRAP(i + bp->filled_offset)]);
        return 0;
}

uint8_t *buffer_get_empty(struct buffer_pool *bp)
{
        uint8_t *ret;
        LOCK(bp->mutex,{
                if(bp->empty_len == 0) {
                        ret = NULL;
                } else {
                        /* Pop from empty */
                        ret = bp->empty[bp->empty_offset];
                        bp->empty_offset = WRAP(bp->empty_offset + 1);
                        bp->empty_len--;
                        if(bp->empty_len < bp->min_empty)
                                bp->min_empty = bp->empty_len;
                }
        });
        return ret;
}

int buffer_mark_filled(struct buffer_pool *bp, uint8_t *buffer, int buflen)
{
        int ret;
        LOCK(bp->mutex,{
                if((bp->empty_len + bp->filled_len) >= BUFFER_POOL) {
                        verb("tried␣to␣add␣filled␣buffer,␣but␣already␣full?\n");
                        ret = -1;
                } else {
                        /* Push to filled */
                        int off = WRAP(bp->filled_offset + bp->filled_len);
                        bp->filled[off] = buffer;
                        bp->filled_buflen[off] = buflen;
                        bp->filled_len++;
                        pthread_cond_signal(&bp->cond);
                        ret = 0;
                }
        });
        return ret;
}

uint8_t * buffer_get_filled (struct buffer_pool *bp, int *buflen)
{
        uint8_t *ret;
        LOCK(bp->mutex,{
                /* Wait for data */
                while(bp->filled_len == 0)
                        pthread_cond_wait(&bp->cond, &bp->mutex);
                /* Pop from filled */
                ret = bp->filled[bp->filled_offset];
                *buflen = bp->filled_buflen[bp->filled_offset];
                bp->filled_offset = WRAP(bp->filled_offset + 1);
                bp->filled_len--;
        });
        return ret;
}

int buffer_mark_empty(struct buffer_pool *bp, uint8_t *buffer)
{
        int ret;
        LOCK(bp->mutex,{
                if((bp->empty_len + bp->filled_len) >= BUFFER_POOL) {
                        verb("tried to add empty buffer, but already full?\n");
                        ret = -1;
                } else {
                        /* Push to empty */
                        bp->empty[WRAP(bp->empty_offset + bp->empty_len)]
                                = buffer;
                        bp->empty_len++;
                        ret = 0;
                }
        });
        return ret;
}
```

ftdirom.h

Routines to manipulate the data in the FT245BM EEPROM chip, header file.

```
/*
```

```
 * ftdi-adc
 * Copyright (c) 2003 Jim Paris <jim@jtan.com>
 *
 * This is free software; you can redistribute it and/or modify it and
 * it is provided under the terms of version 2 of the GNU General Public
 * License as published by the Free Software Foundation; see COPYING.
 */

#ifndef FTDI_ROM
#define FTDI_ROM

#include <inttypes.h>

#define FTDI_ROM_SIZE 128

#define ROM_RELEASE_NONE 0x0000
#define ROM_RELEASE_AM 0x0200
#define ROM_RELEASE_BM 0x0400

#define ROM_USBCONFIG_BASE 0x80
#define ROM_USBCONFIG_SELF_POWER 0x40
#define ROM_USBCONFIG_REMOTE_WAKE 0x20

#define ROM_CONFIG_NONE 0x00
#define ROM_CONFIG_IN_ISO 0x01
#define ROM_CONFIG_OUT_ISO 0x02
#define ROM_CONFIG_SUSPEND_PULLDOWN 0x04
#define ROM_CONFIG_USE_SERIAL 0x08
#define ROM_CONFIG_CHANGEVER 0x10

#define LE(x)                                   \
({                                              \
        register uint16_t in = x;               \
        uint16_t out;                           \
        ((uint8_t *)&out)[0] = in & 0xff;       \
        ((uint8_t *)&out)[1] = (in >> 8) & 0xff;\
        out;                                    \
})

typedef struct {
        uint16_t vid;
        uint16_t pid;

        uint16_t release;       /* See ROM_RELEASE_*            */
        uint8_t usbconfig;      /* See ROM_USBCONFIG_*          */
        uint8_t max_power;      /* times 2mA                    */
        uint16_t config;        /* See ROM_CONFIG_*             */

        uint16_t usbver;        /* If ROM_CONFIG_CHANGEVER set */

        char manuf_string[64];  /* Null-terminated; must all    */
        char prod_string[64];   /* fit within maximum eeprom length */
        char serial_string[64];
} ftdi_rom;
```

```
void ftdi_rom_init(ftdi_rom *r);
void ftdi_rom_print(ftdi_rom *r);
void ftdi_rom_from_buffer(ftdi_rom *r, uint8_t *b);
int ftdi_rom_to_buffer (ftdi_rom *r, uint8_t *b);

#endif
```

ftdirom.c

Routines to manipulate the data in the FT245BM EEPROM chip, implementation.

```
/*
 * ftdi-adc
 * Copyright (c) 2003 Jim Paris <jim@jtan.com>
 *
 * This is free software; you can redistribute it and/or modify it and
 * it is provided under the terms of version 2 of the GNU General Public
 * License as published by the Free Software Foundation; see COPYING.
 */

#include "ftdirom.h"
#include <err.h>
#include <stdio.h>
#include <string.h>

uint16_t ftdi_rom_checksum(uint8_t *b)
{
        int i;
        uint16_t c;
        uint16_t *w = (uint16_t *)b;

        c = 0xAAAA;
        for(i=0;i<63;i++) {
                c ^= LE(w[i]);
                c = (c << 1) | (c >> 15);
        }
        return c;
}

void ftdi_rom_init(ftdi_rom *r)
{
        /* Use the defaults on the DLP-USB245M board,
           plus whatever we want for manuf, prod, serial */
        r->vid = 0x0403;
        r->pid = 0x6001;
        r->release = ROM_RELEASE_BM;
        r->usbconfig = ROM_USBCONFIG_BASE|ROM_USBCONFIG_REMOTE_WAKE;
        r->max_power = 22;
        r->config = ROM_CONFIG_NONE;
        r->usbver = 0x0000;
        strcpy(r->manuf_string,"FTDI");
        strcpy(r->prod_string,"FT245BM");
        strcpy(r->serial_string,"");
}

void ftdi_rom_print(ftdi_rom *r)
{
        printf("EEPROM␣contents:\n");
        printf("␣␣␣␣␣␣␣VID:␣0x%04x\n",r->vid);
        printf("␣␣␣␣␣␣␣PID:␣0x%04x\n",r->pid);
        printf("␣␣␣Release:");
        if(r->release & ROM_RELEASE_AM) printf("␣am");
        if(r->release & ROM_RELEASE_BM) printf("␣bm");
        if(r->release ==ROM_RELEASE_NONE) printf("␣none");
        printf("\n");
        printf("USB␣config:␣0x%04x",r->usbconfig);
        if(r->usbconfig & ROM_USBCONFIG_SELF_POWER) printf("␣self_power");
        if(r->usbconfig & ROM_USBCONFIG_REMOTE_WAKE) printf("␣remote_wake");
        if(r->usbconfig ==ROM_USBCONFIG_BASE) printf("␣none");
        printf("\n");
        printf("␣Max␣power:␣%d␣mA\n", r->max_power*2);
        printf("␣␣␣␣Config:");
        if(r->config & ROM_CONFIG_IN_ISO) printf("␣in_iso");
        if(r->config & ROM_CONFIG_OUT_ISO) printf("␣out_iso");
        if(r->config & ROM_CONFIG_SUSPEND_PULLDOWN) printf("␣suspend_pulldown");
        if(r->config & ROM_CONFIG_USE_SERIAL) printf("␣use_serial");
        if(r->config & ROM_CONFIG_CHANGEVER) printf("␣change_version");
        if(r->config ==ROM_CONFIG_NONE) printf("␣none");
        printf("\n");
        printf("␣␣␣USB␣ver:␣0x%04x\n",r->usbver);
        printf("␣␣␣␣␣Manuf:␣'%s'\n",r->manuf_string);
        printf("␣␣␣Product:␣'%s'\n",r->prod_string);
        printf("␣␣␣␣Serial:␣'%s'\n",r->serial_string);
}

void ftdi_rom_from_buffer(ftdi_rom *r, uint8_t *b)
{
        uint16_t *w = (uint16_t *)b;
        int i, off, len;

        if(ftdi_rom_checksum(b) != LE(w[63])) {
                warnx("eeprom␣checksum␣is␣invalid:␣got␣0x%04x,␣wanted␣0x%04x",
                      ftdi_rom_checksum(b), LE(w[63]));
                warnx("using␣defaults");
                ftdi_rom_init(r);
        } else {
                r->vid = LE(w[1]);
                r->pid = LE(w[2]);
                r->release = LE(w[3]);
                r->usbconfig = b[8];
                r->max_power = b[9];
                r->config = LE(w[5]);
                r->usbver = LE(w[6]);
```

```
#define grab_string(name, index)                              \
                r->name[0] = 0;                               \
                off = b[index] - 0x80;                        \
                len = (b[index+1] - 2) >> 1;                  \
                if( off<0x80 && len>0 && len<0x40 &&          \
                   b[off]==(len<<1)+2 && b[off+1]==0x03) { \
                        for(i=0;i<len;i++)                    \
                                r->name[i] = b[i*2+off+2];    \
                        r->name[i]=0;                         \
                }

                grab_string(manuf_string, 14);
                grab_string(prod_string, 16);
                if(r->config & ROM_CONFIG_USE_SERIAL) {
                        grab_string( serial_string , 18);
                } else
                        r->serial_string[0]=0;
#undef grab_string
        }
}

/* Returns -1 if the strings don't fit . */
int ftdi_rom_to_buffer(ftdi_rom *r, uint8_t *b)
{
        int i;
        uint16_t *w = (uint16_t *)b;
        int n, len;

        if((28 +
            2*strlen(r->manuf_string) +
            2*strlen(r->prod_string) +
            2*strlen(r->serial_string))>=128) {
                return -1;
        }

        memset(b, 0, FTDI_ROM_SIZE);

        w[0] = 0;
        w[1] = LE(r->vid);
        w[2] = LE(r->pid);
        w[3] = LE(r->release);
        b[8] = r->usbconfig;
        b[9] = r->max_power;
        w[5] = LE(r->config);
        w[6] = LE(r->usbver);

        n = 20;
#define emit_string(name,index)         \
        len = strlen(r->name);          \
        b[index] = n | 0x80;            \
        b[index+1] = (len << 1) + 2;    \
        b[n++] = (len << 1) + 2;        \
        b[n++] = 0x03;                  \
        for(i=0;i<len;i++) {            \
                b[n++]=r->name[i];    \
                b[n++]=0;             \
        }

        emit_string(manuf_string, 14);
        emit_string(prod_string, 16);
        emit_string( serial_string , 18);
#undef emit_string

        w[63] = LE(ftdi_rom_checksum(b));

        return 0;
}
```

---

opt.h

Command-line option parsing, header file.

```
/*
 * Copyright (c) 2003 Jim Paris <jim@jtan.com>
 *
 * This is free software; you can redistribute it and/or modify it and
 * it is provided under the terms of version 2 of the GNU General Public
 * License as published by the Free Software Foundation; see COPYING.
 */

#ifndef OPT_H
#define OPT_H

#include <stdlib.h>

struct options {
        char shortopt;
        char *longopt;
        char *arg;
        char *help;
};

void opt_init(int *optind);

char opt_parse(int argc, char **argv, int *optind, char **optarg,
               struct options *opt);

void opt_help(struct options *opt, FILE *out);

#endif
```

---

opt.c

Command-line option parsing, implementation.

```
/*
 * Copyright (c) 2003 Jim Paris <jim@jtan.com>
 *
 * This is free software; you can redistribute it and/or modify it and
 * it is provided under the terms of version 2 of the GNU General Public
 * License as published by the Free Software Foundation; see COPYING.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "opt.h"

void opt_init(int *optind) {
        *optind=0;
}

char opt_parse(int argc, char **argv, int *optind, char **optarg,
               struct options *opt) {
        char c;
        int i;
        (*optind)++;
        if(*optind>=argc)
                return 0;

        if(argv[*optind][0]=='-' &&
           argv[*optind][1]!='-' &&
           argv[*optind][1]!=0) {
                /* Short option (or a bunch of 'em) */
                /* Save this and shift others over */
                c=argv[*optind][1];
                for(i=2;argv[*optind][i]!=0;i++)
                        argv[*optind][i-1]=argv[*optind][i];
                argv[*optind][i-1]=0;
                if(argv[*optind][1]!=0)
                        (*optind)--;
                /* Now find it */
                for(i=0;opt[i].shortopt!=0;i++)
                        if(opt[i].shortopt==c)
                                break;
                if(opt[i].shortopt==0) {
                        fprintf(stderr,"Error: unknown option '-%c'\n",c);
                        return '?';
                }
                if(opt[i].arg==NULL)
                        return c;
                (*optind)++;
                if(*optind>=argc || (argv[*optind][0]=='-' &&
                        argv[*optind][1]!=0)) {
```

```
                        fprintf(stderr,"Error:␣option␣'-%c'␣requires␣an␣"
                                "argument\n",c);
                        return '?';
                }
                (*optarg)=argv[*optind];
                return c;
        } else if(argv[*optind][0]=='-' &&
                  argv[*optind][1]=='-' &&
                  argv[*optind][2]!=0) {
                /* Long option */
                for(i=0;(c=opt[i].shortopt)!=0;i++)
                        if(strcmp(opt[i].longopt,argv[*optind]+2)==0)
                                break;
                if(opt[i].shortopt==0) {
                        fprintf(stderr,"Error:␣unknown␣option␣'%s'\n",
                                argv[*optind]);
                        return '?';
                }
                if(opt[i].arg==NULL)
                        return c;
                (*optind)++;
                if(*optind>=argc || (argv[*optind][0]=='-' &&
                        argv[*optind][1]!=0)) {
                        fprintf(stderr,"Error:␣option␣'%s'␣requires␣an␣"
                                "argument\n",argv[*optind-1]);
                        return '?';
                }
                (*optarg)=argv[*optind];
                return c;
        } else {
                /* End of options */
                return 0;
        }
}

void opt_help(struct options *opt, FILE *out) {
        int i;
        int printed;

        for(i=0;opt[i].shortopt!=0;i++) {
                fprintf(out,"␣␣-%c,␣--%s%n",opt[i].shortopt,
                        opt[i].longopt,&printed);
                fprintf(out,"␣%-*s%s\n",30-printed,
                        opt[i].arg?opt[i].arg:"",opt[i].help);
        }
}
```

A.2.3 `convert`

Converts the `ftdi-adc` output to a text-based ASCII format.

```
/*
 *  convert.c
 *
 *  Converts USB ADC data to ASCII
 *
 *  copyright (c) 2003, Montana State University
 *  Copyright (c) 2003 Jim Paris <jim@jtan.com>
 *
 *  Written by Steven R. Shaw, (sshaw@alum.mit.edu)
 *  Modified by Jim Paris <jim@jtan.com>
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2, or (at your option)
 * any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 */

#include <stdio.h>

int getrow(unsigned short *data, FILE *f)
{
        unsigned short datum;
        size_t rval;
        int k = 0;

        do {
                rval = fread(&datum, sizeof(unsigned short), 1, f);

                if(rval) {
                        data[k] = datum & 0x3fff;
                        k++;
                }
        } while(k < 8 && rval && !feof(f) && (datum&0x8000) == 0);

        if((datum&0x8000) == 0)
                k = -1; /* if not a full row */

        return k;
}

void printrow(unsigned short *d, int m)
{
        int i;

        if(m > 0) {
                for(i = 0; i < m; i++)
                        printf("%d\t", d[i]);
                printf("\n");
                fflush(stdout);
        }
}

int main(int argc, char *argv[])
{
        unsigned short d1[8],d2[8];
        unsigned char a, b;
        int n,m;

        /* Try forever; whatever is feeding us may reconnect to the
           device, making the stream discontinuous */
        while(!feof(stdin))
        {
                /* Read two rows, since first  might be incomplete */
                n = getrow(d1, stdin);
                m = getrow(d2, stdin);

                if(n == m)
                        printrow(d1,n);

                do {
                        printrow(d2,m);
                        m = getrow(d2,stdin);
                } while(m > 0);
        }
}
```

A.3 Support Software

A.3.1 `menu-system` components

The following scripts are the primary components of the menu system used for data collection.

`nilm-init`

Startup script that initializes the hardware and restores state.

```
#! /bin/sh
# /etc/init.d/nilm
#
# Written by Jim Paris <jim@jtan.com>

set -e
```

```
case "$1" in
  start)
    echo -n "Loading␣NILM␣modules:␣"
    insmod /home/nilm/lib/pci1710.o
    echo "done."
    $0 reload || true
    echo -n "Restoring␣last␣state:␣"
    su nilm -c "`cat␣/home/nilm/last-command`" || true
    echo "done."
    ;;
  stop)
    echo -n "Stopping␣capture:␣"
    su nilm -c /home/nilm/scripts/stop-run || true
    echo "done."
    echo -n "Unloading␣NILM␣modules:␣"
    rmmod pci1710
    echo "done."
      ;;
  restart)
    $0 stop
    $0 start
    ;;
  reload|force-reload)
    echo "Setting␣up␣PCI1710:␣"
    /home/nilm/bin/set1710 -d /dev/pci1710 < /home/nilm/nilmcfg.pci
    echo "done."
    ;;
  *)
    echo "Usage:␣$0␣{start|stop|restart|reload}"
    exit 1
    ;;
esac

exit 0
```

menu-system

Loads and displays the main menu.

```
#!/bin/sh

# menu-system 2003-06-06
# 2005-01-05 added dvd count
# Jim Paris <jim@jtan.com>
#
# Simple menu interface to the data collection

unset DISPLAY

while /bin/true ; do
```

```
cd /home/nilm

# Redirect stdout to a temporary file
trap "rm␣-f␣menu-system.tmp.$$;␣exit" 0 1 2 11 15
exec 3>&1
exec >menu-system.tmp.$$

# Write the message
echo -n "Data␣collection␣is␣currently␣"
if killall -q -0 run-prep.pl ; then
    running=1
    runmsg="Stop␣collection"
    echo running.
else
    running=0
    runmsg="Start␣collection␣-->"
    echo stopped.
fi

date +"Current␣time␣is␣%Y%m%d-%H%M%S"

if ! ls data | grep -q snapshot ; then
    totalsize =0
    echo "No␣snapshot␣files␣available."
else
    echo -n "Earliest␣non-empty␣snapshot␣is␣"
    basename "`ls␣-r␣data/snapshot*␣|␣tail␣-1`" .gz
    echo -n "␣␣Latest␣non-empty␣snapshot␣is␣"
    basename "`ls␣data/snapshot*␣|␣tail␣-1`" .gz

    totalsize =`du -scl --apparent-size -k data/snapshot* | tail -1 | cut -f 1`
    [ $totalsize = 0 ] && totalsize=1
    echo -n "Total␣size␣of␣compressed␣snapshots␣is␣${totalsize}K␣("
    cds=$(((totalsize+649999)/650000))
    if [ $cds = 1 ] ; then
        echo -n "1␣CD␣or␣"
    else
        echo -n "$cds␣CDs␣or␣"
    fi
    dvds=$(((totalsize+4499999)/4500000))
    if [ $dvds = 1 ] ; then
        echo -n "1␣DVD"
    else
        echo -n "$dvds␣DVDs"
    fi
    echo ")"
fi
if [ -e data/current ] ; then
    echo -n "`du␣-k␣data/current␣|␣cut␣-f␣1`K␣of␣current␣data␣"
    echo "not␣yet␣included␣in␣snapshots"
else
    echo "All␣data␣is␣included␣in␣snapshots"
fi
```

```
# Message complete, redirect stdout back
exec 1>&3 3>&-

 title ="NILM␣(`hostname␣2>/dev/null`)" || title="NILM"

IFS='' gdialog --title "$title" --menu "`cat␣menu-system.tmp.$$`" 17 74 5 \
  1 "Refresh␣this␣screen" \
  2 "$runmsg" \
  3 "Create␣a␣snapshot␣now" \
  4 "Burn␣snapshot␣data␣to␣CD␣or␣DVD" \
  5 "Shell␣prompt" 2>menu-system.tmp.$$

 result =`cat menu-system.tmp.$$`
rm -f menu-system.tmp.$$

case ␣$result in
    ␣1)
        # Refresh
        ;;
    ␣2)
        if [ $running = 0 ] ; then
            IFS='' gdialog --title 'Start Collection' --menu \
"Choose␣the␣type␣of␣collection.␣␣Preprocessing
is␣typically␣used␣and␣has␣a␣low␣data␣rate.␣␣Raw␣data
is␣for␣special␣testing␣and␣comes␣at␣a␣very␣high␣rate." \
                12 60 3 \
                1 "Collect␣preprocessed␣data" \
                2 "Collect␣raw␣data" \
                3 "Go␣back" 2>menu-system.tmp2.$$

                result2 =`cat menu-system.tmp2.$$`
                rm -f menu-system.tmp2.$$

                case ␣$result2 in
                    ␣1)
                        # Preprocessed
                        gdialog --infobox \
                            "Starting␣data␣collection,␣please␣wait ... " 3 60
                        sh scripts/start-run
                        ;;
                    ␣2)
                        # Raw
                        if IFS='' gdialog --title "Collect␣Raw␣Data" --yesno\
                            "Are␣you␣sure?␣␣This␣will␣generate␣a␣lot␣of␣data."\
                            4 60 ; then
                        gdialog --infobox \
                            "Starting␣raw␣data␣collection,␣please␣wait ... " 3 60
                        sh scripts/start-run raw
                        fi
                        ;;
                    *)
                        # go back
                        ;;
                esac
        else
            gdialog --infobox "Stopping␣data␣collection,␣please␣wait..." 3 60
            sh scripts/stop-run
        fi
        ;;
    ␣3)
        gdialog --infobox "Creating␣and␣compressing␣snapshot,␣please␣wait..." \
            3 60
        sh scripts/snapshot-data
        ;;
    ␣4)
        sh scripts/burn-cd
        ;;
    ␣5)
        clear
        echo "Type␣'exit'␣to␣return␣to␣the␣menu␣system"
        echo "Type␣'halt'␣to␣shut␣down␣the␣computer"
        SKIPMENU=1 /bin/bash -login
        ;;
    *)
        # Cancel, or some error; just die (init should restart us)
        exit
        ;;
esac

done
```

run-prep.pl

Executes the capture and preprocessor and adds comments to the output.

```
#!/usr/bin/perl -w

# run-prep.pl 2003-06-05
# Jim Paris <jim@jtan.com>
#
# Run preprocessor on the PCI1710 output, saving data to a file.
# A modified version of prep is used that flushes the output buffer
# after each line.  A SIGHUP will cause this script to reopen the
# output file.
#
# Give an argument of "raw" to skip the preprocessor and store raw data.
#
# If the output file exists, it will be appended to.
#
# When this script starts, is sent a SIGHUP, or terminated with
# SIGTERM or SIGINT, it includes a header or footer giving a
# timestamp. Header and footer lines are prefixed with '#'
#
# Timestamp labels:
# start - first start capture
# startraw - first start capture (raw data)
# stop - ending capture
# reopening - about to start writing to a new file
# reopened - just started writing to a new file

use strict;
use POSIX qw(strftime);

# Prep may generate a gmon.out, so go somewhere that it can be written.
chdir "/home/nilm";

my $rawmsg="";

my $hostname=`hostname` || "nilm"; chomp $hostname;

my $prepcmd="/home/nilm/bin/prep /dev/pci1710 -p /home/nilm/lib/prep-msufp.so";

if($ARGV[0] eq 'raw') {
    $rawmsg = " raw";
    $prepcmd = "/bin/sh -c '/bin/dd if=/dev/pci1710 | /home/nilm/bin/pci2asc'"
}

# Where to put the output
my $output = "/home/nilm/data/current";

$|=1;

open(PREP,$prepcmd . "|")
    or die "can't execute '$prepcmd': $!";

sub write_timestamp {
    my $desc = shift;
    print OUTPUT "# $hostname $desc = " .
        strftime("%Y%m%d-%H%M%S",localtime) . "\n";
}

sub open_output {
    open(OUTPUT,">>" . $output)
        or die "can't append to $output: $!";
};

sub reopen_output {
    write_timestamp("reopening$rawmsg");
    open_output;
    write_timestamp("reopened$rawmsg");
};

sub term_handler {
    write_timestamp("stop$rawmsg");
    close OUTPUT;
    close PREP;
    print "Terminating.\n";
    exit;
};
```

```
$SIG{HUP} = \&reopen_output;
$SIG{TERM} = $SIG{INT} = \&term_handler;

open_output;
print OUTPUT "#␣$hostname␣prepcmd␣=␣$prepcmd\n";
write_timestamp("start$rawmsg");

print "Running.\n";

print OUTPUT $_ while(<PREP>);

print "Warning:␣reached␣EOF\n";

write_timestamp("stop$rawmsg");

close OUTPUT;
close PREP;
```

start-run

Starts the `run-prep.pl` process if it is not already running.

```
#!/bin/sh

# start-run 2003-06-05
# Jim Paris <jim@jtan.com>
#
# Start executing run-prep.pl in the background if it's not already running.

preplog=/home/nilm/logs/run-prep

# Redirect to output
exec >>/home/nilm/logs/start-run 2>&1

echo /home/nilm/scripts/start-run $1 > /home/nilm/last-command

if killall -q -0 run-prep.pl ; then
        echo Error: run-prep.pl is already running on `date`
        exit 1
fi

/home/nilm/scripts/initialize
/home/nilm/scripts/run-prep.pl $1 >>"$preplog" 2>&1 &

sleep 2 # give it time to start
```

stop-run

Stops the `run-prep.pl` process if it is running.

```
#!/bin/sh

# stop-run 2003-06-05
# Jim Paris <jim@jtan.com>
#
# Stop executing run-prep.pl if it's running

# Redirect to output
exec >>/home/nilm/logs/stop-run 2>&1

echo /home/nilm/scripts/stop-run > /home/nilm/last-command

if ! killall -q -TERM run-prep.pl ; then
        echo Error: run-prep.pl isn\'t running on `date`
        exit 1
fi

sleep 2 # give it time to stop
```

snapshot-data

Creates, names, and compresses a snapshot of the current capture data.

```
#!/bin/sh

# snapshot-data 2003-06-05
# Jim Paris <jim@jtan.com>
#
# Take a "snapshot" of the prep output.
# This is meant to be run on a regular basis via cron, or
# manually (creates a lock)

prep_output="/home/nilm/data/current"
snapshot_dir="/home/nilm/data"
snapshot_name=`date +"snapshot-%Y%m%d-%H%M%S"`

# Redirect to output
exec >>/home/nilm/logs/snapshot-data 2>&1

#duh
# if killall -q -0 snapshot-data ; then
#     echo "snapshot-data already running; not creating $snapshot_name"
#     sleep 1
#     exit
# fi

if [ ! -e "$prep_output" ] ; then
```

```
    echo "No output found; not creating $snapshot_name"
    sleep 1
    exit
fi

# Move the output and tell run-prep to start making a new one.
mv -- "$prep_output" "$snapshot_dir/$snapshot_name"
killall -q -HUP run-prep.pl # OK if this fails.

# Compress it
gzip -- "$snapshot_dir/$snapshot_name"

# And we're done.
sleep 1  # so they can't snapshot more than once per second
```

burn-cd

Interactively compiles and burns CD and DVD images.

```
#!/bin/sh

# burn-cd 2003-06-06
# 2005-01-05 added dvd support
# Jim Paris <jim@jtan.com>
#
# Put snapshots into an ISO image on a CD and remove them,
# with interactive dialogs.

unset DISPLAY

exec 2>>/home/nilm/logs/burn-cd

echo "burn-cd starting on `date`" 1>&2

# Gather a list of snapshots in name-order that will fit.
first_snapshot=
last_snapshot=
snapshot_list=
total_size=0
cd /home/nilm
if ! ls data | grep -q snapshot ; then
    gdialog --title "Error" --msgbox "No snapshot files available." 5 50
    exit
fi

IFS='' gdialog --title 'Burn Disc' --menu \
    "Would you like to burn a CD or DVD?" \
    12 60 3 \
    1 "Burn CD-R" \
    2 "Burn DVD+R or DVD+RW" \
    3 "Go back" 2>menu-system.tmp2.$$
```

```
result2=`cat menu-system.tmp2.$$`
rm -f menu-system.tmp2.$$

case ␣$result2 in
    ␣1)
        disc=CD
        discdesc="CD-R"
        discsize=650000
        ;;
    ␣2)
        disc=DVD
        discdesc="DVD+R␣or␣DVD+RW"
        discsize=4500000
        ;;
    *)
        exit
        ;;
esac

for snapshot in `ls data/snapshot*`; do
    [ x$first_snapshot = x ] && first_snapshot=$snapshot;
    size=`du -lk --apparent-size $snapshot | cut -f 1`
    [ $size = 0 ] && size=1
    if [ $(($size + $total_size)) -lt $discsize ] ; then
        last_snapshot=$snapshot
        snapshot_list="$snapshot_list␣$snapshot"
        total_size=$(($total_size + $size))
    else
        break
    fi
done

if [ "x$snapshot_list" = "x" ] ; then
    gdialog --title "Error" --msgbox "Snapshots␣too␣big␣for␣$disc!" 5 50
    exit
fi

trap "rm␣-rf␣burn-cd-tmp-$$;␣echo␣exiting␣1>&2;␣exit" 0 1 2 11 15
cat >burn-cd-tmp-$$ <<EOF
 First file : `basename $first_snapshot`
  Last file : `basename $last_snapshot`
Total files : `echo $snapshot_list | wc -w | tr -d ' '`
 Total size : ${total_size}K
EOF
if ! IFS="" gdialog --title "Build␣$disc␣image?" \
    --yesno "`cat␣burn-cd-tmp-$$`" 8 60 ; then
    exit
fi

# Link to them in a temporary directory
trap "rm␣-rf␣burn-cd-{tmp,tmpdir}-$$;␣echo␣exiting␣1>&2;␣exit" 0 1 2 11 15
if ! mkdir -p burn-cd-tmpdir-$$ ; then
    gdialog --title "Error" \
                --msgbox 'Error making temporary directory; try again' 5 60
    exit
fi
for i in $snapshot_list ; do ln -f $i burn-cd-tmpdir-$$/`basename $i` ; done

# Build the ISO filename
# eg snapshot-20030601-123456-through-20030603-012345.iso
 isofirst =`cd burn-cd-tmpdir-$$;ls | head -1 | sed -e 's/snapshot-//'`
 isofirst =`basename $isofirst .gz`
 isolast =`cd burn-cd-tmpdir-$$;ls | tail -1 | sed -e 's/snapshot-//'`
 isolast =`basename $isolast .gz`
isoname="snapshot-$isofirst-through-$isolast.iso"
isopath="burn-cd-tmpdir-$$/$isoname"

echo "iso $isoname containing files $snapshot_list" 1>&2

# Run mkisofs
if ! IFS=" mkisofs -gui -J -r -no-cache-inodes -o $isopath \
    $snapshot_list 2>&1 | \
    perl -e '$|=1;while(<>){/ *(\d+)[.]\d+% done/ and print "$1\n";}' | \
    gdialog --title "Building $disc image, please wait..." \
        --guage "`cat burn-cd-tmp-$$`" 9 60 0 ; then
    echo "mkisofs returned error code $?" 1>&2
    gdialog --title "Error" \
                --msgbox "An error occured while generating the $disc image." 5 60
    exit
fi
sleep 1 # so the dialog box stays up for a sec

# OK, now we have CD image in $isopath -- let's burn it.

# Make sure the dudes insert a CD
if ! gdialog --title "Ready" --msgbox "The $disc image is ready to burn. \
Please insert a $discdesc in the drive \
and hit ENTER to continue." 7 60 ; then
    echo "ctrl-c before burn" 1>&2
    exit
fi

# Do this in a loop so they can burn it again in case of problem, etc.
burned=0
while true ; do
    clear
    B="\033[;1m" # Bold
    N="\033[;0m" # Normal
    echo -e "${B}Writing data to $disc.  Please wait.${N}"
    eject -t /dev/scd0
    if [ $disc = CD ] ; then
        if cdrecord dev=0,0,0 speed=24 gracetime=0 -v $isopath ; then
            echo "cdrecord success, return code $?" 1>&2
            success=1
        else
            echo "cdrecord failed, error code $?" 1>&2
            success=0
            fi
        else
            if growisofs -speed=4 -dvd-compat -Z /dev/scd0=$isopath ; then
                echo "growisofs␣success,␣return␣code␣$?" 1>&2
                success=1
            else
                echo "growisofs␣failed,␣error␣code␣$?" 1>&2
                success=0
            fi
        fi
        eject /dev/scd0

        if [ $success = 0 ] ; then
            sleep 2 # so the error doesn't vanish immediately
            if ! gdialog --title "Error␣burning␣$disc" --yesno "An␣error␣\
occured␣while␣burning␣the␣$disc.␣␣There␣may␣be␣a␣problem␣with␣the␣media,␣\
or␣this␣may␣just␣be␣a␣temporary␣problem.␣␣Would␣you␣like␣to␣try␣again?" \
                7 60 ; then
                echo "user␣wants␣to␣give␣up" 1>&2
                break
            fi
            echo "user␣wants␣to␣keep␣trying" 1>&2
            retrytitle ="Retry"
                # fall down
        else
            burned=$(($burned + 1))
            if ! gdialog --title 'Success!' \
                --yesno "The␣$disc␣was␣successfully␣written.␣␣It␣now␣contains␣\
snapshot␣data␣from␣$isofirst␣through␣$isolast.␣␣Would␣you␣like␣to␣burn␣\
a␣duplicate␣$disc␣containing␣the␣same␣snapshot␣data?" 8 60 ; then
                echo "user␣doesn't␣want␣any␣more␣cds" 1>&2
                break
            fi
            retrytitle ="Duplicate"
        fi
        if ! gdialog --title $retrytitle --msgbox "Please␣insert␣a␣new␣\
$discdesc␣in␣the␣drive␣and␣hit␣ENTER␣to␣continue." 6 60 ; then
            echo "ctrl-c␣before␣retry␣or␣duplicate␣burn" 1>&2
            break
        fi
done

if [ $burned -gt 0 ] ; then

        # Do we want to give them this option?
#       count="$burned CDs";
#       [ $burned -eq 1 ] && count="1 CD";
#       if ! gdialog --title 'Remove old snapshots' \
#               --yesno "Snapshot data from $isofirst through $isolast has now \
#been archived to $count.  Would you like to remove this data from the \
#active directory?  It needs to be removed before you can burn later \
#snapshots to CD." 9 60 ; then
#               echo "user wants to save  files " 1>&2
#               exit
#    fi

    gdialog --title "Please wait" --infobox "Removing old snapshots..." 3 50

    # Move the burned iso into /home/nilm/burned
    mv -f $isopath /home/nilm/burned/`basename $isopath`

    # And remove the snapshots.
    rm -rf burn-cd-tmpdir-$$
    for i in $snapshot_list ; do rm $i ; done

    sleep 1 # so the dialog box stays up for a sec
fi

echo "burn-cd ending on `date`, burned=$burned" 1>&2
```

check-diskspace

Periodically checks available space and removes outdated files as necessary.

```
#!/bin/sh

# check-diskspace 2003-06-06
# Jim Paris <jim@jtan.com>
#
# If disk space is low (less than a gig), remove an old burned ISO

exec >>/home/nilm/logs/check-diskspace 2>&1

if [ `df -m /dev/hda1 | awk '/hda1/{print $4}'` -lt 1024 ] ; then
    rmfile=`ls -tr /home/nilm/burned/*iso | head -1`
    echo "Disk space is low; removing burned file $rmfile"
    rm -f $rmfile
fi
```

A.3.2 nilmgui

nilmplugin.h

Header file describing the `nilmgui` plugin architecture and interface.

```
#ifndef NILMPLUGIN_H
#define NILMPLUGIN_H

/* Plugin structures are filled in by each plugin.
   Nonexistant or unimplemented functions should be set to NULL */

typedef struct {
        char *name;
        /* The "value" pointers will be deallocated on a destroy, so
           free it and dynamically allocate the memory when changing */
        char *value;
} plugin_config_item;

typedef struct {
        int num_items;
        plugin_config_item *pci;
} plugin_config;

/* Functions that return int return nonzero on success */
typedef struct {
        char *name;
        char *desc;

        /* Initialize ; called before anything */
        int (*init)(void);

        /* Deinitialize ; called after everything */
        void (*destroy)(void);

        /* Plugin configuration; call set_config to inform the
           plugin after making changes. */
        plugin_config *(*get_config)(void);
        int (*set_config)(void);

        /* Start or stop capturing data */
        int (*start)(void);
        int (*stop)(void);

        /* Optional fd to watch; if non-NULL, host will
           call process whenever it has data available */
        int *watchfd;
        int (*process)(void);

        /* Query how many items can be read */
        int (*query)(void);

        /* Read at least 1 and up to n records; returns number
           actually read (blocking) */
        int (*read)(int n, float (*data)[8]);
} inputplugin;

#endif
```

cmdline.c

Implementation of the cmdline input plugin for nilmgui data.

```
#include "cmdline.h"
```

```
#include <sched.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <signal.h>
#include "xpopen.h"

#ifndef max
#define max(a,b) ((a)>(b)?(a):(b))
#endif
#ifndef min
#define min(a,b) ((a)<(b)?(a):(b))
#endif

int initialize (void);
void destroy(void);
plugin_config *get_config(void);
int set_config(void);
int start(void);
int stop(void);
int process(void);
int query(void);
int pluginread(int n, float (*data)[8]);
int watchfd;
FILE *output;
int childpid;
int running;

#define MAXLEN 1024
char buf[MAXLEN+1];
int buflen=0;
float databuf[MAXLEN][8];
int databufstart=0, databufend=0;

inputplugin plugin_info = {
        .name =         "cmdline",
        .desc =         "Execute the given shell commands to generate data",
        .init =         &initialize,
        .destroy =      &destroy,
        .get_config =   &get_config,
        .set_config =   &set_config,
        .start =        &start,
        .stop =         &stop,
        .watchfd =      &watchfd,
        .process =      &process,
        .query =        &query,
        .read =         &pluginread,
};

inputplugin *get_iplugin_info(void) { return &plugin_info; }

plugin_config pc;

int initialize (void)
```

```
{
        watchfd=-1;

        pc.num_items = 1;
        pc.pci = (plugin_config_item *)malloc(sizeof(plugin_config_item) * pc.num_items);
        if(pc.pci==NULL)
                return 0;
        pc.pci[0].name="Command_line";
        pc.pci[0].value=strdup("/bin/false");

        running=0;

        return 1;
}

void destroy(void)
{
        if(pc.pci[0].value!=NULL)
                free(pc.pci[0].value);
        if(pc.pci!=NULL)
                free(pc.pci);
}

plugin_config *get_config(void)
{
        return &pc;
}

int set_config(void)
{
        printf("new_commandline: %s\n",pc.pci[0].value);
        if(running) {
                if(!stop())
                        return 0;
                if(!start())
                        return 0;
        }
        return 1;
}

int start(void)
{
        if(running) stop();

        running=0;

        if((output=xpopen(pc.pci[0].value,"r",&childpid))==NULL) {
                perror("popen");
                return 0;
        }
        sched_yield();
        if(waitpid(childpid,NULL,WNOHANG)==childpid) {
                fprintf(stderr,"Subprocess exited prematurely\n");
                return 0;
        }
        watchfd=fileno(output);

        running=1;

        return 1;
}

int stop(void)
{
        if(!running) return 1;

        (void)kill(childpid,SIGHUP);
        sched_yield();
        (void)kill(childpid,SIGKILL);
        xpclose(output,childpid);

        running=0;

        return 1;
}

int process(void)
{
        int len;
        char *c;
        static int justwarned=0;

        if(feof(output)) {
                printf("reached end of file, closing pipe\n");
                pclose(output);
                return 0;
        }

        if(!running || feof(output) || buflen>=MAXLEN) {
                printf("process returning, can't read\n");
                return 0;
        }

        if((len=read(fileno(output),buf+buflen,MAXLEN-buflen))<=0) {
                if(len==0) {
                        printf("called process but no data was ready\n");
                        return 0;
                } else {
                        printf("read failed: error=%s\n",strerror(errno));
                        return 0;
                }
        }
        buflen+=len;

        buf[buflen]=0;
        while((c=strchr(buf,'\n'))!=NULL) {
                int i;
                int off=0, j;
                *c='\0';
                for(i=0;i<8;i++) {
                        j=0;
                        if(sscanf(buf+off,"%f%n",
                                  &databuf[databufend][i],&j)<1) {
                                fprintf(stderr,"malformed␣input\n");
                                return 0;
//                              databuf[databufend][i]=0;
                        }
                        off+=j;
                }
                if(((databufend+1)%MAXLEN)==databufstart) {
                        if(!justwarned)
                                fprintf(stderr,"buffer␣overflow;␣"
                                        "discarding␣data\n");
                        justwarned=1;
                }
                else {
                        databufend=(databufend+1)%MAXLEN;
                        justwarned=0;
                }

                buflen-=(c+1-buf);
                memmove(buf,c+1,buflen);
                buf[buflen]=0;
        }
        return 1;
}

int query(void)
{
        return ((databufend+MAXLEN)-databufstart)%MAXLEN;
}

int pluginread(int n, float (*data)[8])
{
        int i,j;
        int len = min(n,query());
        for(i=0;i<len;i++)
                for(j=0;j<8;j++)
                        data[i][j]=databuf[(i+databufstart)%MAXLEN][j];
        databufstart = (databufstart + len)%MAXLEN;
        return len;
}
```

`nilmgui.cpp`

Main `nilmgui` application implementation.

```
#include <qapplication.h>
```

```
#include <qtextedit.h>
#include <qfile.h>
#include <unistd.h>
#include <signal.h>

#include <iostream>
using namespace std;

#include <nilmimp.h>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    app.connect(&app, SIGNAL(lastWindowClosed()), &app, SLOT(quit()));

    NilmImp *win = new NilmImp();
    win->show();

    printf("input_plugin_dir: %s\n",INPUT_PLUGIN_DIR);

    return app.exec();
}
```

nilmimp.h

Main window drawing and event handling, header file.

```
#ifndef NILMIMP_H
#define NILMIMP_H

#include "nilmwindow.h"
#include "trainingparameters.h"
#include <nilmplugin.h>
#include <qsocketnotifier.h>
#include "classifier.h"
#include <qarray.h>
#include "nilmtypes.h"

class ExemplarDialog;

class NilmImp : public NilmWindow
{
        Q_OBJECT

public:

        NilmImp( QWidget* parent = 0, const char* name = 0, WFlags f = 0 );
        ~NilmImp();
public slots:
        //
        void fileSave ();
        void fileExit ();
        void editUndo();
        void editRedo();
        void helpAbout();
        void menubar_activated(int);
        void doInit();
        void doSave();
        void doLoad();
        void inputDataAvailable();
        void doExemplar();
        void doPause(bool);
        void doConfig();
        void zoomFit();
        void matchFound(QString name,
                        double quality, double scale);


signals:
        void newData(QArray<dataPoint> da);

private:
        ExemplarDialog *ed;
        QSocketNotifier *sn;
        inputplugin *ip;
        void *handle;
        Classifier * classify ;

public:
        TrainingParameters trainingparameters;
        double classify_tolerance;
};

#endif
```

nilmimp.cpp

Main window drawing and event handling, implementation.

```
#include "nilmimp.h"
#include <qvalidator.h>
#include "nilmwindow.h"
#include "nilmconfigbase.h"
#include "plot.h"
#include <qtextedit.h>
#include <qmessagebox.h>
#include "trainingparameters.h"
#include <qsocketnotifier.h>
#include <qfiledialog.h>
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>
```

```
#include <string.h>
#include <unistd.h>
#include <sys/select.h>
#include <signal.h>
#include <qstring.h>
#include "nilmtypes.h"
#include "exemplardialog.h"
#include <qvariant.h>
#include <qpushbutton.h>
#include <qlabel.h>
#include <qlineedit.h>
#include <qlayout.h>
#include <qtooltip.h>
#include <qwhatsthis.h>
#include <qsplitter.h>

#define infobox(...) infoBox->append(QString().sprintf( __VA_ARGS__ ))
#define info(...) do{printf(__VA_ARGS__);printf("\n");}while(0)

NilmImp::NilmImp(QWidget* parent, const char* name, WFlags f )
        : NilmWindow( parent, name, f )
{
        ip=NULL;
        classify_tolerance = 0.95;

        ed = new ExemplarDialog(this,"Exemplars",false);

        QValueList<int> sizes;
        sizes << height()*2/3 << height()*1/3; // to start
        NilmWindow::splitter2->setSizes(sizes);

        doInit ();
}

NilmImp::~NilmImp()
{
        delete ed;
}

void NilmImp::fileSave() {}
void NilmImp::fileExit() {}
void NilmImp::editUndo() {}
void NilmImp::editRedo() {}
void NilmImp::helpAbout() {}
void NilmImp::menubar_activated(int) {}

void NilmImp::doConfig()
{
        NilmConfigBase *cfg = new NilmConfigBase(this,"Config",true);
        // QDoubleValidator doesn't actually prevent them from going out of range,
        // but it keeps characters outta there.
        cfg->varianceEdit->setText(QString::number(trainingparameters.settle_variance_max));
        cfg->extraEdit->setText(QString::number(trainingparameters.extra_transient_length));
        cfg->matchEdit->setText(QString::number(classify_tolerance));
        cfg->varianceEdit->setValidator(new QDoubleValidator(0,1e9,10,cfg->varianceEdit));
        cfg->extraEdit->setValidator(new QIntValidator(0,500,cfg->extraEdit));
        cfg->matchEdit->setValidator(new QDoubleValidator(0,1,10,cfg->matchEdit));
        if(cfg->exec() == QDialog::Accepted) {
                double tmp;
                int tmp2;
                bool ok;
                tmp = cfg->varianceEdit->text().toDouble(&ok);
                if(ok && tmp > 1e-6 && tmp < 1e6)
                        trainingparameters.settle_variance_max = tmp;
                tmp2 = cfg->extraEdit->text().toInt(&ok);
                if(ok && tmp2>0 && tmp2<500)
                        trainingparameters.extra_transient_length = tmp2;
                tmp = cfg->matchEdit->text().toDouble(&ok);
                if(ok && tmp > 0 && tmp < 1)
                        classify_tolerance = tmp;
        }
        delete cfg;
}

void NilmImp::doExemplar()
{
        if(ed->isVisible())
                ed->hide();
        else
                ed->show();
}

void NilmImp::inputDataAvailable()
{
        if(!ip->process()) {
                info("Processing failed (end of data?); shutting down input plugin");
                ip->stop();
                ip->destroy();
                dlclose(handle);
                ip=NULL;
                delete sn;
//              delete classify;
                disconnect(this,SIGNAL(newData(QArray<dataPoint>)),0,0);
                return;
        }

        while(ip->query()>0) {
                float d[128][8];
                int count = ip->read(128,d);
                QArray<dataPoint> da(count);
                for(int i=0;i<count;i++)
                        for(int j=0;j<8;j++)
                                da[i][j]=d[i][j];
                newData(da);
//              plotBox->addData(d,count);
        }
}
```

```
void NilmImp::doPause(bool p)
{
        plotBox->setPause(p);
}

void NilmImp::zoomFit()
{
        plotBox->zoomFit();
}

void NilmImp::doSave()
{
        if(ip==NULL || classify==NULL) return;
        int size = classify->getExemplars().size();
        if(size==0) {
                QMessageBox::warning(this,"Save","No exemplars to save!",
                                     QMessageBox::Ok,QMessageBox::NoButton);
                return;
        }
trysaveagain:
        QString filename = QFileDialog::getSaveFileName(
                QString::null,
                "Exemplars (*.ex);;All Files (*)",
                this,
                "save dialog",
                QString().sprintf("Save %d exemplar%s in...",size,(size==1)?"":"s"));

        if(filename==QString::null) return;

        if(QFile::exists(filename) &&
           QMessageBox::question( this, "Save Exemplars",
                                 filename + " already exists.  Overwrite?",
                                 "&Yes", "&No")==1)
                goto trysaveagain;

        printf("saving in %s\n",(const char *)filename);

        QFile out(filename);
        QString errormessage = "Can't create " + filename;
        if(out.open(IO_WriteOnly)) {
                QTextStream stream(&out);
                stream << "NILM Exemplar File Format v1.0\n";
                stream << size << "\n";
                for(int i=0;i<size;i++) {
                        if( classify->getExemplars()[i]->write(stream)==false) {
                                errormessage = "Error writing to " + filename;
                                goto saveerror;
                        }
                }
                out.close();
                infobox("Saved %d exemplar%s in %s",size,
                        (size==1)?"":"s",(const char *)filename);
                return;
        }
```

```
saveerror:
        QMessageBox::critical(this,"Save",errormessage,
                              QMessageBox::Ok,QMessageBox::NoButton);
}

void NilmImp::doLoad()
{
        if(ip==NULL || classify==NULL) return;
        int size = classify->getExemplars().size();
        if(size!=0) {
                QString num;
                if(size==1) num="exemplar";
                else num.sprintf("%d exemplars",size);
                if(QMessageBox::question( this, "Load Exemplars",
                                          "Loading will overwrite the " + num +
                                          " currently in memory.  Continue?",
                                          "&Yes", "&No")==1)
                        return;
        }

        QString filename = QFileDialog::getOpenFileName(
                QString::null,
                "Exemplars (*.ex);;All Files (*)",
                this,
                "load dialog",
                "Load exemplars from...");

        if(filename==QString::null) return;

        QPtrVector<Exemplar> e;

        printf("loading from %s\n",(const char *)filename);

        QFile in(filename);
        QString errormessage = "Can't open " + filename;
        if(in.open(IO_ReadOnly)) {
                errormessage = "Error parsing contents of " + filename;

                QTextStream stream(&in);
                QString t;

                if((t=stream.readLine())==QString::null ||
                   t!="NILM Exemplar File Format v1.0") goto loaderror;

                int size = 0;
                stream >> size;
                if(size<=0 || !e.resize(size)) goto loaderror;

                for(int i=0;i<size;i++) {
                        printf("loading exemplar %d/%d\n",i,size);
                        e.insert(i, new Exemplar());
                        if(!e[i]->read(stream)) {
                                for(int j=0;j<=i;j++)
                                        delete e[i];
                                        goto loaderror;
                                }
                        }
                        in.close();
                        for(unsigned int i=0;i<classify->getExemplars().size();i++)
                                delete classify->getExemplars()[i];
                        classify->setExemplars(e);
                        ed->setExemplars(e);
                        infobox("Loaded %d exemplar%s from %s",size,
                                (size==1)?"":"s",(const char *)filename);
                        return;
                }
loaderror:
                QMessageBox::critical(this,"Save",errormessage,
                                        QMessageBox::Ok,QMessageBox::NoButton);

}

void NilmImp::matchFound(QString name,
                        double quality, double scale)
{
        infobox("Matched event: " +
                QString().sprintf("(%lf, %lf)\t",
                                quality,scale) +
                name);
}

void NilmImp::doInit()
{
        printf("in doInit()\n");

        if(ip!=NULL) {
                info("Already initialized!\n");
                return;
        }

        inputplugin *(*get_iplugin_info)(void);
        plugin_config *pc;
        char *error;

        info("opening %s/libcmdline.so",INPUT_PLUGIN_DIR);
        handle=dlopen(INPUT_PLUGIN_DIR "/libcmdline.so", RTLD_NOW);
        if(!handle) {
                info(dlerror());
                return;
        }

        get_iplugin_info = (inputplugin *(*)())dlsym(handle, "get_iplugin_info");
        if((error=dlerror())!=NULL) {
                info(error);
                return;
        }

        ip = (*get_iplugin_info)();
```

```
info("name: %s",ip->name);
info("description: %s",ip->desc);

ip->init();

pc=ip->get_config();
info("plugin has %d config items",pc->num_items);
{
        int i;
        for(i=0;i<pc->num_items;i++) {
                printf("%d name: %s\n",i,pc->pci[i].name);
                printf(" value: %s\n",pc->pci[i].value);
                if(strcasecmp(pc->pci[i].name,"Command line")==0) {
                        char *mycmd = "/usr/local/bin/nilmgui-data";
                        info("setting commandline to: %s",mycmd);
                        free(pc->pci[i].value);
                        pc->pci[i].value=strdup(mycmd);
                }
        }
}
ip->set_config();

info("starting plugin");
if(ip->start()==0 || ip->watchfd==NULL || *ip->watchfd<0) {
        info("start failed, can command be executed?");
        ip->destroy();
        ip=NULL;
        dlclose(handle);
        return;
}

info("requesting notifications on fd %d",*ip->watchfd);

sn = new QSocketNotifier( *ip->watchfd, QSocketNotifier::Read, this );
QObject::connect( sn, SIGNAL(activated(int)), this, SLOT(inputDataAvailable()) );

info("creating classifier object");
classify = new Classifier(&classify_tolerance);

info("connecting data to receiver, classifier, and trainer");
connect(this,SIGNAL(newData(QArray<dataPoint>)),
        plotBox,SLOT(addData(QArray<dataPoint>)));
connect(this,SIGNAL(newData(QArray<dataPoint>)),
        classify,SLOT(addData(QArray<dataPoint>)));
connect(this,SIGNAL(newData(QArray<dataPoint>)),
        ed,SIGNAL(newData(QArray<dataPoint>)));

info("connecting ExemplarDialog::newExemplars to Classifier::setExemplars");
connect(ed,SIGNAL(newExemplars(QPtrVector<Exemplar>)),
        classify,SLOT(setExemplars(QPtrVector<Exemplar>)));

info("connecting match event to us");
connect(classify,SIGNAL(matchFound(QString,double,double)),
                this,SLOT(matchFound(QString,double,double)));

        info("telling␣ed␣to␣tell␣classifier␣about␣exemplars");
        ed->refreshExemplars();

        info("Initialization␣complete.");
        infobox("Initialized.");

}
```

---

plot.h

Graphical window providing the scrolling waveform plot, header file.

```
#ifndef PLOT_H
#define PLOT_H

#include <qwt/qwt_plot.h>
#include "nilmtypes.h"
#include <qarray.h>

const int plotpoints = 1024;

class NilmPlot: public QwtPlot
{
        Q_OBJECT
public:
        NilmPlot(QWidget *parent, const char *title);
        ~NilmPlot();

public slots:
        void addData(QArray<dataPoint>);
        void setPause(bool);
        void zoomFit(bool shrink=true);
        void toggleCurve(long key);

protected:
        void drawCanvas(QPainter *p);

private:
        void updateCurvePens();

        double miny, maxy;

        bool paused;

        long curve_key[8];
        bool curve_shown[8];

        double *t, **data;
};
```

```
#endif
```

plot.cpp

Graphical window providing the scrolling waveform plot, implementation.

```
#include "plot.h"

#include <qwt/qwt_plot_canvas.h>
#include <qwt/qwt_plot_dict.h>
#include <qwt/qwt_legend.h>

const QString curve_title[8] =
{ "P", "Q", "3P", "3Q", "5P", "5Q", "7P", "7Q" };
const QColor curve_color[8] =
{ Qt::yellow, Qt::cyan, Qt::green, Qt::red,
  Qt::magenta, Qt::darkCyan, Qt::blue, Qt::darkMagenta };

NilmPlot::NilmPlot(QWidget *parent, const char *title):
        QwtPlot(parent)
{
        (void) title;

        connect(this,SIGNAL(legendClicked(long)),this,SLOT(toggleCurve(long)));

        setCanvasBackground(black);
        legend()->setPaletteBackgroundColor(Qt::black);
        legend()->setPaletteForegroundColor(Qt::white);

        // legend
        setAutoLegend(FALSE);
        enableLegend(FALSE);
        setLegendPos(Qwt::Right);
        setLegendFrameStyle(QFrame::Box|QFrame::Sunken);

        // grid
        setGridMajPen(QPen(gray, 0, SolidLine));

        // axes
        setAxisTitle(QwtPlot::xBottom, "Time (s)");
        setAxisScale(QwtPlot::xBottom, 0, 10.24);
        setAxisTitle(QwtPlot::yLeft, "Power");

        for(int i=0;i<8;i++) {
                curve_key[i] = insertCurve(curve_title[i]);
                setCurveYAxis(curve_key[i], QwtPlot::yLeft);
                curve_shown[i] = false;
        }
        curve_shown[0] = true;
        curve_shown[1] = true;
        updateCurvePens();

        t = new double[plotpoints];
        data = new double *[8];
        for(int j=0;j<8;j++)
                data[j] = new double[plotpoints];

        for(int i=0;i<plotpoints;i++) {
                for(int j=0;j<8;j++)
                        data[j][i]=0;
                t[i]=i/100.0;
        }

        miny = 0;
        maxy = 500;

        paused=false;

        QArray<dataPoint> tmp;
        addData(tmp);
}

void NilmPlot::toggleCurve(long key)
{
        for(int i=0;i<8;i++)
                if(curve_key[i]==key)
                        curve_shown[i]=!curve_shown[i];
        updateCurvePens();
        zoomFit(false);
}

void NilmPlot::updateCurvePens()
{
        for(int i=0;i<8;i++) {
                setCurvePen(curve_key[i], QPen(curve_color[i], 0,
                                           curve_shown[i] ? SolidLine : NoPen));
                setCurveTitle(curve_key[i], curve_title [i] +
                              (curve_shown[i] ? QString(" *") : QString("")));
        }
}

NilmPlot::~NilmPlot()
{
        delete[] t;
        for(int j=0;j<8;j++)
                delete[] data[j];
        delete[] data;

        disconnect(this,SIGNAL(legendClicked(long)),this,SLOT(toggleCurve(long)));
}

void NilmPlot::setPause(bool p)
{
        paused = p;
```

```
}

void NilmPlot::zoomFit(bool shrink)
{
        if(shrink) {
                miny = 0;
                maxy = 100;
        }
        for(int i=0;i<8;i++) {
                if(curve_shown[i]) {
                        if(curve(curve_key[i])->maxYValue() > maxy)
                                maxy = curve(curve_key[i])->maxYValue();
                        if(curve(curve_key[i])->minYValue() < miny)
                                miny = curve(curve_key[i])->minYValue();
                }
        }
        setAxisScale(QwtPlot::yLeft, miny-(maxy-miny)*0.05, maxy+(maxy-miny)*0.05);
        replot ();
}

void NilmPlot::addData(QArray<dataPoint> dp)
{
        int count = dp.size();
        double tmp;
        int i, j;

        if(count>=plotpoints) count=plotpoints;

        for(j=0;j<8;j++)
                memmove(&data[j][0],
                        &data[j][count],
                        (plotpoints-count)*sizeof(**data));

        for(j=0;j<8;j++) {
                for(i=0;i<count;i++) {
                        tmp = dp[i][j];
                        if(curve_shown[j]) {
                                if(tmp<miny) miny=tmp;
                                if(tmp>maxy) maxy=tmp;
                        }
                        data[j][plotpoints-count+i]=tmp;
                }
        }
        if(paused) return;

        setAxisScale(QwtPlot::yLeft, miny-(maxy-miny)*0.05, maxy+(maxy-miny)*0.05);

        for(j=0;j<8;j++) {
                setCurveData(curve_key[j], t, data[j], plotpoints);
        }
        replot ();
}

/* This function is mostly identical to that in QwtPlot,
   but draws curves in reverse order */

void NilmPlot::drawCanvas(QPainter *p)
{
    QwtDiMap map[axisCnt];
    for ( int axis = 0; axis < axisCnt; axis++ )
        map[axis] = canvasMap(axis);

    QRect rect = canvas()->contentsRect();

    //
    // draw grid
    //
    if ( grid().enabled() &&
         axisEnabled( grid().xAxis() ) &&
         axisEnabled( grid().yAxis() ) )
    {
        grid().draw(p, rect, map[grid().xAxis()], map[grid().yAxis()]);
    }

    //
    // draw curves
    //
    QwtPlotCurveIterator itc = curveIterator();
    if(itc.count() > 0) {
            QwtPlotCurve **curvelist = new QwtPlotCurve *[itc.count()];
            int cc = 0;
            for (QwtPlotCurve *curve = itc.toFirst(); curve != 0; curve = ++itc )
                    curvelist [cc++] = curve;
            for(cc--;cc>=0;cc--) {
                    QwtPlotCurve *curve = curvelist[cc];
                    if ( curve->enabled() &&
                         axisEnabled( curve->xAxis() ) &&
                         axisEnabled( curve->yAxis() ) )
                    {
                            curve->draw(p, map[curve->xAxis()], map[curve->yAxis()]);
                    }
            }
            delete[] curvelist;
    }

    //
    // draw markers
    //
    QwtPlotMarkerIterator itm = markerIterator();
    for (QwtPlotMarker *marker = itm.toFirst(); marker != 0; marker = ++itm )
    {
        if ( marker->enabled() )
        {
            marker->draw(p,
                map[marker->xAxis()].transform(marker->xValue()),
                map[marker->yAxis()].transform(marker->yValue()),
                rect);
        }
    }
}
```

nilmtypes.h

Definitions of the Exemplar and dataPoint types, header file.

```
#ifndef NILMTYPES_H
#define NILMTYPES_H

#include <qstring.h>
#include <qlistbox.h>
#include <qarray.h>

#ifndef min
#define min(a,b) (((a)<(b))?(a):(b))
#endif
#ifndef max
#define max(a,b) (((a)>(b))?(a):(b))
#endif

class QTextStream;

class dataPoint
{
public:
        double& operator[](int n) { return d[n]; }
private:
        double d[8];
};

class Exemplar
{
public:
        Exemplar(QString newdesc="Unnamed")
                : desc(newdesc) {}
        Exemplar(QArray<dataPoint> &newx, QString newdesc="Unnamed")
                : desc(newdesc), x(newx) {}

        QString desc;
        QArray<dataPoint> x;
        dataPoint norm; // of each component individually, before it was normalized

        bool write(QTextStream &out);
        bool read(QTextStream &in);
};

#endif
```

nilmtypes.cpp

Definitions of the Exemplar and dataPoint types, implementation.

```
#include "nilmtypes.h"
#include <qfile.h>

bool Exemplar::write(QTextStream &out)
{
        out << desc << "\n";
        out << x.size() << "\n";
        for(unsigned int i=0;i<x.size();i++) {
                for(int j=0;j<8;j++) {
                        out << x[i][j] << ((j==7)?"\n":"␣");
                }
        }
        for(int j=0;j<8;j++)
                out << norm[j] << ((j==7)?"\n":"␣");

        return true;
}

bool Exemplar::read(QTextStream &in)
{
        int size=0;
        desc = "";
        // Might get the tail of the previous line, so keep trying
        while(desc=="") {
                if((desc = in.readLine())==QString::null) {
                        printf("no␣description\n");
                        return false;
                }
        }
        size=0;
        in >> size;
        if(size==0) {
                printf("no␣size\n");
                return false;
        }
        x.resize(size);
        for(int i=0;i<size;i++)
                for(int j=0;j<8;j++)
                        in >> x[i][j];
        for(int j=0;j<8;j++)
                in >> norm[j];
        return true;
}
```

exemplardialog.h

The “Exemplars” dialog window, header file.

```
#ifndef EXEMPLARDIALOG_H
#define EXEMPLARDIALOG_H

#include "exemplardialogbase.h"
#include "nilmtypes.h"
#include <qlistbox.h>
#include <qarray.h>
#include <qptrvector.h>

class ExemplarItem : public QListBoxText
{
public:
        ExemplarItem(Exemplar *ne) : e(ne) { setText(e->desc); }
        Exemplar *e;
public:
        void rename(QString s) { e->desc=s; setText(s); }
};

class ExemplarDialog : public ExemplarDialogBase
{
        Q_OBJECT
public:
        ExemplarDialog(QWidget *parent=0, const char *name = 0, WFlags f = 0);

public slots:
        void doTrain();
        void doRename();
        void doDelete();
        void drawExemplar();
        void refreshExemplars();
        void setExemplars(QPtrVector<Exemplar>);
signals:
        void newExemplars(QPtrVector<Exemplar>);
        void newData(QArray<dataPoint>);

private:
        void makeExemplars(void);
        void refreshList ();

        QPtrVector<Exemplar> ex;
};

#endif
```

exemplardialog.cpp

The “Exemplars” dialog window, implementation.

```
#include "exemplardialog.h"
#include "nilmimp.h"
#include <qtextedit.h>
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <qstring.h>
#include "nilmtypes.h"
#include <qwt/qwt_plot.h>
#include <qmessagebox.h>
#include <qinputdialog.h>
#include "traindialog.h"

ExemplarDialog::ExemplarDialog(QWidget* parent, const char* name, WFlags f )
        : ExemplarDialogBase( parent, name, f )
{
        plot->setCanvasBackground(black);
        plot->enableOutline(false);
        plot->enableLegend(false);
        plot->setAxisMaxMajor(plot->yLeft,4);
        plot->setAxisMaxMajor(plot->xBottom,3);
        plot->setGridMajPen(QPen(gray,0,SolidLine));
}

void ExemplarDialog::doTrain(void)
{
        int i=0;
        QString suggest;
        QString prompt="Please enter a name for the new exemplar:";

        do suggest=QString().sprintf("Unnamed%d",++i);
        while(list->findItem(suggest,Qt::ExactMatch)!=0);

        bool ok;
retry:
        QString s = QInputDialog::getText("Train New Exemplar",prompt,
                                          QLineEdit::Normal, suggest, &ok, this);
        if(!ok) return;
        /* Qt::ExactMatch is not case-sensitive */
        if( list->findItem(s,Qt::ExactMatch)!=0) {
                prompt=QString().sprintf("An exemplar called '%s' already exists.\n"
                                         "Please enter a name for the new exemplar:",
                                         (const char *)s);
                suggest=s;
                goto retry;
        }

        printf("Starting training, name=%s\n",(const char *)s);

        Exemplar *n = NULL;
        TrainDialog *train = new
                TrainDialog(this,&n,&((NilmImp *)parent())->trainingparameters);
        QObject::connect(this,SIGNAL(newData(QArray<dataPoint>)),
                         train,SLOT(addData(QArray<dataPoint>)));
        train->exec();
        QObject::disconnect(this,SIGNAL(newData(QArray<dataPoint>)),
                            train,SLOT(addData(QArray<dataPoint>)));
        delete train;

        if(n==NULL) {
                printf("training␣cancelled␣or␣failed\n");
                return;
        } else {
                n->desc = s;
                ExemplarItem *ei = new ExemplarItem(n);
                list->insertItem(ei);
                list->sort();
                list->setCurrentItem(ei);
                drawExemplar();
                makeExemplars();
        }
}

void ExemplarDialog::doRename(void)
{
        ExemplarItem *ei = (ExemplarItem *)list->selectedItem();
        if(ei==NULL) {
                QMessageBox::warning(this,"Rename","No␣exemplar␣selected!",
                                     QMessageBox::Ok,QMessageBox::NoButton);
                return;
        }

        bool ok;
        QString s = QInputDialog::getText("Rename␣Exemplar",
                                          "Please␣enter␣a␣new␣name␣for␣'" + ei->text() + "':",
                                          QLineEdit::Normal, ei->text(), &ok, this);
        if(!ok) return;

        ei->rename(s);

        makeExemplars();
}

void ExemplarDialog::doDelete(void)
{
        ExemplarItem *ei = (ExemplarItem *)list->selectedItem();
        if(ei==NULL) {
                QMessageBox::warning(this,"Delete","No␣exemplar␣selected!",
                                     QMessageBox::Ok,QMessageBox::NoButton);
                return;
        }
        if(QMessageBox::warning(this,"Delete","Are␣you␣sure␣you␣wish␣to␣delete␣"
                                + ei->text(),
                                QMessageBox::Yes,QMessageBox::No)!=QMessageBox::Yes)
                return;

        /* Save a copy of the exemplar
           (shouldn't destroy it until we notify classifier) */
        Exemplar *saved = ei->e;
        delete ei;
        makeExemplars();

        delete saved;
}

void ExemplarDialog::drawExemplar(void)
{
        plot->removeCurves();
        plot->replot();

        ExemplarItem *ei = (ExemplarItem *)list->selectedItem();
        if(ei==NULL || ei->e->x.size()==0)
                return;

        long crv1, crv2;
        unsigned int len = ei->e->x.size();
        double *p, *q, *t;
        unsigned int i;

        if((p = new double[len])==NULL ||
           (q = new double[len])==NULL ||
           (t = new double[len])==NULL) return;

        crv2 = plot->insertCurve( "Q");
        plot->setCurvePen(crv2, QPen(cyan));
        plot->setCurveYAxis(crv2, QwtPlot::yLeft);

        crv1 = plot->insertCurve("P");
        plot->setCurvePen(crv1, QPen(yellow));
        plot->setCurveYAxis(crv1, QwtPlot::yLeft);

        double minx, maxx, miny, maxy;
        minx=maxx=miny=maxy=0;
        for(i=0;i<len;i++) {
                p[i]=ei->e->x[i][0];
                q[i]=ei->e->x[i][1];
                t[i]=i;
                miny = min(miny, p[i]);
                maxy = max(maxy, p[i]);
                miny = min(miny, q[i]);
                maxy = max(maxy, q[i]);
                minx = min(minx, t[i]);
                maxx = max(maxx, t[i]);
        }
        double yextra = (maxy-miny)*0.05;
        miny-=yextra;
        maxy+=yextra;
        double xextra = (maxx-minx)*0.05;
        minx-=xextra;
        maxx+=xextra;
        plot->setAxisScale(plot->yLeft,miny,maxy);
        plot->setAxisScale(plot->xBottom,minx,maxx);
        plot->setCurveData(crv1, t, p, len);
        plot->setCurveData(crv2, t, q, len);
        plot->replot();

        delete p;
        delete q;
        delete t;
}

void ExemplarDialog::makeExemplars(void)
{
        QPtrVector<Exemplar> ev(list->count());
        for(unsigned int i=0;i<list->count();i++)
                ev.insert(i,((ExemplarItem *)list->item(i))->e);
        emit newExemplars(ev);
}

void ExemplarDialog::refreshExemplars()
{
        makeExemplars();
}

void ExemplarDialog::setExemplars(QPtrVector<Exemplar> e)
{
        unsigned int i;
        list->clear();
        for(i=0;i<e.size(); i++) {
                ExemplarItem *ei = new ExemplarItem(e[i]);
                list->insertItem(ei);
        }
        list->sort();
        list->setCurrentItem(0);
        drawExemplar();

}
```

runningstats.h

Routines to compute and store running statistics on incoming data, header file.

```
#ifndef RUNNINGSTATS_H
#define RUNNINGSTATS_H

/* Keep running statistics over a window of points */
/* Also keep statistics on two smaller windows on left and right */
class RunningStats {
public:
        RunningStats(int windowsize = 100,
                     int leftsize = 10,
                     int rightsize = 10,
                     bool dovariance = true);
        ~RunningStats();
        void add(double v);

public:
        double lmean, lvariance;
        double rmean, rvariance;
        double mean, variance;
        bool addedenough;

        int size;

        // Stored in a circular buffer fashion, so
        // if the order matters, you MUST use
        // orderedvalue and orderedacvalue
        double *values;
        double *acvalues;
        double orderedvalue(int i) { return values[(i+offset)%size]; }
        double orderedacvalue(int i) { return acvalues[(i+offset)%size]; }

private:
        bool dovariance;
        int added;
        int lsize, rsize, offset;
};

#endif
```

runningstats.cpp

Routines to compute and store running statistics on incoming data, implementation.

```
#include "runningstats.h"
#include <math.h>
#include <stdio.h>

RunningStats::RunningStats(int windowsize, int leftsize, int rightsize, bool dv)
{
        size=windowsize;
        lsize=leftsize;
        rsize=rightsize;
        if(lsize>size) lsize=size;
        if(rsize>size) rsize=size;
        if(lsize<1) lsize=1;
        if(rsize<1) rsize=1;
        values = new double[size];
        acvalues = new double[size];
        for(int i=0;i<size;i++) values[i]=acvalues[i]=0;
        offset=0;
        added=0;
        mean=lmean=rmean=0;
        variance=lvariance=rvariance=0;
        dovariance=dv;
}

RunningStats::~RunningStats(void)
{
        delete[] values;
        delete[] acvalues;
}

void RunningStats::add(double v)
{
        /* Update means */
        /* 0 1 2 3 4 5 6 7 */
        /*      ^-- offset (new value replaces old here) */
        /* for mean, subtract 2 then add (2+size)%size */
        /* for lmean, had 23 needs 34 so subtract 2 add 4 */
        /* for rmean, had 01 needs 12 so subtract 0 add 2 */

        added++;
        if(added>size) addedenough=true;

        mean-= values[offset]/size;
        lmean-=values[offset]/lsize;
        rmean-=values[(size+offset-rsize)%size]/rsize;

        values[offset]=v;

        mean+= values[offset]/size;
        lmean+=values[(offset+lsize)%size]/lsize;
        rmean+=values[offset]/rsize;

        offset++;
        offset%=size;

        for(int i=0;i<size;i++)
                acvalues[i]=values[i]-mean;

        if(dovariance) {
                /* Fully recalculate variances (not optimal) */
                variance=0;
                for(int i=0;i<size;i++)
                        variance+=acvalues[i]*acvalues[i];
                variance/=size;

                lvariance=0;
                for(int i=0;i<lsize;i++)
                        lvariance+=(values[i]-lmean)*(values[i]-lmean);
                lvariance/=lsize;

                rvariance=0;
                for(int i=size-rsize;i<size;i++)
                        rvariance+=(values[i]-rmean)*(values[i]-rmean);
                rvariance/=rsize;
        }
}
```

trainingparameters.h

Header file specifying the tunable parameters used in exemplar training.

```
#ifndef TRAINING_PARAMETERS
#define TRAINING_PARAMETERS

class TrainingParameters
{
public:
        TrainingParameters() {
                // Lengths are in samples (100/sec in current version of prep)

                // length of long statistic window (for finding primary mean and variance)
                long_window_length = 200;

                // length of short statistic window (for detecting start/end of transient)
                short_window_length = 20;

                // how many samples before and after the transient event to save
                extra_transient_length = 50;

                // maximum variance for a settled signal == this times absolute mean
                settle_variance_max = 2;

                // Two ways that a transient is detected:
                // 1) take (mean(short window) - mean(long window))^2/variance(long window)
                //    and see if this has changed by more than max_delta_score
                //    (ie. the square of the z-score is higher than max_delta_score)
                // 2) check variance(short window)/variance(long window)
                //    and see if this is greater than max_variance_ratio
                //    (ie. the variance has changed by a factor of more
                //     than max_variance_ratio)
                max_delta_score = 5;
                max_variance_ratio = 50;

                // Maximum exemplar length (safety net)
                max_exemplar_length = 500; // 5 seconds
        }

        int long_window_length, short_window_length;
        int extra_transient_length ;
        double settle_variance_max;
        double max_delta_score, max_variance_ratio;
        unsigned int max_exemplar_length;
};

#endif
```

classifier.h

Exemplar matching and classification algorithm, header file.

```
#ifndef CLASSIFIER_H
#define CLASSIFIER_H

#include <qobject.h>
#include <qarray.h>
#include <qptrvector.h>
#include "nilmtypes.h"
#include "runningstats.h"

class Classifier : public QObject
{
        Q_OBJECT
public:
         Classifier (double *newtolerance);

public slots:
        void setExemplars(QPtrVector<Exemplar>);
        void addData(QArray<dataPoint>);

signals:
        void matchFound(QString name,
                        double quality, double scale);

public:
        QPtrVector<Exemplar> getExemplars(void);

private:
        void addPoint(dataPoint d);
        QPtrVector<Exemplar> ex;
        RunningStats **rsp;
        RunningStats **rsq;

        int match;
        double matchscore, matchratio, matchtime;

        double *tolerance;
};

#endif
```

classifier.cpp

Exemplar matching and classification algorithm, implementation.

```
#include "classifier.h"
#include "nilmtypes.h"
#include "nilmimp.h"
```

```
#include <math.h>

 Classifier :: Classifier (double *newtolerance)
{
        tolerance = newtolerance;
        printf(" classifier startup\n");
        ex = QPtrVector<Exemplar>();
        setExemplars(ex);
}

QPtrVector<Exemplar> Classifier::getExemplars(void)
{
        return ex;
}

void Classifier :: setExemplars(QPtrVector<Exemplar> e)
{
        unsigned int i;
        printf(" classifier got new exemplars (%d)\n",e.size());
        if(ex.size()>0) {
                for(i=0;i<ex.size(); i++) {
                        delete rsp[i];
                        delete rsq[i];
                }
                delete rsp;
                delete rsq;
        }
        ex = e;
        if(ex.size()>0) {
                rsp = new RunningStats *[ex.size()];
                rsq = new RunningStats *[ex.size()];
                for(i=0;i<ex.size(); i++) {
                        rsp[i] = new RunningStats(ex[i]->x.size(),1,1,false);
                        rsq[i] = new RunningStats(ex[i]->x.size(),1,1,false);
                }
        }
}

void Classifier :: addPoint(dataPoint d)
{
        unsigned int i;

        if(matchtime > 0) {
                matchtime--;
                if(matchtime == 0)
                        emit matchFound(ex[match]->desc, matchscore, matchratio);
        }

        for(i=0;i<ex.size(); i++) {
                rsp[i]->add(d[0]);
                rsq[i]->add(d[1]);
                if (!rsp[i]->addedenough || !rsq[i]->addedenough)
                        continue;
        // Normalize P and Q individually
        double normp=0, normq=0;
        for(int j=0;j<rsp[i]->size;j++) {
                normp += rsp[i]->acvalues[j] * rsp[i]->acvalues[j];
                normq += rsq[i]->acvalues[j] * rsq[i]->acvalues[j];
        }
        normp = sqrt(max(normp,1e-8)); // avoid divide by zero
        normq = sqrt(max(normq,1e-8));

        // Test the match by computing dot product
        double dp=0, dq=0;
        for(int j=0;j<rsp[i]->size;j++) {
                dp += ex[i]->x[j][0] * rsp[i]->orderedacvalue(j);
                dq += ex[i]->x[j][1] * rsq[i]->orderedacvalue(j);
        }
        dp /= normp;
        dq /= normq;

        double ratiop = normp/ex[i]->norm[0];
        double ratioq = normq/ex[i]->norm[1];
        if(ratiop>1) ratiop=1/ratiop;
        if(ratioq>1) ratioq=1/ratioq;

        double weightp = (normp+ex[i]->norm[0])/2;
        double weightq = (normq+ex[i]->norm[1])/2;
//      if((weightp+weightq)<250) continue;

        double weightedscore = (dp*weightp + dq*weightq)/(weightp+weightq);
        double weightedratio = (ratiop*weightp + ratioq*weightq)/(weightp+weightq);

        double tol = *tolerance;
        double lowtol = 1 - ((1 - tol) * 0.2 / 0.15);
        double hightol = 1 - ((1 - tol) * 0.1 / 0.15);
        double supertol = 1 - ((1 - tol) * 0.05 / 0.15);
        if((weightedscore > tol && weightedratio > tol) ||
           (weightedscore > hightol && weightedratio > lowtol) ||
           (weightedscore > lowtol && weightedratio > tol) ||
           (normp > 100 && dp > supertol && ratiop > hightol) ||
           (normq > 100 && dq > supertol && ratioq > hightol))
        {
                if(matchtime <= 0) {
                        /* Report this match after a little bit of time */
                        matchtime = rsp[i]->size;
                        match = i;
                        matchscore = weightedscore;
                        matchratio = weightedratio;
                }
                else {
                        /* Make this match the best one, if it's better
                           than previous. Better is defined as having
                           a higher weightedscore with weightedratio
                           at least 95% as good */
                        if((weightedscore > matchscore) &&
                           (weightedratio > (0.95 * matchratio))) {
```

```
                                        match = i;
                                        matchscore = weightedscore;
                                        matchratio = weightedratio;
                                }
                        }
                }
        }
}

void Classifier :: addData(QArray<dataPoint> d)
{
        for(unsigned int i=0;i<d.size();i++) addPoint(d[i]);
}
```

---

traindialog.h

Training logic and dialog window, header file.

```
#ifndef TRAINDIALOG_H
#define TRAINDIALOG_H

#include "traindialogbase.h"
#include "nilmtypes.h"
#include "runningstats.h"
#include <qarray.h>
#include "trainingparameters.h"

class TrainDialog : public TrainDialogBase
{
        Q_OBJECT
public:
        TrainDialog(QWidget *parent, Exemplar **ne, TrainingParameters *newtp);
        ~TrainDialog();

public slots:
        void addData(QArray<dataPoint>);
signals:

private:
        void addSinglePoint(dataPoint);

        Exemplar **dest;
        Exemplar *e;

        enum TrainStatus {
                None,  // Waiting for first data point
                Fill,  // Waiting to fill data buffers
                Pre,   // Waiting for quiet signal
                Trans, // Waiting for transient to start
                Post,  // Waiting for transient to end
                Done,  // Recording trailing edge of transient
                Temp    // Do nothing
        };

        TrainStatus state;

        TrainingParameters *tp;
        RunningStats *rs[2];      // p and q
};

#endif
```

---

traindialog.cpp

Training logic and dialog window, implementation.

```
#include "traindialog.h"
#include <qlabel.h>
#include <math.h>
#include "nilmtypes.h"

const int debugState = 0;

TrainDialog::TrainDialog(QWidget* parent, Exemplar **ne, TrainingParameters *newtp)
        : TrainDialogBase( parent, "Train", 0 )
{
        dest=ne;
        *dest=NULL;
        e = new Exemplar();
        status->setText("No_data_received.");
        state = None;
        tp = newtp;
        for(int i=0;i<2;i++) {
                rs[i]=new RunningStats(tp->long_window_length,
                                       tp->short_window_length,
                                       tp->short_window_length);
        }
}

TrainDialog::~TrainDialog()
{
        if(*dest==NULL)
                delete e;
        for(int i=0;i<2;i++) {
                delete rs[i];
        }
}

bool settled(double variance_max, RunningStats *rs)
{
        // See if the signal has settled by determining if the
        // variance/(abs(mean)+100) is less than variance_max.
        double m = rs->variance / (fabs(rs->mean)+100);
        if(debugState) printf(" settle : got variance %lf, maximum %lf\n", m, variance_max);
        return m < variance_max;
}

bool transient(double variance_ratio, double delta_score, RunningStats *rs, bool left)
{
        // left = true if we're looking for when it started (use left edge of statistics )
        // left = false if we're looking for when it started (use right edge of statistics )

        // Wait for transient to start:
        // 1) take (mean(short window) - mean(long window))^2/variance(long window)
        //    and see if this has changed by more than delta_score
        //    (ie. the square of the z-score is higher than delta_score)
        // 2) check variance(short window)/variance(long window)
        //    and see if this is greater than variance_ratio
        //    (ie. the variance has changed by a factor of more than variance_ratio)
        double longVariance = max(rs->variance,0.01); // don't divide by <0.01
        double tmp1 = (left?rs->lmean:rs->rmean) - rs->mean;
        tmp1 = tmp1 * tmp1 / longVariance;
        double tmp2 = (left?rs->lvariance:rs->rvariance) / longVariance;
        if(debugState) {
                printf("transient: got score %lf, wanted %lf\n", tmp1, delta_score);
                printf("           got ratio %lf, wanted %lf\n", tmp2, variance_ratio);
        }
        if(tmp1 > delta_score) {
                printf("transient detected on delta_score = %lf\n",tmp1);
        }
        if(tmp2 > variance_ratio) {
                printf("transient detected on variance_ratio = %lf\n",tmp2);
        }
        return (tmp1 > delta_score || tmp2 > variance_ratio);
}

void TrainDialog::addSinglePoint(dataPoint d)
{
        unsigned int i;
        static int fill_count ;
        static unsigned int exemplar_length;
        static int last_transient_exemplar_length ;

        for(i=0;i<2;i++)
                rs[i]->add(d[i]);

        if(debugState) {
                printf("lmean=%lf mean=%lf rmean=%lf\n",
                        rs[0]->lmean,rs[0]->mean,rs[0]->rmean);
                printf("lvariance=%lf variance=%lf rvariance=%lf\n",
                        rs[0]->lvariance,rs[0]->variance,rs[0]->rvariance);
                printf("lvariance=%lf variance=%lf rvariance=%lf\n",
                        rs[1]->lvariance,rs[1]->variance,rs[1]->rvariance);
        }

        if(state==Fill || state==Pre || state==Trans) {
```

```
        // Copy data into exemplar, pushing all other values back in the array
        dataPoint *dp = e->x.data();
        memmove(&dp[0],&dp[1],sizeof(dataPoint)*(tp->extra_transient_length-1));
        dp[tp->extra_transient_length-1]=d;
}

if(state==Post || state==Done) {
        // Append data to exemplar, growing array in chunks as necessary
        // (so we can't trust e->x.size())
        if((exemplar_length+1)>=e->x.size())
                e->x.resize(exemplar_length+100);
        e->x[exemplar_length] = d;

        // sanity check on length of exemplar
        if(exemplar_length < tp->max_exemplar_length) {
                exemplar_length++;
        } else {
                state = Done;
        }
}

switch(state)
{
case None:
        // Create exemplar long enough to hold pre_transient_length samples
        e->x.resize(tp->extra_transient_length);
        fill_count =0;
        exemplar_length=tp->extra_transient_length;
        last_transient_exemplar_length = 0;
        status->setText("Waiting_for_data_buffers_to_fill.");
        state = Fill;
        break;
case Fill:
        // Let buffers fill up
        fill_count ++;
        if(debugState) printf("Fill :,_got_%d\n",fill_count);
        if( fill_count >= tp->extra_transient_length &&
            fill_count >= tp->long_window_length &&
            fill_count >= tp->short_window_length) {
                status->setText("Waiting_for_signal_to_settle.");
                state = Pre;
        }
        break;
case Pre:
        if( settled (tp->settle_variance_max,rs[0]) &&
            settled (tp->settle_variance_max,rs[1]))
        {
                status->setText("Waiting_for_transient_to_begin.\n"
                                "Please_trigger_the_event_now.");
                state = Trans;
        }
        break;
case Trans:
        if( transient (tp->max_variance_ratio,tp->max_delta_score,rs[0],false) ||
            transient(tp->max_variance_ratio,tp->max_delta_score,rs[1],false))
        {
                status->setText("Recording transient and "
                                "waiting for signal to settle.");
                state = Post;
        }
        break;
case Post:
        // Find the last transient before settle
        if(transient(tp->max_variance_ratio,tp->max_delta_score,rs[0],true) ||
            transient(tp->max_variance_ratio,tp->max_delta_score,rs[1],true))
                last_transient_exemplar_length =
                        exemplar_length - tp->long_window_length;

        // Move onto next state only after transient and settle
        if(last_transient_exemplar_length > 0 &&
            settled(tp->settle_variance_max,rs[0]) &&
            settled(tp->settle_variance_max,rs[1]))
        {
                // Chances are we've already recorded enough data, but we'll see.
                status->setText("Recording trailing data.");
                state = Done;
        }
        break;
case Done:
{
        int extra = exemplar_length - last_transient_exemplar_length;
        // Make sure we have at least last_transient_exemplar_length + extra
        if(debugState) printf("Done: have %d extra\n",extra);
        if(extra > tp->extra_transient_length) {
                // All done. Clean up the exemplar and close.
                if(debugState)
                        printf("resizing to %d (last is %d)\n",
                                last_transient_exemplar_length +
                                tp->extra_transient_length,
                                last_transient_exemplar_length);
                e->x.resize(last_transient_exemplar_length +
                        tp->extra_transient_length);

                double total;
                int j;

                // Fix offset:
                // Calculate and subtract mean from all 8 components
                for(j=0;j<8;j++) {
                        total = 0;
                        for(i=0;i<e->x.size();i++)
                                total += e->x[i][j];
                        total /= i;
                        for(i=0;i<e->x.size();i++)
                                e->x[i][j] -= total;
                }

                /*
```

```
                    // Normalize apparent power: |sqrt(P^2+Q^2)| = 1
                    total = 0;
                    for(i=0;i<e->x.size();i++)
                            total += (e->x[i][0] * e->x[i][0])
                                    + (e->x[i][1] * e->x[i][1]);
                    total = sqrt(max(total, 1e-8)); // don't divide by zero
                    for(j=0;j<8;j++)
                            for(i=0;i<e->x.size();i++)
                                    e->x[i][j] /= total;
                    */

                    // Normalize each component individually for now.
                    for(j=0;j<8;j++) {
                            total=0;
                            for(i=0;i<e->x.size();i++)
                                    total+=e->x[i][j] * e->x[i][j];
                            total = sqrt(max(total, 1e-8));
                            e->norm[j] = total;
                            for(i=0;i<e->x.size();i++)
                                    e->x[i][j] /= total;
                    }

                    // Set the result and close this dialog
                    *dest = e;
                    done(Accepted);
                    state = Temp;
            }
        }
        case Temp:
                break;
        }
}

void TrainDialog::addData(QArray<dataPoint> d)
{
        for(unsigned int i=0;i<d.size();i++)
                addSinglePoint(d[i]);
}
```

A.4 NILM Software Framework

A.4.1 Database

nilmdb.sql

MySQL schema for the NILM database

```
-- $Id: nilmdb.sql 1569 2006-01-18 07:09:39Z jim $
-- NILM Database Schema
-- Jim Paris <jim@jtan.com>

--
-- Database parameters
--
CREATE TABLE `nilmdb` (
        `key`         CHAR(255) NOT NULL PRIMARY KEY,
        `value`       CHAR(255) NOT NULL
) DEFAULT CHARSET=utf8;
INSERT INTO `nilmdb` VALUES ('schema', '1');

--
-- Streams
--
-- Times are stored as doubles, so they can be pretty much anything.
CREATE TABLE `stream` (
        `id`          INT UNSIGNED PRIMARY KEY AUTO_INCREMENT,
        `tag`         VARCHAR(4095) NOT NULL,
        `type`        INT UNSIGNED NOT NULL default '0',
        `flags`       BIGINT UNSIGNED NOT NULL default '0',
        `start_time`  DOUBLE NOT NULL,
        `end_time`    DOUBLE NOT NULL
) DEFAULT CHARSET=utf8;

--
-- Key value pairs, to go with stream metadata
--
CREATE TABLE `metadata` (
        `id`          INT UNSIGNED PRIMARY KEY AUTO_INCREMENT,
        `stream_id`   INT UNSIGNED NOT NULL,
        `ascii`       BOOL NOT NULL,
        `key`         VARCHAR(4095) NOT NULL,
        `value`       BLOB
) DEFAULT CHARSET=utf8;

CREATE INDEX `metadata_idx1` ON `metadata` (`stream_id`, `key`(255));

--
-- Records, associated with each stream, very similar to the metadata
--
CREATE TABLE `record` (
        `id`          INT UNSIGNED PRIMARY KEY AUTO_INCREMENT,
        `stream_id`   INT UNSIGNED NOT NULL,
        `timestamp`   DOUBLE NOT NULL,
        `value`       BLOB
) DEFAULT CHARSET=utf8;
```

A.4.2 Library and Object Interface, `libstream`

`nilmdb.h`

Database access, matching, and manipulation object, header file.

```
#ifndef NILMDB_H
#define NILMDB_H

#include <mysql++.h>
#include <string>
#include "record.h"
#include "exception.h"

using namespace std;
using namespace mysqlpp;

class Stream;

class Nilmdb
{
 public:
        typedef unsigned int StreamId;

        Nilmdb(const char *db = "nilmdb",
               const char *host = "localhost",
               const char *user = "nilmdb",
               const char *pass = "");
        ~Nilmdb();

 public:
        /* Create a new stream in the database.  Updates the database
         * ID in the stream object. */
        void CreateStream(Stream &s);

        /* Append data records to a stream, using a callback to
         * provide data.  Callback returns false when there's no data
         * left. */
        int AppendData(StreamId id, Record::recordCallback source);

        /* Return a list of all streams in the database */
        vector <Stream> FindAllStreams(void) { return FindMatchingStreams("true"); }

        /* Delete the given stream object from the database, and
         * records associated with it.  It must be a simple stream
         * object (not sliced) */
        void DestroyStream(Stream &s);

        /*
         * Update s with a possibly-sliced stream that matches the
         * parameters in the provided stream
         */
        void MatchStream(Stream &s);
        /* Extract data records from the stream, using a callback to
           provide data.  Callback returns false if it wants no more. */
        int GetData(Stream &s, Record::recordCallback dest);

 private:
        /* Return a vector with matching streams from the database. */
        vector <Stream> FindMatchingStreams(string WhereClause);

        /* Retrieve data records matching the given clause */
        int GetMatchingData(string WhereClause,
                            Record::recordCallback dest);

        /* Fetch a single stream from the database */
        Stream FetchStream(StreamId id);

        Connection con;
};

#endif
```

nilmdb.cpp

Database access, matching, and manipulation object, implementation.

```
#include "nilmdb.h"
#include "stream.h"

#define NILMDB_SCHEMA "1"

#define PRECISION 20 /* for doubles to strings */

using namespace std;
using namespace mysqlpp;

Nilmdb::Nilmdb(const char *db, const char *host, const char *user,
               const char *pass) : con(db, host, user, pass)
{
        /* Verify database schema version */
        Query q = con.query();
        q << "SELECT value FROM nilmdb WHERE `key` = 'schema'";
        Result res = q.store();
        if (!res) throw BadQuery("nilmdb parameters not found");
        Row row = res.at(0);
        if (!row) throw BadQuery("nilmdb schema not found");
        if(string(row.at(0)) != NILMDB_SCHEMA)
                throw BadQuery(string("nilmdb schema mismatch, wanted \"") +
                               NILMDB_SCHEMA + string("\" but got \"") +
                               string(row.at(0)) + string("\""));
}
```

```
Nilmdb::~Nilmdb()
{
}

/*
 * Fetch simple stream info with the given id from the database,
 * including metadata.  Throws an exception if stream isn't found.
 * Doesn't fetch the data records.
 */
Stream Nilmdb::FetchStream(StreamId id)
{
        Stream s;
        Result res;
        Row row;

        Query q = con.query();

        /* Main stream info */
        q.precision(PRECISION);
        q << "SELECT `tag`, `type`, `flags`, `start_time`, `end_time` " <<
                "FROM stream WHERE id = " << id;
        res = q.store();
        if(res.num_rows() != 1) throw NilmException("Stream not found");
        row = res.at(0);

        s.tag = string(row["tag"]);
        s.type = row["type"];
        s.flags = row["flags"];
        s.start_time = row["start_time"];
        s.end_time = row["end_time"];

        /* Get the metadata */
        q << "SELECT `key`, `ascii`, `value` "
          << "FROM metadata WHERE stream_id = " << id
          << " ORDER BY `id`";
        res = q.store();

        for (Row::size_type i = 0; i < res.num_rows(); i++) {
                row = res.at(i);
                unsigned long *size = res.fetch_lengths();
                std::string val((cchar *)row.at(2), size[2]);
                s.metadata.push_back(Metadata(row.at(0), row.at(1), val));
        }

        s.database_id.clear();
        s.database_id.push_back(id);

        return s;
}

/*
 * Place the given stream info into the database, including the
 * metadata.  Returns true if successful.  If a stream with the same
 * tag/type exists with overlapping times, or there's any other error,
 * returns false.  Note this doesn't actually do anything with the
 * data records.  Throws mysqlpp::Exception on failure.
 */
void Nilmdb::CreateStream(Stream &s)
{
        Result res;
        Row row;
        Query q = con.query();
        my_ulonglong id;

        if(s.start_time > s.end_time)
                throw NilmException("Stream times aren't properly ordered");

        /* Make sure this tag/type doesn't exist with overlapping times.
           Adjacent times are accepted.  Find all other streams that
           start before this ends, and end after this starts,
           or have the exact same start time */
        q.precision(PRECISION);
        q << "SELECT `id` FROM stream WHERE " <<
                "`tag` = " << quote << s.tag <<
                " AND `type` = " << quote << s.type <<
                " AND ( (`start_time` < " << s.end_time <<
                " AND `end_time` > " << s.start_time << ")" <<
                " OR (`start_time` = " << s.start_time << ") )";
        res = q.store();
        if (res.num_rows() != 0) throw NilmException(
                "New stream overlaps an existing stream with same tag/type");

        /* Put it in */
        q.precision(PRECISION);
        q << "INSERT INTO `stream` (`tag`, `type`, `flags`, "
          << "`start_time`, `end_time`) VALUES (" << quote << s.tag
          << ", " << s.type << ", " << s.flags << ", "
          << s.start_time << ", " << s.end_time << ")";

        q.execute();

        id = con.insert_id();

        /* Add the metadata */
        q << "INSERT INTO `metadata` (`stream_id`, `ascii`, `key`, "
          << "`value`) VALUES (" << id << ", %0, %1q, %2q)";
        q.parse();
        for (size_t i = 0; i < s.metadata.size(); i++) {
                q.execute(s.metadata[i].ascii,
                          s.metadata[i].key,
                          s.metadata[i].value);
        }

        s.database_id.clear();
        s.database_id.push_back(id);

        return;
}
```

```
/* Return a vector with matching streams from the database. */
vector <Stream> Nilmdb::FindMatchingStreams(string WhereClause)
{
        Result res;
        Row row;
        Query q = con.query();
        vector <Stream> v;

        q << "SELECT `id` FROM stream WHERE (" << WhereClause +
                ") ORDER BY `id`";
        res = q.store();
        for (Row::size_type i = 0; i < res.num_rows(); i++) {
                row = res.at(i);
                v.push_back(FetchStream(row.at(0)));
        }

        return v;
}

/*
 * Match against streams in the database, possibly returning a sliced stream.
 * Fields:
 *   tag - must match
 *   type - must match
 *   flags - must match, except for StreamFlag::can_slice
 *   *_time - if >= 0, must match (if !can_slice)
 *             or be satisfiable through slicing
 *   metadata - specified fields must match
 * Returned stream has flags and metadata from the first
 * stream that makes up the returned dataset.
 */
void Nilmdb::MatchStream(Stream &s)
{
        ostringstream q;
        q.precision(PRECISION);
        s.database_id.clear();
        s.sliced = false;

        /* Match tag, type, flags, and times */
        q << "`tag` = " << quote << s.tag << " AND ";
        q << "`type` = " << quote << s.type << " AND ";
        q << "(`flags` & ~" << StreamFlag::can_slice << ") = "
          << quote << (s.flags & ~StreamFlag::can_slice);

        /* If times were provided, only get streams that contain
           data that falls inside the specified range */
        if (s.end_time >= 0)
                q << " AND `start_time` < " << s.end_time;
        if (s.start_time >= 0)
                q << " AND `end_time` >= " << s.start_time;

        /* Now fetch those streams that match */
        vector <Stream> potential = FindMatchingStreams(q.str());
```

```
/* Filter them based on the metadata */
vector <Stream> match;
for (size_t i = 0; i < potential.size(); i++) {
        bool matched = true;
        for (size_t j = 0; j < s.metadata.size(); j++) {
                if (!potential[i].matchMetadata(s.metadata[j])) {
                        matched = false;
                        break;
                }
        }
        if(matched)
                match.push_back(potential[i]);
}

/* Now match contains streams that match everything but time. */
if (match.empty()) throw NilmException(
        "No matches to specified tag, type, metadata, flags");

/* If we can't slice, we require one exact time match */
if (! (match[0].flags & StreamFlag::can_slice)) {
        if (match.size() != 1) throw NilmException(
                "No one-stream matches for non-sliced stream");
        if ((s.end_time >= 0 && match[0].end_time != s.end_time) ||
            (s.start_time >= 0 && match[0].start_time != s.start_time))
                throw NilmException("No time match for non-sliced stream");
        s = match[0];
        return;
}

/* We can slice; start by sorting the matches by their start time. */
sort(match.begin(), match.end(), Stream::compare_start_time);

/* If we didn't get a start/end time specified, get them from the list */
if (s.start_time < 0)
        s.start_time = match[0].start_time;
if (s.end_time < 0)
        s.end_time = match[match.size() - 1].end_time;

/* Now verify that the streams we have are contiguous, and can be sliced */
for (size_t i = 0; i < match.size() - 1; i++) {
        if(!match[i].flags & StreamFlag::can_slice)
                throw NilmException("Can't slice all required streams");
        if(match[i].end_time != match[i+1].start_time)
                throw NilmException(
                        "Can't slice streams; gap or overlap present");
}

/* Make sure the first satisfies our start time, and the last
   satisfies our end time */
if (match[0].start_time > s.start_time ||
    match[match.size() - 1].end_time < s.end_time) throw NilmException(
        "Can't slice streams; no streams extend that far in time");
        /* We should be good to go.  Fill some data into the stream */
        s.sliced = true;
        s.database_id.clear();
        s.flags = match[0].flags;
        s.metadata = match[0].metadata;
        for (size_t i = 0; i < match.size(); i++)
                s.database_id.push_back(match[i].getId());
}

/* Delete a stream and its records. */
void Nilmdb::DestroyStream(Stream &s)
{
        vector <Stream> v;

        /* Figure out which stream to delete based on tag/type and times */
        ostringstream str;
        str.precision(PRECISION);
        str << "`tag` = " << quote << s.tag << " AND "
            << "`type` = " << quote << s.type << " AND "
            << "`start_time` = " << s.start_time << " AND "
            << "`end_time` = " << s.end_time;
        v = Nilmdb::FindMatchingStreams(str.str());

        if(v.size() < 1) throw NilmException(
                "Didn't find matching stream to destroy");
        else if(v.size() > 1) throw NilmException(
                "Found too many matching streams, unexpected error");

        Query q = con.query();

        q << "DELETE FROM `stream` WHERE `id` = " << v[0].database_id[0];
        q.execute();

        q << "DELETE FROM `record` WHERE `stream_id` = " << v[0].database_id[0];
        q.execute();

        return;
}


/*
 * Add data records for the given stream id.  Checks to see if the
 * stream exists, but the record timestamps are not verified.
 * Could probably use some buffering.
 */
int Nilmdb::AppendData(StreamId id, Record::recordCallback source)
{
        ostringstream b;
        b.precision(PRECISION);
        int results = 0;

        /* Just to verify it exists; will throw an exception if not found */
        Stream s;
        s = FetchStream(id);
        if(s.getId() != id)
                throw NilmException("mismatched_IDs?");

        /* Add the records */
        Query q = con.query();
        q.precision(PRECISION);
        q << "INSERT_INTO_`record`_(`stream_id`,_`timestamp`,_`value`)_"
          << "VALUES_(" << id << ",_%0,_%1q)";
        q.parse();

        Record r;
        while((*source)(&r)) {
                results++;
                b.str("");
                b << r.timestamp;
                q.execute(b.str(), r.value);
        }

        return results;
}

/*
 * Retrieve data records for the given stream id that match the given
 * query string.  Could probably use some buffering.  Returns the total
 * number of results that were supplied.
 */
int Nilmdb::GetMatchingData(string WhereClause,
                            Record::recordCallback dest)
{
        ResUse res;
        Row row;
        Record r;
        Query q = con.query();
        int results = 0;

        q << "SELECT_`timestamp`,_`value`_FROM_record_WHERE_("
          << WhereClause << ")_ORDER_BY_`timestamp`";
        res = q.use();
        if (!res) /* No results */
                return 0;

        try {
                while (row = res.fetch_row()) {
                        unsigned long *size = res.fetch_lengths();
                        r.timestamp = row.at(0);
                        r.value = std::string(row.raw_data(1), size[1]);
                        results++;
                        if (!(*dest)(&r)) break;
                }
        } catch (const mysqlpp::EndOfResults&) {
                return results;
        }
        return results;
}
```

```
/* Extract data records from the stream, using a callback to
   provide data.  Callback returns false if it wants no more. */
int Nilmdb::GetData(Stream &s, Record::recordCallback dest)
{
        if (s.database_id.empty()) throw NilmException(
                "Given stream isn't associated with any database records");

        /* Build a query that gives us all records that fall within
           the time range and have the provided stream IDs. */
        ostringstream q;
        q.precision(PRECISION);
        q << "`timestamp` >= " << s.start_time << " AND "
          << "`timestamp` <= " << s.end_time << " AND (FALSE";

        for (size_t i = 0; i < s.database_id.size(); i++) {
                q << " OR `stream_id` = " << s.database_id[i];
        }
        q << ")";

        return GetMatchingData(q.str(), dest);
}
```

stream.h

Definition of the Stream object and functions, header file.

```
#ifndef STREAM_H
#define STREAM_H

/* Definition of a NILM stream. This contains everything
   except the actual stream data, which stays in the database.
*/

#include "metadata.h"
#include "streamflag.h"
#include "streamtype.h"
#include "nilmdb.h"

#include <string>

using namespace std;

class Stream {
 public:
        Stream();
        ~Stream();

 public:
        string tag;
        StreamType::type type;
        StreamFlag::type flags;
        double start_time;
        double end_time;

        vector <Metadata> metadata;

 public:
        /* Return human-readable representation */
        string asText(void);

        /* Return true if m matches one of our metadata items.
         * Empty value means return true if the key is not present. */
        bool matchMetadata(Metadata m);

        /* Get database id and sliced status */
        Nilmdb::StreamId getId(void) {
                return database_id.size() ? database_id[0] : 0;
        }
        bool isSliced(void) {
                return sliced;
        }


 private:
        friend class Nilmdb;
        /* ID of this stream in the database, if this came from there.
         * A time-ordered list of the stream IDs for this data, if
         * this is sliced. */
        vector <Nilmdb::StreamId> database_id;
        bool sliced;

        /* Predicates for sorting */
        static bool compare_start_time(const Stream &a, const Stream &b) {
                return a.start_time < b.start_time;
        }
        static bool compare_end_time(const Stream &a, const Stream &b) {
                return a.end_time < b.end_time;
        }


};

#endif
```

stream.cpp

Definition of the Stream object and functions, implementation.

```
#include "stream.h"
#include <stdio.h>
#include <time.h>
```

```
using namespace std;

Stream::Stream(void)
{
        type = 0;
        flags = StreamFlag::time_real;
        start_time = 0;
        end_time = 0;
        database_id.clear ();
        sliced = false;
}

Stream::~Stream()
{
}

string Stream::asText()
{
        string s;
        char b[256];
        time_t t;
        bool showtime =
                (flags & StreamFlag::time_real) &&
                (flags & StreamFlag::time_absolute);

        s  = "␣␣␣␣Tag:␣" + tag + "\n";
        sprintf(b,"(%ld)␣", (long int) type);

        s += "␣␣␣␣Type:␣" + string(b);
        s += string(StreamType::toString(type)) + "\n";
        sprintf(b,"(%08lx)␣", (unsigned long int) flags);

        s += "␣␣␣Flags:";
        if (flags & StreamFlag::can_slice)        s += "␣can_slice";
        if (flags & StreamFlag::time_real)        s += "␣time_real";
        if (flags & StreamFlag::time_absolute) s += "␣time_absolute";
        if (flags & StreamFlag::ascii_data)       s += "␣ascii_data";
        if (!flags) s += "␣(none)";
        s += "\n";

        s += "␣␣␣Start:␣";
        if (showtime && start_time >= 0)
        {
                char b[256];
                t = (time_t) start_time;
                strftime(b, 256, "(%c)␣", gmtime(&t));
                s += b;
        }
        sprintf(b,"%lf\n", start_time);
        s += b;

        s += "␣␣␣␣␣End:␣";
        if (showtime && end_time >= 0)
```

```
    {
        char b[256];
        t = (time_t) end_time;
        strftime(b, 256, "(%c)␣", gmtime(&t));
        s += b;
    }
    sprintf(b,"%lf\n", end_time);
    s += b;

    s += "Metadata:␣";
    sprintf(b, "%d␣items\n", (int)metadata.size()); s += b;
    for(unsigned int i=0; i<metadata.size(); i++) {
        s += "␣␣\"";
        s += metadata[i].key + "\"␣->␣";
        if(metadata[i]. ascii ) {
            s += '"' + metadata[i].value + '"';
        } else {
            for(unsigned int j=0; j<metadata[i].value.size(); j++) {
                sprintf(b, "0x%02x␣", metadata[i].value[j]);
                s += b;
            }
        }
        s += "\n";
    }

    if ( sliced ) s += "␣␣Sliced:␣yes\n";

    return s;
}

bool Stream::matchMetadata(Metadata m)
{
    size_t i;
    bool test_not_present = m.value.empty() ? true : false;

    for(i=0; i<metadata.size(); i++) {
        if(metadata[i].key == m.key) {
            /* Found the key, but wanted it to be missing */
            if( test_not_present )
                return false;

            /* Found the key, and it matched our value */
            if(metadata[i].value == m.value)
                return true;
        }
    }
    /* Key was not present */
    if( test_not_present )
        return true;
    else
        return false;
}
```

streamtype.h

Definition of the standard stream types and helpers, header file.

```
#ifndef STREAMTYPE_H
#define STREAMTYPE_H

/* Definition of some standard NILM stream types.
   These are just for convenience, and applications are free to use
   whatever integers they'd like */

#include <string>

namespace StreamType {
        typedef unsigned int type;

        /* Standard stream types */
        const char *const ascii[] = {
                /* 0 */ "Unspecified",
                /* 1 */ "Raw",
                /* 2 */ "Envelope",
                /* 3 */ "FFT",
                0
        };

        type fromString(const char *s);
        const char *toString(type num);

        std::string describe(void);
}

#endif
```

streamtype.cpp

Definition of the standard stream types and helpers, implementation.

```
#include "streamtype.h"
#include <string.h>
#include <stdlib.h>

namespace StreamType {
        /* Return the stream type number from the (case insensitive) string,
           or (StreamType::type)-1 if it's not found. */
        type fromString(const char *s) {
                type i;
                char *endptr;
                for (i=0; ascii[i]; i++) {
                        if(strcasecmp(s, ascii[i]) == 0)
                                return i;
                }
        /* Didn't match, maybe it's a number */
        i = (type)strtoul(s, &endptr, 0);
        if(*endptr == '\0')
            return i;
        else
            return (type)-1;
    }

    /* Return the stream string from the number, or a (static) text
       representation of the number if it's not found by name. */
    const char *toString(type num) {
        type i;
        static char s[32];
        for (i=0; i<num && ascii[i]; i++)
            continue;
        if(ascii[i] != NULL)
            return ascii[i];
        sprintf(s,"%ld",(long int)num);
        return s;
    }

    /* Return list of valid types */
    std::string describe(void) {
        std::string s;
        for (int i=0; ascii[i]; i++) {
            if (i) s += ", ";
            s += ascii[i];
        }
        return s + ", [any number]";
    }
}
```

streamflag.h

Definition of the available stream flags and helpers, header file.

```
#ifndef STREAMFLAG_H
#define STREAMFLAG_H

#include <string>

/* NILM stream flags. */
namespace StreamFlag {
    typedef unsigned long type;

    /* All flags, entry n is text for flag 1 << n */
    const char *const ascii[] = {
        /* 0 */ "CanSlice",
        /* 1 */ "TimeIsSeconds",
        /* 2 */ "TimeIsAbsolute",
        /* 3 */ "AsciiData",
                NULL
        };

        const int none = 0;

        /* Can perform slicing */
        const int can_slice = 1 << 0;

        /* Timestamp in records is a real time in seconds */
        const int time_real = 1 << 1;

        /* Timestamp in records is absolute, versus relative */
        const int time_absolute = 1 << 2;

        /* Record blobs are ascii, versus binary */
        const int ascii_data = 1 << 3;

        type fromString(const char *s);
        std::string toString(type flags);

        std::string describe(void);
}

#endif
```

streamflag.cpp

Definition of the available stream flags and helpers, implementation.

```
#include "streamflag.h"

namespace StreamFlag {
        /* Return the stream flags number from the (case insensitive)
           string, or 0 if it's not found. */
        type fromString(const char *s) {
                type i;
                for (i=0; ascii[i]; i++) {
                        if(strcasecmp(s, ascii[i]) == 0)
                                return 1 << i;
                }
                return 0;
        }

        /* Return the stream string from the number, or NULL if undefined. */
        std::string toString(type num) {
                std::string s;
                for (int i=0; ascii[i]; i++) {
                        if(num & (1 << i)) {
                                if(s != "") s += " ";
                                s += ascii[i];
                        }
                }
                if(s == "") s += "None";
                return s;
        }

        /* Return list of valid flags */
        std::string describe(void) {
                std::string s;
                for (int i=0; ascii[i]; i++) {
                        if (i) s += "␣";
                        s += ascii[i];
                }
                return s;
        }
}
```

metadata.h

Definition of a stream metadata object, header file.

```
#ifndef METADATA_H
#define METADATA_H

/* A key/value metadata pair */

#include <string>

class Metadata
{
 public:
        Metadata(const char *newkey, bool isascii, std::string &valuedata);
        Metadata(const char *keyval);

 public:
        std::string key;
        bool ascii;
        std::string value;
};

#endif
```

metadata.cpp

Definition of a stream metadata object, implementation.

```
#include "metadata.h"
#include "mysql++.h"
#include <string.h>
```

```
using namespace std;

Metadata::Metadata(const char *newkey, bool isascii, std::string &valuedata)
{
        key = newkey;
        ascii = isascii;
        value = valuedata;
}

Metadata::Metadata(const char *keyval)
{
        char *eq = strchr(keyval, '=');
        if (!eq) {
                key = "";
                return;
        }
        key = std::string(keyval, eq - keyval);
        ascii = 1;
        value = std::string(eq + 1);
}
```

record.h

Definition of the a stream record object, header file.

```
#ifndef RECORD_H
#define RECORD_H

/* A timestamp/value record pair */

#include <string>

class Record
{
 public:
        typedef bool (*recordCallback)(Record *r);

        Record();
        Record(double ntimestamp, std::string &nvalue);

        double timestamp;
        std::string value;
};

#endif
```

record.cpp

Definition of the a stream record object, implementation.

```
#include "record.h"
#include <string.h>

Record::Record(double ntimestamp, std::string &nvalue)
{
        timestamp = ntimestamp;
        value = nvalue;
}

Record::Record()
{
}
```

util.h

Date and time parsing utility functions, header file.

```
#ifndef NILMUTIL_H
#define NILMUTIL_H

namespace NilmUtil
{
        const char *const datespec =
                "Available␣date␣string␣formats:\n"
                "␣␣1)␣Real␣number␣(double):\n"
                "␣␣␣␣␣965086161.14\n"
                "␣␣2)␣Absolute␣date␣in␣the␣form␣YYMMDD−HHMM[SS]:\n"
                "␣␣␣␣␣20051205−130505\n"
                "␣␣3)␣Textual␣format:\n"
                "␣␣␣␣␣\"Mon␣Dec␣05␣13:05:05␣2005\"\n"
                "␣␣If␣using␣absolute␣seconds,␣#1␣is␣seconds␣since␣epoch,"
                "␣and␣all␣times␣are␣UTC.\n"
                "␣␣If␣using␣relative␣time␣or␣arbitrary␣units,␣only␣#1␣is␣valid.\n";
        double string_to_date(const char *s);
        void chomp(char *s);
}

#endif
```

util.cpp

Date and time parsing utility functions, implementation.

```
#include "util.h"
#include <time.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
```

```
#include <ctype.h>

namespace NilmUtil
{
        double string_to_date(const char *s)
        {
                double d;
                char *endptr;
                int len;
                struct tm t;
                int yy,mm,dd,hh,nn,ss;
                char tmp[256];

                /* If it's a double, use it like that */
                d = strtod(s, &endptr);
                if(*endptr == '\0')
                        return d;

                /* Use UTC everywhere. */
                setenv("TZ", "", 1);
                tzset ();

                /* check for the snapshot format YYYYMMDD-HHMM[SS] */
                len = sscanf(s, "%04d%02d%02d-%02d%02d%02d%c",
                             &yy, &mm, &dd, &hh, &nn, &ss, &tmp[0]);
                if(len == 5) { ss = 0; len++; }
                if(len == 6) {
                        /* Manipulating "struct tm" fields directly
                           is a no-no, so go through a string */
                        sprintf(tmp, "%04d %02d %02d %02d %02d %02d",
                                yy, mm, dd, hh, nn, ss);
                        endptr = strptime(tmp, "%Y %m %d %H %M %S", &t);
                        if(endptr && (*endptr == '\0'))
                                return mktime(&t);
                        else {
                                fprintf(stderr," internal date error\n");
                                return -1;
                        }
                }



                t.tm_year -= 1900;
                t.tm_mon -= 1;
                if(len == 5) { t.tm_sec = 0; len++; }
                if(len == 6)
                        return (unsigned int)mktime(&t);

                /* Like "date" except no timezone */
                endptr = strptime(s, "%a %b %d %H:%M:%S %Y", &t);
                if(endptr && (*endptr == '\0'))
                        return mktime(&t);

                return -1;
        }

        void chomp(char *s) {
                int len = strlen(s) - 1;
                while(len > 0 && isspace(s[len]))
                      s[len--] = 0;
        }
}
```

dbopt.h

Database option parsing routines, header file.

```
#include <string>
#include "opt.h"

using namespace std;

namespace dbopt {
        const char *const default_db = "nilmdb";
        const char *const default_host = "localhost";
        const char *const default_user = "nilmdb";
        const char *const default_pass = "";

        const struct options db = { 'D', "db", "name", "NILM database name" };
        const struct options host = { 'H', "host", "host", "NILM database host" };
        const struct options user = { 'U', "user", "user", "NILM database user" };
        const struct options pass = { 'P', "pass", "pass", "NILM database pass" };

        bool parse(char c, string &db, string &host, string &user, string &pass,
                   const char *optarg);
}
```

dbopt.cpp

Database option parsing routines, implementation.

```
#include "dbopt.h"
#include <string>

using namespace std;

namespace dbopt
{
        bool parse(char c, string &db, string &host, string &user, string &pass,
                   const char *optarg)
        {
                switch(c) {
                case 0:
                    db = default_db;
                    host = default_host;
                    user = default_user;
                    pass = default_pass;
                    return true;
                case 'D':
                    db = optarg;
                    return true;
                case 'H':
                    host = optarg;
                    return true;
                case 'U':
                    user = optarg;
                    return true;
                case 'P':
                    pass = optarg;
                    return true;
                default:
                    return false;

                }
        }
}
```

A.4.3 Client Programs

insert.cpp

Implementation of insert, which creates and stores streams in the database.

```
#include <mysql++.h>
#include "stream.h"
#include "nilmdb.h"
#include "opt.h"
#include "dbopt.h"
#include "streamtype.h"
#include "streamflag.h"
#include "util.h"
#include <time.h>
#include <sys/time.h>
#include <signal.h>

using namespace std;

const char svnid[] = "$Id: insert.cpp 1597 2006-01-22 01:00:08Z jim $";
const char svnrev[] = "$Rev: 1597 $";

struct options opt[] = {
        dbopt::db, dbopt::host, dbopt::user, dbopt::pass, opt_blank, /* D,H,U,P */
        { 't', "tag", "string", "tag to use for new stream" },
        { 's', "start", "datespec", "starting␣time␣of␣the␣data" },
        { 'e', "end", "datespec", "ending␣time␣of␣the␣data" },
        { 'T', "type", "typespec", "type␣of␣data␣in␣the␣stream" },
        { 'm', "metadata", "key=val", "extra␣metadata␣to␣add␣to␣the␣stream" },
        opt_blank,
        { 'S', "step", "time", "timestep␣between␣each␣line␣of␣input" },
        { 'L', "lines", "count", "total␣lines␣of␣input␣that␣will␣be␣supplied" },
        { 'A', "annotate", NULL, "incoming␣data␣is␣annotated␣with␣timestamps" },
        { 'R', "realtime", NULL, "data␣is␣supplied␣on␣stdin␣in␣realtime" },
        opt_blank,
        { 'r', "relative", NULL, "times␣are␣relative,␣not␣absolute" },
        { 'a', "arbitrary", NULL, "times␣are␣arbitrary,␣not␣seconds" },
        { 'n', "noslice", NULL, "cannot␣slice␣this␣stream" },
        opt_blank,
        { 'd', "delete", NULL, "delete␣the␣stream␣if␣an␣error␣or␣signal␣occurs" },
        { 'V', "version", NULL, "show␣program␣version␣and␣exit" },
        { 'h', "help", NULL, "this␣help" },
        opt_end
};

static bool got_signal = false;
void handle_sig(int sig) { got_signal = true; }
static bool annotate = false;
static bool realtime = false;
static bool deletestream = false;

static double global_time;
static long double global_timestep;
bool supply_data(Record *r)
{
        struct timeval t;
        static char line[1024];

        /* Give one record's worth of data, or return false if we're done */
        do {
                if(got_signal) return false;
                fgets(line, 1024, stdin);
                if(got_signal || feof(stdin)) return false;
        } while(line[0] == '#');        /* Ignore comments */

        NilmUtil::chomp(line);

        if(annotate) {
                /* If annotated, extract the time. Expected format:
                   <double timestamp><single space><data>\n */
                char *end;
                r->timestamp = strtod(line, &end);
                if(*end != '␣') {
                        fprintf(stderr,"***␣Insert:␣Bad␣annotation␣format\n");
                        got_signal = true; /* so we delete it */
                        return false;
                }
                r->value = end+1;
        } else if(realtime) {
```

```
        /* Use the current system time */
        gettimeofday(&t, NULL);
        r->timestamp = (double)t.tv_sec + (double)t.tv_usec / 1e6;
        r->value = line;
    } else {
        /* Compute from the specified timestemp */
        r->timestamp = global_time;
        global_time += global_timestep;
        r->value = line;
    }

    return true;
}

void create_stream(Nilmdb &db, Stream &s)
{
    Nilmdb::StreamId id;

    /* Setup signal handlers to cleanly abort this if we get interrupted */
    struct sigaction sa;
    sa.sa_handler = handle_sig;
    sigemptyset(&sa.sa_mask);
    sigaction(SIGHUP, &sa, NULL);
    sigaction(SIGINT, &sa, NULL);
    sigaction(SIGQUIT, &sa, NULL);
    sigaction(SIGABRT, &sa, NULL);
    sigaction(SIGTERM, &sa, NULL);

    /* Do it */
    fprintf(stderr,"Creating stream:\n%s", s.asText().c_str());
    db.CreateStream(s);
    id = s.getId();

    fprintf(stderr,"New stream ID: %d\n", (int)id);
    fprintf(stderr,"Reading data from stdin\n");

    global_time = s.start_time;
    int count = db.AppendData(id, supply_data);

    if(!got_signal && count) {
        /* Success */
        fprintf(stderr,"Inserted %d records\n", count);
        return;
    }

    if(got_signal) fprintf(stderr,"Interrupted\n");
    else if(!count) fprintf(stderr,"No data received\n");

    if(deletestream) {
        fprintf(stderr,"Deleting stream\n");
        db.DestroyStream(s);
    }
}
```

```
int main(int argc, char *argv[])
{
        int optind;
        char *optarg;
        FILE *help = stderr;
        char c;
        string db, host, user, pass;
        double timestep = 0;
        long lines = 0;
        char *endptr;

        /* Initialize stream */
        Stream s;
        s.start_time = -1;
        s.end_time = -1;
        s.type = (StreamType::type)-1;
        s.flags = StreamFlag::can_slice | StreamFlag::time_real
                | StreamFlag::ascii_data | StreamFlag::time_absolute;

        /* Parse and validate options */
        opt_init(&optind);
        dbopt::parse(0, db, host, user, pass, 0);
        while((c=opt_parse(argc,argv,&optind,&optarg,opt))!=0) {
                if (dbopt::parse(c, db, host, user, pass, optarg))
                    continue;
                switch(c) {
                case 't': // tag
                        s.tag = optarg;
                        break;
                case 's': // start
                        s.start_time = NilmUtil::string_to_date(optarg);
                        if(s.start_time < 0) {
                                fprintf(stderr, "invalid date: \"%s\"\n", optarg);
                                goto printhelp;
                        }
                        break;
                case 'e': // end
                        s.end_time = NilmUtil::string_to_date(optarg);
                        if(s.end_time < 0) {
                                fprintf(stderr, "invalid date: \"%s\"\n", optarg);
                                goto printhelp;
                        }
                        break;
                case 'm': // metadata
                {
                        Metadata m(optarg);
                        if(m.key == "") {
                                fprintf(stderr,"invalid metadata: \"%s\"\n", optarg);
                                goto printhelp;
                        }
                        s.metadata.push_back(m);
                        break;
                }
                case 'T': // type
```

```
{
          s.type = StreamType::fromString(optarg);
          if(s.type == (StreamType::type)-1) {
                    fprintf(stderr,"invalid␣type:␣\"%s\"\n", optarg);
                    goto printhelp;
          }
          break;
}

case 'S': // timestep
          timestep = strtod(optarg, &endptr);
          if(*endptr != '\0' || timestep <= 0) {
                    fprintf(stderr,"invalid␣timestep:␣\"%s\"\n", optarg);
                    goto printhelp;
          }
          break;
case 'L': // lines
          lines = strtoul(optarg, &endptr, 0);
          if(*endptr != '\0' || lines == 0) {
                    fprintf(stderr,"invalid␣line␣count:␣\"%s\"\n", optarg);
                    goto printhelp;
          }
          break;

case 'r': // relative
          s.flags &= ~StreamFlag::time_absolute;
          break;
case 'a': // arbitrary
          s.flags &= ~StreamFlag::time_real;
          break;
case 'n': // noslice
          s.flags &= ~StreamFlag::can_slice;
          break;

case 'A': // annotate
          annotate = true;
          break;

case 'R': // realtime
          realtime = true;

case 'd': // delete
          deletestream = true;
          break;

case 'V':
          printf("insert␣%s\nId:\n␣%s\n", svnrev, svnid);
          printf("Written␣by␣Jim␣Paris␣<jim@jtan.com>.\n");
          return 0;
          break;
case 'h':
          help=stdout;
printhelp:
          fprintf(help,"\n");
```

```
        default:
                fprintf(help,"Inserts␣records␣into␣the␣NILM␣database.\n");
                fprintf(help,"Data␣is␣supplied␣as␣ASCII␣text␣on␣stdin.\n\n");
                fprintf(help,"Usage:␣%s␣[options]\n\n",*argv);
                opt_help(opt,help);
                fprintf(help,"\n%s", NilmUtil::datespec);
                fprintf(help,"\nValid␣stream␣types:\n␣␣%s\n",
                        StreamType::describe().c_str());
                return 1;
        }
    }

    if(s.tag == "" || s.type == (StreamType::type)-1) {
            fprintf(help, "***␣Insert:␣you␣must␣specify␣a␣stream␣tag␣and␣type\n");
            goto printhelp;
    }

    if(s.start_time < 0 || s.end_time < 0) {
            fprintf(help, "***␣Insert:␣must␣specify␣stream␣starting/ending␣times\n");
            goto printhelp;
    }

    if(s.end_time < s.start_time) {
            fprintf(help, "***␣Insert:␣start␣time␣(%lf)␣must␣be␣before␣"
                    "end␣time␣(%lf)\n", s.start_time, s.end_time);
            goto printhelp;
    }

    if( ((timestep != 0) + (lines != 0) + (annotate == true) +
         (realtime == true)) != 1) {
            fprintf(help,"***␣Insert:␣must␣specify␣exactly␣one␣of␣"
                    "--step,␣--lines,␣--annotate,␣--realtime\n");
            goto printhelp;
    }

    /* Set up timestep between successive input lines */
    if(timestep == 0)
            global_timestep = (s.end_time - s.start_time) / lines;  // our best guess
    else
            global_timestep = timestep;

    /* Now go create the stream */
    try {
            Nilmdb nilm(db.c_str(), host.c_str(), user.c_str(), pass.c_str());
            create_stream(nilm, s);
    } catch (const mysqlpp::Exception &e) {
            fprintf(stderr,"***␣Insert:␣%s\n", e.what());
            return -1;
    }
    return 0;
}
```

extract.cpp

Implementation of extract, which matches and retrieves streams from the database.

```
#include <mysql++.h>
#include "stream.h"
#include "nilmdb.h"
#include "opt.h"
#include "dbopt.h"
#include "streamtype.h"
#include "streamflag.h"
#include "util.h"
#include <time.h>
#include <signal.h>

using namespace std;

const char svnid[] = "$Id: extract.cpp 1585 2006-01-20 10:45:24Z jim $";
const char svnrev[] = "$Rev: 1585 $";

struct options opt[] = {
        dbopt::db, dbopt::host, dbopt::user, dbopt::pass, opt_blank, /* D,H,U,P */
        { 't', "tag", "string", "tag to extract" },
        { 's', "start", "datespec", "starting time of the data (optional)" },
        { 'e', "end", "datespec", "ending time of the data (optional)" },
        { 'T', "type", "typespec", "type of data in the stream (mand)" },
        { 'm', "metadata", "key=val", "extra metadata to match (optional)" },
        opt_blank,
        { 'r', "relative", NULL, "times are relative, not absolute" },
        { 'a', "arbitrary", NULL, "times are arbitrary, not seconds" },
        { 'A', "annotate", NULL, "annotate output data with timestamps" },
        opt_blank,
        { 'q', "quiet", NULL, "just show matched stream info, no data" },
        { 'V', "version", NULL, "show program version and exit" },
        { 'h', "help", NULL, "this help" },
        opt_end
};

static bool got_signal = false;
void handle_sig(int sig) { got_signal = true; }
static bool annotate = false;
static bool quiet = false;

bool receive_data(Record *r)
{
        if(annotate) printf("%lf ", r->timestamp);
        printf("%s\n", r->value.c_str());
        return got_signal ? false : true;
}

void extract_stream(Nilmdb &db, Stream &s)
{
        /* Setup signal handlers to cleanly abort this if we get interrupted */
        struct sigaction sa;
        sa.sa_handler = handle_sig;
        sigemptyset(&sa.sa_mask);
        sigaction(SIGHUP, &sa, NULL);
        sigaction(SIGINT, &sa, NULL);
        sigaction(SIGQUIT, &sa, NULL);
        sigaction(SIGABRT, &sa, NULL);
        sigaction(SIGTERM, &sa, NULL);

        /* Do it */
        fprintf(stderr,"Trying to match stream:\n%s", s.asText().c_str());
        db.MatchStream(s);

        fprintf(stderr,"\nMatched stream:\n%s", s.asText().c_str());

        if(quiet) return;

        fprintf(stderr,"Writing data to stdout\n");
        int count = db.GetData(s, receive_data);

        fprintf(stderr, got_signal ? "Interrupted\n" : "Done\n");
        fprintf(stderr,"Extracted %d records\n", count);
}

int main(int argc, char *argv[])
{
        int optind;
        char *optarg;
        FILE *help = stderr;
        char c;
        string db, host, user, pass;

        /* Initialize stream */
        Stream s;
        s.start_time = -1;
        s.end_time = -1;
        s.type = (StreamType::type)-1;
        s.flags = StreamFlag::time_real
                | StreamFlag::ascii_data
                | StreamFlag::time_absolute;

        /* Parse and validate options */
        opt_init(&optind);
        dbopt::parse(0, db, host, user, pass, 0);
        while((c=opt_parse(argc,argv,&optind,&optarg,opt))!=0) {
                if (dbopt::parse(c, db, host, user, pass, optarg))
                    continue;
                switch(c) {
                case 't': // tag
                        s.tag = optarg;
                        break;
                case 's': // start
                        s.start_time = NilmUtil::string_to_date(optarg);
                        if(s.start_time < 0) {
```

```
                fprintf(stderr, "invalid␣date:␣\"%s\"\n", optarg);
                goto printhelp;
        }
        break;
case 'e': // end
        s.end_time = NilmUtil::string_to_date(optarg);
        if(s.end_time < 0) {
                fprintf(stderr, "invalid␣date:␣\"%s\"\n", optarg);
                goto printhelp;
        }
        break;
case 'm': // metadata
{
        Metadata m(optarg);
        if(m.key == "") {
                fprintf(stderr,"invalid␣metadata:␣\"%s\"\n", optarg);
                goto printhelp;
        }
        s.metadata.push_back(m);
        break;
}
case 'T': // type
{
        s.type = StreamType::fromString(optarg);
        if(s.type == (StreamType::type)-1) {
                fprintf(stderr,"invalid␣type:␣\"%s\"\n", optarg);
                goto printhelp;
        }
        break;
}

case 'r': // relative
        s.flags &= ~StreamFlag::time_absolute;
        break;
case 'a': // arbitrary
        s.flags &= ~StreamFlag::time_real;
        break;

case 'q': // quiet
        quiet = true;
        break;

case 'A': // annotate
        annotate = true;
        break;

case 'V':
        printf("extract␣%s\nId:\n␣%s\n", svnrev, svnid);
        printf("Written␣by␣Jim␣Paris␣<jim@jtan.com>.\n");
        return 0;
        break;
case 'h':
        help=stdout;
printhelp:
```

```
                fprintf(help,"\n");
            default:
                fprintf(help,"Extracts records from the NILM database.\n");
                fprintf(help,"Data is supplied as ASCII text to stdout.\n\n");
                fprintf(help,"Usage: %s [options]\n\n",*argv);
                opt_help(opt,help);
                fprintf(help,"\n%s", NilmUtil::datespec);
                fprintf(help,"\nValid stream types:\n  %s\n",
                        StreamType::describe().c_str());
                return 1;
            }
        }

        if(s.tag == "" || s.type == (StreamType::type)-1) {
            fprintf(help, "*** Extract: you must specify a stream tag and type\n");
            goto printhelp;
        }

        if(s.end_time > 0 && s.end_time < s.start_time) {
            fprintf(help, "*** Extract: start time (%lf) must be before "
                    "end time (%lf)\n", s.start_time, s.end_time);
            goto printhelp;
        }

        /* Now go extract the stream, if we can */
        try {
            Nilmdb nilm(db.c_str(), host.c_str(), user.c_str(), pass.c_str());
            extract_stream(nilm, s);
        } catch (const mysqlpp::Exception &e) {
            fprintf(stderr,"*** Extract: %s\n", e.what());
            return -1;
        }
        return 0;
}
```

dump.cpp

Implementation of `dump`, which prints information about streams in the database.

```
#include <mysql++.h>
#include "stream.h"
#include "nilmdb.h"
#include "opt.h"
#include "dbopt.h"

using namespace std;

const char svnid[] = "$Id: dump.cpp 1579 2006-01-20 04:12:30Z jim $";
const char svnrev[] = "$Rev: 1579 $";

struct options opt[] = {
        dbopt::db, dbopt::host, dbopt::user, dbopt::pass,
        { 'i', "id", "num", "only␣dump␣the␣given␣stream␣id" },
        { 'V', "version", NULL, "show␣program␣version␣and␣exit" },
        { 'h', "help", NULL, "this␣help" },
        { 0, NULL, NULL, NULL }
};

int main(int argc, char *argv[])
{
        int optind;
        char *optarg;
        Nilmdb::StreamId id = 0;
        FILE *help = stderr;
        char c;
        string db, host, user, pass;
        int printed = 0;

        opt_init (&optind);
        dbopt::parse(0, db, host, user, pass, 0);
        while((c=opt_parse(argc,argv,&optind,&optarg,opt))!=0) {
                if (dbopt::parse(c, db, host, user, pass, optarg))
                    continue;
                switch(c) {
                case 'i':
                        id = (Nilmdb::StreamId) atoi(optarg);
                        break;
                case 'V':
                        printf("dump␣%s\nId:\n␣%s\n", svnrev, svnid);
                        printf("Written␣by␣Jim␣Paris␣<jim@jtan.com>.\n");
                        return 0;
                        break;
                case 'h':
                        help=stdout;
                default:
                        fprintf(help,"Usage:␣%s␣[options]\n\n",*argv);
                        opt_help(opt,help);
                        return 1;
                }
        }

        vector <Stream> v;

        printf("Dumping␣all␣database␣stream␣metainfo:\n");

        try {
                Nilmdb nilm(db.c_str(), host.c_str (), user.c_str (), pass.c_str ());

                v = nilm.FindAllStreams();
                for (unsigned int i = 0; i < v.size (); i++) {
                        if(id && v[i].getId() != id)
                                continue;
                        printf("\nDatabase␣Stream␣ID:␣%d\n", v[i].getId());
                        printf("Stream␣dump:\n%s", v[i].asText().c_str());
                        printed++;
                }
                if(id) {
                        if (!printed) printf("No matches\n");
                } else printf("\n%d shown\n", printed);
        } catch (const mysqlpp::Exception &e) {
                fprintf(stderr,"Error: %s\n", e.what());
                return -1;
        }

        return 0;
}
```

remove.cpp

Implementation of `remove`, which removes a stream from the database.

```
#include <mysql++.h>
#include "stream.h"
#include "nilmdb.h"
#include "opt.h"
#include "dbopt.h"

using namespace std;

const char svnid[] = "$Id: remove.cpp 1557 2006-01-16 05:54:18Z jim $";
const char svnrev[] = "$Rev: 1557 $";

struct options opt[] = {
        dbopt::db, dbopt::host, dbopt::user, dbopt::pass,
        { 'i', "id", "num", "id of the stream to remove" },
        { 'V', "version", NULL, "show program version and exit" },
        { 'h', "help", NULL, "this help" },
        { 0, NULL, NULL, NULL }
};

int main(int argc, char *argv[])
{
        int optind;
        char *optarg;
        Nilmdb::StreamId id = 0;
        FILE *help = stderr;
        char c;
        string db, host, user, pass;

        opt_init(&optind);
        dbopt::parse(0, db, host, user, pass, 0);
        while((c=opt_parse(argc,argv,&optind,&optarg,opt))!=0) {
                if (dbopt::parse(c, db, host, user, pass, optarg))
                    continue;
                switch(c) {
                case 'i':
```

```
                id = (Nilmdb::StreamId) atoi(optarg);
                break;
            case 'V':
                printf("remove␣%s\nId:\n␣%s\n", svnrev, svnid);
                printf("Written␣by␣Jim␣Paris␣<jim@jtan.com>.\n");
                return 0;
                break;
            case 'h':
                help=stdout;
            printhelp:
            default:
                fprintf(help,"Usage:␣%s␣[options]\n\n",*argv);
                opt_help(opt,help);
                return 1;
            }
    }

    if(!id) {
            fprintf(help, "error:␣id␣is␣mandatory\n");
            goto printhelp;
    }

    vector <Stream> v;

    try {
            Nilmdb nilm(db.c_str(), host.c_str(), user.c_str(), pass.c_str());

            v = nilm.FindAllStreams();
            for (unsigned int i = 0; i < v.size(); i++) {
                    if(v[i].getId() != id)
                            continue;
                    printf("Removing␣stream␣%d␣and␣associated␣data\n", (int)id);
                    nilm.DestroyStream(v[i]);
                    id = 0;
                    break;
            }
            if(id) {
                    char b[256];
                    sprintf(b,"Specified␣ID␣%d␣not␣found", (int)id);
                    throw NilmException(b);
            }
    } catch (const mysqlpp::Exception &e) {
            fprintf(stderr,"Error:␣%s\n", e.what());
            return -1;
    }

    return 0;
}
```

filter.pl

Perl script to manage the execution of external programs as stream filters.

```
#!/usr/bin/perl

# Use extract to get data, pass it to the specified program, then use
# insert to put it back, maintaining timestamps throughout.

use strict;
use warnings;
use Getopt::Long;
use FindBin qw($Bin);
use IO::File;
use POSIX qw(tmpnam);

# Attempt to find extract & insert commands
my $extract="$Bin/../extract/extract";
$extract="extract" unless (-x $extract);
my $insert="$Bin/../insert/insert";
$insert="insert" unless (-x $extract);

# Usage string
my $usage = <<EOF;
Usage: $0 [options]

  --in 'string'        Metadata options to pass extract, see below
  --out 'string'       Metadata options to pass insert, see below
  --filter 'command'   Filter command and arguments
  --start 'datespec'   Start time of input and output stream
  --end 'datespec'     End time of input and output stream
  --annotate           Filter expects and provides timestamps
  --tempfile           Always use temporary files for storage

Metadata options are passed as a complete string with options:
  --tag --type --metadata --relative --absolute --noslice
as appropriate, as well as standard database selection options.
The tag, type, and start and end times are mandatory.

All arguments are passed to another shell, so special characters
must be quoted, e.g.  --filter 'tr e \\*'
EOF

# Get options
my ($in, $out, $filtercmd, $start, $end);
my ($annotate, $tempfile) = (0, 0);
GetOptions("in=s" => \$in,
           "out=s" => \$out,
           "filter=s" => \$filtercmd,
           "start=s" => \$start,
           "end=s" => \$end,
           "annotate!" => \$annotate,
           "tempfile!" => \$tempfile);
```

```
# Process options
die $usage unless (defined $in and defined $out and defined $filtercmd
                   and defined $start and defined $end);

# Explictly check for and disallow behavior-changing options in the
# metadata strings.
my @badopts = qw/-s --start -e --end -S --step -L --lines -A --annotate
    -R --realtime -d --delete/;
map { my $cmd=$_; map { die "Error:␣metadata␣options␣shouldn't␣include␣\"$_\"\n"
                        if ($cmd =~ qr/(^|\s)$_($|\s)/); } @badopts } ($in, $out);

# Build command we'll run
my $extractcmd = "$extract␣$in␣--start␣\"$start\"␣--end␣\"$end\"";
my $insertcmd = "$insert␣$out␣--start␣\"$start\"␣--end␣\"$end\"␣--delete";

if ($annotate and !$tempfile) {
    # Use a single pipeline for everything.  This is the easy case.
    my $cmd = "$extractcmd␣--annotate␣|␣$filtercmd␣|␣$insertcmd␣--annotate";
    print "Single␣pipeline,␣executing:␣$cmd\n";
    exec($cmd) or die "exec␣failed\n";
}

# Use a two-step process through temporary files:
# extract -> temp1 -> filter -> temp2 -> insert
my ($temp1, $temp2, $temp1fh, $temp2fh);
do { $temp1 = tmpnam() } until $temp1fh =
    IO::File->new($temp1, O_RDWR|O_CREAT|O_EXCL);
do { $temp2 = tmpnam() } until $temp2fh =
    IO::File->new($temp2, O_RDWR|O_CREAT|O_EXCL);
END {
    if ( defined($temp1) ) {
        unlink($temp1) or die "Couldn't␣unlink␣$temp1␣:␣$!\n";
    }
    if ( defined($temp2) ) {
        unlink($temp2) or die "Couldn't␣unlink␣$temp2␣:␣$!\n";
    }
}

# First run extract
open(PIPE, "$extractcmd|") or die "Can't␣execute␣$extractcmd:␣$!\n";
my $ecount = 0;
while(<PIPE>) {
    print $temp1fh $_;
    $ecount++;
}
close(PIPE);
print "Filter:␣got␣$ecount␣records␣from␣extract\n";

my $fcount = 0;
# Now run the filter.
my $cmd;
if ($tempfile) {
    # It wants filenames, so just call it and count the lines in the result.
    $cmd = "$filtercmd $temp1 $temp2";
    system($cmd) or die "Can't execute $cmd: $!\n";
    seek($temp2fh, 0, 0);
    while(<$temp2fh>) {
        $fcount++;
    }
} else {
    # It wants stdin/stdout, so we can count lines as they come
    open(PIPE,"$filtercmd < $temp1 |");
    while(<PIPE>) {
        print $temp2fh $_;
        $fcount++;
    }
}

print "Filter: got $fcount records from filter \n";

# Finally, pass the temp file into insert, with the right number of lines
my $icount = 0;
open(PIPE, "|$insertcmd --lines $fcount") or die "Can't execute $insertcmd: $!\n";
seek($temp2fh, 0, 0);
while(<$temp2fh>) {
    print PIPE $_;
    $icount++;
}
close(PIPE);

print "Filter: wrote $icount records to insert\n";

exit 0;
```

A.4.4 Matlab/Octave Functions

insert.m

Matlab/Octave function that uses the `insert` program to store data.

```
function insert(v, a, b)
%%% Insert data into the NILM database by running the "insert" binary
%%% and passing the data in v. Arguments are as specified by "insert".
%%% Example usage:
%%%
%%% data = rand(10);
%%% insert(data, ...
%%%        '--tag "Hello" --type 0 -r -a --start 0 --end 10 --lines 10');
%%%
%%% $Id: insert.m 1598 2006-01-22 01:00:33Z jim $
  if (nargin < 2)
    error('Need to supply data and arguments to "insert" binary');
  end
  if (nargin < 3)
    %% Assume path to insert if they don't pass it
    command = ['../prog/insert/insert␣' a];
  else
    command = [a '␣' b];
  end

  %% Make a temporary filename
  temp = tempname;

  %% Octave and Matlab use different syntax for saving
  disp('Storing␣data␣to␣be␣inserted ... ');
  if which('octave_config_info') ~= ''
    save('-text',temp,'v');
  else
    save(temp,'v','-ascii');
  end

  %% Spawn the insert binary with the given arguments, and save output
  cmd = ['env␣-i␣' command '␣<␣' temp] ;
  disp(['Executing:␣' cmd ]);

  %% Octave and Matlab use different syntax for system command
  if which('octave_config_info') ~= ''
    fflush (1);
    [output, status] = system(cmd);
    if status ~= 0
      delete(temp);
      error(['Error␣executing␣command!' char(10)]);
    end
  else
    [status, output] = system(cmd);
    disp(output);
    if status ~= 0
      delete(temp);
      error('Error␣executing␣command!');
    end
  end

  %% All done
  delete(temp);
end
```

extract.m

Matlab/Octave function that uses the `extract` program to retrieve data.

```
function v = extract(a, b)
%%% Extract data from the NILM database by running the "extract" binary
%%% and collecting the output. Arguments are as specified by "extract".
%%% Example usage:
```

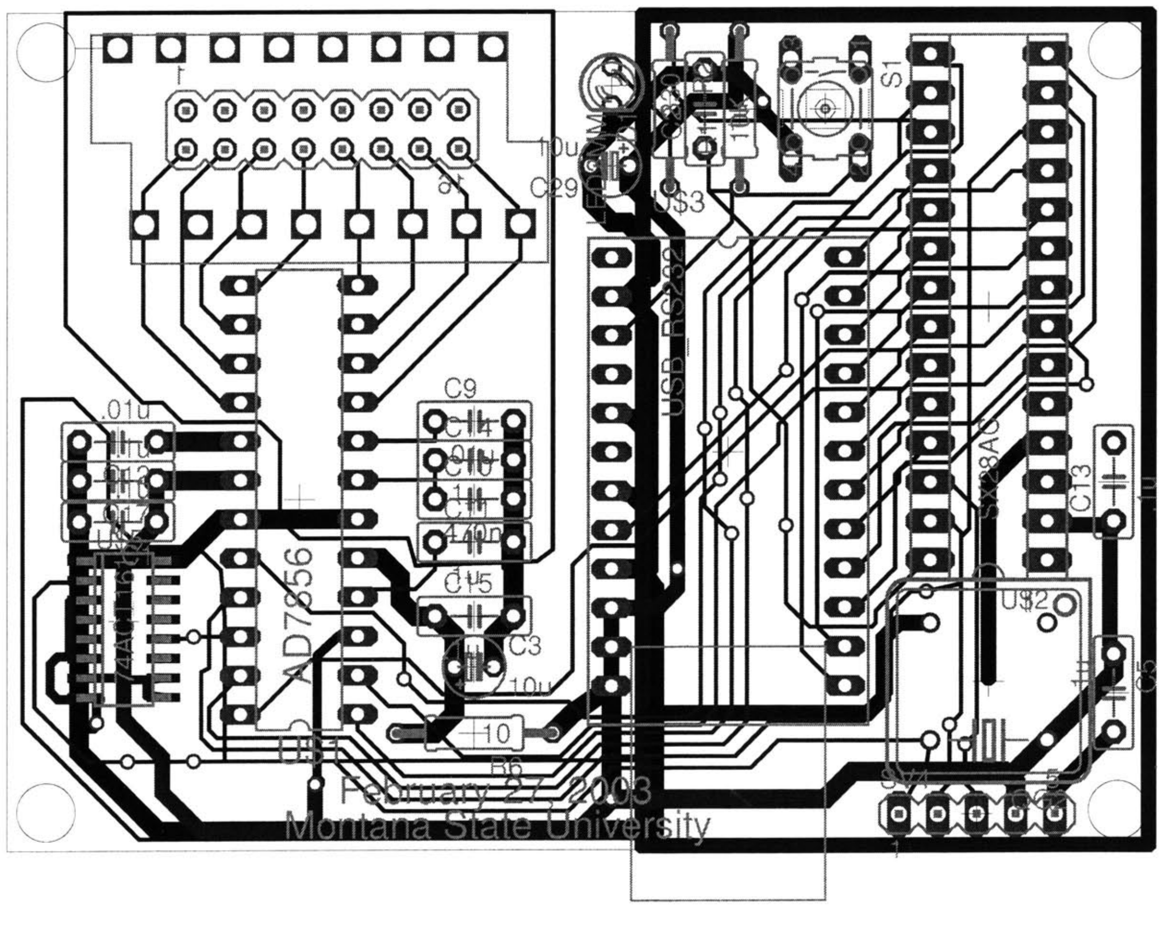
```
%%%
%%% data = extract('--tag "Hello" --type 0');
%%%
%%% $Id: extract.m 1598 2006-01-22 01:00:33Z jim $
  if (nargin < 1)
    error('Need to supply arguments to "extract" binary');
  end
  if (nargin < 2)
    %% Assume path to extract if they don't pass it
    command = ['../prog/extract/extract ' a];
  else
    command = [a ' ' b];
  end

  %% Make a temporary filename
  temp = tempname;

  %% Spawn the extract binary with the given arguments, and save output
  cmd = ['env -i ' command ' > ' temp] ;
  disp(['Executing: ' cmd ]);

  %% Octave and Matlab use different syntax for system command
  if which('octave_config_info') ~= ''
    fflush (1);
    [output, status] = system(cmd);
    if status ~= 0
      delete(temp);
      error(['Error executing command!' char(10)]);
    end
  else
    [status, output] = system(cmd);
    disp(output);
    if status ~= 0
      delete(temp);
      error('Error executing command!');
    end
  end

  %% Read in the file
  disp('Loading extracted data...');
  errmsg='Error parsing the output file, bad data in db?';
  eval('v = load(temp);','delete(temp); error(errmsg);');

  %% All done
  delete(temp);
end
```

Appendix B

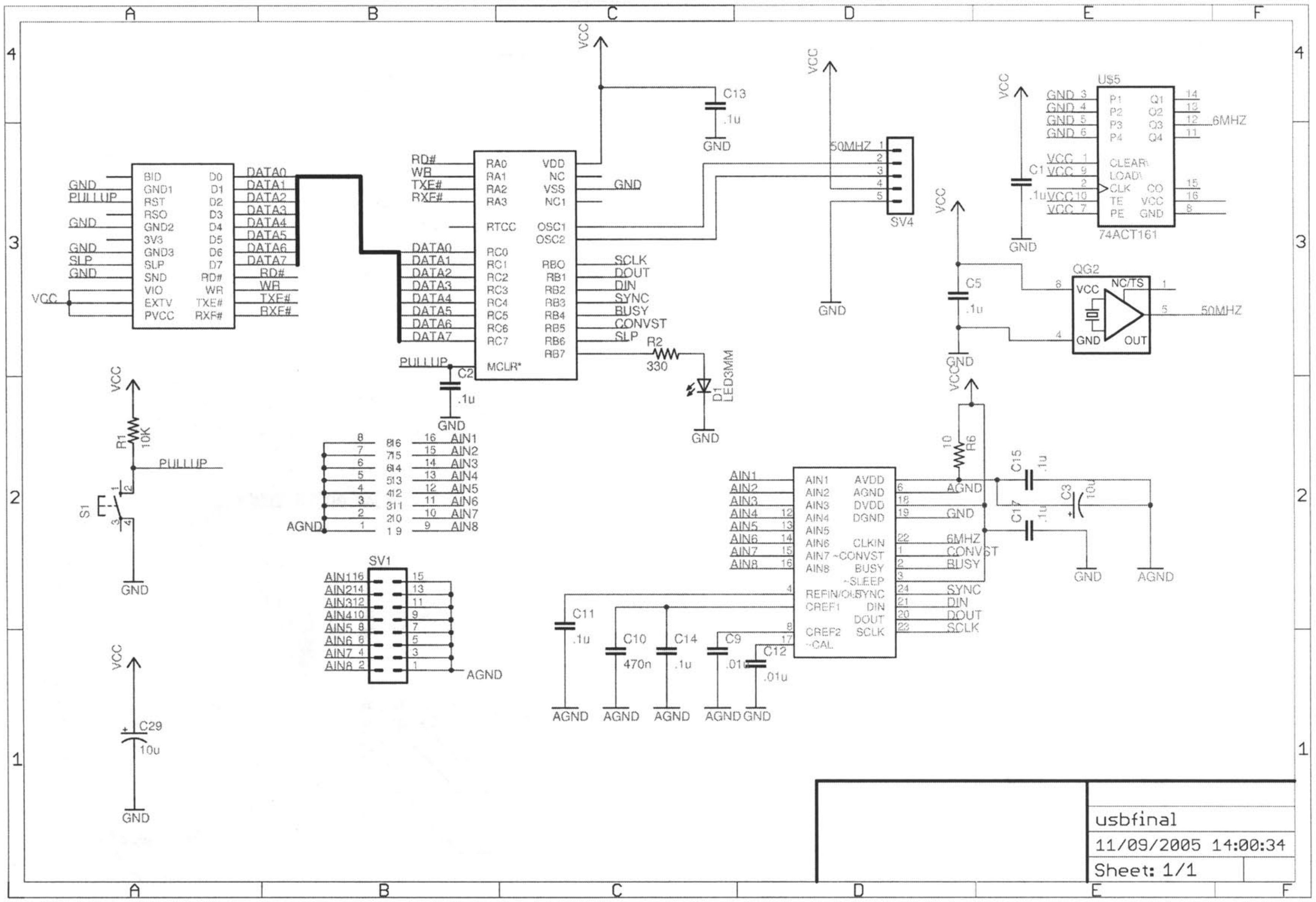
Hardware Design

These layouts were provided by Professor Steven R. Shaw of the Montana State University.

B.1 USB ADC Board Layout



B.2 USB ADC Schematic



usbfinal
11/09/2005 14:00:34
Sheet: 1/1
BID GND1 RST RSO GND2 3V3 GND3 SLP SND VIO EXTV PVCC
D0 D1 D2 D3 D4 D5 D6 D7 RD# WR TXE# RXF#
DATA0 DATA1 DATA2 DATA3 DATA4 DATA5 DATA6 DATA7
GND PULLUP SLP VCC
RA0 RA1 RA2 RA3 RTCC RC0 RC1 RC2 RC3 RC4 RC5 RC6 RC7 MCLR*
VDD NC VSS NC1 OSC1 OSC2 RB0 RB1 RB2 RB3 RB4 RB5 RB6 RB7
SCLK DOUT DIN SYNC BUSY CONVST SLP
C13 .1u
C2 .1u
R2 330
D1 LED3MM
50MHZ SV4
U$5 74ACT161
P1 P2 P3 P4 CLEAR\ LOAD\ CLK TE PE
Q1 Q2 Q3 Q4 CO VCC GND
6MHZ
C1 .1u
QG2 VCC NC/TS GND OUT
C5 .1u
R1 10K
S1
C29 10u
SV1
AIN1 AIN2 AIN3 AIN4 AIN5 AIN6 AIN7 AIN8
AGND
AVDD AGND DVDD DGND CLKIN ~CONVST BUSY ~SLEEP REFIN/OUT SYNC CREF1 DIN DOUT CREF2 SCLK ~CAL
R6 10
C15 .1u
C3 10u
C17 .1u
C11 .1u
C10 470n
C14 .1u
C9 .01u
C12 .01u

Bibliography

[1] *American Society of Naval Engineers Reconfigurability and Survivability Symposium*, Atlantic Beach, Florida, February 2005.

[2] D. P. Bertsekas and J. N. Tsitsiklis. *Introduction to Probability.* Athena Scientific, Belmont, MA, 2002.

[3] L. Carmichael. Nonintrusive appliance load monitoring system. *EPRI Journal*, pages 45–47, September 1990.

[4] K. R. Cho. Detection of broken rotor bars in induction motors using parameter and state estimation. Master's thesis, Massachusetts Institute of Technology, June 1989.

[5] Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, and Philips. Universal serial bus specification, revision 2.0, April 2000.

[6] T. DeNucci. Diagnostic indicators for shipboard systems using non-intrusive load monitoring. Master's thesis, Massachusetts Institute of Technology, Department of Mechanical Engineering, June 2005.

[7] T. DeNucci, R. Cox, S. B. Leeb, J. Paris, T. J. McCoy, C. Laughman, and W. Greene. Diagnostic indicators for shipboard systems using non-intrusive load monitoring. In *IEEE Electric Ship Technologies Symposium*, Philadelphia, Pennsylvania, July 2005.

[8] W. Greene. Evaluation of non-intrusive monitoring for conditional based maintenance applications on us navy propulsion plants. Master's thesis, Massachusetts

Institute of Technology, Department of Ocean Engineering and Department of Mechanical Engineering, June 2005.

[9] W. Greene, J. S. Ramsey, S. B. Leeb, T. DeNucci, J. Paris, M. Obar, R. Cox, C. Laughman, and T. J. McCoy. Non-intrusive monitoring for condition-based maintenance. In ASNE [1].

[10] C. R. Laughman, S. R. Shaw, S. B. Leeb, L. K Norford, R. W. Cox, K. D. Lee, and P. Armstrong. Power signature analysis. *IEEE Power and Energy Magazine*, pages 56–63, March 2003.

[11] S. B. Leeb. *A Conjoint Pattern Recognition Approach to Nonintrusive Load Monitoring.* Phd, MIT, Department of Electrical Engineering and Computer Science, February 1993.

[12] J. S. Ramsey, S. B. Leeb, T. DeNucci, J. Paris, M. Obar, R. Cox, C. Laughman, and T. J. McCoy. Shipboard applications of non-intrusive load monitoring. In ASNE [1].

[13] S. R. Shaw. *System Identification Techniques and Modeling for Nonintrusive Load Diagnostics.* Phd, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, February 2000.

[14] S. R. Shaw and C. R. Laughman. A kalman-filter spectral envelope preprocessor. Submitted to IEEE Transactions on Instrumentation and Measurement, February 2004.

[15] E. W. Weisstein. Mathworld. Available `http://mathworld.wolfram.com/`.

[16] Wikipedia. Erlang distribution. Available `http://en.wikipedia.org/w/index.php?title=Erlang_distribution&oldid=28498166`.

[17] Wikipedia. Queueing theory. Available `http://en.wikipedia.org/w/index.php?title=Queueing_theory&oldid=34256671`.

3262-68