

Wattsworth:
A Vision for Cyber Physical System Design

by

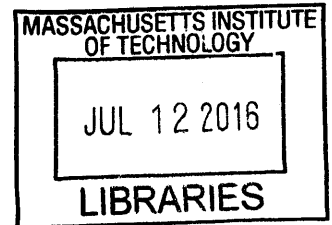
John Donnal

B.S.E., Princeton University (2007)

M.S., University of Maryland University College (2009)

M.S., Massachusetts Institute of Technology (2013)

Eng., Massachusetts Institute of Technology (2015)



ARCHIVES

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

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2016

© Massachusetts Institute of Technology 2016. All rights reserved.

Signature redacted

Author

.....

Department of Electrical Engineering and Computer Science

May 20, 2016

Signature redacted

Certified by ...

.....

Steven B. Leeb

Professor of Electrical Engineering and Computer Science

Signature redacted Thesis Supervisor

Accepted by

.....

✓ ✓ Leslie A. Kolodziejski

Chair, Committee on Graduate Students

Wattsworth: A Vision for Cyber Physical System Design

by

John Donnal

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2016, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

The combination of powerful and inexpensive embedded computers, advanced sensor technology, and high speed wireless networks could revolutionize how we interact with our physical environment. Sensor networks that provide real time feedback offer significant value in terms of energy reduction, fault detection, equipment diagnostics, monitoring, security and more. This revolution will not happen in a positive way without a clear vision of how sensor, network, and control technologies can be applied to enhance human abilities and improve our lives.

Such systems have been frustratingly difficult to implement. An old dilemma is becoming increasingly apparent. Networking provides remote access to information and control inputs. Gathering useful information, however, may require the installation of an expensive and intrusive array of sensors. Without this array, networked control provides colorful but minimally useful real information. Technological marvels like solid-state or micro-electromechanical sensors may ultimately reduce the cost of individual sensors through mass-production. They may not, however, reduce installation expense. They also do nothing to recover waste of resources. Even with the array, it may be difficult for a facilities operator to make informed control and maintenance decisions that intelligently affect mission critical components. Large datasets remain difficult to use.

This thesis presents a design approach for creating cyber physical infrastructure that addresses these challenges to delivering actionable real time feedback. At the core of the system is a suite of non-intrusive sensors that dramatically reduce the cost of data acquisition. These sensors process and store data locally, without any dependency on external servers. This removes the security and privacy concerns that plague conventional sensor networks. A decentralized cloud infrastructure securely connects users to sensor platforms and provides powerful visualization and programming interfaces to customize data presentation.

This work covers the complete system design from embedded analog sensors to enterprise grade backend server architecture, to the frontend human computer interface. Such a wholistic design approach is critical to ensure a cyber physical system delivers quantifiable value to the end user. Several case studies illustrate the success of this design approach, including an automatic watch stander system for the

US Coast Guard, an energy monitoring platform for a US Army Base, and realtime equipment diagnostic platforms installed in a wide variety of environments including an Army hospital and a local elementary school.

Thesis Supervisor: Steven B. Leeb

Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

None of this research would have been possible without my advisor, Professor Steven Leeb. Thank you for your mentorship and thoughtful guidance throughout my graduate studies. Your passion for teaching is infectious and has inspired me to pursue a career in academics. I look forward to a career of exciting collaborations. I also wish to thank Professor Steven Shaw, Professor Leslie Norford, and Professor James Kirtley for their input and assistance throughout my research and for serving on my thesis committee. This work would not have been possible without their help.

Essential support for this research was graciously provided by the MIT Energy Initiative, the BP-MIT Research Alliance, The Grainger Foundation, the Office of Naval Research, and the Massachusetts School Board Authority.

I am extremely grateful to everybody who has worked to bring this research out of the lab and make it a practical reality. In particular Ken Wertz and Scott Schertz of Cottage Elementary School, the Coast Guard crews of the USCGC SPENCER, USCGC ESCANABA, and USCGC SENECA, the US Navy crews of the LPD SAN DIEGO, LCS INDEPENDENCE, and DDG MICHAEL MURPHY, Robert Strong, Michael Mobley, and Richard Lucas of MIT Maintenance and Utilities, the support staff from the Base Camp Integration Lab at Fort Devens, especially Bill Singleton, and the many family and friends who opened their apartments and homes for this project.

I would also like to extend thanks to my many peers and colleagues including Mark Gillman, Ryan Zachar, William Cotta, Pete Lindahl, Greg Bredariol, Kristen Severson, David Lawrence, Bart Sievenpiper, Katy Gerhard, Chris Schantz, Jin Moon, Arijit Banerjee, Yiou He, Uzoma Orji, Kawin Surakitbovorn, and Dan Vickery. Dr. James Paris provided the foundation of this work and continues to be a deep technical collaborator. He is not only a great colleague but a great friend.

Finally and most importantly, thank you to my family for their endless and unconditional encouragement. Nicky, thank you for letting me stay in school so long. This is my last degree! Promise! John Wadsworth, now it's your turn.

Contents

1	Introduction	19
1.1	Non-Intrusive Sensors	20
1.2	Embedded Signal Processing	22
1.3	Distributed Cloud Architecture	25
2	Non-Intrusive Sensors	27
2.1	Introduction	27
2.2	Non-Contact Current Sensor	28
2.2.1	Hall Effect Sensor	28
2.2.2	Tunneling Magnetoresistive Sensor	31
2.3	Non-Contact Voltage Sensor	36
2.3.1	Principle of operation	38
2.3.2	Analog implementation	40
2.3.3	Digital signal processing	44
2.3.4	Experimental results	50
2.4	Ancillary Sensor Platforms	56
2.4.1	Captcha: A Vibration Diagnostic Platform	57
2.4.1.1	Hardware	58
2.4.1.2	Firmware	59
2.4.1.3	Software	61
2.4.1.4	Operational Verification	61
2.4.2	Hottee: A multi-domain thermal imager	63
2.4.2.1	System Architecture	63

2.4.2.2	Signal Processing	64
2.4.2.3	FPGA Implementation	66
2.4.2.4	Image Reconstruction	67
3	Signal Processing	69
3.1	Introduction	69
3.2	Non-Contact Power Measurements	70
3.2.1	Multi-Conductor Power Systems	70
3.2.1.1	Monitoring a Circuit Breaker Panel	70
3.2.1.2	Cables with Neutral Return Path	72
3.2.1.3	Example Reconstruction	73
3.2.2	System Calibration	74
3.2.2.1	Single Phase Systems	75
3.2.2.2	Multiphase Systems	80
3.2.2.3	Rapid Calibration	82
3.2.3	Power Measurement Example	82
3.3	Vibration Transfer Function	86
3.3.1	Motivation	86
3.3.2	Background	87
3.3.3	Sensor Measurements and eVTF Generation	91
3.3.3.1	Data Collection	91
3.3.3.2	Short-Time Fourier Transform Analysis	95
3.3.4	Tests on Purpose Built Machine Set	100
3.3.4.1	Comparison of Spin-down eVTF and Steady-state eVTF100	
3.3.4.2	eVTFs with Condition Changes	102
3.3.5	Field Tests on U.S. Navy Equipment	104
3.3.6	Conclusions	108
4	Distributed Cloud	109
4.1	Introduction	110
4.2	NILM Virtual Private Network	111

4.3	Web Platform	113
4.4	Data Visualization	113
4.4.1	Stream Configuration	113
4.4.2	Presentation	115
4.5	Data Processing	116
4.5.1	Management and Preprocessor	117
4.5.1.1	Multistream Wrapper	117
4.5.1.2	Resampler	118
4.5.2	Stream Iterators	118
4.5.3	Reports	119
4.5.4	NILM Manager IDE	121
4.6	Designing Energy Apps	122
4.6.1	Cycling System App	122
4.6.2	Power Quality App	124
4.6.3	Adding Control to an Energy App	127
5	Case Studies	131
5.1	Cottage Elementary: Energy Scorekeeping	131
5.1.1	Electrical System Background	132
5.1.2	Load Disaggregation	135
5.1.3	Case Study Results	138
5.1.4	Implications	143
5.2	US Army: Continuous Commissioning	144
5.2.1	Fort Devens, MA	145
5.2.2	Fort Polk, LA	146
5.2.3	Top-Down Monitoring for Energy Savings	146
5.2.3.1	HVAC Operation Schedule	147
5.2.3.2	Misconfigured ECUs	148
5.2.4	Bottom-Up Fault Detection and Diagnostics	149
5.2.4.1	ECU Fault	149

5.2.5	Implications	151
5.3	US Coast Guard: Equipment Monitoring	151
5.3.1	Shipboard Automatic Watchstander	152
5.3.2	Automatic Logging	152
5.3.2.1	A Bellwether for Crew Performance	154
5.3.2.2	Ensuring Compliance with Operating Procedures . . .	155
5.3.3	Installation on the USCGC SPENCER	156
5.3.4	Signal Processing and Transient Identification	160
5.3.5	Case Study Results	166
5.3.6	Implications	169
A	Documentation	171
A.1	User Quick Start Guides	172
A.1.1	Setting Up a Contact Meter	173
A.1.2	Setting up a Non-Contact Meter	176
A.1.3	Configuring Smart Plugs	179
A.1.4	Adding Custom Datasets	181
A.2	User Reference	183
A.2.1	Hardware Configuration	183
A.2.2	Software Configuration	188
A.2.3	Smart Plugs	200
A.2.4	NILM Administration	215
A.2.5	Data Explorer	220
A.2.6	Designing NILM Filters	226
A.2.7	Designing NILM Analyzers	242
A.2.8	Automatic Transient Finder	252
A.2.9	NILM Process Manager	255
A.2.10	Command Line Interface	259
B	Implementation	287
B.1	Flex Sensor	288

B.1.1	PCB Design	291
B.1.2	Selected Firmware Files	293
B.2	NILM Smart Plug	299
B.2.1	PCB Design	302
B.2.2	Bill of Materials	304
B.2.3	Assembly Notes	305
B.2.4	Selected Firmware Files	307
B.3	NILM Software Stack	317
B.3.1	Data Capture	319
B.3.1.1	Standalone Capture	319
B.3.1.2	Embedded Capture	322
B.3.2	Embedded Management	324
B.3.3	NILM Board Schematics	327
B.3.4	NILM Board Bill of Materials	341
B.4	Server Architecture	342
B.4.1	Firewall	342
B.4.2	Backbone	342
B.4.3	Web	343
B.4.4	Metrics	343
B.4.5	Archive	343
B.4.6	Devops	343
B.4.7	Git	344

List of Figures

1 Introduction

1-1	Non-Intrusive Sensor Designs	21
1-2	Retrofit flow sensor for utility water meter	22
1-3	Vibration diagnostic sensor for rotating machinery	23
1-4	Hardware renderings of the Wattsworth power meter	24
1-5	Distributed cloud architecture	26

2 Non-Intrusive Sensors

2-1	Non-contact Hall Effect sensor	29
2-2	Hall Effect non-contact sensor prototype	29
2-3	Experimental setup for sensor characterization	30
2-4	Hall Effect Linearity Plot	31
2-5	Uncompensated TMR response	32
2-6	TMR compensation circuit	33
2-7	TMR feedback technique	33
2-8	Compensated TMR Linearity Plot	34
2-9	Utility water meter	35
2-10	Compensated TMR schematic for water flow	35
2-11	Retrofit flow meter using TMR sensors	36
2-12	Circuit for differential non-contact sensing of an AC voltage	38
2-13	Non-contact A voltage sensors schematic and 3D PCB stackup	41
2-14	Non-contact voltage sensor theory of operation	42

2-15	Impulse response h_1 for $N = 25$	46
2-16	Impulse response h_2 for $N = 25$	47
2-17	Filter magnitudes for $N = 25$	47
2-18	Relative deviation from ideal frequency response for $N = 25$	48
2-19	Simulated filter response to an impulsive disturbance	48
2-20	Simulated response of each filter to the same sequence of pink noise . .	49
2-21	Non-contact power meter experimental setup	51
2-22	Data collected by non-contact and traditional power meters	52
2-23	Non-contact voltage sensor experimental setup	53
2-24	Response to electromagnetic disturbance	54
2-25	“Captcha” vibration diagnostic platform	57
2-26	“Captcha” sensor on USS SAN DIEGO (LPD 22)	58
2-27	Functional groups on vibration sensor circuit board	59
2-28	Captcha circuit board	60
2-29	Accelerometer chip mounted with a pipe clamp	62
2-30	Custom sensor package adhesively mounted	62
2-31	Composite thermal overlay	63
2-32	Thermal imager system architecture	64
2-33	Thermal imager colormap	67
2-34	Composite thermal overlay	68

3 Signal Processing

3-1	Magnetic field sensor versus contact current sensor	71
3-2	Monitoring a circuit breaker panel with TMR sensors	71
3-3	Corrected magnetic field sensor versus contact current sensor	73
3-4	PWM calibration load as detected by a non-contact sensor	75
3-5	Monitoring a three phase power line with non-contact sensors	83
3-6	Visualization on non-contact sensor vectors	84
3-7	Three phase non-contact measurements	85
3-8	Electromechanical machinery and mount system	88

3-9	Non-contact back-EMF speed sensor	93
3-10	Example time-domain signals used for eVTF generation	96
3-11	Windowed virtual input	97
3-12	Hanning Windows and Masked Inputs and Outputs	97
3-13	FFT of Masked Inputs and Outputs	99
3-14	Test platform for eVTF diagnostics	101
3-15	Spin-down versus Steady State eVTF's	102
3-16	Diagnostic trends in the eVTF	103
3-17	Mine countermeasure ship eVTFs	106
3-18	Auxiliary seawater pumps 1 and 2 spin-down generated eVTFs	107
4	Distributed Cloud	
4-1	NILM Manager system architecture	111
4-2	NILM network topology as visualized by Nagios.	112
4-3	The NILM configuration interface.	114
4-4	Data visualization using the web plotting tool	114
4-5	NILM Manager plot types	116
4-6	The stages of a NILM report	120
4-7	NILM Manager IDE for designing Energy Apps	121
4-8	Example NILM report	123
4-9	Training the load identifier on shop equipment	125
4-10	Identifying large shop loads	126
4-11	Energy App to identify the causes of voltage transients.	128
4-12	Smart plugs allow the NILM to monitor and control loads	129
4-13	Power Quality App protecting the 3D printer	130
5	Case Studies	
5-1	Cottage NILM Schematic and Installation	132
5-2	Cottage Heating System	133
5-3	Loads monitored by NILM at Cottage Elementary School	134

5-4	One hour of real power data at Cottage	135
5-5	Load transients modeled with step functions	136
5-6	Comparison of original signal with modeled signal	137
5-7	Average Usage and Cost per machine over 6-day period	139
5-8	Cost comparison of Boiler 1 to Boiler 2	141
5-9	Voltage transient caused by VFD	142
5-10	An example training FOB located at Fort Polk	145
5-11	Loads at Fort Devens itemized by energy consumption	146
5-12	Power usage at Fort Devens over an occupied weekend.	147
5-13	Fort Polk ECU conflict	149
5-14	Fort Polk healthy versus damaged ECU	150
5-15	Typical USCG engine room watch	154
5-16	The USCGC Spencer: 270ft Famous Class Cutter	156
5-17	Standard Coast Guard exhaust fan.	157
5-18	USCGC SPENCER engine room	158
5-19	Controllable Pitch Propeller Pump	158
5-20	MPDE prelube pump	159
5-21	A month of electrical power data	160
5-22	Monitoring data from 10 December 2014	161
5-23	The MPDE starting as observed by the nonintrusive SAW.	162
5-24	Characteristic CPP on and off event.	162
5-25	Monitoring data from 10 December 2014	163
5-26	MPDE Stop Sequence	164
5-27	Image of unusual readings from CPP pump after maintenance.	165
5-28	Unusual readings from CPP pump during mooring	166
5-29	Startup transients for USCGC SPENCER equipment	167
5-30	Machinery log maintained by crew	168
5-31	Automatically generated machinery log	169

B Implementation

B-1 Mylar pickup schematic	288
B-2 Mylar pickup	289
B-3 Host PCB	289
B-4 Complete flex sensor	290
B-5 Control PCB top view	300
B-6 Control PCB bottom view	300
B-7 Solid state meter detail	301
B-8 Smart Plug components	301
B-9 Embedded NILM Hardware	319
B-10 NILM data capture	320
B-11 Sensor reader	321
B-12 Embedded data capture	323
B-13 Embedded system management	326

Chapter 1

Introduction

Much of the current thinking for making systems “smart” takes advantage of inexpensive hardware and fast wireless networks in a loose design approach that gives little thought to the actual problems facing the end user. In energy monitoring, for example, utilities have installed millions of smart meters that provide little or no actionable information to the facilities owner to help manage or reduce consumption. Engine rooms of modern naval vessels bristle with sensors, each generating data that is faithfully recorded, and typically ignored. Data meant to help becomes a hindrance when operators have to debug hundreds of sensors to find the source of a false alarm. From the smart home to the smart grid, sensors generate clouds of data that overwhelm instead of inform. Databases storing this information have grown so large that analyzing them has become an academic discipline in itself. Even sophisticated players in the data analysis market find themselves unable to capitalize on the promise of cyber-physical systems with both Microsoft and Google canceling their respective energy monitoring projects soon after inception [1]. Designing truly functional cyber-physical systems requires both an analytic mastery and practical expertise. Industry fails to appreciate the complexities of sensor design, looking for quick profit with off-the-shelf components, while academics focus on isolated algorithms and circuits at the expense of the system. This research combines analytical rigor with attention to the practical details required to deploy a complete system from the front-end sensor and signal processing to the back-end network encryption and server architecture

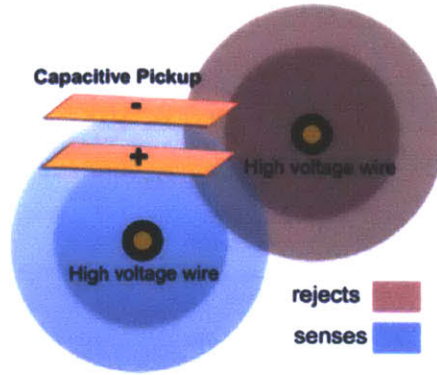
necessary to bring actionable information to the end user. This work is running in homes, ships, military bases, and schools.

1.1 Non-Intrusive Sensors

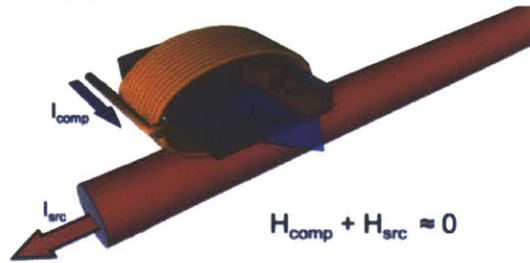
System design begins with the sensor. Real time electricity meters enable energy conservation but require current and voltage sensors that are expensive and inconvenient. These sensors require Ohmic contact to measure voltage and geometric isolation of each phase to measure current. Installing such a system involves a trained electrician and a service interruption that often costs more than the savings promised. To make these systems practical I designed non-contact sensors that can measure both current and voltage from outside the insulation of a power line making it safe and easy to install. The sensor measures voltage by sampling the electric field with the differential capacitive design shown in Figure 1-1a. The currents are extracted from the magnetic field, which is a superposition of the phase and neutral lines. These currents produce zero net flux outside the cable, but sensitive devices like tunneling magnetoresistive (TMR) elements can detect the magnetic dipoles that escape due to the wire geometry. TMR elements offer high sensitivity but poor linearity, a drawback that I resolved with magnetic feedback to set a zero bias operating point at the sensor interface (see Figure 1-1b) [2].

Multiple magnetic sensors along two flexible arms collect linearly independent measurements, and recover the current in each conductor through an inversion matrix. The complete sensor, shown in Figure 1-4a, costs thirty dollars and takes less than five minutes to install and calibrate.

Water flow meters are similarly intrusive, requiring a plumber to insert monitoring equipment inline with the pipe. Working with a mechanical engineer I designed a retrofit sleeve that converts a standard meter used for utility billing into a high bandwidth flow rate monitor (see Figure 1-2). The system attaches with a zip tie and matches the accuracy of industry standard inline devices [3]. Two compensated TMR elements on the sleeve track the rotation of the magnetic displacement wheel within



(a) Non-intrusive voltage sensor



(b) Non-intrusive current sensor

Figure 1-1: Non-Intrusive sensors provide access to rich datasets with minimum installation cost and no service interruption.

the meter’s brass enclosure. Using a version of the Hilbert transform this rotational frequency provides a precise flow rate measurement [4].

Both the energy meter and flow rate sensors are designed to be accurate as well as non-intrusive. As hardware costs continue to decrease sensor installation and maintenance begin to dominate the total cost of ownership, making non-intrusive designs particularly advantageous. Non-intrusive systems also provide exciting possibilities for in-situ diagnostics where physical and operational constraints make traditional sensor platforms impractical. For example, the US Navy and Coast Guard want to develop better ways to perform diagnostics on ship motors and generators. Current practice requires intrusive instrumentation. Sensors developed in this thesis can provide similar diagnostics without electrical or mechanical connection to the equipment which means they can run underway [5,6]. Figure 1-3 shows these sensors installed aboard the USS SAN DIEGO. This gives operators real time assessment of machin-

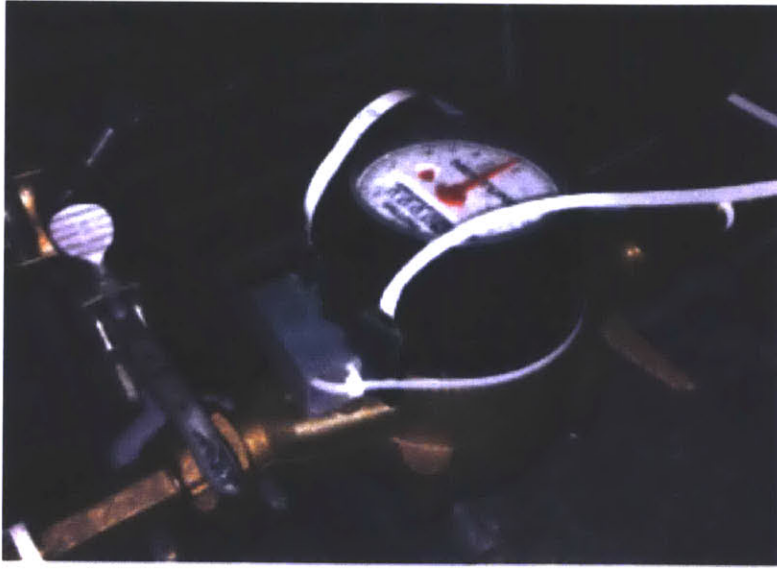


Figure 1-2: Retrofit flow rate sensor for a utility water meter

ery health in forward deployed environments where such information is critical for survivability.

1.2 Embedded Signal Processing

Non-intrusive implies reducing total sensor count. By applying the appropriate signal processing a single sensor placed in the right location can provide data equivalent to dozens of distributed sensors that would require complex communication protocols, power, and of course installation. When only one sensor is required, more resources can be devoted to its design. Recent advancements in mobile computing have lowered the cost and power consumption of microcontrollers to the point where powerful 32 bit systems with floating point DSP can be directly embedded in the sensor platform. The next generation of sensors should not just measure and transmit; they should process their own data locally. This design enables richer signal acquisition because data does not have to move across a network. It also eliminates the ethical complexities of using external storage providers like Microsoft and Google, who may have ulterior motives with user data. Current smart meters report power consumption at most once per second, but electrical diagnostics and load identification require sampling

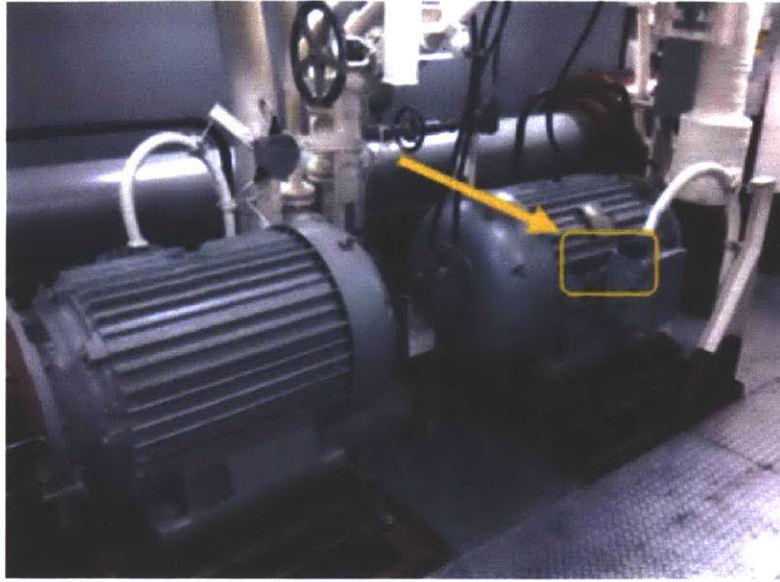
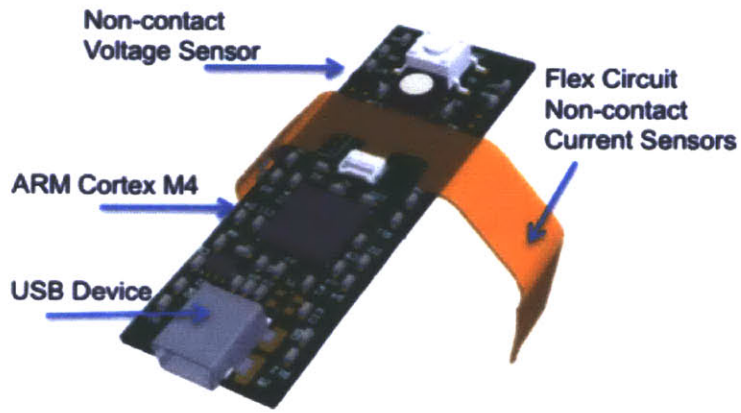
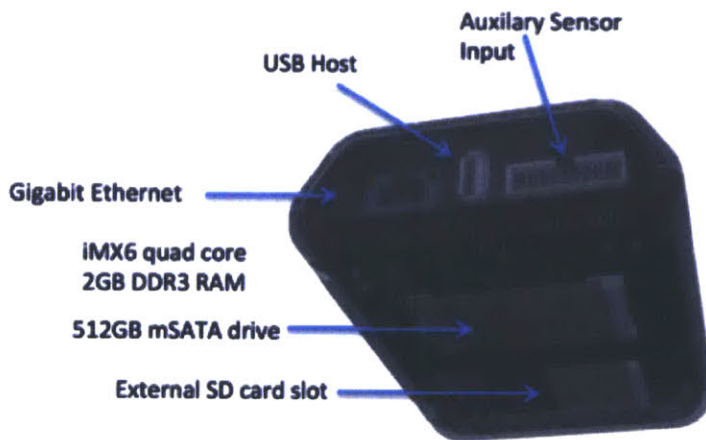


Figure 1-3: Vibration diagnostic sensor for rotating machinery

at a kilohertz or more. I have devoted a large portion of my research to the design of a power meter that is not constrained to these low sample rates by using highly optimized signal processing to store and process meter data locally. The Wattsworth power meter shown in Figure 1-4 uses non-contact sensors to measure current and voltage at 3kHz, which on a three-phase system generates over 2GB of data per day [7]. An embedded Linux device the size of a card deck (Figure 1-4b) processes all of this data locally. Manipulating high bandwidth data on an embedded system instead of a desktop or server requires very efficient data structures and algorithms. The processor reads thousands of samples per second off the sensor, transforms the magnetic field measurements to currents, phase aligns these to the sensed voltage, uses the combination to compute real (P) and reactive (Q) power, and extracts the envelope of the first, third, fifth and seventh harmonic for both P and Q each line cycle. This data is then time stamped by microsecond and stored in a custom database optimized for high bandwidth time series [8,9]. As implemented, this entire signal processing chain consumes less than 10% of the embedded system's resources.



(a) non-contact flex sensor



(b) single board computer

Figure 1-4: Hardware renderings of the Wattsworth power meter system.

1.3 Distributed Cloud Architecture

Retrieving actionable data from these embedded sensors presents a different challenge. Without a means to communicate, non-intrusive sensors, however sophisticated, would not provide a practical solution to any problem. This thesis introduces a cloud architecture that connects end users with remote sensors. In typical usage “cloud” describes a central server hosting content that is consumed by remote clients. When the content is located on the sensors themselves, the “cloud” acts as an intermediary instead of a repository. Figure 1-5 illustrates this distributed architecture. Sensors and users authenticate to a trusted server that authorizes transactions between the two. Encrypted tunnels between the sensor and user ensure that data remains confidential. This is in stark contrast to current cloud models where users have little to no control over where or how their data is used. In addition to enforcing authorization policies, the servers also provide powerful tools for users to visualize and manipulate sensor data. A powerful plotting interface supports data visualization. Embedded processors at each sensor cache iteratively decimated copies of the raw data.

These decimated versions are used to construct plots matching the resolution of the client’s screen. Devices return higher resolution data as the requested time range decreases. Dynamic pan and zoom controls populate the interface with new data seamlessly, giving the impression that all of the data is available on the user’s local machine. The cloud servers coordinate this visualization between multiple sensors allowing users to simultaneously plot data that resides across multiple different physical locations. With this tool, the user can plot data from any of sensor over any timescale using minimal network resources. To deliver truly actionable information, sensor platforms must be customizable. Not only are the requirements of residential, commercial and industrial consumers quite diverse, they also change quickly. In centralized frameworks users are at the mercy of the service provider for data processing and analytics. By moving the data and processing tools out of the cloud and onto the sensor, this new distributed framework supports an unprecedented level of customiza-

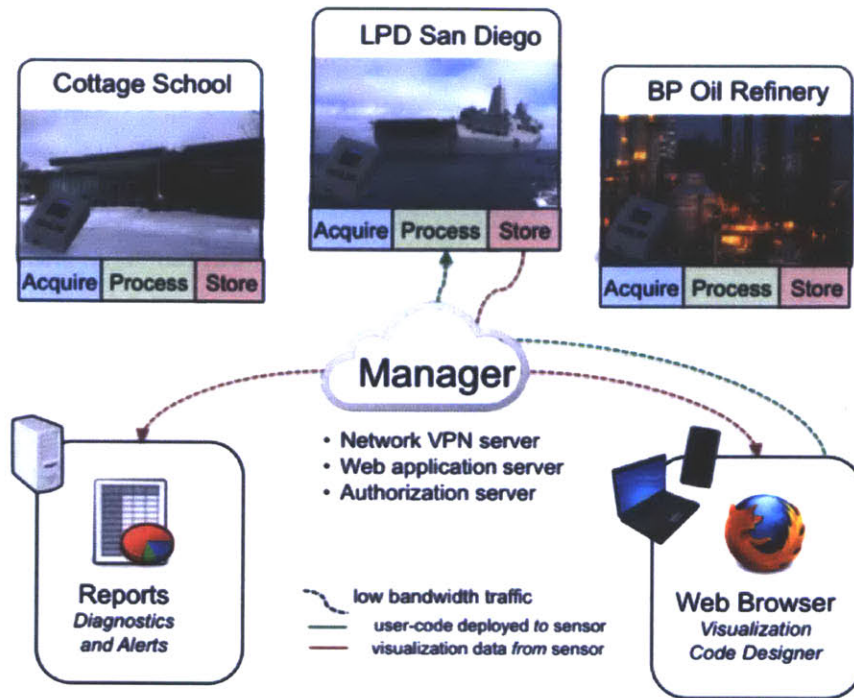


Figure 1-5: Distributed cloud architecture securely connects end users with remote Wattsworth platforms

tion. To realize the benefits of this new design approach I have developed a complete suite of development tools to support user-designed signal processing and data visualization on embedded sensors. This work has two primary components. First, an application programming interface (API) allows users to inject custom code or “apps” into the sensor’s data processing pipeline. User apps can create new data streams, generate graphic reports, alert based on particular conditions, and more. The second component is a web-based integrated development environment (IDE) where users can write, debug and share sensor apps enabling a new type of decentralized interaction where data is private but the code is collaborative [10].

Chapter 2

Non-Intrusive Sensors

2.1 Introduction

Advances in MEMs and integrated fabrication and nanotechnology have both increased sensor resolution and decreased sensor cost. This has led to a proliferation of commercial products that promise to make it easy to fully instrument a facility for any metric of interest. However the cost of the sensor itself is only a fraction of the total “cost of ownership”. The cost of installation and maintenance is often significantly greater than the hardware itself. This is because installation can involve equipment downtime and require skilled professional labor for installation. In many environments bringing equipment offline is prohibitively expensive or simply not possible, as in the case of Navy ships or Army bases.

In order to realize the benefits of this new technology, the sensor platform must be non-intrusive. That is, the sensor should retrofit or “design-in” to existing and new equipment, and installation should involve no skilled labor or equipment downtime. The sensors presented in this Chapter meet these requirements. They have been deployed in real world environments where traditional sensor platforms are at best inconvenient, and at worst infeasible.

Section 2.2 presents a non-contact current sensor that monitors current flow in an multi-conductor wire bundle by measuring the magnetic field. Section 2.3 discusses the design of an electric field sensor that can measure the voltage of an AC conductor

with no Ohmic contact. This work was done in collaboration with David Lawrence and also appears in [11]. Together these non-contact sensors provide a complete power monitoring solution. Section 2.4 presents two other non-intrusive sensor designs that have been developed to augment the non-contact power monitor in particular applications.

2.2 Non-Contact Current Sensor

One of the primary difficulties in non-contact power monitoring is designing a sensor capable of measuring current flow at a distance. Ampere's Law establishes the linear relationship between magnetic fields and current, but without a closed path around the conductor, accurately measuring this magnetic field is a challenging task. On the surface of a circuit breaker and the exterior of a power cable, the magnetic fields are not uniform or symmetric, and depending on the particular geometry, can be very small- less than 1 Gauss for bench top load currents in typical wires. Sections 2.2.1 and 2.2.2 introduce two circuit topologies that can accurately sense these small fields and can do so even in the presence of DC offsets introduced by nearby magnetic elements such as steel breaker panels, and the Earth itself.

The first circuit, based on a Hall Effect sensor, is a cost effective solution suitable for measuring larger loads or in situations where the wire topology exposes a relatively strong magnetic field. The second non-contact circuit uses a Tunneling Magnetoresistive (TMR) element (a recently introduced sensor technology [12]) with an inductive feedback technique to accurately measure extremely small fields.

2.2.1 Hall Effect Sensor

The schematic for this circuit is shown in Figure 2-1 and the fabricated prototype is shown in Figure 2-2. The Hall Effect is widely known and used in many current sensor designs. One of the most sensitive devices available in quantity is Allegro MicroSystem's A1362 Hall Effect sensor [13]. The A1362 has a programmable gain which can be set up to 16 mV/G, sufficient to resolve the magnetic fields around a

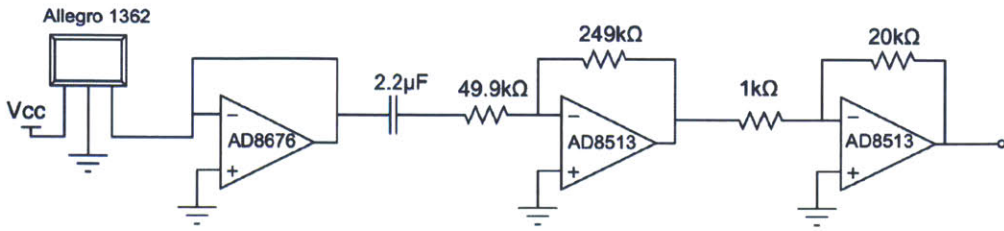
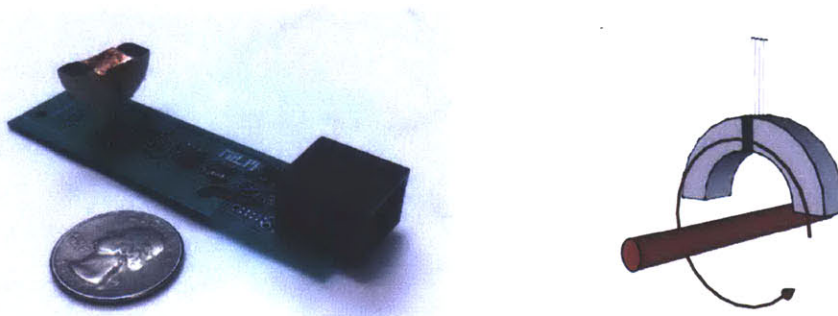


Figure 2-1: Schematic of Hall Effect non-contact sensor



(a) Fabricated prototype

(b) Magnetics for flux concentration

Figure 2-2: Hall Effect non-contact sensor prototype. Ferrite legs focus the field onto the sensor surface

standard power line. The quiescent output level is also programmable but not tightly controlled. Therefore, in order to measure small fields without saturating the output, we add a high pass filter with a cutoff at 1.5 Hz to AC-couple the sensor to the inverting amplifier gain stage. The large capacitive input of the filter stage requires a follower to buffer the sensor output. Overall gain can be adjusted by tuning the feedback leg of the gain stage.

The circuit is evaluated in the experimental setup shown in Figure 2-3a. A signal generator coupled with a power amplifier drives a solenoid at 200Hz to generate a magnetic field around the sensor. The circuit's output is compared to the field strength as measured with a fluxgate-magnetometer (an Aim Instrument I-prober 520). Results are shown in Figure 2-4. Two levels of field strength illustrate the degree of hysteresis in the sensor response. Steeper slope reflects higher sensitivity.

In situations where the geometry of the fields is approximately known, the response of the Hall Effect circuit can be improved by attaching magnetic material parallel to

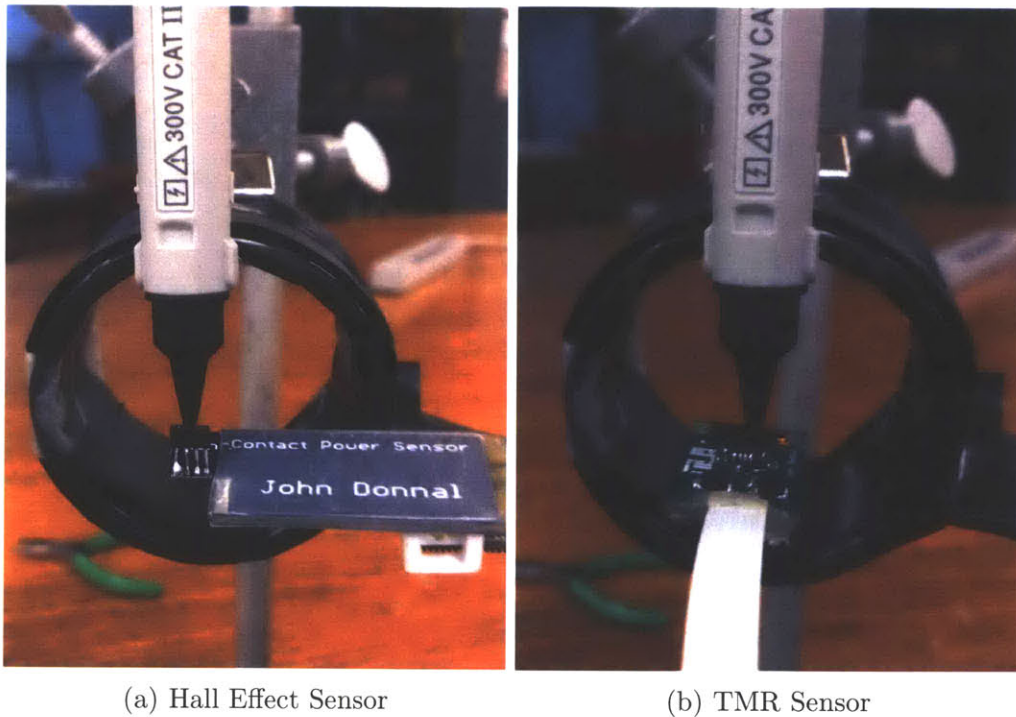


Figure 2-3: Non-contact current sensors are placed in an air core solenoid to evaluate their sensitivity and linearity to applied magnetic fields.

the field lines around the A1362 chip. The prototype in Figure 2-2 uses two ferrite segments to form a partial torus around the sensor package. This geometry captures radial fields near the surface of a multiconductor cable such residential and commercial power lines.

As Hall Effect technology continues to improve, new commercial sensors offer higher sensitivity at a similar cost and footprint. The non-contact sensor described in this section has been significantly improved by the substitution of the Allegro IC with the Melexis MLX91206 Triaxis IMC-Hall Current Sensor [14]. This sensor offers improved sensitivity- up to 40mV/G in a SOIC 8 package. At this sensitivity the sensor can be directly connected to an ADC without intermediate analog gain stages. See Appendix B.1 for a reference design.

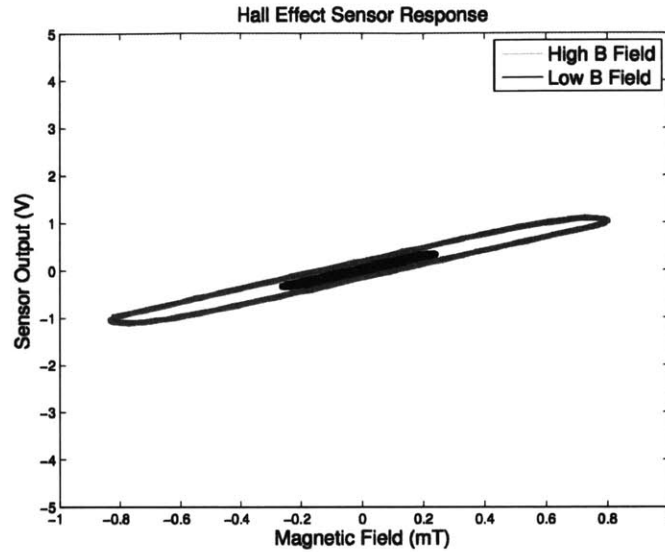


Figure 2-4: Response of Hall Effect-based sensor to applied fields

2.2.2 Tunneling Magnetoresistive Sensor

The TMR effect describes the change in resistance of a particular material due to applied magnetic fields. An explanation of the effect was first published in the 1970's but garnered little interest because practical implementations generated relatively small changes in material resistance [15]. Recent advancements using new materials and advanced fabrication techniques have improved the sensitivity of TMR devices. Modern state of the art sensors have a tunnel magnetoresistance ratio of over 600% at room temperature [16, 17]. Interest in these devices has increased as they have become integrated into high density magnetic disk drives and MRAM [18].

The STJ-340 is a TMR Wheatstone bridge sensor produced by MicroMagnetics. The sensor has four active TMR elements, arranged in a Wheatstone bridge architecture. Changes in the field induce an imbalance in the bridge which can be measured by a differential amplifier [12]. While the STJ-340 can detect very small fields (25mV/G as constructed), there are two significant challenges in using it as a current sensor. First, as with the Hall Effect-based sensor, DC offset errors quickly saturate the sensor output. The offset errors from the environment and from imbalance in the bridge itself (which can be up to 10%) must be removed before applying any significant gain

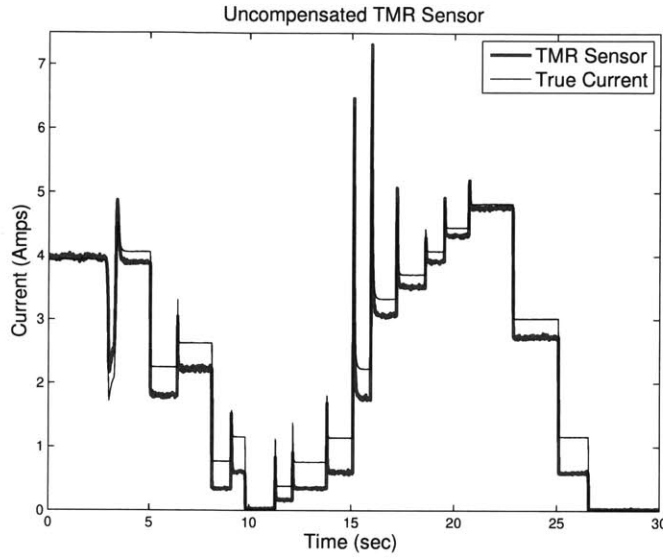


Figure 2-5: Non-linear response of an uncompensated TMR-based sensor. The sensor does not have a consistent response to a given change in current.

to the output. More troubling is that the TMR sensor's response to large changes in magnetic field is inconsistent and non-proportional; that is, there is no constant ratio between the change in the magnetic field and the resulting change in the sensor output. Figure 2-5 compares the true current as measured by a commercial current sensor (an LEM LA-55-P) to the output of an uncompensated TMR-based sensor. Even with proper amplification and DC offset removal, step changes in the load current produce non-linear responses in the sensor output.

The circuit shown in Figure 2-6 addresses both the DC offset and the non-linearity problems of the TMR sensor. The DC offset error is corrected by an integrator connected to the REF pin of the instrumentation amplifier. Any DC component is subtracted from the amplifier output resulting in a purely AC signal. This output is then fed through a high gain stage which drives an air core solenoid wrapped around the STJ-340. The current through this solenoid builds a magnetic field that opposes the applied field, creating a feedback loop that zeros the operating point of the STJ-340. Keeping the sensor element exposed to very small fields improves the sensor linearity and increases its range of operation. The current driven in the compensation solenoid is sensed as a voltage across a 150Ω resistor. The final stage

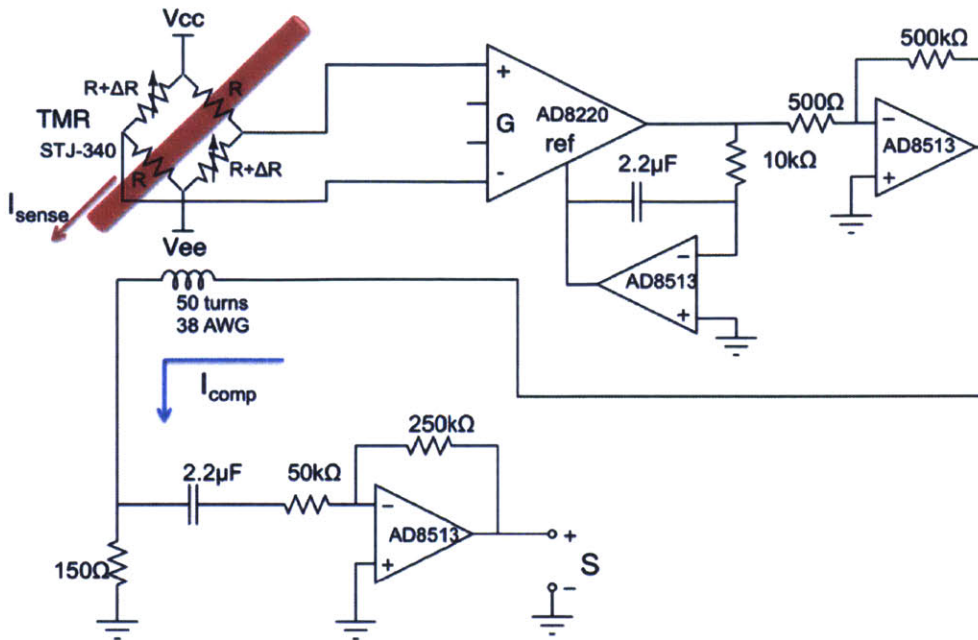


Figure 2-6: Schematic of the compensated TMR-based current sensor

is a high pass filter and gain stage that removes any offset not compensated for in the integrator.

The conceptual operation of the feedback topology is shown in Figure 2-7. In steady-state operation the sensed H_{src} and driven H_{comp} fields are approximately equal and the TMR element is exposed to only a very small residual field. The air-core solenoid has proven remarkably effective because closed-loop feedback is used to control the compensation coil.

The circuit is evaluated using the same procedure as the Hall Effect circuit. The

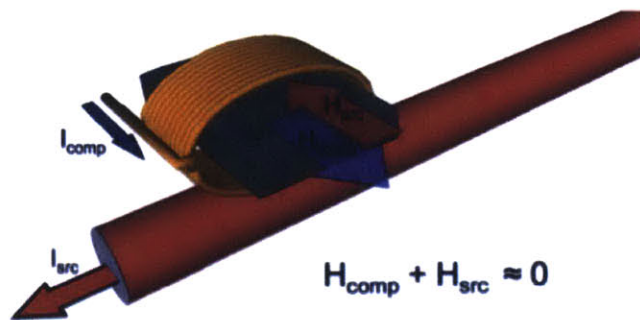


Figure 2-7: Illustration of TMR feedback technique

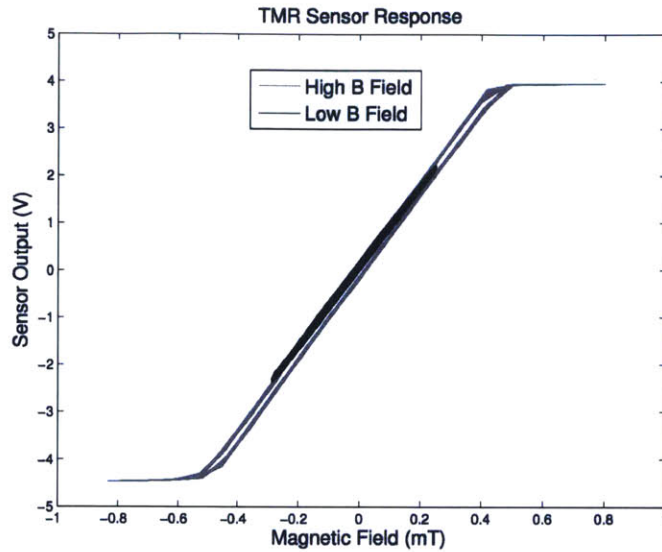


Figure 2-8: Response of compensated TMR sensor to applied fields

experimental setup is presented in Figure 2-3b. The results in Figure 2-8 show the high sensitivity (relative to the Hall Effect sensor in Figure 2-4) and linear response of the compensated TMR-based sensor. The saturation effects are due to the power rails and not the operation of the circuit itself. The hysteresis in the high field data series is also a result of the power rail limits. The feedback loop cannot drive sufficient current to eliminate the magnetic field at the sensor surface thus, exposing the TMR's inherent non-linearities.

The high sensitivity of the compensated TMR circuit makes it useful for applications outside of power monitoring. In particular the sensor can be used to measure water flow rate by retrofitting it onto a commercial utility water meter. The majority of utility water meters share a common core design. A solid brass enclosure with a positive displacement paddle wheel spins as water flows through the meter. As shown in Figure 2-9, the paddle wheel has a magnet on its axle that couples to a similar magnet on the billing hardware outside the brass enclosure. Usually the billing hardware is a simple mechanical accumulator. Using magnets instead of a mechanical coupling ensures the structural integrity of the pipe and reduces the chance for leaks.

The TMR sensor measures the small magnetic fields that escape the meter enclosure. The frequency of oscillation is proportional to the flow rate of the water.

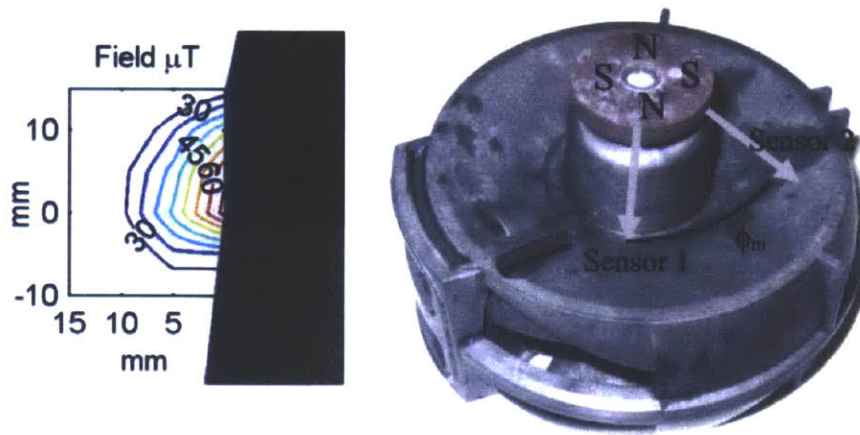


Figure 2-9: Utility water meter. Internal paddle wheel couples magnetically to billing hardware outside the watertight enclosure.

Unlike current signals which are fixed at the line frequency (50 or 60 Hz), the frequency of the field generated by the meter varies from zero up to the maximum pipe flow rate. The feedback design in the compensation circuit eliminates low frequency components of the magnetic field. By adding an additional output from the *ref* pin of the instrumentation amplifier, these low frequency components can be recombined with the sensor output in software. The adjusted circuit is shown in Figure 2-10

The signal processing for this application was developed by Chris Schantz and is presented in [4]. The complete prototype is shown in Figure 2-11. This design uses

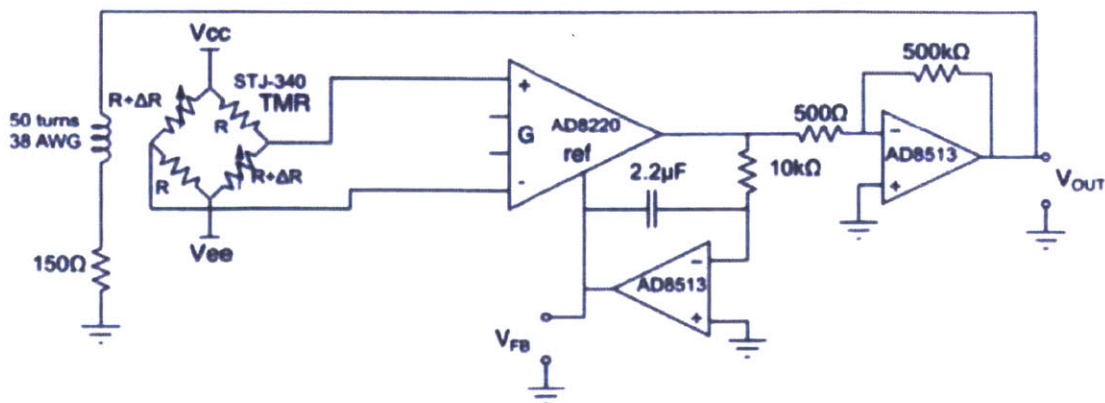


Figure 2-10: Compensated TMR schematic for water flow monitoring

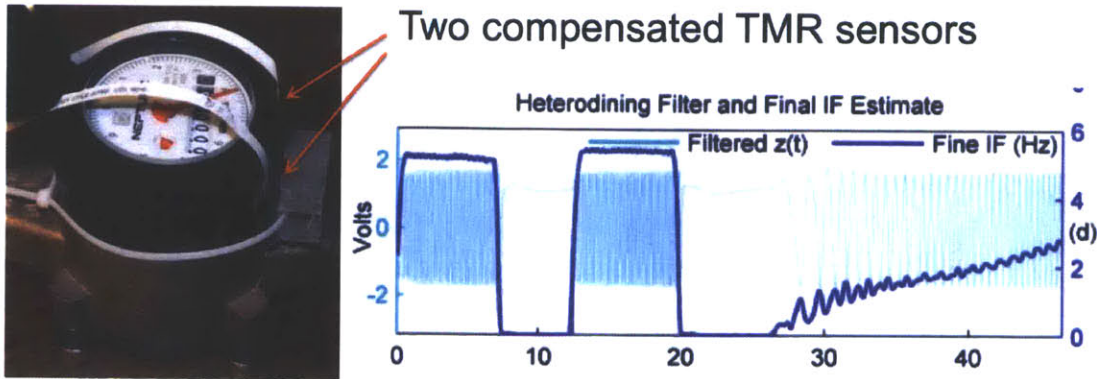


Figure 2-11: Retrofit flow meter using compensated TMR sensors

two TMR sensor elements to generate an analytic signal which allows for more robust frequency estimation and provides information about direction of flow as well as rate.

2.3 Non-Contact Voltage Sensor

Non-contact measurement of electric potential has proven useful for circumstances in which it is difficult to establish Ohmic contact with the conductors in question. Non-contact sensors offer ease of installation and robust high-voltage isolation in exchange for lower accuracy and increased susceptibility to external disturbances.

Non-contact measurement of static electric potentials was first proposed by [19] in 1928. A vibrating plate is placed near an unknown potential, forming a time-varying capacitance. The voltage of the vibrating plate is adjusted until the vibrations induce no current through the plate, indicating that the plate's potential is equal to the unknown potential. The bandwidth of the sensor is limited by the vibration frequency of the plate. Reference [20] proposes a method by which the residual current through the sensor plate is integrated to determine the higher frequency components of the unknown potential.

Recent work has focused on capacitive sensors that do not vibrate. The induced current is integrated to obtain the unknown potential at all frequencies of interest. Reference [21] uses a capacitor to perform the integration. References [22] and [23] are optimized for the geometry of high voltage transmission lines. However, the gain

of non-vibrating capacitive sensors is dependent upon the distance to the unknown conductor. Two sensor plates can be separately measured to compensate for this dependence [24]. Alternatively, large sensor plates can be placed close to a wire in order to enter a regime of operation in which the transfer function is not dependent on the separation distance [25].

The unique challenge of non-contact voltage sensing is *reconstructing the input signal while rejecting pickup from other sources*. Specifically, the currents induced by the input signal must be integrated in order to recover the input voltage. However, the currents induced by other sources have significant low-frequency components, which are amplified by an ideal integrator. There is a fundamental trade-off between the accuracy of voltage measurements and a sensor's signal-to-noise ratio.

This design is a non-contact sensor that takes a differential measurement of two vertically stacked non-vibrating sensor plates in order to *maximize* the dependence of gain on plate-to-wire distance, so that the signal from a nearby wire is selected and the signals from more distant wires are rejected. The sensor is especially well suited measurements that do not require the absolute scaling factor to be determined (e.g. total harmonic distortion and line regulation).

A three layer capacitor with a uni-polar analog circuit improves the design presented in [2] in both resolution and cost. The sensor uses a digital FIR filter optimized for measuring 60 Hz line harmonics with improved behavior across the range of worldwide utility frequencies relative to the previous analog design. The FIR filter has an exceptionally short impulse response for a digital integrator which improves the rejection of electric field impulses generated by voltage discontinuity in large inductive loads (eg when a motor is turned off). Cost reduction is achieved with careful component selection, uni-polar operation, and a board design that incorporates the capacitive pickups directly into the PCB layers rather than requiring copper foil applied as a post processing step.

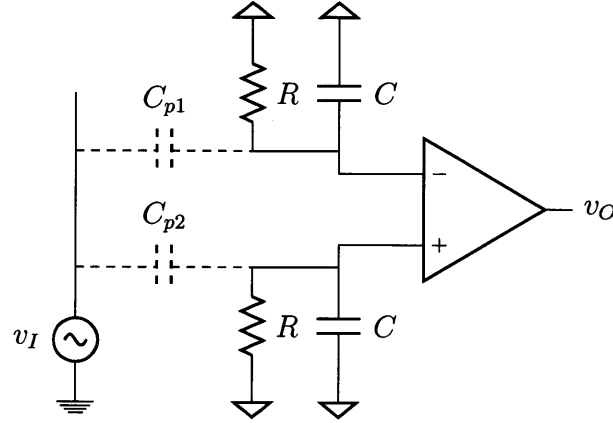


Figure 2-12: Circuit for differential non-contact sensing of an AC voltage. Equation (2.3) provides the transfer function of this circuit.

2.3.1 Principle of operation

A parasitic capacitance C_p develops between a sensor plate and a nearby wire (C_{p1} and C_{p2} in Figure 2-12). The sensor plate is attached to AC ground by a resistance R and a capacitance C . The transfer function from the wire voltage to the sensor plate voltage is given by

$$\frac{V_o(s)}{V_i(s)} = \frac{sRC_p}{sR(C + C_p) + 1}. \quad (2.1)$$

Conventional capacitive-divider sensors choose R to be very large. The transfer function is then approximated by

$$\frac{V_o(s)}{V_i(s)} \approx \frac{C_p}{C + C_p}.$$

If C is kept much smaller than C_p (which requires careful construction), the equation simplifies further to $V_o(s) \approx V_i(s)$. Unfortunately, this approach is not practical for the new sensor because C_p is tiny and the resistance required would impractically large. Instead, the new sensor operates in the regime where

$$|sR(C + C_p)| \ll 1$$

and so

$$\frac{V_o(s)}{V_i(s)} \approx sRC_p. \quad (2.2)$$

The sensitivity of the sensor is proportional to frequency. It is inversely proportional to the distance d between the wire and the sensor plate, because

$$C_p \propto \frac{1}{d}.$$

Note that the sensor measures the input signal v_I relative to its own ground, which must be connected (or at least AC coupled) to the input signal's ground.

As proposed in [2], improved localization is obtained by taking a differential measurement from two stacked sensor plates. This arrangement is shown in Fig. 2-12. Parasitic capacitance between the two plates is neglected from this model because in the differential mode it is equivalent to additional capacitance between each plate and ground.

The full transfer function of the differential sensor is given by

$$\frac{V_o(s)}{V_i(s)} = \frac{sR(C_{p2} - C_{p1})(sRC + 1)}{(sR(C + C_{p1}) + 1)(sR(C + C_{p2}) + 1)}. \quad (2.3)$$

For frequencies satisfying $|sRC| \ll 1$, the transfer function is approximated by

$$\frac{V_o(s)}{V_i(s)} \approx sR(C_{p2} - C_{p1}) \quad (2.4)$$

which is analogous to (2.2) for the single-plate sensor.

If the sensor plates are at a distance d from the wire and separated from each other by a distance $d_0 \ll d$, the differential capacitance is

$$C_{p2} - C_{p1} \propto \frac{1}{d} - \frac{1}{d + d_0} \approx \frac{d_0}{d^2}.$$

Therefore the sensitivity of the differential sensor is inversely proportional to the *square* of the distance between the wire and the sensor plates.

An alternative approximation aids in understanding the frequency-dependent be-

havior of the differential sensor. When $C_{p1} \ll C$ and $C_{p2} \ll C$, the transfer function is roughly

$$\frac{V_o(s)}{V_i(s)} \approx \frac{sR(C_{p2} - C_{p1})}{sRC + 1}. \quad (2.5)$$

The input voltage is recovered by integrating the output voltage—in other words, the zero at the origin is canceled by a new pole at the origin. At low frequencies, the remaining pole at $s = -1/RC$ has minimal effect. As the signal frequency increases, first order low-pass behavior will be observed.

Once the output is integrated, the differential capacitance $C_{p2} - C_{p1}$ must be determined in order to identify the sensor gain and recover the original input signal. If this capacitance is not known, the output will include an unknown constant scaling factor.

2.3.2 Analog implementation

There are two factors which determine the sensitivity and performance of the sensor: the geometry of the sensor plates, and the quality of the differential amplifier that is attached to them. Since the sensor should measure the voltage on one nearby wire without mixing in voltages from more distant wires, the sensor plates should not be made too large.

Based on the size of service entry cable and typical clearance constraints around existing wiring, the sensor plates are designed to have an area of 1 cm^2 . To minimize the cost of fabrication, the plates are built into the bottom two layers of a standard 1.6 mm four-layer printed circuit board (PCB). In a standard FR4 PCB, the bottom two layers are separated by 0.25 mm of laminate with a dielectric constant of approximately 4.5. Therefore the inter-plate capacitance is

$$C_{ip} = 4.5 \cdot \epsilon_0 \cdot \frac{1 \text{ cm}^2}{0.25 \text{ mm}} = 15.9 \text{ pF}.$$

The prepreg thickness can be adjusted to fine tune the inter-plate capacitance but in practice the default fabrication thickness senses line voltage well and is optimal in terms of production cost.

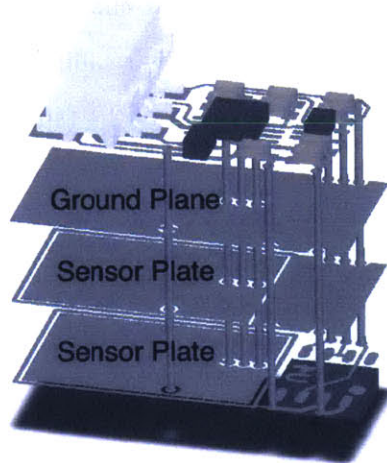
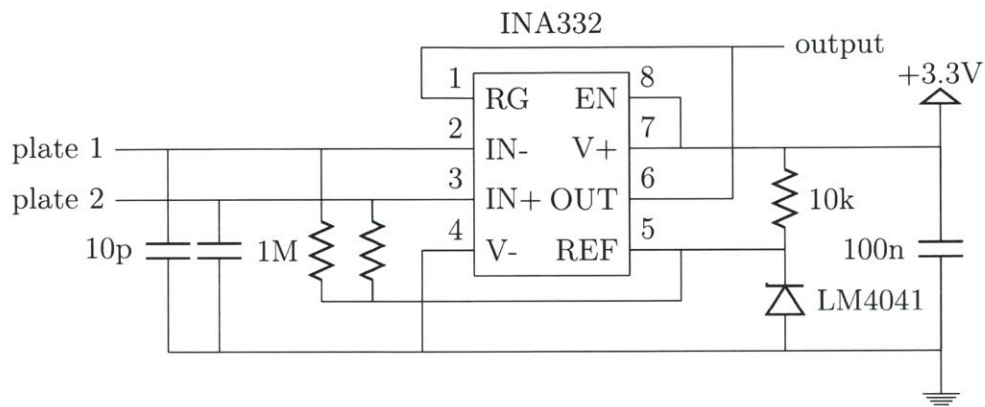


Figure 2-13: Non-contact voltage sensor schematic and 3D view of PCB stackup. The board dimensions are 1 cm by 2 cm. From top to bottom, layers contain: (1) connector, instrumentation amplifier, and supporting components, (2) ground plane, (3) negative differential sensor plate, (4) positive differential sensor plate and Hall effect IC.

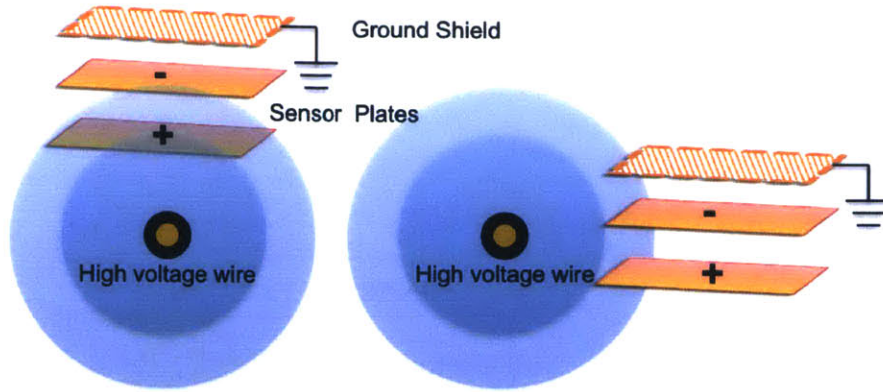


Figure 2-14: The sensor plate geometry selectively targets conductors directly below the sensor surface, shown on the left. The ground shield blocks fields originating above the sensor and the differential pickup rejects common mode fields originating on the sides of the sensor as illustrated on the right

With this information, the differential capacitance between the sensor plates and a nearby wire can be estimated. Suppose that the effective area of overlap between a wire and the sensor plates is 0.5 cm^2 , and the wire and the closer plate are separated by 1 mm of insulation with a dielectric constant of 2.1 (such as Teflon). The capacitance between the wire and the closer plate is

$$C_{p2} = 2.1 \cdot \epsilon_0 \cdot 0.5 \text{ cm}^2 \cdot 1 \text{ mm} = 0.930 \text{ pF}.$$

Then C_{p1} is given by the series combination of C_{p2} and C_{ip} , i.e.

$$\frac{1}{C_{p1}} = \frac{1}{C_{p2}} + \frac{1}{C_{ip}}$$

and the differential capacitance is

$$C_{p2} - C_{p1} = \frac{C_{p2}^2}{C_{ip} + C_{p2}} = 0.051 \text{ pF}.$$

The amplifier's input bias currents must be much smaller than the currents injected into the bias resistors by C_{p1} and C_{p2} . (This requirement is independent of the resistor values.) The limiting case is the lowest voltage of interest at the lowest frequency of interest—for design purposes, a 1 V signal at 60 Hz. The differential

current produced by this signal is

$$2\pi f(C_{p2} - C_{p1})V = 2\pi \cdot 60 \text{ Hz} \cdot 0.051 \text{ pF} \cdot 1 \text{ V} = 19 \text{ pA}.$$

To avoid distorting the signal, the amplifier's input bias currents should not exceed about 1 pA. The Texas Instruments INA332, meets this specification and is significantly less expensive than the AD8421 used in [2].

In the differential mode, the inter-plate capacitance of C_{ip} is equivalent to a capacitance between each plate and ground of

$$2C_{ip} = 31.8 \text{ pF}.$$

This capacitance reduces the bandwidth of the sensor and should be kept as small as possible. However, the amplifier is susceptible to common-mode disturbances which cause its inputs to exceed their allowable voltage range. In order to have some capacitive filtering of common mode inputs, an additional capacitance of 10 pF is provided between each sensor plate and ground. This gives a total differential mode plate-to-ground capacitance of

$$C = 41.8 \text{ pF}.$$

The last design task is to select the bias resistors attached to the sensor plates. The sensor gain is given by sRC_d , so to maximize sensitivity R should be as large as possible. However, larger values of R increase the time constant RC and decrease the sensor bandwidth. A good balance between these requirements is achieved by $R = 1 \text{ M}\Omega$. The breakpoint of the input network is placed at

$$\frac{1}{2\pi RC} = 3.81 \text{ kHz}$$

which is significantly faster than the signals of interest, but the sensor gain remains large enough to obtain usable voltage signals out of the amplifier.

Using (2.5), the transfer function of the specified analog sensor is

$$\frac{V_o(s)}{V_i(s)} \approx \frac{s \cdot 51 \text{ ns}}{s \cdot 42 \mu\text{s} + 1}. \quad (2.6)$$

For sufficiently low frequencies, (2.4) applies and

$$\frac{V_o(s)}{V_i(s)} \approx s \cdot 51 \text{ ns}.$$

The final analog sensor schematic is given in Fig. 2-13 and the PCB is depicted in Fig. 2-14. Because of the high impedances present on the PCB, special care must be taken to include guard traces around sensitive nodes and to clean conductive residue from the board after assembly. (Because the new voltage sensor is intended for non-contact power metering applications, this PCB also includes footprints for a Hall effect-based magnetic field sensor, an EEPROM, and a connector for cabled attachment to a microcontroller.)

2.3.3 Digital signal processing

The sensor output must be integrated to recover the original voltage being measured. Past implementations have used an analog integrating filter [2], but better performance is possible by performing the integration digitally. The design of the integrating filter presents a fundamental tradeoff between accuracy and disturbance rejection. Specifically, there are three design requirements:

1. The filter must faithfully reconstruct the voltage being measured.
2. The filter must reject low frequency disturbances, such as those caused by thermal drift.
3. The filter must recover quickly from impulsive disturbances.

These requirements correspond to the following three properties of a linear filter:

1. The filter should act as an integrator at line voltage and its harmonics. That is, the frequency response should be inversely proportional to the frequency, and

introduce 90 degrees of phase lag, for every frequency present in the voltage being measured.

2. The filter's frequency response should roll off quickly below the frequencies of interest.
3. The filter's impulse response should be short.

These goals have previously been realized by a cascade of two analog filters: a high-pass filter which admits the signals of interest but blocks low frequency disturbances, followed by an integrator to recover the original voltage signal. The challenge is that a causal analog filter cannot have a sharp transition between its stop band and pass band without introducing significant phase distortion—but if the transition to the stop band is gradual, low frequency disturbances will be admitted and amplified by the integrator.

Throughout this section, ω refers to a normalized angular frequency with units of radians per sample. Suppose that there are $2N$ samples per line cycle, so that the frequency of the n th harmonic is $\pi n/N$ radians per sample. The frequency response of an ideal integrating filter is given by

$$H_i(\omega) = \frac{\pi}{j\omega N}. \quad (2.7)$$

(This filter is “ideal” only in that it integrates signals perfectly and has a unit magnitude response at line frequency. It does not satisfy the second and third filter requirements.)

If the sampled line frequency of π/N radians per sample corresponds to 60 Hz in continuous time, the frequency response of the analog filter in [2] is given by

$$H_a(\omega) = \frac{j\omega\pi/N}{(j\omega + 1/\tau_0)(j\omega + 1/\tau_1)} \quad (2.8)$$

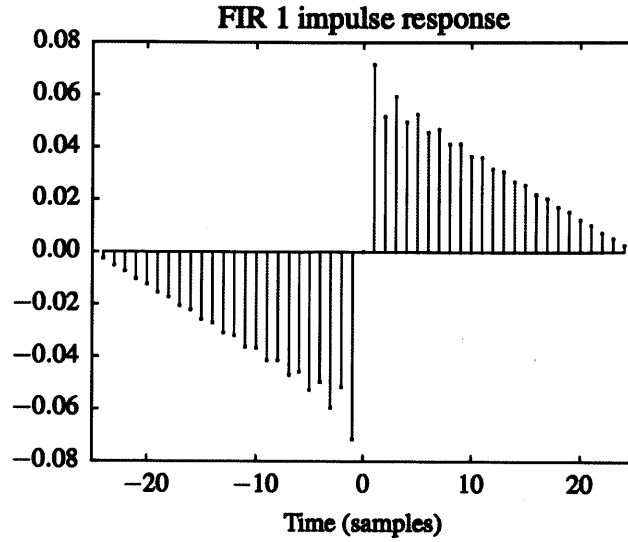


Figure 2-15: Impulse response $h_1[t]$ for $N = 25$. The impulse response is zero when $|t| \geq 25$.

with

$$\tau_0 = (2.2 \mu\text{F}) \cdot (12.1 \text{ k}\Omega) \cdot (60 \text{ Hz}) \cdot 2N$$

$$\tau_1 = (2.2 \mu\text{F}) \cdot (47 \text{ k}\Omega) \cdot (60 \text{ Hz}) \cdot 2N.$$

This analog filter is compared with two digital finite impulse response (FIR) filters. The FIR filters have antisymmetric impulse responses (such filters are known as “Type 3” FIR filters). As a consequence, they have zero group delay, introduce 90 degrees of phase lag at all frequencies, and do not pass signals at zero frequency or at the Nyquist rate.

The first FIR filter is the Type 3 filter with $2N - 1$ taps whose frequency response H_1 satisfies

$$H_1\left(\frac{\pi n}{N}\right) = \frac{1}{jn} \quad n \in \mathbb{Z}, 1 \leq |n| < N.$$

The second FIR filter is the Type 3 filter with $4N - 1$ taps whose frequency response H_2 satisfies

$$H_2\left(\frac{\pi n}{2N}\right) = \frac{2c_n}{jn} \quad n \in \mathbb{Z}, 1 \leq |n| < 2N$$

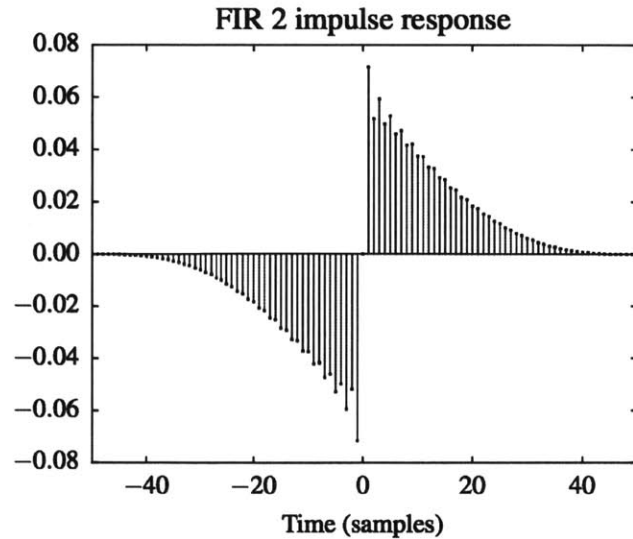


Figure 2-16: Impulse response $h_2[t]$ for $N = 25$. The impulse response is zero when $|t| \geq 50$.

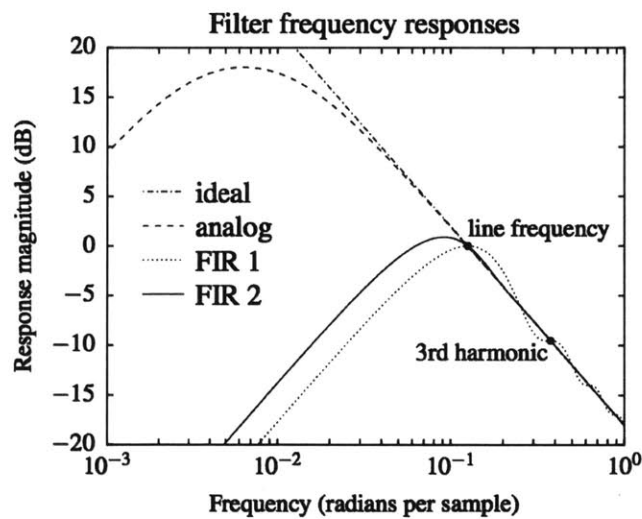


Figure 2-17: Magnitude behavior of the filters for $N = 25$. Note the logarithmic horizontal scale. The analog filter introduces phase distortion which is not depicted on this plot. Amplification of low frequency disturbances is roughly proportional to the area under the left half of the response curve.

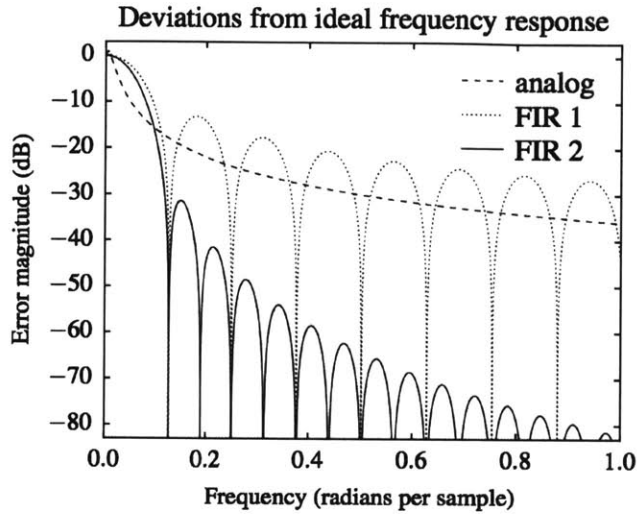


Figure 2-18: Magnitudes of the relative deviations from the ideal frequency response for $N = 25$. Both of the FIR filters have zero error at line frequency and its harmonics. Deviation from the ideal response is necessary and desirable at frequencies below line frequency.

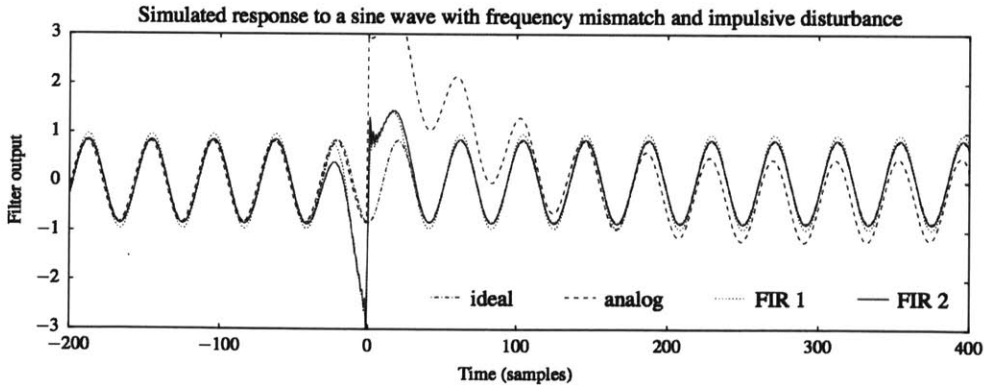


Figure 2-19: Simulated response of the filters to an impulsive disturbance with magnitude 30 at $t = 0$. The disturbance affects FIR 1 for $-25 < t < 25$ and FIR 2 for $-50 < t < 50$, but the analog filter has not yet recovered from the disturbance at $t = 400$. The filters are designed for a line frequency of 50 Hz with $N = 25$, but the input signal is provided at 60 Hz to demonstrate that the filters perform well even when line frequency is not known in advance.

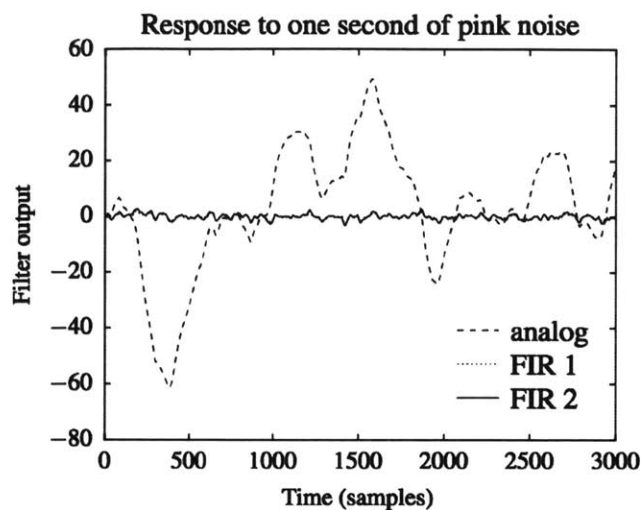


Figure 2-20: Simulated response of each filter to the same sequence of pink noise. The pink noise was generated as the cumulative sum of a sequence of numbers chosen uniformly at random between -0.5 and 0.5 .

with

$$c_n = \begin{cases} 1/2 & |n| = 1 \\ 1 & 2 \leq |n| < 2N - 1 \\ 3/4 & |n| = 2N - 1. \end{cases}$$

The filter impulse responses are computed using the inverse discrete Fourier transform:

$$h_1[t] = \frac{1}{N} \sum_{n=1}^{N-1} \left(\frac{1}{n} \cdot \sin \left(\frac{\pi n t}{N} \right) \right) \quad |t| < N \quad (2.9)$$

$$h_2[t] = \frac{1}{N} \sum_{n=1}^{2N-1} \left(\frac{c_n}{n} \cdot \sin \left(\frac{\pi n t}{N} \right) \right) \quad |t| < 2N \quad (2.10)$$

where t is an integer representing the discrete time. The impulse responses are plotted in Fig. 2-15 and Fig. 2-16.

By definition, $H_1(\omega) = H_2(\omega) = H_i(\omega)$ at line frequency and all of its harmonics below the Nyquist rate. Therefore these filters are optimal for the target application of sensing line voltage harmonics. h_1 is the shortest impulse response whose Fourier

transform has this property, and h_2 is designed to have a smoother frequency response at the expense of being twice as long as h_1 .

From the impulse responses, the discrete time Fourier transform gives the continuous frequency responses. The analytical expressions are omitted here because they provide no additional insight. Fig. 2-17 shows the magnitude response of each filter and Fig. 2-18 shows the relative magnitude of the difference between each filter's response and the ideal response. We consider the response of each filter to the signal

$$x[t] = \sin(\pi t/N \cdot 60/50) + 30\delta[t].$$

This represents the case where the digital filters were designed for a line frequency of 50 Hz but the actual frequency is 60 Hz, and an impulsive disturbance of magnitude 30 occurs at time $t = 0$.¹ These responses are plotted in Fig. 2-19 and exemplify the benefits and drawbacks of each type of filter.

Lastly, to illustrate the superior disturbance rejection of the digital filters, the output of each filter is computed for the same input sequence of pink (i.e. $1/f$) noise. The results are plotted in Fig. 2-20. Clearly, the analog filter exhibits a greater amount of error amplification.

Although the FIR filters are non-causal, both become causal when composed with a finite time delay. It is therefore possible to implement them, with the caveat that the output will not be known in real time. In particular, the first FIR filter delays its outputs by half of a line cycle and the second FIR filter delays its outputs by one full line cycle.

2.3.4 Experimental results

Three experiments are reviewed in this section to demonstrate the improved performance of the new voltage sensor. As the design is optimized for sensing line voltage,

¹Such disturbances often occur when a large inductive load is disconnected, resulting in a high instantaneous rate of voltage change on the inductor. This produces a powerful electric field which causes the sensor plate voltages to briefly exceed the common-mode input range of the amplifier. The INA332 drives its output to the positive rail when this condition occurs.

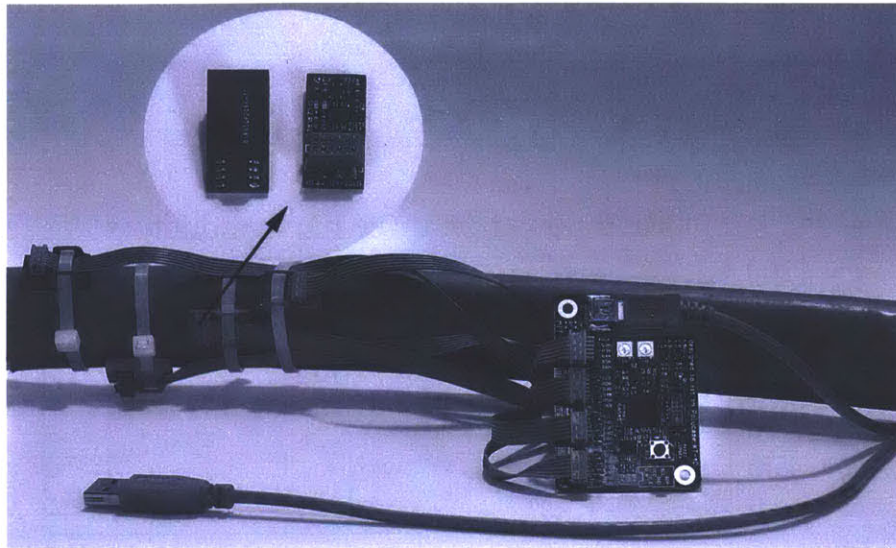


Figure 2-21: Non-contact power meter using the new voltage sensors installed on a service entrance cable for power metering.

the experiments use an additional set of non-contact current sensors to illustrate the performance of the sensor in a power monitoring application. In the first experiment, the sensor is used to extract harmonic envelopes of real power in a three phase cable bundle. The second experiment illustrates the improved performance of the FIR filter versus the previous analog filter across a wide range line frequency harmonics. Finally the third experiment shows the improved disturbance rejection of the FIR filter to variations in the electric field generated by inductive appliances.

The first experiment illustrates the utility of the voltage sensor in a power monitoring application. The voltage sensor was installed along with non-contact current sensors on a three phase bundle shown in Figure 2-21. A traditional power meter using commercial voltage sensors and current transformers was installed in parallel so that the results could be compared. Various electrical loads were switched on and off in order to obtain the time series data depicted in Figure 2-22. Mismatch between the traditional power meter and the non-contact power meter did not exceed 10 W over a dynamic range of 1000 W, showing that the new voltage sensor was able to accurately distinguish real and reactive power.

In order to obtain more detailed results showing the performance of the new

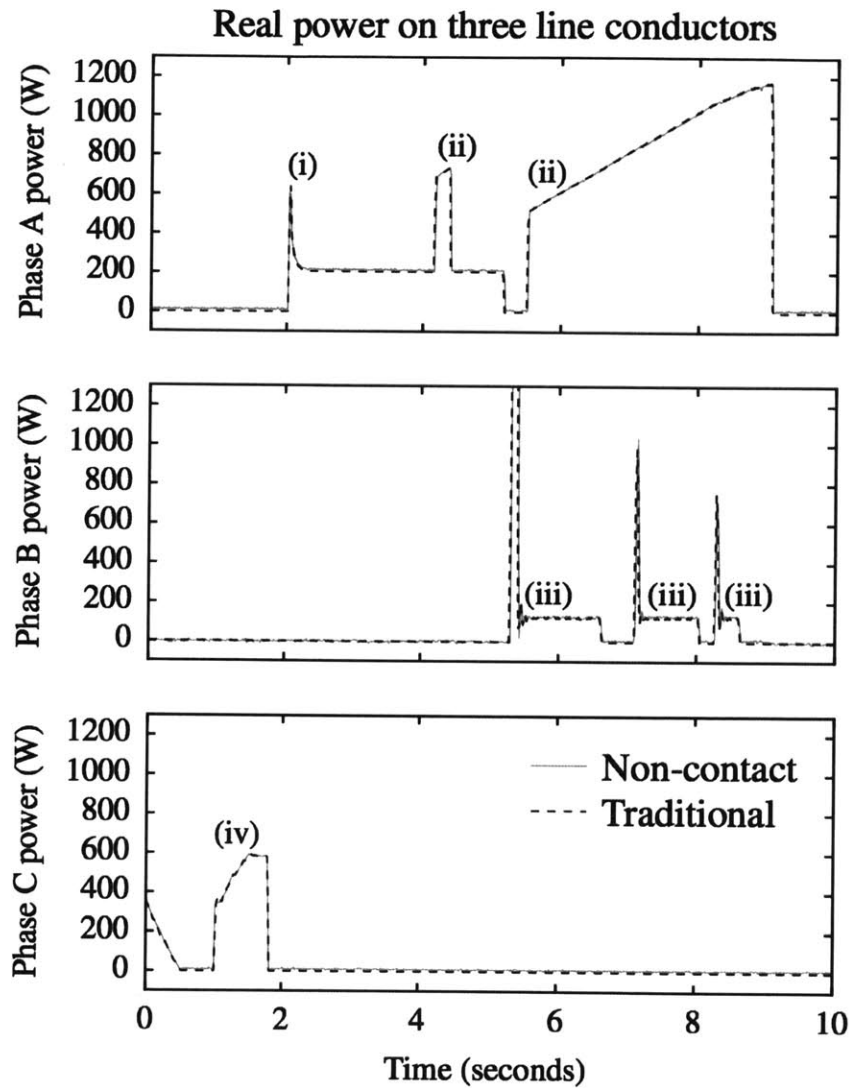


Figure 2-22: Data collected by non-contact and traditional power meters. The turn-on transients depicted are from (i) a 250 W incandescent light bulb, (ii) a 1500 W space heater, (iii) an 0.25 hp induction motor, and (iv) a 600 W bank of dimmable incandescent light bulbs.

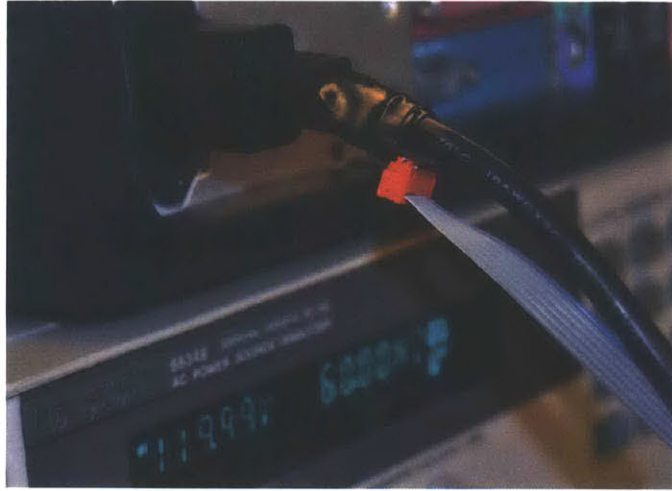


Figure 2-23: Non-contact voltage sensor attached to an 18-AWG computer power cable powered by an HP 6834B AC source. The ribbon cable leads to a printed circuit board which implements the analog filter and two digital FIR filters.

digital filters, the voltage sensor was attached to an 18-AWG computer power cable with line voltage supplied by an HP 6834B AC source. (This cable was chosen because thinner conductors produce the smallest coupling capacitance and therefore pose the most difficult sensing challenge.) This experimental setup is shown in Figure 2-23. The sensor was attached to an Atmel SAM4S microcontroller, which sampled the sensor with its built-in ADC at a sample rate of 3 kHz and a desktop Linux installation processed the signal using both of the FIR filters. The analog filter of [2] was constructed using a Texas Instruments OPA4376 operational amplifier and its output was connected to a second ADC channel. The output from all three filters was streamed from the microcontroller to a computer. With a line frequency of 60 Hz, there were 50 samples per line cycle and $N = 25$.

The output voltage from each filter was measured for sinusoidal inputs at various voltages and frequencies. Equation (2.4) was solved to find that the differential capacitance was 1.22 pF at 120 V and 60 Hz. At other voltages and frequencies, the percent magnitude error was computed for the output of each filter. The phase error of the analog filter relative to the (zero-phase) digital filters was also computed. This data is given in tables 2-1 and 2-2.

The collected data shows that the digital filters significantly outperform the analog

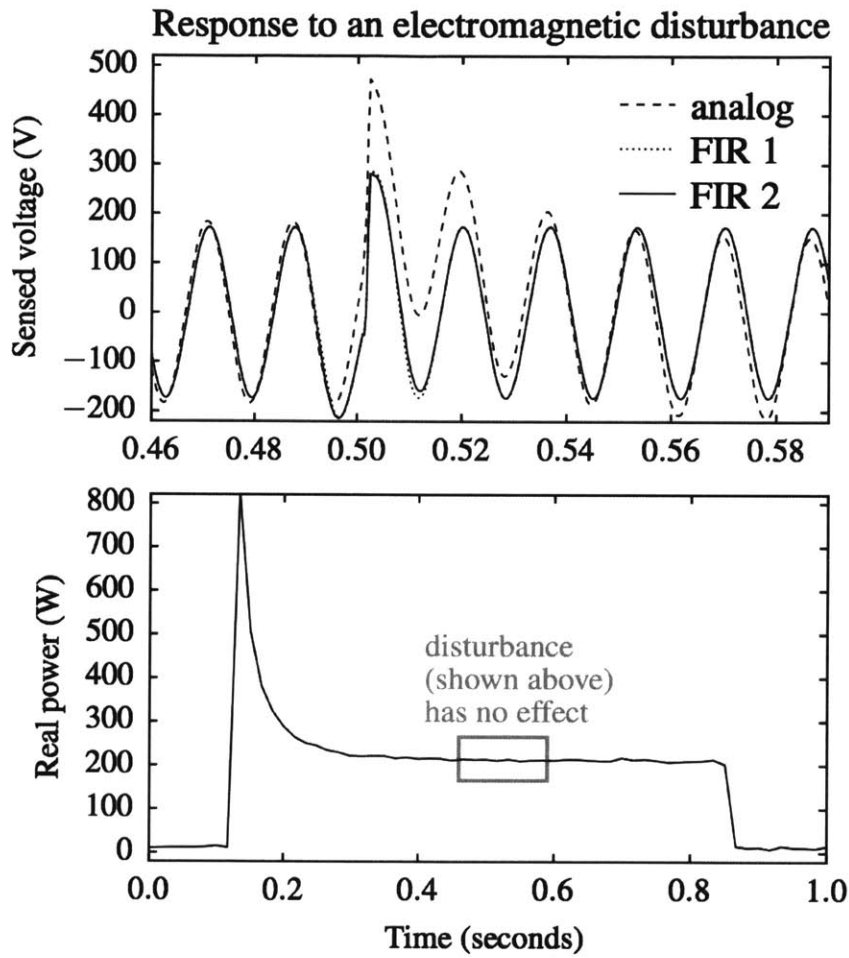


Figure 2-24: Response of the voltage sensor to a 100 mA fan motor being turned off 30 cm away from the sensor at $t \approx 0.5$. The digital filter recovers from the electromagnetic disturbance quickly, so *non-contact power metering is not affected by the disturbance*.

Input V RMS	Analog % error	FIR 1 % error	FIR 2 % error	Analog phase error, degrees
30	5.7	2.3	2.3	8.92
60	3.7	1.6	1.6	9.04
90	1.1	0.0	0.0	9.07
120	0.3	0.0	0.0	9.09
150	1.0	1.2	1.2	9.11
180	-0.5	0.8	0.8	9.16
210	-2.2	-0.0	-0.0	9.21
240	-3.3	-0.1	-0.1	9.29
270	-3.8	0.3	0.3	9.34
300	-4.5	0.5	0.5	9.40

Table 2-1: Output error from each filter for various input voltages at 60 Hz.

Input Hz	Analog % error	FIR 1 % error	FIR 2 % error	Analog phase error, degrees
60	5.7	2.3	2.3	8.92
120	2.9	-1.7	-1.7	4.98
180	2.5	-2.7	-2.7	3.69
240	1.9	-3.8	-3.8	3.06
300	1.6	-4.2	-4.2	2.69
360	0.8	-5.2	-5.2	2.48
420	0.1	-6.2	-6.2	2.33
480	-1.3	-7.6	-7.6	2.22
540	-2.8	-9.1	-9.1	2.16
600	-4.8	-11.2	-11.2	2.14

Table 2-2: Output error from each filter for various input frequencies at 30 V RMS.

filter with respect to phase lag and voltage linearity. As predicted by (2.5), all filters suffer from frequency-dependent gain, with a slightly more pronounced effect for the digital filters. Finally, the disturbance rejection of each filter was tested by turning off a 100 mA fan motor at a distance of 30 cm away from the sensor. (The motor does not have a clamp circuit, so an inductive voltage spike generates a strong electric field every time it is turned off.) The response of the three filters to this situation is shown in Fig. 2-24. There is good agreement with the simulated behavior in Fig. 2-19. The digital filters are only affected by the disturbance for one or two line cycles, but the analog filter has not recovered after many line cycles. Fig. 2-24 also shows that the digital filters prevent the disturbance from affecting power metering.

2.4 Ancillary Sensor Platforms

The previous sections have presented sensors designed for electric power monitoring. While a great deal of information can be extracted from power signals, some systems are better characterized by other physical metrics. Rotating machines (pumps, motors, generators) produce vibrations related to their mechanical health and mount condition. In many systems temperature indicates pending failures. A hot spot in a breaker panel indicates a loose or corroded connection. A hot gear or rotor in a transmission may indicate a bearing failure. In these situations augmenting electric power data with ancillary metrics can greatly improve the diagnostic capability of a sensor system.

Augmenting a non-intrusive platform must be done carefully. The total number of sensors must be minimal and the additional sensors must be easy to install without skilled labor or equipment downtime. The Captcha, developed in collaboration with Jim Paris is a vibration sensor designed to capture motor dynamics, and “Hottee” explores how multi-domain images can provide high resolution thermal information.



Figure 2-25: “Captcha” vibration diagnostic platform

2.4.1 Captcha: A Vibration Diagnostic Platform

One of the most interesting parameters to measure with large machinery is vibration. This is appealing because vibration naturally radiates through a structure allowing a sensor to be placed in an easily accessible location, and a large body of work supports the use of vibration monitoring as a diagnostic tool. This work was performed in close collaboration with the the US Navy and Coast Guard to develop pump vibration monitoring tools that enable underway diagnostics. Shipboard applications have a unique set of requirements. The sensors are installed on pumps before an underway cruise, left running for the duration of the cruise and then recovered. Therefore, the sensors must be rugged enough to withstand the harsh operating environment of a ship engine room, they must supply their own power, and they must be able to record large amounts of data. The “Captcha”, developed with Jim Paris, is a battery powered platform that uses an ADXL345 MEM’s accelerometer which records vibration measurements to a micro SD cards at 3.2 kHz. Such high bandwidth data transfer to an SD card is non-trivial when constrained to a low power micro controller

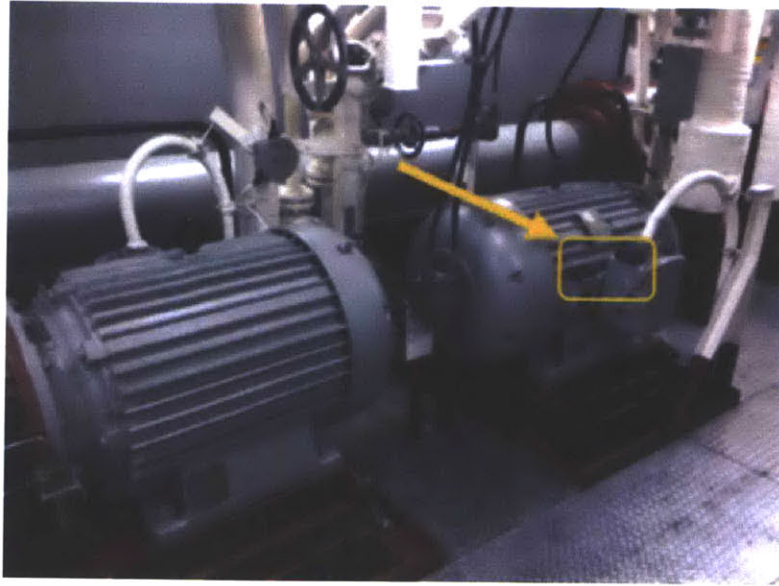


Figure 2-26: “Captcha” on USS SAN DIEGO (LPD 22)

and significant effort has gone into the firmware and hardware design of the board to enable this high data rate. The sensor and its batteries are sealed in a rugged plastic enclosure and adhesively mounted to the pump’s exterior housing. Figure 2-25 shows the first revision of the circuitry mounted in its plastic enclosure. Figure 2-26 shows the fully assembled sensor mounted to pumps onboard the Navy ships.

2.4.1.1 Hardware

The sensor board has several components that work together to provide the embedded data logging functionality. Figures 2-28a and 2-28b show the current revision of the hardware. Figure 2-27 lists the functionality of each area of the board. Power is provided by a 3.7 volt rechargeable Lithium-Ion battery which provides 3-Amp hours of charge. This is enough to run the data capture continuously for approximately 5 days. The charge management circuitry (B) recharges the battery over USB. A buck/boost converter (C) maintains the 3.3 volt power rail required by the digital circuitry. This allows the device to operate while recharging (when the charging circuitry raises the battery to over 4 volts) and also guarantees a stable voltage across the lifespan of the battery. The SD card has variable response times to write

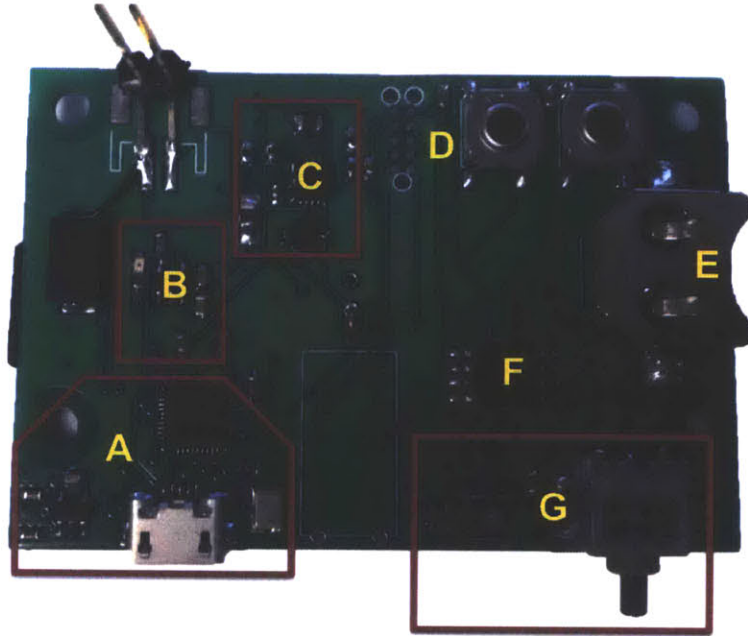
Annotation	Function
A	High speed USB management chip
B	Lithium-Ion battery charging
C	Power converter for 3.3V circuitry
D	ISP programming header
E	Coin cell backup for RTC
F	8MB of non-volatile FRAM
G	Panel button and status light
H	Micro SD Card (32 GB)
I	Real time clock (RTC)
J	AVR microcontroller (AT90USB)
K	ADXL345 3-axis accelerometer

Figure 2-27: Functional groups on vibration sensor circuit board

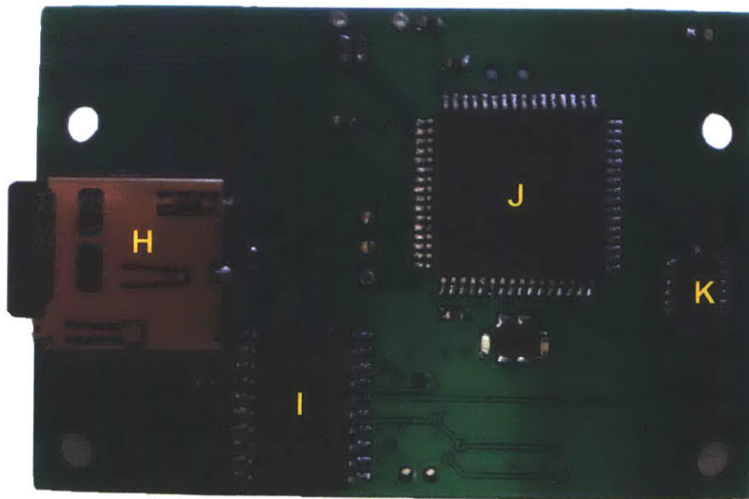
requests and periodically stalls for hundreds of milliseconds. In order to prevent data loss while the SD card is not accepting writes, extra data is stored on an 8MB FRAM chip (F). This memory can buffer several seconds of accelerometer data and flush out the data to the card when it is ready again. A real time clock with internal oscillator (I) provides accurate time stamps for all of the data. A coin cell backup (E) maintains the clock time even if the main battery is fully discharged or disconnected. The accelerometer (K) is placed between two mounting points to ensure a stable and accurate vibration measurement. An AT90USB1286 AVR micro controller (J) coordinates all of these parts and is programmed using a standard ISP header (D), although it can also run a boot loader which accepts programming commands over USB.

2.4.1.2 Firmware

In addition to recording vibration data to the SD card, the micro controller provides a terminal interface when connected over USB. This allows the user to view the status of the device, the number of files currently stored, and issue commands to start and stop data capture. The firmware also has a self test diagnostic feature that reports the status of all of the different components of the board. In addition to acting as a terminal emulation device, the sensor can be mounted as a high speed USB storage device if it is plugged into a computer while depressing the panel button (G). In this



(a) Top



(b) Bottom

Figure 2-28: Captcha circuit board

mode the USB management chip (A) connects the SD card (H) directly to the host computer which can then transfer data off the device just like a portable thumb drive.

2.4.1.3 Software

To improve the write speed and storage capacity of the device, all of the samples and timestamps are stored in a binary stream. A simple python script converts the binary data to standard ASCII text. The resulting file has one line per sample with 4 columns: time stamp, x-axis force, y-axis force, and z-axis force. This data file can be directly loaded into Matlab for further analysis.

2.4.1.4 Operational Verification

Vibration measurements are traditionally taken from accelerometers which are hard fixed to the machine under inspection. Some pumps have threaded mount points for specialized sensors but most do not and technicians affix the sensor to the machine with “super glue” or epoxy adhesives. The Captcha is designed to be easily installed and removed from a wide variety of pump equipment as non-intrusively as possible. In order to minimize the impact to the machine under test the Captcha enclosure is secured with adhesive tape pads instead of glue. In order to verify that using adhesives rather than epoxy or bolts does not distort the vibration signal, several different mounting techniques were tested in the lab. For a baseline an accelerometer was mounted directly to the body of a standard ventilation fan which was power cycled three times. A spectrogram of the collected data is shown in Figure 2-29. This was compared to the signal recorded by an adhesively mounted Captcha also on the fan body. The Captcha spectrogram is shown in Figure 2-30. Both spectrograms show the same general information with some attenuation present in the adhesively mounted device- which is to be expected. Given that the dominant vibration modes occur at frequencies on the order of motor rotation the adhesive mounts are suitable for this application. Indeed since the adhesive mounts act as a low pass filter, they can help in preventing aliasing of high frequency content into the sampled data.

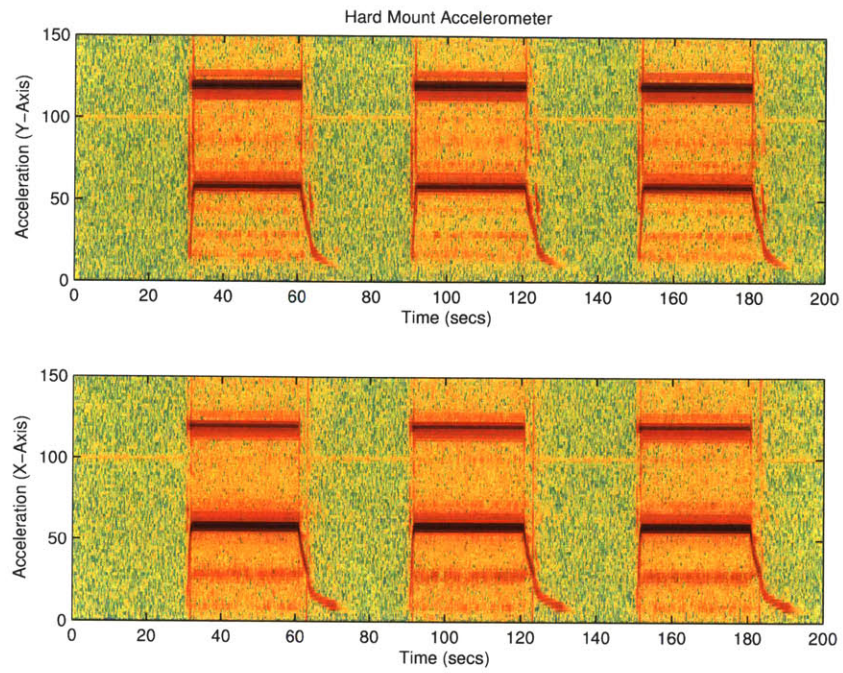


Figure 2-29: Accelerometer chip mounted with a pipe clamp

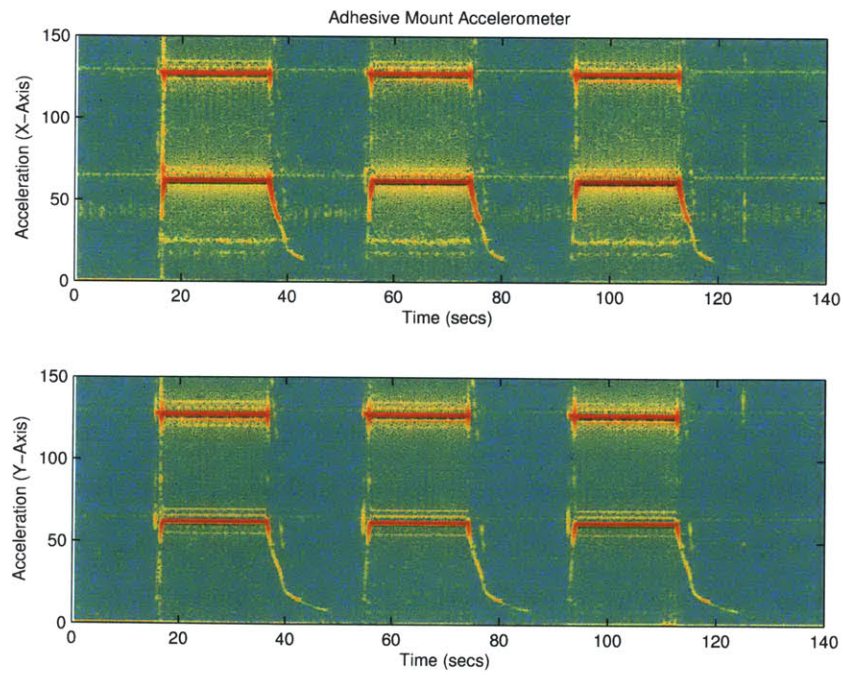
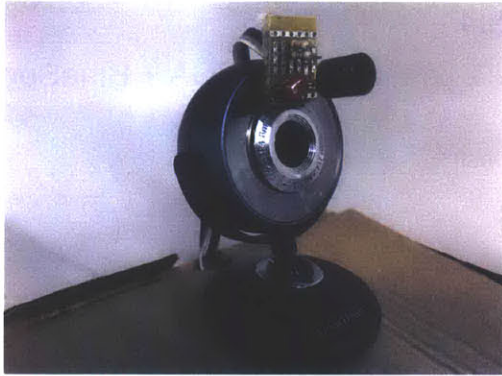
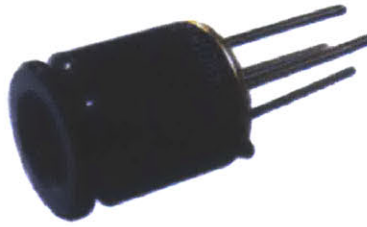


Figure 2-30: Custom sensor package adhesively mounted



(a) Combined video + IR



(b) MLX90620 64 pixel IR module

Figure 2-31: System output synthesizing thermal and RGB data sets in realtime video

2.4.2 Hottee: A multi-domain thermal imager

Thermal video records infrared energy and produces heat maps of the imaged objects. Thermal images of motors and electric power distribution equipment can help identify mechanical faults before they become dangerous failures. Traditional thermal cameras are prohibitively expensive but a new compact device manufactured by Melexis, the MLX90620, provides a resolution of 16x4 in a compact package that can be connected to an FPGA or traditional microcontroller with I2C. This device is a combination EEPROM and IR Array in a single TO-39 package. The low resolution limits the utility of the device as a standalone sensor but when the thermal data is superimposed against a visual spectrum video of much higher resolution the effective thermal resolution can be greatly increased. This is because the high resolution image can be used to infer the dominant source of temperature in a low resolution thermal pixel.

2.4.2.1 System Architecture

“Hottee” uses low resolution thermal video and high resolution visual spectrum video to provide realtime thermal diagnostics. The prototype in Figure 2-31 uses a USB webcam for visual video and an AVR microcontroller to interface with the MLX90620. The AVR enumerates as a USB device similar to the webcam. Processing high resolu-

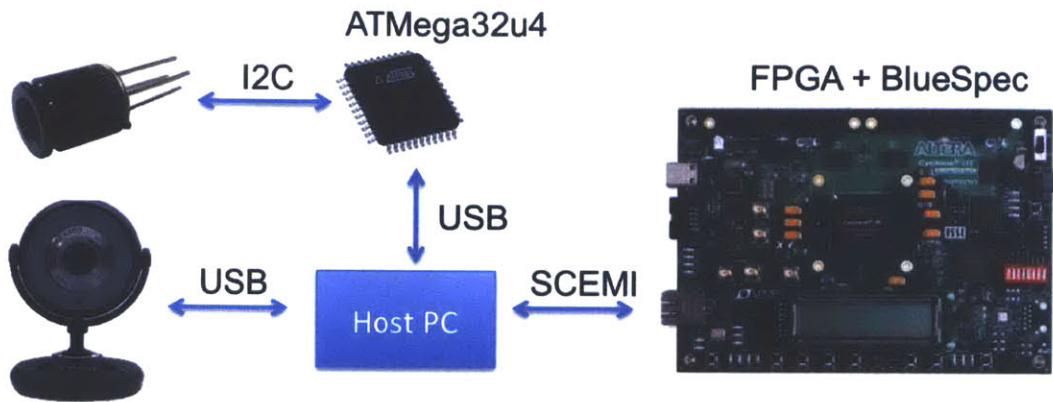


Figure 2-32: Thermal imager system architecture

tion video can be difficult in a microcontroller so the prototype design is implemented on an FPGA which is expected to perform the signal processing algorithms faster and with higher efficiency. To facilitate the production of the prototype the Bluespec hardware design language was used to program the FPGA. This imposed some limitations on the prototype architecture. The Bluespec interface does not support USB directly so a host computer is used to process the camera images and send them to the FPGA for processing over the Scemi buffer (a Bluespec bus architecture). This toolchain is illustrated in Figure 2-32

2.4.2.2 Signal Processing

The thermal sensor output is a nonlinear function of IR intensity and each pixel has separate scaling coefficients that are calibrated by the factory and stored in the sensor EEPROM. The pixels also have a dynamic offset relative to the die temperature so the sensor temperature itself must be calculated. These operations require significant computational resources but can be optimized to run in hardware using the Bluespec floating point modules.

The signal processing consists of three main steps:

1. Computation of die temperature
2. Per pixel compensation

3. Pixel temperature calculation

Once the die temperature is computed the pixel calculations are independent and can be run in parallel as hardware resources allow. In most situations the die temperature should change at a much lower rate than the object being imaged. In this case the computation is fully parallel for the duration the die temperature is unchanged (which is easy to detect by simply reading the PTAT register value).

Die Temperature The die temperature is calculated by (2.11) using a PTAT sensor on the camera module (V_{TH}). Coefficients K_{T1} and K_{T2} are stored factory calibrations stored in EEPROM.

$$T_a = \frac{-K_{T1} + \sqrt{K_{T1}^2 - 4K_{T2}[V_{TH} - \text{PTAT_data}]}}{2K_{T2}} \quad (2.11)$$

Pixel Compensation The pixel voltages are relative to the die temperature so to recover absolute temperature the voltages must be offset by T_a . The size of this offset varies by pixel. The pixel offset to temperature is calculated at the factory and stored in EEPROM as a pair of constants A_i and B_i per pixel (i, j) .

$$V_{IR(i,j)_OFF_COMP} = V_{IR(i,j)} - \left(A_{i(i,j)} + \frac{B_{i(i,j)}}{2^{B_{i_scale}}} (T_a - T_{a0}) \right) \quad (2.12)$$

Each pixel voltage is then compensated for thermal gradient correction again using coefficients stored in EEPROM.

$$V_{IR(i,j)_TGC_COMP} = V_{IR(i,j)_OFF_COMP} - \frac{\text{TGC}}{32} \times V_{IR_CP_OFF_COMP} \quad (2.13)$$

Additionally, if the emissivity ϵ of the material being imaged is known this can be included in the compensation calculation to improve the temperature accuracy.

$$V_{IR(i,j)_COMPENSATED} = \frac{V_{IR(i,j)_TGC_COMP}}{\epsilon} \quad (2.14)$$

Pixel Temperature The 64 compensated pixel voltages are then used to compute the per-pixel temperature. The voltage temperature relationship is nonlinear and

unique to each pixel. Each pixel is scaled by a factory calibrated coefficient α computed by (2.15).

$$\alpha_{(i,j)} = \frac{\alpha_0}{2^{\alpha_0_SCALE}} + \frac{\Delta\alpha_{(i,j)}}{2^{\Delta\alpha_SCALE}} \quad (2.15)$$

Combining the compensated pixel voltage, scaling coefficient, and die temperature, (2.16) provides temperature imaged by the pixel.

$$T_{O(i,j)} = \sqrt[4]{\frac{V_{IR(i,j)_COMPENSATED}}{\alpha_{(i,j)}} + (T_a + 273.15)^4} - 273.15 \quad (2.16)$$

2.4.2.3 FPGA Implementation

As suggested above the operations to calculate temperature are best done with floating point. The presence of powers of 4 and square roots make fixed point implementations very difficult and error prone. The FPGA is programmed using the Bluespec software suite which does not support floating point natively so a set of Verilog libraries provided by Xilinx are used instead. The AWB/Leap computing group in CSAIL has published a set of wrappers to expose the floating point modules to Bluespec [26]. One drawback of using raw Verilog is that designs can no longer be run in Bluesim (the Bluespec simulation environment). This makes working with floating point much more difficult because everything must be synthesized and run directly on the FPGA to check if it works. In order to preserve the ability to use Bluesim for other parts of the FPGA design there are two versions of the “Hottee” temperature calculation module. The first uses floating point modules to calculate the temperature, and the second fills the temperature vector with a set of dummy values and does no other math. The dummy module can be used for quick simulations and the real module is substituted in for synthesis. The Xilinx libraries are exposed as modules with a server/client interface. Modules receive requests with either one or two operands, and produce responses containing the result as well as the floating point status flags.

This design use the following floating point modules: addition, multiplication,

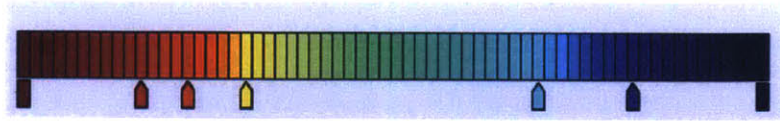


Figure 2-33: Thermal imager colormap used for video overlay

division, square root, integer to double precision conversion, and double precision to integer conversion. Constants are stored as reals and converted to floats statically using the helper function `$realtobits()`. One additional complication with the Xilinx libraries is that the Verilog files are only headers and cannot be synthesized. The actual functional code is in a set of `*.ngc` files which is done to protect the Xilinx IP. Unfortunately the standard build utility for Bluespec does not allow the inclusion of additional `*.ngc` files. so a custom build script had to be designed. The build script adds the necessary flags to the Xilinx tool chain to incorporate the libraries. This is wrapped in `custom_build.sh` and can be called just like the normal build command.

2.4.2.4 Image Reconstruction

The result from both the thermal and visual video pipelines must be combined to create a single image. The visual video pipeline has 3 channels for R, G, and B but the thermal pipeline only has the temperature values. To convert the temperatures to colors a color map lookup table is used. The color map based off the common Jet profile using Matlab's color map editor. This provides an intuitive interface to assign colors to values. The reds are placed around human body temperature and the blues around the freezing temperature of water since this covers the majority of the images we expect to encounter, although the sensor provides readings from -50 to 300 C. The color map is show in Figure 2-33.

The map has 70 entries and the temperatures in Fahrenheit are indexed into this table. Any temperature outside $[30, 100]$ is fixed to the respective max or min. The next step is to map each thermal pixel to the region of pixels it matches on the visual image. This is a set of static offsets and scale factors that are specified at compile time since the cameras are physically attached so the aspect does not change with time.



(a) Stubbed floating point for Bluesim (b) Full floating point implementation

Figure 2-34: System output synthesizing thermal and RGB data sets in realtime video of a cold water glass (left) and hot glue gun (right)

Finally the thermal image (now three-channel RGB) must be masked with the visual image. This is done with alpha compositing which is simplified from its general form with the assumption that the background image is opaque [27]:

$$\text{out}_{RGB} = \text{src}_{RGB} \times \text{src}_{\alpha} + \text{dst}_{RGB} \times (1 - \text{src}_{\alpha}) \quad (2.17)$$

The final combined image is sent back to the user via a SCEMI port. This is fully implemented in Bluespec and works on the FPGA.

Chapter 3

Signal Processing

3.1 Introduction

Nonintrusive sensing can impose an extra signal processing burden to extract and process useful information from an aggregate data stream. Recent advances in micro processors driven primarily by the smart phone market has resulted in powerful, energy efficient processors that can perform sophisticated signal processing in an embedded environment. By combining the non-intrusive sensor with an embedded processor programmed to run the signal processing algorithm, non-intrusive systems can serve as drop in replacement for conventional sensors.

Section 3.2 presents the signal processing algorithms required to convert data from the non-contact sensors in Sections 2.2 and 2.3 into current and voltage. This requires a calibration process but it can be performed on a completely live system by an end user without any specialized training.

Section 3.3 presents the signal processing algorithms that provide underway diagnostics for ships using the vibration diagnostic platform in Section 2.4.1 and a modified non-contact voltage sensor. This work was done in collaboration with Ryan Zachar and Pete Lindahl. The experimental setup is presented in detail in [28] and the algorithm and results are published in [29].

3.2 Non-Contact Power Measurements

Non-contact sensors provide measurements of magnetic and electric fields, not current and voltage. In a single conductor system the fields are linearly proportional to current and voltage making this conversion trivial, but in many systems of interest there are multiple conductors and multiple phases so that both the electric and magnetic fields are a superposition of several currents and voltages. In this case signal processing can be used to disaggregate these fields to provide the current and voltage data.

3.2.1 Multi-Conductor Power Systems

In many systems of interest there are multiple current-carrying conductors. If the magnetic fields of the conductors overlap, the output of any single non-contact sensor will be a combination of these fields, misrepresenting the current flowing in the nominal conductor of interest. Figure 3-1 compares the output of Hall Effect non-contact sensors to traditional LEM current sensors on two conductors in close proximity. Each non-contact sensor picks up significant interference from current in the neighboring conductor. This section introduces techniques to accurately measure individual currents with non-contact sensors in environments with complex, superposed magnetic fields.

3.2.1.1 Monitoring a Circuit Breaker Panel

Due to the close proximity of circuits on a breaker panel and the steel construction of the panel itself, the magnetic fields are often fully mixed so that any single sensor detects some portion of every current flowing through the panel or cable. Even if a precise location for minimal interference could be determined, the narrow dimensions of many breaker panels limit placement options as seen in Figure 3-2. Assuming the breaker currents are linearly independent, the number of sensors (X) must be equal to the number of breakers (Y) in order to calculate the currents in the panel. The x^{th} sensor output for such a system can be expressed as:

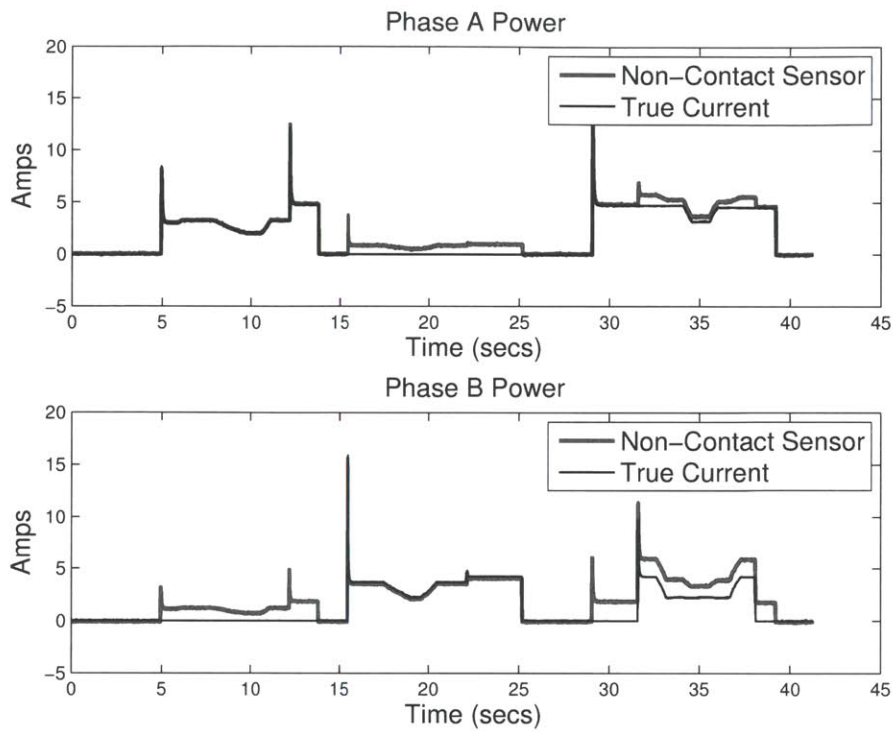


Figure 3-1: Comparing actual current to Hall Effect non-contact sensors on a multi-conductor cable. Interfering magnetic fields corrupt non-contact sensor measurements.



Figure 3-2: Monitoring a circuit breaker panel with TMR sensors

$$S_x = M_{x1}I_1 + M_{x2}I_2 + \dots + M_{xY}I_Y \quad (3.1)$$

Or, equivalently using the inverse relationship, the y^{th} breaker current can be expressed as:

$$I_y = K_{1y}S_1 + K_{2y}S_2 + \dots + K_{Xy}S_X \quad (3.2)$$

The full system can be expressed in matrix form where the current flowing in the breaker directly under each sensor is represented by the diagonal K values and the interference terms are the off-diagonal K 's.

$$\begin{bmatrix} I_1 \\ I_2 \\ I_3 \\ \vdots \end{bmatrix} = \begin{bmatrix} K_{11} & K_{12} & K_{13} & \dots \\ K_{21} & K_{22} & K_{23} & \dots \\ K_{31} & K_{32} & K_{33} & \dots \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix} \times \begin{bmatrix} S_1 \\ S_2 \\ S_3 \\ \vdots \end{bmatrix} \quad (3.3)$$

3.2.1.2 Cables with Neutral Return Path

The equations are slightly different for a multiple conductor power cables. These systems do not have fully independent conductors and are subject to the additional constraint of Kirchoff's Current Law (KCL):

$$I_1 + I_2 + I_3 + \dots + I_{neutral} = 0 \quad (3.4)$$

This equation reduces the dimension of the solution space. Standard power cables have only two current-carrying wires- hot and neutral. In this simple case only a single sensor is needed. The equations to find current are:

$$\begin{aligned} I_{hot} &= KS \\ I_{neutral} &= -I_{hot} \end{aligned} \quad (3.5)$$

The same technique can be extended for multiple phases and a common neutral.

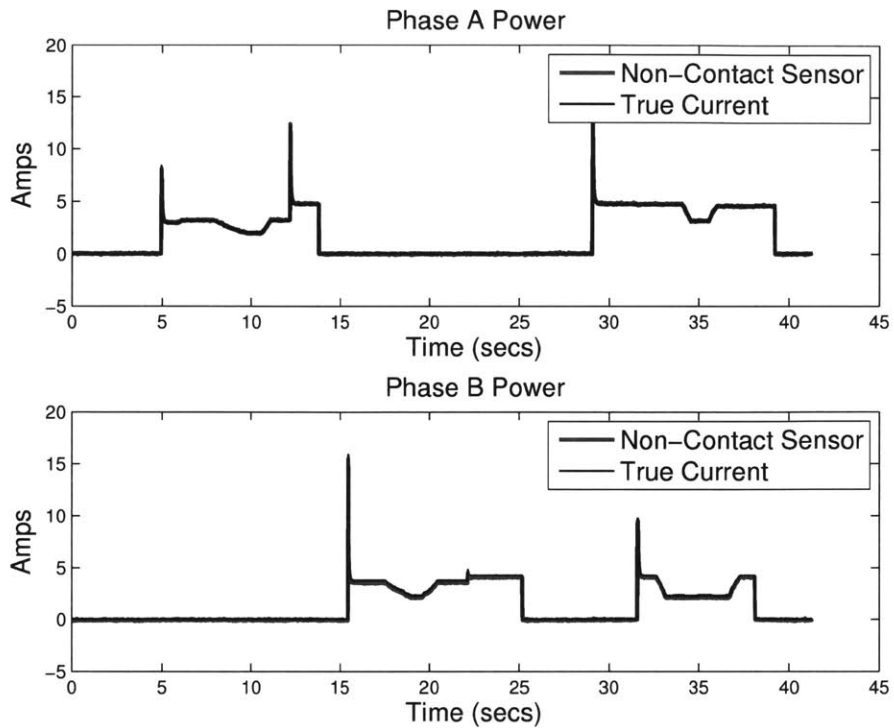


Figure 3-3: By applying the appropriate fit matrix interfering magnetic fields can be corrected to correctly measure line currents

For a three phase power cable, such as the one shown in Figure 3-5, there are four current carrying wires so the full matrix has 16 elements but KCL reduces the number of unknown currents by one. A nine element matrix using only three sensors is enough to determine all the currents. The equations for a three phase power cable are:

$$\begin{bmatrix} I_1 \\ I_2 \\ I_3 \end{bmatrix} = \begin{bmatrix} K_{11} & K_{12} & K_{13} \\ K_{21} & K_{22} & K_{23} \\ K_{31} & K_{32} & K_{33} \end{bmatrix} \times \begin{bmatrix} S_1 \\ S_2 \\ S_3 \end{bmatrix} \quad (3.6)$$

$$I_{neutral} = -(I_1 + I_2 + I_3)$$

3.2.1.3 Example Reconstruction

By applying the fit matrix $[K]$ for the waveforms in Figure 3-3 the non-contact sensors accurately measure the true current waveforms in each line.

3.2.2 System Calibration

Equations (3.3), (3.5) and (3.6) can calculate all currents of interest in complex systems, but they cannot be used until the K_{xy} terms in the fit matrix are determined. If only one current is present, the calibration matrix $[M]$ reduces to a set of equations relating the current to the output of a specific sensor (S_x):

$$\begin{bmatrix} S_1 \\ S_2 \\ S_3 \\ \vdots \end{bmatrix} = \begin{bmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ M_{31} & M_{32} & M_{33} & \dots \\ & \vdots & & \end{bmatrix} \times \begin{bmatrix} I_1 \\ 0 \\ 0 \\ \vdots \end{bmatrix} \quad (3.7)$$

$$\Downarrow$$

$$\begin{aligned} S_1 &= M_{11}I_1 \\ S_2 &= M_{21}I_1 \\ S_3 &= M_{31}I_1 \\ &\vdots \end{aligned}$$

Iterating with a known current on each conductor produces the full matrix $[M]$. The fit matrix can be found as

$$[K] = [M]^{-1} \quad (3.8)$$

While technically correct, this method places an undue burden on the user to first shut down all connected loads and then connect a single known load to each conductor in sequence. If the system of interest is a circuit breaker panel this type of calibration is unrealistic – a homeowner or facilities manager is unlikely to shut off the power and walk around in the dark connecting test loads. In environments with mission critical equipment, such as a microgrid on an Army Forward Operating Base (FOB), this type of calibration is impossible.

In order to calculate the elements of the $[M]$ matrix without interrupting service, a known current must be separated from the background environment. This can be done by applying pulse width modulation (PWM) to a calibration load to create an identifiable pattern in the current waveform. There are a variety of methods to design

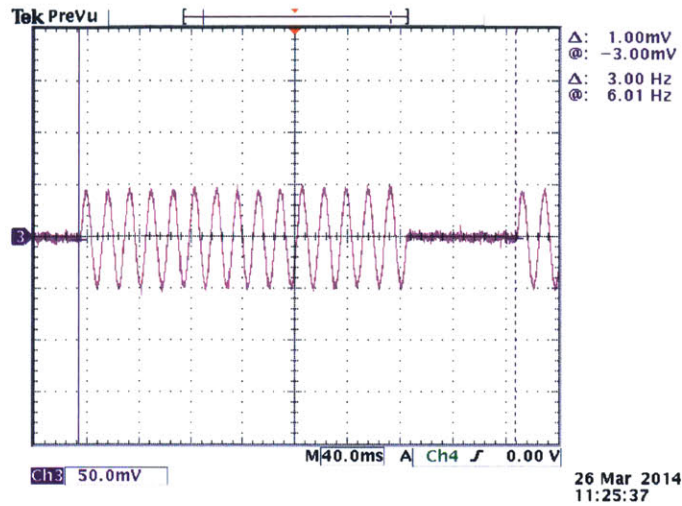


Figure 3-4: PWM calibration load as detected by a non-contact sensor

a PWM load. Our calibration load tracks the input voltage and draws power for 15 out of every 20 line cycles generating a 75% duty cycle. The full design of the calibration load is presented in [30]. On a 60 Hz service this corresponds to a PWM frequency of 3Hz as shown in Figure 3-4. Assuming there are no other significant loads cycling at 3Hz, the calibration load can be differentiated from the background environment using spectral analysis. A complete calibration procedure using this PWM load is developed first for a single phase system and then extended for multiphase systems.

3.2.2.1 Single Phase Systems

To determine the coefficients of matrix $[M]$ for a given system of conductors, the calibration load is introduced in turn to each of the conductors. In each case, the outputs of the non-contact voltage and currents sensors are fed to a preprocessing algorithm which calculates real and reactive current flow. The preprocessor, fully described in [9,31] uses the positive zero crossings of the voltage waveform to compute estimates of real (P) and reactive (Q) current each for line cycle. If all conductors are on the same phase (as in the case of a single phase breaker panel), then the zero crossings of the line voltage correspond exactly to the zero crossings of the non-contact voltage sensors. The calibration load is resistive, drawing purely real power, so only the P output of the preprocessor is used for the calibration procedure.

The preprocessor computes P and Q each line cycle and the calibrator PWM waveform is also defined by line cycles (rather than absolute frequency) which allows the same calibration procedure to be performed on both 50 and 60Hz services and is also robust against line frequency variation during calibration.

If the calibration load is operated in isolation, the real component of the preprocessor output is a line cycle time series that can be defined as follows:

$$P_{cal}[n] = \begin{cases} I_{cal}, & |n| \leq 7 \\ 0, & 7 < |n| \leq 10 \end{cases} \quad (3.9)$$

and

$$P_{cal}[n + 20] = P_{cal}[n]$$

where I_{cal} is the known current draw of the calibration load and n is the line cycle. There is a subtle caveat in the case where the non-contact sensor is 180 degrees out of phase with the true current. Since the preprocessor computes P and Q on positive zero crossings the edges of the P_{cal} pulse will be $0.5I_{cal}$ instead of I_{cal} . In practice this distortion contributes negligible error to the calibration process and can safely be ignored (see Figures 3-3 and 3-7).

In a live environment other loads draw arbitrary power throughout the calibration process. Therefore the real component of the preprocessor output for a sensor x is the combination of the calibration load on a conductor y plus an unknown amount of background load:

$$P_x[n] = M_{xy}(P_{cal_y}[n] + P_{bkgd}[n]) \quad (3.10)$$

where M_{xy} is the unknown scale factor representing non-contact sensor x 's response to the calibration load on conductor y and P_{bkgd} is the current drawn by other loads in the system. The goal of this analysis is to find the value of M_{xy} . These coefficients are used to form the matrix $[M]$.

First P_{bkgd} must be removed from the signal. At the harmonics of the calibration waveform, P_{bkgd} is 0 based on the assumption that the calibrator is the dominant load

at its PWM frequency. Using the Discrete Fourier Transform (DFT) defined as:

$$\hat{x}[k] = \frac{1}{N} \sum_{n=0}^{N-1} x[n] e^{-2\pi j \frac{kn}{N}} \quad (3.11)$$

and only considering k 's corresponding to harmonics of P_{cal} , the signal measured by the non-contact sensor can be represented in the frequency domain as:

$$\hat{P}_x[k] = M_{xy} \mathcal{F}\{P_{cal_y}[n]\} \quad (3.12)$$

Using a 200 point DFT and considering only the fundamental of the calibration waveform, Eq 3.12 becomes:

$$\hat{P}_x[10] = M_{xy} \times \hat{P}_{cal_y}[10] \quad (3.13)$$

$\hat{P}_{cal_y}[10]$ is a constant defined by the structure of the calibration waveform. The Fourier Series coefficients of a unit amplitude rectangular pulse with period T and width T_1 are [32]:

$$a_k = \begin{cases} \frac{\sin[(2\pi k/T)(T_1 + \frac{1}{2})]}{T \sin[2\pi k/2T]} & k \neq 0, \pm T, \pm 2T, \dots \\ \frac{2T_1 + 1}{T} & k = 0, \pm T, \pm 2T, \dots \end{cases} \quad (3.14)$$

$P_{cal_y}[10]$ corresponds to term a_1 . With the parameters of the load defined in Eq 3.9:

$$P_{cal_y}[10] = I_{cal} \frac{\sin(3\pi/4)}{20 \sin(\pi/20)} \equiv C_1 \quad (3.15)$$

where C_1 is introduced for notational convenience. Substituting Eq 3.15 into Eq 3.13 yields an equation for M_{xy} :

$$M_{xy} = \frac{|\hat{P}_x[10]|}{C_1} \quad (3.16)$$

The magnitude of $\hat{P}_x[10]$ is required in Eq 3.13 because the calibration waveform detected by the pickup is actually $P_{cal_y}[n + n_0]$ where n_0 is an uncontrolled time shift due to the fact that the calibration load is not time aligned with the sampling

interval. This time shift becomes a phase shift in the frequency domain [33] making $\hat{P}_x[10]$ complex:

$$\hat{P}_x[10] = M_{xy}C_1e^{-jk(2\pi/20)n_0} \quad (3.17)$$

$$= M_{xy}C_1e^{-j\Phi} \quad (3.18)$$

where $k = 1$ and Φ is an unknown phase shift. By using only the magnitude in Eq 3.16 this phase term is eliminated, but the sign of $\hat{P}_x[10]$ is eliminated as well.

Fortunately, the sign can be recovered by using higher harmonics of the calibration waveform. In a 200 point DFT the second calibration harmonic is present at $\hat{P}_x[20]$. Expressing the second harmonic in the same form as Eq 3.18 yields:

$$\hat{P}_x[20] = M_{xy}C_2e^{-j2\Phi} \quad (3.19)$$

where C_2 corresponds to $\hat{P}_{cal_y}[20]$ which, like C_1 , is a constant that can be determined using Eq 3.9 and Eq 3.14.

The compensated phase difference between $\hat{P}_x[10]$ and $\hat{P}_x[20]$ can be used to recover the sign of M_{xy} . We define the compensated phase difference between the fundamental and the k^{th} harmonic in a Fourier Series as:

$$\Delta_{ph}(k) \equiv k\angle a_1 - \angle a_k \quad (3.20)$$

In the case where M_{xy} is positive, the phases of these two terms are:

$$\angle \hat{P}_x[10] = -\Phi \quad (3.21)$$

$$\angle \hat{P}_x[20] = -2\Phi \quad (3.22)$$

The compensated phase difference between these terms is:

$$\begin{aligned} \Delta_{ph}(2) &= 2(-\Phi) - (-2\Phi) \\ &= 0 \end{aligned} \quad (3.23)$$

However if M_{xy} is negative, the phases of the same two terms are:

$$\angle \hat{P}_x[10] = \pi - \Phi \quad (3.24)$$

$$\angle \hat{P}_x[20] = \pi - 2\Phi \quad (3.25)$$

which results in a compensated phase difference of:

$$\begin{aligned} \Delta_{ph}(2) &= 2(\pi - \Phi) - (\pi - 2\Phi) \\ &= \pi \end{aligned} \quad (3.26)$$

Thus the final expression for M_{xy} incorporating both magnitude and sign is:

$$M_{xy} = \begin{cases} \frac{|\hat{P}_{nc}[10]|}{C_1}, & \Delta_{ph}(2) = 0 \\ -\frac{|\hat{P}_{nc}[10]|}{C_1}, & \Delta_{ph}(2) = \pi \end{cases} \quad (3.27)$$

where

$$\Delta_{ph}(2) = 2\angle \hat{P}_x[10] - \angle \hat{P}_x[20]$$

This analysis relies on the presence of even harmonics in P_{cal} . In the case of a symmetric waveform with no even harmonics, the compensated phase difference cannot be used to determine the sign of M_{xy} . To see why this is the case consider the first two non-zero terms of the Fourier Series for a symmetric waveform:

$$\text{positive: } a_1 e^{-j\Phi} + a_3 e^{-j3\Phi} \quad (3.28)$$

$$\text{negative: } -a_1 e^{-j\Phi} - a_3 e^{-j3\Phi} \quad (3.29)$$

The compensated phase difference between the fundamental and the third harmonic is:

$$\Delta_{ph}(3) = 3\angle a_1 - \angle a_3 \quad (3.30)$$

Substituting in the phases for each coefficient yields:

$$\text{positive: } 3(-\Phi) - (-3\Phi) = 0 \quad (3.31)$$

$$\text{negative: } 3(\pi - \Phi) - (\pi - 3\Phi) = 2\pi = 0 \quad (3.32)$$

The compensated phase difference is the same for both positive and negative waveforms. Intuitively this makes sense because the polarity of a symmetric waveform is ambiguous without a DC component (a_0). This is why the calibration load has a PWM duty cycle of 75% rather than 50%.

3.2.2.2 Multiphase Systems

The preprocessor requires accurate voltage phase information to calculate real (P) and reactive (Q) current. In single phase systems the electric field is always in phase ($\pm\pi$) with the line voltage regardless of the number of conductors. In multiphase systems this is not necessarily the case. In typical three phase systems the voltages of the conductors are mutually offset by 60 degrees resulting in a basis for the electric field that spans \mathbb{R}^2 . To accurately measure phase in these complex environments, a correction factor must be applied to the non-contact voltage sensor output.

Once the voltage waveforms for each phase are known, the calibration procedure to find the M_{xy} coefficients for the non-contact current sensors is identical to the single phase procedure described previously.

To understand the difficulty in voltage reconstruction, consider the output of a non-contact voltage sensor in a three phase system:

$$v_{nc}[n] = S_1v_1 + S_2v_2 + S_3v_3 \quad (3.33)$$

where S_x is sensitivity to the field produced by voltage v_x . The v_x terms can be expressed as complex sinusoids with the following form:

$$v_x = V_x e^{j\omega t + \Phi_n} \quad (3.34)$$

Because all of the v_x terms have the same frequency (ω), Eq 3.33 can be rewritten as:

$$v_{nc}[n] = \mathcal{R}\{e^{j\omega t}(S_1V_1e^{j\Phi_1} + S_2V_2e^{j\Phi_2} + S_3V_3e^{j\Phi_3})\} \quad (3.35)$$

The sum of $S_xV_xe^{j\Phi_x}$ terms can be represented as a single complex exponential:

$$v_{nc}[n] = \mathcal{R}\{Ae^{j\omega t}e^{j\phi}\} \quad (3.36)$$

where A is the amplitude of the sensor output and ϕ is the phase. Depending on the particular geometry of the system there may be degenerate nodes where the electric fields contributed by each phase sum to zero. In this case $A = 0$ and the sensor should be repositioned.

If the voltage amplitude V is a known constant for all phases, the output of a single non-contact sensor can be used to reconstruct the line voltages:

$$v_1[n] = v_{nc}[n]\left(\frac{V}{A}e^{j\psi_1}\right) \quad (3.37)$$

$$v_2[n] = v_{nc}[n]\left(\frac{V}{A}e^{j\psi_2}\right) \quad (3.38)$$

$$v_3[n] = v_{nc}[n]\left(\frac{V}{A}e^{j\psi_3}\right) \quad (3.39)$$

The only unknowns are the correction terms (ψ_x) which align the measured phase (ϕ) to a particular line phase.

The correction terms can be calculated to within $\pm\pi$ using the calibration load. The calibration load draws only real power so the output of the preprocessor at the PWM frequency should be all P and no Q. However, a misalignment between the line phase and the phase of the non-contact sensor will cause the preprocessor to compute a different ratio of P and Q. The correction factor ψ_n is the rotation required to produce all P and zero Q. This is simply the negative of the power factor angle calculated by the preprocessor:

$$\psi_x = -\tan^{-1}\left(\frac{Q_x}{P_x}\right) \quad (3.40)$$

Due to the interference of background loads, Equation 3.40 is only valid at the PWM frequency and its harmonics. Using a 200 point DFT to measure the fundamental frequency of P and Q gives the following equation for ψ_x :

$$\psi_x = \tan^{-1}\left(\frac{\hat{Q}_x[10]}{\hat{P}_x[10]}\right) \quad (3.41)$$

It is important to note that this procedure calculates ψ_x to within a factor of $\pm\pi$ which means the sign of the voltage waveform cannot be uniquely determined. However, this does not affect the accuracy of the preprocessor's calculation of real (P) and reactive (Q) power. Conceptually an offset of $\pm\pi$ in ψ_x is equivalent to a current sensor being flipped 180 degrees spatially. Both introduce the same apparent phase difference between sensed voltage and sensed current. Eq 3.28 associates a sign to each term of the fit matrix $[M]$ to correct for this difference. Therefore the computed real (P) and reactive (Q) power is always correct despite the ambiguity in ψ_x .

3.2.2.3 Rapid Calibration

To more efficiently compute the fit matrix in a multi-conductor system, calibration loads can be connected to each phase and run simultaneously. This is advantageous when multiple phases are available at a single point such as the 240V dryer outlets in residential environments and three phase outlets in industrial environments. Simultaneous calibration requires that each load toggle at a distinct frequency such as 0.5Hz, 3Hz, and 7Hz so that the Fourier coefficients of the fundamentals do not interfere and their harmonics do not overlap.

3.2.3 Power Measurement Example

In Figure 3-5, three non-contact sensor prototypes are mounted with custom enclosures to a three phase power cable. The close proximity of the conductors causes significant overlap in the magnetic fields outside the cable. A visualization of the non-contact current sensor vectors is shown in Figure 3-6. The “traditional” current sensors form an orthonormal basis shown in dashed lines. Despite the mixed fields,

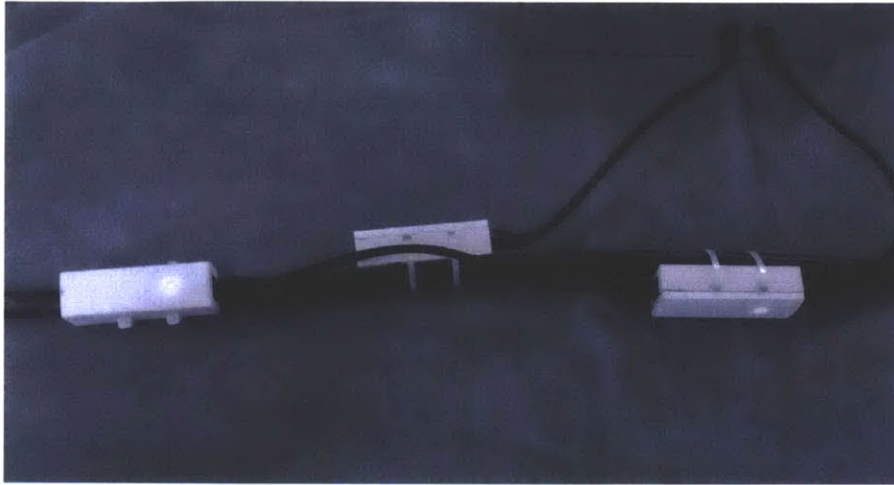


Figure 3-5: Monitoring a three phase power line with non-contact sensors

the non-contact sensors are still linearly independent and span \mathbb{R}^3 . After applying the calibration load to each phase, the fit matrix $[K]$ was calculated using the process described. Figure 3-7 shows a comparison between standard current sensors and the non-contact sensors for loads on all three phases.

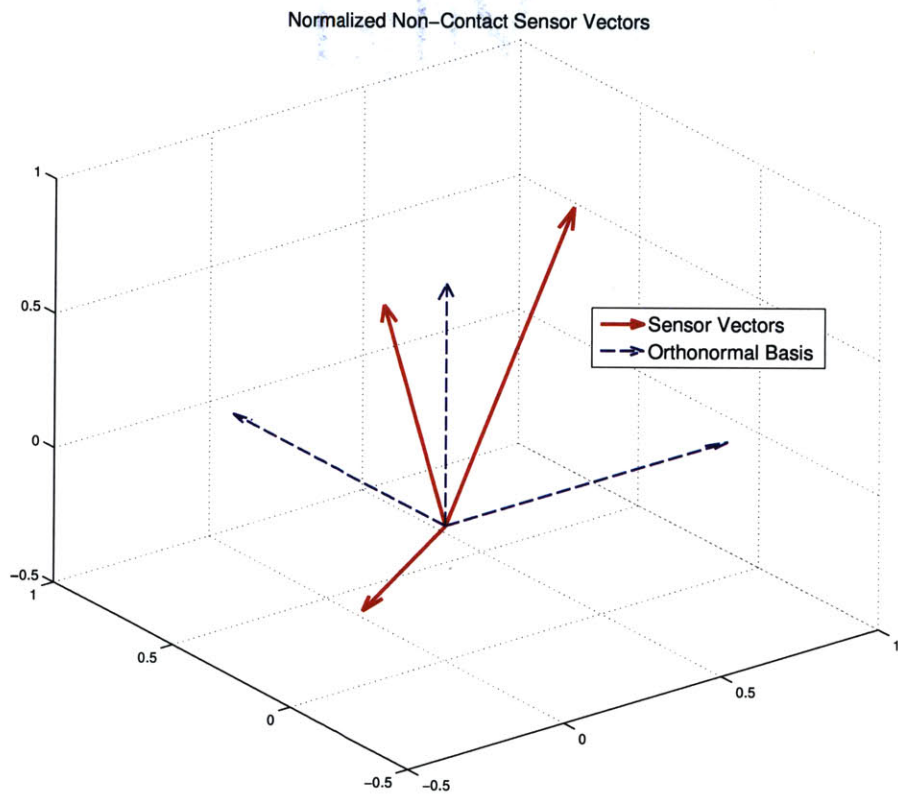


Figure 3-6: Visualization of normalized sensor vectors versus the orthonormal phase basis

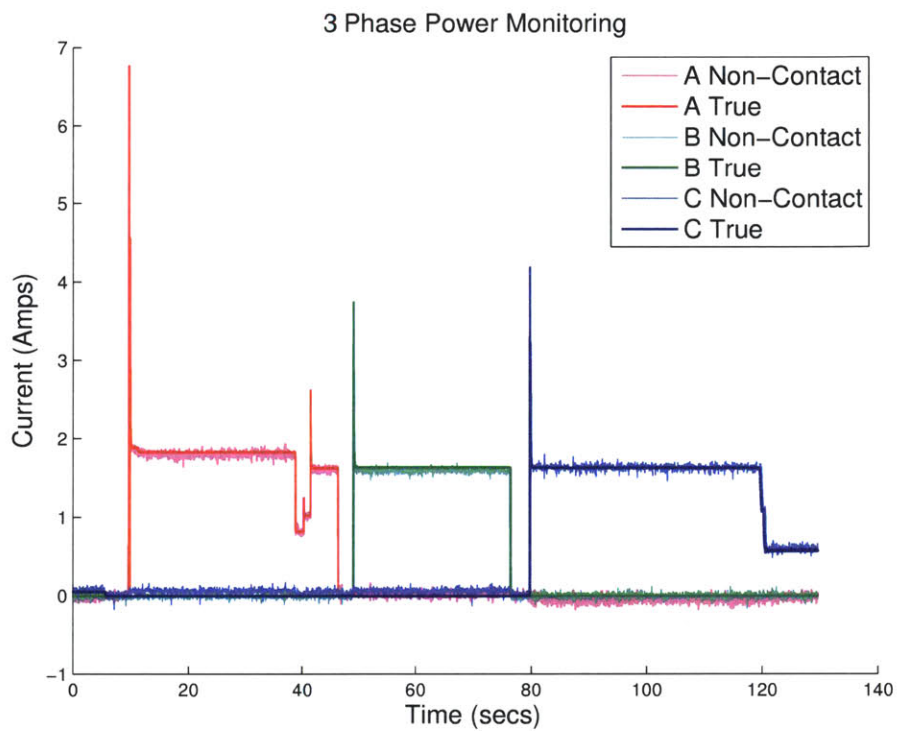


Figure 3-7: Hall Effect non-contact sensors and traditional current sensors on a 3 phase system

3.3 Vibration Transfer Function

The non-contact voltage sensor of Section 2.3 enables a fresh approach to another critical electromechanic diagnostic technique, vibration monitoring. Applying the voltage sensor for back-EMF sensing allows an electromagnetic machine to be used as its own mechanical network analyzer for structural diagnostics. This section introduces a signal processing technique to estimate this vibration transfer function from non-intrusive sensors.

3.3.1 Motivation

Profiles for the maintenance of electromechanical systems arise from essentially three different engineering management strategies. Maintenance can occur when a system breaks or becomes excessively revealing, essentially deferring costs to a “day of reckoning” when the system is guaranteed to be unavailable. Alternatively, critical maintenance can occur on a scheduled or routine basis to attempt to ensure system availability though with recurring expenses. Or lastly, one can attempt to optimize maintenance costs and system availability by working to predict needed repairs before systems become extreme failures [34–36]. This third option, often referred to as condition-based maintenance, is appealing in the sense that, if the requisite condition-monitoring is efficient and effective, a desired degree of mission capability can be achieved with well-reasoned expenses [34,35].

For electromechanical systems, vibration measurements are often utilized as an input to condition-based maintenance decision-making [34–37], and many organizations have issued standards for mechanical vibration and condition-based maintenance [38, 39]. Further, researchers continue to develop condition monitoring and diagnostic procedures which utilize vibration sensors. For examples, using various signal processing techniques, vibrational monitor outputs can be used to detect frequency modulations characteristic of ball bearing defects [40–43], and to supplement traditional motor current signature analysis in detecting rotor faults [44,45].

The majority of these vibration-based algorithms rely on measurements taken at

rated speeds. As such, the information gained for diagnostic purposes is limited to discrete excitation frequencies, i.e. the electrical and mechanical fundamental, harmonic, and modulated frequencies. This information, while clearly useful for specific diagnostic purposes, may lack the richness to distinguish actuator pathologies from degraded mechanical structure, e.g. the vibration mounts. A machine’s vibrational transfer function (VTF), which relates excitation, e.g. rotor speed, to vibration over a range of operating speeds, offers a more complete view of the system and is commonly used to determine diagnostic information such as the natural resonance characteristics as well as noise transfer paths [46].

Empirical characterization of vibrational frequency responses in mechanical systems typically requires special experimentation, e.g. a strike-hammer or a shaker for wide-band excitation [46, 47]. This paper presents an alternative, less intrusive approach for VTF characterization that takes advantage of a machine’s spin-down. During turn-off, a machine’s operation covers a continuous wide-frequency band, i.e. from rated operating speed to stand-still. This operating interval of swept operation allows the estimation of the machine’s VTF *in-situ* and with minimal sensor installation. This paper describes this minimal set-up and the signal processing methods utilized for VTF extraction, and demonstrates spin-down estimation of the VTF in laboratory experiments and field applications aboard serving U.S. Navy warships.

3.3.2 Background

From a simplified perspective, an electric motor or generator mounted on resilient mounts can be modeled as a spring-mass-damper system with an eccentric mass vibration [47], as depicted in Figure 3-8.

The equation which governs the motion of the actuator mass is,

$$m\ddot{x}_m(t) + c\dot{x}_m(t) + kx_m(t) = F_m(t), \quad (3.42)$$

where m is mass, x_m is the position of the system, k is the spring constant, c is the damping ratio associated with the mount, t is time, and $F_m(t)$ is a forcing function.

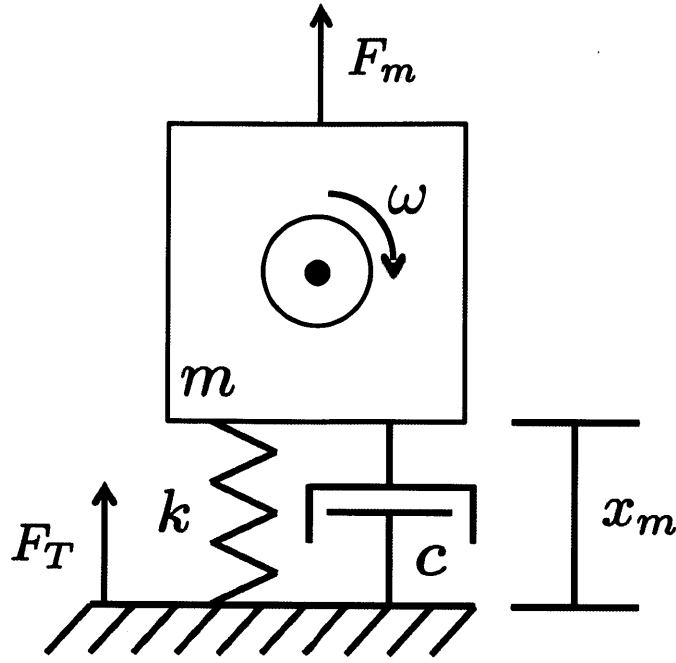


Figure 3-8: Free body diagram for electromechanical machinery and mount system.

This equation can be rewritten in terms of system acceleration as,

$$ma_m(t) + c \int_{-\infty}^t a_m(\tau) d\tau + k \iint_{-\infty}^t a_m(\tau) d\tau = F_m(t), \quad (3.43)$$

where $a_m(t)$ is the acceleration of the motor. Taking the Laplace transform of this equation and rearranging into transfer function form, reveals the Laplace-domain relationship between the acceleration, $A_m(s)$, and the system forcing, $F_m(s)$, as,

$$\frac{A_m(s)}{F_m(s)} = \frac{\frac{s^2}{m}}{s^2 + \frac{c}{m}s + \frac{k}{m}}, \quad (3.44)$$

where s is the Laplace transform operator.

For a rotating machine with an eccentric mass under steady-state conditions, this system forcing, $F_m(t)$, takes the form,

$$F_m(t) = C\omega_m^2 \cos(\omega_m t), \quad (3.45)$$

where C is a constant related to load mass and imbalances, and ω_m is the speed of

the rotating shaft. A proportionally related virtual input function can be defined as,

$$\Phi_m(t) = \omega_m^2 \cos(\omega_m t), \quad (3.46)$$

and its corresponding Laplace representation substituted into (3.44) to get,

$$\frac{A_m(s)}{\Phi_m(s)} = \frac{C \frac{g^2}{m}}{s^2 + \frac{c}{m}s + \frac{k}{m}}. \quad (3.47)$$

This is the equation we refer to as the vibration transfer function (VTF) with units of kg^{-1} assuming (3.46) maintains the units of (3.45).

(3.46) is a function that can be derived from the motor speed, ω_m . When combined with accelerometer measurements, this information can generate (3.47). This VTF is a transfer function that contains the same dynamic properties, e.g. natural frequency, as (3.44), and also scales as the forcing coefficient, C , changes, e.g. due to an increased load imbalance. During a motor spin-down when the speed is reducing from the motor's steady-state operating speed to stand-still, speed and vibration sensors can be used to generate an empirical representation or "eVTF" of (3.47) in that frequency range. The properties of this eVTF can then be used for machinery diagnostics.

Example Application: Machine Radiated Noise

One area of concern to many machinery operators is the force transmitted from the machine to the surface underneath, F_T (Figure 3-8). For example, a Navy ship may need to maintain its radiated acoustic noise at a minimum to avoid detection, and an increase in force transmitted from the machine can lead to increased noise levels. With knowledge of ω_n , an estimation of the force transmitted by the machine through its mounts to the supporting structure, relative to a value when the force is known to be acceptable, can be made from single accelerometer measurements on the machine system itself.

In steady-state, an electromechanical machine has a transmissibility in the isolation range that is similar to that for zero damping. Therefore, c can be ignored as long

as the operating speed is away from resonance [48, 49]. This means the force transmitted through the mounts to the baseplate occurs primarily through the stiffness of the mounting (the spring in Figure 3-8), and follows Hooke's law,

$$F_T \approx k \cdot x(t). \quad (3.48)$$

From (3.47), the natural frequency of this 2nd order system is given by $\omega_n = \sqrt{\frac{k}{m}}$. Since the machine is following simple harmonic motion with acceleration described as $a(t) = A_{m,ss} \cos(\omega_{ss}t)$, where the ss denotes steady state conditions, (3.48) can be written in terms of m , ω_n , ω_{ss} , and $a(t)$, as,

$$F_T = -\frac{m\omega_n^2 a(t)}{\omega_{ss}^2}. \quad (3.49)$$

For diagnostic purposes, only the magnitude of F_T is of concern; taking the magnitude of (3.49) yields,

$$|F_T| = \frac{m\omega_n^2 A_{m,ss}}{\omega_{ss}^2}, \quad (3.50)$$

where A_m is the magnitude of the acceleration. In many situations, the system mass, m , and the machine's steady-state operating speed, ω_{ss} , are consistent whenever the machine is in normal operation. As such, an estimate of the ratio of force transmissions from a time when the machine is in a known "good" condition, $|F_T|$, to the present condition, $|F_T|'$, can be achieved from only acceleration and speed measurements of the motor, i.e.

$$\frac{|F_T|'}{|F_T|} = \frac{\omega_n'^2 A'_{m,ss}}{\omega_n^2 A_{m,ss}} \quad (3.51)$$

Here, ω_n^2 and $A_{m,ss}$ are values from when the machine is in the "good" condition and $\omega_n'^2$, and $A'_{m,ss}$ are derived from the most recent measurements. Practically, (3.51) indicates the value of estimating the eVTF for (3.47) during machine spin-down. From the eVTF, it is possible to estimate the transfer function peak or, essentially

the natural frequency of the system from the observed resonant peak. Estimation of the natural frequency of the mount from the eVTF makes it possible to distinguish machine imbalance from degradation of the mount, both of which can cause increased transmitted vibration that might be indistinguishable from steady-state measurements alone. Changes in the forcing function, e.g. the vibration energy created by operating the machine, will generally increase the magnitude of the entire eVTF. Aging or degradation of the mount alone will shift the resonant frequency of the eVTF. Comparison of successively observed eVTF's can be used to distinguish progressive imbalance from aging of the mount.

3.3.3 Sensor Measurements and eVTF Generation

In practice, estimation of the eVTF requires knowledge of the actuator vibration and speed during spin-down or during a similar operating sweep. We have developed an electronic sensor that can generate an eVTF without installation of a tachometer, strictly from relatively non-intrusive electrical measurements.

3.3.3.1 Data Collection

A single-axis accelerometer mounted vertically is used to measure vibration. This stream is considered the “output” of the system and requires little preprocessing other than scaling the output from mV to m/s^2 to provide the estimate, $\hat{A}_m(t)$.

The “input” to the system, (3.46), is estimated from the motor speed. This speed is inferred using a back-EMF sensor measuring winding voltages on the machine. A back-EMF sensor is the preferred method for gathering spin-down speed as it is accurate, portable, and easy to install in the field. When a motor is disconnected from its power supply, or the prime mover is turned off in the case of a generator, the rotor will continue spinning due to its inertia. Residual magnetism generates voltage on the stator. The characteristics of this voltage, e.g. amplitude and zero-crossings, can then be used to estimate rotor speed.

The back-EMF sensor Figure 3-9 employed in this study uses non-contact differential capacitive sensing to detect the electric field generated by the phase lines of

the machine. Three copper plates, shown in Figure 3-9a, are secured against the insulating jackets of the phase lines inside the machine's terminal box. These plates are also electrically connected to a circuit with the simplified schematic shown in Figure 3-9b. Here, plate A (P_A) connects to the (+) side of the first AD8421 differential amplifier, plate B (P_B) connects to the (-) side of the first AD8421 as well as the (+) side of the second AD8421, and plate C (P_C) connects to the (-) side of this second amplifier. These plates capacitively couple to the phase line voltages, V_A , V_B , and V_C , respectively. A more detailed explanation of the non contact voltage sensor can be found in [2].

Under this configuration, the voltages generated at the outputs of the back-EMF sensor are given by,

$$V_{o1} = g_{a1}V_A - g_{b1}V_B + g_{n1}V_n, \quad (3.52)$$

and,

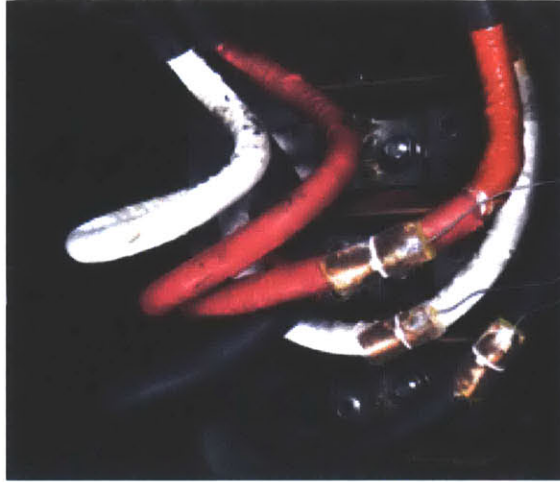
$$V_{o2} = g_{b2}V_B - g_{c2}V_C + g_{n2}V_n. \quad (3.53)$$

In these equations, V_n represents a common-mode background noise present at the output of the circuit, and the g terms represent combined gains of the capacitive coupling and amplifier stages of the circuitry. The ratio, $r_g = \frac{g_{n1}}{g_{n2}}$ can be estimated by performing a scalar fit of $V_{o1} = r_g V_{o2}$ with measurements achieved when the machine is at stand-still and the phase voltages are zero. Then, the differential calculation, $V_o = V_{o1} - r_g V_{o2}$, gives the voltage measurement,

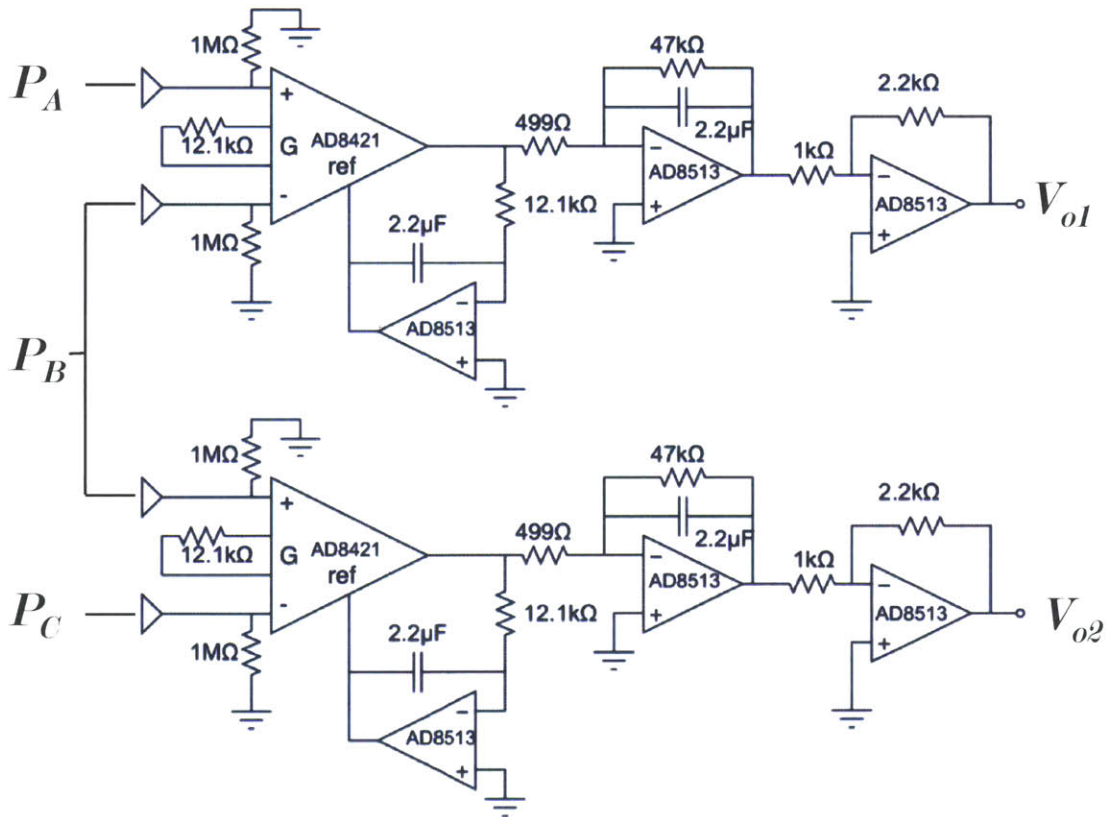
$$V_o = (g_{a1}V_A - g_{b1}V_B) - r_g (g_{b2}V_B - g_{c2}V_C), \quad (3.54)$$

which has the common-mode noise term eliminated. While this voltage signal does not have a physical meaning related to the system, it is linearly proportional to the rotor's back-EMF voltage.

This mechanical speed can be estimated from (3.54) in a number of ways based on the signal's amplitude and/or frequency. For this research project, if the following conditions are met,



(a) Photo of Sensor Plates



(b) Back-EMF Sensor Schematic.

Figure 3-9: The non-contact back-EMF sensor system used for estimating motor speed.

- There is no clipping in the back-EMF sensor waveforms during steady-state operation, and
- There is no active electromagnetic control (e.g. braking) applied to the system during spin-down,

then the machinery spin-down speed profile is extracted from the signal envelope using a Hilbert transform based method. Here, the Hilbert transform produces a waveform linearly related to the signal amplitude. This waveform is then scaled to match the mechanical excitation speed, ω_m , based on knowledge of the machine's steady-state speed rating. Further information on this method is given in [5, 50].

If these conditions are not met however, then the signal amplitude is not linearly related to speed throughout the entire spin-down. In this case, the electrical-speed, ω_e , profile is achieved based on signal frequency estimates gained from a zero-crossing detection procedure. In this procedure, the k_{th} zero-crossing is identified by a change of sign between two adjacent waveform samples at t_n and t_{n+1} , and its time-location, $t[k]$ estimated based on the zero-value of the linear interpolation of the signal values $V_o(t_n)$ and $V_o(t_{n+1})$. Then, the signal frequency at time instance, $t[k]$, is estimated as,

$$\hat{\omega}_e(t[k]) = \frac{2\pi}{t[k+1] - t[k-1]}. \quad (3.55)$$

In the discussion above n represents the sample time-index, and k represents the time indexing of zero-crossings.

In general, this method may be less noise-immune than the Hilbert transform method as it is prone to uncertainty around zero-crossings, particularly at low frequencies as the signal amplitude also decreases with speed. While many methods exist for accurate zero-crossing detection in noisy environments, e.g. [51–53], for the purpose of gaining eVTF estimates here, all zero-crossings including those generated by additive noise were detected as described above and outliers removed prior to the frequency estimation of (3.55). From the $\hat{\omega}_e$ estimates, the mechanical speed estimate, $\hat{\omega}_m$, is calculated based on the number of pole-pairs in the machinery. Then, this stream is linearly interpolated to match the sampling rate of the original mea-

measurements for calculating the virtual input function, $\Phi_m(t)$, from (3.46). Finally, the virtual input is calculated. In the case where the vibrational excitation is provided directly from eccentricities in the machine's rotor, $\Phi_m(t)$ is estimated from $\hat{\omega}_m(t)$ as,

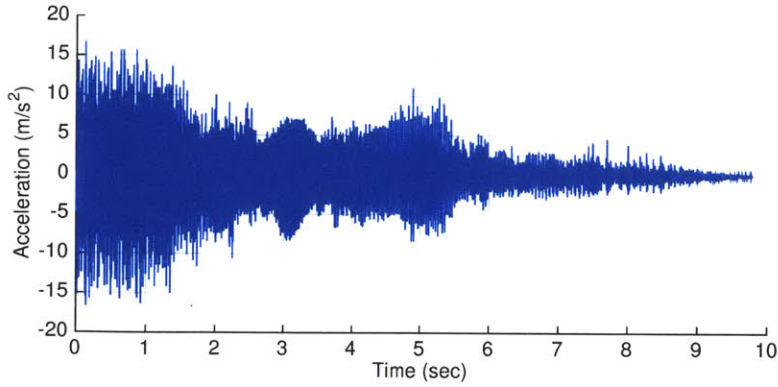
$$\Phi_m(t) = \hat{\omega}_m^2(t) \cos \left(\int_0^t \hat{\omega}_m(\tau) d\tau \right). \quad (3.56)$$

An example of the two time-domain signals required for the eVTF are shown in Figure 3-10. The top plot shows the measured acceleration of the machinery during the spin down, which starts around the 1 second mark as noted by the beginning of the attenuation of the speed curves in the bottom plot. Here, the curves are normalized by their peak values for simultaneous plotting. The blue curve shows the estimated speed, in this particular case, based on the Hilbert transform procedure. The orange curve is the square of the blue curve, and the yellow curve is the calculated virtual input, $\Phi_m(t)$.

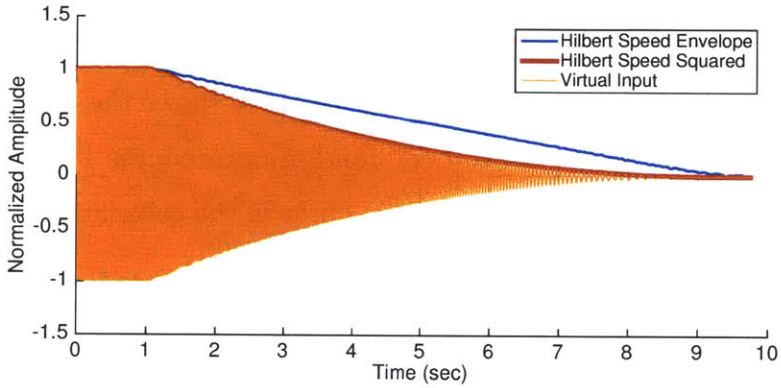
3.3.3.2 Short-Time Fourier Transform Analysis

In accordance with (3.47), the time-domain spin-down signals of Figure 3-10 need to be transformed into frequency domain signals. Empirically estimating a transfer function through spin-down analysis can be susceptible to noise from other nearby machinery [49]. Further complicating the process, the excitation frequency changes continuously with rotor speed. As such, the Short-time Fourier Transform (STFT) is used to process the signals to reduce the uncorrelated noise and minimize frequency of excitation spreading in the analysis. Under this method, the time domain waveforms are windowed at a series of time locations during the spin down process, and the Fast Fourier Transform (FFT) is used to process each modified time signal resulting in a time-binned frequency representation of the original signals.

For this STFT application, a Hanning window is used as the mask for the input and output waveforms. This window is tunable through two parameters, the time-width of the window, T_w , and the overlap between adjacent windows, o_v . The Hanning windows are multiplied with the input and output waveforms to create "masked" versions.



(a) Vibration Sensor Output



(b) Speed Estimate and Virtual Input

Figure 3-10: Example time-domain signals used for eVTF generation.

As an example, the windowing of the virtual input at the 1.5 second mark during a motor spin-down is shown in Figure 3-11. The same processes is also performed on the output.

A full series of Hanning windowed inputs and outputs for a motor during spin-down is shown in Figure 3-12. Here, the tunable parameters are set to $T_w = 1$ second and for clarity, $o_v = 0\%$. The top plot shows the individual windows and their time locations, the middle plot shows the series of resulting masked virtual input waveforms, and the bottom plot the series of masked vibrational output waveforms.

Via the Fast Fourier Transform (FFT), each Hanning-windowed input and output allows for the generation of a frequency spectrum specific to a short period of time during the spin down process. These spectrums can be indexed as $\Phi_{m,i}(j\omega)$ and $A_{m,i}(j\omega)$, respectively, where i denotes the Hanning window index. A corresponding

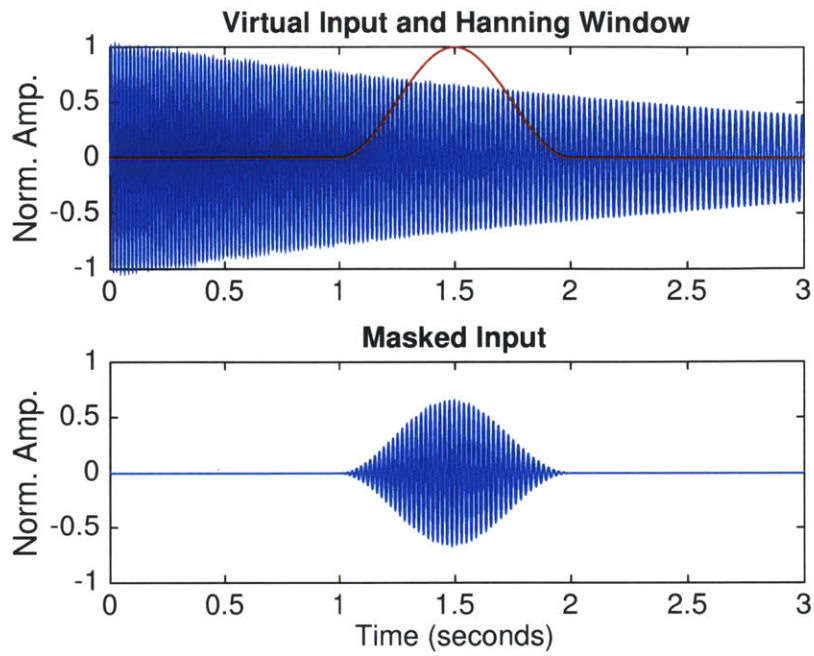


Figure 3-11: A Hanning window masking of the normalized virtual input centered at 1.5 seconds.

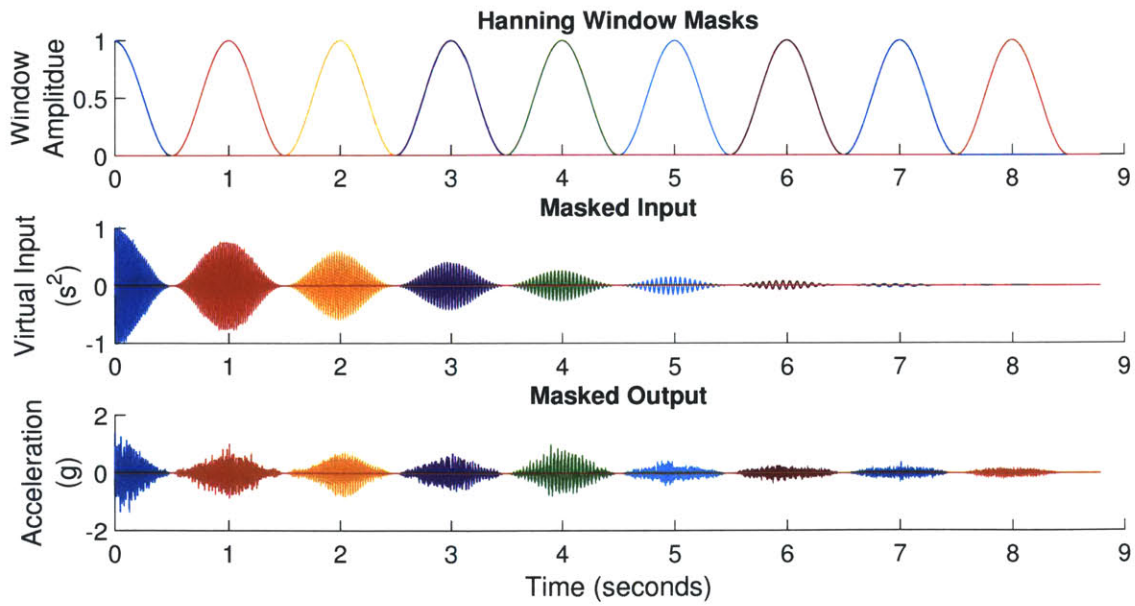


Figure 3-12: Hanning Windows and Masked Inputs and Outputs

“center” frequency for each index can be defined as,

$$\omega_i = \operatorname{argmax}_{\omega} \left| \frac{A_{m,i}(j\omega)}{\Phi_{m,i}(j\omega)} \right|. \quad (3.57)$$

To generate the eVTF, first all the output spectrums are “masked” around the corresponding center frequencies such that,

$$A'_{m,i}(j\omega) = \begin{cases} 0 & \omega \leq \omega_i - w/2 \\ A_{m,i}(j\omega) & \omega_i - w/2 \leq \omega \leq \omega_i + w/2, \\ 0 & \omega \geq \omega_i + w/2 \end{cases} \quad (3.58)$$

where the variable w limits the “valid” vibrational response frequencies to those around the frequency of virtual input excitation. This is done to preserve linearity in the eVTF and to ignore sources of noise at extraneous frequencies. Maximum envelope excitation and response spectrums are then defined as,

$$\Phi_{env}(j\omega) = \max_i \left[\operatorname{argmax}_{\Phi_{m,i}(j\omega)} |\Phi_{m,i}(j\omega)| \right], \quad (3.59)$$

and

$$A_{env}(j\omega) = \max_i \left[\operatorname{argmax}_{A'_{m,i}(j\omega)} |A'_{m,i}(j\omega)| \right], \quad (3.60)$$

respectively. That is, at a given frequency, these envelopes are defined as equal to the corresponding indexed spectrum with the maximum magnitude at that particular frequency. The eVTF is then determined as,

$$\frac{A_m(j\omega)}{\Phi_m(j\omega)} = \frac{A_{env}(j\omega)}{\Phi_{env}(j\omega)}. \quad (3.61)$$

Generating the eVTF in this manor allows the maximum amount of information gained during the STFT based analysis to be passed on to the eVTF while also ensuring linearity.

Figure 3-13 shows a graphical representation of this process. In this figure, the

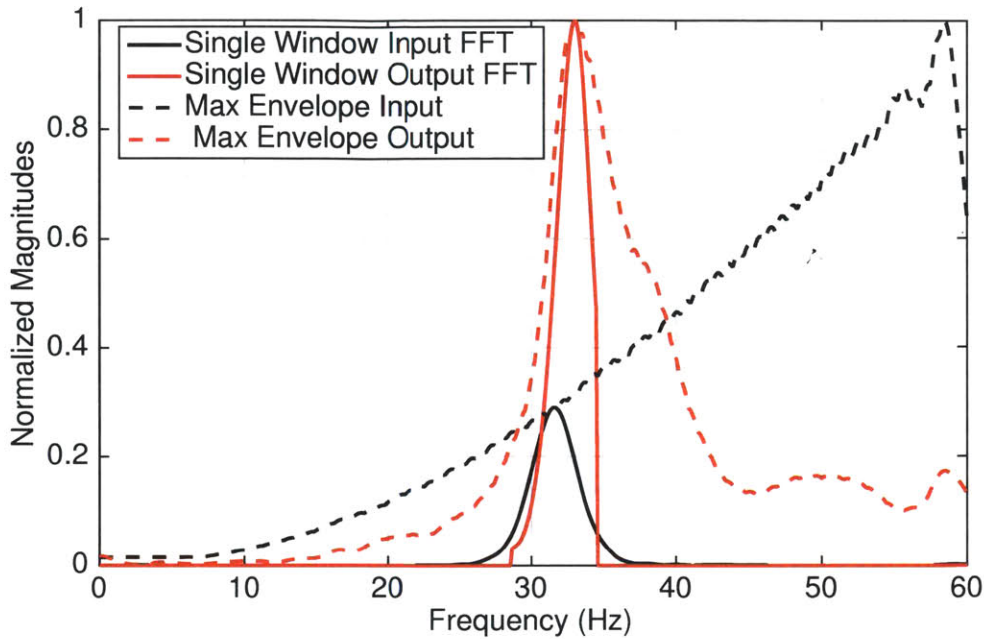


Figure 3-13: Fast Fourier Transform of Masked Inputs and Masked Outputs with Envelopes

magnitudes shown on the y-axis are normalized to the maximum values of each input and output spectrum series, $\Phi_{m,i}(j\omega)$ and $A_{m,i}(j\omega)$, respectively. The dashed curves represent the STFT envelopes of (3.59) and (3.60), while the solid lines represent individual spectrums from a windowed time segment roughly half-way through the spin-down. This period also corresponds to the peak resonance in the system. The figure shows that the FFT for the virtual input signal has a maximum at around 31.5 Hz while the resulting FFT for the output vibration signal has a maximum at around 33 Hz. This offset is due to the fact that there is a time delay in the vibrational response to the forcing input. For this analysis, the width parameter, w , for the output response is set to 6 Hz. As such, the indexed response spectrum is cut off at the frequencies of 28.5 Hz and 34.5 Hz to ensure the maximum envelope is created from the linear vibrational response while still allowing the capturing of the delayed peak response. This particular analysis results in the eVTF plot for the 50A durometer mounts in Figure 3-16.

3.3.4 Tests on Purpose Built Machine Set

To test the method for generating the eVTFs as described in the previous section, and to showcase application of this hardware and signal processing for machinery diagnostics, a test stand consisting of a prime-mover, inertia, and generator was constructed. This test stand, shown in Figure 3-14, centers around a DC permanent magnet motor with dual couplings. One end is coupled to a three-phase induction machine, and the other is connected to a single phase synchronous AC motor, which is disabled and acts only as a flywheel. The three machines are mounted onto a single metal sub-base as is typical for many industrial actuator installations. The sub-base is mounted to a steel box girder with vibration reducing mounts at eight points. A second shaft coupler is attached to the induction motor as an attachment point for an imbalance. This is done to simulate a rotor imbalance, a type of machinery fault, which should appear in the eVTF as an increase in vibration magnitude at resonance. Five commercial vibration dampening mounts of different durometer (30A, 40A, 50A, 60A and 70A) were used to emulate a scenario where the mounts' stiffness increases or decreases over time, the effect of which should appear in the eVTF as a shift in resonance.

3.3.4.1 Comparison of Spin-down eVTF and Steady-state eVTF

Initial tests were performed on a subset of mounts to validate the eVTFs generated during the machinery spin-down by comparing them against eVTFs generated from steady-state measurements. A Python script is used to automate the steady-state measurement collection by commanding a series of DC-voltages from a power supply connected to the DC motor. At each discrete voltage level, the script reads motor speed from a shaft encoder and measures the sub-base vibration with a standard industrial accelerometer. The script removes data collected prior to when the machines reach steady-state at each voltage level, and the remaining vibration data is analyzed using Welch's power spectral density estimate [54]. At each voltage level, the peak of the power spectral density at a frequency closest to but greater than the measured

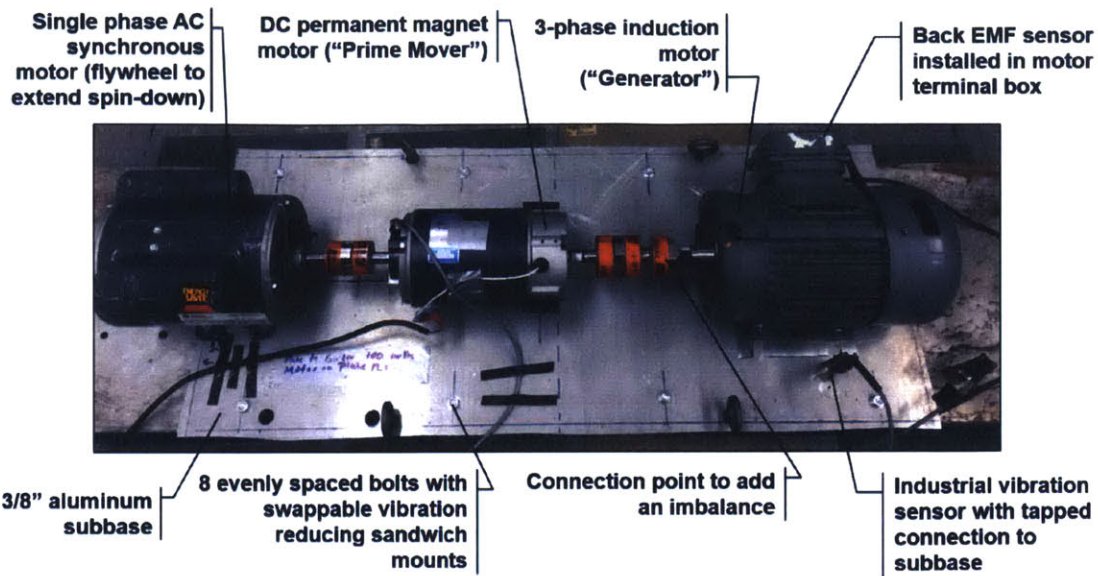


Figure 3-14: Image showing the purpose built machine set for testing eVTF generation and machinery trends useful in diagnostics

speed of the motor is taken as the vibration magnitude at the measured speed. Finally, each corresponding measured operating speed is squared to generate a virtual input for generation of the steady-state eVTF.

Figure 3-15 gives a comparison of a spin-down eVTF with a steady-state eVTF for the 60A durometer mounts and machine setup described above. For this particular test, the DC motor was run in steady-state at 105 discrete voltage levels with 4 seconds of steady-state operation at each resulting motor speed. The spin-down eVTF was calculated from the data collected following the motor's power supply turning off after the maximum voltage level was reached. This spin-down process took approximately 10 seconds, and the analysis parameters were set to $T_w = 1s$, $\sigma_v = 90\%$, and $w = 3$ Hz.

As seen in the figure, the two curves are very similar and show the same resonant peak, indicating that the spin-down procedure accurately captures the important features of the system's VTF. The spin-down eVTF appears much smoother with less variance in measurements because the virtual input is derived from the Hilbert Transform and lowpass filtered rather than calculated directly from speed measurements

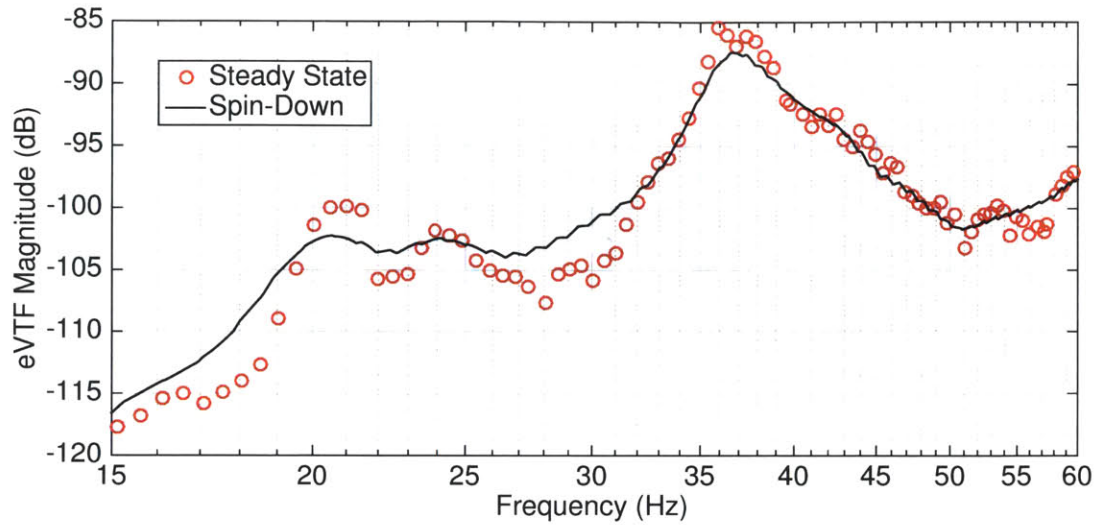


Figure 3-15: Comparison of a spin-down derived eVTF and steady-state measurement derived eVTF.

as is the case for the steady-state eVTF. This particular result is well representative of the results gained from other durometer mounts.

It should be noted that the number of features present in the eVTFs exceeds those allowed by the second order model described in Section 3.3.2. This is due to the fact that the model assumes a single-degree of freedom and linearity in the system, while real-world machinery systems have three-degrees of freedom and many sources of non-linearity. Nevertheless, for the purposes of diagnostics, e.g. sensing changes in transferrable noise, the measured eVTF around the resonant peak approximates a second order system and provides useful parametric estimations as illustrated below.

3.3.4.2 eVTFs with Condition Changes

With confidence in the eVTF method described, the process was applied to a total of 10 conditions across the 5 mount types and with and without a 17 g imbalance attached to the rotor system. These tests were done to show the utility of the eVTF method towards machinery diagnostics in accordance with the example application described in Section 3.3.2. The premise of this method is that eVTFs gained opportunistically during a machinery spin-down can provide useful information, e.g. changes in vibration amplitude and resonant frequency, indicative of system failures,

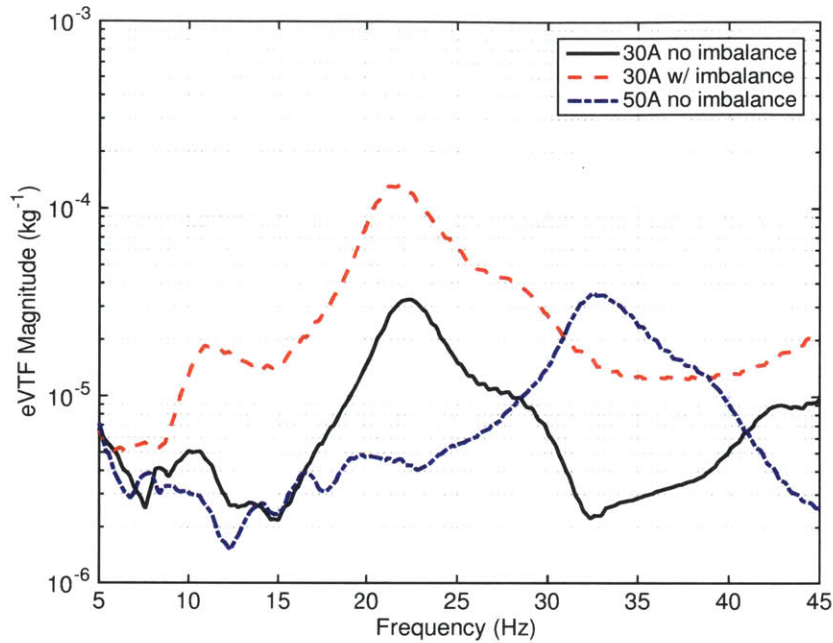


Figure 3-16: eVTFs from tests of the 30A and 50A mounts showing a shift in natural frequency with a change in durometer, and an increase in amplitude due to an increase in system imbalance.

i.e. machinery imbalance and mount degradation, respectively.

Figure 3-16 illustrates this premise with eVTFs gained during testing. In this figure, the solid black curve shows the eVTF gained during machinery spin-down when the 30A mounts were installed and no additional imbalance applied to the system. This curve shows a resonant peak just above 22 Hz at a magnitude of approximately $3.2 \times 10^{-5} \text{ kg}^{-1}$. When the 17g imbalance was added to the shaft, the magnitude of the vibration increased significantly, but without a significant shift in resonance as shown by the dashed red plot, which has a peak around $1.4 \times 10^{-4} \text{ kg}^{-1}$ at just under 22 Hz. However, when 50A durometer mounts were used without an imbalance, while the magnitude of the eVTF remained similar (approximately $3.6 \times 10^{-5} \text{ kg}^{-1}$), the resonant frequency shifted significantly to 33 Hz (dot-dashed blue curve). Thus, the spin-down generated eVTF increases the amount of distinguishing information available to an operator in diagnosing issues in the machinery system.

Table 3-1 compiles the results from all tests involving the 5 mount types described

Table 3-1: Comparison of eVTF characteristics for various durometer mounts and rotor imbalances.

Duro- meter	Imbalance (g)	Number of Tests	Natural Frequency (Hz) [mean (<i>std</i>)]	Peak Amplitude (kg^{-1}) [mean (<i>std</i>)]
30A	0	4	22.3 (0.170)	$32.5 (1.98) \times 10^{-6}$
	17	4	21.8 (0.153)	$136 (1.45) \times 10^{-6}$
40A	0	4	26.1 (0.117)	$39.9 (0.463) \times 10^{-6}$
	17	4	25.6 (0.125)	$158 (1.38) \times 10^{-6}$
50A	0	4	32.7 (0.193)	$36.1 (0.386) \times 10^{-6}$
	17	2	31.2 (0.155)	$143 (2.20) \times 10^{-6}$
60A	0	4	38.5 (0.108)	$33.2 (0.387) \times 10^{-6}$
	17	3	36.4 (0.122)	$134 (10.55) \times 10^{-6}$
70A	0	5	44.3 (0.195)	$41.2 (0.887) \times 10^{-6}$
	17	4	41.4 (0.083)	$161 (1.92) \times 10^{-6}$

above. The 3rd column of this table indicates the total number of tests performed on each mount and imbalance. Mean values from each set of tests for the system's natural frequency and peak eVTF amplitude are given in columns four and five, respectively. Also, in each of these columns the accompanying standard deviations are included to give an indication of repeatability. As observed there, the tests are very consistent for each configuration with the standard deviation in measured natural frequency less than 1% of the mean value and that of the peak amplitude less than 8% for all tests.

3.3.5 Field Tests on U.S. Navy Equipment

A series of field tests were conducted on an active U.S. Navy mine countermeasures ship (MCM). The ship has 3 ship service diesel generators (SSDGs) each rated at 375 kW. Each generator set is mounted on a metal sub-base, which is attached to the hull of the ship via 8 resilient mounts in a configuration similar to the laboratory setup described in the previous section. The U.S. Navy is interested in non-intrusive, in-situ characterization of these mounts for tracking changes in the vibration energy

Table 3-2: Statistical characteristics for the eVTFs gained during MCM generator spin-down.

Generator Number	Number of Tests	Natural Frequency (Hz) [mean (<i>std</i>)]	Peak Amplitude (kg^{-1}) [mean (<i>std</i>)]
1	5	18.4 (<i>0.146</i>)	136 (<i>9.12</i>) $\times 10^{-6}$
2	5	19.0 (<i>0.109</i>)	103 (<i>7.77</i>) $\times 10^{-6}$
3	4	19.2 (<i>0.0618</i>)	96.8 (<i>4.35</i>) $\times 10^{-6}$

passed to the hull of the ship. The work described here is part of a larger project to integrate a self-sustaining sensor [5,6,55] inside the SSDG terminal box, which would alert the operator when the mounts are beginning to degrade. For this experiment however, the standard industrial accelerometers and the back-EMF sensors described earlier were used.

Each generator set contains a 6-cylinder, 4-stroke diesel engine for driving the prime mover at 30 Hz (1800 RPM). The cylinders are fired in pairs at 15 Hz with a 120 degree phase shift between pair firings. This generates significant vibrational energy at 45 Hz. For the subsequent analysis presented here, this vibrational component from the piston firing rather than the rotational speed of the shaft was used to generate the eVTF plots, with the virtual input appropriately scaled so that,

$$\Phi_m(t) = (1.5\hat{\omega}_m(t))^2 \cos\left(1.5 \int_0^t \hat{\omega}_m(\tau) d\tau\right), \quad (3.62)$$

Additionally, during these tests the back-emf sensor outputs were observed to be clipping in the steady-state so the zero-crossing method was used for estimating ω_m . The spin-down of these generators took approximately 25 seconds, and the analysis parameters were set to $T_w = 1\text{s}$, $\sigma_v = 90\%$, and $w = 4\text{ Hz}$.

A total of 14 spin-downs were measured on the generators under various ship conditions including while the ship was in-port with no other machinery operating, and under-way when several other pieces of machinery were simultaneously operating. Fig. 3-17 shows example eVTFs gained during spin-down of each of the three ship

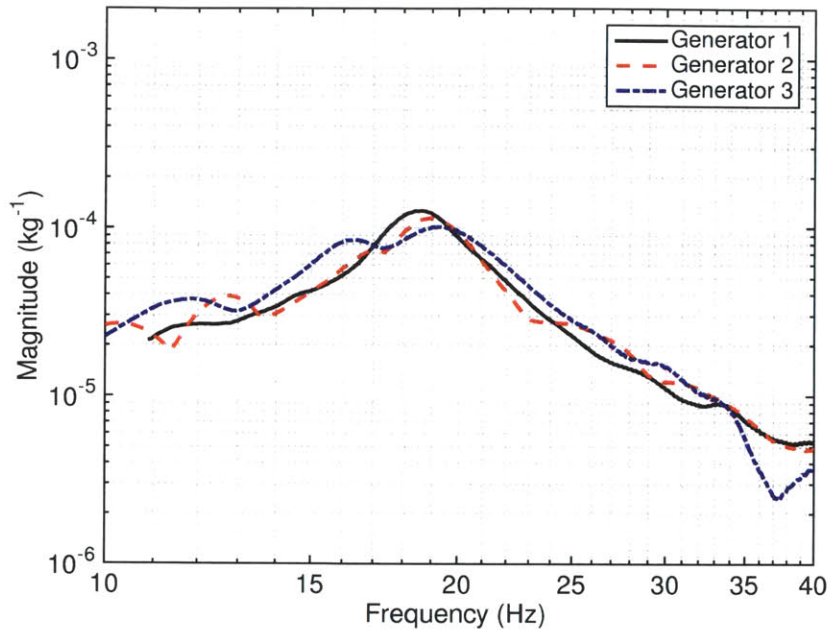


Figure 3-17: Mine countermeasure ship generator eVTFs as measured during generator spin-downs.

generators, and Table 3-2 compiles the statistical characteristics for all the eVTFs tests performed. Fig. 3-17 shows clear resonant peaks in all three generators around 19 Hz and all with a similar amplitude of approximately 10^{-4} kg^{-1} . These curves are well representative of all the eVTFs gained for each corresponding generator, regardless of the ship's condition, and simultaneous electromechanical load use. This is confirmed by comparing the standard deviations in resonant peak amplitude and frequency locations with their corresponding mean values in Table 3-2. Here, for all tests, the standard deviation in peak amplitude estimates was less than 8% of the mean, while those in the peak amplitude frequency locations were less than 1% for each test. Thus, while there is little exposed by these tests to differentiate any physical characteristics of each generator, the consistency in each test gives confidence in the repeatability of the method. Additional research is required to track the evolution of the eVTF characteristics over a longer period of time, and plans are in place to do so as a part of future self-sustained sensor developments discussed in [5, 6, 55].

In addition to the generator experiments, vibrational analysis tests were performed

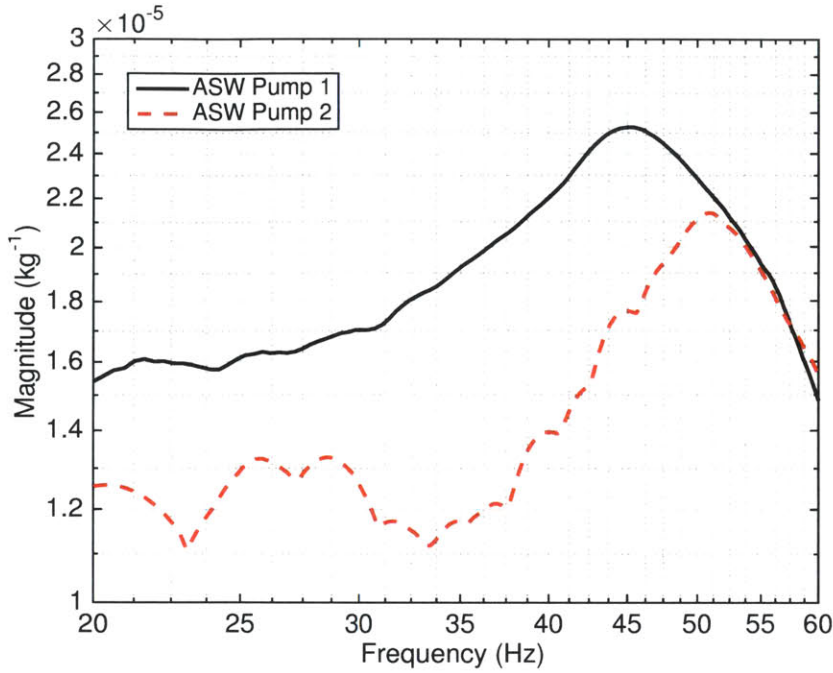


Figure 3-18: Auxiliary seawater pumps 1 and 2 spin-down generated eVTFs

Table 3-3: Statistical characteristics for the eVTFs gained during auxiliary seawater pump spin-downs.

Pump Number	Number of Tests	Natural Frequency (Hz) [mean (<i>std</i>)]	Peak Amplitude (kg^{-1}) [mean (<i>std</i>)]
1	8	44.7 (0.328)	$26.0 (0.359) \times 10^{-6}$
2	8	50.9 (0.629)	$20.4 (0.750) \times 10^{-6}$

on two of the ship’s auxiliary seawater (ASW) pumps. Each pump operates at 59 Hz (3545 RPM) and is driven by a 3-phase induction motor rated at 15 kW (20 HP). The crew had noted that one of the ASW pumps (Pump 2) had recently been overhauled, while the overhaul date of the other pump (Pump 1) was unknown. The spin-down times for these pumps were approximately 4 seconds. As such, the window-width parameter, T_w , was shortened to 0.25s, the overlap parameter, o_v , was increased to 95%, and the masking width, w , set to 8 Hz. Example eVTFs resulting from this analysis are shown in Figure 3-18, while Table 3-3 give the statistical characteristics

of all tests performed.

As can be seen in Figure 3-18, there are characteristic differences between the two pump eVTFs with the recently overhauled ASW Pump 2 showing a peak magnitude about 14% lower than ASW Pump 1 and at a frequency about 6 Hz higher. The results shown here are well representative of all the tests performed on the two pumps as observed in Table 3-3, which shows the standard deviation in the resonant peak amplitude was less than 4% of the mean and that of its frequency location less than 1.5% of the mean for both pumps.

While Figure 3-18 shows more compelling differences between machines than the eVTFs generated from the generator spin-downs (Figure 3-17), concrete conclusions on the causes of these differences cannot be made. The most recently overhauled pump (ASW 2) does show a decreased magnitude in its eVTFs compared with those of ASW 1, though their resonant peak frequency locations are increased, which as shown in Section 3.3.4 would suggest a stiffening of the rubber typical of aging mounts. Still, these characteristic differences were clear and repeatable for all tests.

3.3.6 Conclusions

This section has demonstrated the value of a non-intrusive vibration measurement and analysis technique for use during an electromechanical machine's spin-down procedure. As the speed of the machine decreases from its normal operating speed to stand-still, the process inherently provides vibrational-excitation swept across a range of frequencies, permitting estimation of the vibrational transfer function (eVTF). As shown in laboratory tests, this transfer function reveals characteristics useful for machinery diagnostics which are unavailable for estimation during steady-state operation. The consistency of the field test results combined with the interpretability of laboratory test results demonstrates that this analysis method can provide useful information for electromechanical machinery diagnostics. Specifically, the eVTF approach with back-EMF sensing can be deployed as part of a self-contained sensor that provides effective and useful information about both electromechanical machinery and the health of an associated mount.

Chapter 4

Distributed Cloud

The final challenge is providing secure, reliable access to the information acquired by non-intrusive sensors to the end user in a format that is both informative and actionable. The Wattsworth system provides this mechanism through NILM Manager, a web application that allows users to visualize their data and design “Energy Apps” that customize the operation of their non-intrusive sensors. NILM Manager is a distributed application that operates very differently than traditional sensor network platforms. Traditional sensor networks centralize data storage and processing in large server centers. This presents significant security and privacy concerns and also requires a persistent Internet connection for both users and sensors. Many locations such as military or industrial facilities are unwilling or unable to install such a sensor network. The Wattsworth system requires no central storage and control and as such can be deployed and scaled with much greater flexibility. Non-intrusive monitors can join a NILM Manager instance running on the public Internet, hosted on a private network, or even running locally on the monitor itself. This section presents the Wattsworth distributed cloud architecture and describes how NILM Manager can be used for both visualization and control. See Appendix B.4 for a discussion of the servers and configurations used for a Wattsworth cloud deployed at MIT.

4.1 Introduction

Recording current and voltage with enough resolution to identify load characteristics requires sampling at relatively high rates. NILMs capable of interesting diagnostics and load recognition typically generate very large data sets. (4.1) can be used to estimate the storage requirements for a typical installation:

$$R = 2N_\phi \times f_s \times B_{adc} \quad (4.1)$$

where R is the data rate in bytes per second, N_ϕ is the number of phases (usually two for residential and three for industrial environments), f_s is the sampling frequency, and B_{adc} is the ADC resolution, or the number of bytes used to represent a sample measurement (usually about two). The product of these factors is multiplied by two because both current and voltage waveforms are recorded for each phase.

Using (4.1), a NILM running at $f_s = 8\text{kHz}$ will produce over 5GB of data per day for a standard home. Data sets of this size are difficult to transmit over a residential network. In NILMs deployed in [56], for example, equipment operators mailed hard drives and DVD's back to the lab for analysis. The cost in resources and man-hours make this type of installation impractical for all but the most limited deployment scenarios. Even if data can be reliably collected, plotting the current and voltage over a single day involves billions of individual samples which is beyond the capability of many standard software packages (such as Excel). Previous work has focused on using signature detection to reduce the dataset size, storing only equipment “on” and “off” events instead of current and voltage [57–59]. However, such an aggressive data reduction step without a clearly defined outcome or monitoring objective artificially limits the utility of the NILM.

NILM Manager, a cloud platform that enables quick and easy access to NILM data, solves the access and analysis challenges created by high bandwidth or “big data” power monitoring. A “remote” NILM is installed at a facility to be monitored. Desktop-power computing is readily available in “deck of cards” sized hardware that

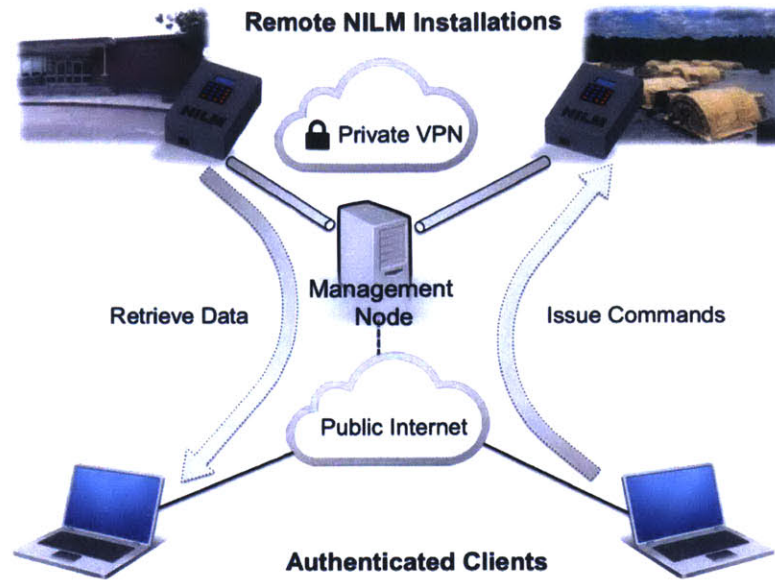


Figure 4-1: NILM Manager system architecture. The management node relays requests from authenticated clients to remote NILMs over a secure VPN. Clients can issue commands and retrieve data over a web interface.

can be installed quickly at a site with terabytes of local storage, at prices comparable to those of a modern solid state electricity meter. Data collected by this remote NILM is never fully transmitted from the site, minimizing network traffic. Rather, data is managed locally on the site computer by custom high-speed database software, NilmDB, described in [8]. NILM Manager provides a central management node that connects multiple remote NILMs with a virtual private network (VPN) and hosts a website that allows authorized users to view and analyze data collected by NILM systems.

4.2 NILM Virtual Private Network

NILM Manager controls the computing “center” of a virtual private network that securely connects remote NILMs, each running NilmDB, to the management node. The network is virtual in the sense that all communication occurs over the public Internet but is encrypted so only NILMs and the management node can decipher the content. Extensive computation on acquired data is relegated to local computing managed by

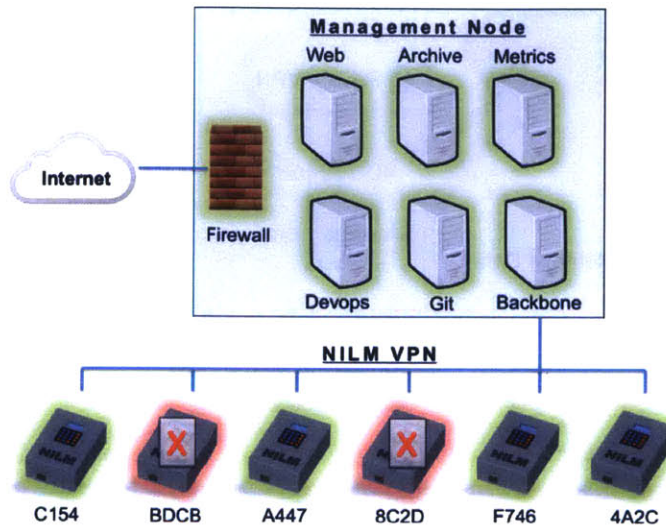


Figure 4-2: NILM network topology as visualized by Nagios [60]. Servers in the management node are outlined in the labeled box. NILM’s connect to the management node through the backbone server. Nagios provides realtime visibility of the network enabling rapid fault detection and diagnosis. The two NILM’s marked with red crosses indicate they are down for maintenance.

NilmDB at on the non-intrusive monitor itself. New programs or “energy apps” can be downloaded from NILM Manager to a NilmDB installation. New analysis results can be uploaded from a remote site to NILM Manager for web presentation, which can of course be through secure connections. Small energy apps and small reports or analysis results, typically a few kilobytes, can provide full, powerful access to remote high bandwidth data with minimal network data requirements. A (low technology) cell phone can and has provided more than enough bandwidth for managing a full industrial monitor in our experiments.

Figure 4-1 shows a conceptual view of the NILM VPN. Users can request data from a NILM and send it commands all without any physical access to the machine. The management node coordinates VPN traffic and ensures that only authorized users have access to NILM systems.

4.3 Web Platform

Users interact with NILMs through a website hosted by the management node. The website is available over the public Internet which means it is accessible from any connected device including tablets and cell phones. Users authenticate with a username and password although certificate based authentication or other forms of protection could be implemented if additional security is required.

By presenting users with a web interface rather than a direct connection (for example via SSH) to the remote meter, the user interaction tools (NILM Manager) are decoupled from NILM system tools. This means NilmDB and other backend software on the NILM can be updated without affecting how the user interacts with the NILM data.

4.4 Data Visualization

One of the primary difficulties in Nonintrusive Load Monitoring is visualizing the high bandwidth data collected by the current and voltage sensors. A NILM produces thousands of data points each second. Tools such as Excel and MATLAB consume significant system resources to produce plots for datasets of this size. Complicating matters further, NILMs often have limited network bandwidth making transmission of the raw data to a workstation difficult or impossible. NILM Manager solves this problem by using a decimation algorithm to visualize large datasets.

4.4.1 Stream Configuration

NILMs connected to the management node are configured through the web interface shown in Figure 4-3. This interface presents the data collected by the NILM as a series of files organized into directories. Users can navigate through the data on the NILM just as they would navigate folders on their desktop. While they appear as flat datatypes on the web interface, each file corresponds to a hierarchy of streams on the NILM itself.

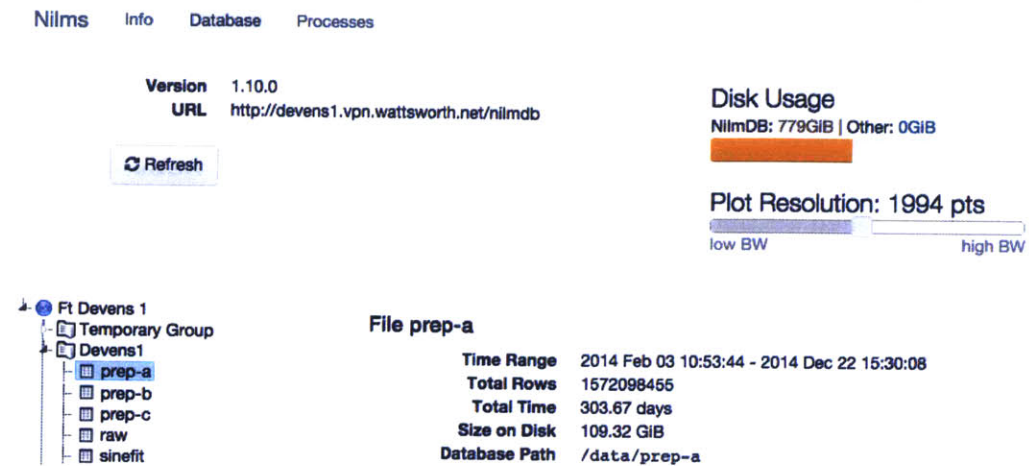


Figure 4-3: The NILM configuration interface. The plot resolution slider controls how many data points are displayed in data visualization tool (shown in Figure 4-4).

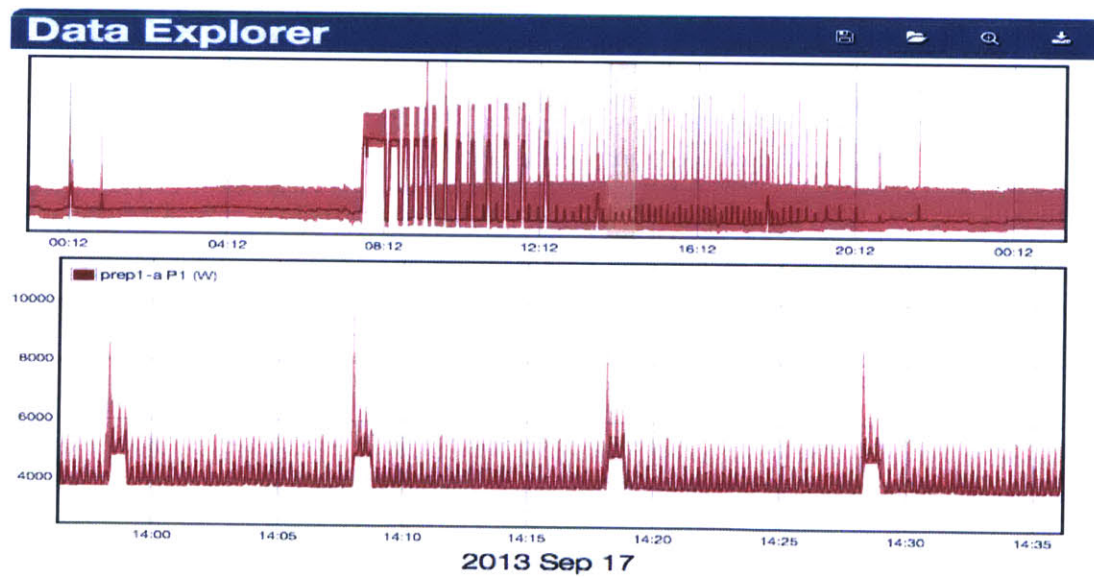


Figure 4-4: Data visualization using the web plotting tool. The upper plot shows 24 hours of power data and the lower plot shows a higher resolution view of the highlighted segment. This view represents 5.1M samples but is drawn using just 2K decimated samples (less than 0.05% of the raw samples)

As the NILM adds data to a stream, it simultaneously computes a decimated child stream. For every four elements in the parent stream, the child contains a single [min, max, mean] tuple. This process is performed recursively with each successive child containing a factor of four fewer elements than its parent. When this process is carried out to completion (the final child containing only one sample), the total storage requirement only increases by a factor of two [8].

The plot resolution slider in the top right of the interface sets the number of data points returned by the NILM when a user requests an interval of data. NILM Manager checks how many data points are contained in the requested interval and returns the lowest decimated child stream that fits within the configured plot resolution. The raw data is only returned if the interval requested is small enough that there are fewer raw samples than the plot resolution setting.

4.4.2 Presentation

The NILM Manager website provides an intuitive plotting interface shown in Fig. 4-4. Note that this figure shows real data from a monitoring site in our research program. The system has been running for nearly two years, and the remote monitor contains tremendously detailed data, down to the envelopes of individual electrical transients as shown in the figure. Nevertheless, access to this data is almost instantaneous from any web connection anywhere. The interface uses decimated streams to allow users to view any dataset from any remote NILM at any time scale. Panning and zooming through the data operates like Google Maps with progressively higher resolution data returned as a user “zooms in” to a particular area of a waveform. Progressive views are delivered essentially instantaneously.

The plotting interface is implemented in Javascript which runs in the client browser. The code is derived from the open source “Flot jQuery” plugin although it has been highly customized for this application [61]. The plotting code has three display modes. If the time interval is short enough that the raw data fits within the plot resolution setting, a simple line graph is displayed. If, as is usually the case, the raw data contains too many samples, data from the selected decimation level is displayed as a

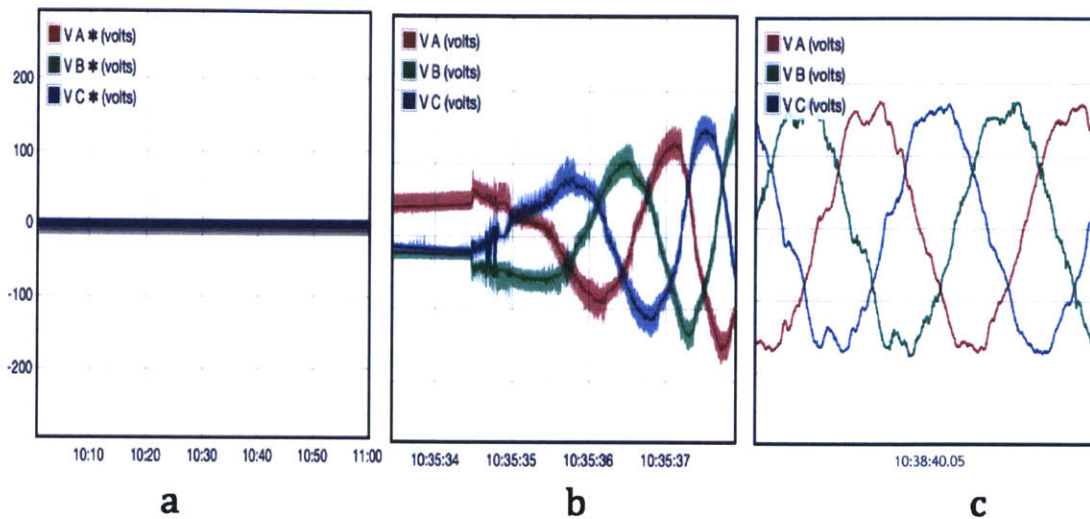


Figure 4-5: NILM Manager automatically adjusts the plot type based on how many points are in the selected dataset. (a) When a stream has too much data and no available decimations a solid line indicates the plot cannot be displayed. (b) If decimations are available an envelope of the dataset is shown, and (c) if the time interval is short enough, the raw data is plotted directly.

[min,max] envelope around the mean which is plotted as a line graph. The envelope is the same color as the mean with added transparency. This provides feedback about the structure of the data without obscuring other time series on the same plot (as in the case of Matlab or Excel). Finally if a time interval contains too much data in all available decimation levels (which occurs when a NILM has not yet decimated a new stream), a thick horizontal line is drawn in place of the data and an asterisk is added to the legend indicating the inability to plot the particular stream at the selected time scale. Figure 4-5 shows the three plot display styles. The client code automatically switches between styles as the user navigates between datasets and timescales.

4.5 Data Processing

Current smart meters typically transmit their measurements wirelessly to a central monitoring node which limits their resolution, as these links generally cannot carry sufficiently large amounts of data [62–64]. Exposing raw data also exacerbates privacy concerns. The on-board CPU cores in even a low-cost NILM process data locally.

Data need never be moved in bulk from the monitoring site. Short, actionable reports and analyses can be transmitted to a facilities manager or service provider as privacy restrictions permit. The information can also be used locally for control. Moving computation from a centralized server to a distributed embedded environment requires an efficient data processing framework. The following sections describe this framework, and illustrate how “apps” on distributed NILM energy boxes can analyze, report on, and control power systems.

4.5.1 Management and Preprocessor

NILMs support remote management through a specialized application programming interface (API), which allows clients to upload and execute custom scripts. This API is exposed to the management node over HTTP with security provided by the VPN tunnel. The management node uses this API for system administration tasks such as database cleanup, software updates and system diagnostics. The management node establishes a sandbox on top of this API in which end users can execute their own scripts called “Energy Apps.” These scripts use input hooks to link to data streams stored on the NILM. An app can use data from multiple streams, each of which may have different intervals of data and sampling rates. The NILM runs a two-stage preprocessor that consolidates input data from diverse source streams into a single time stamped array which makes it easier to write energy processing algorithms.

4.5.1.1 Multistream Wrapper

Data streams may be electrical measurements, data from secondary sensors, or outputs from other NILM processes. For processes that require inputs from multiple streams care must be taken to schedule the process appropriately and only run it over time intervals where all of its input streams are available. Sensor data may arrive in bursts with significant lag, and streams produced by other processes create scheduling dependencies. The multistream wrapper manages these dependencies and ensures that a process is only run over intervals where its inputs are available.

4.5.1.2 Resampler

Once the input streams have been assembled, the `resampler` produces a single composite data set with timestamped rows where each column is a process input. If all inputs come from a common source stream then this array is straightforward to assemble, however apps using inputs from different streams generally require resampling. For example, an app that uses outside temperature and real power consumption as inputs (to compute energy usage as a function of weather for example) must use either down-sampled energy measurements or up-sampled temperature measurements. When multiple streams are used as inputs the user specifies a “master” stream and `resampler` runs a linear interpolator or a decimator on the other inputs to create a uniformly sampled dataset.

The stream iterator framework makes it easy to write custom applications that use this dataset to run analysis and control algorithms.

4.5.2 Stream Iterators

Each “energy app” is based around a stream iterator which enables computation on large NILM datasets. Traditional iterators such as `for` and `while` loops operate on static datasets, but NILM data arrives continuously. Stream iterators provide the ability to operate on continuous datasets by combining a traditional looping iterator with a persistent state. When the stream iterator has finished processing the available data it saves its state variables so that when it runs on the next chunk of data, it can pick up exactly where it left off. This allows the programmer to treat the datasets as continuous streams while giving the NILM flexibility to choose chunk size and processing rate based on the available system resources.

Listing 4-1: The setup function for a NILM stream iterator

```
a = [...]; b = [...] #filter coeffs
def setup(state):
    zi = [...] #initial state for filter
    state.initializeSlot("filter_zi", zi)
```

Building a stream iterator is a two step process. First, the user defines a `setup`

function (see Listing 4-1). This function initializes a **state** object which provides persistent storage between process runs. Data is stored in slots which are accessed by string identifiers, similar to a dictionary. This function only runs the first time the app is executed. The setup function in Listing 4-1 initializes **state** for an example app which runs a linear filter on a NILM data stream. The filter coefficients do not need to be stored in **state** because they are constants which do not change between runs of the process.

Listing 4-2: The run function for a NILM stream iterator

```
#data is 2 column array: [timestamp, sample]
def run(data, state, insert):
    #initialize filter with saved zi values
    zi = state.retrieveSlot("filter_zi")
    #run filter against this chunk of data
    (y,zf)=scipy.signal.lfilter(b,a,data[:,1],zi=zi)
    data[:,1]=y #update data in place
    insert(data) #save output
    state.saveSlot(zi,"filter_zi") #update zi
```

After initializing the app state in **setup**, the user then defines a **run** function. This function receives the resampled input streams from the preprocessor and performs the actual data processing (see Listing 4-2). In this function traditional iterators and third party libraries can be used to build complex signal processing algorithms. Listing 4-2 shows a simple example which runs a linear filter using SciPy, an open source Python library. More advanced code could perform load identification, equipment diagnostics or a variety of other data analysis. The **insert** argument is a function handle for saving results to an output data stream. After processing the input, any variables that should persist between runs are stored in the **state** object. The NILM repeatedly runs this function as more input data becomes available.

4.5.3 Reports

In addition to generating output data streams (such as the filter example in Listings 4-1 and 4-2), “energy apps” can also produce reports. Reports run over a specific interval of data and produce an HTML document that can contain custom text, plots, and tables. After the stream iterator has processed the specified duration of

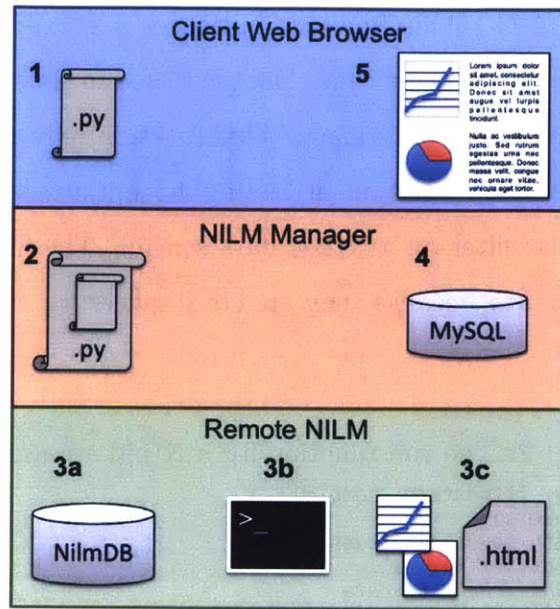


Figure 4-6: The stages of a NILM report. 1: The end user designs a report in the Web IDE. 2: The management node adds support code to build an executable script which it then sends to the remote NILM. 3a-b: The script links to streams in the NilmDB, and runs to completion. 3c: The HTML report and associated figures are sent back to management node. 4: The management node stores the report in its MySQL database. 5: Authorized users can view the report in their web browser.

data (eg: hour, day, week, etc.), an HTML generator produces the report document. A report is defined by an analysis function and an HTML template. The analysis function uses the process `state` to compute summary statistics and figures. These are injected into the report template to create a full HTML document.

Figure 4-6 shows the process of creating an energy app report. In step 1, the user defines the stream iterator, analysis function, and HTML template. Next, the management node adds the support code making an executable script which is sent to the target NILM. In steps 3a-3c the NILM runs the energy app which generates an HTML document and associated figures. The NILM returns these files to the management which stores them in a MySQL database and makes them accessible through the web interface to authorized users. Hosting the report document on the management node rather than the NILM insulates the NILM from external network traffic providing an additional layer of security and reducing the demand for its limited bandwidth. If privacy is a greater concern than network bandwidth, the NILM can

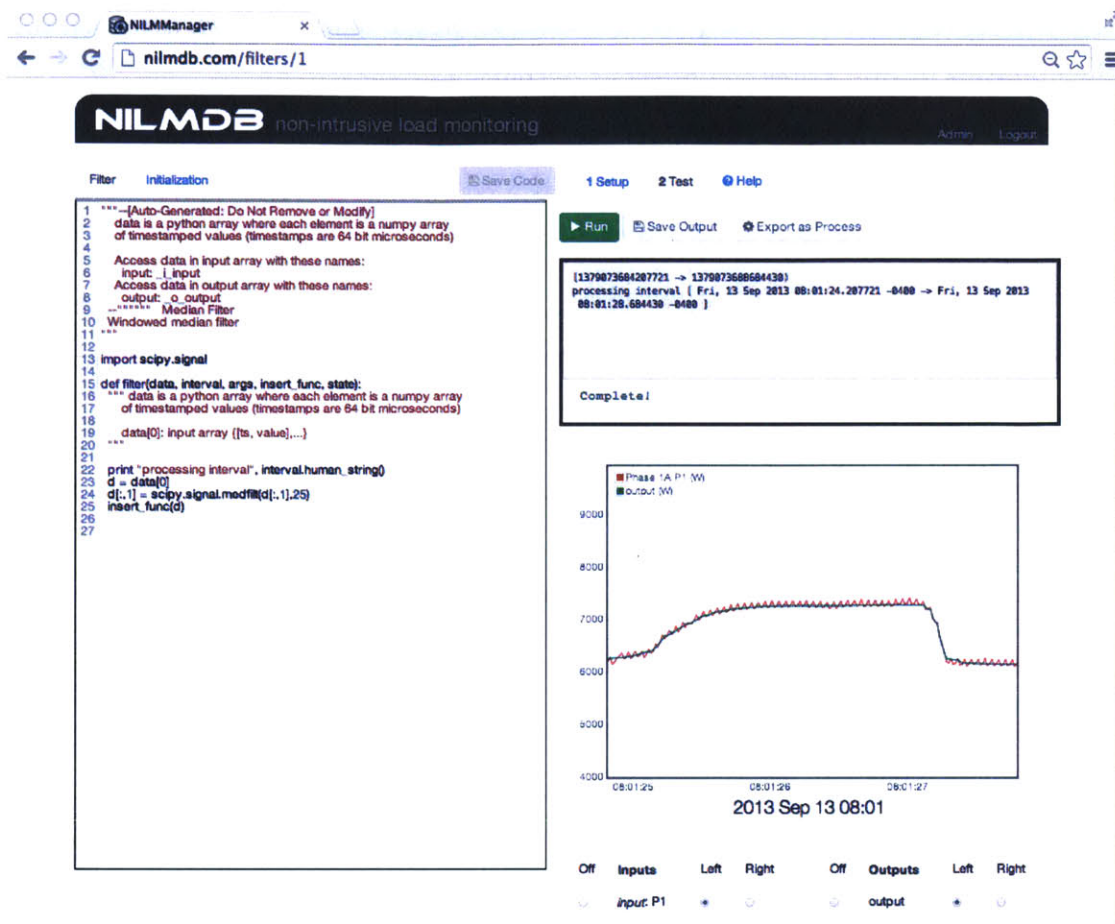


Figure 4-7: NILM Manager IDE for designing Energy Apps

retain the report in local storage instead.

4.5.4 NILM Manager IDE

The NILM Manager website provides a complete integrated development environment (IDE) to write, test, and deploy “energy apps”. Figure 4-7 shows the app designer interface. The left hand panel is a syntax highlighting code editor with multiple tabs for app initialization, stream iterator definitions, and report templates. To run the app in development mode the user selects input streams and a time range using the plotting window on the bottom right. Text output generated by the app is continuously retrieved from the NILM and displayed in the upper right hand panel. This panel also displays debugging information in the event of an error. In development

mode the output stream is temporarily allocated on the remote NILM, and each time the app runs, it overwrites the previous output.

Once the user is satisfied with the app’s performance, the code can be deployed to one or more target NILMs and scheduled as a continuous process. The management node tracks deployed processes and archives system logs and metrics so that users can manage the computational resources on their NILMs appropriately. When an app is deployed in production mode its output stream is permanently allocated on the target NILM and made available to other users either to plot or use as an input to other apps.

4.6 Designing Energy Apps

Users with appropriate security permissions can now design useful applications on the NILM to monitor and control their power systems. “Energy apps” run entirely on the NILM itself and do not rely on external services or high bandwidth network connections. The following example shows examples of how these apps are designed at real monitoring sites.

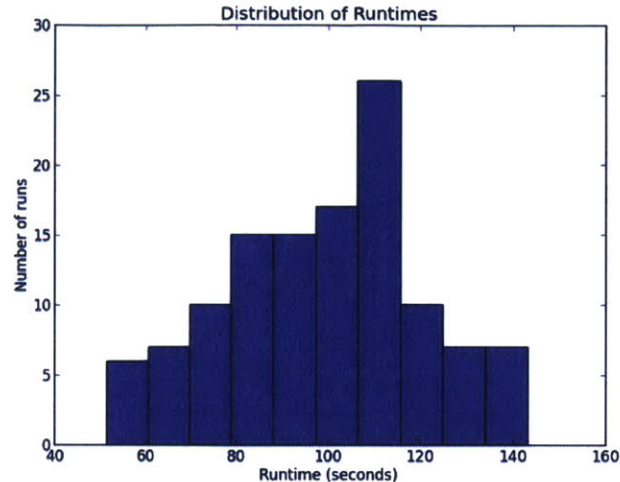
4.6.1 Cycling System App

Reports present actionable information to end users turning NILMs into powerful monitoring and diagnostic tools. Consider a standard cycling system such as a shop air compressor. This system requires periodic maintenance based on hours of operation and excessive runtimes may indicate leaks or abnormal usage, but adding sensors to track air compressor runs is generally too expensive for the benefit it provides. NILM is the cost effective solution. A single NILM can monitor multiple air compressors, and indeed any electric machine in a shop, eliminating costly (and maintenance-prone) sensor networks [65].

Figure 4-8 shows an example of a report for tracking trends in air compressor runtime. The report is built in two stages. First a stream iterator, defined by `setup` and `run` functions, processes data over a specified time interval. The stream iterator

Report for Air Compressor

120 runs for a total operating time of 197 minutes



Machine status: maintenance required

Figure 4-8: Example of a NILM report for monitoring an air compressor (generated by Listing 4-4)

identifies machine turn on and turn off events by tracking transients in the power waveform. When a machinery run is detected it is added to an array stored in the process `state`.

After the stream iterator processes the data, the analysis function in Listing 4-3 generates summary statistics and builds a histogram of machine runtimes. The statistics are added to the process `state`, and the `saveFigure` function saves the plot using a similar string-tag syntax.

Finally the HTML generator builds the report using the template shown in Listing 4-4. Markdown is used for simplicity although raw HTML and CSS can be mixed in for finer grained control of the document format. Content from the process `state` is injected into the template using double braces `{{...}}`, and the `insertFigure` command embeds plots as HTML images.

The HTML document and plot image are sent back to the management node and hosted through the web interface. Reports like this example, can be scheduled to

Listing 4-3: The analyze function for a report process

```
def analyze(state, save_fig):
    #retrieve data calculated by [run] function
    runtimes = state.retrieveSlot("runtimes")
    #calculate statistics
    mins = int(np.sum(runtimes)/60) #minutes
    state.initializeSlot("time",mins)
    state.initializeSlot("runs",len(runtimes))
    #if any runtime > 3 hours raise alarm
    if(np.max(runtimes)>180)
        state.initializeSlot("status",
            "maintenance required")
    else
        state.initializeSlot("status","OK")
    #make a histogram of the runtimes
    fig = plt.plot(runtimes)
    save_fig("runtime_histogram",fig)
    #...additional plot formatting not shown
```

Listing 4-4: Template for report HTML

```
Report for Air Compressor
-----
{{runs}} runs for a total
operating time of **{{time}}** minutes

{{insertFigure("hist")}}

####Machine status: {{status}}
```

run once or run continuously. When set for continuous operation the user specifies a repeat interval and duration. For example a report can be set to run every hour using the past 24 hours of data. The web interface provides a navigation tool to browse series of reports which can be help identify trends and spot abnormalities in equipment operation.

4.6.2 Power Quality App

In modern machine shops sensitive devices like CNC tools and 3D printers are co-located with other large equipment that can interfere with the line voltage causing droops and harmonics. In this experiment, a 3D printer shares shop space with a laser cutter and an air compressor, both of which introduce power quality problems including voltage sags. Shop preference is to avoid sharp voltage sags of more than two volts during operation of the 3D printer. A NILM monitors the aggregate current and voltage for the entire shop. Figure 4-9 shows the power consumption of the shop

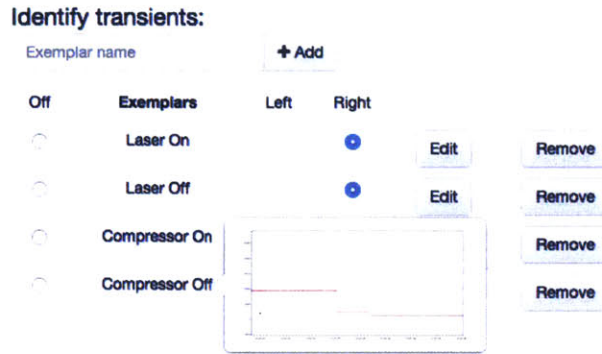


Figure 4-9: Training the load identifier on shop equipment. The cross correlator presented in [8] uses exemplars to identify turn-on/off events of machines. The exemplar for “Compressor Off” is shown in the popup window.

during normal operation. A cross correlator (discussed in [8]) is trained to identify these loads. The turn-on and turn-off events are indicated in the figure by colored bars. Here, four transients are identified corresponding to a run of the laser cutter and air compressor respectively. The lower power cycling waveform is the PWM bed heater of the 3D printer. Energy apps on the NILM can both quantify the shop’s power quality and improve the power quality to the 3D printer during operation by ensuring proper scheduling of the loads.

Over this time interval the NILM detected voltage disturbances large enough to interfere with the 3D printer’s operation. Figure 4-11 shows the power waveform as well as the line voltage as measured by the NILM. The app outlined in Listing 4-5 identifies voltage transients larger than 2V. When such a transient occurs the app checks the machine events identified by the cross correlator to determine which piece of equipment caused the transient. If no events occurred at the time of the transient, the voltage disturbance is due to an external load not monitored by the NILM. Such information can be used to quantify power quality complaints when negotiating with the utility. The bars on Figure 4-11 indicate voltage transients and the colors assign responsibility either a piece of equipment in the machine shop or to the utility in the case of external loads.

While the laser cutter does create a large voltage droop it does so gradually and so does not disturb the printer. The air compressor has much more rapid transients

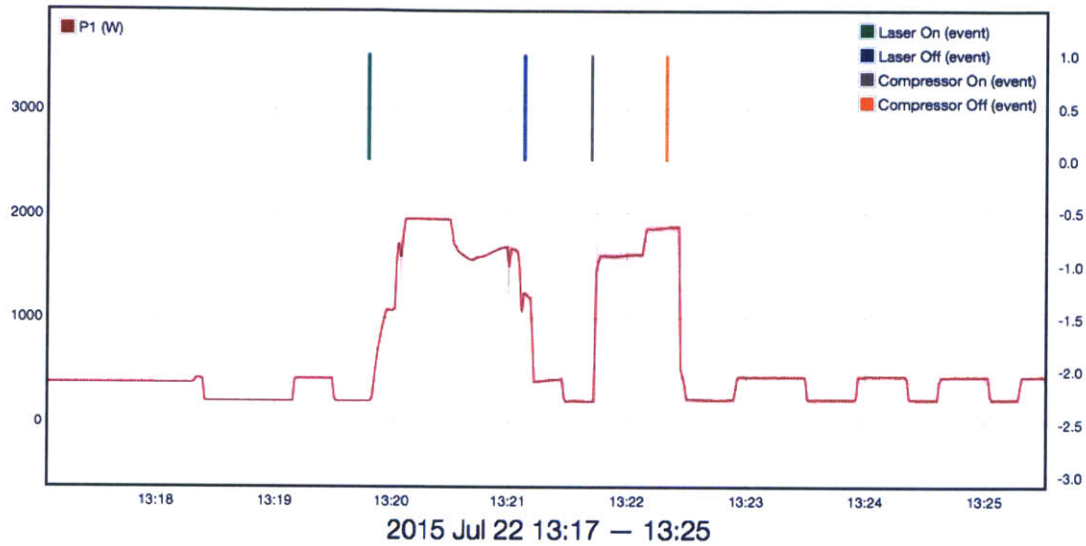


Figure 4-10: Identifying large shop loads. The cross correlator identifies transients of machines in the shop that might interfere with the 3D printer.

Listing 4-5: Energy App pseudo code for identifying and assigning responsibility for voltage transients. `find_equipment_transient` is implemented by the cross correlator trained in Fig 4-9

```
def run(data, state, insert): #pseudo-code
    for dv in diff(volts):
        if(abs(dv)>2): #Voltage transient > 2V
            eqp = find_equipment_transient(dv)
            if(eqp==None): #no equipment turn-on/off
                insert("utility")
            else
                insert(eqp.name)
```

and is identified as an interfering load by the app. There is also a voltage transient that cannot be associated with machines in the shop and the app assigns the transient to the “utility”. In fact this transient was due to a nearby shop vac that, while not physically located in the machine shop, did cause voltage disturbances on the line. This type of disturbance is typical of power quality problems induced by operations “outside” of the facility. The NILM, configured as an energy box, is not only capable of controlling load sequencing within a facility, it is also able to recognize internal versus external power quality offenders.

4.6.3 Adding Control to an Energy App

As desired, energy apps can also control loads directly using smart plugs. Smart plugs connect to a WiFi network and allow remote clients to control an embedded relay to switch a load on or off. These plugs are available from a variety of vendors [66,67] but use proprietary protocols that make them difficult to use outside of their private commercial ecosystem. The plug in Figure 4-12 is a modified Belkin WeMo Insight. The stock Insight only communicates with a smart phone app and provides limited metering capability. We designed a drop-in replacement control PCB that provides the stock functionality as well as persistent storage to an SD Card, a battery backed real time clock, and high bandwidth metering. This plug interfaces directly with the NILM so energy apps can monitor and control individual loads.

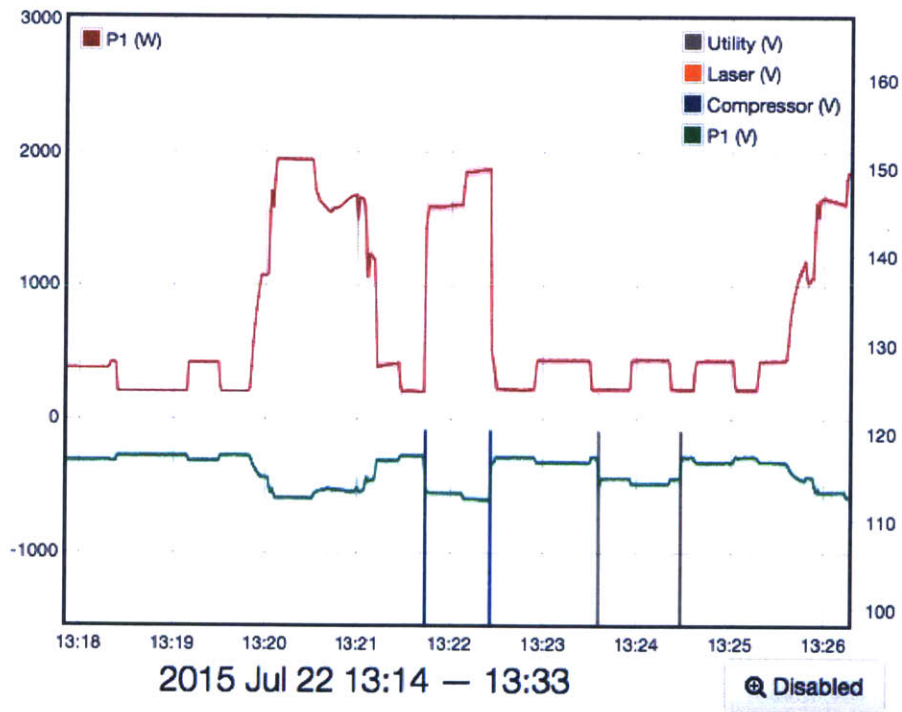
The app outlined in Listing 4-6 uses one of these smart plugs attached to the air compressor to improve the power quality to the 3D printer. When the 3D printer turns on (as detected by the cross correlator), the app turns the air compressor off. When the printer has been inactive for at least `WAIT_TIME` seconds, the compressor is turned back on. The actual compressor runs are determined by a pressure gauge on the machine itself.


```

initializing state
[1437585429669952 -> 1437585933669953]
Air Compressor at: Wed, 22 Jul 2015 13:21:41.652130 -0400
Air Compressor at: Wed, 22 Jul 2015 13:22:19.323992 -0400
Utility at: Wed, 22 Jul 2015 13:22:19.323992 -0400
Utility at: Wed, 22 Jul 2015 13:22:19.323992 -0400

Complete!

```



Off	Inputs	Left	Right	Off	Outputs	Left	Right
<input type="radio"/>	Power [P1]	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	Utility	<input type="radio"/>	<input checked="" type="radio"/>
<input type="radio"/>	Voltage [P1]	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	Laser	<input type="radio"/>	<input checked="" type="radio"/>

Figure 4-11: Identifying the causes of voltage transients. An Energy App correlates machine turn-on/off events with voltage transients. If a transient occurred without a matching machine event, the disturbance is assigned to the utility.

Listing 4-6: Energy app pseudo code that only allows the air compressor to run when the 3D printer is off. The app identifies the 3D printer by the bed heater waveform and only enables the smart plug relay for the air compressor with the bed has been off WAIT_TIME or longer.

```
def run(data, state, insert): #pseudo-code
    if(detect_printer(data)):
        #printer is running: disable the compressor
        set_compressor_relay(OFF)
        last_run = cur_time
    elif(cur_time-last_run > WAIT_TIME):
        #printer has been off at least WAIT_TIME:
        # enable the compressor
        set_compressor_relay(ON)
```

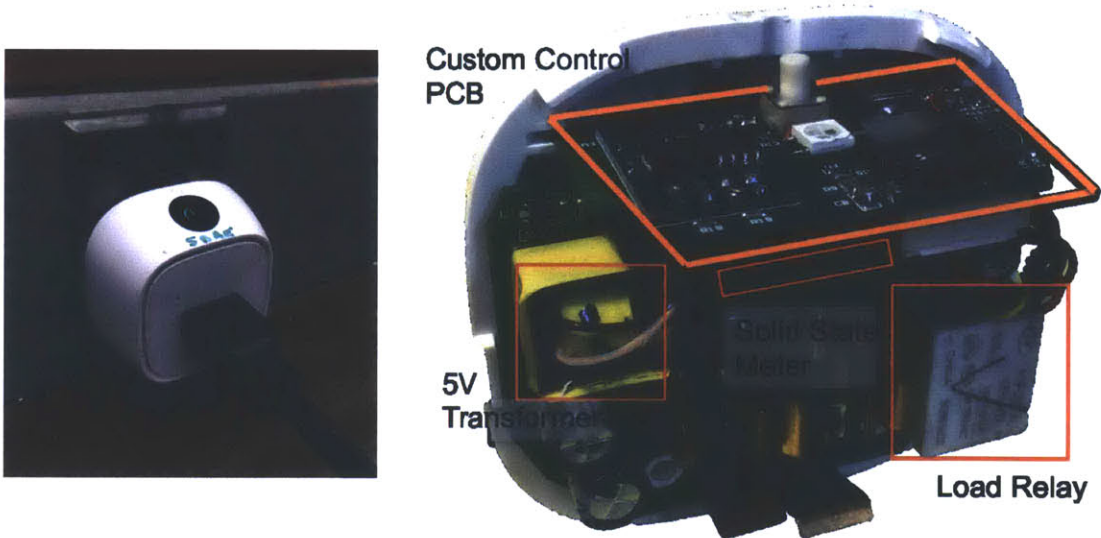


Figure 4-12: Custom smart plugs allow the NILM to monitor and control individual loads. This plug is a commercial Belkin WeMo [66] retrofitted with a custom control PCB. Energy Apps can control the plug relay and read the embedded solid state meter.

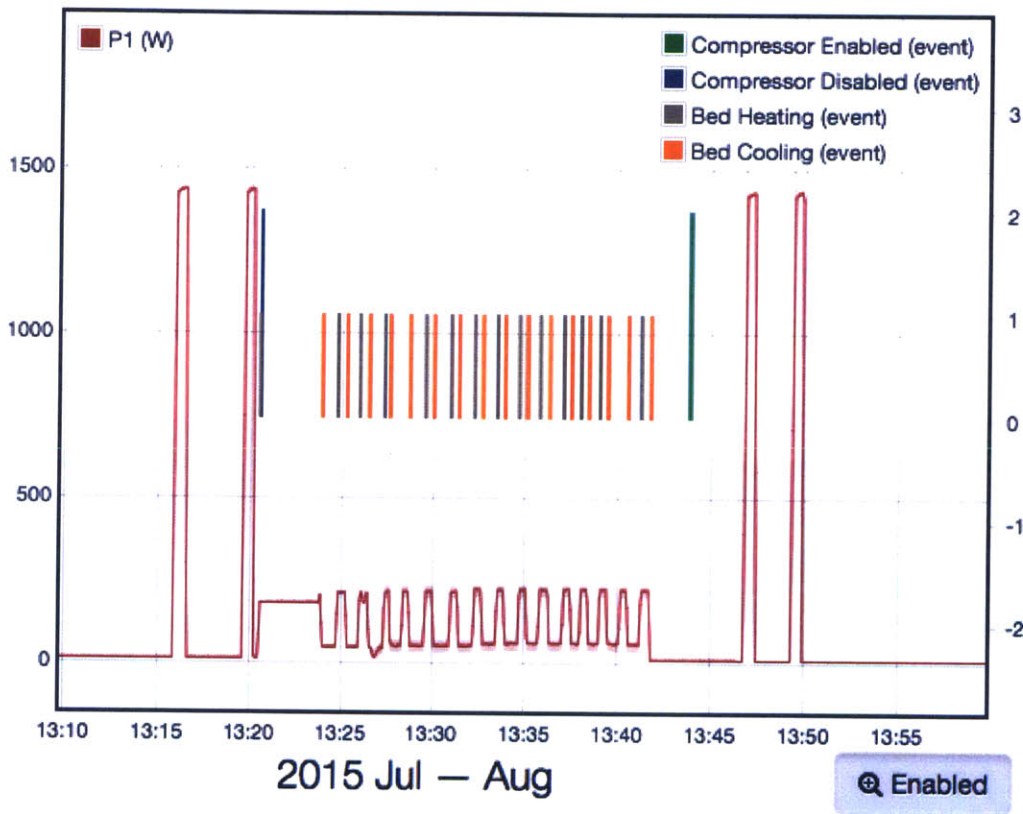


Figure 4-13: The Power Quality App in action: running the code in Listing 4-6 protects the 3D printer by disabling the air compressor during print jobs.

Chapter 5

Case Studies

This chapter explores the real world applications of non-intrusive power monitoring using NILM and the Wattsworth infrastructure. These deployments were done in close collaboration with numerous graduate students and the facilities staff at each of the respective locations. Section 5.1 shows how a NILM can track energy consumption by load and quantify energy savings measures. This work was done with Mark Gillman and James Paris and is covered in greater detail in [68]. Section 5.2 uses a NILM to diagnose configuration problems and equipment failures that are difficult to detect in an operational building. This work was done with William Cotta and Mark Gillman and is covered in detail in [69]. Section 5.3 uses a complete Wattsworth system to provide automatic ship logs to the crew of a Coast Guard cutter. This work was done with Greg Bredariol and is presented in [70].

5.1 Cottage Elementary: Energy Scorekeeping

The Cottage Elementary School in the Sharon School District in Massachusetts has served as a fascinating and representative test bed to demonstrate the NilmDB/NILM Manager approach for monitoring. The school is actively used by hundreds of students and teachers. The load sizes, types, and levels of automation seen here are uncommon to residences. Many of the devices are systems of loads, an extension of multistage loads. The boiler, for instance, has a draft fan, blend pump, actuators,

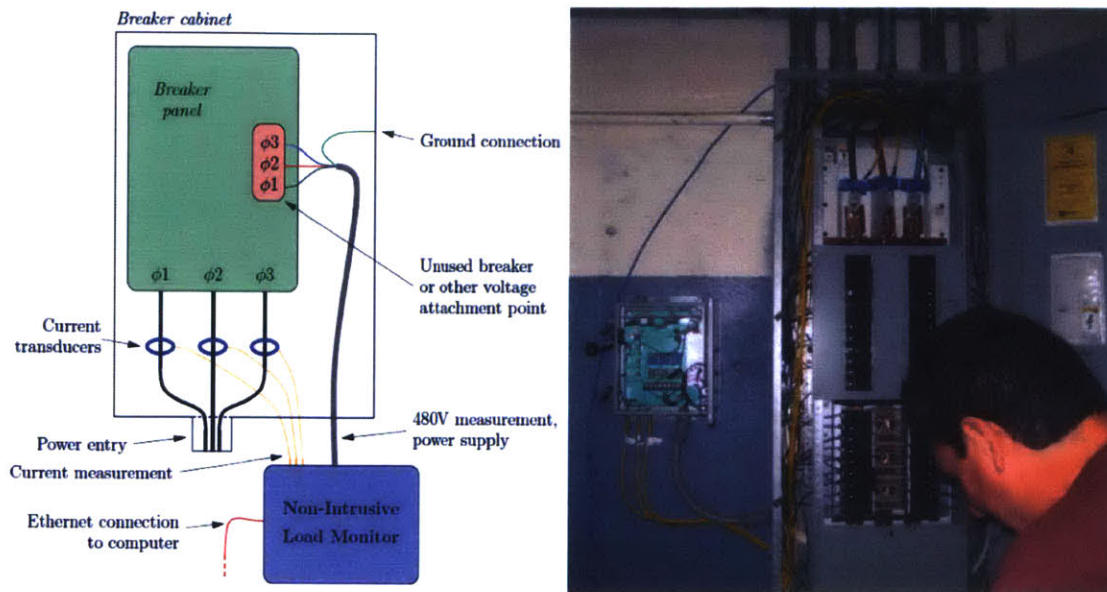


Figure 5-1: Cottage NILM Schematic and Installation

burner controls, and a transformer igniter. Each component has a unique signature and a prescribed sequence of operation in non-pathological operation.

An electrician installed the NILM system using traditional contact voltage and current sensors on a 3-phase subpanel known as the emergency panel (EBPP). Figure 5-1 shows the connection scheme. The EBPP is the critical electrical node servicing the school's communications, heating system, kitchen appliances, septic system, and other important loads. In the event of a power outage, the backup generator supplies power to this panel enabling the school to provide shelter, heat, food, and communication capabilities to the surrounding community. There are more than 30 subpanels at Cottage, but the EBPP accounts for about 1/4th of the school's total electrical power consumption during winter months.

5.1.1 Electrical System Background

In cold weather, the largest power draw on this panel is from the machinery involved in creating and distributing heat. Cottage's heat system is a closed-loop reverse-return hot water system regulated by an integrated building control system (see Figure 5-2). Operation of the heat system depends on several user-established inputs. If the

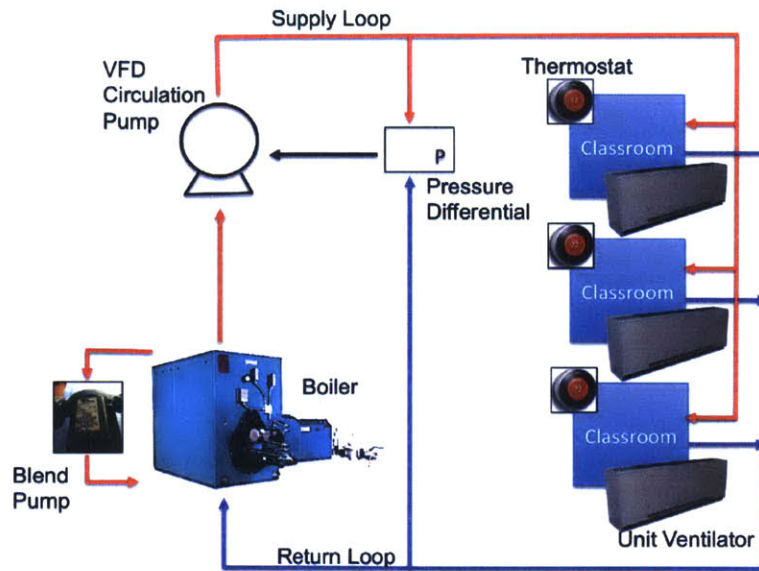


Figure 5-2: Cottage Heating System

outside air temperature is below 55 °F, the boilers will operate according to water temperature settings in the loop. If the return-loop temperature is below 170 °F, the boilers will operate until it reaches 185 °F. To prevent cracking inside the boiler, a blend pump mixes return water with supply water. Cottage’s boilers heat water using natural gas, but the electrical signatures of the draft fan and blend pump are detectable during operation. The Variable Frequency Drive (VFD) circulation pumps pressurize the supply loop and move the water through the piping system to the school. The VFD operational speed depends on system pressure. Head pressure, like voltage, maintains the desired flow inside the system. Upper and lower limits are set and measured by the pressure differential. When the pressure is too low or too high, the pumps will speed up or slow down by increasing or reducing voltage frequency.

Cottage’s emergency panel has many other loads unrelated to the heat system, including the IT Room, hot water pumps, large kitchen appliances, etc. These are listed by circuit-breaker number in Figure 5-3. The minimum and maximum kW values correspond to the range of their unloaded and loaded power draw. Some devices, including lights, are frequently on or off, while others continuously operate.

Breaker	Load	Min (kW)	Max (kW)	Breaker	Load	Min (kW)	Max (kW)
1	IT Room:			2	septic pumps	0	1.3
	Extreme	0.14		4	septic pumps	0	1.3
	UPS	0.06		6	septic pumps	0.02	1.32
3	IT Room:			8	VFD circ pumps	0	1.5
	Phones	0.09		10	VFD circ pumps	0	1.5
	SMC	0.07		12	VFD circ pumps	0	1.5
	SMC	0.07		14	freezer	0	
	Sonic Wall Video	0.1		16	chair lift	0	
	AC Pump	0	0.06	18	Make Up Air Fan	0	0.28
	UPS	0.045		20	Make Up Air Fan	0	0.28
5	IT Room:			22	Make Up Air Fan	0	0.28
	cable amplifier	0.025		24	fire protection	0	
	PA/clocks	0.16		26	heat control	0.28	
	desktop CPU	0.07		28		0	
	apple CPU	0.025		30	elev rm lights	0	
	server	0.07		32	elev rm outlets	0.07	
	UPS	0.21		34	boiler rm lights	0	
	UPS	0.13		36	boiler rm outlets	0	
7	IT Room			38	sump pump	0	
9	unknown off/on a lot	0	1.37	40	sump pump	0.05	
13	unknown 208V	0.9	0.9	42	unknown	0.09	0.56
15	unknown 208V	0.25	0.25				
21	security	0					
23	generator controls	0					
25	freezer	1.2	1.2				
27	unknown ~17 min run time	0.05	0.6				
29	Boiler System:						
	Boiler 1 Draft Fan	0	0.86				
	Boiler 2 Draft Fan	0	0.74				
	Transformer 1						
	Low-flame solenoids 1						
	Transformer 2						
	Low-flame solenoids 2						
	High-flame solenoid 1						
	High-flame solenoid 2						
control	0.27						
31	Boiler:						
	Boiler 1 Blend Pump	0	0.35				
	Boiler 2 Blend Pump	0	0.54				
	boiler control	0.13	0.13				
33	Y-pump Kitchen	0.11	0.11				
35	R-pump	0	0.07				
37	Y-pump School	0.11	0.11				

Legend
Always On
Baseline Load

Figure 5-3: Loads monitored by NILM at Cottage Elementary School

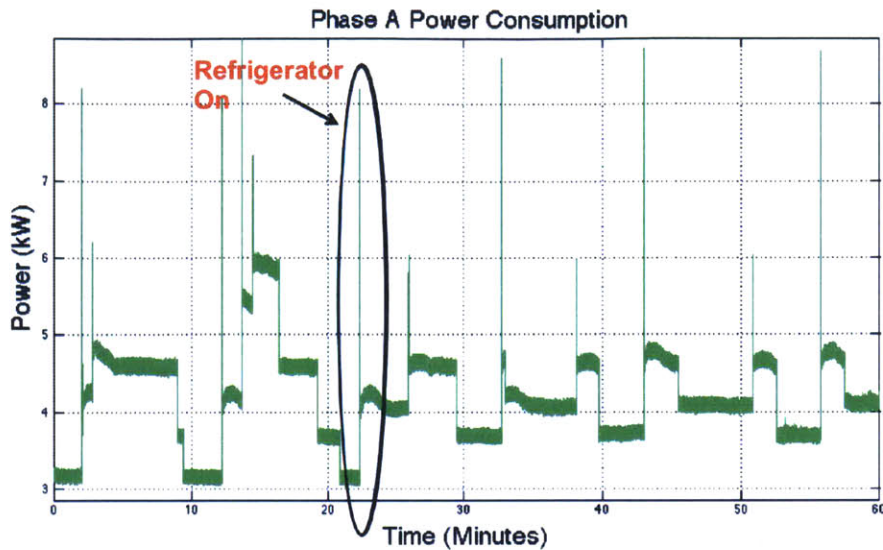


Figure 5-4: One hour of real power data at Cottage. A refrigerator turning on is one of several events during this period

Others quietly consume power keeping their internal systems running on standby, even when not in full use. Also of note is the number of 3-phase loads, indicated by multiple breakers with the same label, such as the Make-Up Air fan in the kitchen. The total draw from such loads is the sum of the power drawn on each phase. For instance, the circulation pumps are 3-phase VFD motors drawing a maximum of 1.5 kW per phase, or 4.5kW total. Other loads in the building create a base load present on the panel electrical phases. Loads not of interest for tracking the heat system, to include smaller pumps and much of the electronic equipment in the IT room, were purposely set aside to draw attention to the larger, more energy-consuming equipment.

5.1.2 Load Disaggregation

Power signals at a central point are simply the sum of each individual load's power draw. As an example, Figure 5-4 depicts one hour of the collective power signal on phase A on Monday, March 26, 2013, from 12:00-1:00 PM. Two Python programming scripts developed for Cottage filtered the preprocessed data. The following figures demonstrate how the filters decompose this signal into its individual loads. With the

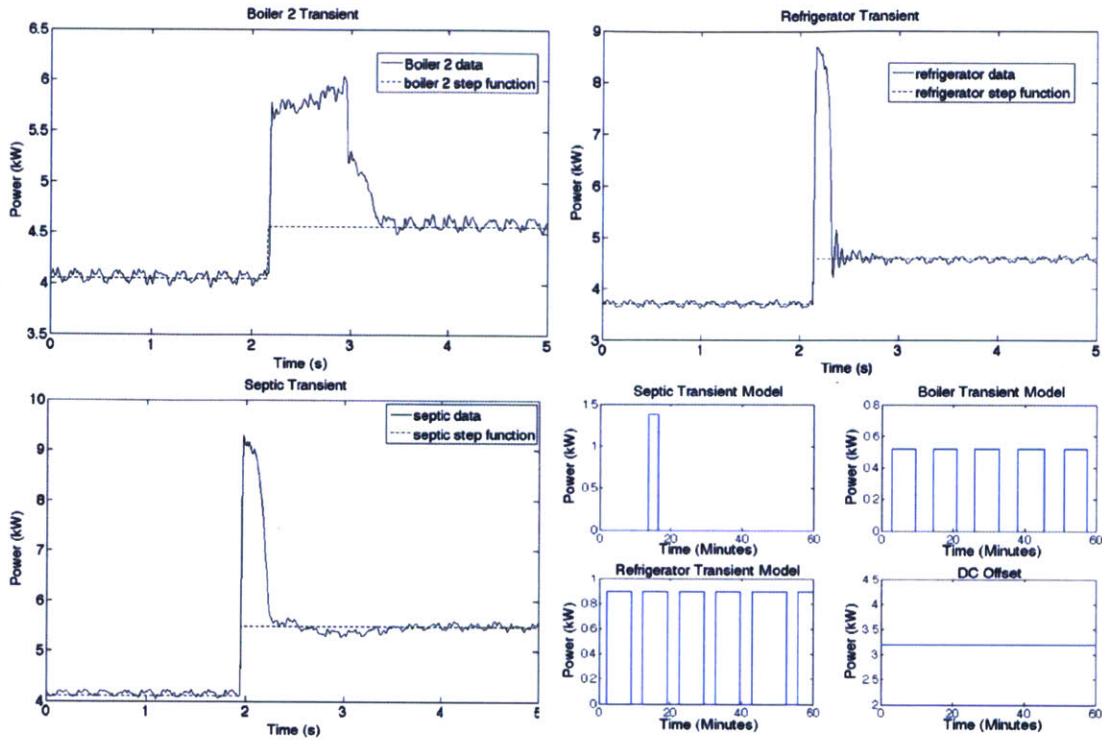


Figure 5-5: Load transients modeled with step functions

DC offset removed, each transient is then modeled with a step function, the superposition of which effectively reconstructs the original signal. The filtering software detected 23 transients during this hour. Two key features are apparent from the graph: the transients and the baseline. One refrigerator transient is circled. The baseline, about 3.2 kW, is the power draw of all machines that remained on for the entire hour. Graphically, it is the low point on the plot. From Figure 5-5, the other phase-A loads that make up the baseline are the Freezer, Make-Up Air Unit, circulation pumps, and several smaller loads (control equipment, communications equipment, etc.).

Step functions with a magnitude equal to the average delta kW values for each device were used to model the changes in steady state power level as loads activated. Actual turn-on transients for most machines are not clean step functions but in fact vary according to the physical task it performs [71]. In Cottage, most loads were distinguishable using relatively simple characterizations of the “load transient,” i.e., just the change in steady power consumption.

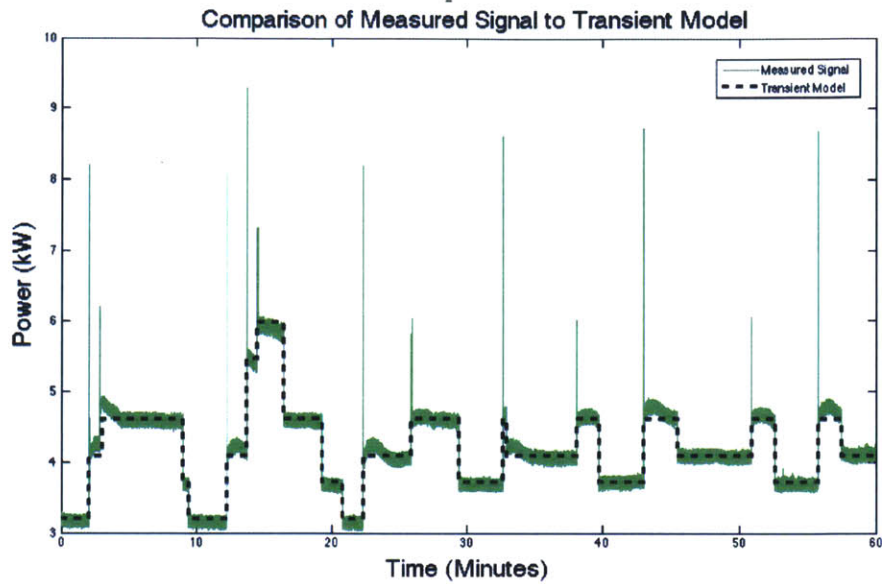


Figure 5-6: Comparison of original signal with modeled signal

For example, see Figure 5-5. The Boiler Pump must physically move water that is initially static and thus requires more force at first to overcome inertia. As more laminar flow is reached, however, the power requirements on the pump quickly approach steady state operation. Inrush current peaks at about 2 kW for fractions of a second. Power fluctuates for another few milliseconds before leveling off at a level that is somewhere in the range of 0.46 -0.58 kW at steady state. The other transients follow similar patterns for moving air and sewage. Given that these transients each reach a quasi-steady state within a few seconds and given that the operating durations are on the order of minutes, the step function is a good approximation (less than 5% error) to use to determine kWh consumed. Recalling that real power consumption is the area under the power curve, the turn-on and turn-off transients disclose the duration of each machine's operation. Using this logic, the boiler 1 pump can be modeled using a step function of ± 0.51 kW, the refrigerator ± 0.89 kW, and the septic pump ± 1.38 kW. The baseline, or the DC offset, is about 3.2 kW. Each machine's operation over this one-hour period, graphed separately, is shown in the bottom right corner of Figure 5-5.

The superposition of these three individual transient models closely approximates

the original power signal (Figure 5-6), validating the efficacy of this modeling method. Once the edges can be detected, named, and kWh can be approximated, we can then keep score of each machine's activity and cost.

5.1.3 Case Study Results

The NILM system detected some 5100 events over a period of 6 winter days in 2013 (11-13 March and 24-26 March). Peak hours featured more than 50 events, while the minimum number in a one-hour period was 18. Software corroborated the results. When events were classified, flags were raised if the same machine turned on twice without turning off in between. All such errors were checked graphically. In total, more than 98% of the events were classified without error. Most often, the issue was simultaneous events. A few events were also missed because they occurred too close to the hour (within a few samples). This issue has been remedied for future experiments by eliminating the one-hour file sizes, opting instead to concatenate stored file segments into one large file. With the errors visually corrected, daily run times, cycle durations, and power consumption costs were tallied based off of the NILM output. The results are shown in Figure 5-7. From the monthly power bill, Cottage paid just over 9 cents per kWh to the utility company.

The heat system represents the highest cost on the EBPP, more than \$10 per day. It is made up of the 3-phase circulation pumps and the boilers. Broken down into its subsystems the largest single loads are the circulation pumps. One VFD pump is always on while the heating system is on, though the speed and thus power draw fluctuates. From recorded data, these pumps consume, as a rough average, 1.3 kW per phase costing over \$8 per day. Combined, the creation and transmission of heat represented almost 11% of the monthly bill in March. Note that this does not include the contribution of the uni-vents in all of the classrooms that distribute the heat to the tenants.

In this experiment, NILM showed promise as a plausible sensor for natural gas sub-metering. Since the burner specifications and boiler hours-of-operation are known, then the amount of natural gas consumed by the boilers is estimable. Therefore

	Boiler 1 Pump	Boiler 1 Fan	Boiler 2 Pump	Boiler 2 Fan	Circulation Pumps
Times used per day	15.3	15.3	63.5	63.5	1.0
Duration ON per cycle (min)	9.7	9.7	7.2	7.2	1440.0
Duration ON per day (min)	289.7	289.7	544.2	544.2	1440.0
Daily Cost	\$0.15	\$0.37	\$0.44	\$0.63	\$8.71

	Head End Septic	Head End Room	Refrig- Freezer	Refrig- erator	Unknown	Make-Up Air
Times used per day	2.5	1.0	120.0	69.7	19.2	0.8
Duration ON per cycle (min)	2.7	1440.0	8.0	16.2	21.9	420.0
Duration ON per day (min)	6.7	1440.0	957.3	1075.8	412.2	478.7
Daily Cost	\$0.04	\$2.61	\$1.23	\$2.11	\$0.31	\$0.62

Figure 5-7: Average Usage and Cost per machine over 6-day period

even though natural gas is not sub-metered at Cottage the gas consumption can be estimated. Using this method as an estimate, the Gas utility billed the school for 6554 ccf during the month monitored. Using data from the six-day period highlighted above, the combined (both boilers) average run-time is 8 minutes and 25 seconds per cycle. This is the duration that the draft fan is operating. From the burner manual (Gordon-Piatt R-8 Model), the first 90 seconds (on a timer) of fan time purges the system. No gas flows into the boiler. For the next 10 seconds afterwards, low-flow gas is injected into the burner to facilitate ignition. Considering only the high gas consumption time, there are 6.75 minutes per cycle. From the results, the boilers run an average of 90 cycles per day, which equates to 10.1 hours of high-gas operation time per day. The firing rate of the burner is 2136 MBH according to the data plate, which represents the maximum numbers of BTUs per hour through the burner. Thus, the total number of MBTUs per month is

$$2136\text{MBTU/hr} \times 10.1\text{hr/day} \times 30\text{day/month} = 647.208 \times 10^6 \text{ MBTU/month} \quad (5.1)$$

From the utility statement, the gas conversion rate is 1 cf = 1.02 MBTU. Converting the MBTUs to cf, we estimate the monthly gas consumption of the boilers during this month to be 6345 cf, which closely resembles the 6554 ccf utility bill. There are other gas appliances whose combined capacity is about 25% of a boiler burner, but

it is interesting to note that this estimate produces a close approximation and merits further study.

Another result made possible by the NILM is a comparison between the boilers. Each has two main electrical components, a draft fan motor and a blend pump. The make and model of the two draft fans are dissimilar between boilers. The blend pumps also differ in model. It is common practice to set unoccupied times on building such as this school. It allows the school to maintain a colder temperature during off hours. As the the temperature dips lower more energy is required to warm the building back up the next morning. This ramp-up period was monitored closely so that a comparison between the boilers could be made. Boiler use is frequently alternated between Boiler 1 and 2 for maintenance purposes. On March 11th, Boiler 1 operated alone from midnight to 8AM. On March 12th and 13th, Boiler 2 ran alone during the same time frame. The temperature profiles for those days being similar (lows of 36, 38, and 32 degrees, respectively), we determined that while Boiler 1's blend pump uses 20% less power, the draft fan uses 50% more power when running. The duration times of operation varied drastically, with Boiler 1 staying on nearly 2 hours longer to create (presumably) the same amount of heat. Their operation profile differed as well. Boiler 1 ran 15 times with an average duration of about 22 minutes compared to Boiler 2, which ran 26 and 29 times on consecutive days, respectively. Figure 5-8 contains a summary of their head-to-head statistics, revealing that Boiler 1 is about 22% more expensive to operate than Boiler 2 and also puts more about 28% more hours on the machinery for comparable work.

The NILM also measured the effect of a major change to the system. On 25 March, the weather turned warmer. This led to complaints from the teachers about the heat in the rooms. In response, the maintenance technicians throttled all heat valves remotely from their central control station. A corresponding three-phase power reduction of 2.7 kW was observed instantly (Figure 5-9). Because the VFD pumps are pressure controlled, a sudden decrease in demand caused an increase in pressure, and the active pump responded by slowing down significantly. This decrease was observed for the remainder of the school day (about 6 hours), only to increase again

Boiler Cost Comparison \$8.49/watt/amp	Boiler 1 11 March	Boiler 2 12 March	Boiler 2 13 March
Draft Fan (Make, Model)	General Elec Model 5KC49N	Marathon Elec Model EPL 56B34D202BEP	
hp	1.5	1.5	
FLA (115V)	9.2	6.7	
kW	0.83	0.71	
ON duration (min)	337	262	257
cost	\$0.42	\$0.28	\$0.28
Blend Pump (Make, Model)	Taco Model 0012-F4-1	Taco Model 0012-F4-1	
hp	1/8	1/8	
FLA (115V)	18.4	13.4	
kW	0.33	0.51	
ON duration (min)	337	262	257
cost/day	\$0.17	\$0.20	\$0.20
Total Cost Day	\$0.59	\$0.48	\$0.48

Figure 5-8: Cost comparison of Boiler 1 to Boiler 2

during the evening hours when the weather cooled off and demand again increased. In total, this saved about \$1.50. Knowing the actual savings, rather than relying on assumptions or rumors, empowers the customer with actionable feedback for future decision-making.

The power study uncovered useful information during the training phase as well. First, there are at least 24 loads that are always drawing power, 14 of which are in the Head-End room housing all of the network switches and other communications equipment. Including the Uninterrupted Power Supply (UPS), they draw a collective 1.2 kW at rest (while school is not in session). The UPS was in permanent bypass mode because it was not operating correctly, which the network administrator knew. It was already scheduled for replacement. What was not known was that, even in bypass, the UPS continued to draw about 0.4 kW at a monthly cost of about \$26 just to cool itself and maintain standby posture. Measurements of the total load connected to the UPS also led to the recommendation to reduce the size of the replacement UPS from 10kVA to between 5-8 kVA as their maximum load was less than 2 kW.

The reconstructed model in Figure 5-6 accurately models the original signal, validating NILM's disaggregation method. While the model is visually similar in its basic shape, there are elements of the original that are clearly not in the reconstructed model. First, the power peaks, including their peak amplitudes, are not shown as

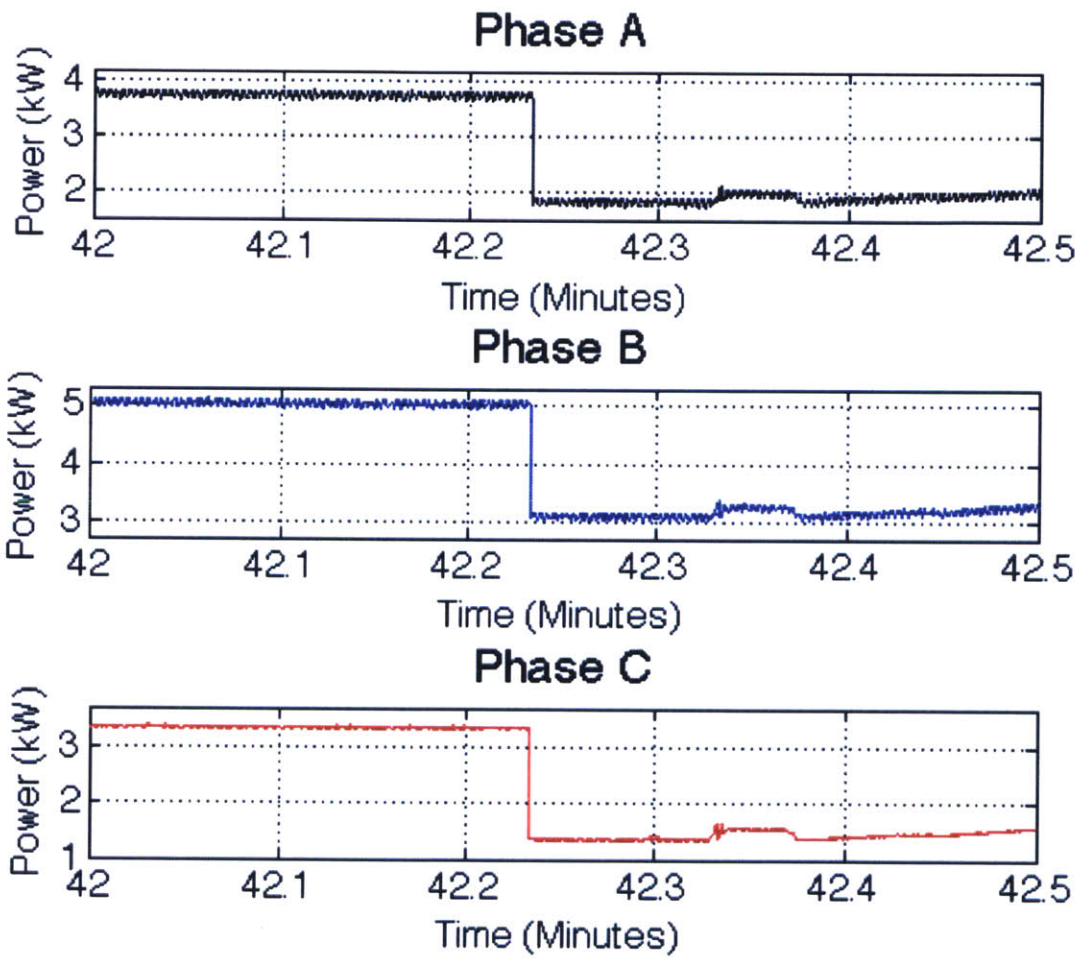


Figure 5-9: Significant three-phase power drop when the VFD demand drops suddenly

explained in Section II. Second, the slow, smooth fluctuations, such as the subtle changes in the variable speed drive, are not accounted for. In general, these represent room for improvement but do not invalidate the approach. While important, the precision of the kWh measurements is secondary to the accuracy of cataloging the individual device patterns from an aggregate feed.

Understanding the details of electrical systems empowers decision makers to make changes without service interruptions or sacrificing environmental comfort levels. Systems like Cottage that employ integrated control systems are commissioned when first emplaced. Over time, as equipment or conditions change, these settings require updates to keep the system optimal. U.S. Department of Energy calls this “continuous commissioning,” or updating system controls over time as conditions change [72]. NILM is able to provide early warning that conditions have changed.

Some limitations became obvious from this experiment. The higher the load count, the higher the likelihood of ambiguous results. Two (or more) loads may turn on, off, or one-on/one-off at the exact same time. Higher sampling frequency could improve resolution, but this would bring the added requirement of more memory. Previous research has advocated collecting all questionable identifications after filtering in order to run “anomaly” algorithms. These make successive comparisons of the anomaly delta kW against both combinations of known transient delta kW and known machine states (on or off) [71]. Also, only changes are visible with the NILM. If loads rarely (or never) cycle, i.e. they are always on, then they are not uniquely distinguishable. The sum of continuous loads comprises the baseline load, which can be discretely determined only by shutting everything off and then back on one at a time.

5.1.4 Implications

The study at Cottage demonstrated a new software architecture for nonintrusive power system monitoring that takes advantage of low-cost *in-situ* or on-site computing. Detailed appliance-level consumption feedback is possible through NILM. The hardware footprint is minimal, a low cost embedded computer and associated sensors. Network bandwidth requirements are very small. A commercial or

industrial setting with or without automation could benefit greatly from the energy scorekeeping provided by the NILM software suite. This field test also demonstrated the ability of the NILM to “derive” details of other utility consumption like natural gas. This directly points out the value of increased local “intelligence” or signal processing in unraveling the “big data” problem associated with consumption feedback and diagnostic monitoring.

The NILM system is uniquely suited for austere electrical networks where loads are standardized. It may be a spectacular tool for assisting with micro grid control and economization. For small or islanded networks, the library of loads can theoretically be pre-set, reducing the extent or possibly eliminating the need for a training phase. In terms of network requirements, the bandwidth required for communication is very tractable. We have already begun to examine this approach in oil refineries, and other possible examples include oilrigs, solar plants, wind farms, industrial parks, and other micro grid installations such as military forward operating bases. In the military’s case, where there is already a mandate to reduce consumption [73,74], accountability is made available quickly and inexpensively.

5.2 US Army: Continuous Commissioning

Current building and facility commissioning methods and maintenance/FDD programs focus on providing top down or bottom up analysis of building systems, respectively. The NILM combines functionality of both methods, and does so without requiring a network of machine sensors typical of conventional monitoring systems. Additionally, the NILM provides a platform which can be further developed. Using only new software, an entirely new suite of monitoring tools can be uploaded to provide additional monitoring functionality. Moreover, the back end database allows comparison of different NILM installations allowing best practices to be analyzed against similar facilities.

Two main site locations were utilized for NILM installations and real-world demonstration of NILM methods for identifying energy inefficiencies. These locations are

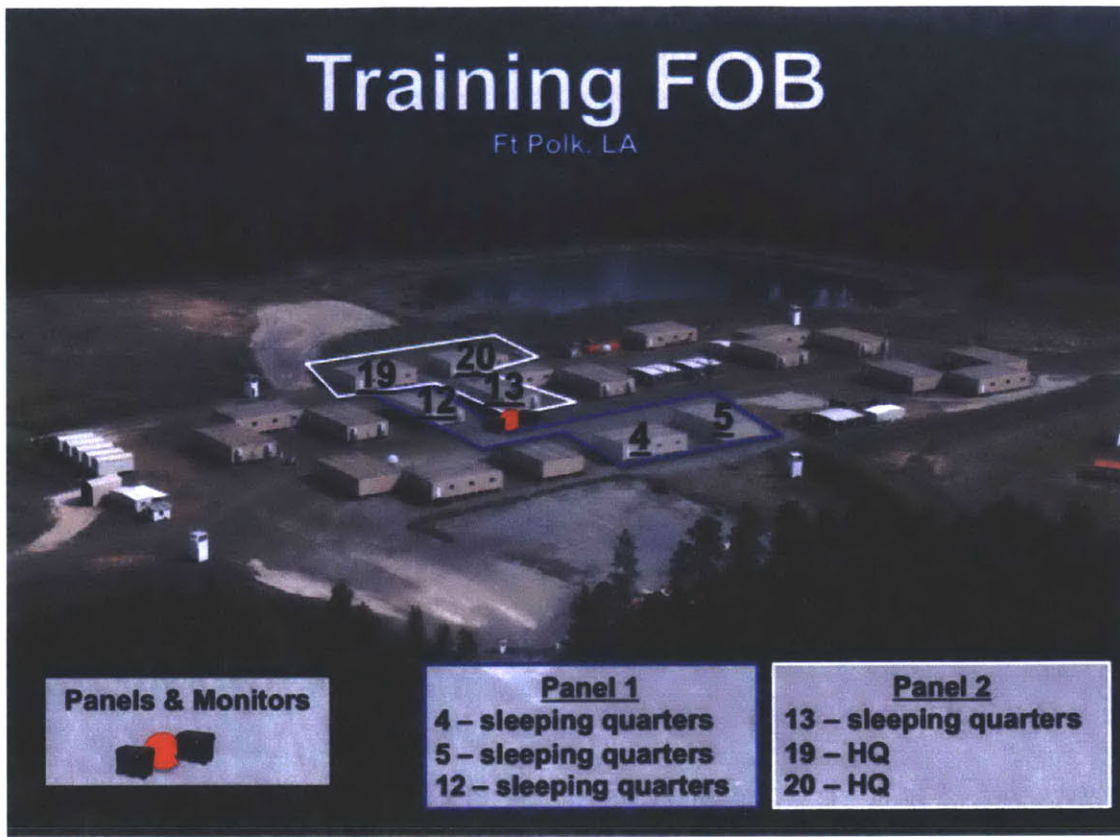


Figure 5-10: An example training FOB located at Fort Polk

described below.

5.2.1 Fort Devens, MA

Fort Devens, MA houses the Army's Base Camp Integration Lab, a test bed for new technology and a training site for Army reserve units. This facility includes climate-controlled insulated tents, latrines, showers, laundry, and electric kitchen and dining facility. This rapidly deployable package, capable of supporting 150 troops, is powered from two three-phase 120 V, 600 A electrical panels, both monitored by NILM systems. The primary power-consuming loads at the base are the HVAC systems, water pumps, kitchen refrigeration units, and lighting systems.

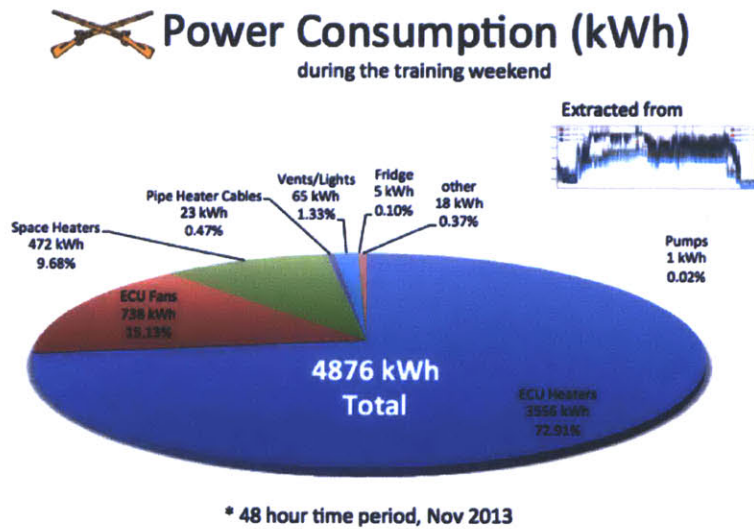


Figure 5-11: Largest loads at Fort Devens itemized by energy consumption

5.2.2 Fort Polk, LA

Fort Polk, LA is an active duty Army post and home to one of the Army’s three Combat Training Centers. This training center has several FOB base camps used to emulate conditions in current theaters of operation. Figure 5-10 shows an aerial view of one FOB camp. At Fort Polk, the FOB electrical service is distributed with each panel receiving feeders directly from pole-mount transformers and typically serving three structures. As such, a single NILM cannot monitor all loads in the camp from the same location. In the interest of relating energy monitoring to occupant usage trends, two panels were monitored: one powering three buildings all used as sleeping quarters (outlined in blue in Figure 5-10), and one powering two headquarter buildings and one sleeping quarter (outlined in white). These two building types are equivalent in size and major loads, i.e. environmental control units (ECUs) and lighting systems, but their usage schedules differ.

5.2.3 Top-Down Monitoring for Energy Savings

Energy saving objectives at U.S. Army FOBs are driven by the high financial and casualty costs of resupply missions [75,76]. However, any implemented energy savings

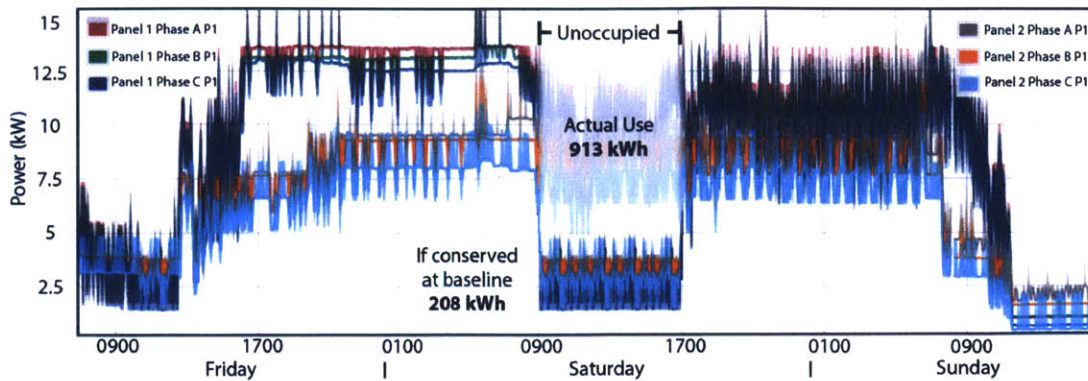


Figure 5-12: Power usage at Fort Devens over an occupied weekend. The shaded area represents unnecessary power usage during a time when the unit was away from the base.

measures are constrained by the necessity to perform critical mission tasks. As such, the data collected at the Fort Devens and Fort Polk FOB test sites were used to test top-down approaches towards ongoing commissioning. Specifically, these approaches were designed to provide a local commander whole-system and disaggregated energy usage data for easy comparison with unit schedules and building occupancies. This sort of information allows the commander to make informed decisions aimed at reducing energy expenditures through changes in human activity without sacrificing the unit’s ability to perform critical tasks or use critical equipment.

5.2.3.1 HVAC Operation Schedule

A first example comes from Fort Devens, MA, when a 90 person Army unit occupied the base from 08-10 November, 2013, for a weekend of training. The average temperature for the weekend was 40°F, with a high of 55°F and low of 26°F. The unit occupied the FOB continuously for 48 hrs with the exception of a training session conducted on the weapons range from 0900 to 1700. The NILM itemized the power consumption of the largest loads over the weekend as seen in Figure 5-11. 73% of the energy went towards ECU heating coils; an additional 15% went to the supply fans, which circulate air across the heating coils. Overall 88% of the energy cost was attributed to the 11 ECUs. Adding in the smaller space heaters, which include the

window unit air conditioners used in the showers, latrines, and kitchen, 98% of the total cost was attributed to HVAC.

When viewing the base's power usage (Figure 5-12) while considering the unit's schedule, it's clear energy savings measures were not employed when the unit attended training at the weapons range. In fact, the NILM detected heater runs during this time period. Considering the minimally insulated buildings that compose a FOB, this represents a large inefficiency. As depicted in Figure 5-12 by the shaded region, the actual energy use during the unoccupied period was 913 kWh; overlaid during the unoccupied time is the "ideal" power draw, i.e. the estimated power draw of only baseline loads. Based on a similar-temperature day when only these loads were operating, the minimal energy use for the unoccupied 8 hours was estimated as 208 kWh. The difference of 705 kWh over 8 hrs represents 14% of the total energy consumption for the weekend.

5.2.3.2 Misconfigured ECUs

A second example comes from the monitored FOB at Fort Polk. Within the same sleeping quarters two ECUs were set in opposition to each other, with one set to heat mode and the other set to cool mode while the building was unoccupied. The NILM was able to detect the transient pattern of a heater and an air-conditioner each cycling on/off. Figure 5-13 shows the real power streams of the two-phase distribution system for this sleeping quarter over an 8-hour period during which the outside temperature remained relatively constant between 70°F-75°F. In this figure, the transient signatures for the heater and A/C cycling are identified. The NILM reveals a limit-cycle type operation where the the two ECUs "dual," with the heater operating and increasing temperature for approximately 16 minutes and the A/C responding to decrease temperature for about 24 minutes. Thus, every hour during this otherwise idle period represents an energy waste of 4.3 kWh, with the heater drawing around 10kW (5 kW per phase) when on and the compressor about 4 kW (2kW per phase). For the run-time shown in Figure 5-13 alone, user negligence contributed up to 40 kWh of energy inefficiency.

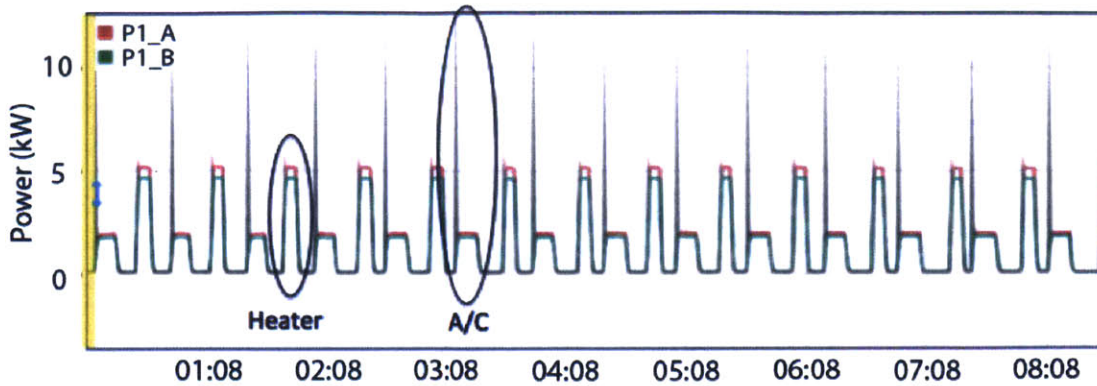


Figure 5-13: Heater and air conditioner in the same room dueling over temperature

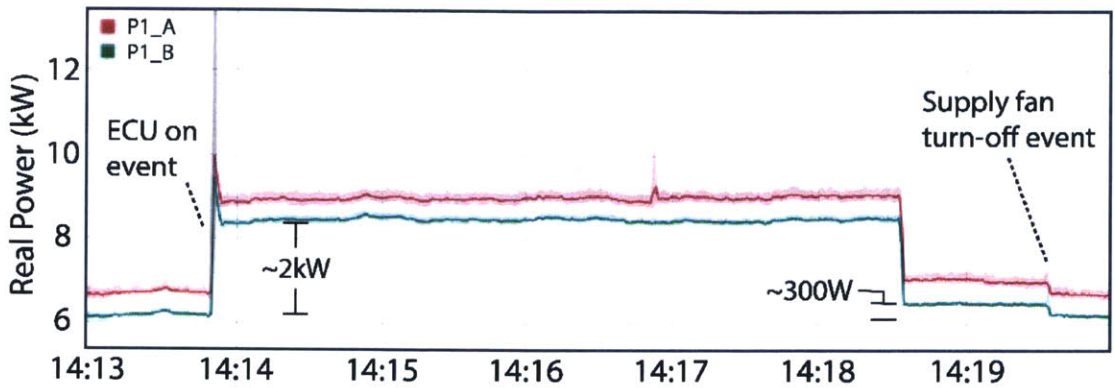
5.2.4 Bottom-Up Fault Detection and Diagnostics

The two top-down monitoring examples of Section 5.2.3 display the utility of NILM in identifying energy efficiency improvements through a reduction in user negligence. In contrast, bottom-up methods are adept at detecting and diagnosing machinery faults (FDD) as monitoring occurs at the subsystem level [77]. Once FDD is performed, the implications on top-level optimizations, e.g. building energy use or occupant comfort, can be assessed to determine the proper course of action.

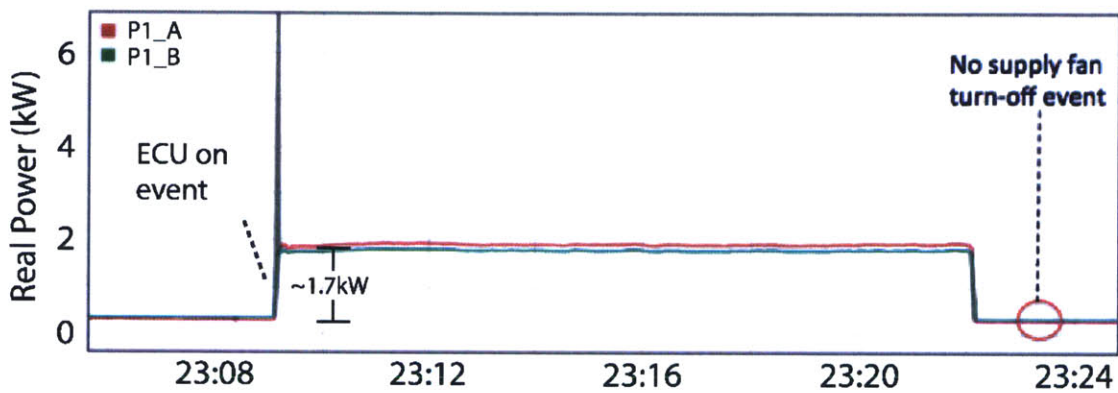
Using conventional monitoring techniques, bottom-up commissioning requires machinery sensors on all systems of interest. However, the NILM affords virtual sub-metering on any load with a detectable transient. This provides the opportunity for bottom-up methods without the complicated network of machine sensors. In the example here, data collected by the NILMs was used to perform FDD on an HVAC system at Fort Polk and to estimate the energy implications of a missed detection.

5.2.4.1 ECU Fault

On the morning of March 28, 2014 at Fort Polk, a wall-mounted ECU faulted when the supply fan ceased to work. This caused a change in the characteristic signature of the ECU unit, and thus appeared as a previously unidentified load on the NILM. Specifically, two deviations from the known on and off transient characteristics oc-



(a) Healthy ECU On/Off Cycle



(b) Broken ECU On/Off Cycle

Figure 5-14: Fort Polk ECU AC mode real power on/off transients for a (a) healthy ECU and (b) broken ECU. The offset between P1_A and P1_B in (a) is due to a simultaneously running single-phase load.

curred, which allowed the NILM users to detect the fault.

Figure 5-14 shows a comparison of full on/off cycle data collected a few days prior to the fault (Figure 5-14a) and just following the fault (Fig. 5-14b). As noted in the figures, a change in the off transient was detected as the fan off signature was missing from all transients following the fault. Further, the power consumed by the ECU after the fault was approximately 600 W (300 W per phase) less than before. Compared to a healthy ECU, the fault resulted in power consumption dropping from about 4 kW (2 kW per phase) to around 3.4 kW. This difference is equal to the consumption of the supply fan.

Without the supply fan circulating the cool air out of the ECU could only cool

the room through unforced ventilation, a significantly less effective method than the forced convective cooling with the fan. For a 5-hour period, the hottest part of the day during March 29th, the broken ECU had a cumulative on-time of 3.1 hours. During that same period, a healthy ECU operating in a different sleeping quarter and at the same temperature set point operated for only 1.4 hours total, less than half the time of the broken ECU. Thus, due to the machinery failure, the broken ECU required almost double the energy of the normally functioning ECU.

Without the NILM, this broken ECU may have gone unnoticed as the room was still cooled to a comfortable temperature, just at a significantly larger energy cost. With the NILM however, the unit commander knows that replacement of the ECU can enact an improvement in long-term FOB energy efficiency.

5.2.5 Implications

Each of these examples represents value-added information for actionable feedback towards improving whole facility energy efficiency. For the unit commanders at Army FOBs, acting on this information results in energy savings translating directly into a reduced demand for fuel convoys, and a concomitant reduction in casualties. For an industrial facility or commercial building manager, the NILMs instead provides energy efficiency insight towards lowering utility bills and increased profits; for the environmentally conscious homeowner, the NILM gives positive reinforcement towards reducing their carbon footprint. Thus, while the Army FOB camps provided the environment for these NILM research and demonstration projects, the monitors, software platforms, and energy-saving commissioning and FDD techniques described here are easily extendable across many sectors and useful for a variety of optimization goals.

5.3 US Coast Guard: Equipment Monitoring

Modern Navy, Coast Guard, and commercial maritime crew sizes continue to shrink as there is a shift to “optimally” and minimally manned crews completing more complex and varied mission sets. Smaller crews rely on sensors and automatic operation to

perform a host of duties once completed manually. This generates a substantial need for monitoring systems to ensure proper operation of equipment and maintain safety at sea. These systems require a significant infrastructure of sensors, wires, and intermediate panels or data collation sites. Because conventional monitoring systems rely on a substantial, distributed hardware installation, communications losses and sensor failures can become commonplace and crippling. Reduced manning may also mean reduced repair hours, and crews with complex but difficult to maintain sensor systems may effectively be left without needed monitoring equipment.

The NILM poses interesting possibilities for shipboard use and maintenance [78]. In particular, the NILM can serve as a “shipboard automatic watchstander” or SAW for tracking machinery operation. This case study demonstrates the operation of a NILM as a watchstander aid during underway operation of USCG SPENCER, a 270 foot Famous-class cutter.

5.3.1 Shipboard Automatic Watchstander

The NILM offers several important benefits to both increasing the precision and automation of Coast Guard watchstanders. First, NILM systems can discern machinery status and automatically generate a log of operation. Second, they can compare machinery operation and sequencing, reflecting the demands of the crew, to a known operational status to relay to decision makers crew fatigue and operational tempo. Third, the NILM can ensure that operational procedures are followed and that automatic functions of machinery are operating as designed to improve or maintain the life of machinery. Because NILM systems require limited access from an aggregate measurement in the power system, they provide a single robust monitoring point that doesn't rely on complex networks of sensors.

5.3.2 Automatic Logging

In the USCG the current method for keeping machinery logs is manual entry. This approach relies heavily on accurate human observation and annotation. Logs are forwarded to maintainers and operational commanders, and accurate log keeping is an

important function for the USCG. Logs are critical for recording operational history and supporting maintenance decisions, and are entered as official legal documents. Generally, watchstanders maintain a “rough log” which is a handwritten document containing times of events and operations including fuel transfers, machinery status changes (starts and stops) and other key events. This is then periodically transferred to a typed document, an overall method that is clearly open to flaws. Watchstanders can be extremely taxed while making normal rounds on equipment. Verifying operation, safety, performing maintenance, training new crew members, and performing casualty response are just some of the watchstander’s normal duties. Accurate log keeping can become an afterthought in stressful or repetitive situations. Manual logging also poses the possibility of distracting the watchstander from the equally important task of monitoring machinery health, possibly allowing for a casualty to go unnoticed.

This case study presents results from a SAW based on nonintrusive monitoring technology for automatically logging start and stop times of machinery operation. This technology reduces the impact of human error and potentially allows human watchstanders to focus on more important and less repetitive tasks. As budget constraints tighten and technologies increase to allow for remote operation of systems as well as increased automation, crew sizes have decreased, in some cases to 50% of manning on legacy assets. Each remaining crew member performs a new multitude of tasks. It is not uncommon for a single crew member to be responsible for monitoring machinery health through frequent rounds, wipe up oil, complete oil viscosity tests, check oil levels, check temperatures, verify pressures, start generators, pump sewage, refill head tanks, and other duties. A SAW creating automatic logs could improve safety and efficiency by decreasing the amount of time a crew member has to spend logging machinery operation manually. Additionally, precision could be increased through logging of exact times and ensuring that no or relatively few events are missed.

Hourly tracking of operations underlies the USCG’s maintenance planning, as many maintenance tasks are based on accumulated operating time, e.g., for major



Figure 5-15: Typical USCG engine room watch. Notice the logbook, termed the “rough log,” used to keep machinery operation times and log critical events. Also note the watchstanders monitoring equipment while simultaneously attempting to maintain logs.

overhauls, oil changes, and other cyclical maintenance. Accurate, automated tracking could greatly improve maintenance planning, decreasing costly corrective maintenance completed after casualties. Also, current systems rely on maintainers receiving aggregated reports from operators on a bi-annual or annual basis. Infrequent information flow easily creates disparities between projected hours and actual operational hours and creates a gap in planning. Automated tracking could ease data collation and access for decision making.

5.3.2.1 A Bellwether for Crew Performance

Certain operations pose increased risk. These “special evolutions” include events such as flight operations, anchoring, maneuvering close to shallow waters, towing, battle quarters, refueling at sea, and law enforcement operations, among other higher risk

evolutions. For these evolutions, vessels institute a condition of operation called the Restricted Maneuvering Doctrine (RMD) that sets additional precautions to mitigate risk. This doctrine is a balance, in that a majority of the crew receive complex or additional duties, creating a potentially fatiguing burden. As crew fatigue increases, the likelihood for mishaps increases. Whenever possible operational commanders should be aware of cumulative time spent at RMD to properly evaluate risk when assigning missions.

There is currently no objective metric to measure fatigue. In the naval community, the USCG employs the GAR model which evaluates crew fatigue on a subjective 1-10 scale. This metric is difficult to evaluate objectively. The aviation community, on the other hand, has a more objective metric, accounting hours of operation and requiring hours of rest [79].

5.3.2.2 Ensuring Compliance with Operating Procedures

When setting RMD, certain pieces of equipment are generally energized and an equipment status is prescribed. Knowing this status and equating it to RMD, a SAW could sense the amount of time a vessel spends at RMD, providing operational commanders a hard metric for crew fatigue when evaluating risk and gain for missions. Also, the SAW can evaluate compliance with prescribed operating procedures and detect deviations in crew performance. Each piece of machinery has a specified or “standard operating procedure” (SOP). These SOP’s contain step by step instructions on machinery alignment and operation. In several systems, the order of operations is extremely important to ensure that catastrophic damage does not occur. For instance, the reverse osmosis feedwater pumps must be started before high pressure pumping commences. Deviations remove cooling and impellers or high pressure pistons could be destroyed. Similarly, diesel engines must be prelubed before starting to ensure lubrication of parts. If a NILM can detect these sequences to prevent or warn an operator when they have missed a step in the sequence, millions of dollars in costly corrective maintenance could be saved across the USCG’s fleet assets. Also, deviations from SOP could potentially serve as an additional indicator of crew fatigue.



Figure 5-16: USCGC SPENCER is a 270ft long Famous Class Cutter whose primary missions include search and rescue, law enforcement, and living marine resources.

5.3.3 Installation on the USCGC SPENCER

To explore these possible uses, a SAW system was installed on the USCGC SPENCER (WMEC-905) shown in Figure 5-16 from November 2014 to December 2014. USCGC SPENCER is a 270ft Famous Class cutter stationed in Boston, MA. Two nonintrusive systems were installed, one on the #2 Main Propulsion Diesel Engine (MPDE) auxiliary supply panel in the engine room, and another on the main exhaust fan for the engine room. These monitoring systems collected data during underway operations to test the potential for automatic event logging and centralized data analysis from the ship power system to classify the status of the vessel and its machinery.

The machinery plant of the cutter consists of three ALCO V-18 propulsion diesel engines and two electric diesel generators rated for 475KW as well as an emergency generator rated for 500KW. It carries a crew of approximately 100 and maintains a rigorous schedule of over 185 days deployed per year [80].

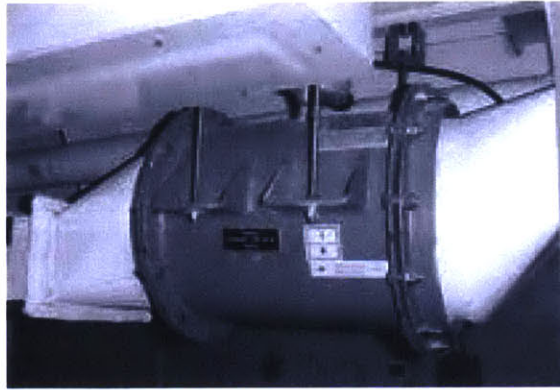


Figure 5-17: Standard Coast Guard exhaust fan.

The installed nonintrusive monitors observed the exhaust fan pictured in Figure 5-17 and the #2 MPDE auxiliary supply panel. Mission critical loads fed from this supply panel include the #2 Main Propulsion Diesel Engine (MPDE) prelube pump, the #2 “C” Controllable Pitch Propeller Pump, the MPDE lube oil heater, and the MPDE Jacket Water Heater. The location and placement of the monitoring boxes can be seen in Figure 5-18. By observing data from these monitors, the machinery plant status can be reconstructed, as shown in the next section. This reconstruction can be compared with required operating procedures. For example, the exhaust fan should be turned on prior to the MPDE starting in order to ventilate the space. The fan is deactivated when the MPDE is not running to keep the engine warm while offline.

The CPP “C” pump motor is a 3 phase motor nominally rated at 440V, 13.6 Amperes, and 10 hp. Figure 5-19 shows the motor. The MPDE prelube pump motor is a 440 V 3 phase 4.9 ampere and 3 hp motor and can be seen in Figure 5-20. The two heaters connected to the system, the MPDE jacket water heater and the MPDE lube oil sump heater, are both 440 V, 3 phase resistive heaters. The JW heater is rated at 9KW and the lube oil sump heater is rated at 12 KW.

The monitored equipment works separately and together to ensure that the MPDE functions correctly. For example, the monitored prelube pump runs continuously when the engine is offline, but does not run while the engine is online. The Lube Oil heater will automatically energize when required and then turn off when temperatures



Figure 5-18: USCGC SPENCER engine room. Nonintrusive monitors are installed forward of the #2 MPDE just above the monitored panel.



Figure 5-19: CPP "C" pump motor that supplies pressure to vary the pitch on the blades of the propeller. These pumps are energized when RMD is set and the vessel enters a time of increased risk and fatigue for the crew.

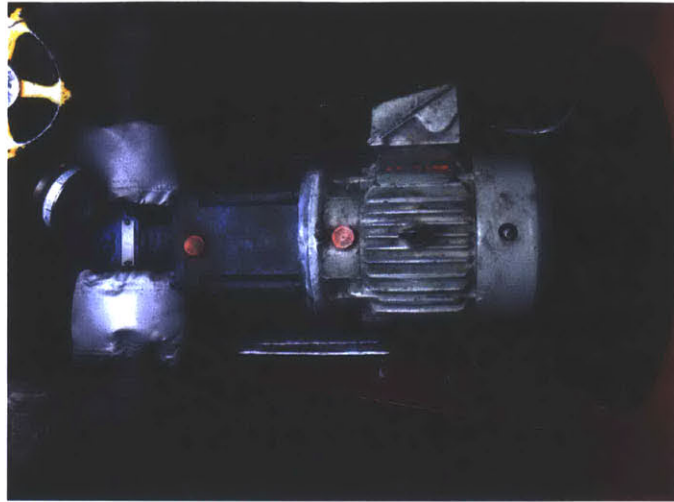


Figure 5-20: The 3 hp motor for the MPDE prelube pump. It is continuously energized when the MPDE is off and automatically stops when the engine is started. By monitoring its status (on/off) the engine's status can be inferred.

are adequate, maintaining oil temperature between 90 and 120 degrees Fahrenheit). The jacket water (JW) heater works in the same fashion, automatically energizing when required. When the engine shuts down, the prelube pumps turn on, the JW heater turns on, and the exhaust motor should be turned off by the crew. When the engine starts, the prelube pumps turn off, the JW heater turns off and the exhaust motor should be turned on.

The controllable pitch propeller is powered by a hydraulic loop. Hydraulic pressure is sent through a hollow shaft to the blades of the propeller, altering blade angle to quickly alter speed. Hydraulic pumps control the amount of hydraulic oil sent to the propellers to change pitch (by changing pitch the speed/direction of ship movement is controlled). The "C" Controllable Pitch Propeller (CPP) pump is unique in that it is energized for "Special Evolutions" that require that RMD be set. By energizing the "C" pumps, the operators are given greater handling performance and faster response.

During special evolutions the restricted maneuvering doctrine (RMD) is set and a prescribed machinery status is initiated. The nonintrusive SAW can detect machinery status by identifying the transients and recognizing equipment used uniquely during RMD. High level mission commanders could receive automatically logged summaries of times and durations when operational tempo increased. This could move the

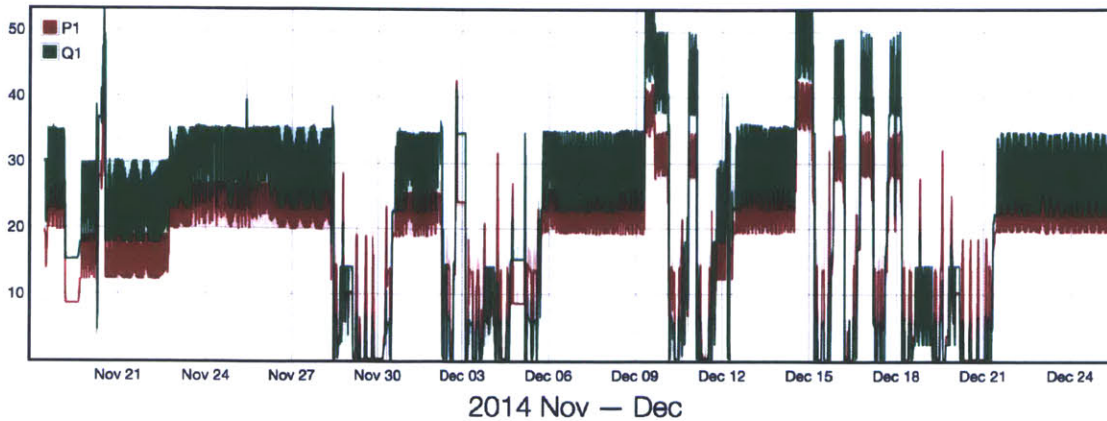


Figure 5-21: One month of electrical power data collected by a NILM installed on the USCGC Spencer.

surface fleet towards the aviation model where given a certain number of hours would equate to a certain level of crew fatigue. It is not uncommon for watchstanders to go without rest due to special evolutions combined with normal routine.

The next section illustrates the findings of a prototype SAW examining underway data obtained from the USCGC SPENCER.

5.3.4 Signal Processing and Transient Identification

The SAW’s functionality is critically dependent on its ability to disaggregate and identify transients of interest. This signal processing problem of extracting individual transients from aggregate data is an intellectually exciting and mathematically tractable problem for power systems of the size found on SPENCER, for example. Aggregate power data for a window of time aboard the SPENCER is shown in Fig. 5-21. The data clearly shows a complex mix of events occurring on the ship power system, and hints at the challenges in disaggregating transients.

Careful examination of the ship’s data reveals that start/stop events can be identified given exemplars representing individual load transients. During the cruise summarized by Figure 5-21, the cutter participated in an extended training period during which it made frequent port calls. This made monitoring interesting, with many transient load activations. The data from 10 December 2014 is one example, shown in

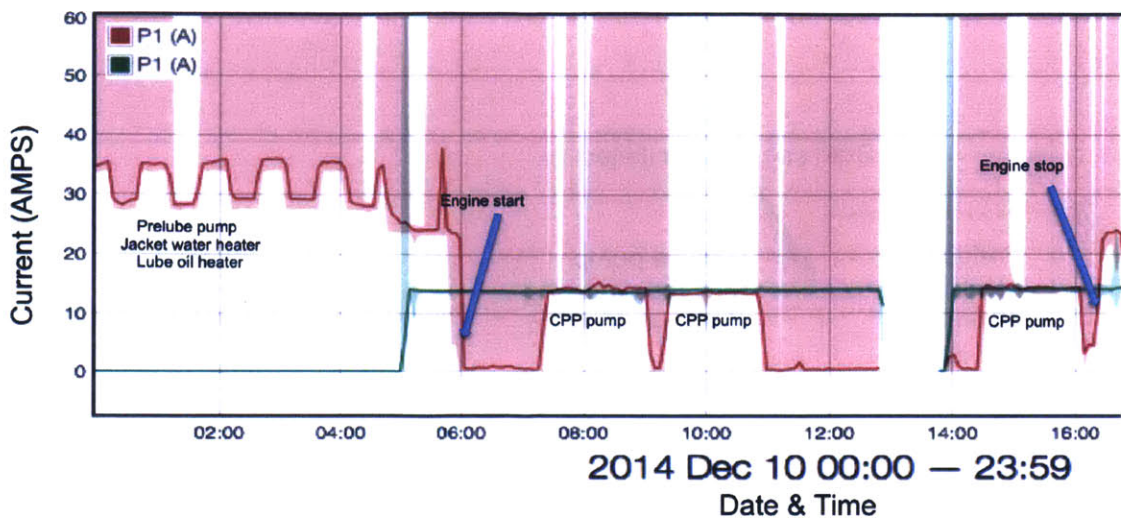


Figure 5-22: Data retrieved from monitoring on 10 December, 2014.

Figure 5-22 below.

For example, on this day from 0000 local time until 0600, the ship is in-port on shore power, with a collection of loads operating that would be typical for “bravo status,” i.e., the ship is active but not underway. The step function in power is the jacket water (JW) heater turning on and off. The prelube pump is also running. Figure 5-23 shows the first crew action for getting underway, the starting of the engine shown in Figure 5-23 at 0601. At this time, the prelube pump is deactivated in preparation for engine start.

At 0601 on SPENCER, nothing on the observed panel is operating. The MPDE is on at this time, providing its own lubrication with an attached shaft-driven pump. The electrical support loads are off, therefore, at this time. As described earlier, also on this panel is the CPP “C” pump which is energized when the cutter enters a special operational status of perceived higher risk, or RMD. The first “on” event for this can be seen at 0720 where the pump energizes as shown in Figure 5-24. This corresponds to the cutter leaving port, and RMD condition.

The data from the CPP can be highly telling as it relates directly to a condition of steaming of the vessel, the setting and securing of RMD. From the data, one can extrapolate that the cutter was at RMD from 0720 to 0903, 0919 to 1055, and 1426

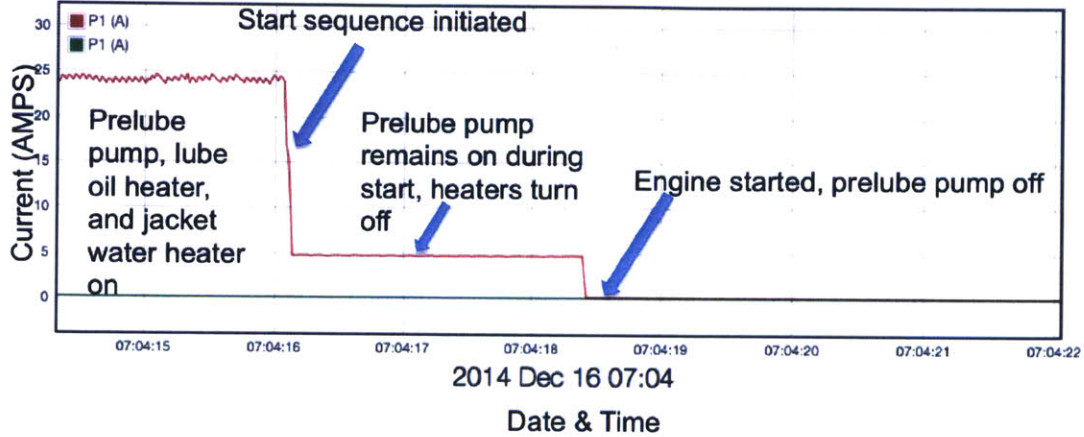


Figure 5-23: The MPDE starting as observed by the nonintrusive SAW. Note that there is a progression as the heating elements turn off once the start sequence initiates. The prelude pump remains energized until after the start sequence is completed to ensure lubrication during start (approximately 2-3 additional seconds).

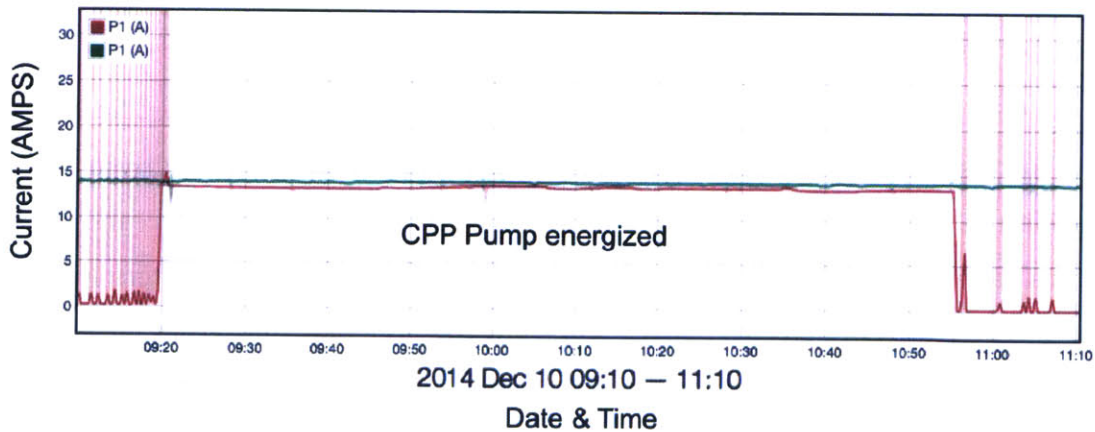


Figure 5-24: Characteristic CPP on and off event. The CPP motor is rated for 13.6A.

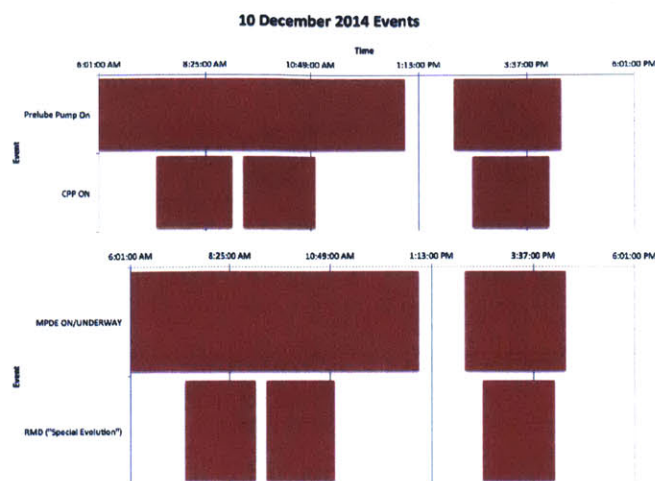


Figure 5-25: This figure shows the data taken from 10 December 2014 from the SPENCER SAW.

to 1607. For that day alone, the cutter was at a heightened state of readiness for 5 hours. This is a telling metric for mission commanders and planners and shows that there is a high probability of fatigue for this crew on this day. In evaluating the risk for an additional potential evolution, this metric could be crucial to weighing the risk and gain of a proposed mission. These start and stop times match manually logged start and stop times. Figure 5-25 shows the data on MPDE start and stop times as well as CPP times taken from the NILM box sensing of the prelube pump status and the CPP pump status.

Nonintrusively acquired and interpreted power data can be more reliable than human entry. Human entry relies upon an already overtaxed watchstander to log items. This leaves a high probability that during high stress or high tempo times, the log becomes an estimate at best, at worst a distraction to a watchstander trying to maintain equipment. An example of this is missed entries in the log. Taking the log from 10 December for an example, the SAW was able to identify two key events (the setting of RMD denoted by starting the CPP “C” pumps) that were not logged in the watchstander’s log. Thus the NILM offers a way to deconflict the work of the watchstander while still providing an accurate log, in this case more reliable than one created by human entry. It is important to note that the NILM also provides exact

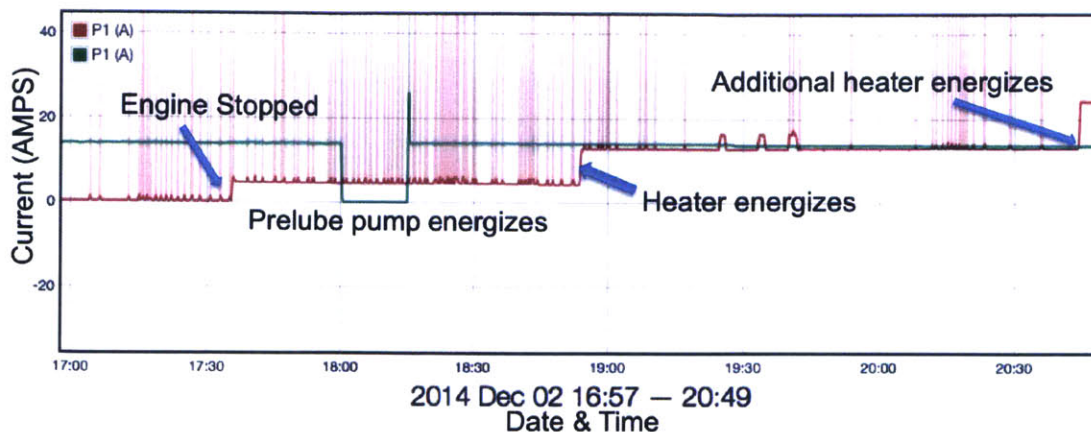


Figure 5-26: MPDE Stop Sequence

time data whereas the logs were off from the NILM data by up to 15 minutes.

When the MPDE is stopped, there is a similar sequence of relevant transients and load activations. The prelube pump should start immediately upon engine stop. Other loads associated with the MPDE (JW heater and lube oil heater) will not start immediately as these are controlled by temperature thermostats. The engine stopping can be seen in Figure 5-26. Then, approximately 15 minutes later, the heater loads activate as shown.

The SAW is able to identify sequences and ensure prerequisites for proper load activation and operation are met. On Famous Class cutters the prelube pumps are continually running. However on many ships, such as the USCG 210ft Reliance Class cutters, prelubing must be done manually before starting the MPDE's or increased wear will develop on the MPDE. This requires a human input that may be skipped if the operator is not following the SOP correctly. An installed NILM would be able to sense the prelube pump and ensure it is started before starting the MPDE. This has many applications as many systems work in this manner where prerequisites should be met before other steps are taken or damage can occur to the system. On newer ships, many of these fail safes are built into the systems. For example, on the newest USCG cutter, the WMSL, the controls software will not allow start of the MPDE without prelube to a specified pressure; however on legacy class cutters this function does not exist and they rely on human input, which, especially during high stress

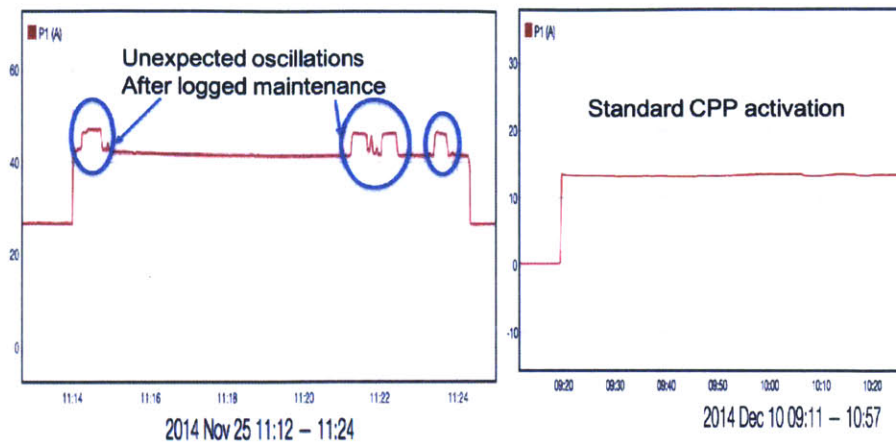


Figure 5-27: Image of unusual readings from CPP pump after maintenance.

situations, can result in errors. The SAW can be used to ensure proper steps are followed, producing a warning signal if proper sequencing is not observed.

The nonintrusive SAW also offers opportunities for tracking machinery health. There are characteristic start up and steady state signatures associated with each piece of machinery. If these signatures change, then there may be an identifiable maintenance issue. For example, pump damage can occur when pumps are run without a pumping medium (run “dry”). This situation occurred with a CPP pump onboard SPENCER during the cruise under observation here. After performing maintenance, the CPP pump was started and run without a medium for several seconds (system was purged of fluid during maintenance). Figure 5-27 shows power fluctuations possibly caused by loss of suction in the pump. By identifying these events, the nonintrusive monitor can be used for health monitoring.

Another example of this is when the cutter moors (enters port) or leaves port. During these evolutions, there are rapid calls for rudder and propeller commands. These rapid movements can be seen in unusual patterns in the CPP’s power draw as the motor is worked aggressively. This can cause damage to the pump or motor beyond normal expectations and, depending on conditions, could indicate that operators should be encouraged to decrease command frequency. At a minimum, these records can be used as a kind of “odometer” to indicated the potential need for maintenance. Figure 5-28 shows these heavy use patterns.

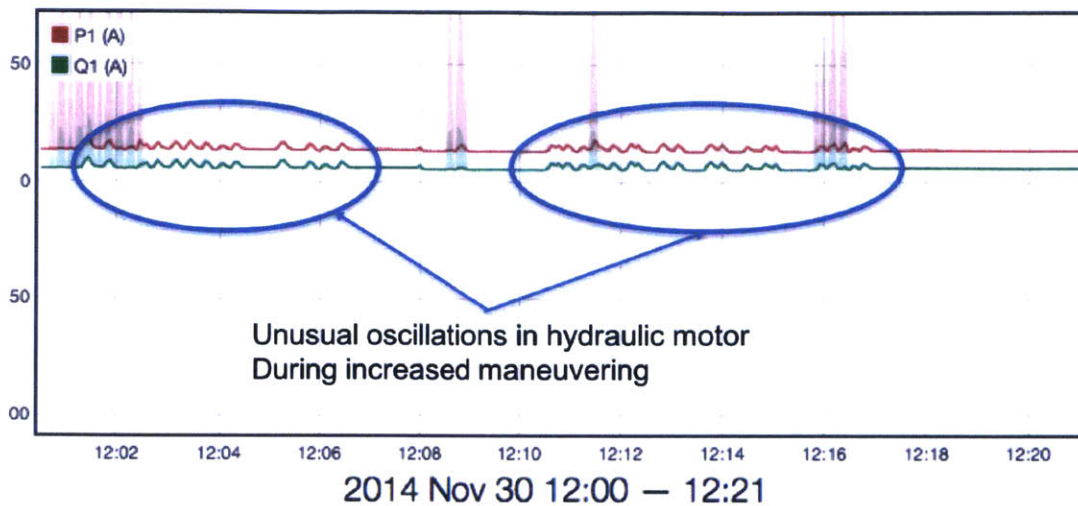


Figure 5-28: Image of unusual readings from CPP pump during mooring; oscillations indicate aggressive operation of the pump motor.

5.3.5 Case Study Results

Each piece of machinery on the ship can be characterized on initial installation of the SAW during a training phase. There are a variety of methods for acquiring training data. During the training phase, for example, readings are taken on three phases A-B-C while loads on-board the ship are activated or observed during dock-side operation. Power signatures are recorded in real and reactive power and higher current harmonics, associated as fingerprints for each load of interest. Different machinery draws different amounts of real and reactive power and harmonics consistent with the different physical tasks performed by different machines. Each piece of equipment may also exhibit unique turn-on transients. For example, Figure 5-29a shows the real and reactive power demanded during the startup transient of a CPP pump. For comparison, Figure 5-29b shows the turn-on transient of a lube-oil heater. These distinctive, predictable, and reproducible waveforms can serve as fingerprints for recognizing and disaggregating load operating schedule by examining the aggregate power feed to the loads, even when several loads are operating at the same time.

Some machines operate with strong periodicities, such as heaters that have a period of operation where they are on for a relatively constant amount of time and then

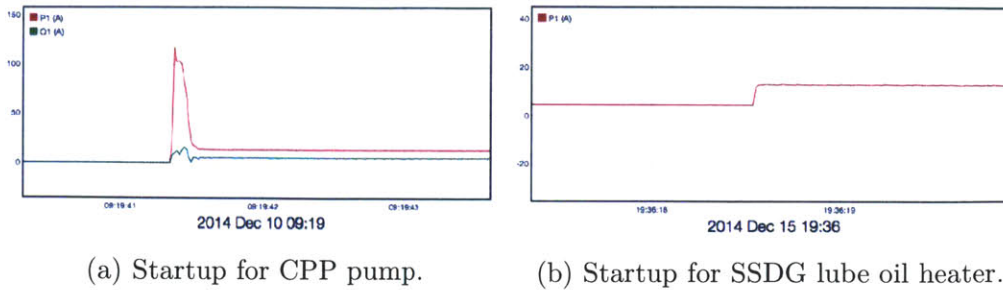


Figure 5-29: Startup transients for USCGC SPENCER equipment

off for a constant and relatively predictable amount of time for certain underway conditions. Other machines follow a predictable sequence of events, e.g., if one piece of machinery starts, another should start at a given interval later, e.g. the pumps in a reverse-osmosis water system. Through knowledge of proper system operation and prediction of events, an expert system can be developed that uses non intrusively observed power transients to identify not only particular loads but also particular cycles of operation or ship state, such as an engine start. The Wattsworth programming environment can be used to flexibly implement an expert system that analyzes and summarizes ship state and condition based on observed load transients and operation. First, by running a load identification filter over an incoming set of data, load events can be identified and tagged. Streams of tagged events can be further analyzed to identify sequences of load operation that correspond to correct or improper use of a multi-load system. These analyses can be summarized automatically as log reports, providing high accuracy automated replacements for human generated logs, unburdening the crew of this labor. For example, Figure 5-30 shows an actual log report from a SPENCER watchstander during a recent underway cruise. The SAW onboard SPENCER automatically generated the log shown in Figure 5-31. Note the similarities and the more exact times found by the NILM system.

Additional metrics can be parsed from this data. For example, fuel oil transfer pumps periodically transfer diesel fuel to a ready service tank feeding propulsion and power generating prime movers. These fuel pumps deliver a relatively consistent flow rate. The SAW is capable of tracking the operating time of the fuel pump electrically, and then deriving the implied fuel transferred to the service tank, essentially

MACHINERY LOG

U.S. Coast Guard Cutter SPENCER (WMEC 905)

Date: 10 DEC 14

WATCH OFFICER'S REMARKS:

0000-0400

Vessel is moored at the above location. Ship's status is B-12. Electrical power, potable water, and sewage discharge are being supplied via shore-tie. Reviewed Tag-Out Log and the following machinery is listed as OOC: Oily water separator system, Bridge window heater STBD center, #2 MG set, VCT suction valve, Clutch control PLTHS pushbutton and STBD CPP L/O Heater. All other machinery is IAW the Midnight Machinery Status sheet. 0345 Carried out the watch routine.

0400-0800

Vessel is moored as before. 0425 Energized the pre-lube pumps for both SSDG's. 0430 Started both SSDG's and secured the pre-lube pumps. 0500 Shifted from shore power to ship's power. 0501 Paralleled the #1 SSDG to the main bus. 0535 Conducted round of the ship and the pier, all secure. 0545 Conducted round of the ship and the pier, all secure. 0550 Conducted blow downs of BMDE's. 0600 Started BMDE's. 0630 Shifted BMDE's from E-MAN to ECC Auto. 0700 Completed Light-Off Schedule. 0720 Set the Special Sea Detail. Set RMD in the Engine Room. 0735 Clutched in BMDE's, placed in PHC Auto. 0745 Carried out the watch routine.

0800-1200

Vessel is moored as before. 0814 Vessel U/W. 0825 Conducted round of the E/R, all secure. 0855 Secured from Special Sea Detail, secured from RMD in the E/R. 0905 Secured BCPP C Pumps. 0906 Energized BMDE and BSSDG L/O spinners. 0921 Set Restricted Maneuvering Doctrine and Set GE. 0922 Energized BCPP C Pumps. 0924 Energized #1 and #2 Fire Pumps. 1055 Secured from GE and Secured from Restricted Maneuvering Doctrine. 1056 Secured #1 and #2 Fire Pumps. 1057 Secured BCPP C Pumps. Carried out the watch routine.

Figure 5-30: The machinery log generated manually by the ship's crew by logging events and times.

automating the tracking of fuel consumption. Currently, fuel transfer amounts are discovered through manual soundings or an Automatic Tank Level Indicator (TLI) on newer vessels. This approximation from the SAW can be a check of these readings. In heavy weather, manual soundings and TLI readings are extremely unreliable and difficult to achieve due to the motion of the vessel. In rough seas the NILM is capable of providing the most accurate fuel consumption estimate. Several other metrics like this can be developed as well, including machinery hours (an important maintenance factor), crew fatigue from hours at heightened alert, proper sequencing and alignment of multi-pump systems like RO, and metrics on days underway.

The robust nature and simplicity of the NILM system offers several advantages to the distributed sensor network currently used by USCG, USN and commercial fleets. Communications losses are common place in distributed systems with long cable runs, large numbers of complex sensors, and complex communications systems

MACHINERY LOG

U.S. Coast Guard Cutter SPENCER (WMEC 905)

0000-0400

Vessel is moored as before. Watch properly relieved, carried out watch routine.

0400-0800

Vessel is moored as before. 05:01: Started #2 SSDG. 07:20 :Set RMD. Watch properly relieved, carried out watch routine.

0800-1200

Vessel is moored as before. 07:37. Secured Inport ASW pump. Vessel Underway. 09:03: Secured RMD 09:07: Secured #2 SSDG. 09:21: Set RMD. 10:55: Secured RMD 11:29: Started #2 SSDG. Watch properly relieved, carried out watch routine.

1200-1600

Vessel is underway as before. 13:56:Started #2 Main Propulsion Diesel Engine Prelube Pump 14:01:Stopped #2 Main Propulsion Diesel Engine Prelube Pump 14:04: Shifted to Underway ASW pump. Underway. 14:04: Started #2 SSDG. 14:26: Set RMD Watch properly relieved, carried out watch routine.

Figure 5-31: Automatic machinery log generated by a SAW onboard SPENCER.

involving repeaters. Compared to these systems, the nonintrusive approach requires much less maintenance, requires less equipment, and is a fraction of the price to purchase and to install.

5.3.6 Implications

Observation from the USCGC SPENCER field data indicates that nonintrusive power monitoring can have an important role in ship operation and maintenance. Here, we observed a single important breaker panel with eight systems during a month of underway operation. The nonintrusive SAW accurately tracked all systems. The SAW also developed derived metrics using “expert” knowledge of the combined operation of ship systems. For example, the SAW tracked MPDE auxiliaries to determine when the MPDE was online and offline as well as track the usage of the CPP “C” pumps. By tracking the auxiliaries systems of the MPDE, order of operations can be verified to ensure health of the machinery as well as proper start up and securing procedures.

Using this information, top level commanders can easily see the exact state of the vessel, underway, at RMD, or other modes of operation. Work is currently underway to install a NILM system on the USCGC SPENCER to monitor the entire, ship-wide machinery status by placing the NILM at the ship's central power distribution panel. In preliminary observation, loads were observed turning on and off, similar to the observations from the single panel experiment described here. Future work is aimed at expanding the ability of the NILM to detect events in more complex aggregate data streams, and produce comprehensive logs and reports for the entire ship.

Appendix A

Documentation

This appendix covers the installation and operation of non-intrusive power monitors and the usage of the NILM Manager web platform. This appendix is available in its entirety in the NILM Manager Help section. From a standalone system visit <http://nilm.standalone/help> or from any Internet connected device visit <http://www.wattsworth.net/help>. Appendix A.1 provides quick start guides to common operations. This covers the installation and configuration of non-contact and contact power meters as well as smart plugs. Appendix A.2 is a comprehensive guide to all of the Wattsworth hardware and software arranged as follows:

Hardware Configuration Setup and install contact or non-contact power meters

Software Configuration Configure data capture with one or more power meters

Smart Plugs Setup and use smart plugs with a NILM system

Administration Manage databases, processes, and global settings

NILM Explorer View data streams in the web browser plotting interface

NILM Filter Run custom python scripts that create new data streams

NILM Analyzer Run custom python scripts that generate HTML reports

NILM Finder Find and identify loads using exemplars

Processes Automate NILM filters and analyzers

Command Line Interface Low level command line tool to interact with the NILM

Quickstart Guide To Setting up a Contact Meter

1. Install current and voltage sensors

Configure the resistor settings for the current sensors

The LEM current sensors are converted to voltages by a resistive load on the meter circuit board. The value of this load is set by a series of DIP switches for each current sensor. See [Setting up a Contact Meter](#) for details on setting the DIP switches. The load resistance should be set to spread the expected signal as widely as possible across the +/- 5V input range. The upper limit on the load resistance is specified in the LEM datasheet. See [LA 55-P datasheet](#) for an example.

Connect the LEM current sensors and mount the meter

The voltage sensor wires and current sensors must be installed by a trained electrician. Make sure all equipment is fully de-energized and tagged out before installation. It is best practice to install the voltage sensors on a separate breaker so the meter can be de-energized without shutting off other appliances. Use the illustration in [Setting up a Contact Meter](#) to determine the mapping of voltage and current sensor phases to the LabJack indices. This information will be required **Step 3**.

2. Setup the ethernet connection

Pick a network topology

Single Meter: If there is only one contact meter in the installation, it should be directly connected to the host computer's built in ethernet port or to a USB to ethernet converter. No special configuration of the LabJack IP address is required. The host computer interface must be a unique address on the same subnet. With a LabJack set to factory defaults the host address should be 192.168.1.200. [see an example](#)

Multiple Meters: If multiple contact meter connect to the same host there are two possible network configurations. Both require reconfiguration of the LabJack IP address.

Switched Subnet: In this topology the meter's and the host computer connect to a common network switch. The meters and the host computer must be on the same subnet and each LabJack must have a unique address. [see an example](#)

Multiple Interfaces: In this topology each meter connects directly to the host computer either to a built-in ethernet port or through a USB to ethernet converter. Each meter and its host interface share a subnet and must have unique addresses. [see an example](#)

Configure the LabJack IP address

The LabJack settings are configured with LJStream. This is a *Windows only* application. Check the company website for the latest release or download the local copy of [LabJack-2015-11-19.exe](#). The Windows machine can be configured to talk to the LabJack over ethernet or the LabJack can be physically removed from the meter and connected to the Windows machine by USB.

Configure the Host IP address

The host computer IP address is configured with Network Connections. One way to access Network Connections is through the Dash ([see example](#)). This will bring up a dialog with all of the current network connections. Select the interface used by the LabJack and follow the steps in [Network Configuration](#) to change the IP address settings.

3. Configure the host computer

Set up a secondary hard drive

The NILM database (NilmDB) should be stored on a separate hard drive if possible. This protects the main system from being completely filled by NilmDB and becoming unstable (although a properly configured `meters.yml` should prevent the database from growing too large). Placing NilmDB on a separate drive also makes it possible to get data off a meter by simply removing the drive and replacing it with a new one. See [Setting Up Storage](#) for more details.

Add configuration to `meters.yml`

Open `meters.yml` (on the Desktop) in a text editor or simply double click the file icon to use the default editor. Copy the template and follow the instructions in [Setting up a Contact Meter](#). *Note:* If you are configuring multiple meters, each meter needs a unique name (`meter1`, `meter2`, ... `meterN`). Adjust the template configuration accordingly.

Configure BIOS for Auto-Boot

NILM's installed at remote sites should be configured to restart if power is lost and later restored (auto-boot). By default the BIOS will leave a computer off if power is lost, requiring a manual button press to turn the system back on. Each BIOS is different but most can be configured by holding F12 or DEL during system boot. It is best practice to install an uninterruptible power supply (UPS) with the meter both to reduce the chance of power failure and to record current and voltage signals during the power glitch for later diagnostics. Make sure the UPS is configured to restart automatically if it is discharged entirely before power is restored.

4. Verify the configuration

Check `meters.yml`

Use `nilm-check-config` to check `meters.yml` for syntax errors and configuration warnings. Run this command from a terminal window ([how to access the terminal](#)). Make sure there are no errors and that you fix or fully understand any warnings. The configuration check also displays the amount of storage required. It is best practice to use only 80% or less of the available space even if NilmDB is on a separate drive. This keeps space available for data generated by filters or snapshots of interesting data that would otherwise be erased by the `keep` settings in `meters.yml`.

Check the sensors

Use `nilm-scope` to check the sensors. Run this command from a terminal window ([how to access the terminal](#)). Check the relative phases of the current and voltage to make sure the sensor indices in `meters.yml` are correct. Note that the currents will not line up exactly with the voltages if the power factor is less than unity. This is common when there are large motors or other inductive loads on a phase.

5. Start capturing data from the meter

Start `nilm-capture` service


Start (or re-start) the [nilm-capture service](#) using the terminal ([how to access the terminal](#)).

Check the system logs


Check the files in `/var/log/nilm` periodically for error messages. See [NILM System Logs](#) for details.

6. View meter data from the website

Refresh the web manager

Open the [Installation Administration](#) page, select the "Database" tab, and click  twice. This will update the web interface with the new meter. You should the new meter and its data streams appear in the navigation tree. See [Managing the NILM Database](#) for more details.

Plot meter data

Open the [Data Explorer](#) and select one or more streams from the new meter. Click  to view the latest data and verify that the data is being captured and processed correctly. Edit `meters.yml` to fix any of the following errors:

- If the ratio of P and Q are wrong adjust the phase's `sinefit-rotation` angle.
- If current or voltage scales are incorrect adjust the respective `sensor_scales`. Each phase can be fine tuned by using an array of scale factors instead of a single value.
- If the amplitude of prep is wrong but the current and voltage are correct adjust the `nominal-rms-voltage` setting.

See [Configuring a Contact Meter](#) for more details. Note that you must restart the `nilm-capture` service for changes in `meters.yml` to take effect.

Quickstart Guide To Setting up a Non-Contact Meter

1. Install the non-contact sensors

Select a sensor configuration

Select the type of sensor platform you want to use. The [Flex Sensor](#) is an all-in-one platform that is ideal for multiple conductor powerlines where the location of each conductor inside the bundle is not known. The [D-A-Y](#) board is ideal for monitoring multiple physical conductors or where the location of each conductor in a wire bundle is known. In general the D-A-Y board package provides a more flexible solution and is the recommended platform. The non-contact sensor (D-Board) has four input ports. Sensor boards (A Boards) can connect directly to these input ports. Each A Board provides a current and a derivative of voltage signal. Connecting a sensor to each input port provides a total of four current measurements and four derivative of voltage measurements. This is the recommended configuration. If you want a different combination of sensor inputs or want true voltage rather than derivative of voltage you can use a [Y Board](#) to adjust the sensor configuration. The Y-Board connects to any input port on the D-Board and provides two connection ports for sensors (A-Boards). A set of jumpers selects which signals are passed back to the D-Board. See the a close up of the jumper configuration diagram [here](#).

Attach sensors

If using the D-A-Y board platform, attach the A-Boards to the powerline with zip ties. Multiple A-Boards can be connected with a single zip tie or each board can be connected individually as shown in the example [figures](#). Make sure each board is connected securely to ensure accurate measurements. Connect the ribbon cable plugs to the D-Board *before* putting the D-Board in its enclosure. Once all cables are seated, connect the USB cable and seal the D-Board in its enclosure. If using the Flex Board secure the sensor to the powerline two zip ties, one at the base and the other around the flexible arms.

2. Configure the host computer

Set up a secondary hard drive

The NILM database (NilmDB) should be stored on a separate hard drive if possible. This protects the main system from being completely filled by NilmDB and becoming unstable (although a properly configured `meters.yml` should prevent the database from growing too large). Placing NilmDB on a separate drive also makes it possible to get data off a meter by simply removing the drive and replacing it with a new one. See [Setting Up Storage](#) for more details.

Add configuration to `meters.yml`

Open `meters.yml` (on the Desktop) in a text editor or simply double click the file icon to use the default editor. Copy the template and follow the instructions in [Configuring a Non-Contact Meter](#). *Note:* If you are configuring multiple meters, each meter needs a unique name (`meter1`, `meter2`, ... `meterN`). Adjust the template configuration accordingly.

Configure BIOS for Auto-Boot

Meters deployed to remote sites should be configured to restart if power is lost and later restored (auto-boot). By default the BIOS will leave a computer off if power is lost, requiring a manual button press to turn the system back on. Each BIOS is different but most can be configured by holding F12 or DEL during system boot. It is best practice to install an uninterruptible power supply (UPS) with the meter both to reduce the chance of power failure and to record current and voltage signals during the power glitch for later diagnostics. Make sure the UPS is configured to restart automatically if it is discharged entirely before power is restored.

3. Verify the configuration

Check `meters.yml`

Use `nilm-check-config` to check `meters.yml` for syntax errors and configuration warnings. Run this command from a terminal window ([how to access the terminal](#)). Make sure there are no errors and that you fix or fully understand any warnings. The configuration check also displays the amount of storage required. It is best practice to use only 80% or less of the available space even if NilMDB is on on a separate drive. This keeps space available for data generated by filters or snapshots of interesting data that would otherwise be erased by the `keep` settings in `meters.yml`.

Check the sensors

Use `nilm-scope` to check the sensors. Run this command from a terminal window ([how to access the terminal](#)). Check that the magnetic sensors have strong signals and show a diverse response to currents on different phases. Adjust the sensor location if the signal is low or the sensor responses appear too similar. Check that the voltage sensor has a strong and stable signal. If you are using an A-Board connected directly to the meter the signal will look like a "messy" sine wave. If you are using a Y-Board with integration the signal should be a "clean" sine wave. The legend labels the sensors by type. Make sure the magnetic sensor channels are labeled "current" and the voltage sensor channel is labeled "voltage". Any channel not in the `meters.yml` configuration will be labeled "--unused--". Note that all channels are stored in the `sensor` data stream regardless of sensor type.

4. Calibrate the non-contact sensors

Run the `nilm-calibrate` program

Use `nilm-calibrate` to finish setting up the non-contact meter. Run this command from a terminal window ([how to access the terminal](#)). Calibration requires a NILM smart plug and a resistive load like an incandescent light bulb and a micro USB cable.

5. Start capturing data from the meter

Start `nilm-capture` service

Start (or re-start) the `nilm-capture service` using the terminal ([how to access the terminal](#)).

Check the system logs

Check the files in `/var/log/nilm` periodically for error messages. See [NILM System Logs](#) for details.

6. View meter data from the website

Refresh the web manager

Open the [Installation Administration](#) page and click the local repository **Manage** button. On the Repository Management page, select the "Database" tab, and click **Refresh** twice. This will update the web interface with the new meter. You should see the new meter and its data streams appear in the navigation tree. See [Managing the NILM Database](#) for more details.

Plot meter data

Open the [Data Explorer](#) and select one or more streams from the new meter. Click **Start Live Update** to view the latest data and verify that the data is being captured and processed correctly. If the data is correct but noisy or shows significant "cross talk" between phases try adjusting the sensor position and recalibrating. If the data looks incorrectly calibrated try increasing the duration of the calibration and/or the power of the calibration load. You can re-calibrate the system at any time by running `nilm-calibrate`.

Quickstart Guide To Smart Plugs

1. Hardware Setup

Configure Wireless Router

The plugs will work with most commercial routers. The plug [documentation](#) provides detailed instructions on configuring a TP-Link modem.

Connect Plug

Before using a plug it must be configured over USB. Power up the plug by placing it in an outlet and connect it to the NILM using a micro USB cable. The status LED will alternate between green and blue. Wait until the LED is solid green before continuing to **step 2**. See a closeup view of the USB connection [here](#)

2. Configure the Plug

Plug Command Line Interface (CLI)

From a terminal window, open up the plug command line interface by running the command below ([how to access the terminal](#)). *NOTE:* If multiple plugs are connected to the same NILM over USB then you must specify the `/dev/NODE` name of the plug. See [USB Plugs](#) for details.

```
1 $ nilm-plug --cli
2 /dev/smart_plug, 115200 baud
3 ^C to exit
4 -----
5 Wattsworth WEMO(R) Plug v1.0
6 [help] for more information
7
8 >
```

Disabling Calibration Mode *optional*

If the plug is configured as a calibration load it will not operate as a smart plug until you explicitly stop the calibration. Run the commands shown below. (documentation for [calibrate](#), [config](#), [restart](#))

```
1 > calibrate stop
2 > config set standalone false
3 > restart
```

This will restart the plug and you will have to enter the command line interface by running the `nilm-plug --cli` command again

Configuring Wifi Parameters

The plugs will work on most standard WiFi networks. They will operate on 802.11a/b/g with WPA2, WEP or no encryption. WPA2 is recommended for the best security. Run the following commands replacing `network_name` and `network_password` with the appropriate values. Ignore the password setting if your network is not protected. (documentation for [config](#))

```
1 > config set wifi_ssid network_name
2 > config set wifi_pwd network_password
```

This is all that's needed to get the plug on your network. To check the plug connectivity run the following commands to view debug output from the plug as it connects to the network: (documentation for [debug](#), [wifi](#))

```
1 > debug 5
2 > wifi on
```

You should see several AT commands and ultimately a success message with an IP address. This should match the address reservation on the wireless modem as configured in **Step 1**. Alternatively you can look at the plug log which should end with an entry similar to the following: (documentation for [log](#))

```
1 > log read
2   ... other entries ...
3 [2016-03-04 20:26:55]: Joined [MIT] with IP [18.111.127.86]
```

3. Using the Plug

Control from a terminal window with `nilm-plug`

Once the plug is configured for the wireless network it can be disconnected from the NILM and placed in any outlet within range of the WiFi access point. After the plug has connected to the network you can interact with it from a terminal window using `nilm-plug`. This command lets you read the power meter data, and set the relay.

Control with Python scripts

An application programming interface (API) is available in Python. See [Plug API](#) for full details. Python scripts using the API can be executed directly from the command line or built into NILM Filters and Reports.

Quickstart Guide To Adding Custom Data

1. Format Data and Add Timestamps

NILM data is organized as rows of timestamped measurements. A measurement may be multiple columns. All values must be numeric. No string sequences are allowed. For example a three axis accelerometer could generate three columns of data corresponding to X, Y, and Z measurements, or a weather station might record barometric pressure, windspeed, wind angle, and temperature. The first column of each row must be a a microsecond Unix microsecond timestamp. Each timestamp must be unique and the set must be monotonic and ascending. All values must be space separated. The example below shows sample timestamped data with five floating point values:

```
1360262088608397 0.0543 0.3084 0.5181 0.0917 0.1640
1360262088625069 0.0864 0.5944 0.6732 0.2295 0.7166
1360262088641741 0.2242 0.8805 0.6136 0.3385 0.9002
1360262088658413 0.1411 0.6335 0.2040 0.2504 0.0136
1360262088675085 0.7833 0.1320 0.3238 0.1469 0.1157
1360262088691756 0.2791 0.7644 0.2437 0.2097 0.1057
1360262088708428 0.8472 0.3904 0.1049 0.0328 0.4186
```

2. Create a New Database Stream

Stream Naming Convention

Data is stored in uniquely named streams. The name has two components, `group_name` and `file_name`. These are separated by a forward slash similar to a Unix path name:

```
/group_name/file_name
```

Streams are indexed according to these names. Streams store a fixed number of columns of a particular datatype. See the [NilMDB reference](#) for a full list of valid data types. The naming convention is `type_columns` where `columns` is the number of values in a sample not including the timestamp. The data in Step 1 above could be stored in a stream with the following data format:

```
float32_5
```

Create Group Stream *optional*

If you want to have your dataset appear in the same group as other streams already on the system, use the same group name as these other streams. If you want your dataset in its own group you will need to create an additional group stream. Use the `nilmtool create` command replacing `group_name` with a unique lowercase alphabetic string as shown below:

```
1 $ nilmtool create /group_name/info uint8_1
```

Create Data Stream

Determine the datatype and the number of columns (not including the timestamp) for the stream. If you are unsure of the datatype use `float32`. Create the stream replacing `group_name`, `file_name`, and `columns` with the appropriate values.

```
1 $ nilmtool create /group_name/file_name float32_columns
```

3. Write Data to the Stream

Determine Start and End Timestamps

Data is stored as bounded intervals. In order to add raw data to a stream you must specify the interval of time that it covers.

The minimum valid interval is first timestamp in the data to one microsecond past the last timestamp in the data. For example the data in Step 1 could be inserted with the following interval:

```
start: 1360262088608397 end: 1360262088708429
```

Write the Data

Use the `nilmtool insert` command to write the data into the new stream as shown below replacing `data.txt` with the timestamped data created in Step 1 and the appropriate timestamps for `start` and `end`:

```
1 $ nilm-insert /group_name/file_name -s start -e end < data.txt
```

4. Decimate Stream (optional)

In order to plot streams using [NILM Explorer](#) they must be decimated. This roughly doubles the disk space required by the stream so if you do not need to plot it, skipping this step will save space. Use the `nilm-decimate-auto` command as shown below:

```
1 $ nilm-decimate-auto /group_name/file_name
```

5. Configure Stream Attributes

Open the [Administration](#) window and refresh the database twice. If you do not see the new stream make sure that the `group_name` matches an existing group or if you wanted to create a new group make sure you added the group stream in Step 2. Select the new file and update the stream attributes, in particular select the plottable checkbox if you want to use it in [NILM Explorer](#). See [Administration](#) for more details.

Hardware Setup

Introduction

Non-Intrusive Load Monitors (NILM) collect data using one or more hardware meters. There are two different kinds of meters: contact and non-contact. The contact system uses commercial current and voltage sensors and should be installed by an electrician. The non-contact system uses electromagnetic field sensors to measure current and voltage without touching the powerline. The contact system is more accurate but expensive and difficult to install. The non-contact system can be installed quickly and easily but requires an additional calibration step. A single NILM can manage one or more of either type of meter.

Contact Meter

The contact meter uses LEM voltage and current sensors. The sensors are digitized by a UE9 LabJack. The LabJack has multiple input channels. The sensor to LabJack channel mapping (`index`) is show in the figure below. The current sensors connect to the PCB with Molex plugs. Trace the twisted cable from the plug to the panel mounts and label the outside of the box before installing the system. This will make it easier to assign the correct channel mappings.

Connections



Index	Legend	Type
0		Current
1		Current
2		Current
3		Voltage
4		Voltage
5		Voltage

The sensors must be scaled correctly to convert the measurements to volts and amps. The voltage scale factor is set in hardware to 0.0919 and the current scale factor can be calculated using the formula in the figure below. α_{LEM} is the *Conversion ratio* found on the LEM sensor datasheet. This is usually on the order of 1000. See [LA 55-P datasheet](#) for an example. R is the load resistance set by the channel DIP switches (see the chart below).

Conversion Factors



Switch	OFF Value
1	80 Ω
2	40 Ω
3	20 Ω
4	10 Ω
5	5 Ω

ON switches add 0 Ω

Current Conversion Factor

$$\frac{5.07 - (-5.18)}{2^{16}} \times \frac{\alpha_{LEM}}{R}$$

α_{LEM} : from LEM datasheet

Voltage Conversion Factor

0.0919 (fixed)

Write down the LabJack index to sensor mapping and the scale factor required for each sensor. Once the meter is installed and connected to the host machine, enter these values into `meters.yml`. See [Setting up a Contact Meter](#) for details.

Non-Contact Meter

The non-contact meter uses magnetic and electric field sensors to indirectly measure the current and voltage in a powerline. This system is easier to install than a contact meter but requires an additional calibration step. There are two different non-contact sensor hardware platforms. The Flex Board and the D-A-Y Boards.

Flex Board

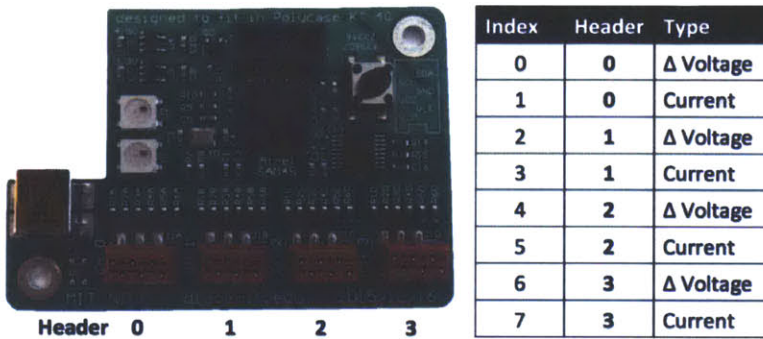
The Flex Board is an all-in-one platform with both sensors and a microcontroller for data acquisition. The main PCB has an electric field sensor, and a magnetic field sensor. There is also compensation circuitry to provide an integrated electric field output which is proportional to the line voltage. The magnetic sensor also contains a PTAT (proportional to absolute temperature) sensor. The flexible arms hold four more magnetic sensors. The microcontroller samples all eight sensor outputs and provides the values over USB. The sensor indices are provided in the figure below. Unless there is a need for a hardware integrated signal, the raw electric field output (index 0) with digital integration is the recommended configuration.

Index	Type
0	Δ Voltage
1	Current
2	Current
3	Current
4	Current
5	Current
6	Voltage
7	PTAT

Flex board index to sensor mapping. See the board silk screen header pinout.

D-A-Y Boards

The other non-contact platform is a suite of three separate boards. The standard configuration is a data acquisition (D) Board and at least as many analog sensor (A) boards as phases (eg 3 phase system = 3 or more A Boards). Each A-Board has a current sensor and a "derivative of voltage" sensor. The figure below shows the channel to sensor mapping when A-Boards are connected directly to the D-Board. Only one of the four A-Boards is used by the NILM to determine the line voltage. Once you have configured the meter in `meters.yml`, use the `nilm-scope` utility to determine which A-Board has the strongest reading and use its index as the `voltage:sensor_index` in `meters.yml`.

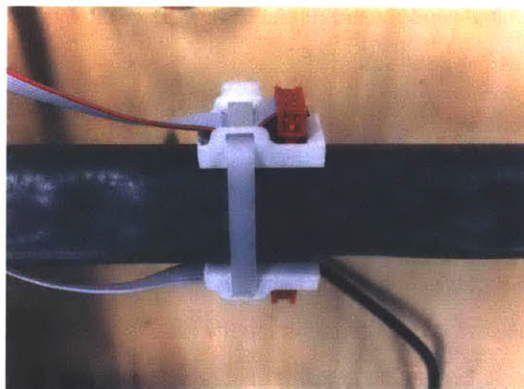


D-Board index to sensor mapping. Each A-Board measures current and the derivative of voltage

Securely connect each A-Board to the power cable using a zip tie as shown. The accuracy of the meter depends on the stability of this connection. If the A-Boards rotate after installation the meter will need to be recalibrated. The quality of the output depends on how well the sensors are positioned. If the location of the conductors is visible try to align each A-Boards to a separate conductor. If the conductor geometry is not known use the `nilm-scope` utility once you have set up the meter in `meters.yml` to determine optimal sensor placement. Loads on each phase should exicte a different "mix" of the sensors.



Capture the ribbon cable and securely attach to



Multiple sensors can be connected with a

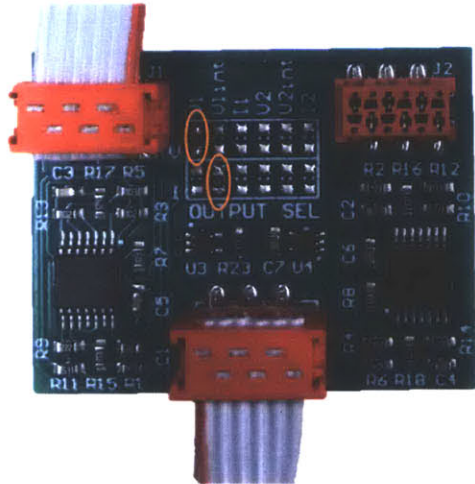
the powerline

single zip tie

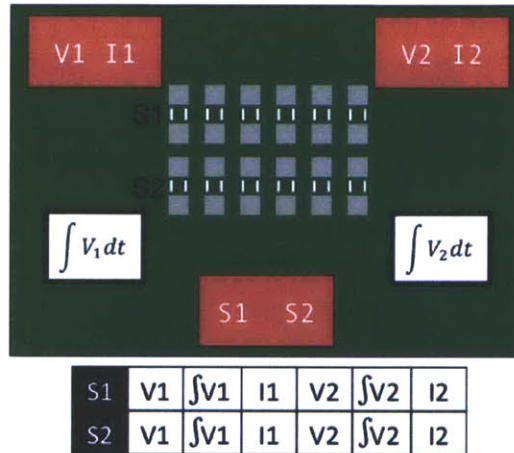
Using the Y Board

The non-contact meter can adapt to a wide variety of installation types through the use of the Y Board adapter. The Y Board multiplexes two A Boards into a single pair of channels and provides optional hardware integration. The lefthand figure below shows a cartoon of the Y Board schematic. Zero ohm 0603 jumpers are used to select which of the six available inputs are placed on each output (S1 and S2).

Note: only use one jumper per channel



Y Board configured for V1 and integrated V1 output



Y Board jumper pinout. Use 0603 zero ohm resistors

Calibration

Non-contact meters must be calibrated before they can collect current and voltage measurements. You will need a resistive load, a smart plug and a micro-USB cable. The resistive load should be greater than 200W- incandescent light bulbs work well. Once you have chosen a load and the sensors are installed, configure the meter in the meters.yml file as described in [Setting up a Non-Contact Meter](#). To calibrate the meter, open a terminal window and run `nilm-calibrate` with the name of the meter in meters.yml (meter1, meter2, etc.):

```
1 $ nilm-calibrate meterX
```

The script will first provide a summary of the meter configuration. If this looks correct, answer `y`. Any other key will cancel the calibration process. Plug the smart plug into an outlet and connect it to the computer with the USB cable. The status light will alternate between blue and green. Once it is solid green, answer `y` at the prompt for programming the plug. If you already have a plug in calibration mode you can answer `n` and reuse the plug for this calibration. If an error occurs programming the plug wait until the plug LED is solid green and try again.

Once the plug is programmed (or the step is skipped) you will be instructed to connect the load to the first phase. Connect the smart plug to any outlet and then connect the load. Once the load is turning on and off, hit [ENTER]. *Do not hit [ENTER] until the smart plug is turning on and off.*

If the meter is configured for multiphase operation you will be prompted to move the plug to an outlet on each additional phase. Do not worry about identifying which outlet is wired to which phase. The calibration program will automatically detect if you have connected the load to a previous phase and prompt you to move it.

Once the calibration program has run a sufficient number of measurements you will be asked if you want to save the results. Choose *y* to commit the calibration or *n* to discard. If you do not save the results the previous calibration values will be used (if available).

After calibration you may have to start the data capture manually, if you do the script will instruct you to run the following command:

```
1 $ sudo service nilm-capture start
```

The meter LED will turn blue when it is actively recording data.

Software Setup

Introduction

The NILM Software suite provides a complete set of tools to acquire, process, and manage high bandwidth NILM datasets. The following sections cover the software installation and configuration. Usage of particular components of the software suite are covered separately by topic from the main help page. For advanced use see the [NILM command line interface](#) for a full listing of available NILM programs and commands.

Installation

The NILM software suite is divided into data acquisition tools and the web manager. The data acquisition tools can be installed on any Ubuntu distribution (verified up to 15). The web manager is only supported on Ubuntu 14.04.3 LTS (Trusty Tahr). The system requires upstart support which is a proprietary Ubuntu init protocol, therefore the system will not run on other Linux distros. The software is installed off a USB stick. To create the installation media, format a USB stick with a single `ext4` partition and copy over the contents of the `installer` folder which can be found on the Desktop of any NILM system. See [Storage Setup](#) for details on partitioning and formatting a drive.

Install Full Software Suite

Install a fresh copy of Ubuntu 14.04.3 on the system with username `nilm`. Once the installation is complete restart the computer and insert the NILM USB stick. Open a terminal window and change to the NILM USB stick directory. Then run the `install` script as shown below:

```
1 $ cd /media/nilm/<name of usb stick>
2 $ bash install.sh
```

This will take a while to run. When it is complete you should see the following:

```
-----Install Complete-----
***RESTART YOUR MACHINE***
Configure meters in meters.yml on the Desktop
Go to http://nilm.standalone to manage this system
```

If this does not appear, an error occurred. Check the log output for details and correct the error then run the script again.

Configuration

The NILM is completely configured by the `meters.yml` located on the Desktop. This file lists the configurations for each connected power meter. Both contact and non-contact meters can be set up in this file. A single NILM can run multiple meters of either type. Meters are listed sequentially as `meter1` to

meterN. Any changes to this file will only on a reboot or manually restarting the `nilm-capture` service. Some settings may not be changed after a meter is initialized (eg the number of phases), others may be changed at any time (eg the amount of data to keep on disk). The next two sections explain how to configure both types of power meters.

Setting up a contact meter

The contact meter should be installed before it is configured. Make careful note of the current sensor connections and their relative orientations. Run the command below to check the communication link between the computer and the meter.

```
1 $ ping <LabJack IP Address> -c 4
```

This should return with a line similar to

```
--- 192.168.1.209 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 2998ms
rtt min/avg/max/mdev = 0.026/0.031/0.035/0.006 ms
```

Open `meters.yml` in a text editor and copy over the example configuration below. If there is already a `meter1`, give this meter a unique index (eg `meter2`). If you reuse a meter name, for example change `meter1` from a contact meter to a non-contact meter, or change the meter from 2 phase to 3 phase operation, you must either rename or completely flush the previous meter's data streams from the Nilm database. See [command line tools](#) for details on moving and renaming streams in NilmDB.

Example Configuration

Note that all fields are required. Details about each field are itemized below the example. Standard defaults are provided but **sensor_scales** must be set for your specific installation (see [contact meter hardware](#)). The `nilm-scope` utility is useful for ensuring the channel mappings and sinefit rotations are correct.

```

1 meter1:
2   type: contact
3   enabled: true           # set to false to disable this meter
4   ip_address: 192.168.1.209 # default LabJack address
5   phases: 3              # 1 - 3
6   sensors:
7     voltage:
8       sensor_indices: [3,4,5] # maps to phase A,B,C
9       sensor_scales: 0.0919 # built-in constant
10      sinefit_phase: A        # [A,B,C] voltage used by sinefit
11      nominal_rms_voltage: 120 # used to scale prep to watts
12    current:
13      sensor_indices: [0,1,2] # maps to phase A,B,C
14      sensor_scales: XX      # set by resistors and LEM, see hard
15      sinefit_rotations: [0,120,240] # relative to sinefit_phase voltage
16 streams:
17   sinefit:
18     decimate: true         # if [false] only the base stream w:
19     keep: 1m              # how much data to keep as [amount]]
20   iv:                     #   h: hours, d: days, w: weeks
21     decimate: true       #   m: months, y: years
22     keep: 1w             # if [false] no data will be saved
23   prep:
24     decimate: true
25     keep: 3w

```

Contact Meter Fields:

- type** This should be set to contact
- enabled** true/false If false, no data is captured from this meter
- ip_address** The IP address of the LabJack. This defaults to 192.168.1.209 but can be set to another value using the LabJack software (Window's only). When more than one Contact NILM is used or if the 192.168.1.0 subnet is unavailable you must change this default value. Note that each LabJack must be on it's own subnet. See [Network Configuration](#) for more details.
- phases** A value between 1 and 3. This depends on the power system being monitored.

Sensor Configuration

Voltage

- sensor_indices** These are the voltage sensor LabJack channels. The sensors are connected to channels 3 4 and 5. The order of this list is used to name Phases A, B, and C. When

you are installing the system note the phase connection and order this list accordingly. See the figure in [Contact Meter Connections](#) for the voltage sensor connections.

sensor_scales The scaling coefficient to convert sensor output to Volts. This is hardwired to 0.0919. Separate scaling factors may be applied to each sensor by using an array instead of a single value: [XA, XB, XC] where X is the scale factor for each phase.

sinefit_phase A,B,C The phase used by the sinefit processor.

nominal_rms_... This is used to scale prep to approximate power (Watts)

Current

sensor_indices These are the current sensor LabJack channels. The sensors are connected to channels 0 1 and 2. The order of this list should match the order used in the voltage configuration. See the figure in [NEMO Channel Connections](#) for the current sensor connections.

sensor_scales The scaling coefficient to convert sensor output to Amps. See the figure in [NEMO Configuration](#) for details on calculating this value. Separate scaling factors may be applied to each sensor by using an array instead of a single value: [XA, XB, XC] where X is the scale factor for each phase.

sinefit_rotation The phase difference in degrees from the `sinefit_phase` voltage for each phase (note that the coefficient for the phase used by sinefit should be a 0 in this array)

Stream Configuration

The **streams** configuration block determine how much data is retained for each stream and whether or not the stream is decimated as it is collected. Once the NILM has been running for a few hours the [nilm-cleanup](#) command line tool can be used to estimate how much space the database will take up given the amount of data the streams are configured to store. It is very important to make sure the total size required by the database will fit on your storage drive.

decimate `true/false` Whether or not the stream should be decimated. This makes the stream plottable in the web interface but roughly doubles the amount of space required to store this data. There is also some network overhead associated with decimation so if you are experiencing high processor load try disabling this for some streams.

keep How much data to keep for this stream. This is specified as a coefficient and unit. Valid units are `h` for hours, `w` for weeks, `m` for months, and `y` for years. Set to `false` to discard the stream data.

Setting up a non-contact meter

There are many different configurations for the non-contact sensors. See [Non-Contact Sensors](#) on the Hardware help page for detailed information on connecting and configuring the sensor hardware. The [nilm-scope](#) utility can be helpful in finding optimal installation sites for the non-contact sensors.

Open `meters.yml` in a text editor and copy over the example configuration below. If there is already a `meter1`, give this meter a unique index (eg `meter2`). If you reuse a meter name, for example change `meter1` from a contact meter to a non-contact meter, or change the meter from two phase to three phase

operation, you must either rename or completely flush the previous meter's data streams from the Nilmb database. See [command line tools](#) for details on removing and renaming streams in NilmbDB.

```
1 meter1:
2   type: noncontact
3   enabled: true                # set to false to disable this meter
4   serial_number: meterXXXX    # found on the meter case
5   phases: 2                   # 1 - 3
6   sensors:
7     voltage:
8       sensor_index: 0         # electric field sensor
9       digitally_integrate: true # if true, integrate using FIR filter
10      nominal_rms_voltage: 120 # used to scale the electric field
11     current:                 # --uncomment a line below, or custom
12       sensor_indices: [1,3,5,7] # D-Board with 4 A-Boards
13       #sensor_indices: [1,2,3,4,5] # Flex Board
14   calibration:
15     duration: 30              # length of calibration in seconds
16     watts: 200                # power consumed by calibration load
17     has_neutral: true        # [false] if the system has no neutral
18   streams:
19     sinefit:
20       decimate: true          # if [false] only the base stream will be saved
21       keep: 1m               # how much data to keep as [amount]
22       iv:                    # h: hours, d: days, w: weeks,
23       decimate: true         # m: months, y: years
24       keep: 1w               # if [false] no data will be saved
25     prep:
26       decimate: true
27       keep: 3w
28     sensor:
29       keep: false
```

Non-Contact Meter Fields:

- type** This should be set to noncontact
- enabled** true/false If false, no data is captured from this meter
- serial_number** This is printed on the DBoard meter case. See [DBoard Setup](#) for more information.
- phases** A value between 1 and 3. This depends on the power system being monitored.

Sensor Configuration

Voltage

- sensor_index** This is the index of the electric field sensor to used to calculate the effective voltage waveform. Use [nilm-scope](#) to determine which electric field sensor has the strongest signal and use its index here.
- digitally_integr...** `true/false` If the sensor specified in **sensor_index** does not have hardware integration, set this to true. See [non-contact sensors](#) for information on hardware versus software integration
- nominal_rms_...** The rated line voltage. This is used to scale the voltage output

Current

- sensor_indices** These are the magnetic sensor indices. Place all available sensors in this array. There must be at least as many sensors as phases. The order does not matter.

Calibration

- duration** Time in seconds to run the calibration load on each phase. Longer durations can improve calibration results especially in environments with many background loads. Do not use less than 30 seconds on a production system.
- watts** The power draw of the calibration load. This is used to scale the prep output
- has_neutral** `true/false` If the system has a neutral bus (most common), set this to true.

Stream Configuration

The **stream** configuration block determine how much data is retained for each stream and whether or not the stream is decimated as it is collected. The **sensor** stream is the raw output from the DBoard and is always eight channels whether or not all eight sensor inputs are used. Once the NILM has been running for a few hours the [nilm-cleanup](#) command line tool can be used to estimate how much space the database will take up given the amount of data the streams are configured to store. It is very important to make sure the total size required by the database will fit on your storage drive.

- decimate** `true/false` Whether or not the stream should be decimated. This makes the stream plottable in the web interface but roughly doubles the amount of space required to store this data. There is also some network overhead associated with decimation so if you are experiencing high processor load try disabling this for some streams.
- keep** How much data to keep for this stream. The is specified as a coefficient and unit. Valid units are `h` for hours, `w` for weeks, `m` for months, and `y` for years. Set to `false` to discard the stream data.

Storage Setup

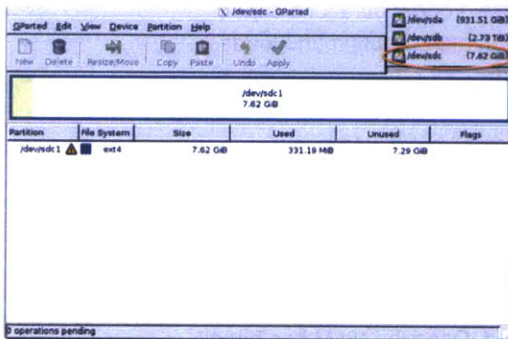
This section explains how to properly format and configure an extra hard drive for a NILM system. It is usually a good idea to place the NilM Database on a seperate hard drive. This prevents the database from filling the primary drive to the point where the system become unusable and also makes it easy to retrieve collected data from an installation by simply swapping out the extra harddrive.

Formatting a drive After installing the drive and booting the system, the first step to use the drive is to place a usable filesystem on it. There are many tools that can be used for this but one of the easiest is GParted. This program must be run as root. From the command line type the following:

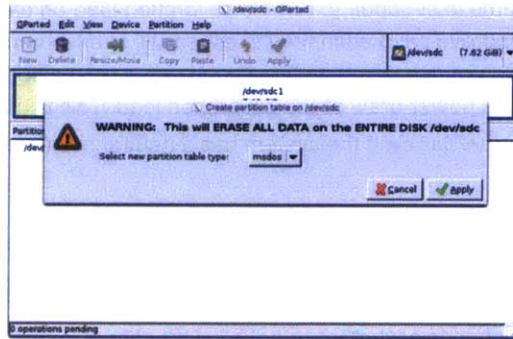
⚠ Be very careful with gparted, formatting the primary drive will destroy the installation

```
1 $ sudo gparted
```

Select the extra drive from the dropdown menu as shown. Generally the extra drive should be **/dev/sdb** but this is not always the case. If the drive already has multiple partitions this most likely means it is the primary drive.

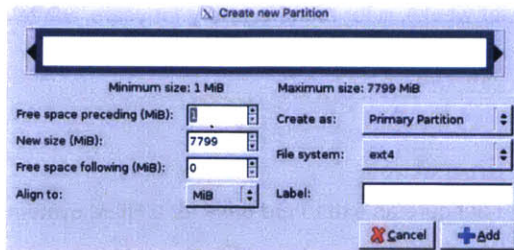


Carefully select the device node for the extra drive



Create a new msdos partition table. This will erase the drive.

Select **Device > Create Partition Table** to bring up the Create Partition dialog. Select **msdos** and click Apply. Select **Partition > New** to bring up the New Partition dialog. Add a new **ext4** partition to the drive and assign it the full extents of the disk (this is the default). Click Add to close the dialog. Finally click **Apply** to format the disk and then close the program.



Add an **ext4** partition to fill the disk

To use the drive for the Nilm database it must be mounted to the correct location in the filesystem. Edit **/etc/fstab** in a word processor and add the following line where **/dev/sdX1** is the name of the drive you just formatted. The number one refers to the first (and only) partition. Note that you will need to run the

word processor as root (sudo) in order to edit this file.

```
/dev/sdX1 /opt/data ext4 errors=remount-ro 0 1
```

Run the following commands to mount the drive and setup the permissions. Use **df** to verify the configuration:

```
1 $ sudo service nilm-capture stop #stop data capture if it is already running
2 $ sudo mount -a
3 $ sudo chown -R nilm:nilm /opt/data #assign the drive to the nilm user
4 $ df -h
```

The output from **df** should look similar to that below:

```
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda1        1.8T  5.3G  1.7T   1% /
... other mount points ...
/dev/sdc1        917G   72M  871G   1% /opt/data
```

If you have already configured your meters and want to start collecting data, run:

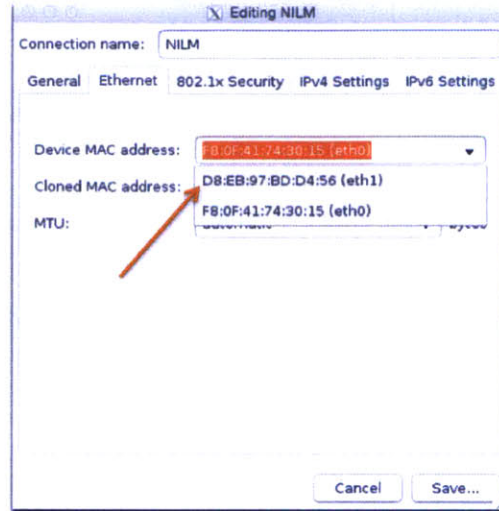
```
1 $ sudo service nilm-capture start
```

Network Setup

The LabJacks in contact meters require an ethernet connection. If you need more ethernet ports than the computer has available, use a USB to ethernet adapter. While most adapters should work out of the box, the recommended adapter is the **TU2-ET100** from TRENDnet. After you connect the contact meter to the computer note the MAC address of the USB adapter. This is found on the backside of the TU2-ET100 as shown in the figure below. Open the network connection editor and configure the interface for the MAC address used by the NILM. Each interface has a unique MAC address and it is helpful to rename the connection to "NILM" or "Web" depending on the role of the interface. For interfaces connecting to the web, set the IPv4 address method to **DHCP** from the **IPv4 Settings** tab.

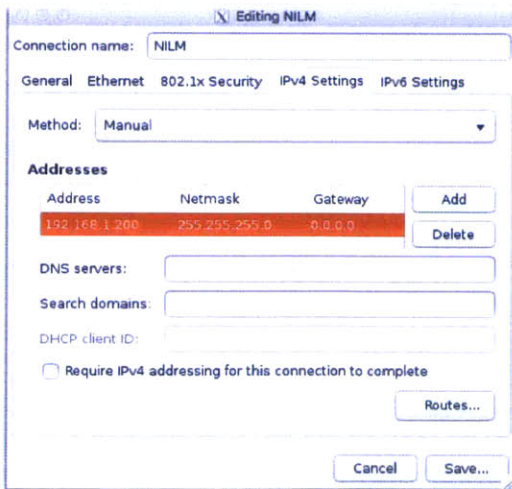


Find the MAC address for the LabJack connection. On the TU2-ET100 this is on the bottom sticker.

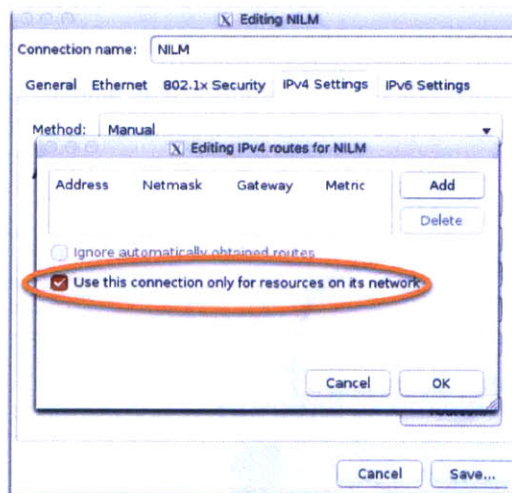


Configure the network interface for the MAC address associated with the LabJack.

For interfaces connected to a contact meter select the **IPv4 Settings** tab and set Method to **manual**. Click **Add** and enter an IP address on the same subnet as the LabJack. Unless you have set up an advanced configuration the IP address should share the first three numbers as the LabJack and use a different final number. For example, the default LabJack address is 192.168.1.209 and a valid host IP address would be 192.168.1.200. The **Netmask** should be 24 or 255.255.255.0 and the **Gateway** should be left blank. Note that each interface must be on a separate subnet. If you need more than one contact meter you will have to assign a different IP address to the additional LabJack devices. This is done with the commercial configuration tool available through the [LabJack website](#).



Assign an address in the same subnet as the LabJack



Select the local routing check box in the Routes dialog

For multiple LabJack installations address them as 192.168.X.209 where X is different for each device (between 1 - 255). After you assign the IP address click **Routes...** and select the checkbox for local routing only as shown in the figure. This prevents web bound traffic from getting sent to the LabJack. Once you have completed the IP address configuration click **Save...** to close the network configuration dialog. Open a terminal and enter the following command to reconfigure the network interfaces to use the new values:

```
1 $ sudo service network-manager restart
```

Cluster Setup

The NILM system can be managed by a Wattsworth Cluster. If the system is connected to a cluster, it cannot be managed locally. Using the local NILM Manager site (`nilm.standalone`) while connected to a cluster will corrupt the NILM filters and analyzers. There are two different types of clusters: **VPN** and **Peer-to-Peer**. The **VPN** cluster is managed by a central server which provides secure remote access to any NILM with a Internet connection. The **Peer-to-Peer** cluster allows one NILM to manage another NILM. This configuration can be used to manage a "headless" NILM from another NILM that has the Manager framework installed.

1.) Stop Local Manager: Before joining a cluster, make sure that the local NILM Manager cron tasks are disabled. Do this by commenting out the NILM Manager lines. See the comments in the crontab for more details. To edit the crontab open a terminal and run the following command:

```
1 $ crontab -e
```

2.) Select a Cluster Type: Decide which type of cluster you want to join. The **VPN** cluster requires an Internet connection and the **Peer-to-Peer** cluster only requires a network connection between the two NILMs.

2a.) Join a VPN Cluster: Open a terminal and enter the following commands below. This will require you to enter the cluster administrator's password so have it ready before you begin.

```
1 $ cd ~/Desktop/cluster #switch to the cluster folder on the Desktop
2 $ sudo ./join-cluster.sh cluster_name
```

The `cluster_name` is the fully qualified domain name (FQDN) of the cluster server. Omit this argument to see a list of available clusters. Wait for about a minute and run the command:

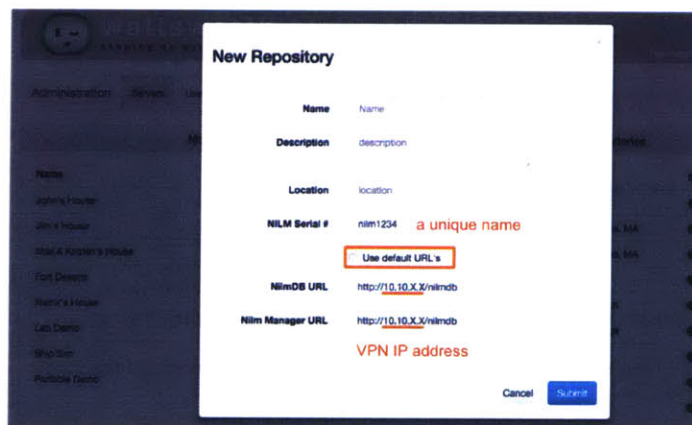
```
1 $ ifconfig
```

This will display all the current network interfaces. Look for an interface that begins with `tun` usually this will be `tun0`. Find the IP address for this interface, it will look like `10.x.x.x`. Write this down, you will need it for the next step. If this command has no output this means the NILM was not able to join the cluster VPN. To debug VPN problems, run the command below and fix any errors that are displayed:

```
1 #FOR DEBUGGING ONLY#
2 #run this if there is no tun interface and fix any errors that occur#
3 $ sudo openvpn /etc/openvpn/client.conf
```

2b.) Join a peer-to-peer Cluster: You must establish a network connection between the two NILMs. Connect the systems with an ethernet cable and assign both NILM's static IP addresses on the same subnet. Record the IP address of the *managed* NILM for the next step.

3.) Add the NILM: Log in to the Cluster NILM Manager site as an administrator and open up the admin panel. Click **Add Repository** to bring up the New Repository dialog. Fill in the name, description and location fields. Uncheck the "default URL" box to display the custom URL text boxes. The Nilmdb URL should be `x.x.x.x/nilmdb` and the Nilm Manager URL should be `x.x.x.x/nilmrun` where `x.x.x.x` is the IP address (which is found using the steps above).



4.) Leave the Cluster: If you want to remove the NILM from the cluster delete it from the `server` tab in the Global Administration panel. See [Global Administration](#) for details.

4a.) Leave a N cluster

```
1 $ cd ~/Desktop/cluster #switch to the cluster folder on the Desktop
2 $ sudo ./leave-cluster.sh <vpn-name>
```

Then re-enable the local NILM Manager cron tasks by uncommenting the lines commented out in **Step 1**

Secondary Setup

The NILM supports a secondary drive that can be used to copy datasets from the primary database or preview data collected from another NILM system. Any storage device can be used as a secondary drive including internal SATA drives, USB flash drives, and eSATA drives. The drive must be an ext4 volume. See [Storage](#) for details on using GParted to format a drive. Ubuntu often automounts removable media, if it does click the eject button on the Desktop to unmount the volume. Compare the output of the following command before and after you connect the drive to determine the assigned device name:

```
1 $ ls /dev/sd*
```

When the drive is connected the output should include an extra `/dev/sd<letter>1` where `<letter>` is b,c,d etc. This is the device name. Now mount the volume to the secondary disk location

```
1 $ sudo mount /dev/sdXX /opt/secondary
2 $ sudo chown -R nilm:nilm /opt/secondary #update drive permissions
```

Restart the apache web server to initialize the secondary system:

```
1 $ sudo apache2ctl restart
```

Once Apache restarts you will be able to access the backup server at `http://nilm.secondary/nilmdb`. To view the data using the NILM Manager website refresh the secondary database from the administration interface. See [NILM Administration: Database Tab](#) for details. Also, any nilmtool commands can be used on the secondary server. The most useful commands are the following:

```
1 $ nilm-copy-wildcard -u localhost/nilmdb -U nilm.secondary/nilmdb /meterX/
2 $ nilm-copy-rename -u localhost/nilmdb -U nilm.secondary/nilmdb /meterX/
```

The first will copy all streams associated with `meterX` to the secondary volume using the same name. The second command renames the streams which can be helpful when using a single drive to backup multiple NILM's each with a `meterX` group. Run either command with the `-h` flag to see a full list of options. When you are done copying streams, run the following commands to stop the secondary server and unmount the drive:

```
1 $ sudo umount /opt/secondary
2 $ sudo apache2ctl restart
```


Smart Plugs

Introduction

The NILM smart plug is a WiFi-enabled plug with a software controlled relay and solid state power meter. These plugs can be controlled over a wireless network or by USB. There are three main tools to interact with these plugs. The easiest tool is the command line `nilm-plug` utility which can control most plug features over USB or WiFi. For more advanced operation the plug has a terminal mode that is accessed using `nilm-plug --cli`. See [Plug CLI](#) for more details. Finally there is a set of python modules for scripting and integration with NILM filters. See [Plug API](#) for more details. In order to use the plugs on a network a wireless access point should be set up with MAC address reservations and static IP bindings for each plug. See the next section for details on configuring a wireless access point.



Smart Plug connected to a NILM over a wireless network. The green LED indicates the plug has authenticated to the network.



Smart Plug connected to a NILM by USB. Blue LED indicates an active USB communication between the NILM plug.

WiFi Setup

Any standard wireless access point can be used to interact with NILM smart plugs. This documentation covers the TP-Link **TL-WR841N**. A similar sequence of steps should work on most devices. When setup for the first time the router will advertise a default SSID. The network name and password can be found on the back of the router as shown the figure below. Open a browser and navigate to <http://tplinkwifi.net>. Authenticate with username [admin] and password [admin].

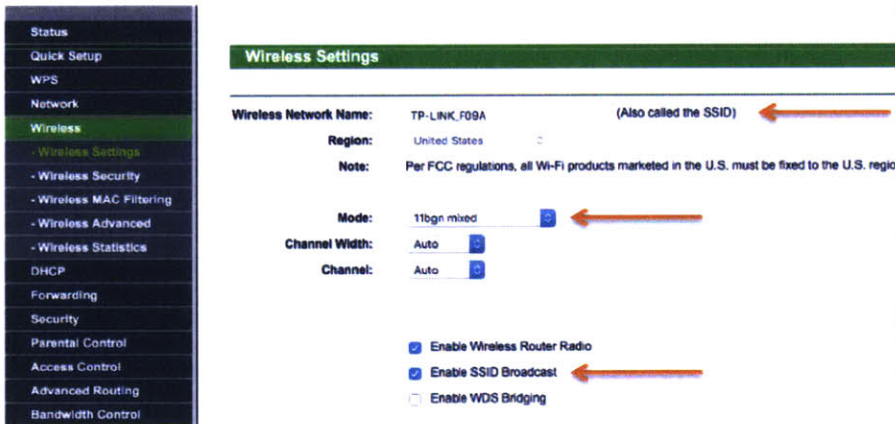


TP-LINK TL-WR841N Wireless Router



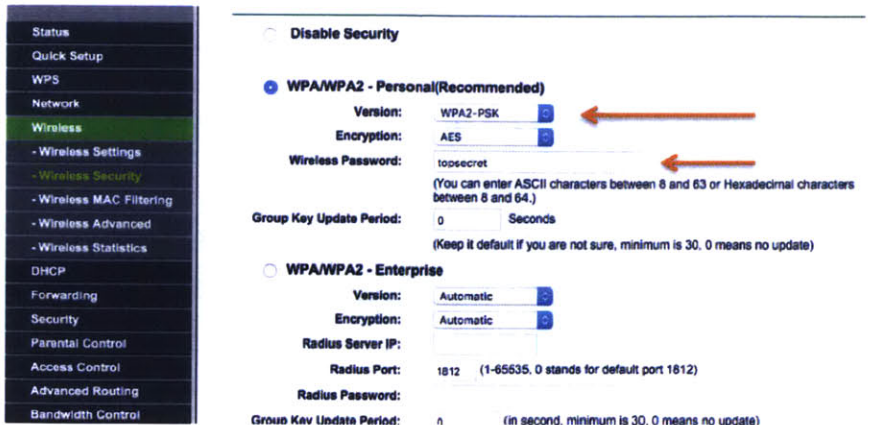
The default SSID and password are on the back of the router

Change the wireless network name (SSID). Do not use spaces in the name. If you are configuring multiple access points that will be in close proximity make sure each has a unique name. Make sure the mode supports 802.11a, b and g. This is the default. The broadcast SSID checkbox may be disabled if you do not want to advertise the presence of the network. This helps prevent people casually trying to connect to the network. Note that this does not improve the security of the network. If you change the name of the network you will have to reconnect to the new SSID once the router restarts.



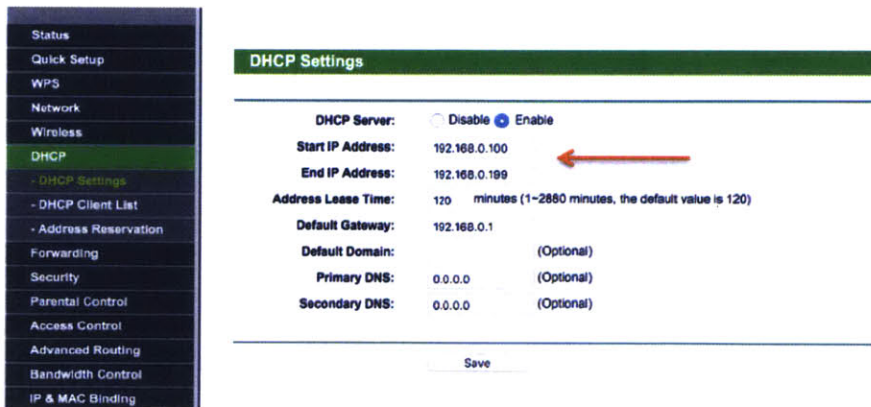
Configuring the wireless access point for NILM smart plugs

While not strictly necessary it is strongly recommended to enable wireless security. The plugs work best with WPA2 Personal. The authentication should be set to PSK (pre-shared key) and AES encryption. Select a strong password and save the settings. You will have to reconnect to the network using the new password.



Recommended security settings for the wireless access point.

The NILM smart plugs are controlled using their IP addresses. Wireless networks by default assign random addresses to each client. The router must be configured to assign a consistent fixed address to each plug. Each plug as a unique MAC address which must be entered into the router to reserve an address. The first step is to make sure DHCP services are enabled on the router and set to a range of addresses that do not overlap with any other networks on the machine including contact meters. The default address space should work well for most configurations. The router will use a pool of the addresses in this space to assign to clients as they connect. This range is specified in the boxes highlighted below. You will need to assign IP addresses to the plugs that are *outside* of this range. Here the DHCP server will use addresses above *.100 leaving *.2-*.99 available for reservations. Note that *.0 and *.1 are reserved and should not be used for plugs.



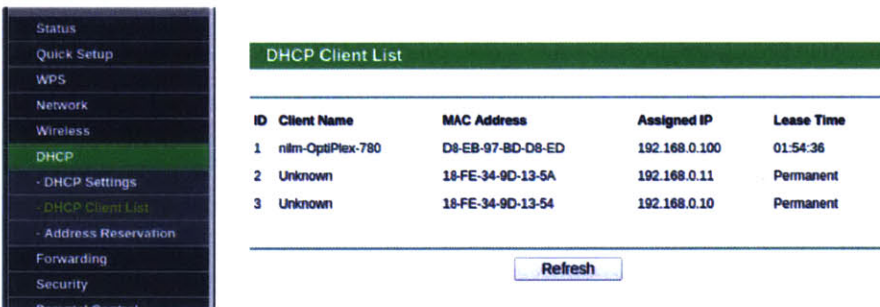
Configuring DHCP services on the wireless access point.

Create and address reservation for each plug and record the mapping so that you will know which plug is attached to which address. Note that the format for the MAC address field is XX-XX-XX-XX-XX-XX. The first three octets are the organization identifier (OUI) and are the same for all the plugs. The last three are unique for each plug. The MAC address is labeled on the case of each plug.



Create address reservations for each plug. The address should not be in the DHCP dynamic address range.

Connect the host NILM either with WiFi or to one of the local (yellow) ethernet ports on the back of the router. If it is connected to the WAN port (blue) the NILM will not be able to communicate with the plugs. Once the NILM and plugs are configured you should see a status similar to the figure below. All of the plugs (recognizable by their OUI) should have reserved addresses and the computer should have a leased address.



The DHCP Client List shows the current address assignments. Plugs should have a reserved address (Permanent)

nilm-plug

The easiest way to interact with the NILM smart plug is with the `nilm-plug` command line utility. It can be used with USB or WiFi connected plugs. The program takes two arguments, an `action` and a `device_address`. If a single plug is connected over USB the address can be omitted because the system can find the plug automatically. When multiple plugs are connected by USB or plugs are connected over WiFi the address must be specified. For plugs connected by USB the address is `/dev/ttyACM#` where `#` is a number dynamically assigned to the plug by the operating system. Run `ls /dev/ttyACM*` before and after connecting the plug. The new entry is the plug's `device_address`. The `device_address` for wirelessly connected plugs is their IP address.

Read plug data

```
nilm-plug --read [--file filename] [plug_address]
```

This command retrieves meter data from the plug and stores the result in a comma separated text (CSV) file. If no file is specified the output is written to the file `plug.dat` in the current directory. If the file exists, data is appended to the end. Each row of data has eight values specified below:

```
ts | vrms | irms | watts | pavg | pf | freq | kwh

ts  Timestamp in milliseconds since 1970 (UNIX time)
vrms RMS voltage (V)
irms RMS current (A)
watts Current power usage (W)
pavg 30 second average of power usage (W)
pf   Power Factor
freq Line Frequency (Hz)
kwh  Energy used since last plugged in (kWh)
```

The plug collects data in one minute packets. The timestamp for each packet is displayed as it is retrieved from the plug. For plugs connected wirelessly only the most recent data packet is available. If a wireless plug is queried before a new packet is ready it will return an empty packet and the output file will remain unchanged. If a plug is connected by USB, the read command returns all of the data packets stored on the SD Card. If the output file already exists, only new data is appended to the end of the file. This command is designed to be run iteratively with each plug using a separate output file.

If a plug is read wirelessly it still stores data to the SD Card which can be retrieved later over USB. This allows you to query the plug over an intermittent WiFi connection and fill in any missing data packets once the plug is connected by USB

Erase plug data (USB only)

```
nilm-plug --erase [plug_address]
```

This command erases all data from the plug's SD Card. Data can only be erased when a plug is connected by USB.

Read and erase plug data (USB only)

```
nilm-plug --read_erase [--file filename] [plug_address]
```

This command is identical to running `read` and then `erase`. This is the recommended command to use with USB plugs to prevent the SD card from filling up with data that has already been retrieved.

Control plug relay

```
nilm-plug --relay on|off [plug_address]
```

This command turns the plug on or off. If the requested relay state matches the current relay state this command is ignored. This works for both USB and WiFi plugs.

Display help and examples

```
nilm-plug --help
```

This command displays usage examples and a copy of this documentation.

Plug API

The Plug Application Programming Interface (API) allows you to control NILM smart plugs from Python scripts. Before you can use the API in a script you must import the nilmplug module:

```
1 from nilmplug import plug
```

This module contains a single object `Plug`. The constructor takes two arguments, the `device_address` and a flag to indicate whether this address is a USB device. The code below shows examples of both plug types:

```
1 #manage the plug over a USB cable
2 my_plug = plug.Plug("/dev/NODE",usb=True)
3 # *OR* manage the plug over wifi
4 my_plug = plug.Plug("192.168.1.XX")
```

The API provides methods to read meter data, control the relay and set the LED color over USB or WiFi. The plug data can be erased over USB.

Read plug data

```
1 my_plug.get_data(last_ts=0)
2 #return data *after* last_ts
```

This method returns a numpy array of data. The columns are described [here](#). If the plug has no new data this will return an empty array. The optional argument `last_ts` limits the returned data to samples *after* this timestamp. This can be helpful when you don't want to erase the data after reading it. Setting this value to the last timestamp received from the previous read will return only new data. Omitting this parameter will return *all* data on the SD card.

Control plug relay

```
1 my_plug.set_relay(value)
2 #value "on" or "off"
```

This method controls the plug relay. The parameter `value` must be either "on" or "off". This method returns 0 on success or -1 on error. Errors can be caused by a poor WiFi connection.

Control plug LED

The method controls the multicolor LED. By default the LED is green when the plug is on and running normally, blue when it is actively connected over USB, and red if an error has occurred.


```
1 my_plug.set_led(red,green,blue,blink)
2 # red,green,blue: 0-255
3 # blink: 0 solid, rate in ms
```

The red, green, and blue parameters are 8 bit values (0–255). Many online references provide common RGB color combinations. The blink value sets the blink rate in milliseconds. Set to 0 to disable blinking (solid).

Erase plug data (USB only)

```
1 my_plug.erase_data()
```

This method removes all data from the plug's SD card. This method can only be used with USB connected plugs.

Plug CLI

NILM smart plugs provide a complete command line interface accessible with a standard terminal emulator. The `nilm-plug` program provides a built-in terminal interface that is started by using the `--cli` flag:

```
1 $ nilm-plug --cli
2 /dev/smart_plug, 115200 baud
3 ^C to exit
4 -----
5 Wattsworth WEMO(R) Plug v1.0
6 [help] for more information
7 >
```

The following sections explain the commands available through this interface. You can also use the `help` in the CLI to see a brief summary of command options.

calibrate - start or stop calibration mode

Usage

```
calibrate stop|start on_time off_time
```

Description

Run the plug in calibration mode. This disables data collection and toggles the relay with a duty cycle of `on_time` and `off_time` milliseconds. This is used to calibrate non-contact power meters. Use the `stop` flag followed by the `restart` command to return the plug to normal operation. This is a persistent setting which means that a plug will remain in calibration mode until set otherwise.

Arguments

action stop|start
on_time duration the plug is on in milliseconds
off_time duration the plug is off in milliseconds

config - get or set a configuration parameter

Usage

config action parameter [value]

Description

Retrieve or set the plug configuration parameters. Parameters are persistent and stored on the SD card. If a blank SD card is inserted into a plug the default parameters are empty strings unless otherwise specified. The last three parameters are used to support DHCP environments where neither the plug or the NILM IP addresses are fixed. If you are using IP address reservations as recommended in [WiFi Setup](#), these parameters should be left blank.

Arguments

action get|set
parameter configuration parameter, see list below
value new value of parameter, leave blank to clear parameter setting

Parameters

wifi_ssid wireless network name
wifi_pwd wireless network password. Leave blank for open network. (*write only*)
standalone [true|false] If true, do not attempt to connect to a wireless network. Default is true
serial_number unique string to indentify the plug
nilm_id the NILM associated with this plug (*optional*)
nilm_ip the IP address of the associated NILM (*optional*)
mgr_url management website URL (*optional*)

Example

```
1 > config set wifi_ssid nilmplug
2 Set [wifi_ssid] to [nilmplug]
3 > config get serial_number
4 plugAD49
```

data - read or clear the data

Usage

data action

Description

Use this command to retrieve data packets stored on the SD card. The data is returned in the binary format described below and should only be used in scripts, not for printing to the terminal. If you want

to view data from the command line, use the `meter` command. Use the `erase` flag to erase the data file.

Arguments

```
action [read|erase]
```

Structure

Data is returned in packet chunks with the binary format shown below. End of file (EOF) is signaled by a packet containing the character `x` in every byte. See the `plug.py` source code for an example of how to parse this data structure in python.

```
1 //binary structure returned by [data read] command
2 #define PKT_SIZE 30
3 #define PKT_TIMESTAMP_BUF_SIZE 20
4 typedef struct power_pkt_struct {
5     int32_t vrms[PKT_SIZE];    //RMS voltage
6     int32_t irms[PKT_SIZE];    //RMS current
7     int32_t watts[PKT_SIZE];   //watts
8     int32_t pavg[PKT_SIZE];    //Average power (30s window)
9     int32_t freq[PKT_SIZE];    //Line frequency
10    int32_t pf[PKT_SIZE];       //Power factor
11    int32_t kwh[PKT_SIZE];      //kWh since turn on
12    char timestamp[PKT_TIMESTAMP_BUF_SIZE]; //YYYY-MM-DD HH:MM:ss
13    uint8_t status;            //struct valid flag
14 } power_pkt;
15
16 //end of file signal returned by [data read] command
17 char* EOF = (char*)malloc(sizeof(power_pkt));
18 memset(&EOF, 'x', sizeof(power_pkt));
```

rtc - get or set the real time clock (RTC) value

Usage

```
rtc get|set year month date dw hour min sec
```

Description

Get or set the value of the real time clock. The battery backed rtc is used to timestamp data collected by the plug. Query the current time using `get`, or set the time and specify the full date, see the arguments below:

Arguments

get	return the current time formatted as a string. If the battery fails this will report a corrupt date
set	set the clock value, all of the following parameters must be set to valid values for this command to execute successfully
year	year as a two digit number 2016 = 16
month	two digit value 0-12
date	two digit value 0-31
dw	day of week 0-7
hour	two digit value 0-24
min	minute as a two digit value 0-59
sec	second as a two digit number 0-59

Example

```

1 > rtc set 16 02 01 01 03 16 51
2 Set time to: 2016-02-01 03:16:51
3 > rtc get
4 Time: 2016-02-01 03:16:51

```

log - read or clear the log

Usage

log action

Description

The SD card records a persistent log of plug events. This includes network connectivity, general errors, and system restarts. A [general reset] occurs when the plug is connected to power and a [software reset] is indicates the **restart** command was used. Use the **clear** flag to erase the log file.

Arguments

action [read|erase]

Example

```

1 > log read
2 ...
3 [2016-01-29 14:59:06]: general reset
4 [2016-01-31 03:41:41]: no requests from NILM, resetting WiFi
5 [2016-01-31 03:42:00]: Joined [nilmplug] with IP [192.168.0.11]
6 [2016-01-31 03:44:25]: software reset

```

relay - control the plug relay

Usage

```
relay action
```

Description

Control the plug relay to turn the connected appliance on or off. Executing this command with the existing relay state has no effect.

Arguments

```
action    on|off
```

Advanced Commands

The following commands are specialized diagnostic tools that are less commonly used. They are listed alphabetically

```
collect_data - start or stop power logging
```

Usage

```
collect_data action
```

Description

Start or stop data collection from the power meter. By default the plug collects data from the power meter. This command can be useful for debugging communication with the solid state meter hardware.

Arguments

```
action [true|false]
```

```
debug - set debug level
```

Usage

```
level
```

Description

Controls the verbosity of console information. Set to 5 to see AT command traffic to the ESP8266.

This is useful for diagnosing network connectivity issues.

Arguments

```
level [0-5] Default is 0 (lowest). Level >=3 show wireless TX messages. Level >=4 echo ESP8266 AT traffic
```

Example

The wireless interface is controlled by a separate ESP8266 module which communicates with the main processor through a UART with AT commands. The following shows the output of a successful wireless bootstrap. Note that the output of this debug session also displays the plug MAC address

(CIFSR:STAMAC). This should be labeled on the plug case but if it is not present use the combination of these two commands to find it.

```
1 > debug 4
2 debug level: 4
3 > wifi on
4 AT+RST
5
6 OK
7 c\xfec\xcfRSvfgj\xd7\xe2j\xd3jS&\xeaJ\xf3\x82k\xfaf\xd2fW\xf2@
8 Ai-Thinker Technology Co. Ltd.
9
10 ready
11 AT+CWMODE=1
12
13 OK
14 AT+CWJAP="nilmplug","topsecret"
15
16 OK
17 AT+CIFSR
18 +CIFSR:STAIP,"192.168.0.11"
19 +CIFSR:STAMAC,"18:fe:34:9d:13:5a"
20
21 OK
22 AT+CIPMUX=1
23
24 OK
25 AT+CIPSERVER=1,1336
26
27 Joined [nilmplug] with IP [192.168.0.11]
28 wifi on
29 >
```

echo - turn echo on or off

Usage

echo action

Description

Turn the prompt echo on or off. This can be useful when interacting with the plug via scripts. This value is not persistent and defaults to `on`.

Arguments

`action` `on|off`

```
led - set the led
```

Usage

```
led red green blue blink
```

Description

Set the LED color and control the blink rate. This overrides the default color but any change to the system (eg, error or USB connect event) will change the LED back the system setting. This is not a persistent setting, it resets to the system default on powerloss.

Arguments

`red` `[0-255]`
`green` `[0-255]`
`blue` `[0-255]`
`blink` blink rate in milliseconds, set to 0 for no blink. Maximum value is 65536.

```
ls - view files on SD Card
```

Usage

```
ls no arguments>
```

Description

View SD card file statistics, similar to the standard linux `ls` command. There should be three files.

The timestamp is the last modified date as recorded by the plug RTC. The files are:

`log.txt` `system log`
`config.txt` `persistent configuration settings`
`power.dat` `data collected by the plug power meter in binary format`

Example

```
1 > ls
2 FILE          SIZE          DATE
3 log.txt       29445         16/01/31 03:44
4 config.txt    197           16/01/28 19:41
5 power.dat     825984        16/01/31 03:43
```

memory - show memory statistics

Usage

memory *no arguments*>

Description

The plug firmware dynamically allocates memory using a set of reserved blocks similar to a heap. This command is useful for debugging memory allocation. A steadily increasing allocation percentage indicates a memory leak in the code.

Example

```
1 > memory
2 Allocated 200 of 11200 bytes (1%)
3 Largest free block: 1000 bytes
4 Smallest free block: 200 bytes
```

meter - view plug meter

Usage

meter *no arguments*>

Description

Displays the last full data packet collected by the meter. If the meter is not collecting data an error message is displayed instead. This is a convenience function for debugging see data command for retrieving meter data.

Example

```
1 > meter
2 **this data may be up to a minute behind**
3 _____
4 voltage | 119.25
5 current | 0.51
6 watts   | 59.62
7 avg pwr | 55.30
8 freq    | 59.59
9 pf      | 0.95
10 kwh    | 0.81
```

restart - software reset

Usage

restart bootloader

Description

This command issues a soft reset to the processor. The optional `bootloader` flag restarts the processor in SAM-BA bootloader mode. This should only be used when the plug needs to be reflashed over USB

Arguments

`bootloader` boot into SAM-BA firmware **this should only be used for reflashing the plug**

version - firmware info

Usage

version *no arguments*>

Description

Print the firmware version and compilation date. The firmware version is set by the `VERSION_STR` define in `inc/monitor.h`.

Example

```
1 > version
2   Firmware [v1.1]
3   Date: [Jan  5 2016 17:08:53]
```

wifi - turn wifi on or off

Usage

wifi action

Description

Turn the wifi system (ESP8266) on or off. The wifi system is turned on a system boot if `wifi_ssid!=""` and `standalone==false`. Issuing this command with the `on` will reset the wifi module and attempt to connect to the specified network. Use in conjunction with the `debug` command to diagnose network connectivity issues.


Arguments

action `on|off`

Administration


Introduction

There are two interfaces for administration. The **system administration** interface and the **global administration** interface. The system interface is used to configure and manage NILM systems. The global interface is used to manage the manager site itself (this website).

The **system administration** interface is accessible from the dashboard (or click [here](#) to open it in a new tab). From the main administration page find the installation you are interested in and click . If you are on a standalone system there will only be one installation listed. If you are on a clustered system there may be multiple installations listed. Once you select a particular installation you will see that installation's administration page.

The **global administration** interface is accessible from the `admin` link on the page header (or click [here](#) to open it in a new tab). Note that you must have admin privileges to view this interface and the header link will be hidden if you do not have access. The global interface allows you to configure user accounts and add or remove NILM systems. This interface is available on all manager sites but is most useful on cluster managers and not standalone systems.

System Administration

The System Administration interface is used to configure a particular NILM system. Access this interface by selecting the *Administration* tile from the dashboard and then clicking  on a NILM system. This button will only be available on NILM's you administer. The interface has four main tabs, *Setup*, *Database*, *Processes*, and *Cloud*. Each of these tabs are explained below.

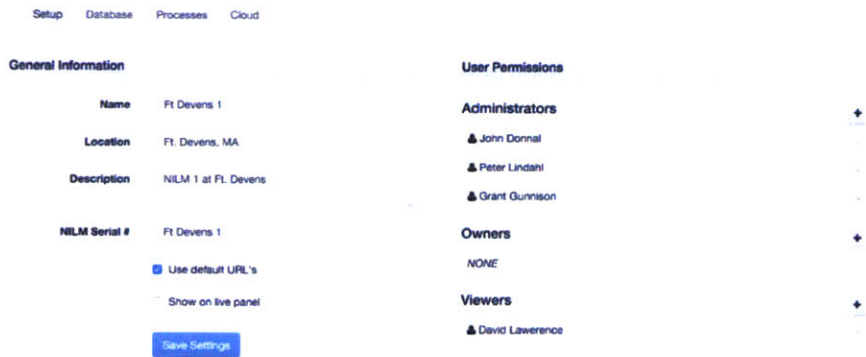
Setup Tab

This tab configures general information about the NILM and user permissions. See the figure below for an example. The **Name**, and the **Serial #** must be unique, other general information fields must be present but not necessarily unique. On a standalone system most of this information is not important, but on a cluster this helps identify a particular machine. You must **uncheck Use default URL's** and configure a custom address when adding a NILM to a cluster- see [Wattsworth Clusters](#) for details. If you would like to set a live meter view instead of the Dashboard as the site homepage, check **Show on Live Panel**, then select a meter dataset to display. Any user that has permissions on this NILM will see this meter in their live panel. Users who do not have permissions on this NILM will not be affected. Any changes to these settings will only take effect if you click the *Save Settings* button.

Permissions are configured on the righthand side of the interface. There are three levels, *Administrator*, *Owner*, and *Viewer*. Administrators are the highest level and Viewers are the lowest. Each level has the following permissions:

Administrator	access system administration interface, run processes (filters and analyzers), view data
Owner	run processes (filters and analyzers), view data
Viewer	view data

To add a user to a permission group, click the **+** button next to the permission. To remove a permission click the **x** next to the user's name. To change a user's permission level, remove the current permission and add them back to the desired level.



System Administration *Setup* Tab. Configure general information and user permissions.

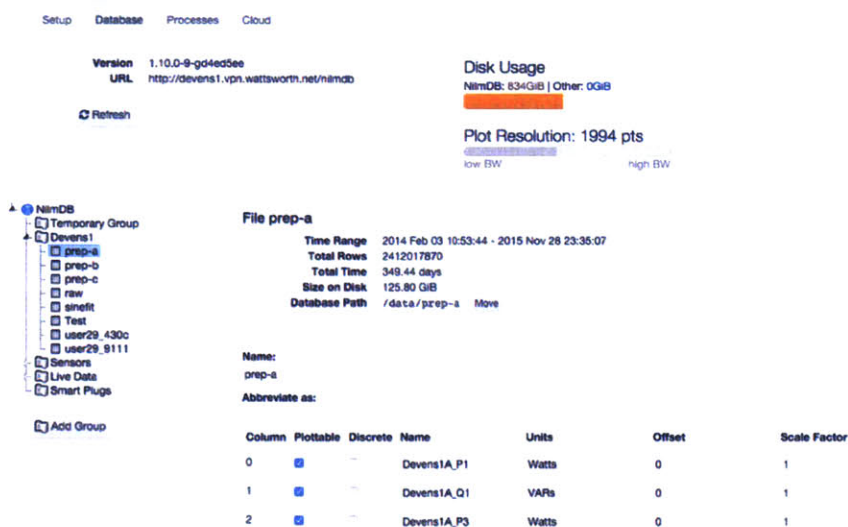
Database Tab

This tab provides information about the NILM database and has numerous tools for configuring the data streams. The interface is divided into three sections: *General Information*, *Database Tree*, and *Group/File Configuration*.

General Information: The top section shows general information about the database. The **Version** is the NilMDB software version and the **URL** is the location of the database. This will be a localhost URL on a standalone system and will be an IP address on a cluster system. The **Refresh** button updates the interface with the latest values from the database. The system automatically refreshes every half hour but in some cases it is helpful to force a refresh (for example when adding and configuring new meters). The **Disk Usage** chart shows how much space is used by NILM data in orange and how much space is used by the rest of the system in blue. The keep settings in `meters.yml` should be set appropriately so the disk does not fill completely with NILM data. The **Plot Resolution** slider configures how many samples are displayed in in *NILM Explorer* plots. Higher plot resolution requires more bandwidth and processing time on the NILM system. In general standalone systems should be configured for highest bandwidth and

clustered systems should be configured for a bandwidth that is appropriate for their network connection. Adjusting this slider automatically updates the server. Adjust this slider as you work with a plot to find an appropriate setting.

Database Tree: The left panel shows a tree view of the database. Database groups are listed below the NilMDB node. Clicking on the triangle next to a group toggles visibility of the files beneath it. Clicking a group or a file will select the node and display configuration options for it in the *Group/File Configuration* panel. The **Add Group** button adds a new empty group to the database. Group names must be unique. Groups can only be added when a connection to the NilMDB system is available which may be intermittent on a cluster system depending on the quality of the network connection.



System Administration *Database* Tab. View and Configure the NILM database.

Group/File Configuration: The center panel is reserved for group and file configuration options. Select a group or file to configure by clicking it in the *Database Tree*. If you change any values click the **Save Changes** button to submit the changes, or click **Reset** to change all values back to their current state. To remove a group or file click the **Delete** button. This completely erases all of the data in the group or file and is not reversible. To prevent accidental deletion, click the **Lock** button. This removes the option to delete the file or group. To delete a locked object you must manually erase each stream in the group or file from the command line using `nilmtool destroy`.

When a group is selected the interface will look like the left hand figure below. The group name must be unique and the description should be a short sentence or phrase. The time range is the maximum extent of the files in this group, and the size is the sum of the file sizes. The **Hidden** checkbox controls whether the group is visible in the NILM Explorer interface.

When a file is selected the interface will look like the right hand figure below. The summary information at the top shows the extents of the data, number of samples and size on disk. Note that the total time covers the intervals with data which may be much than the Time Range than the extents if there are gaps in the data. The `keep` settings in `meters.yml` track the Total Time parameter and not the Time Range. The **Database Path** is the NilMDB stream name for the file. The **Move** button allows you to move the file to a different group. This should only be used for static datasets. *Do not move active files, that is, inputs or outputs to a NILM process or raw streams like prep or sinefit.*

The file **Name** must be unique in the group. **Abbreviate as** is an optional field that is added to the plot legend in NILM Explorer. If it is blank the legend will just be the stream name. Below these fields is an array of entries to configure each column (stream) of data in the file. Select **Plottable** to display the stream in NILM Explorer. Select **Discrete** to plot the data as vertical bars instead of a line graph. **Name** must be unique in the file. **Units** should be an SI value or descriptive word like "event". NILM Explorer requires that all datasets on an axis share the same unit. If unit is left blank the stream will only be plottable with other blank unit streams. The **Offset** and **Scale** fields are floating point values that can apply a linear transform to the data. The formula for the transform is:

$$y = (x - \text{offset}) * \text{scale}$$

The transform is applied "live", that is the underlying data is never changed. The transform is used both for NILM processes (filters and analyzers) and in NILM Explorer. **Default Min** and **Default Max** fix the auto scale bounds to the specified values. If left blank the plot will autoscale to the extents in the plotted dataset. This is useful if you want autoscale to provide a consistent scale to help building an intuition of what is "large" and "small". Otherwise zooming into any dataset and clicking autoscale will make the result fill the plot window. You must click **Save Changes** before selecting any other file or group from the *Database Tree* otherwise your changes will be lost.

Group USCGC Spencer NC

Time Range 2015 Dec 04 15:35:41 - 2016 Feb 19 10:16:33
 Size on Disk 56.73 GiB

Name USCGC Spencer NC Description meter0003

Hidden

[Save Changes](#) [Reset](#)

Group Configuration options

File g1-prep-a

Time Range 2015 Dec 09 13:54:17 - 2016 Feb 19 10:16:33
 Total Rows 72609000
 Total Time 70.02 days
 Size on Disk 29.75 GiB
 Database Path /spencer/nc/prep [Move](#)

Name: g1-prep-a

Abbreviate as:

Column	Plottable	Discrete	Name	Units	Offset
0	<input checked="" type="checkbox"/>	<input type="checkbox"/>	P1	W	0
1	<input checked="" type="checkbox"/>	<input type="checkbox"/>	Q1	W	0
2	<input checked="" type="checkbox"/>	<input type="checkbox"/>	P3	W	0

File Configuration options

Processes Tab

TODO


Cloud Tab

This version of NILM Manager does not provide cloud support

Global Administration

The global administration interface is used to add users or NILM's to the system. This is primarily used on cluster systems although the interface is available on standalone systems as well. To access this interface your user account must have administrative privileges. The interface is composed of three tabs, *Servers*, *Users*, and *Data Views*. Each of these tabs is described below.



Server Tab

The server tab is used to add NILM's to the system. The only type of NILM that can be added to this version of the manager is a **Repository**. Click [Add Repository](#) to access the new repository dialogue. Full details on adding NILM's to a cluster using this interface are in [Wattsworth Cluster](#). To remove a NILM from the system click the  icon. This will remove the NILM from active management by the system but will not erase any data or settings on the NILM itself. You can add and remove the same NILM multiple times using this interface without any problems. The **Status** icon indicates whether the NILM is online (green) or not available (red).

User Tab

The user tab allows you to add, edit, and remove user accounts from the system. Note that any account you add you must manually confirm by clicking the button next to the user account. Usually the confirmation would be done by the user through an e-mail but this version of NILM Manager does not have an e-mail server. The **Engineer** attribute is not currently used. The **Admin** attribute controls whether the account has administrative privileges (i.e. the ability to use this interface). Note this is not related to any privileges an account has on a particular NILM.

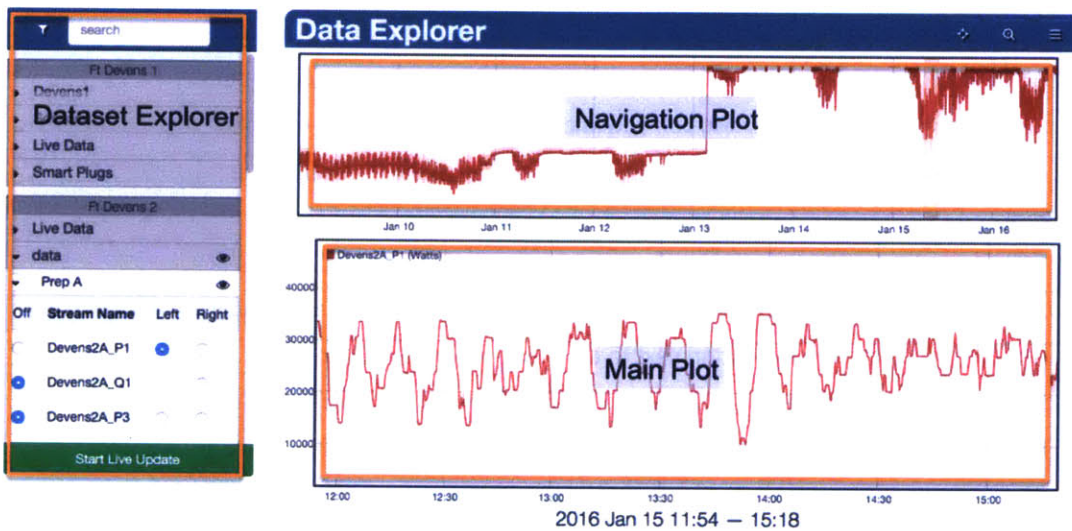
Data Views Tab

This tab displays all of the stored dataviews on the system. You can edit the name and description of the views by clicking the  icon and delete it by clicking the  icon.

NILM Explorer


Introduction


NILM Explorer is an interactive plotting tool that provides visualizations of remote datasets using very little network bandwidth. This interface seamlessly combines data from one or more remote NILM systems and is capable of plotting data at any time scale. The interface works best with a three button mouse (left, right and scroll wheel), although a touch pad can be used. The interface consists of three main panels, the *Dataset Explorer*, *Navigation Plot*, and *Main Plot*. The figure below labels each panel on the interface. The sections below describe these panels in detail.




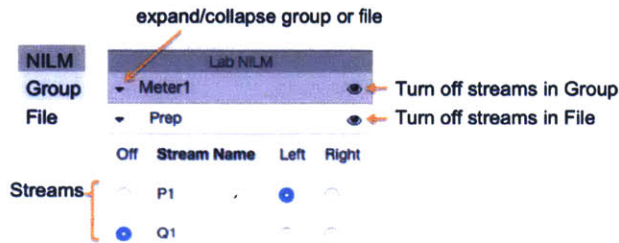
The three main panels of the NILM Explorer interface: *Dataset Explorer*, *Navigation Plot*, and *Main Plot*

Dataset Explorer

The dataset explorer shows all the plottable datasets for all the NILM's managed by the system. If this is a standalone system there will only be one NILM listed. On a cluster multiple NILM's are separated by dark grey title bars. Select which NILM's are displayed by using the  button next to the search bar.

The groups on each NILM are listed below the NILM title bar. Click the  button to expand a group and show its files. Each file can in turn be expanded to show the streams it contains. Streams have three radio buttons that control their visibility on the plot. By default all streams are *Off* meaning they are not displayed. Plot a stream by selecting either the *Left* or *Right* radio buttons:


Left and Right refer to left and right Y-axes of the plot. When at least one stream is selected for plotting both the Navigation and Main Plot panels will appear. Multiple streams may be plotted simultaneously from any file, group or even other NILM's. However only streams must share the same unit to be plotted on the same axis. Radio buttons are automatically disabled if the stream's unit does not match the current axis unit. When a stream is displayed on the plot an  is displayed next to the stream, file and group name. Clicking this icon on the stream will remove it from the plot. Clicking this on the file will remove any of the file's streams from the plot, and clicking it on the group will remove any stream from any the group's files from the plot. This is a quick way to clear a plot with many streams displayed.

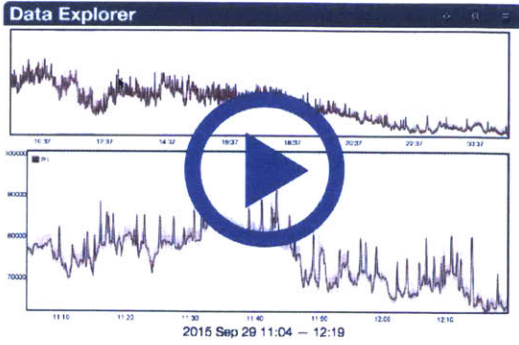


The Dataset Explorer interface

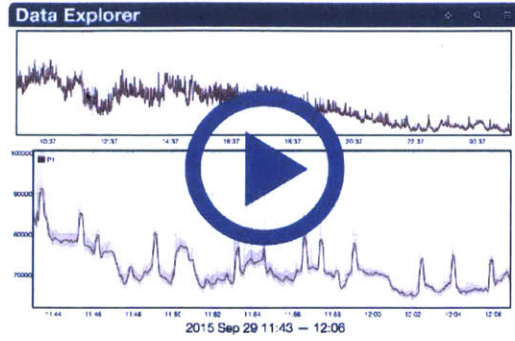
The first stream to be displayed sets the bounds of the plots. The x-axis is set to the full range of data in the stream and the y-axis is set to either the range of the data or to the streams `Default Min` and `Default Max` values if set (see [Stream Configuration](#) for this setting). If you want the plot to track the latest data select **Start Live Update**. This will lock the time axis of the Navigation Plot to the last hour and the Main Plot to the last twenty minutes. Click **Stop Live Update** to disable time axis tracking.

Navigation Plot


The Navigation Plot shows a fixed overview of the data and highlights the portion displayed in the Main Plot. The y-axis is fixed to the autoscale values of the data (either the range of the plotted data or the `Default Max` and `Default Min` of the streams). In the default navigation mode is clicking and dragging on the plot selects the subset of data displayed in the Main Plot window. This mode is animated in the left hand figure below. The mode can be changed by clicking the  and selecting **Lock Navplot Selection**. When this box is checked the time range of the selection is locked. Clicking and dragging the selection window changes the fixed time range displayed in the Main Plot. This mode is animated in the right hand figure below.



Navigation Plot in default mode. Click and drag to select a portion of data to display in the Main Plot.

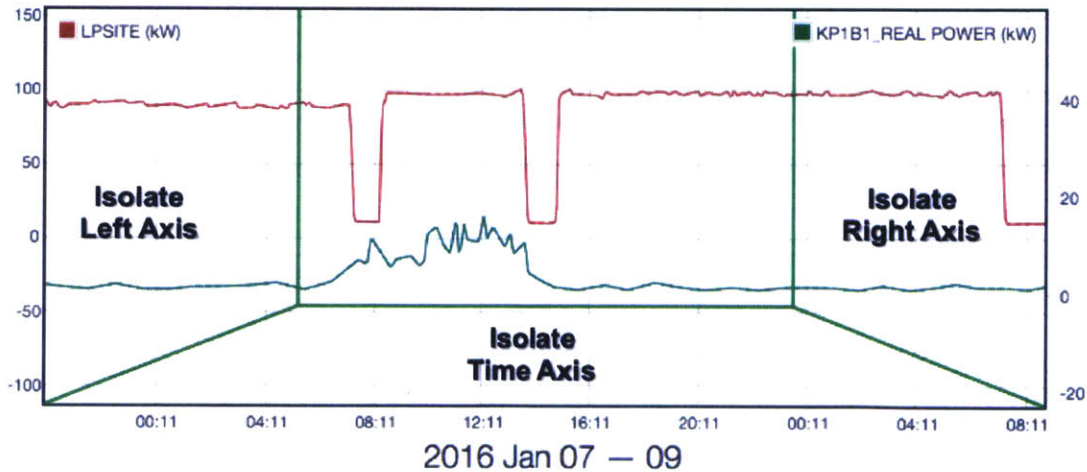


Navigation Plot with **Lock Navplot Selection** checked. Click and drag to move the selection window across the data.


To change the time range of data displayed in the Navigation Plot, zoom to the desired time range in the Main Plot, click the  button and select **Sync Navplot to View**. This will set the Navigation Plot time range to the Main Plot time range.

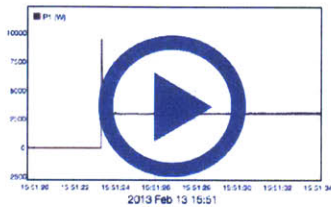
Main Plot

The Main Plot interface supports pan and zoom on three axes (left and right y axis and the x axis). Click and drag to pan, and scroll to zoom. The plot is divided into four regions as shown in the figure below. The zoom and pan controls operate differently depending on which region the cursor is in. When the cursor is in the center of the plot, pan and zoom operates simultaneously on all the axes.

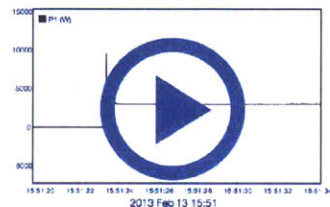


The Main Plot interface is divided into four zones. When the cursor is close to an axis the zoom and pan controls operate only on that axis.

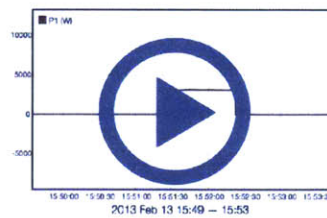
When the cursor is placed close to an axis the pan and zoom only operate on that axis. This is indicated visually by a yellow highlight on the isolated axis. The three animations below show demonstrations of (a) isolating the time axis, (b) isolating the y axis, and (c) operating all axes simultaneously. In general it is easier to navigate datasets with isolated axes rather than zooming or panning all axes together. If the cursor is moved on top of the axis a  icon appears on the axis. Clicking this icon will autoscale the axis. On the time axis this means the bounds will change to cover all of the data from all currently plotted streams. On the y axes this means the bounds will adjust to the maximum range of either the plotted samples or the Default Min/Default Max settings of the streams. **The cursor must remain in the plot grid to pan or zoom, moving the cursor outside the axes will disable the plot controls.**




A.) Isolated zoom on the time axis.



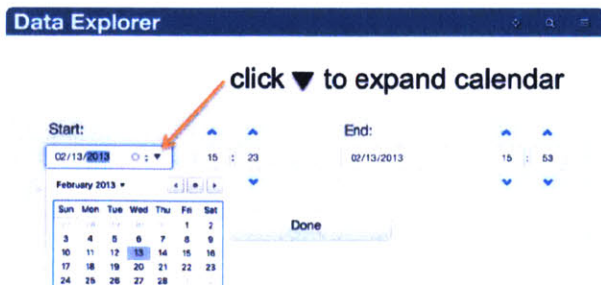
B.) Isolated zoom on the y axis.



C.) Simultaneous zoom on both axes.


The cursor will automatically highlight datapoints on the plot with a circle overlay. This can be helpful to determine the resolution of a particular plot. The  button in the plot header toggles the cursor tooltip. The tooltip displays the numeric value of the data as the cursor moves over a plot. This is useful for determining the exact value of a sample

The date label below the Main Plot automatically adjusts according to the range of data plotted. Click the date to bring up the time selection overlay shown below. This allows you to specify a particular time range of data. Click the date field to open up a calendar view or use the mouse wheel to scroll individual fields. The overlay is translucent so you can see the data adjusting as you specify the date. If the end time is before the start time the selection is invalid and no change will be made to the plot.



Click the date below the Main Plot to open the *Date Selector Overlay*

Open / Save / Download Data

The plot menu is accessed by clicking the  button in the header bar. This expands a dropdown menu with options to **Open**, **Save**, and **Download** plots. To open a previously saved plot, select **Open** and find the desired plot in the dialogue. The search feature filters plots on description and title. Click a plot figure to open it. To save a plot select **Save** and enter a title and description. The installation drop down box is for categorizing plots by location. On a standalone system there is only one installation, but on a cluster system there will be multiple installations listed. Note the saved plots are lightweight indexes to the underlying datasets. Depending on the **keep** settings of the datastreams plots using live datasets may be erased. It is recommended to archive sections of datasets you would like to keep to a separate stream that will not be erased by the cleanup routine.

The **Download** menu allows you to retrieve the raw data behind a plot as a comma separated value (CSV) file. Each file must be downloaded separately since the timeseries differ between files. The downloaded file has an informational header that describes the dataset. An example datafile is shown below

```
#####
#
# Source: Archive
#   Bucket archive
#
# group:   LEES' Compressor
# file:    prep
# database:
# url:     http://bucket.vpn.wattsworth.net/nilmdb
# path:    /no-leak/prep
#
# start:   2013-02-12 13:21:05 -0500
# end:     2013-02-12 13:21:10 -0500
# total time: less than a minute
# total rows: 281
#
#The raw data file can be retrieved at the following URL:
# http://bucket.vpn.wattsworth.net/nilmdb/stream/extract?path=...
#
# to import in matlab run:
#   nilm = importdata('thisfilename.txt')
#
# nilm.textdata: this help text
# nilm.data:     the data
#
# The data has 2 columns with the following format:
#
# Column 1: Timestamp (microseconds)
# Column 2: P1 (W)
#
#####
1360693265540800, 0.1496984
1360693265557470, 0.689765
1360693265574140, -0.2659167
1360693265590810, 0.4024591
.... data continues below ....
```

In many cases the plotted dataset is too large to download over an HTTP (web) connection. In this case the download file will provide instructions for using `nilmtool` commands on the terminal to retrieve the raw data. `nilmtool` retrieves all streams in a file and the data is not scaled. The file provides equations to scale the streams according the their scale and offset settings.

```
#####
#
# Source: Archive
#   Bucket archive
#
# group:      LEES' Compressor
# file:       prep
# database:
#   url:      http://bucket.vpn.wattsworth.net/nilmdb
#   path:     /no-leak/prep
#
# start:      2013-02-12 14:34:53 -0500
# end:        2013-02-13 23:29:51 -0500
# total time: 1 day
# total rows: 2969042 <==== OVER 2 MILLION ROWS
#
#The raw data file can be retrieved at the following URL:
# http://bucket.vpn.wattsworth.net/nilmdb/stream/extract?path=...
#
#####
# There is too much data to download. If you really need
# this data you can extract it directly using nilmtool
#
# The data has 9 columns with the following format:
#
# Column 1: Timestamp (microseconds)
# Column 2: P1 (W) [y=(x-0.0)*-1.0]
# Column 3: Q1 (prep) [y=(x-0.0)*1.0]
# Column 4: P3 (prep) [y=(x-0.0)*1.0]
# Column 5: Q3 (prep) [y=(x-0.0)*1.0]
# Column 6: P5 (prep) [y=(x-0.0)*1.0]
# Column 7: Q5 (prep) [y=(x-0.0)*1.0]
# Column 8: P7 (prep) [y=(x-0.0)*1.0]
# Column 9: Q7 (prep) [y=(x-0.0)*1.0]
#
# -- this file can be run directly as a script --
#
# -----
nilmtool --url http://bucket.vpn.wattsworth.net/nilmdb extract ...
```

The file can be run directly as a bash script which will dump the timestamped file data to standard output. Redirect this to a file to save the data. Note that raw data can be *very large*, be careful with your disk usage and the network bandwidth when retrieving datasets with `nilmtool`

```
1 #print data to the terminal
2 $ bash ~/Downloads/nilm_data.txt
3 1360697693233225 6.951723e-01 -2.935897e-01 9.620408e-01 ...
4 1360697693249891 1.101132e-02 4.109371e-01 -1.420663e-02 ...
5 1360697693266557 2.287447e-01 1.076953e-01 4.715213e-01 ...
6 1360697693283223 -5.571978e-01 -6.638092e-04 7.577136e-01 ...
7 .... data continues ...
8 #save data to text file
9 $ bash ~/Downloads/nilm_data.txt > saved_data.txt
```




NILM Filter


Introduction

Filters are the NILM's data processing engine. Unlike static datasets which have a known fixed size, NILM datasets are constantly growing and are generally too large to operate on as a single array. Filters are iterative code blocks that allow for efficient processing on these large datasets. The tutorials below introduce the basic concepts. See the labeled sections for more detailed information.

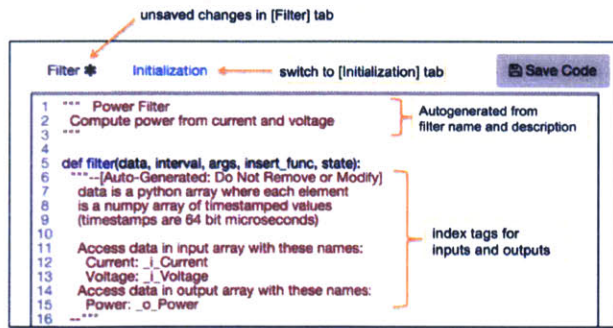
Tutorial 1: Hello World

This example is an introduction to the NILM Filter tool. In this example you will build a basic median filter to smooth a dataset. This example will show you how to create a new filter, set up bindings, and test the filter in the development environment. To get started, click the [NILM Filter](#) tile on the dashboard to load the Filter Listing page.

Filter Listing: Filters are listed by Title and Description. Filters can either be public or private. The Editing column shows the permission setting. Private filters can only be edited by their author while public filters can be edited by any user. The current permission is displayed as a private  or public  icon. Click the button to toggle the permission setting. To other users private filters are marked with a  icon. The private and public setting only applies to *editing*. Any filter can be installed on any NILM by a user with at least `owner` privileges on the NILM. Click the [+ Create a new filter](#) button to enter the New Filter Page.

New Filter Page: Enter a name and description for the filter. The name must be unique and the description should be a short sentence that describes what the filter is doing. Filters have one or more input data sources. Input sources are generic hooks that are bound to data streams at runtime. Enter "source" in the input text box and click [+ Add](#). The input will be added to the filter. If you mistyped the entry click the  next to the input and type the input name again. You can add more inputs the same way, but for now just use a single input. **Do not** click the "Resample streams" check box. Filters produce one or more output data streams. The output streams are stored in a single file that is dynamically created on the NILM at runtime. When run in the development environment this output file is stored temporarily by default (although it can be stored permanently). When a filter is run as a [NILM Process](#) the output file is stored permanently. For this filter add a single output named "filtered". Click [+ Add](#) to add the output stream to the filter. Click [Create Filter](#) to enter the Filter Development Environment.

Filter Development Environment: The development interface is divided into two main panels each with two tabs. The left panel is a code editor and the right panel provides options for configuring and testing the filter. Select the **Setup** tab on the right panel.



Filter Design Code Panel

Filter Development Environment (TODO: fix picture)

Filter Setup

Filters run on a particular NILM. In order to test the filter in the development environment you must pick an available NILM and associate the filter inputs with data streams on this NILM. Select an available NILM from the combo box. If this is a standalone installation there will only be one entry. If this is a cluster installation there may be multiple NILMs listed. After you select a NILM, bind the filter inputs to data streams that are available on the NILM. This filter only has one input which we will bind to a P1 stream. Start by selecting a data group. For this tutorial select a meter group that has available data. Then select the `Prep A` file and `P1` stream. Next we need to configure the output stream. The output will be a filtered copy of the input so the units should match the input stream. `P1` is in watts so enter "w" in the unit field. We want to plot the filter output so select the `plottable` checkbox. This will notify the filter engine that the output must be decimated. Click **Save Changes** to update the filter with these new settings. Whenever you adjust the settings in the `Setup` tab you must click this button to apply the changes. A green success message appears at the top of the tab once the save operation is complete. The `setup` tab also lets you add or remove inputs and outputs to the filter. This has the same effect as adding inputs and outputs on the *New Filter Page*. A filter must have at least one input and one output. Now you can begin writing code for the filter.

Code: Filter

NILM filters are written in Python 2.7. If you are new to Python, there are many books on the subject as well as several good websites. O'Reilly's "Learning Python" and "Programming Python" are great introductory texts. Websites come and go, at the time of this writing, [Code Academy](#) is a good resource to learn Python and many other languages. Filters rely on `numpy` and `scipy` for most signal processing routines. The [online documentation](#) for these tools is excellent. O'Reilly's "Python for Data Analysis" provides a good introduction to these tools and many Python data processing packages.

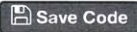

Select the **Filter** tab in the code editor. Lines 1–3 are auto generated from the filter name and description. Add the `import` statement in line 5 to your filter. This imports the `medfilt` function from `Numpy`. Lines 8–17 are auto generated from the `Setup` tab settings. These lines should match the auto

generated comments in your filter exactly. If the comments do not match check to make sure you have not selected "Resample Streams" and that your input is named source and output is named filtered. Correct any errors and save the changes. This comment block will automatically update. When the filter is first created the body of the filter function is empty. Copy lines 18–29 and paste it in place of the #TODO...pass lines to complete the function.



```
1 """  Median Filter
2  Windowed median filter
3 """
4
5 from scipy.signal import medfilt
6
7 def filter(data, interval, args, insert_func, state):
8  """--[Auto-Generated: Do Not Remove or Modify]
9  data is a python array where each element
10 is a numpy array of timestamped values
11 (timestamps are 64 bit microseconds)
12
13 Access data in input array with these names:
14     source: _i_source
15 Access data in output array with these names:
16     filtered: _o_filtered
17 --"""
18 #filter parameters
19 WINDOW_SIZE = 25
20 #shorthand to access timestamps and values
21 ts = data[_i_source][:,0]
22 vals = data[_i_source][:,1]
23 #run the median filter
24 result = medfilt(vals,WINDOW_SIZE)
25 #add timestamps to the result array by
26 #stacking it together with the ts array
27 output=np.hstack((ts[:,None], result[:,None]))
28 #insert the timestamped data into the output stream
29 insert_func(output)
```

How it works: NILM data streams can be very large (eg billions of samples). In order to process large data sets on a machine with limited resources we have to break the data into discrete chunks. The NILM Filter engine handles this data chunking transparently allowing developers to write code that assumes all the data is available as a continuous array. The framework defines two functions that must be specified by the developer: `filter`, and `initialize`. Within `filter` the data can be processed like a traditional array. The data processing engine calls `filter` iteratively with chunks of input data.

This example applies a windowed median filter. The median filter algorithm is imported from Numpy in line 5. Line 19 defines `WINDOW_SIZE` which is a parameter to `medfilt`. Explicitly defining tunable constants is a good way to make your code more readable and easier to maintain. Lines 21–22 set up local variables from the input data. The `data` parameter is a list of Numpy arrays from each input. This filter has only one input so a numerical index would be straight forward, but it is best practice to retrieve inputs from `data` using the logical index generated by the filter framework, `_i_source`. The next index operation is into the Numpy array. The Numpy array has two columns, timestamp (0) and value (1). The `[:, X]` notation is shorthand for "all rows of a specific column". This initializes `ts` to a 1D array of timestamps and `vals` to a 1D array of input values.

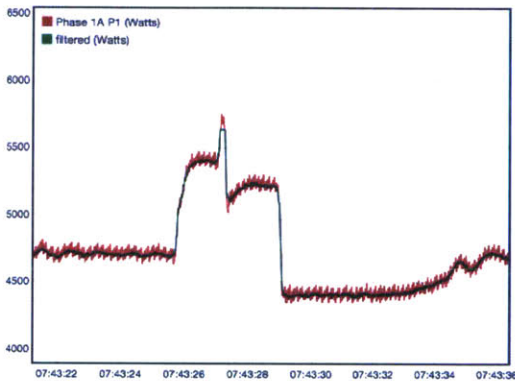
Line 24 applies the median filter. This is the same code used for processing "standard" continuous arrays of data. The filter engine makes it possible to port standard signal processing algorithms directly to a chunk-processing architecture. Line 27 applies timestamps to the median filtered data so it can be stored as an output stream. The median filter is time invariant so the output has the same timestamps as the input. Line 29 inserts the median filtered data into the output stream. `insert_func` takes one argument, an array of timestamped values to save. Output timestamps must be monotonically increasing. This means each element of the array must have a unique timestamp that is greater than the timestamps before it. In this case the timestamps are copied from the input stream so we are guaranteed to meet this requirement. Click  when you have finished entering the code. Whenever you have unsaved changes to the code an  will appear next to the tab with changes. *Important: You must save changes before running the filter or leaving the page, otherwise any changes will be lost.*

Testing the Filter

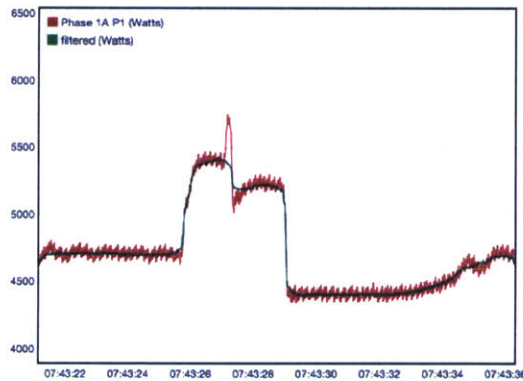
Once you have finished writing the processing code, its time to test the filter on a sample dataset. Select the **Test** tab from the right panel. The tab is divided into two sections. The top section has filter controls and displays the console output. The bottom section is a truncated [NILM Explorer](#) interface. Filter inputs and outputs are listed below the plot window. The inputs are listed by name with the associated stream in brackets. Plot the `source[P1]` stream on the left axis. Pan and zoom the plot to a short (less than a minute) section of data. The filter will run against the plotted time range only so selecting a smaller amount of data will make the filter run faster. Once you have selected a suitable range click the  button to lock the plot. Click  to start the filter. The console output will display

information as the filter runs including the output of any `print` statements. If an error occurs the console will display the python debug dump. Fix any errors and run the filter again. When the filter finishes successfully the console status will display `Complete!`

When the filter is finished the outputs will be plottable. Plot the `filtered` output on the same axis as the input. If the axis is disabled the units do not match with the input. Go back to the **Setup** tab and set the output stream units to match your input (w). Depending on how much data your filter processed, the output may require decimation. This happens automatically but takes some time. If the decimation isn't complete you will see an `*` next to the stream in the plot legend. Pan or zoom the plot to refresh the view until the decimation process is complete. The figures below show output from this filter with different `WINDOW_LENGTH` settings. The development environment makes it easy to quickly iterate and tune your filters. When you are satisfied with its performance you can set the filter up as a [NILM Process](#) to continuously on input data.



Median filter example using a window size of 15



Median filter example using a window size of 55

Tutorial 2: Calculating Power

This filter calculates instantaneous power from current and voltage inputs. Since this filter has multiple inputs we will resample the input array to a single time series which makes the data easier to process with standard signal processing techniques.

Resampled inputs: When a filter has multiple inputs you have the option of resampling them to a common time series. The filter engine can automatically interpolate the inputs and create a composite input array. The resampled input is a two dimensional numpy array where the first column is the time stamp and each subsequent column corresponds to a filter input. See [Filter Inputs](#) for more information. The filter engine requires a master stream to establish the time series. This is usually the highest bandwidth stream, although some applications may require a different choice.

Filter Setup

Create a new filter with two inputs labeled `current` and `voltage` and select the "Resample streams" check box. Add one output labeled `power`. From the filter setup panel, attach input streams to the bindings from a raw NILM file and make the output plottable in units of watts (w) and save your changes. Select the `current` input as the master stream. *A note on performance:* resampling adds overhead which can slow down filters with highbandwidth inputs. In this example a faster implementation would be to use a single bulk input on the raw NILM file. See the [Filter Inputs](#) section for information on bulk inputs.

Code: Filter

Select the **Filter** tab in the code editor. Lines 1–3 are auto generated from the filter name and description. Lines 6–18 are auto generated from the Setup tab settings. These lines should match the auto generated comments in your filter exactly. If the comments do not match check to make sure you have selected "Resample Streams" and that your inputs are named `current` and `voltage`. Correct any errors and save the changes. This comment block will automatically update. When the filter is first created the body of the filter function is empty. Copy lines 16–19 and paste it in place of the `#TODO...pass` lines to complete the function.

```
1 """    Power Filter
2 single phase power
3 """
4
5 def filter(data, interval, args, insert_func, state):
6     """--[Auto-Generated: Do Not Remove or Modify]
7     data is a 2D numpy array, each row is a sample
8     column[0]: 64 bit timestamp (us)
9
10    Access data in input array with these names:
11     current: _i_current
12     voltage: _i_voltage
13    Access data in output array with these names:
14     power: _o_power
15    --"""
16    power = data[:,_i_current]*data[:,_i_voltage]
17    time = data[:,0]
18    output = np.hstack((time[:,None],power[:,None]))
19    insert_func(output)
```

How it works: The `data` parameter is a 2D Numpy array. Column 0 is the timestamp and the subsequent columns are the input stream values. The resampling engine linearly interpolates the inputs to a single timeseries so the inputs appear to be simultaneously sampled. This makes the power calculation very straightforward. Line 16 computes the power directly by multiplying the `current` and `voltage` columns of the array. Using logical indices for the `data` array makes the code self commenting and easier to maintain if you later want to add or rearrange the inputs.

The `output` array is created by stacking the input timestamps with the power array in line 18. This is inserted into the output stream in line 19.

After you have entered this code, save the changes and run the filter against a short section of input data. Notice the ripple at twice the line frequency.

Testing: Changing to 3 Phase

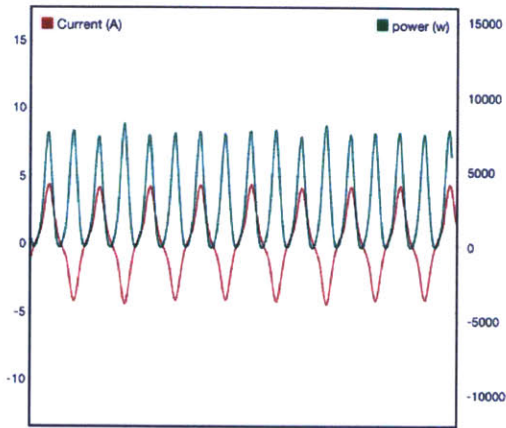
Many industrial systems use three phase power to avoid this ripple. This filter can be easily adapted to measure three phase power. Delete the filter inputs and add the following six inputs `IA`, `IB`, `IC`, `VA`, `VB`, `VC`. Bind these inputs to their respective streams in the raw file and save the changes. Notice the comment section in the `filter` function updates with the new configuration. Copy the new filter code below:

```

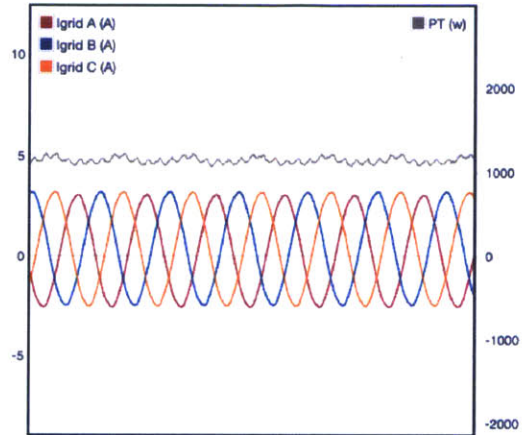
1 """    Power Filter
2 calculate power from current and voltage
3 """
4
5 def filter(data, interval, args, insert_func, state):
6     """--[Auto-Generated: Do Not Remove or Modify]
7         data is a 2D numpy array, each row is a sample
8         column[0]: 64 bit timestamp (us)
9
10        Access data in input array with these names:
11            I1: _i_IA
12            V1: _i_IB
13            I2: _i_IC
14            V2: _i_VA
15            I3: _i_IB
16            V3: _i_VC
17        Access data in output array with these names:
18            PT: _o_PT
19    --"""
20    pA = data[:,_i_I1]*data[:,_i_V1]
21    pB = data[:,_i_I2]*data[:,_i_V2]
22    pC = data[:,_i_I3]*data[:,_i_V3]
23    pT = p1+p2+p3
24    time = data[:,0]
25    output = np.hstack((time[:,None],pT[:,None]))
26    insert_func(output)

```

The output from this filter is shown below. Notice that the power is relatively constant. Try adjusting this filter to have multiple outputs for PA, PB, PC. These outputs will look similar to the single phase power plot.



Instantaneous power for a single phase system



Instantaneous power for a three phase system

So far we have only seen filters compute instantaneous metrics, many times we are interested in cumulative metrics like energy consumption or average power. Computing these types of metrics requires a new type of filter tool called *state*. See the next tutorial for information on how to build stateful filters.

Tutorial 3: Energy Consumption

This filter computes cumulative energy consumption from a prep input. This requires maintaining a persistent running sum. The filter engine maintains persistent variables through the *state* object. This object is initialized prior to the first filter run and then passed in as an argument to the *filter* function.

Filter State: State provides a type of "memory" between runs of the *filter* function. The state is passed into *filter* as a parameter. The *state* object stores persistent data in slots. Slots must be initialized and before they can be used. Once a slot is initialized any data type can be stored into it and retrieved later. Slots can be used many ways but the recommended design pattern is as follows:

State Workflow:

1. Create state slots for each persistent variable in the *initialize* function
2. At the start of the *filter* function, retrieve state objects into local variables
3. Manipulate the local copies of the state objects
4. At the end of the *filter* function, return local copies back to the state

Step 4 is not necessary for mutable objects because Python passes them by reference. Immutable objects like lists, and numbers must be returned explicitly since they are passed by value. When in doubt, return local variables to the state.

Filter Setup

Create a new filter with one input labeled `prep`. **Do not** select the "Resample streams" check box. Add one output labeled `energy`. From the filter setup panel, attach a P1 prep stream to the input binding. Make the output plottable in units of kilowatt hours (kWh) and save your changes.

Code: Initialize

Select the **Initialize** tab in the code editor. Lines 1–3 are auto generated from the filter name and description. When the filter is first created the body of the `initialize` function is empty. Copy lines 6–8 and paste it in place of the `#TODO...pass` lines to complete the function.

```
1 """ Initialization for Energy Integrator
2 Tutorial filter for energy
3 """
4
5 def initialize(state):
6     #1.) ---initialize state slots---
7     state.initializeSlot("C",0)
8     state.initializeSlot("last_ts",None)
9     state.initializeSlot("last_y",None)
```

How it works: This function runs *before* the data chunks are passed through `filter`. This is step 1 of the State Workflow. Slots in `state` are created by called `initializeSlot`. This function takes two parameters, a unique string identifier and the initial value for the slot. This filter uses three state slots: `C` is the integration offset. It is initialized to 0 which means the output energy stream will start at 0. The next two slots, `last_ts` and `last_y` are the last timestamp and value of the previous data chunk. These two slots are initialized with `None` to indicate they don't have a valid initial value. The next section describes how these state variables are used.

Code: Filter

Select the **Filter** tab in the code editor. Lines 1–3 are auto generated from the filter name and description. Add the `import` statement in line 5 to your filter. This imports the `cumtrapz` integration function from Numpy. Lines 8–17 are auto generated from the Setup tab settings. These lines should match the auto generated comments in your filter exactly. If the comments do not match check to make sure you have not selected "Resample Streams" and that your input is named `source` and output is named `filtered`. Correct any errors and save the changes. This comment block will automatically update. When the filter is first created the body of the filter function is empty. Copy lines 18–49 and paste it in place of the `#TODO...pass` lines to complete the function.

```
1 """ Energy Integrator
2 Tutorial filter for energy
3 """
4
5 import numpy as np
6
7 #1.) ---initialize state slots---
8 state.initializeSlot("C",0)
9 state.initializeSlot("last_ts",None)
10 state.initializeSlot("last_y",None)
11
12 #2.) ---filter function---
13
14 #3.) ---update state slots---
15
16 #4.) ---return filtered data---
17
18 #TODO...pass
```



```

4
5 from scipy.integrate import cumtrapz
6
7 def filter(data, interval, args, insert_func, state):
8     """--[Auto-Generated: Do Not Remove or Modify]
9     data is a python array where each element
10    is a numpy array of timestamped values
11    (timestamps are 64 bit microseconds)
12
13    Access data in input array with these names:
14        prep: _i_prep
15    Access data in output array with these names:
16        energy: _o_energy
17    --"""
18    #2.) ---retrieve state into local variables---
19    last_ts = state.retrieveSlot("last_ts")
20    last_y = state.retrieveSlot("last_y")
21    C = state.retrieveSlot("C")
22    #shorthand to access timestamps and values
23    ts = data[_i_prep][:,0]
24    y = data[_i_prep][:,1]
25
26    #3.) ---run data processing---
27    #initialize last states on first pass
28    if(last_ts is None):
29        last_ts = ts[0]
30    if(last_y is None):
31        last_y = y[0]
32    #append last values to current arrays
33    ts = np.insert(ts,0,last_ts)
34    y = np.insert(y,0,last_y)
35    #integrate 'y' with respect to 'ts'
36    # scale output to kWh
37    kWh = cumtrapz(y,ts/(1e6*60*60*1e3))
38    #the integration is offset by the previous
39    # run of the filter function
40    kWh += C

```

```

41
42 #4.) ---return local variables to the state---
43 state.updateSlot("C",kwh[-1])
44 state.updateSlot("last_ts",ts[-1])
45 state.updateSlot("last_y",y[-1])
46
47 result = np.hstack((ts[1:,None],kwh[:,None]))
48
49 insert_func(result)

```

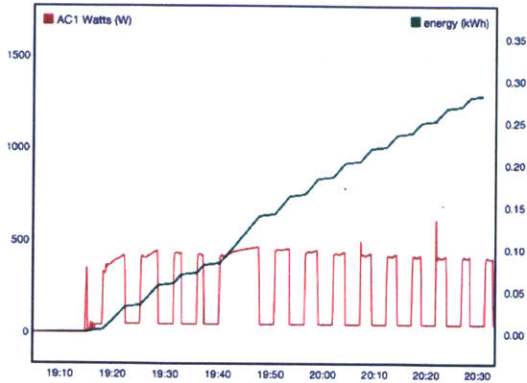
How it works: This example is more complex than the previous two, but it consists of several simple steps. The first section of this code (lines 19–22) retrieves the state slots into local variables which is step 2 of the State Workflow. Next, lines 24–25 provide shorthand variables into the data array so we can easily access the timestamps (ts) and values (y).

Lines 27–41 perform the data processing which is step 3 of the State Workflow. Lines 28–32 initialize last_ts and last_y if this is the first chunk of data. The reason for these variables is somewhat subtle. The output of a numerical integration is one element shorter than the input. This might seem insignificant, but the filter function executes iteratively and losing a value with each iteration will cause the output (energy) to lag the input (power). To keep the streams in sync, the input timestamps and power are padded with the last value of the previous input arrays. This is done in lines 34–35 with the call to np.insert. Lines 38–41 calculate the energy integral. The cumtrapz is imported from the Numpy integration library in line 5. This function performs efficient trapezoidal integration. The integral is offset by C which is the value of the energy integral from the last data chunk. Without this addition the integral would "reset" to 0 with every data chunk.

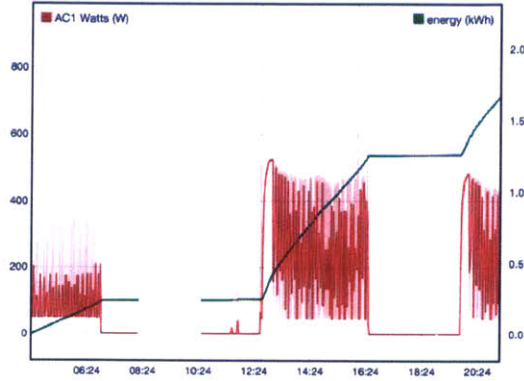
Lines 45–47 complete step 4 of the State Workflow. The slots are updated from the local variable values with calls to updateSlot. Finally, the energy array is timestamped and passed to insert_func for insertion into the output stream.

Testing

The figures below show the filter output over two different time intervals. Gaps in the input stream propagate to the output stream. If a filter has multiple input streams the output is only computed over the intersection of available input data. The state is maintained across gaps in the input data so the filter code does not need to explicitly check for areas of missing data. If you do want to check for breaks in the data use the args["isEnd"] flag, see [args documentation](#) for details.



Calculating energy from power using a stateful filter.



Gaps in the input data do not affect filter state.

Reference

Inputs

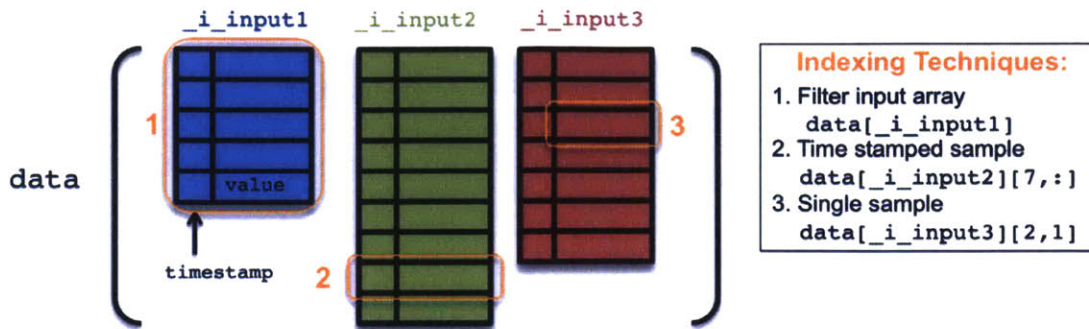
Filters have one or more inputs. For example, a filter that computes power would have two inputs—voltage and current. Inputs are hooks that are associated with data sets on the NILM at runtime. Hooks allow the same filter to be used multiple times with different datasets. This makes the code more efficient and reusable. For example instead of designing a three phase median filter, an instance of the filter from Tutorial 1 could be scheduled for each phase. In design mode, the filter inputs are bound to datasets in the **Setup** tab. First select a NILM to run the filter from the dropdown box at the top of the tab (default is Primary). Specify the binding by selecting the Group, File, and Stream for each input using the dropdown boxes. An input is fully specified when all three boxes have a valid selection. The filter will not run if any of its inputs are not fully specified. Filter inputs can be removed by clicking the trashcan icon to the right of the combo boxes and new inputs can be added by typing a name in the text box below the current inputs and clicking the **+** icon. All inputs must have unique names. The figure below shows an example of the filter input binding interface:

	Group	File	Stream	
Input1	Live Data	Machinery	heaters	✓ Fully specified
Input2	mysql	Tent 3 ECU		✗ Missing Stream
Input3	Live Data		-	✗ Missing File and Stream

Specifying filter inputs. All inputs must be specified before a filter can run.

Inputs can be presented to the filter either as a list individual Numpy arrays or a single resampled Numpy array. Arrays are 2D structures where column 0 is a microsecond timestamp and columns 1–N are the data. The filter receives data through the `data` argument. If resampling is *not* enabled (default), this argument is a Python list of Numpy arrays as shown in the figure below. This example shows a three input

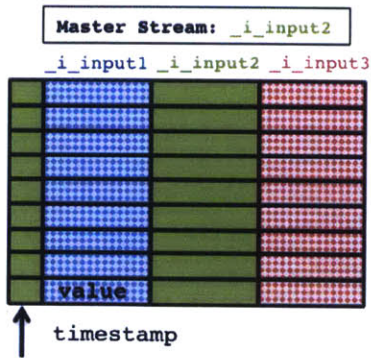
filter. The input arrays are different lengths because NILM streams do not necessarily have the same sampling rate. Over a one minute time range for example, `prep` data will have 3600 samples while `sensor` data will have up to 480K samples (when using a contact NILM).



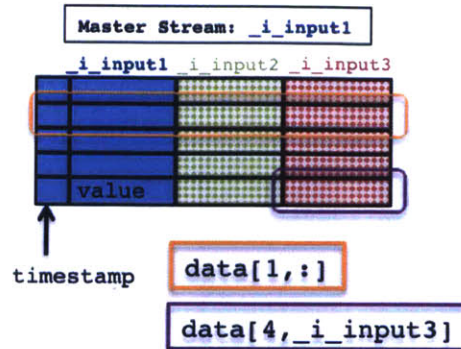
The `data` argument for a three input filter *without* resampling. Example indexing schemes for accessing different parts of the data.

To facilitate indexing into the `data` array the filter engine provides index tags for each input. These tags are inserted into the comments field below the `filter` declaration. They are formatted as `_i_inputname`. Using index tags instead of integers make the code easier to read and less brittle if inputs are added or removed later. Some common indexing techniques are shown in the figure above.

For many algorithms it is more convenient to have synchronously sampled inputs. The filter `resample` feature provides exactly this capability. In this mode, a single master stream (selected by the user) is used as the base timeseries and the other filter inputs are interpolated to fit these timestamps. In general the highest bandwidth stream should be selected as the master to reduce the effects of aliasing. The figure below has the same three inputs as before but uses resampling to collapse the inputs into a single array. In the first case, input 2 is the master so inputs 1 and 3 have additional interpolated samples to match this higher bandwidth time series. In the second case, input 1 is the master so inputs 2 and 3 are decimated (and interpolated) to match the lower bandwidth time series. Index tags are provided and should be used to index into the *column* of each sample. The righthand figure below shows some common indexing examples.

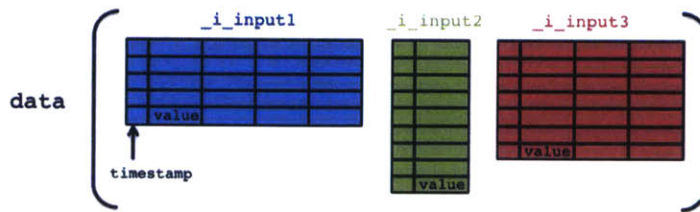


The `data` argument for a resampled three input filter. Using `_i_input2` as the master stream. Other inputs are interpolated.



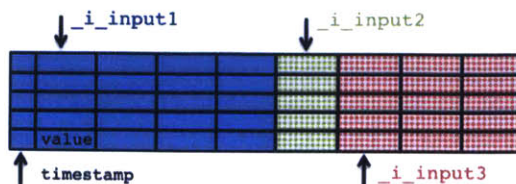
The `data` argument for a resampled three input filter. Using `_i_input1` as the master stream. Other inputs are decimated.

Resampling is useful for combining inputs from different files, but if the inputs all come from the same file resampling is not necessary since the datasets already share a common timeseries. When you are using more than one stream from a file it can be faster and more convenient to use a **bulk input**. Specify a bulk input by selecting `--all--` from the stream dropdown box (the first entry). Bulk inputs provide every stream from the file. Without resampling the input data will look similar to the figure below. As with normal inputs, use the logical indexing variables to locate each input array in `data`.



The `data` argument for a three input filter with two bulk inputs (input1 and input3).

Like normal inputs, **bulk_inputs** can be resampled. However care must be taken when indexing into this array because the logical index variables are pointers to the base of the bulk input block. See the figure below for an example.



The `data` argument for a three input filter with

two bulk inputs (`input1` and `input3`)
resampled using `input1` as the master.

Outputs

Filters produce one or more outputs. These outputs are stored in a single file that is dynamically managed by the NILM. When a filter runs in the development environment, the NILM creates a fresh output file each time the filter runs. When you are satisfied with the filter's performance it can be scheduled to run as a process. Process filters are assigned a persistent output file which is then available for plotting in the [explorer] view and can be used as inputs to other filters or analyzers.

The filter output setup is below the filter input controls. Each filter output has optional plot settings. Selecting the [discrete] option plots the output as sticks rather than a continuous line. The output must be plottable to be shown (some outputs might not make sense to plot so they can be hidden from view by unchecking this option). The units, scale factor, and offset options are optional and can be adjusted later using the standard NILM administration interface. Like filter inputs, outputs can be removed by clicking the trashcan icon and new outputs can be added using the controls below the current outputs.


After you make any changes to the filter setup you must click [Save Changes] to put them into effect




NILM Analyzers


Introduction




Analyzers produce reports from NILM data. Before working with Analyzers make sure you fully understand [NILM Filters](#). Analyzers iterate over input data exactly the same as filters but have an additional post processing step to generate an HTML report document. Reports can contain both text and graphics. Analyzers can be installed as a [NILM processes](#) to automatically generate reports as new data arrives. Reports generated by processes are accessible from the Report Groups dashboard tile.

Tutorial 1: Hello World

Analyzers use NILM data to produce reports. This example reports the average of a dataset. To get started, click the [NILM Analyzer](#)  tile on the dashboard to load the NILM Analyzer Listing.

NILM Analyzer Listing: Analyzers are listed by `Title` and `Description`. Analyzers can either be public or private. The `Editing` column shows the permission setting. Private analyzers can only be edited by their author while public analyzers can be edited by any user. The current permission is displayed as a private  or public  icon. Click the button to toggle the permission setting. To other users private analyzers are marked with a  icon. The private and public setting only applies to *editing*. Any analyzer can be installed on any NILM by a user with at least `owner` privileges on the NILM. Click the

 button to enter the New Analyzer Page.

New Analyzer Page: Enter a name and description for the Analyzer. The name must be unique and the description should be a short sentence that describes what the analyzer is doing. Analyzers have one or more input data sources. Input sources are generic hooks that are bound to data streams at runtime. Enter "source" in the input text box and click . The input will be added to the analyzer. If you mistyped the entry click the  next to the input and type the input name again. You can add more inputs the same way, but for now just use a single input. **Do not** click the "Resample streams" check box. Click  to enter the NILM Analyzer Page.

NILM Analyzer Page: The NILM Analyzer interface is divided into two main panels each with three tabs. The left panel is a code editor and the right panel provides options for configuring and testing the analyzer. Select the 1 `Setup` tab on the right panel.

Analyzer Setup

Analyzers run on a particular NILM. In order to test the analyzer during development you must pick an available NILM and associate the analyzer inputs with data streams on this NILM. Select an available NILM from the combo box. If this is a standalone installation there will only be one entry. If this is a cluster installation there may be multiple NILMs listed. After you select a NILM you must bind the analyzer inputs to data streams that are available on the NILM. This analyzer only has one input which we will bind to a

P1 stream. Start by selecting a data group. For this tutorial select a meter group that has available data. Then select the `Prep A` file and `P1` stream. Click [Save Changes](#) to update the analyzer with these new settings. Whenever you adjust the settings in the `Setup` tab you must click this button to apply the changes. A green success message appears at the top of the tab once the save operation is complete. The `setup` tab also lets you add or remove inputs to the analyzer. This has the same effect as adding inputs on the *New Analyzer Page*. An analyzer must have at least one input. Now you can begin writing code for the analyzer.

Code: `Iteration`

Select the `Iteration` tab in the code editor. Lines 1–3 are auto generated from the Analyzer *name* and *description*. Lines 6–12 are auto generated from the `Setup` tab settings. These lines should match the auto generated comments in your analyzer *exactly*. If the comments do not match check to make sure you have not selected "Resample Streams" and that your input is named `source`. Correct any errors and save the changes. This comment block will automatically update. When the analyzer is first created the body of the `iteration` function is empty. Copy lines 13–28 and paste it in place of the `#TODO...pass` lines to complete the function.

```

1 """ Average
2 Compute the average of a dataset
3 """
4
5 def iteration(data, interval, args, state):
6     """--[Auto-Generated: Do Not Remove or Modify]
7     data is a python array where each element is a numpy array
8     of timestamped values (timestamps are 64 bit microseconds)
9
10    Access data in input array with these names:
11        source: _i_source
12    --"""
13    #shorthand to access values in _i_source
14    vals = data[_i_source][:,1]
15    #local variables initialized from the state
16    avg = state.retrieveSlot("avg")
17    count = state.retrieveSlot("count")
18    #compute the local average
19    local_avg = np.mean(vals)
20    if(avg==None):
21        avg = local_avg
22    else:
23        #... or compute the weighted average
24        avg = avg*count/(count+len(vals)) + \
25            local_avg*len(vals)/(count+len(vals))
26    #update the state from the local variables
27    state.updateSlot("count",count+len(vals))
28    state.updateSlot("avg",avg)

```

How it works: The analyzer iteration function is very similar to the filter function in NILM Filters. You should already be comfortable with NILM Filters. In line 14 we extract the values from the input stream into `vals`. The data parameter is a list of Numpy arrays from each input. This analyzer has only one input so a numerical index would be straight forward, but it is best practice to retrieve inputs from the data list using the logical index generated by the analyzer framework, `_i_source`. The next index operation is into the Numpy array. The Numpy array has two columns, timestamp (0) and value (1). We are only interested in the values so we strip out the second column and store it in the local `vals` variable.

In lines 16–17 we retrieve the analyzer state into local variables. This analyzer has two state slots, `avg` and `count`. `avg` stores the average and `count` stores the number of samples in the average. Lines 18–25 perform the averaging computation. The Numpy built-in `np.mean(x)` is used to compute the average of the `vals` array. If this is the first data chunk through the analyzer the `avg` state variable will be `None` and we simply store the `local_avg` as the `avg` (line 21). If this is not the first data chunk, `avg` will be the average of the previous samples and we have to take the weighted combination of the previous average with the average of the new data. This is done in lines 24–25.

In lines 27–28 the state slots are updated from the local variables. `count` is incremented by the length of the `vals` array and `avg` is set to the new average.

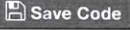
Code: Support

Select the `Support` tab in the code editor. Lines 1–3 are auto generated from the Analyzer *name* and *description*. This tab defines two functions: `initialize` and `analyze`. The `initialize` function runs *before* the data is processed by `iteration`. Like NILM Filters, this function is used to set up state slots. This analyzer has two slots, `avg` and `count`. The value for `avg` is unknown until the data has been processed so this slot is initialized to `None`. The `count` slot is used to store the total number of samples in the average and is initialized to 0.

The `analyze` function runs *after* the data is processed by `iteration`. This function can use the values stored in `state` to compute statistics on the data and produce plots with the `matplotlib` library. For this tutorial we simply print the average and number of samples processed.

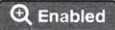

```
1 """ Initialization for Average
2 Compute the average of a dataset
3 """
4
5 def initialize(state):
6     state.initializeSlot("count",0)
7     state.initializeSlot("avg",None)
8
9 def analyze(state, saveFigure, args):
10    #retrieve state into local variables
11    avg = state.retrieveSlot("avg")
12    count = state.retrieveSlot("count")
13    #display result in the terminal
14    print "Processed %d values"%count
15    print "The average is: %f"%avg
```

Code: Report

Select the **Report** tab in the code editor. This is the report template. Reports are written using Markdown which is a simplified version of HTML. See [markdown syntax](#) for details. Lines 1–3 are autogenerated from the analyzer name and description. Lines 5–6 display our result. `state` values are injected into the report with double braces. Any `state` variable with a string representation can be injected into a report. Click  when you have finished entering the code for all three tabs.

```
1 Average
2 -----
3 Compute the average of a dataset
4
5 Processed {{count}} values
6 The average is {{avg}}
```

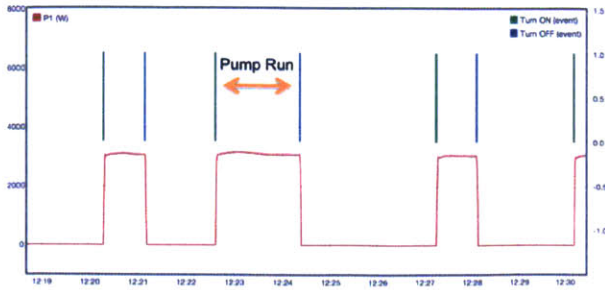
Testing the Analyzer

Once you have saved the code it is time to test the analyzer on a sample dataset. Select the **Test** tab from the right panel. The test tab is divided into two sections. The top section has analyzer controls and displays the console output. The bottom section is a truncated NILM explorer interface. Analyzer inputs are listed below the plot window. Plot the `source` stream on the left axis. Pan and zoom the plot to a short (less than a minute) section of data. Once you have selected a suitable range click the  button to lock the plot. Click  to start the analyzer. The console output will display information as the analyzer runs including the output of any `print` statements. If an error occurs the console will display the python debug dump. Fix any errors and run the analyzer again. When the analyzer finishes successfully the console status will update to `Complete!`

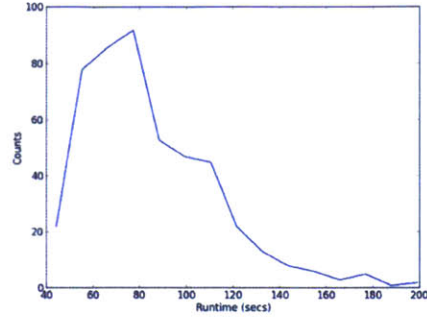
After the analyzer has run successfully select the **Output** tab on the right. This tab displays the analyzer report. You should see the number of values processed and the overall average. Try changing the report template or the time range in the plot and run the analyzer again. Remember to click the save code button to commit any changes.

Tutorial 2: Pump Health

This analyzer example generates diagnostics for an air compressor. Air compressors maintain the pressure in a tank. During normal operation the compressor runs intermittently. If there is a leak in the system the frequency of runs will increase. If the compressor itself has a fault the duration of the runs will increase since the compressor is less efficient. By plotting a histogram of runtimes we can detect these types of error conditions before they become a critical failure. The input to this analyzer is a stream of pump turn on and off events identified by [NILM Finder](#).



Source data for Tutorial 2. The pump transients are identified by NILM Finder.



The runtime histogram indicates pump health.

Analyzer Setup

This analyzer requires two inputs but both of the inputs are part of the same file so we can use a single bulk input binding. Create a new Analyzer and add a single input, `Events`. **Do not** click the "Resample streams" check box. From the **Setup** tab bind the input to the `--all--` stream on the events file generated by NILM Finder, and save the changes.

Events	Group	File	Stream
Pump	:	Events	--all--

Use `--all--` to create a **bulk input** from all streams in the file.

Code: Iteration

Select the `Iteration` tab in the code editor. Lines 1-3 are auto generated from the Analyzer *name* and *description*. Lines 6-12 are auto generated from the `Setup` tab settings. These lines should match the auto generated comments in your analyzer *exactly*. If the comments do not match check to make sure you have not selected "Resample Streams" and that your input is named `events`. Correct any errors and save the changes. This comment block will automatically update. When the analyzer is first created the body of the `iteration` function is empty. Copy lines 13-38 and paste it in place of the `#TODO...pass` lines to complete the function.

```

1 """  Cycling Systems Analysis
2  Create histograms of runtime
3 """
4
5 def iteration(data, interval, args, state):
6  """--[Auto-Generated: Do Not Remove or Modify]
7  data is a python array where each element is a numpy array
8  of timestamped values (timestamps are 64 bit microseconds)

```



```

9
10     Access data in input array with these names:
11         Events: _i_Events
12     --""
13     #indices into bulk input _i_events
14     ON = 1; OFF = 2
15     #local variables initialized from the state
16     turn_on_time = state.retrieveSlot("turn_on_time")
17     is_on = state.retrieveSlot("is_on")
18     runtimes = state.retrieveSlot("runtimes")
19     #compute runtimes
20     for d in data[_i_Events]:
21         if(d[OFF]):
22             if(is_on):
23                 #machine turned off, add a runtime statistic
24                 runtimes.append((d[0]-turn_on_time)/1e6)
25                 is_on = False
26             else:
27                 #machine turned off twice??
28                 print "double turn OFF at ",timestamp_to_human(d[0])
29         if(d[ON]):
30             if(is_on):
31                 #machine turned on twice??
32                 print "double turn ON at ",timestamp_to_human(d[0])
33                 turn_on_time = d[0]
34                 is_on = True
35     if(args["isEnd"] and is_on):
36         #machine must be off at the end of an interval
37         is_on = False
38         print "forcing off at end of interval"
39     #update the state from the local variables
40     state.updateSlot("turn_on_time",turn_on_time)
41     state.updateSlot("is_on",is_on)
42     state.updateSlot("runtimes",runtimes)

```

How it works: Line 14 sets up logical indices for the bulk input events. In lines 16–18 we retrieve the analyzer state into local variables. This analyzer uses three state slots: `turn_on_time` is the timestamp from the last ON event, `is_on` is the pump state (`true/false`), and `runtimes` is an array of the pump runtimes in seconds. Lines 20–34 loop over the input to calculate the pump runtimes. A runtime starts with a turn on event and ends with a turn off event. Each element in the input array is an event (ON or OFF). If the event is an OFF event (line 21) and `is_on` is true (line 22) this is the end of a runtime. The runtime is the difference between `turn_on_time` and the time of the OFF event. This is converted to seconds and appended to the `runtimes` array (line 24). Line 25 sets `is_on` to `False` to indicate the pump is off.

If the event is an ON event (line 29) `turn_on_time` is set to the event time (line 33) and `is_on` is set to `true` (line 34). Two additional clauses catch spurious transients. If the pump is off according to `is_on`, and another OFF event occurs, a warning is printed to the console and the event is ignored (line 28). Similarly, if the pump is on according to `is_on` and another ON event occurs a warning is printed (line 32) and `turn_on_time` is set to the more recent ON event (because the execution falls through to line 33).

Line 35 forces `is_on` to `false` at the end of an interval. This check is necessary because the data stream is intermittent. Ideally we would have continuous data but in reality sensors fail periodically so we have some gaps in the event stream. If the pump is on when the sensor stops recording we will have an ON event with no matching OFF. Without this check the runtime will extend until the sensor is back online and has recorded another OFF transient. Finally lines 40–42 update the state slots from the local variables.

Code: Support

Select the `Support` tab in the code editor. Lines 1–3 are auto generated from the Analyzer *name* and *description*. The `initialize` function defines the three state slots. `runtimes` is initialized to an empty array, `is_on` is initialized to `false` because we assume the pump is off before the event stream starts. If it is actually on we ignore the first OFF transient (line 27 in `iterate`) and start tracking runtimes with the first ON transient. The `turn_on_time` is left blank since we can't provide a valid initial value.

```

1 """ Initialization for Cycling Systems Analysis
2 Create histograms of runtime
3 """
4
5 def initialize(state):
6     state.initializeSlot("runtimes",[])
7     state.initializeSlot("is_on",False)
8     state.initializeSlot("turn_on_time",None)
9
10 def analyze(state, saveFigure, args):
11     #retrieve state into local variables
12     runtimes = state.retrieveSlot("runtimes")
13     #compute histogram of runtimes
14     (hist,bins) = np.histogram(runtimes,bins=15)
15     center = (bins[:-1]+bins[1:])/2
16     #create a matplotlib figure with histogram
17     plt.figure()
18     fig = plt.plot(center,hist)
19     plt.xlabel("Runtime (secs)")
20     plt.ylabel("Counts")
21     #store total number of runs
22     state.initializeSlot("count",len(runtimes))
23     #save figure for use in report
24     saveFigure(fig,"runtimes")

```

The analyze function generates a histogram plot from the runtimes array. In line 12 the runtime state slot is retrieved into a local variable. Lines 14–15 compute the histogram using Numpy's builtin histogram function. See the numpy documentation for a full list of arguments. This function returns the counts per bin in hist and the bin edges in bin. To plot the count per bin we compute the bin centers in lines 15. center and hist have the same length so they can be plotted together. Analyzers use matplotlib to generate graphics. Matplotlib's pyplot module is imported as plt. Line 17 creates a new figure and line 18 plots the histogram in this figure. Lines 18–19 set up the plot labels. Line 22 initializes a new state slot to store the total number of runtimes. This slot is used to populate the report template. Line 24 uses the saveFigure function to store the plot image so it can be used in the report template. This function takes two arguments, the figure and a unique string identifier.

Code: Report


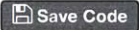
Select the **Report** tab in the code editor. Lines 1–3 are auto generated from the Analyzer *name* and

description. The double brackets in lines 5–6 indicate code injection. Line 5 displays the total number of runs by using the `count` state slot initialized in line 22 of `analyze`. Line 6 adds the histogram plot.

`insertFigure` takes a single argument, the unique string identifier of a figure.

```
1 Cycling Systems Analysis
2 -----
3 ### Histogram of runtime duration
4
5 Total number of runs: {{count}}
6 {{insertFigure("runtimes")}}
```

Testing the Analyzer

Select the **Test** tab from the right panel. Plot the turn ON and turn OFF events on the left axis and select a region of time to run the analyzer over. Notice there are several gaps in the data indicating times when the NILM was offline. Even if you expect to have an uninterrupted dataset it is always best to design the analyzer to support these kinds of gaps (eg with `check` at line 35 of `analyze`). Click  to start the analyzer. Select the **Report** tab to see the result. The histogram becomes smoother as more pump runtimes are included in the analysis. Try to display the max and min runtimes in the report. Experiment with different numbers of `bins` in `np.histogram` and different lengths of input data. Remember to click  to update changes before running the analyzer again.

Tutorial 3: Energy Dashboard

Analyzers use the `matplotlib` package to provide powerful plotting and graphical presentation tools.

`Matplotlib`'s `pyplot` is imported by default as `plt`. The code below shows how to create a basic graph using these tools.


```
1     fig = plt.figure()
2     x = np.arange(0,2*pi)
3     y = np.sin(x)
4     plt.plot(x,y)
5     #save the figure
6     saveFigure(fig,"example")
7
```

NILM Finder





Introduction

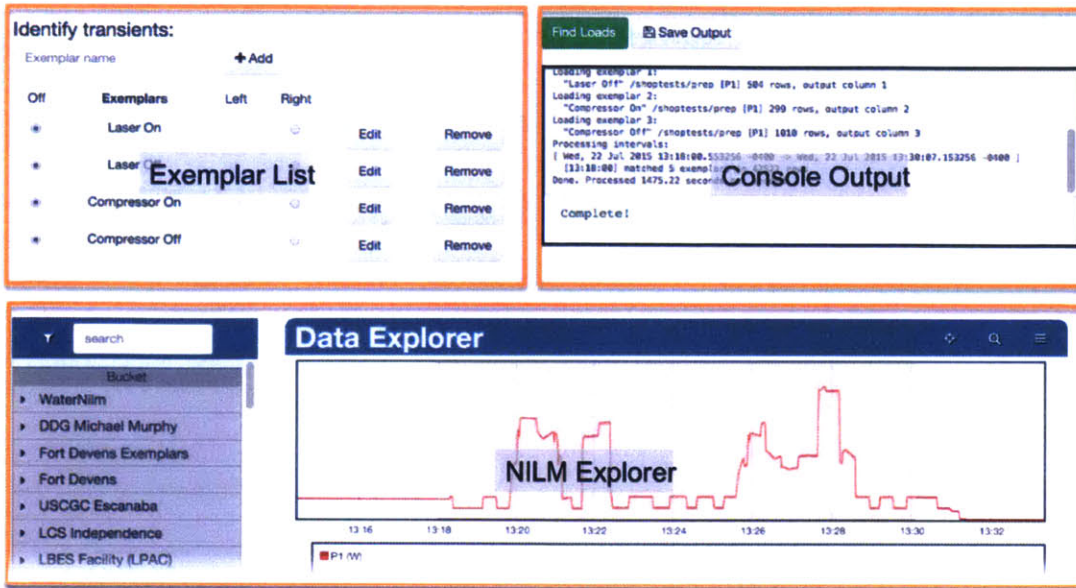
The NILM Finder uses exemplar pattern matching to identify load transients. Exemplars are short sections of a waveform that correspond to a load turn on or turn off event. Users identify and label exemplars and can then run a load identification filter across a dataset to extract matching transients. The exemplar engine uses a cross correlation algorithm to determine match events. An exemplar can consist of multiple streams (eg P1, Q1, P3, etc) but they must all be part of the same file.

Exemplar Groups

Exemplars are classified into groups. The NILM Finder page lists available exemplar groups. Click a group to enter the exemplar matching page (shown in the figure below). You can create a new exemplar group by clicking the blue button at the bottom of the group list. Each group must have a unique name. The  button deletes the group.

Identify Loads

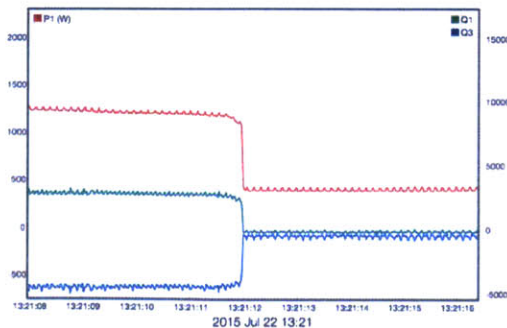
The Exemplar Matching interface has three main panels labeled in the figure below. Start by finding an isolated transient in the data using the embedded [NILM Explorer](#) interface (click the link for details about this interface). A transient exemplar can consist of multiple streams but *all streams must be in the same file*. Once you have zoomed in to just the section of data corresponding to a transient enter the name of the transient (eg Fridge On) in the **Exemplar List** and click . Click the name of an exemplar to display a thumbnail of the transient. Click the name again to hide the thumbnail.  displays the transient waveform in the NILM Explorer interface where you can change the bounds of the transient and add or remove streams. Once you are done adjusting the transient click  to update the exemplar with the new transient waveform. Click  to delete an exemplar from the group.



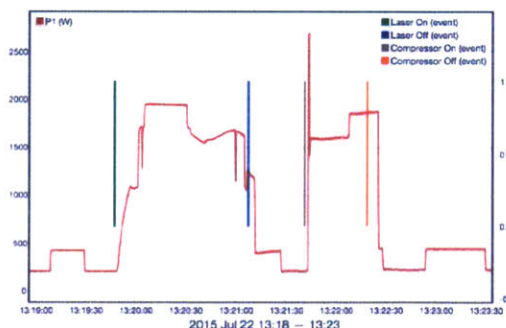
Exemplar matching interface

Match Loads

After you have built a list of exemplars you can then search a dataset for matches. The exemplar matching engine runs across the range of data plotted in the NILM Explorer interface. The exemplars do not have to be part of the dataset you are matching against but the target file must have all of the streams required by the exemplar. Therefore if a transient is defined as a section of P1 and Q1 waveforms it can be used to match any prep file that also has P1 and Q1 streams. Click **Find Loads** to start the exemplar matching engine. Once the process has completed you can plot the identified exemplars on the left or right axis by selecting the appropriate radio button in the Exemplar List. By default the matches are only stored in a temporary stream, if you want to save the match results click **Save Output** and select a destination group and file name *before* running the matching engine.



An off transient defined by P1, Q1 and Q3.
Transients should be 30 seconds or less and all



NILM Finder automatically identifies loads based on exemplar transients. Matched

streams must be in the same file.

exemplars are plotted as vertical pipes.



Process Manager

Introduction

Filters and Analyzers can be installed as NILM processes and run automatically. The NILM Process Manager shows currently installed processes and allows users to schedule new processes on demand. The NILM process framework is based around **NilmRun**, a remote execution service integrated with the NILM. The `nilmrun` server provides command line tools but these are only for advanced diagnostics and debugging.

The Process Manager interface is available to any user with privileges on the NILM but only `admins` and `owners` can schedule new processes. `admins` can remove currently executing processes using the **Administration** interface.

Process List

The Process Manager overview shows all installed processes. On a standalone system the only NILM listed will be the local device, but on a cluster this view will contain process listings from all of NILM's in the cluster. The **Process** column shows the type of process (Filter or Analyzer) and the name. The **Owner** is the name of the user who installed the process. **Nilm** is the serial number of the NILM holding the process. **Status** provides information about the state of the process. A green icon ● indicates a running process and a red icon ● indicates an inactive process. Processes may be inactive for several reasons. The process may be `waiting` for more data, `finished` if it is a one shot process, or halted with an `error` condition. If the process has failed with an error the return code is displayed in brackets. Processes can be scheduled to execute as one shot or to repeat as new data becomes available. **Repeat** processes are indicated with a ✓ in the Repeat column. Click  to view the console output from a process. The  button uninstalls the process from the NILM.

Process Manager

Process	Owner	Nilm	Status	Repeat	Action
Filter [Power Quality]	John Ledner	nilm1234	● Error [1]	✓	 
Filter [Median Filter]	John Ledner	nilm1234	● waiting	✓	 
Analyzer [Load Usage]	Sam Fallows	nilm8621	● running		 
Analyzer [Pump Health]	Jim Nance	nilmF923	● running		 
Filter [Harmonics]	John Ledner	nilm1234	● waiting	✓	 
Analyzer [FOB Dashboard]	Alex Trush	nilmF923	● running	✓	 
Filter [Harmonics]	Alice Fox	nilm8631	● Error[1]	✓	 

New Process

Click the date below the Main Plot to open the *Date Selector Overlay*

New Processes

To install a process click the **New Process** button at the bottom of the Process Manager page. There are two different types of processes, filters and analyzers. See [NILM Filters](#) and [NILM Analyzers](#) for more details. Select which type of process you would like to install. Next specify the start and end times for the process. There are several different time options. The start time can be set to **Earliest available** or to a specific date. If there is a large amount of data in the input streams and you only want to process new data going forward, set the start time to the current date. The end time can also be set to a specific date or to **Latest Available**. If you select Latest Available then you can also select **Repeat Process** which will continue to run the process indefinitely as new data becomes available. Selecting all three check boxes will process all data that is currently available and all future data as it arrives.

Process Type:

Filter

Start Time

Earliest available

2016-03-10

End Time

Latest available

2016-03-11

Process just the data bounded by the start and end times.

Process Type:

Filter

Start Time

Earliest available


End Time

Latest available

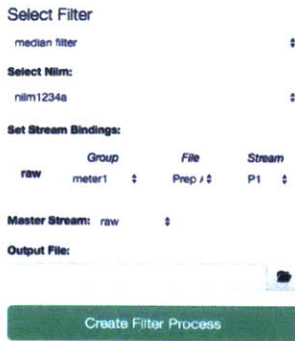
Repeat process?

Process all available data and any new data as it arrives.

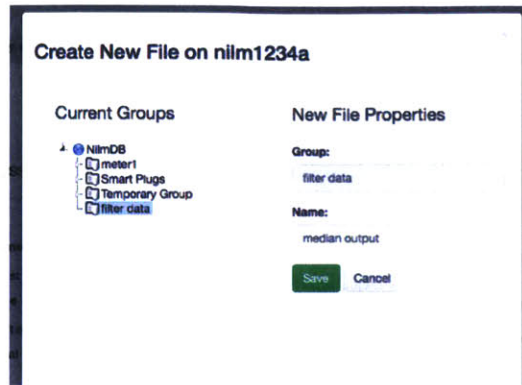
Filter Process

Select "filter" from the Process Type combo box to display the filter specific configuration options. Select the filter and the NILM you want to run it on. Any filter can be installed on any NILM. Filters have one or more inputs. These must be bound to data streams that are available on the NILM. If the filter was designed on the same NILM, the stream binding will be populated with the values used in the filter editor. Finally you must create an output file for the process. Clicking the  button brings up the file selector. Click the group where you want to store the file and enter a name for the file.

Once all of the settings are correct click **Create Filter Process**. The process will be automatically installed on the NILM and started. You will be redirected to the Process Manager page where you can check the status of the new process.



Filter Process configuration panel



Select a destination group and a unique file name

Analyzer Process

Select "analyzer" from the Process Type combo box to display the analyzer specific configuration options. Select the analyzer and the NILM you want to run it on. Any analyzer can be installed on any NILM. Analyzers have one or more inputs. These must be bound to data streams that are available on the NILM. If the analyzer was designed on the same NILM, the stream binding will be populated with the values used in the analyzer editor.

Next you must configure the report group structure. The group must have a unique name and description. The group should also be correlated with an installation. The installation should match the NILM where the process will be installed. The analyzer is run repeatedly across the data to generate reports which are stored in this group. The **Create Report Every** setting controls the frequency of reports and the **Each Report Covers** setting controls how much data is processed by each report. The figure below shows how these settings can be combined to create several different types of report structures. The units for these settings can be set to minutes, hours, or days.

Once all of the settings are correct click **Create Analyzer Process**. The process will be automatically installed on the NILM and started. You will be redirected to the Process Manager page where you can check the status of the new process.

Select Analyzer
 Pump Health

Select Nilim:
 nilmd200

Set Stream Bindings:

input	Group	File	Stream
data		prep-1	P1

Setup Report Group:

Name
 Pump Health

Description
 Runtime statistics for vacuum pump

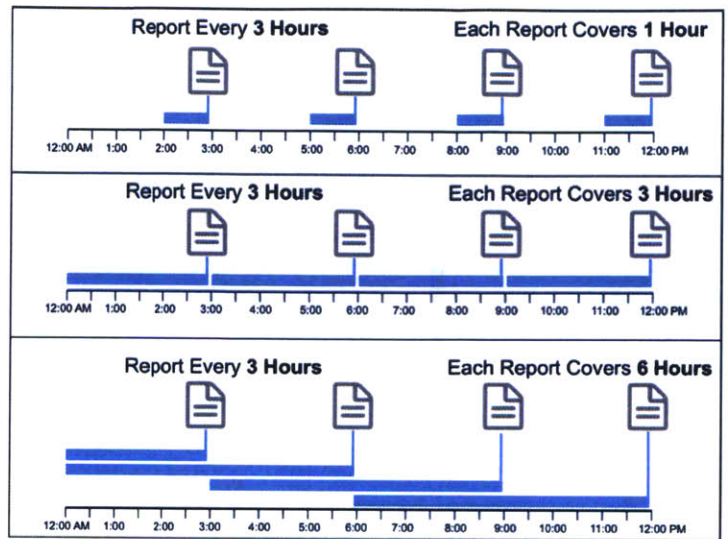
Installation
 Lab Bench

Create Report Every:
 2 hours

Each Report Covers:
 24 hours

Create Analyzer Process

Filter Process configuration panel



Filter Process configuration panel

NILM Command Line Interface

Introduction

Command-line arguments can often be supplied in both short and long forms, and many arguments are optional. The following documentation uses these conventions:

- An argument that takes an additional parameter is denoted `-f FILE`.
- The syntax `-f FILE`, `--file FILE` indicates that either the short form (`-f`) or long form (`--file`) can be used interchangeably.
- Square brackets (`[]`) denote optional arguments.
- Pipes (`A | B`) indicate that either `A` or `B` can be specified, but not both.
- Curly braces (`{ }`) indicate a list of mutually-exclusive argument choices.

Many of the programs support arguments that represent a NilMDB timestamp. This timestamp is specified as a free-form string, as supported by the `parse_time` client library function, described in Section 3.2.2.4 of the [NilMDB reference guide](#). Examples of accepted formats are shown in Table 3-19 on page 133 of that document.

NILM Diagnostics

`nilm-capture` is the global NILM service that manages data capture from all of the meters (contact and non-contact). This service is automatically managed by the NILM. The rest of the functions are diagnostic utilities that are useful when setting up an installation. **`nilm-scope`** provides a realtime waveform viewer similar to an oscilloscope. **`nilm-calibrate`** calibrates non-contact meters. **`nilm-check-config`** verifies the `meters.yml` syntax, checks if the specified meters are correctly connected, and estimates the disk usage required by the `keep` settings. **`usbstream`** prints non-contact USB meter data to standard output and **`ethstream`** does the same for the contact meters. The tools are designed to be used for temporary measurements while a user is setting up the system. *All of these tools will disable `nilm-capture` while they are running.* Finally, **`NILM logs`** describes the structure and location of log files generated by the NILM. These logs are useful for diagnostics and debugging installed systems.

`nilm-capture` – NILM system service

Usage

```
sudo service nilm-capture action
```

Description

This program controls the entire NILM acquisition and signal processing pipeline. It runs as a system

service. When `nilm-capture` is running data is collected from the meters, processed and stored as streams into NilmDB. When this service is not running the meters are idle and available for use by other diagnostic utilities. `nilm-capture` will automatically start on system boot. Stop data capture by issuing the `stop` action. After any change to the `meters.yml` file you must restart the daemon.

Arguments

`action`
`start|stop|restart|status` (specify a single action)

Example

```
1 $ sudo service nilm-capture restart #reload the meters.yml file and start
```

nilm-scope - View sensor waveforms

Usage

```
nilm-scope meter -c
```

Description

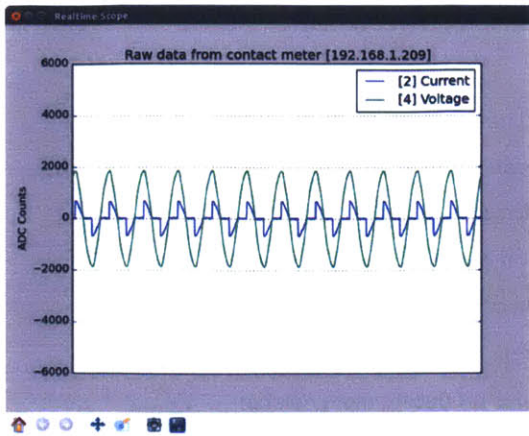
NILM scope can display the sensor waveforms for either a non-contact or contact meter in realtime. For a contact installation this can help determine the phase pairings between the current and voltage sensors and for a non-contact installation this can help when placing the sensors to ensure strong magnetic and electric field pickup. The program requires two parameters, the meter name from the `meters.yml` configuration file and a list of channels to display. A legend is automatically built using the configuration in `meters.yml`.

Arguments

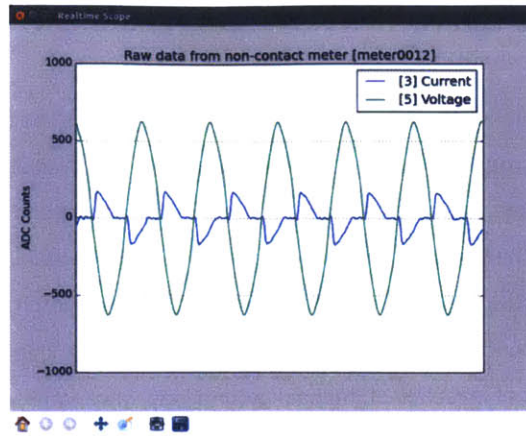
`meter`
meter name from `meters.yml` (`meter1,meter2, etc`)
`-c CHANNELS`
Space seperated list of channel indices to plot [0-5] for contact meters and [0-7] for non-contact meters

Example

```
1 $ nilm-scope meter1 -c 2 4 #display channels 2 and 4 from meter1
```



NILM Scope running on a contact meter



NILM Scope running on a non-contact meter

nilm-check-config - Verify meters.yml syntax and disk usage

Usage

nilm-check-config

Description

This command verifies the syntax of meters.yml. This command should be run after any change to the meters.yml file to ensure that the configuration is valid. If the syntax is valid, it then checks if the specified meters are connected to the NILM. Finally it calculates the disk space required by the keep settings and displays the estimated disk usage. If more space is required than currently available, adjust the keep settings for the meter streams and run this command again.

Arguments

none

Example

```

1 $ nilm-check-config #syntax error in meters.yml
2 [CONFIGURATION_ERROR]: meter1 error in [streams][sinefit][keep] bad syntax
3
4 $ nilm-check-config #no syntax errors, but warning that meter1 is missing
5 [CONFIGURATION_WARNING]: meter1 device [meter0012] is not connected
6 This configuration will require ~176.68 GiB
7 This is 41% of the available space on the disk

```

nilm-calibrate - Calibrate non-contact meters

Usage

nilm-calibrate meter

Description

This command calibrates a non-contact meter. The meter must be configured in `meters.yml`. The calibration routine requires you to connect a smart plug to an outlet on each phase. Before running this command make sure you have a smart plug, micro USB cable, and a resistive load like an incandescent lightbulb. Configure the `meters.yml` **calibration** section with the load **watts** and calibration **duration** in seconds. Longer durations generally improve the calibration result. In general use around 30 seconds for a house and 90 seconds or more a ship or larger building. The calibration wattage should be at least 5% of the background load with higher wattages producing more reliable calibration results. If the power system does not have a neutral bus (eg on a ship) set **has_neutral** to `false` otherwise leave it as `true`. This command can be used to calibrate a meter multiple times so you can experiment with different load sizes and calibration durations until the result is accurate.

Arguments

meter
meter name from `meters.yml` (meter1, meter2, etc)

Example

Before running this command make sure the `meters.yml` file is configured for this meter. In particular adjust the **calibration** section to match your setup. See [Setting Up a Non-Contact Meter](#) for details on configuring `meters.yml`

```
meter1:  
  ...other configurations...  
  calibration:  
    duration: 30           # length of calibration in seconds  
    watts: 200            # power consumed by calibration load  
    has_neutral: true     # [false] if the system has no neutral bus  
  ...other configurations...
```

Once you have configured `meters.yml` run `nilm-calibrate` as shown below:

```
1 $ nilm-calibrate meter1
2
3 Calibrating meter1
4 + The power system has 2 phases and neutral
5 + Digitally integrating sensor 0 for voltage measurement
6 + Using sensors 1,3,5,7 for current measurements
7 + Calibration load is 200W and will run for 30 seconds
8 + The reference voltage is 120V rms
9 + The meter serial number is [meter0001]
10 Is this correct? (y/n) y #answer n to cancel calibration
11 Set up a smart plug for calibration? (y/n) y #smart plug must be connected
12 #... calibration continues ...
```

usbstream - Stream raw sensor data non-contact meter (D-Board)

Usage

```
usbstream meter
```

Description

This command returns timestamped sample data from a non-contact meter D-Board. The meter must be configured in `meters.yml` before running this command. All eight channels are sampled at 3kHz and printed to standard output. The first column is a Unix microsecond timestamp. This command does not apply any scaling or calibration to the sensor values, the output is the raw ADC reading. This command stops the `nilm-capture` process.

Arguments

```
meter
  meter name from meters.yml (meter1, meter2, etc)
```

Example


```

1 $ usbstream meter1
2 1457475967447218 -7071 551 -7090 501 -7048 482 -16247 -16248
3 1457475967447551 -7071 558 -7087 501 -7048 483 -16246 -16248
4 1457475967447885 -7069 566 -7088 504 -7049 486 -16247 -16249
5 1457475967448218 -7067 561 -7089 505 -7046 478 -16248 -16249
6 1457475967448551 -7066 557 -7084 502 -7045 479 -16247 -16250
7 1457475967448884 -7064 561 -7082 504 -7046 484 -16248 -16251
8 ^Ccaught signal [2], stopping #hit Ctrl-C to stop
9 closed usb sensor
10
11 $usbstream meter1 > output.dat #save values to a file

```

ethstream - Stream raw sensor data from contact meter (LabJack)

Usage

ethstream [options]

Description

This command returns raw sample data from a LabJack UE9. The contact meter sensors are connected to channels 0-6 on the LabJack. See [Contact Meter](#) for a mapping of sensors to LabJack channels. There are many options to this command but the most useful for NILM debugging is `-a` address, `-C` channels, and `-L` to force LabJack mode. The default sample rate is 8kHz. The full options are listed below. Use the `-x` flag to show complete usage examples. Before running this command you must manually stop the `nilm-capture` service. When you are done manually interacting with the LabJack, restart the `nilm-capture` service.

Arguments

```

-a, --address stringhost/address of device (192.168.1.209)
-n, --numchannels n sample the first N ADC channels (2)
-C, --channels a,b,c sample channels a, b, and c
-r, --rate hz sample each channel at this rate (8000.0)
-L, --labjack Force LabJack device
-t, --timers a[:A],b[:B] set LabJack timer modes a,b and optional values A,B
-T, --timerdivisor n set LabJack timer divisor to n
-N, --nerdjack Force NerdJack device
-d, --detect Detect NerdJack IP address
-R, --range a,b Set range on NerdJack for channels 0-5,6-11 to either 5 or 10 (10,10)
-g, --gain a,b,c Set Labjack AIN channel gains: 0,1,2,4,8 in -C channel order
-o, --oneshot don't retry in case of errors
-f, --forceretry retry no matter what happens
-c, --convert convert output to volts/temperature
-H, --converthex convert output to hex
-m, --showmem output memory stats with data (NJ only)

```

```

-l, --lines num      if set, output this many lines and quit
-h, --help           this help
-v, --verbose        be verbose
-V, --version        show version number and exit
-i, --info           get info from device (NJ only)
-X, --examples       show ethstream examples and exit

```

Example

```
1 $ ethstream -a 192.168.1.209 -C 0,1,2 -L #record channels 0,1,2 (current sen
```

NILM Logs – System logging

Description

The NILM processes record logging information to a set of log files located in `/var/log/nilm`. The logs are automatically rotated daily and compressed. Logs older than 10 days are removed. View the logs using `tail`. Each meter has a log and the supervisor NILM daemon has a log.

supervisor.log

Global data NILM system events. This log is produced by the `nilm-capture` service.

meterX.log

Individual meter events. Each of these logs is produced by the `nilm-capture-daemon` threads spawned by the `globale nilm-capture` service.

Example

The logs are flat text files but due to their large size it is often easiest to view them using `tail` which displays the last 10 lines of a file (the most recent log events). Use the `-f` flag to follow the file and display new logging events as they are recorded.

Supervisor Correct Operation: *Supervisor log indicates that a capture process has been spawned for `eter1`*

```

1 $ tail /var/log/nilm/supervisor.log
2 2016-03-09 09:06:56,524:INFO:STDOUT:----- Starting Supervisor -----
3 2016-03-09 09:06:56,589:INFO:STDOUT:[supervisor]: waiting 2 minutes to avoi
4 2016-03-09 09:06:57,591:INFO:STDOUT:[supervisor]: starting capture for [met

```

Supervisor Error Condition: *Supervisor log indicates that `eter.y 1` is empty*


```

1 $ tail /var/log/nilm/supervisor.log
2 2016-03-07 13:01:27,386:INFO:STDOUT:----- Starting Supervisor -----
3 2016-03-07 13:01:27,387:INFO:STDOUT:[CONFIGURATION_ERROR]: meters.yml is em
4 2016-03-07 13:01:27,388:INFO:STDOUT:    see [http://nilm.standalone/help/so
5 2016-03-07 13:01:27,388:INFO:STDOUT:## Configuration has errors.
6 2016-03-07 13:01:27,388:INFO:STDOUT:## Run [nilm-check-config] to verify me

```

Meter Correct Operation: *Meter log indicates that data capture has started on eter1*

```

1 $ tail /var/log/nilm/meter1.log
2 2016-03-09 11:57:11,374:
3 2016-03-09 11:57:11,374:-----starting capture on meter1-----
4 2016-03-09 11:57:16,768:[meter1]: beginning interval

```

Meter Error Condition: *Meter log indicates that the non-contact meter (D-Board) is not connected*

eter1

```

1 $ tail /var/log/nilm/meter1.log
2 2016-03-09 11:56:51,378:-----starting capture on meter1-----
3 2016-03-09 11:56:51,735:##### ERROR #####
4 2016-03-09 11:56:51,735:[meter1] Cannot find USB meter with serial number [
5 2016-03-09 11:56:51,735:    check USB connection or set enabled=False in m
6 2016-03-09 11:56:51,735:    could not open port /dev/serial/by-id/usb-MIT_

```

nilmtool

Tools for interacting with the nilm database are wrapped in **nilmtool**, a monolithic multi-purpose program that provides command-line access to most of the NilMDB functionality. Global operation is described first followed by specific documentation for each subcommand.

nilmtool - Multipurpose NilMDB management tool

Usage

```

nilmtool [-h] [-v] [-u URL] {help, info, create, rename, list, intervals,
metadata, insert, extract, remove, destroy} ...

```

Description

Multipurpose tool that provides command-line access to most of the NilMDB functionality. The command-line syntax provides the ability to execute sub- commands: first, global arguments that affect

the behavior of all subcommands can be specified, followed by one subcommand name, followed by arguments for that subcommand. Each defines its own arguments and is documented independently.

Arguments

`-u URL, --url URL`

(default: `http://localhost/nilmdb/`) NilMDB server URL. Must be specified before the subcommand.

`subcommand ...`

The subcommand to run, followed by its arguments. This is required.

`-h, --help`

Print a help message with usage information and details on all supported command-line arguments.

This can also be specified after the subcommand, in which case the usage and arguments of the subcommand are shown instead.

`-v, --version`

Print the nilmtool version.

Environment Variables: Some behaviors of nilmtool subcommands can be configured via environment variables.

`NILMDB_URL`

(default: `http://localhost/nilmdb/`) The default URL of the NilMDB server. This is used if `--url` is not specified, and can be set as an environment variable to avoid the need to specify it on each invocation of nilmtool.

`TZ`

(default: system default timezone) The timezone to use when parsing or displaying times. This is usually of the form `America/New_York`, using the standard TZ names from the IANA Time Zone Database

nilmtool help - Print help for a subcommand

Usage

nilmtool help [-h] subcommand

Description

Print more specific help for a subcommand. `nilmtool help subcommand` is the same as `nilmtool subcommand --help`.

nilmtool info - Server information

Usage

nilmtool info [-h]

Description

Print server information such as software versions, database location, and disk space usage.

Example

```
1 $ nilmtool info
2 Client version: 1.9.7
3 Server version: 1.9.7
4 Server URL: http://localhost/nilmdb/
5 Server database path: /home/nilmdb/db
6 Server disk space used by NilMDB: 143.87 GiB
7 Server disk space used by other: 378.93 GiB
8 Server disk space reserved: 6.86 GiB
9 Server disk space free: 147.17 GiB
```

nilmtool create - Create a new stream

Usage

```
nilmtool create[-h] PATH LAYOUT
```

Description

Create a new empty stream at the specified path and with the specified layout.

Arguments

PATH

Path of the new stream. Stream paths are similar to filesystem paths and must contain at least two components. For example, /foo/bar.

LAYOUT

Layout for the new stream. Layouts are of the form <type>_<count>. The <type> is one of those described in Section 2.2.3 of the [NilMDB Reference Guide](#), such as uint16, int64, or float32. <count> is a numeric count of how many data elements there are, per row. Streams store rows of homogeneous data only, and the largest supported <count> is 1024. Generally, counts should fall within a much lower range, typically between 1 and 32. For example, float32_8.

nilmtool rename - Rename a stream

Usage

```
nilmtool rename [-h] OLDPATH NEWPATH
```

Description

Rename or relocate a stream in the database from one path to another. Metadata and intervals, if any, are relocated to the new path name.

Arguments

OLDPATH

Old existing stream path, e.g. /foo/old

NEWPATH

New stream path, e.g. /foo/bar/new

Notes

Metadata contents are not changed by this operation. Any software tools that store and use path names stored in metadata keys or values will need to update them accordingly.

nilmtool list - List streams

Usage

```
nilmtool list [-h] [-E] [-d] [-s TIME] [-e TIME] [-T] [-l] [-n] [PATH [PATH  
...]]
```

Description

List streams available in the database, optionally filtering by path, and optionally including extended stream info and intervals.

Arguments

PATH

(default: *) If paths are specified, only streams that match the given paths are shown. Wildcards are accepted; for example, /sharon/* will list all streams with a path beginning with /sharon/. Note that, to prevent wildcards from being interpreted by the shell, they should be quoted at the command line; for example:

```
1 $ nilmtool list "/sharon/*"  
2 $ nilmtool list "*raw"
```

-E, --ext

Show extended stream information, like interval extents, total rows of data present, and total amount of time covered by the stream's intervals.

-T, --timestamp-raw

When displaying timestamps in the output, show raw timestamp values from the NilMDB database rather than converting to human-readable times. Raw values are typically measured in microseconds since the Unix time epoch (1970/01/01 00:00 UTC).

-l, --layout

Display the stream layout next to the path name.

-n, --no-decim

Omit streams with paths containing the string "~decim-", to avoid cluttering the output with decimated streams.

-d, --detail

In addition to the normal output, show the time intervals present in each stream. See also nilmtool intervals in Section 3.2.3.7 of the [NilMDB Reference Guide](#), which can display more details about the intervals.

-s TIME, --start TIME

Starting timestamp for intervals (free-form, inclusive).

-e TIME, --end TIME

Ending timestamp for intervals (free-form, noninclusive).

nilmtool intervals - List intervals

Usage

nilmtool intervals [-h] [-d DIFFPATH] [-s TIME] [-e TIME] [-T] [-o] PATH

Description

List intervals in a stream, similar to `nilmtool list --detail`, but with options for calculating set-differences between intervals of two streams, and for optimizing the output by joining adjacent intervals.

Arguments

PATH

List intervals for this path.

-d DIFFPATH, --diff DIFFPATH

(default: none) If specified, perform a set-difference by subtract the intervals in this path; that is, only show interval ranges that are present in the original path but not present in diffpath.

-s TIME, --start TIME

Starting timestamp for intervals (free-form, inclusive).

-e TIME, --end TIME

Ending timestamp for intervals (free-form, noninclusive).

-T, --timestamp-raw

(default: min) (default: max) When displaying timestamps in the output, show raw timestamp values from the NilMDB database rather than converting to human-readable times. Raw values are typically measured in microseconds since the Unix time epoch (1970/01/01 00:00 UTC).

-o, --optimize

Optimize the interval output by merging adjacent intervals. For example, the two intervals [1 → 2) and [2 → 5) would be displayed as one interval [1 → 5).

nilmtool metadata - Manage stream metadata

Usage

```
nilmtool metadata [-h] PATH [-g [KEY ...] | -s KEY=VALUE [...] | -u KEY=VALUE  
[...]] | -d [KEY ...]]
```

Description

Get, set, update, or delete the key/value metadata associated with a stream.

Arguments

PATH

Path of the stream for which to manage metadata. Required, and must be specified before the action arguments.

Action Arguments: These actions are mutually exclusive.

-g [KEY ...], --get [KEY ...]

(default: all) Get and print metadata for the specified key(s). If none are specified, print metadata for all keys. Keys are printed as key=value, one per line.

-s [KEY=VALUE ...], --set [KEY=VALUE ...]

Set metadata. Keys and values are specified as a key=value string. This replaces all existing metadata on the stream with the provided keys; any keys present in the database but not specified on the command line are removed.

-u [KEY=VALUE ...], --update [KEY=VALUE ...]

Update metadata. Keys and values are specified as a key=value string. This is similar to --set, but only adds or changes metadata keys; keys that are present in the database but not specified on the

command line are left unchanged.

`-d [KEY ...], --delete [KEY ...]`

(default: all) Delete metadata for the specified key(s). If none are specified, delete all metadata for the stream.

Example

```
1 $ nilmtool metadata /temp/raw --set "location=Honolulu, HI" "source=NOAA"
2 $ nilmtool metadata /temp/raw --get
3 location=Honolulu, HI
4 source=NOAA
5 $ nilmtool metadata /temp/raw --update "units=F"
6 location=Honolulu, HI
7 source=NOAA
8 units=F
```

nilmtool insert - Insert data

Usage

```
nilmtool insert[-h] [-q] [-t] [-r RATE] [-s TIME | -f] [-e TIME] PATH [FILE]
```

Description

Insert data into a stream. This is a relatively low-level interface analogous to the `/stream/insert` HTTP interface described in Section 3.2.1.13 on the [NilmDB Reference Guide](#). This is the program that should be used when a fixed quantity of text-based data is being inserted into a single interval, with a known start and end time. If the input data does not already have timestamps, they can be optionally added based on the start time and a known data rate. In many cases, using the separate `nilm-insert` program is preferable, particularly when dealing with large amounts of pre-recorded data, or when streaming data from a live source.

Arguments

PATH

Path of the stream into which to insert data. The format of the input data must match the layout of the stream.

FILE

(default: standard input) Input data filename, which must be formatted as uncompressed plain text. Default is to read the input from stdin.

`-q, --quiet`

Suppress printing unnecessary messages.

Timestamping: To add timestamps to data that does not already have it, specify both of these arguments.

The added timestamps are based on the interval start time and the given data rate.

`-t, --timestamp`

Add timestamps to each line

`-r RATE, --rate RATE`

Data rate, in Hz

Start Time: The start time may be manually specified, or it can be determined from the input filename, based on the following options.

- s TIME, --start TIME
Starting timestamp for the new interval (free-form, inclusive)
- f, --filename
Use filename to determine start time

End Time: The ending time should be manually specified. If timestamps are being added, this can be omitted, in which case the end of the interval is set to the last timestamp plus one microsecond.

- e TIME, --end TIME
Ending timestamp for the new interval (free-form, noninclusive)

nilmtool extract - Extract data

Usage

```
nilmtool insert[-h] -s TIME -e TIME [-B] [-b] [-a] [-m] [-T] [-c] PATH
```

Description

Extract rows of data from a specified time interval in a stream, or output a count of how many rows are present in the interval.

Arguments

- PATH
Path of the stream from which to extract data.
- s TIME, --start TIME
Starting timestamp to extract (free-form, inclusive)
- e TIME, --end TIME
Ending timestamp to extract (free-form, noninclusive)

Output Formatting

- B, --binary
Output raw binary data instead of the usual text format. For details on the text and binary formatting, see the documentation of HTTP call /stream/insert in Section 3.2.1.13.
- b, --bare
Omit timestamps from each line of the output.
- a, --annotate
Include comments at the beginning of the output with information about the stream. Comments are lines beginning with #.
- m, --markup
Include comments in the output with information that denotes where the stream's internal intervals begin and end. See the documentation of the markup parameter to HTTP call /stream/extract in Section 3.2.1.14 for details on the format of the comments.
- T, --timestamp-raw
Use raw integer timestamps in the --annotate output instead of human-readable strings.
- c, --count
Instead of outputting the data, output a count of how many rows are present in the given time interval. This is fast as it does not transfer the data from the server.

nilmtool remove - Remove rows of data

Usage

```
nilmtool remove[-h] -s TIME -e TIME [-q] [-c] PATH [PATH ...]
```

Description

Remove all data from a specified time range within the stream at /PATH/. Multiple streams may be specified, and wildcards are supported; the same time range will be removed from all matching streams.

Arguments

PATH

Path(s) of streams. Wildcards are supported. At least one path must provided.

-s TIME, --start TIME

Starting timestamp of data to remove (free-form, inclusive, required).

-e TIME, --end TIME

Ending timestamp of data to remove (free-form, noninclusive, required).

Output Format

-q, --quiet

By default, matching path names are printed when removing from multiple paths. With this option, path names are not printed.

-c, --count

Display a count of the number of rows of data that were removed from each path.

Example

```
1 $ nilmtool remove -s @1364140671600000 -e @1364141576585000 -c "/sh/raw*"
2 Removing from /sh/raw
3 7239364
4 Removing from /sh/raw-decim-4
5 1809841
6 Removing from /sh/raw-decim-16
7 452460
```

nilmtool destroy - Destroy a stream

Usage

```
nilmtool destroy[-h] [-R] [-q] PATH [PATH ...]
```

Description

Destroy the stream at the specified path(s); the opposite of nilmtool create. Metadata related to the stream is permanently deleted. All data must be removed before a stream can be destroyed.

Wildcards are supported.

Arguments

PATH

Path(s) of streams. Wildcards are supported. At least one path must provided.

-R, --remove

If specified, all data is removed before destroying the stream. Equivalent to first running

```
nilmtool remove -s min -e max path.  
-q, --quiet  
    Don't display names when destroying multiple paths
```

Data Processing

The following section documents a variety of programs useful for processing and interacting with NILM data. Each program begins with the prefix `nilm-`.

Many of these programs are filters that process input from one or more source streams into a destination stream. Only regions of time that are present in the source, and not yet present in the destination, are processed. These programs can therefore be re-run with the same command-line arguments multiple times, and they will only process the newly available data each time.

nilm-copy - Copy data between streams

Usage

```
nilm-copy [-h] [-v] [-u URL] [-U DESTURL] [-D] [-F] [-n] [-s TIME] [-e TIME]  
SRCPATH DESTPATH
```

Description

Copy data and metadata from one stream to another. The source and destination streams can reside on different servers. Both streams must have the same layout.

Arguments

```
-u URL, --url URL  
    (default: http://localhost/nilmdb/) Nilmdb server URL for the source stream.  
-U DESTURL, --dest-url DESTURL  
    (default: same as URL) Nilmdb server URL for the destination stream. If unspecified, the same  
    URL is used for both source and destination.  
-D, --dry-run  
    Just print intervals that would be processed, and exit.  
-F, --force-metadata  
    Metadata is copied from the source to the destination. By default, an error is returned if the  
    destination stream metadata conflicts with the source stream metadata. Specify this flag to  
    always overwrite the destination values with those from the source stream.  
-n, --nometa  
    Don't copy or check metadata at all.  
-s TIME, --start TIME  
    (default: min) Starting timestamp of data to copy (free-form, inclusive).  
-e TIME, --end TIME  
    (default: max) Ending timestamp of data to copy (free-form, noninclusive).  
SRCPATH  
    Path of the source stream (on the source server).  
DESTPATH  
    Path of the destination stream (on the destination server).
```

nilm-copy-wildcard – Copy multiple streams*Usage*

```
nilm-copy-wildcard [-h] [-v] [-u URL] [-U DESTURL] [-D] [-F] [-n] [-s TIME]
[-e TIME] PATHS [...]
```

Description

Copy data and metadata, from multiple streams, between two servers. Similar to nilm-copy, except:

- Wildcards and multiple paths are supported in the stream names.
- Streams must always be copied between two servers.
- Stream paths must match on the source and destination server.
- If a stream does not exist on the destination server, it is created with the correct layout automatically.

Arguments

Most arguments are identical to those of nilm-copy (reference it for more details).

PATHS

Path(s) to copy from the source server to the destination server. Wildcards are accepted.

Example

```
1 $ nilm-copy-wildcard -u http://bucket/nilmdb -U http://pilot/nilmdb /bp/
2 Source URL: http://bucket/nilmdb/
3 Dest URL: http://pilot/nilmdb/
4 Creating destination stream /bp/startup/info
5 Creating destination stream /bp/startup/prep-a
6 Creating destination stream /bp/startup/prep-a-decim-4
7 Creating destination stream /bp/startup/prep-a-decim-16 ... etc
```

nilm-decimate – Decimate a stream one level*Usage*

```
nilm-decimate [-h] [-v] [-u URL] [-U DESTURL] [-D] [-F] [-s TIME] [-e TIME]
[-f FACTOR] SRCPATH DESTPATH
```

Description

Decimate the stream at SRCPATH and write the output to DESTPATH. The decimation operation is described in Section 2.4.1; in short, every FACTOR rows in the source are consolidated into one row in the destination, by calculating the mean, minimum, and maximum values for each column.

This program detects if the stream at SRCPATH is already decimated, by the presence of a decimate_source metadata key. If present, subsequent decimations take the existing mean, minimum, and maximum values into account, and the output has the same number of columns as

the input. Otherwise, for the first level of decimation, the output has three times as many columns as the input.

See also `nilm-decimate-auto` (Section 3.4.2.5) for a simpler method of decimating a stream by multiple levels.

Arguments

- `-u URL, --url URL`
(default: `http://localhost/nilmdb/`) NilmDB server URL for the source stream.
 - `-U DESTURL, --dest-url DESTURL`
(default: same as URL) NilmDB server URL for the destination stream. If unspecified, the same URL is used for both source and destination.
 - `-D, --dry-run`
Just print intervals that would be processed, and exit.
 - `-F, --force-metadata`
Overwrite destination metadata even if it conflicts with the values in the “metadata” section below.
 - `-s TIME, --start TIME`
(default: min) Starting timestamp of data to decimate (free-form, inclusive).
 - `-e TIME, --end TIME`
(default: max) Ending timestamp of data to decimate (free-form, noninclusive).
 - `-f FACTOR, --factor FACTOR`
(default: 4) Set the decimation factor. For a source stream with n rows, the output stream will have n/FACTOR rows.
- `SRCPATH`
Path of the source stream (on the source server).
- `DESTPATH`
Path of the destination stream (on the destination server).
- Metadata:** The destination stream has the following metadata keys added:
- `decimate_source`
The source stream from which this data was decimated.
 - `decimate_factor`
The decimation factor used.

nilm-decimate-auto – Decimate a stream completely

Usage

```
nilm-decimate-auto[-h] [-v] [-u URL] [-F] [-f FACTOR] PATH [...]
```

Description

Automatically create multiple decimation levels using from a single source stream, continuing until the last decimated level contains fewer than 500 rows total. Decimations are performed using `nilm-decimate` (Section 3.4.2.4). Wildcards and multiple paths are accepted. Destination streams are automatically named based on the source stream name and the total decimation factor; for example, `/test/raw-decim-4`, `/test/raw-decim-16`, etc. Streams containing the string

"~decim-" are ignored when matching wildcards.

Arguments

- u URL, --url URL
(default: http://localhost/nilmdb/) NilMDB server URL for the source and destination streams.
 - F, --force-metadata
Overwrite destination metadata even if it conflicts with the values in the "metadata" section above.
 - f FACTOR, --factor FACTOR
(default: 4) Set the decimation factor. Each decimation level will have 1/FACTOR as many rows as the previous level.
- PATH [...]
One or more paths to decimate. Wildcards are accepted.

nilm-insert - Insert data from an external source

Usage

```
nilm-insert [-h] [-v] [-u URL] [-D] [-s] [-m SEC] [-r RATE | -d] [-l | -f] [-o SEC] [-O SEC] PATH [INFILE ...]
```

Description

Insert a large amount of text-formatted data from an external source like eth- stream. This is a higher-level tool than nilmtool insert in that it attempts to intelligently manage timestamps. The general concept is that it tracks two timestamps:

1. The data timestamp is the precise timestamp corresponding to a particular row of data, and is the timestamp that gets inserted into the database. It increases by data_delta for every row of input. data_delta can come from one of two sources. If --delta is specified, it is pulled from the first column of data. If --rate is specified, data_delta is set to a fixed value of 1/RATE.
2. The clock timestamp is the less precise timestamp that gives the absolute time. It can come from two sources. If --live is specified, it is pulled directly from the system clock. If --file is specified, it is extracted from the input file every time a new file is opened for read, and from comments that appear in the files.

Small discrepancies between data and clock are ignored. If the data timestamp ever differs from the clock timestamp by more than max_gap seconds:

- If data is running behind, there is a gap in the data, so the timestamp is stepped forward to match clock.
- If data is running ahead, there is overlap in the data, and an error is returned. If --skip is specified, then instead of returning an error, data is dropped and the remainder of the current file is skipped.

Arguments

- u URL, --url URL
(default: http://localhost/nilmdb/) NilMDB server URL.

- D, --dry-run
Parse files and print information, but don't insert any data. Useful for verification before making changes to the database.
- s, --skip
Skip the remainder of input files if the data timestamp runs too far ahead of the clock timestamp. Useful when inserting a large directory of existing files with inaccurate timestamps.
- m SEC, --max-gap SEC
(default: 10.0) Maximum discrepancy between the clock and data timestamps.

Data timestamp

- r RATE, --rate RATE
(default: 8000.0) data_delta is constant 1/RATE (in Hz).
- d, --delta
data_delta is provided as the first number on each input line.

Clock timestamp

- l, --live
Use the live system time for the clock timestamp. This is most useful when piping in data live from a capture device.
- f, --file
Use filename and file comments for the clock timestamp. This is most useful when reading previously saved data.
- o SEC, --offset-filename SEC
(default: -3600.0) Offset to add to timestamps in filenames, when using --file. The default accounts for the existing practice of naming capture files based on the end of the hour in which they were recorded. The filename timestamp plus this offset should equal the time that the first row of data in the file was captured.
- O SEC, --offset-comment SEC
(default: 0.0) Offset to add to timestamps in comments, when using --file. The comment timestamp plus this offset should equal the time that the next row of data was captured.

Path and Input

- PATH
Path of the stream into which to insert data. The layout of the path must match the input data.
- INFILE [...]
(default: standard input) Input data filename(s). Filenames ending with .gz are transparently decompressed as they are read. The default is to read the input from stdin.

nilm-prep - Spectral envelope preprocessor

Usage

```
nilm-prep[-h] [-v] [-u URL] [-U DESTURL] [-D] [-F] [-s TIME] [-e TIME] [-c COLUMN] [-n NHARM] [-N NSHIFT] [-r DEG | -R RAD ] SRCPATH SINEPATH DESTPATH
```

Description

Perform the spectral envelope harmonic coefficient calculation described in Section 4.3.3. Two source streams are provided, one with the raw current data and one with marked zero crossings, typically created by nilm-sinefit (Section 3.4.2.10). The filter processes regions of time that are present in both source streams, and not present in the destination stream.

Arguments

- u URL, --url URL
(default: http://localhost/nilmdb/) NilMDB server URL for the source stream.
- U DESTURL, --dest-url DESTURL
(default: same as URL) NilMDB server URL for the destination stream. If unspecified, the same URL is used for both source and destination.
- D, --dry-run
Just print intervals that would be processed, and exit.
- F, --force-metadata
Overwrite destination metadata even if it conflicts with the values in the "metadata" section below.
- s TIME, --start TIME
(default: min) Starting timestamp of data to filter (free-form, inclusive).
- e TIME, --end TIME
(default: max) Ending timestamp of data to filter (free-form, noninclusive).

Preprocessor Arguments

- c COLUMN, --column COLUMN
Column number in SRCPATH to use for the raw data. The first data column is 1.
 - q NHARM, --nharm NHARM
(default: 4) Number of odd harmonics Nharm to compute and store. For example, Nharm = 2 will store P1, Q1, P3, and Q3.
 - N NSHIFT, --nshift NSHIFT
(default: 1) Number of shifted FFTs Nshift to compute, per period of the raw data. If the input frequency is 60 Hz, the data rate of the preprocessor output is Nshift x 60 Hz.
Note that the calculation used by the similar Kalman-filter preprocessor, described in [54], is equivalent to Nshift = 2.
 - r DEG, --rotate DEG
(default: 0.0°) Apply the additional rotation ϕ_{extra} to the FFT output, in degrees. Typically used to account for known phase offset between voltage and current. This is equivalent to adding a lag of ϕ_{extra} degrees to the zero crossing data.
This is also useful for three-phase systems. For example, the zero crossings can be calculated once with nilm-sinefit on ϕ_A voltage. Then, nilm-prep can be run on ϕ_A , ϕ_B , and ϕ_C currents using rotations of 0, 120, and 240 degrees. The order in which to apply these shifts will depend on the phase ordering in the measured system.
 - R RAD, --rotate-rad RAD
(default: 0 rad) Like --rotate, except specified in radians instead of degrees.
- SRCPATH
Path of the raw source stream, for example, /foo/raw.
- SINEPATH
Path of the sinefit source stream, for example, /foo/sinefit.
- DESTPATH
Path of the prep output, for example, /foo/prep. The destination stream must have $2 \cdot N_{harm}$ columns.

Metadata: The destination stream has the following metadata keys added:

- prep_raw_source
The source stream of the raw data from which these envelopes were calculated.
- prep_sinefit_source

The source stream of the marked zero crossings used for this data.
 prep_column
 The column number of the raw data in the raw data source.
 prep_rotation
 The applied rotation ϕ_{extra} for this data, in radians. prep_nshift The number of shifted FFTs
 Nshift for this data.

nilm-sinefit - Sinusoid fitting

Usage

```
nilm-sinefit [-h] [-v] [-u URL] [-U DESTURL] [-D] [-F] [-s TIME] [-e TIME]
[-c COLUMN] [-f FREQ] [-m MIN_FREQ] [-M MAX_FREQ]. [-a MIN_AMP] SRCPATH
DESTPATH
```

Description

Perform the 4-parameter sinefit fit calculation described in Section 4.3.2. Given a rough estimate of the frequency, this filter looks at successive windows of approximately 3 - 4 periods of the input waveform. For each window, it computes the least-squares best fit sinusoid.

At each of the positive zero crossings ($\phi = 0$) of the fit, the timestamped values f_0 , A , and C corresponding to the subsequent period are stored. The output stream will have one row of output per period of the input stream. The window sliding algorithm is designed to ensure that zero crossings do not occur near the window boundaries in order to reduce error. The fitted sinusoid is checked against frequency and amplitude limits. If the fit falls outside the given bounds, no data points are inserted into the destination stream for that particular window.

General Arguments

-u URL, --url URL
 (default: http://localhost/nilmdb/) NilmDB server URL for the source stream.
 -U DESTURL, --dest-url DESTURL
 (default: same as URL) NilmDB server URL for the destination stream. If unspecified, the same URL is used for both source and destination.
 -D, --dry-run
 Just print intervals that would be processed, and exit.
 -F, --force-metadata
 Overwrite destination metadata even if it conflicts with the values in the "metadata" section below.
 -s TIME, --start TIME
 (default: min) Starting timestamp of data to filter (free-form, inclusive).
 -e TIME, --end TIME
 (default: max) Ending timestamp of data to filter (free-form, noninclusive).

Sinefit Arguments

-c COLUMN, --column COLUMN
 Column number in SRCPATH to use for the source data. The first data column is 1.
 -f FREQ, --frequency FREQ
 (default: 60.0) Rough estimate of the input frequency, used only to determine the size of the

window to analyze and to set defaults for the minimum and maximum frequency. Given an average sampling rate f_s of the input data, the sine wave fit is performed against windows of $N = 3.5 \cdot f_s / \text{FREQ}$ points.

`-m MIN_FREQ, --min-freq MIN_FREQ`
(default: $f_{est}/2$) Minimum valid frequency f_0 of the fitted sinusoid.

`-m MAX_FREQ, --max-freq MAX_FREQ`
(default: $f_{est} \cdot 2$) Maximum valid frequency f_0 of the fitted sinusoid.

`-a MIN_AMP, --min-amp MIN_AMP`
(default: 20.0) Minimum valid amplitude A of the fitted sinusoid.

`SRCPATH`
Path of the raw source stream, for example, `/foo/raw`.

`DESTPATH`
Path of the fitted output parameters, for example, `/foo/sinefit`.

Metadata: The destination stream has the following metadata keys added:

`sinefit_source`
The source stream of the raw data used to fit these parameters.

`sinefit_column`
The column number used from the source stream.

Advanced Command Line Tools

These tools provide low level access to the NILM and are not required for normal system use. Be very careful running these commands.

Nilm Database

The primary NILM database is run as a daemon process and does not require any user interaction. However the primary database or any other database can be run from the command line using the `nilmdb-server` command. Like any complex data storage system, NilmDB is subject to corruption if not shutdown properly. The command line utility `nilmdb-fsck` is designed to verify database consistency and fix most problems that might arise. The primary cause of database corruption is a powerloss while the system is recording data. The management daemon automatically runs `fsck` when a corrupt database is detected. Therefore, this command should be used only on secondary or backup databases. To prevent NILM data from overflowing the available space the system automatically removes old data based off `keep` settings in `meters.yml`. The cleanup service can be run manually using `nilm-cleanup`.

nilmdb-server - Standalone NilmDB server

Usage

`nilmdb-server` `[-h]` `[-v]` `[-a ADDRESS]` `[-p PORT]` `[-d DATABASE]` `[-q]` `[-t]` `[-y]`

Description

Run the standalone NilMDB server. Note that the NilMDB server is typically run as a WSGI process as described in Section 3.1.1.3. This program runs NilMDB using a built-in web server instead.

Arguments

- v, --version
Print the installed NilMDB version.
- a ADDRESS, --address ADDRESS
(default: 0.0.0.0) Only listen on the given IP address. The default is to listen on all addresses.
- p PORT, --port PORT
(default: 12380) Listen on the given TCP port.
- d DATABASE, --database DATABASE
(default: ./db) Local filesystem directory of the NilMDB database.
- q, --quiet
Silence output.

Debug Options

- t, --traceback
Provide tracebacks in the error response for client errors (HTTP status codes 400 - 499). Normally, tracebacks are only provided for server errors (HTTP status codes 500 - 599).
- y, --yappi
Run under the yappi profiler and invoke an interactive shell afterwards. Not intended for normal operation.

nilmdb-fsck - Database check and repair

Usage

```
nilmdb-fsck [-h] [-v] [-f] [-n] DATABASE
```

Description

Check database consistency, and optionally repair errors automatically, when possible. Running this may be necessary after an improper shutdown or other corruption has occurred. This program will refuse to run if the database is currently locked by any other process, like the Apache webserver; such programs should be stopped first.

Arguments

- DATABASE
Local filesystem directory of the NilMDB database to check.
- f, --fix
Attempt to fix errors when possible. Note that this may involve removing intervals or data.
- n, --no-data
Skip the slow full-data check. The earlier, faster checks are likely to find most database corruption, so the data checks may be unnecessary.
- h, --help
Print a help message with usage information and details.
- v, --version
Print the installed NilMDB version. Generally, you should ensure that the version of nilmdb-fsck is newer than the NilMDB version that created, or last used, the given database.

Usage

nilm-cleanup[-h] [-v] [-u URL] [-y] [-e] CONFIGFILE

Description

Clean up old data from streams, using a configuration file to specify which data to remove. The configuration file is a text file in the following format:

```
1 [/stream/path]
2 keep = 3w # keep up to 3 weeks of data
3 rate = 8000 # optional, used for the --estimate option
4 decimated = false # whether to delete decimated data too
5 [*/prep]
6 keep = 3.5m # or 2520h or 105d or 15w or 0.29y
```

Stream paths are specified inside square brackets ([]) and are followed by configuration keywords for the matching streams. Paths can contain wildcards. Supported keywords are:

keep

How much data to keep. Supported suffixes are h for hours, d for days, w for weeks, m for months, and y for years.

rate

(default: automatic) Expected data rate. Only used by the `--estimate` option. If not specified, the rate is guessed based on the existing data in the stream.

decimated

(default: true) If true, delete decimated data too. For stream path `/A/B`, this includes any stream matching the wildcard `/A/B~decim*`. If specified as false, no special treatment is applied to such streams.

Arguments

`-u URL, --url URL`

(default: `http://localhost/nilmdb/`) NilmDB server URL.

`-y, --yes`

Actually remove the data. By default, `nilm-cleanup` only prints what it would have removed, but leaves the data intact.

`-e, --estimate`

Instead of removing data, print an estimated report of the maximum amount of disk space that will be used by the cleaned-up streams. This uses the on-disk size of the stream layout, the estimated data rate, and the space required by decimation levels. Streams not matched in the configuration file are not included in the total.

CONFIGFILE

Path to the configuration file.

Notes

The value `keep` is a maximum amount of data, not a cutoff time. When cleaning data, the oldest data in the stream will be removed, until the total remaining amount of data is less than or equal to `keep`. This means that data older than `keep` will remain if insufficient newer data is present; for example, if new data ceases to be inserted, old data will cease to be deleted.

Nilm Run

NilmRun includes a command line program that allows the server to be run in a standalone mode, using a built-in web server. Additionally, a set of tools is offered for running, listing, and removing processes.

nilmrun-server - Standalone NilmRun server

Usage

```
nilmrun-server[-h] [-v] [-a ADDRESS] [-p PORT] [-q] [-t]
```

Description

Run the standalone NilmRun server. Note that the NilmRun server is typically run as a WSGI process, as described in Section 3.1.2.3. Running it in standalone mode may be insecure, as no access control or authentication is supported.

Arguments

-v, --version
Print the installed NilmRun version.

-a ADDRESS, --address ADDRESS
(default: 0.0.0.0) Only listen on the given IP address. The default is to listen on all addresses.

-p PORT, --port PORT
(default: 12381) Listen on the given TCP port.

-q, --quiet
Silence output.

Debug Options

-t, --traceback
Provide tracebacks in the error response for client errors (HTTP status codes 400 - 499). Normally, tracebacks are only provided for server errors (HTTP status codes 500 - 599).

nilmrun-ps - List processes

Usage

```
nilmrun-ps[-h] [-v] [-u URL] [-n]
```

Description

List processes on a remote NilmRun server. Shows overall system information as well as detailed information about each process.

Arguments

-u URL, --url URL
(default: http://localhost/nilmrun/) NilmRun server URL. For servers that require authentication, it can be included in the URL in the form http://user:password@host/.

-n, --noverify
Disable SSL certificate verification.

Environment Variables

NILMRUN_URL

(default: http://localhost/nilmdb/) The default URL of the NilmRun server.

Output

The output of nilmrun-ps includes overall system information about the number of running processes, CPU usage, and memory usage on the server. Output fields are described below:

PID

Process ID.

STATE

Process status. This is “alive” if the process is still running, “done” if it has exited successfully, and “error” if it has exited with an error.

SINCE

Date and time that the process was started.

PROC

Number of operating system processes associated with this NilmRun process.

CPU

CPU usage of this process as a percentage of a single CPU core.

LOG

Length, in bytes, of the stored output for the process. This output can be retrieved when the process is removed with nilmrun-kill.

Example

```
1 $ nilmrun-ps
2 procs: 2 nilm, 157 other
3  cpu: 29% nilm, 96% other, 200% max
4  mem: 623 MiB used, 3965 MiB total, 16%
5 PID                                STATE SINCE                PROC CPU LOG
6 76d81854-feca-11e2-9b2c-00000002559 alive 08/06-15:00:30 2    3  1337
7 8168f08f-feca-11e2-87fe-0000000255c alive 08/06-15:00:48 1    26  4505
```

nilmrun-run - Run a command on a NilmRun server

Usage

nilmrun-run[-h] [-v] [-u URL] [-n] [-d] CMD [ARG [...]]

Description

Run a command on a NilmRun server. By default, this program will poll the command's output log and display it while waiting for the process to exit.

Arguments

-u URL, --url URL

(default: http://localhost/nilmrun/) NilmRun server URL. For servers that require authentication, it can be included in the URL in the form http://user:password@host/.

-n, --noverify

Disable SSL certificate verification.

Remote Program

-d, --detach

Run process and return immediately, without printing the command output. The process must be later removed with nilmrun-kill.

CMD [ARG [...]]

Remote command to execute, with arguments.

Environment Variables

NILMRUN_URL

(default: http://localhost/nilmdb/) The default URL of the NilmRun server.

nilmrun-kill - Kill/remove a process

Usage

nilmrun-kill[-h] [-v] [-u URL] [-n] [-q] PID [...]

Description

Kill or remove a process from the NilmRun server. This terminates all system-level processes that are running and removes the entry from the NilmRun process listing. Stored log output of the command, if any, is displayed after the process is removed.

Arguments

-u URL, --url URL

(default: http://localhost/nilmrun/) Nilmrun server URL. For servers that require authentication, it can be included in the URL in the form http://user:password@host/.

-n, --noverify

Disable SSL certificate verification.

Remote Program

-q, --quiet

Omit display of the command's final output log.

PID [...]

One or more process IDs to remove. Process IDs should be those listed by nilmrun-ps.

Environment Variables

NILMRUN_URL

(default: http://localhost/nilmdb/) The default URL of the Nilmrun server.

Appendix B

Implementation

This appendix contains design files, bill of materials, and selected source code for the hardware associated with this work. There are four sections in this appendix. Appendix B.1 covers the Flex Sensor. The Flex Sensor source code works on the D-Board hardware described in [11] with adjustments to the Makefile as described in the code README. Appendix B.2 covers the NILM Smart Plug. The board design is a drop in replacement for the control PCB in the Belkin WeMo. There are two hardware versions deployed as of this writing. They can be distinguished by the labeling on the front face of the plug. Plugs with “Belkin” work without modification, plugs with “wemo” require a custom interconnect cable described in the assembly section. Appendix B.3 covers the NILM Board, a single board computer designed to run the NILM software. The NILM Board a combined microcontroller and Freescale iMX6 platform that runs the NILM host software. This board uses an analog input channel for the sensor. It is recommended to use a commercial single board computer (eg Raspberry Pi) with the Flex Sensor since this sensor has an onboard ADC and connects over USB. Finally, Appendix B.4 covers server configuration for the currently deployed management cluster. These are implemented on a set of Dell servers running the Xen hypervisor.

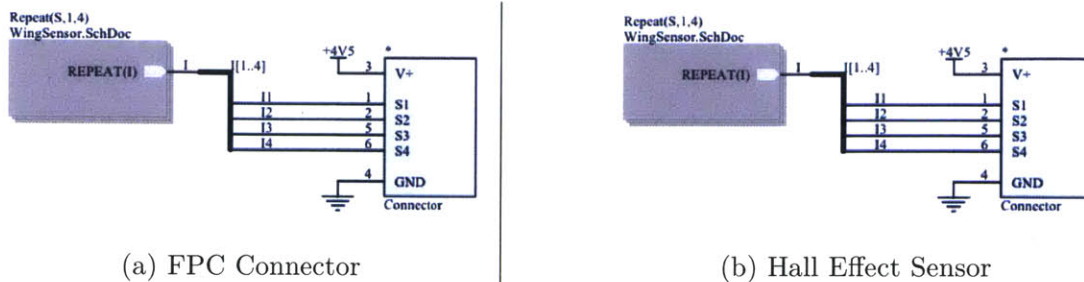


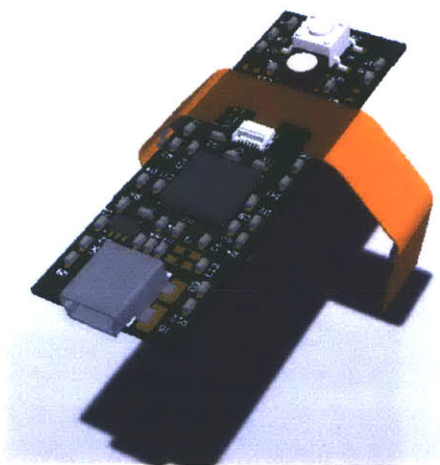
Figure B-1: Mylar pickup consists of an FPC connector (left) and four sensors (right)

B.1 Flex Sensor

The flex sensor integrates multiple magnetic sensors and an electric field sensor with an ARM microcontroller to provide a complete set of non-contact measurements to a NILM system. The sensor uses USB for both power and communication making it easy to connect to any standard computing platform including embedded devices, laptops, and desktop machines. The sensor enumerates as two TTY serial devices. The first serial device is for sensor data and the second interface is for I/O control of the button and LED which is used by embedded NILMs to provide feedback to the end user. Any standard serial program can be used to communicate with the sensor. The software designed in this thesis uses Python’s pySerial package which is an open source serial module available on Window’s, OS X, and Linux platforms.

The hardware consists of two components, the host PCB and a flexible Mylar pickup. The pickup connects to the host board with a friction fit FPC connection. The connection provides electrical connectivity but does not mechanically affix the pickup to the board. An adhesive should be applied to the pickup in order to assure a solid connection with the host PCB. Additionally, it is best practice to passivate both boards to prevent electrical shorts or corrosion of the circuits in a deployed environment.

The following pages present schematics, bills of material and CAD renderings for both the host PCB and Mylar pickup.



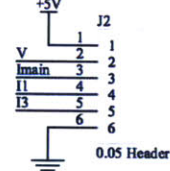
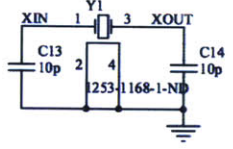
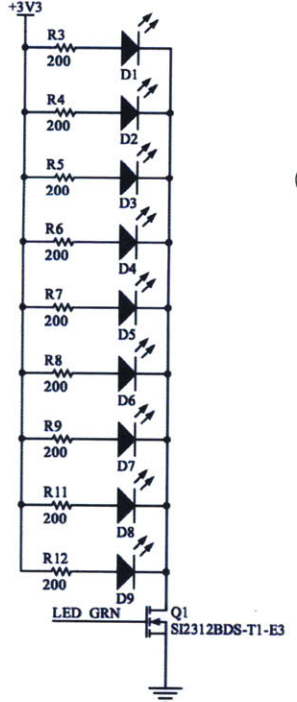
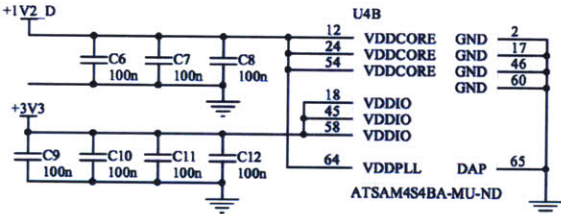
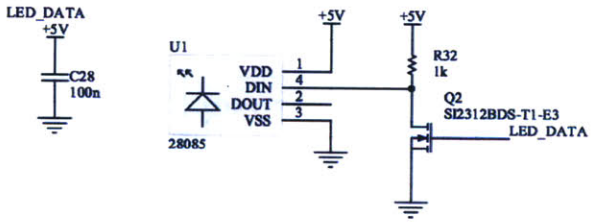
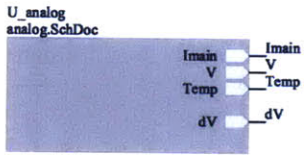
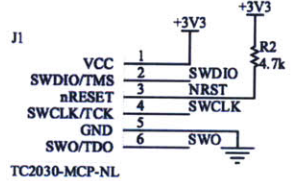
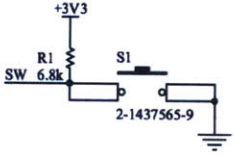
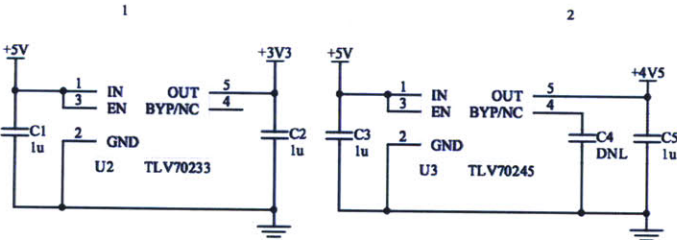
(a) CAD Rendering



(b) As Built

Figure B-4: Flex sensor prototype. The sensor (right) is coated in Plasti Dip for electrical passivation and increased mechanical resilience

Flex Sensor PCB Design

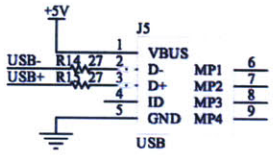
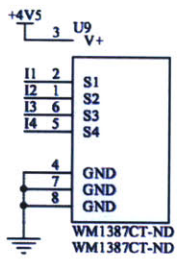
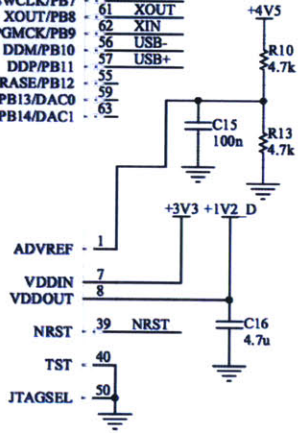


U4A

48	PA0/PGMEN0
47	PA1/PGMEN1
44	PA2/PGMEN2
43	PA3
36	PA4/PGMNCMD
35	PA5/PGMRDY
34	PA6/PGMNOE
32	LED GRN
31	PA7/XIN32/PGMINVALID
30	PA8/XOUT32/PGMM0
29	PA9/PGMM1
28	LED DATA
27	PA10/PGMM2
27	PA11/PGMM3
22	PA12/PGMD0
21	PA13/PGMD1
20	PA14/PGMD2
19	PA15/PGMD3
9	dV
10	PA16/PGMD4
10	PA17/PGMD5/AD0
12	PA18/PGMD6/AD1
11	PA19/PGMD7/AD2
13	PA20/PGMD8/AD3
14	PA21/PGMD9/AD8
14	PA22/PGMD10/AD9
23	PA23/PGMD11
25	PA24/PGMD12
25	PA25/PGMD13
26	PA26/PGMD14
37	PA27/PGMD15
38	PA28
41	PA29
42	PA30
52	PA31

ATSAM4S4BA-MU-ND

3	V
4	Imain
5	Temp
6	
33	TDI/PB4
49	SWO
51	SWDIO
53	SWCLK
61	XOUT
62	XIN
56	USB
57	USB+
55	DDP/PB11
55	ERASE/PB12
59	PB13/DAC0
63	PB14/DAC1



292

C

D

A

B

C

D

B.1.2 Selected Firmware Files

Listing B-1: /src/main.c: System initialization and main loop.

```
Git repository: http://git.wattsworth.net/nilm/iv-sensor
Filename: /src/main.c
Revision: master

1 #include <efc.h>
2 #include <pmc.h>
3 #include <sysclk.h>
4 #include <udi_cdc.h>
5 #include <wdt.h>
6 #include <board.h>
7 //#include <ioport.h>
8 #include <pio.h>
9 #include <twi.h>
10
11 #include "analog.h"
12 #include "buffer.h"
13 #include "debug.h"
14 #include "led.h"
15 #include "usb.h"
16 #include "pots.h"
17
18 void usb_led_update(void);
19 static void button_handler(void);
20
21 int main(void) {
22     // Switch over to the crystal oscillator
23     sysclk_init();
24
25     // Disable the built-in watchdog timer
26     wdt_disable(WDT);
27
28     //Board LED's
29     //pio_set_output(PIOA,PIO_PA7, HIGH, DISABLE, DISABLE);
30     //pio_set(PIOA,PIO_PA7);
31
32     // Initialize peripherals
33     usb_init();
34     analog_init();
35     led_init();
36
37     //Button
38     pmc_enable_periph_clk(ID_PIOA);
39     pio_set_input(PIOA, BUTTON_PIN, PIO_PULLUP);
40     pio_handler_set(PIOA, ID_PIOA, BUTTON_PIN, PIO_IT_EDGE, button_handler);
41     pio_enable_interrupt(PIOA, BUTTON_PIN);
42     NVIC_EnableIRQ(PIOA_IRQn);
43
44     // Default LED state
45     led_set(IO_LED, LED_OFF, 0);
46     led_set(DATA_LED, LED_GREEN, 0);
47
48     for (;;) {
49         // Transmit data if we can
50         while(udi_cdc_multi_is_tx_ready(DATA_PORT)) {
51             uint16_t data;
52             if(pop(&data) != BUFFER_OK) // buffer empty
53                 break;
54             udi_cdc_multi_write_buf(DATA_PORT, &data, sizeof(data));
55         }
56     }
```



```

56     if (buffer_full()) {
57         led_set(DATA_LED, LED_RED, 0);
58     }
59     // Receive data in our free time
60     uint8_t c;
61     //First check DATA_PORT interface
62     if (udi_cdc_multi_is_rx_ready(DATA_PORT)) {
63         c = udi_cdc_multi_getc(DATA_PORT);
64         if (c < 0) {
65             print("Read error");
66         }
67         switch(c){
68             case DATA_START:
69             analog_start();
70             led_set(DATA_LED, LED_BLUE, 0);
71             break;
72             case DATA_STOP:
73             analog_stop();
74             led_set(DATA_LED, LED_LT_GREEN, 0);
75             break;
76             case DATA_BOOTLOADER:
77             // Clear GPNVM 1 to boot from ROM instead of flash
78             efc_perform_command(EFC0, EFC_CMD_CGPB, 1);
79             break;
80             default:
81             //led_set(IO_LED, LED_BLUE, 0);
82             print("unknown command");
83         }
84     }
85     //Next check USERIO_PORT interface
86
87     if (udi_cdc_multi_is_rx_ready(USERIO_PORT)) {
88         c = udi_cdc_multi_getc(USERIO_PORT);
89         if (c < 0) {
90             print("Read error");
91         }
92         switch(c){
93             case IO_RGBLED:
94             usb_led_update();
95             break;
96             default:
97             // led_set(IO_LED, LED_RED, 0);
98             print("unknown command");
99         }
100     }
101
102 }
103 }
104
105 static void button_handler(void){
106     static int b_state = -1; //button starts out unknown
107     //grab the USB port
108     while(!udi_cdc_multi_is_tx_ready(USERIO_PORT));
109     //if its pressed AND the state is NOT pressed
110     if (pio_get(PIOA, PIO_INPUT, BUTTON_PIN) == 0 && b_state!=1){
111         udi_cdc_multi_putc(USERIO_PORT, 'p');
112         pio_set(PIOA, PIO_PA7);
113         b_state = 1; //new state is pressed
114     }
115     //if its released AND the state is NOT released
116     else if(b_state!=0){
117         udi_cdc_multi_putc(USERIO_PORT, 'r');
118         pio_clear(PIOA, PIO_PA7);
119         b_state=0; //new state is released
120     }
121 };
122

```



```

123 void usb_led_update(void){
124     //expect 4 bytes to specify new LED setting
125     // [RED, GREEN, BLUE, BLINK]
126     uint8_t red, green, blue, blink;
127     while(!udi_cdc_multi_is_rx_ready(USERIO_PORT));
128     red = udi_cdc_multi_getc(USERIO_PORT);
129     while(!udi_cdc_multi_is_rx_ready(USERIO_PORT));
130     green = udi_cdc_multi_getc(USERIO_PORT);
131     while(!udi_cdc_multi_is_rx_ready(USERIO_PORT));
132     blue = udi_cdc_multi_getc(USERIO_PORT);
133     while(!udi_cdc_multi_is_rx_ready(USERIO_PORT));
134     blink = udi_cdc_multi_getc(USERIO_PORT);
135     led_set(IO_LED, red, green, blue, blink);
136 }

```

Listing B-2: /src/analog.c: ADC data capture and digital filtering.

Git repository: <http://git.wattsworth.net/nilm/iv-sensor>

Filename: /src/analog.c

Revision: master

```

1  #include <adc.h>
2  #include <arm_math.h>
3  #include <buffer.h>
4  #include <pio.h>
5  #include <pmc.h>
6  #include <sysclk.h>
7  #include <tc.h>
8
9  #include "analog.h"
10 #include "led.h"
11 #include "fir_filter.h"
12
13 // ADC runs at 96 kHz per channel. There are 8 channels.
14 // Data is low-pass filtered by a zero-phase FIR filter which
15 // passes frequencies below 660 Hz and rejects above 1.5 kHz.
16 // Result is decimated by 32x and output at 3 kHz per channel.
17
18 // Raw ADC values are between 0 and 4095 inclusive.
19 // We subtract 2048 and the filter applies a DC gain of 8,
20 // so output values are nominally between -16384 and 16383.
21 // The filter L-infinity norm is 1.035, so pathological inputs
22 // can produce outputs ever so slightly outside of this range
23 // (but they will still fit comfortably in signed 16 bits).
24
25 // Frequency  Filter gain (/8)
26 //   60 Hz    0.9967
27 //  180 Hz    0.9951
28 //  300 Hz    0.9938
29 //  420 Hz    0.9915
30 //  540 Hz    0.9801
31 //  660 Hz    0.9443
32 // 1500 Hz    0.0501
33
34 #define NUM_CHANNELS 8
35
36 //-----two current production boards-----
37 #if(D_BOARD)
38 //mapping depends on sensor configuration
39 static enum adc_channel_num_t channels[NUM_CHANNELS] = {4,5,6,7,8,2,9,3};
40 #elif (FLEX_SENSOR)
41 // 0:dV, 4:V, 6:T, 3:I1, 2:I2, 8:I3, 9:I4, 5:IM
42 static enum adc_channel_num_t channels[NUM_CHANNELS] = {0,3,2,9,8,5,4,6};
43 #else

```

```

44 #error "Define D_BOARD or FLEX_SENSOR (see README)"
45 #endif
46
47
48
49 // Stage 1: decimate 8 times with a 16-tap FIR.
50 #define DEC1 8
51 #define NTAPS1 16
52 static const q15_t coeffs1[NTAPS1] = { // see util/fir.py
53     2305, 2296, 4354, 6819, 9420, 11818, 13661, 14663,
54     14663, 13661, 11818, 9420, 6819, 4354, 2296, 2305
55 };
56 static q15_t buffer1[NUM_CHANNELS][2*NTAPS1];
57
58 // Stage 2: decimate 4 times with a 32-tap FIR.
59 #define DEC2 4
60 #define NTAPS2 32
61 static const q15_t coeffs2[NTAPS2] = { // see util/fir.py
62     64, 147, 234, 261, 139, -195, -715, -1257,
63     -1529, -1196, 0, 2105, 4880, 7823, 10297, 11710,
64     11710, 10297, 7823, 4880, 2105, 0, -1196, -1529,
65     -1257, -715, -195, 139, 261, 234, 147, 64
66 };
67 static q15_t buffer2[NUM_CHANNELS][2*NTAPS2];
68
69
70 // The output format consists of the eight 16-bit channel values
71 // followed by the alignment word 0x807F and a 16-bit status word.
72 // (Note that it is never possible for 0x80 or 0x7F to be the
73 // most significant byte of the channel value or status word.)
74
75 static volatile int fifo_running;
76 static uint16_t status_mask;
77 static int fifo_write(uint16_t value) {
78     int r = push(value);
79     if (r != BUFFER_OK) {
80         status_mask |= ERROR_FIFO;
81     }
82     return r;
83 }
84
85
86 void analog_init(void) {
87     // Enable the switch
88     pmc_enable_periph_clk(ID_PIOA);
89     pio_set_input(PIOA, PIO_PA23, PIO_PULLUP);
90
91     // Initialize the ADC
92     pmc_enable_periph_clk(ID_ADC);
93     adc_init(ADC, sysclk_get_cpu_hz(), 20000000, ADC_STARTUP_TIME_8);
94     adc_configure_timing(ADC, 0, ADC_SETTLING_TIME_0, 1); // "fast"
95
96     // Set up ADC channel sequence
97     adc_configure_sequence(ADC, channels, NUM_CHANNELS);
98     adc_start_sequencer(ADC);
99     for (int i = 0; i < NUM_CHANNELS; ++i)
100         adc_enable_channel(ADC, i); // by *sequence #*, not channel #
101
102     // Enable end-of-conversion (EOC) interrupt for the last channel
103     adc_enable_interrupt(ADC, 1 << channels[NUM_CHANNELS-1]);
104     NVIC_EnableIRQ(ADC_IRQn);
105
106     // Trigger from timer 0
107     pmc_enable_periph_clk(ID_TC0);
108     tc_init(TC0, 0, // channel 0
109            TC_CMR_TCCLKS_TIMER_CLOCK1 // source clock (CLOCK1 = MCLK/2)
110            | TC_CMR_CPCTRIG // up mode with automatic reset on RC match

```

```

111         | TC_CMR_WAVE           // waveform mode
112         | TC_CMR_ACPA_CLEAR    // RA compare effect: clear
113         | TC_CMR_ACPC_SET);    // RC compare effect: set
114 tc_write_ra(TC0, 0, 1);
115 tc_write_rc(TC0, 0, 625); // frequency = (120MHz/2)/625 = 96 kHz
116 adc_configure_trigger(ADC, ADC_TRIG_TIO_CH_0, 0);
117
118 // Start everything
119 adc_start(ADC);
120 tc_start(TC0, 0);
121 }
122
123 void analog_start(void) {
124     flush();
125     status_mask = 0;
126     fifo_running = 1;
127 }
128
129 void analog_stop(void) {
130     fifo_running = 0;
131     flush();
132 }
133
134 void ADC_Handler(void) {
135     static uint32_t sample_index; // counts up forever
136
137     //voltage detection variables
138     #define WINDOW_SIZE 1000
139     #define VOLTAGE_THRESH 100000
140     static int32_t window_index = 0;
141     static int32_t last_mean = 0;
142     static int32_t raw_bucket = 0;
143     static int32_t sqr_bucket = 0;
144     int32_t signed_result = 0;
145
146     for (int i = 0; i < NUM_CHANNELS; i++) {
147         q15_t adc = adc_get_channel_value(ADC, channels[i]) - 2048;
148         // TODO: make ADC error logging PER-CHANNEL.
149         if (adc < -2000 || adc > 2000)
150             status_mask |= (1 << i);
151         buffer1[i][sample_index & (2*NTAPS1-1)] = adc;
152     }
153
154     // Update the switch status
155     if (pio_get(PIOA, PIO_INPUT, PIO_PA23) == 0)
156         status_mask |= SWITCH_PRESSED;
157
158     // We distribute the computational load of running FIR filters
159     // so that the USB interrupt never gets delayed too long.
160     // Let N = sample_index. Run stage 1 on channel N % 8.
161     // If N % 4 = 0, run stage 2 on channel (N/4) % 8.
162     // This computation is hard-coded for NUM_CHANNELS = DEC1.
163     // Quantities DEC1, DEC2, NTAPS1, NTAPS2 must be powers of 2.
164
165     // STAGE 1
166     int base = (sample_index / DEC1) * DEC1;
167     int channel = sample_index & (NUM_CHANNELS-1);
168     q15_t result = fir_filter(coeffs1, NTAPS1, buffer1[channel], base);
169     buffer2[channel][(base / DEC1) & (2*NTAPS2-1)] = result;
170
171     // STAGE 2
172     if (!(sample_index & (DEC2-1))) {
173         base = (sample_index / DEC1 / DEC2) * DEC2;
174         channel = (sample_index / DEC2) & (NUM_CHANNELS-1);
175         result = fir_filter(coeffs2, NTAPS2, buffer2[channel], base);
176
177         // Output result to the FIFO.

```

```

178 // If the status word is successfully written, clear it.
179 if (fifo_running) {
180     if (channel == 0) {
181         if (fifo_write(0x807F) == BUFFER_OK &&
182             fifo_write(status_mask) == BUFFER_OK) {
183             status_mask = 0;
184         }
185     }
186     // If the buffer is full, stop writing data to the buffer.
187     // The host will have to issue a new "start" command to resume.
188     if (fifo_write(result) != BUFFER_OK) {
189         fifo_running = 0;
190     }
191 }
192
193 //-----voltage detection code-----
194 if(channel==0){
195     signed_result = result;
196     raw_bucket+=signed_result;
197     if(last_mean!=0)
198     sqr_bucket+=(signed_result-last_mean)*(signed_result-last_mean);
199     window_index++;
200     if(window_index>=WINDOW_SIZE){
201         //run the voltage detection routine
202         if(last_mean!=0){
203             if((sqr_bucket/WINDOW_SIZE)>=VOLTAGE_THRESH)
204                 led_set(GREEN_LED,LED_GRN_ON);
205             else
206                 led_set(GREEN_LED,LED_GRN_OFF);
207         }
208         //update the mean
209         last_mean = raw_bucket/WINDOW_SIZE;
210         //reset the buckets and index
211         raw_bucket = 0;
212         sqr_bucket = 0;
213         window_index = 0;
214     }
215 }
216 //-----end voltage detection code-----
217 }
218 ++sample_index;
219 }

```

Listing B-3: Makefile: Makefile for Non-Contact Sensor. Set flags for serial number and hardware platform

```

Git repository: http://git.wattsworth.net/nilm/iv-sensor
Filename: Makefile
Revision: master

```

```

1
2 #
3 #-----CONFIGURE THESE FLAGS (see README)-----
4
5 # **** Board Serial Number *****
6 CFLAGS = -D'SERNO="meterXXXX"'
7 # **** Board Type (select one) *****
8 #CFLAGS += -D'D_BOARD' # D-Board with A-board sensors
9 CFLAGS += -D'FLEX_SENSOR' # Flex all in one board
10
11
12
13 #

```



```

14 #-----DO NOT EDIT BELOW THIS LINE-----
15 #
16
17 # Makefile for Atmel SAM4S using cmsis and GNU toolchain.
18
19 # The variables $(SRC), $(INC), $(LIB) are defined in path.mk.
20 include path.mk
21
22 # Object file location and linker script
23 OBJ = $(SRC:%.c=obj/%.o) $(LIB)
24 LD_SCRIPT = asf/sam/utils/linker_scripts/sam4s/sam4s4/gcc/%.ld
25
26 # Compiler and linker flags. Here be dragons.
27 CFLAGS += -mlittle-endian -mthumb -mcpu=cortex-m4
28 CFLAGS += -g -O3 $(INC:%=-I%) -std=c99 -Wall
29 CFLAGS += -DARM_MATH_CM4 -D'_SAM4S4B_' -D'BOARD=USER_BOARD'
30 LFLAGS = $(CFLAGS) -T$(@:bin/%.elf=$(LD_SCRIPT))
31 LFLAGS += -Wl,--entry=Reset_Handler -Wl,--gc-sections
32
33
34 # Targets
35 .PHONY: all clean gdb
36 .SECONDARY: $(OBJ)
37 all: bin/flash.bin bin/flash.elf
38 clean:
39     -rm -rf obj bin
40 gdb: bin/flash.elf
41     @arm-none-eabi-gdb
42 bin/%.bin: bin/%.elf
43     arm-none-eabi-objcopy -O binary $< $@
44 bin/%.hex: bin/%.elf
45     arm-none-eabi-objcopy -O ihex $< $@
46 bin/%.elf: $(OBJ)
47     @mkdir -p $(dir $@)
48     $(info LD $@)
49     @arm-none-eabi-gcc $(LFLAGS) -o $@ $(OBJ)
50 obj/%.o: %.c
51     @mkdir -p $(dir $@)
52     $(info CC $<)
53     @arm-none-eabi-gcc $(CFLAGS) -c $< -o $@

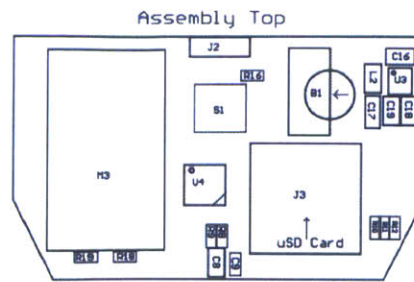
```

B.2 NILM Smart Plug

The NILM smart plug is a retrofitted Belkin WeMo. The stock control PCB is removed and replaced with a custom PCB that adds several additional features. With the custom control PCB the plug can store up to four years of power measurements and accurately timestamp each measurement using a battery backed real time clock. The smart plug connects to a NILM over USB or WiFi and can both transmit power data and receive commands to turn on and off the plug or set the RGB LED to a particular color or pattern. The stock control PCB only reports the power usage of the plug, but the solid state power meter chip (Figure B-7) actually collects many more metrics. The custom control PCB unlocks these additional metrics recording

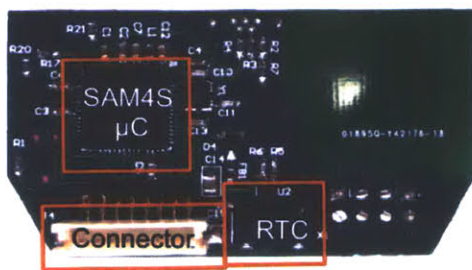


(a) Board Components

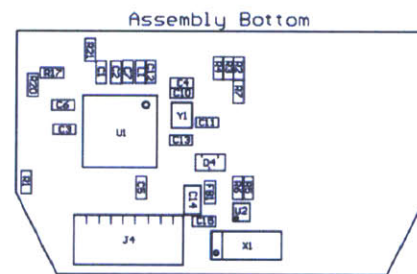


(b) Assembly Drawing

Figure B-5: Custom smart plug control PCB top view



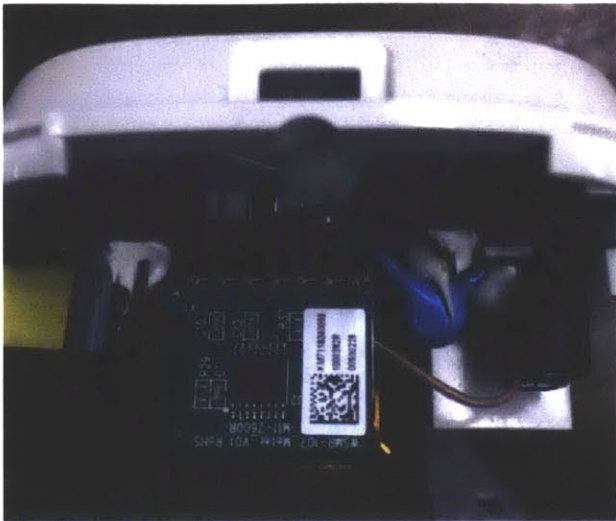
(a) Board Components



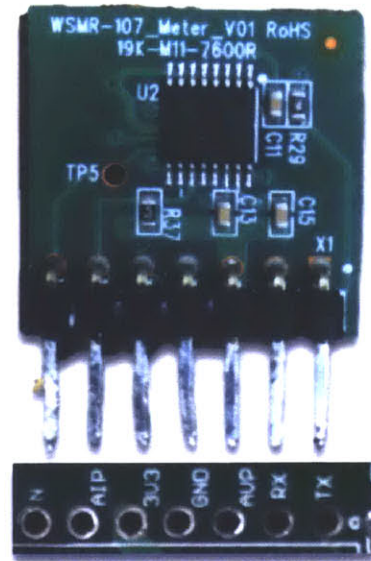
(b) Assembly Drawing

Figure B-6: Custom smart plug control PCB bottom view

not only wattage but also line frequency, voltage, current, and power factor. The following figures show the design and construction of the custom PCB and how it fits into the WeMo plug. See Appendix A.2.3 for documentation on configuring and using these smart plugs with a NILM system.

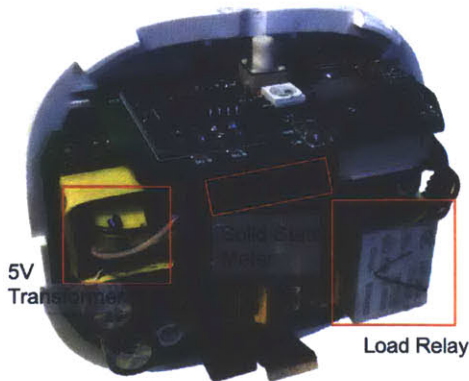


(a) Meter attached to WeMo

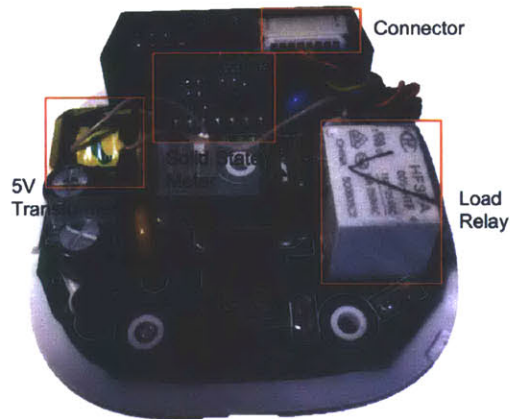


(b) Meter PCB

Figure B-7: The WeMo uses a solid state power meter board that communicates with the control PCB by optically isolated UART

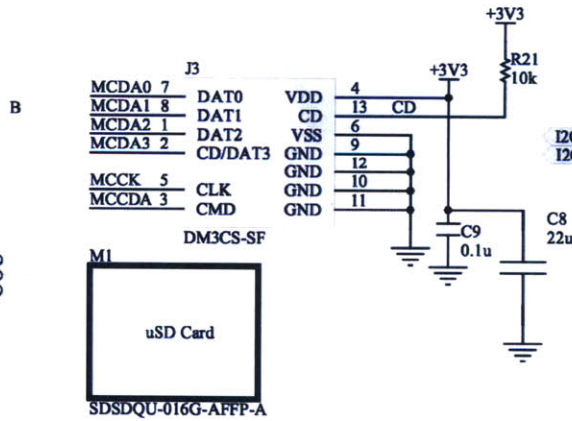
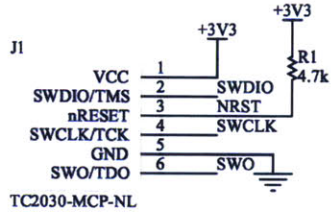
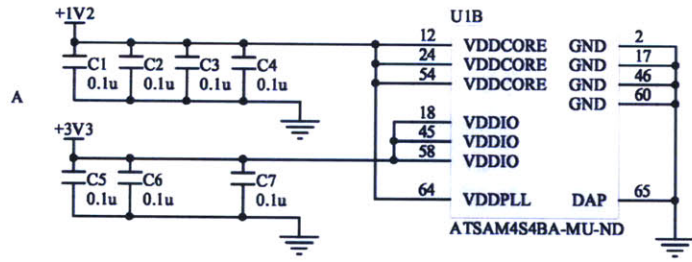


(a) as viewed from top

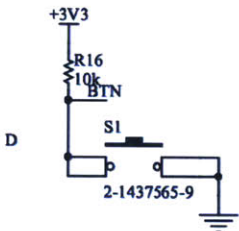
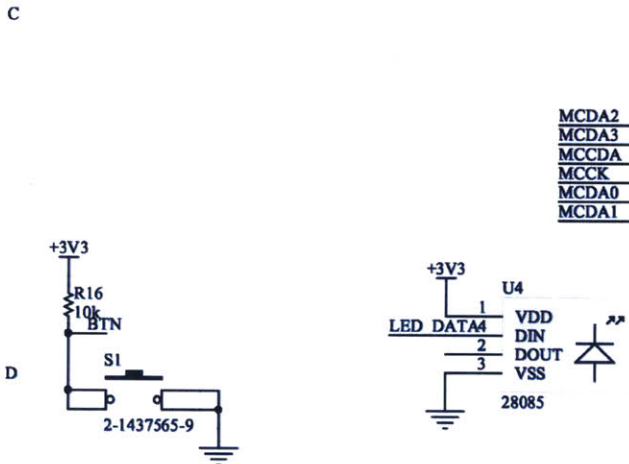
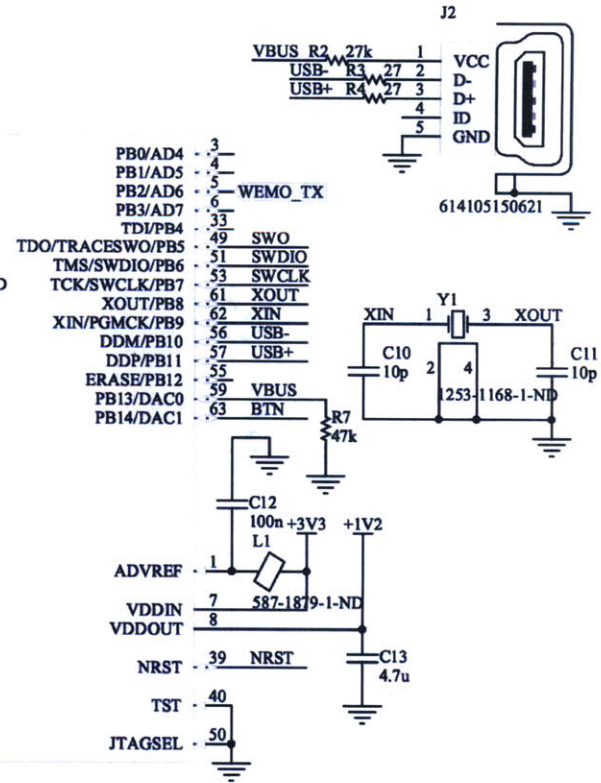
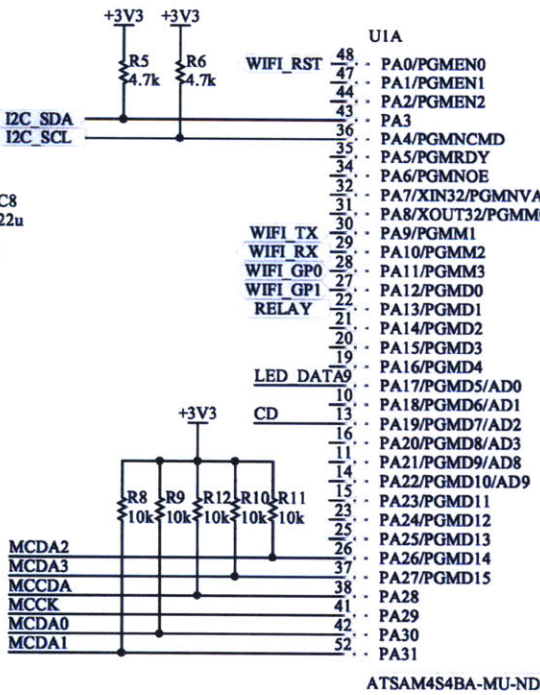


(b) as viewed from bottom

Figure B-8: Assembled smart plug with custom control PCB

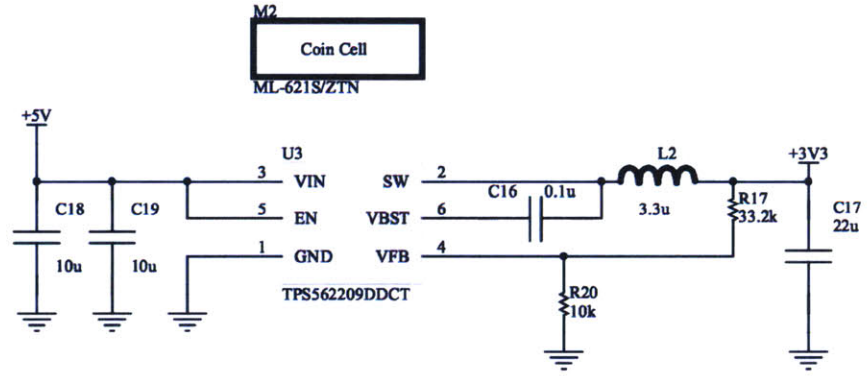
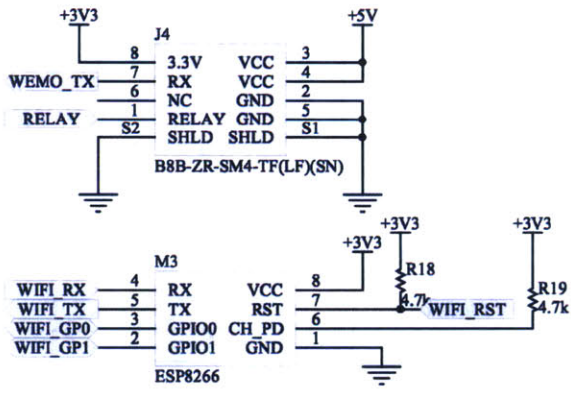
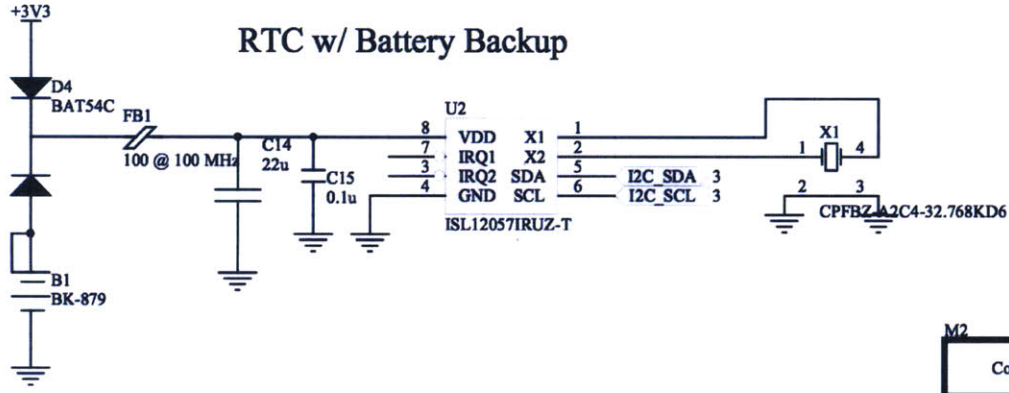


302



Title	Size	Number	Revision
	A		
Date:	12/3/2015		Sheet of
File:	C:\Users\...\wemo_board.SchDoc		Drawn By:

RTC w/ Battery Backup



303

Title

Size Number

Revision

Date: 12/3/2015
File: C:\Users\...\wemo_board2.SchDoc

Sheet of
Drawn By:

WEMO Control Board BOM

Plug Board BOM

304

Parts	Qty	\$100	\$100	Production Mfg_NK	Production P/N_NK	Production Description_NK
B1	1	0.30000	0.30000	MPD	BK-879	RETAINER 6.8MM COIN CELL SMD
C17	1	0.21000	0.21000	TDK	C3216X5R0J226K/1.6	CAP CER 22UF 6.3V X5R 1206
J2	1	1.66000	1.66000	Würth Elektronik	614105150621	CONN RCPT USB TYPE AB
J3	1	1.29360	1.29360	Hirose	DM3CS-SF	CONN MICRO SD CARD HINGED TYPE
J4	1	0.67070	0.67070	JST	B8B-ZR-SM4-TF(LF)(SN)	CONN HEADER ZH TOP 8POS 1.5MM
L2	1	0.23660	0.23660	TDK	MLP2012S3R3MT0S1	FIXED IND 3.3UH 900MA 190 MOHM
S1	1	0.45500	0.45500	APEM	MJTP1138CTR	SWITCH TACTILE SPST-NO 0.05A 12V
U1	1	4.94170	4.94170	Atmel	ATSAM4S4BA-MU	IC MCU 32BIT 256KB FLASH 64QFN
U2	1	1.81770	1.81770	Intersil	ISL12057IRUZ-T	IC RTC CLK/CALENDAR I2C 8-UTDFN
U3	1	1.09020	1.09020	Texas Instruments	TPS562209DDCT	IC REG BUCK ADJ 2A SYNC SOT-23-6
X1	1	0.40820	0.40820	Cardinal Components	CPFYZ-A2C4-32.768KD6	CRYSTAL 32.7680KHZ 6PF SMD
Y1	1	0.47000	0.47000	AVX	CX3225GB12000P0HPQCC	Crystals 12000kHz 20pF With Thermistor
R3, R4	2	0.00570	0.01140	Yageo	RC0603FR-0727RL	RES SMD 27 OHM 1% 1/10W 0603
C16	1	0.02320	0.02320	Yageo	CC0805KRX7R9BB104	CAP CER 0.1UF 50V X7R 0805
C18, C19	2	0.10050	0.20100	Yageo	CC0805MKX5R8BB106	CAP CER 10UF 25V X5R 0805
C8, C14	2	0.09040	0.18080	Yageo	CC0805MKX5R5BB226	CAP CER 22UF 6.3V X5R 0805
C10, C11	2	0.01520	0.03040	Yageo	CC0603JRNPO9BN100	CAP CER 10PF 50V NPO 0603
C13	1	0.04760	0.04760	Yageo	CC0603KRX5R6BB475	CAP CER 4.7UF 10V X5R 0603
D4	1	0.10330	0.10330	NXP Semiconductors	BAT54C,215	DIODE ARRAY SCHOTTKY 30V SOT23
L1, FB1	2	0.04760	0.09520	Murata	BLM18AG121SN1D	FERRITE BEAD 120 OHM 0603
R1, R5, R6, R18, R19	5	0.00270	0.01350	Yageo	RC0603JR-074K7L	RES SMD 4.7K OHM 5% 1/10W 0603
R2	1	0.00570	0.00570	Yageo	RC0603FR-0730KL	RES SMD 30K OHM 1% 1/10W 0603
R7	1	0.00440	0.00440	Yageo	RC0603JR-0747KL	RES SMD 47K OHM 5% 1/10W 0603
R17	1	0.00570	0.00570	Yageo	RC0603FR-0733K2L	RES SMD 33.2K OHM 1% 1/10W 0603
R8, R9, R10, R11, R12, R16, R20, R21	8	0.00348	0.02784	Yageo	RC0603FR-0710KL	RES SMD 10K OHM 1% 1/10W 0603
C1, C2, C3, C4, C5, C6, C7, C9, C15, C12	10	0.00715	0.07150	Yageo	CC0603KRX7R9BB104	CAP CER 0.1UF 50V X7R 0603
U4	1		0.00000	Consignment	WS2812B	RGB LED

Sub-Total 14.37524

Production loss 0.71876

Supplier Shipping Cost 0.60000

PARTS TOTAL 15.69400

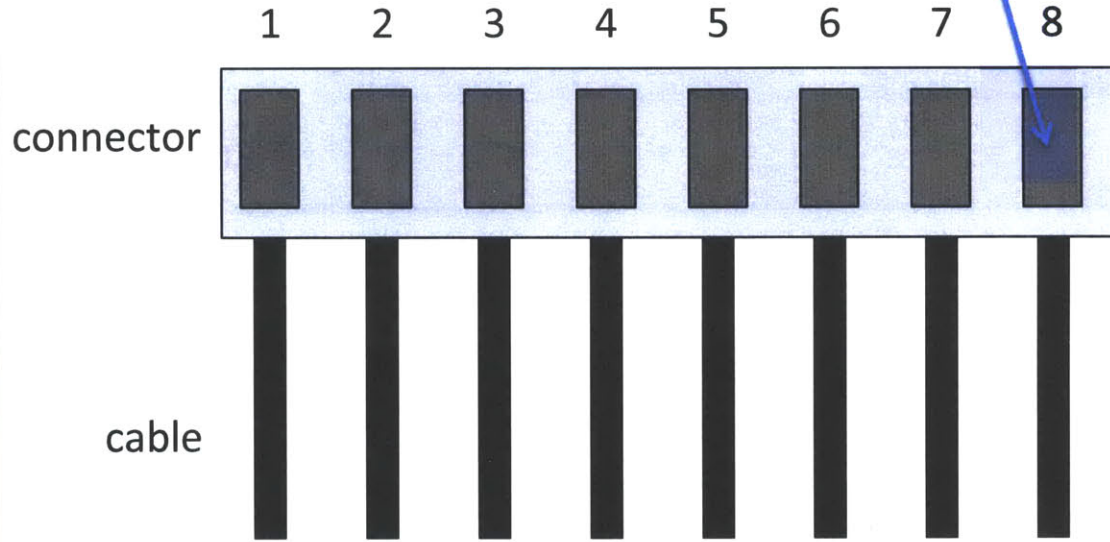


Cable Pinout for Version 2

Plug Assembly

windows facing up (dark squares)

Pinout*	
A ->	B
1 ->	8
2 ->	7
3 ->	4
4 ->	6
5 ->	5
6 ->	3
8 ->	2



* No connection between 7 and 1

Connector: 455-1199-ND

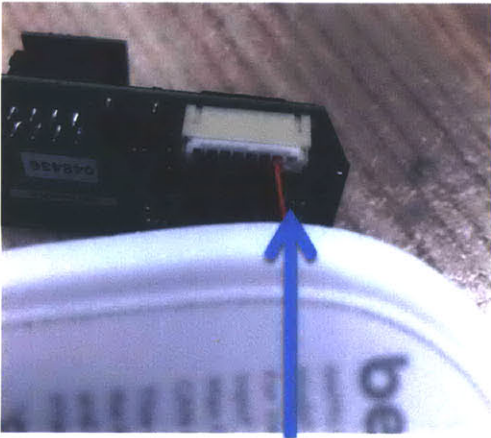
Pin: SZH-002T-P0.5

WEMO Plug Version 1 (original)

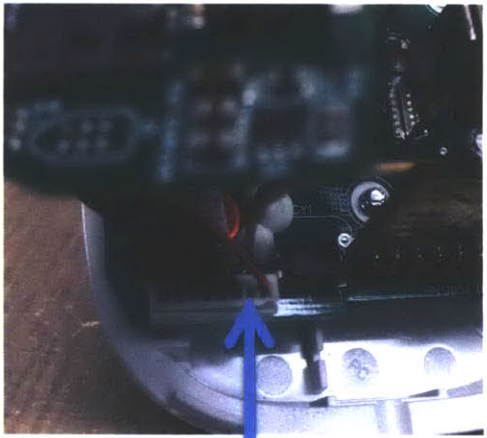


Connect with **existing** cable

WEMO Plug Version 2 (new)



Side **B** to control board



Side **A** to plug PCB

B.2.4 Selected Firmware Files

Listing B-4: firmware/src/wifi.c: Driver for ESP8266. AT+ Command logic for wifi interface

```
Git repository: http://git.wattsworth.net/nilm/wemo-firmware
Filename: firmware/src/wifi.c
Revision: master

1 #include <asf.h>
2 #include <stdio.h>
3 #include "string.h"
4 #include "wifi.h"
5 #include "monitor.h"
6 #include "conf_membag.h"
7 ///////////////////////////////////////////////////////////////////
8 //wifi rx and tx data structures
9 //
10 //resp_buf: uart interrupt fills this buf
11 //      when rx_wait==true && data_tx_status==TX_IDLE
12 //resp_complete_buf: short string we expect wifi module
13 //      to respond with before the timeout
14 uint8_t *resp_buf=NULL;
15 uint32_t resp_buf_idx = 0;
16 char *resp_complete_buf; //expected response of command to ESP8266
17
18 bool rx_wait = false; //we are waiting for response
19 bool rx_complete = false; //flag set by UART int when rx_complete_str matches
20
21 int data_tx_status = TX_IDLE;
22 //index into incoming data buffer (data from another server)
23 //  this is local to wifi module because the core only gets
24 //  the buffer after the full flag is set
25 uint32_t wifi_rx_buf_idx = 0;
26
27 /**declared in header b/c monitor calls this when NILM IP addr changes
28 //int wifi_send_ip(void);
29 int wifi_send_data(int ch, const uint8_t* data, int size);
30 //try to set the baud to 9600 since they come from the factory at 115200
31 int wifi_set_baud(void);
32
33
34 int wifi_init(void){
35     uint32_t BUF_SIZE = MD_BUF_SIZE;
36     uint8_t tmp; //dummy var for flushing UART
37     char *buf;
38     char *tx_buf;
39
40     //allocate static buffers if they are not NULL
41     if(resp_buf==NULL){
42         resp_buf=core_malloc(RESP_BUF_SIZE);
43     }
44     if(resp_complete_buf==NULL){
45         resp_complete_buf=core_malloc(RESP_COMPLETE_BUF_SIZE);
46     }
47     if(wifi_rx_buf==NULL){
48         wifi_rx_buf = core_malloc(WIFI_RX_BUF_SIZE);
49     }
50     //if we are standalone, don't do anything
51     if(wemo_config.standalone){
52         printf("warning: wifi_init called in standalone mode\n");
53         return 0;
54     }
}
```

```

55 //initialize the memory
56 buf = core_malloc(BUF_SIZE);
57 tx_buf = core_malloc(BUF_SIZE);
58
59 //set up the UART
60 static usart_serial_options_t usart_options = {
61     .baudrate = WIFI_UART_BAUDRATE,
62     .charlength = WIFI_UART_CHAR_LENGTH,
63     .paritytype = WIFI_UART_PARITY,
64     .stopbits = WIFI_UART_STOP_BITS
65 };
66 gpio_configure_pin(PIO_PA9_IDX, (PIO_PERIPH_A | PIO_DEFAULT));
67 gpio_configure_pin(PIO_PA10_IDX, (PIO_PERIPH_A | PIO_DEFAULT));
68 pmc_enable_periph_clk(ID_WIFI_UART);
69 sysclk_enable_peripheral_clock(ID_WIFI_UART);
70 usart_serial_init(WIFI_UART,&usart_options);
71 //flush any existing data
72 while(usart_serial_is_rx_ready(WIFI_UART)){
73     usart_serial_getchar(WIFI_UART,&tmp);
74 }
75 // Trigger from timer 0
76 pmc_enable_periph_clk(ID_TC0);
77 tc_init(TC0, 0, // channel 0
78     TC_CMR_TCCLKS_TIMER_CLOCK5 // source clock (CLOCK5 = Slow Clock)
79     | TC_CMR_CPCTRIG // up mode with automatic reset on RC match
80     | TC_CMR_WAVE // waveform mode
81     | TC_CMR_ACPA_CLEAR // RA compare effect: clear
82     | TC_CMR_ACPC_SET // RC compare effect: set
83 );
84 TC0->TC_CHANNEL[0].TC_RA = 0; // doesn't matter
85 TC0->TC_CHANNEL[0].TC_RC = 64000; // sets frequency: 32kHz/32000 = 1 Hz
86 NVIC_ClearPendingIRQ(TC0_IRQn);
87 NVIC_SetPriority(TC0_IRQn,1); //high priority
88 NVIC_EnableIRQ(TC0_IRQn);
89 tc_enable_interrupt(TC0, 0, TC_IER_CPCS);
90 //reset the module
91 if(wifi_send_cmd("AT+RST","ready",buf,BUF_SIZE,1)==0){
92     printf("Error resetting ESP8266\n");
93     //free memory
94     core_free(buf);
95     core_free(tx_buf);
96     //if the baud rate was wrong try to set it to 9600
97     //wifi_set_baud(); **disabled on production firmware**
98     return -1;
99 }
100 //set to mode STA
101 if(wifi_send_cmd("AT+CWMODE=1","OK",buf,BUF_SIZE,1)==0){
102     printf("Error setting ESP8266 mode\n");
103     core_free(buf);
104     core_free(tx_buf);
105     return 0;
106 }
107 //try to join the specified network
108 snprintf(tx_buf,BUF_SIZE,"AT+CWJAP=\"%s\",\"%s\"",
109     wemo_config.wifi_ssid,wemo_config.wifi_pwd);
110 if(wifi_send_cmd(tx_buf,"OK",buf,BUF_SIZE,20)==0){
111     printf("no response to CWJAP\n");
112     //free memory
113     core_free(buf);
114     core_free(tx_buf);
115     return -1;
116 }
117 //make sure the response ends in OK
118 uint8_t len = strlen(buf);
119 if(len<2 || strcmp(&buf[len-2],"OK")!=0){
120     snprintf(tx_buf,BUF_SIZE,"failed to join network [%s]: [%s]\n",
121         wemo_config.wifi_ssid, buf);

```

```

122     printf(tx_buf);
123     core_log(tx_buf);
124     //free memory
125     core_free(buf);
126     core_free(tx_buf);
127     return -1;
128 }
129 //see if we have an IP address
130 wifi_send_cmd("AT+CIFSR","OK",buf,BUF_SIZE,2);
131 if(strstr(buf,"ERROR")==buf){
132     printf("error getting IP address\n");
133     //free the memory
134     core_free(tx_buf);
135     core_free(buf);
136     return -1;
137 }
138 //try to parse the response into an IP address
139 //expect 4 octets but *not* 0.0.0.0
140 int a1,a2,a3,a4;
141 if(!(sscanf(buf,"+CIFSR:STAIP, \"%d.%d.%d.%d\"", &a1,&a2,&a3,&a4)==4 && a1!=0)){
142     printf("error, bad address: %s\n",buf);
143     //free the memory
144     core_free(tx_buf);
145     core_free(buf);
146     return -1;
147 }
148 //save the IP to our config
149 snprintf(buf,BUF_SIZE,"%d.%d.%d.%d",a1,a2,a3,a4);
150 memset(wemo_config.ip_addr,0x0,MAX_CONFIG_LEN);
151 strcpy(wemo_config.ip_addr,buf);
152 //set the mode to multiple connection
153 wifi_send_cmd("AT+CIPMUX=1","OK",buf,BUF_SIZE,2);
154 //start a server on port 1336
155 wifi_send_cmd("AT+CIPSERVER=1,1336","OK",buf,BUF_SIZE,2);
156
157 //if we know the NILM IP address, send it our IP
158 if(strlen(wemo_config.nilm_ip_addr)!=0){
159     if(wifi_send_ip()==TX_ERR_MODULE_RESET){
160         return TX_ERR_MODULE_RESET;
161     }
162 } else {
163     //get the NILM IP address from the manager
164     //once we know the NILM address we send it ours
165     core_get_niln_ip_addr();
166 }
167 //log the event
168 snprintf(buf,BUF_SIZE,"Joined [%s] with IP [%s]",
169     wemo_config.wifi_ssid,wemo_config.ip_addr);
170 printf("\n%s\n",buf);
171 core_log(buf);
172 //free the memory
173 core_free(tx_buf);
174 core_free(buf);
175 return 0;
176 }
177
178 int wifi_send_ip(void){
179     int TX_BUF_SIZE = LG_BUF_SIZE;
180     int PAYLOAD_BUF_SIZE = LG_BUF_SIZE;
181     int r;
182     char *tx_buf,*payload_buf;
183     tx_buf = core_malloc(TX_BUF_SIZE);
184     payload_buf = core_malloc(PAYLOAD_BUF_SIZE);
185     snprintf(payload_buf,PAYLOAD_BUF_SIZE,
186         "{ \"serial_number\": \"%s\", \"ip_addr\": \"%s\" }",
187         wemo_config.serial_number,wemo_config.ip_addr);
188     snprintf(tx_buf,TX_BUF_SIZE,

```



```

189     "POST /config/plugs/update HTTP/1.1\r\n"
190     "User-Agent: WemoPlug\r\n"
191     "Host: NILM\r\nAccept: */*\r\n"
192     "Connection: keep-alive\r\n"
193     "Content-Type: application/json\r\n"
194     "Content-Length: %d\r\n"
195     "\r\n%s",
196     strlen(payload_buf), payload_buf);
197 //send the packet!
198 r = wifi_transmit(wemo_config.nilm_ip_addr, 80, tx_buf);
199 core_free(tx_buf);
200 core_free(payload_buf);
201 return r;
202 }
203
204 int wifi_set_baud(void){
205     core_log("setting ESP8266 baudrate to 9600");
206     //reconfigure UART for 115200
207     static usart_serial_options_t usart_options = {
208         .baudrate = 115200,
209         .charlength = WIFI_UART_CHAR_LENGTH,
210         .paritytype = WIFI_UART_PARITY,
211         .stopbits = WIFI_UART_STOP_BITS
212     };
213     //send baudrate command
214     char* cmd = "AT+CI0BAUD=9600";
215     usart_serial_init(WIFI_UART, &usart_options);
216     usart_serial_write_packet(WIFI_UART, (uint8_t*)cmd, strlen(cmd));
217     //terminate the command
218     usart_serial_putchar(WIFI_UART, '\r');
219     usart_serial_putchar(WIFI_UART, '\n');
220     //wait 1 second
221     tc_start(TC0, 0);
222     rx_wait=true;
223     while(rx_wait);
224     //reconfigure UART for 9600
225     usart_options.baudrate = WIFI_UART_BAUDRATE;
226     usart_serial_init(WIFI_UART, &usart_options);
227     printf("reset baudrate\n");
228 }
229
230 int wifi_transmit(char *url, int port, char *data){
231     int BUF_SIZE = MD_BUF_SIZE;
232     char *cmd;
233     char *buf;
234     int r;
235     //allocate memory
236     cmd = core_malloc(BUF_SIZE);
237     buf = core_malloc(BUF_SIZE);
238     //sometimes the port stays open, so check for both
239     //conditions
240     char *success_str = "4,CONNECT";
241     char *connected_str = "ALREAY CONNECT"; //sic
242     //open a TCP connection on channel 4
243     snprintf(cmd, BUF_SIZE, "AT+CIPSTART=4, \"TCP\", \"%s\", %d",
244             url, port);
245     wifi_send_cmd(cmd, "4,CONNECT", buf, 100, 5);
246     //check if we are able to connect to the NILM
247     if(strstr(buf, "ERROR\r\nUnlink")==buf){
248         printf("can't connect to NILM\n");
249         core_free(cmd);
250         core_free(buf);
251         return TX_BAD_DEST_IP;
252     }
253     //if we are still connected, close and reopen socket
254     if(strstr(buf, connected_str)==buf){
255         wifi_send_cmd("AT+CIPCLOSE=4", "Unlink", buf, 100, 1);

```

```

256     //now try again
257     wifi_send_cmd(cmd,"Linked",buf,100,2);
258 }
259 //check for successful link
260 if(strstr(buf,succes_str)!=buf){
261     printf("error, setting up link\n");
262     core_free(cmd);
263     core_free(buf);
264     return TX_ERROR;
265 }
266 //send the data
267 if((r=wifi_send_txt(4,data))!=0){
268     printf("error transmitting data: %d\n",data_tx_status);
269     core_free(cmd);
270     core_free(buf);
271     return r;
272 }
273 //connection is closed *after* we receive the server's response
274 //this is processed by the core and we discard the response
275 //wifi_send_cmd("AT+CIPCLOSE=4","Unlink",buf,100,1);
276 core_free(cmd);
277 core_free(buf);
278 return r; //success!
279 }
280
281 /*** These are the accessor functions to transmit data ***/
282 int wifi_send_txt(int ch, const char* data){
283     return wifi_send_raw(ch,(uint8_t*)data,strlen(data));
284 }
285
286 int wifi_send_raw(int ch, const uint8_t* data, int size){
287     int BUFFER_SIZE=1000;
288     int i=0;
289     int r;
290     char *tx_buf;
291     tx_buf = core_malloc(MD_BUF_SIZE);
292
293     for(i=0;i<size;i+=BUFFER_SIZE){
294         if(i+BUFFER_SIZE<size)
295             r=wifi_send_data(ch,&data[i],BUFFER_SIZE);
296         else
297             r=wifi_send_data(ch,&data[i],size-i);
298         if(r!=TX_SUCCESS){ //exit early
299             core_free(tx_buf);
300             return r; //fail!
301         }
302     }
303     core_free(tx_buf);
304     return r;
305 }
306
307 /*** Private method that actually sends data ***/
308 int wifi_send_data(int ch, const uint8_t* data, int size){
309     int cmd_buf_size = MD_BUF_SIZE;
310     char *cmd;
311     int timeout = 7; //wait 7 seconds to transmit the data
312     //allocate memory
313     cmd = core_malloc(cmd_buf_size);
314     snprintf(cmd,cmd_buf_size,"AT+CIPSEND=%d,%d\r\n",ch,size);
315     data_tx_status=TX_PENDING;
316     rx_wait=true;
317     resp_buf_idx = 0;
318     memset(resp_buf,0x0,RESP_BUF_SIZE);
319     memset(wifi_rx_buf,0x0,WIFI_RX_BUF_SIZE); //to make debugging easier
320     wifi_rx_buf_idx=0;
321     usart_serial_write_packet(WIFI_UART,(uint8_t*)cmd,strlen(cmd));
322     delay_ms(250); //wait for module to be ready to accept data

```



```

323 usart_serial_write_packet(WIFI_UART,(uint8_t*)data,size);
324 //now wait for the data to be sent
325 while(timeout>0){
326     //start the timer
327     tc_start(TC0, 0);
328     //when timer expires, return what we have in the buffer
329     rx_wait=true; //reset the wait flag
330     while(rx_wait && data_tx_status!=TX_SUCCESS);
331     tc_stop(TC0,0);
332     //the success flag is set *before* we receive the server's response
333     //core_process_wifi_data receives the response but discards it
334     if(data_tx_status==TX_SUCCESS){
335         data_tx_status=TX_IDLE;
336         rx_wait = false;
337         //free memory
338         core_free(cmd);
339         return TX_SUCCESS;
340     }
341     timeout--;
342 }
343 //check if this is a timeout error
344 if(strlen((char*)resp_buf)==0){
345     printf("timeout error\n");
346     core_log("timeout error");
347     data_tx_status = TX_TIMEOUT;
348 }
349 //an error ocured, see if it is due to a module reset
350 else if(strcmp((char*)resp_buf,"\r\nready\r\n")==0){
351     //module reset itself!!!
352     printf("detected module reset\n");
353     core_log("module reset");
354     data_tx_status = TX_ERR_MODULE_RESET;
355 } else {
356     data_tx_status=TX_ERROR;
357     core_log("TX error:");
358     core_log((char*)resp_buf);
359     data_tx_status = TX_ERROR;
360 }
361 //free memory
362 core_free(cmd);
363 return data_tx_status;
364 }
365
366 int wifi_send_cmd(const char* cmd, const char* resp_complete,
367                 char* resp, uint32_t maxlen, int timeout){
368     resp_buf_idx = 0;
369     uint32_t rx_start, rx_end;
370     //clear out the response buffer
371     memset(resp,0x0,maxlen);
372     memset(resp_buf,0x0,RESP_BUF_SIZE);
373     //setup the rx_complete buffer so we know when the command is finished
374     if(strlen(resp_complete)>RESP_COMPLETE_BUF_SIZE-3){
375         printf("resp_complete, too long exiting\n");
376         return -1;
377     }
378     strcpy(resp_complete_buf,resp_complete);
379     strcat(resp_complete_buf,"\r\n");
380     //enable RX interrupts
381     usart_enable_interrupt(WIFI_UART, US_IER_RXRDY);
382     NVIC_SetPriority(WIFI_UART_IRQ,2);
383     NVIC_EnableIRQ(WIFI_UART_IRQ);
384     //write the command
385     rx_wait=true; //we want this data returned in resp_buf
386     rx_complete =false; //reset the early complete flag
387     usart_serial_write_packet(WIFI_UART,(uint8_t*)cmd,strlen(cmd));
388     //terminate the command
389     usart_serial_putchar(WIFI_UART,'\r');

```

```

390 usart_serial_putchar(WIFI_UART, '\n');
391 //wait for [timeout] seconds
392 while(timeout>0){
393     //start the timer
394     tc_start(TC0, 0);
395     //when timer expires, return what we have in the buffer
396     rx_wait=true; //reset the wait flag
397     while(rx_wait);
398     tc_stop(TC0,0);
399     if(rx_complete) //if the uart interrupt signals rx is complete
400         break;
401     timeout--;
402 }
403 //now null terminate the response
404 resp_buf[resp_buf_idx]=0x0;
405 //remove any ECHO
406 if(strstr((char*)resp_buf,cmd)!=(char*)resp_buf){
407     printf("bad echo: %s\n", resp_buf);
408     return 0;
409 }
410 rx_start = strlen(cmd);
411 //remove leading whitespace
412 while(resp_buf[rx_start]=='\r' || resp_buf[rx_start]=='\n')
413     rx_start++;
414 //remove trailing whitespace
415 rx_end = strlen((char*)resp_buf)-1;
416 while(resp_buf[rx_end]=='\r' || resp_buf[rx_end]=='\n')
417     rx_end--;
418 //make sure we have a response
419 if(rx_end<=rx_start){
420     printf("no response by timeout\n");
421     return 0;
422 }
423 //copy the data to the response buffer
424 if((rx_end-rx_start+1)>maxlen){
425     memcpy(resp,&resp_buf[rx_start],maxlen-1);
426     resp[maxlen-1]=0x0;
427     printf((char*)resp_buf);
428     printf("truncated output!\n");
429 } else{
430     memcpy(resp,&resp_buf[rx_start],rx_end-rx_start+1);
431     //null terminate the response buffer
432     resp[rx_end-rx_start+1]=0x0;
433 }
434 return rx_end-rx_start;
435 }
436
437 //Priority 0 (highest)
438 ISR(TC0_Handler)
439 {
440     rx_wait = false;
441     //clear the interrupt so we don't get stuck here
442     tc_get_status(TC0,0);
443 }
444
445 //Priority 1 (middle)
446 ISR(UART0_Handler)
447 {
448     uint8_t tmp, i;
449
450     //state machine vars for handling RX'd data
451     static int rx_bytes_recvd = 0;
452     static bool rx_in_prog = false;
453     static int rx_bytes_expected = 0;
454     static int rx_chan = 0;
455     char *action_buf;
456     int ACTION_BUF_SIZE = MD_BUF_SIZE;

```

```

457
458 usart_serial_getchar(WIFI_UART,&tmp);
459 //if debug level is high enough, print the char
460 if(wemo_config.debug_level>=DEBUG_INFO)
461     core_putc(NULL,tmp);
462 //check whether this is a command response or
463 //new data from the web (unsolicited response)
464 if(rx_wait && data_tx_status!=TX_PENDING){
465     if(resp_buf_idx>=RESP_BUF_SIZE){
466         printf("error!\n");
467         return; //ERROR!!!!
468     }
469     resp_buf[resp_buf_idx++]=(char)tmp;
470     //check for completion_str to indicate completion of command
471     //this is just a speed up, we still timeout regardless
472     //of whether we find this string
473     if(resp_buf_idx>4){
474         if(strstr((char*)resp_buf,resp_complete_buf)==
475            (char*)&resp_buf[resp_buf_idx-strlen(resp_complete_buf)]){
476             rx_complete = true; //early completion!
477             rx_wait = false;
478         }
479     }
480 } else if(rx_wait && data_tx_status==TX_PENDING){
481     //we are transmitting, no need to capture response,
482     //after transmission we receive SEND OK\r\n, responses
483     //are captured by wifi_rx_buf and processed by the core
484     //just wait for SEND OK\r\n, use a 9 char circular buffer
485     if(resp_buf_idx>8){
486         for(i=0;i<8;i++)
487             resp_buf[i]=resp_buf[i+1];
488         resp_buf[8]=tmp;
489         if(strstr((char*)resp_buf,"SEND OK\r\n")== (char*)resp_buf){
490             data_tx_status=TX_SUCCESS;
491         }
492     } else{
493         resp_buf[resp_buf_idx++]=tmp;
494     }
495 }else { //data_tx_status == TX_IDLE
496     //this is unsolicited data, incoming from Internet,
497     //process into wifi_rx_buf and pass off to the the core
498     //when the reception is complete
499     if(wifi_rx_buf_full){
500         printf("error, wifi_rx_buf must be processed by main loop!\n");
501         return; //ERROR!!!
502     }
503     if(wifi_rx_buf_idx>=WIFI_RX_BUF_SIZE){
504         printf("too much data, wifi_rx_buf full!\n");
505         //empty the buffer
506         wifi_rx_buf_idx = 0;
507         return; //ERROR!!!!
508     }
509     //store the data in the rx buffer
510     wifi_rx_buf[wifi_rx_buf_idx++]=(char)tmp;
511     //if a reception is in progress...
512     if(rx_in_prog){
513         rx_bytes_recvd++;
514         if(rx_bytes_recvd==rx_bytes_expected){
515             //data is ready for processing, set flag so main loop
516             //runs core_process_wifi_data
517             wifi_rx_buf_full=true;
518             wifi_rx_buf_idx=0;
519             rx_in_prog = false;
520             return;
521         }
522     }
523     //otherwise check for control sequences....

```



```

524     if(wifi_rx_buf_idx>=6 && !rx_in_prog){
525         //check for incoming data
526         char dummy;
527         if(sscanf(wifi_rx_buf, "\r\n+IPD,%d,%d:%c",
528             &rx_chan,&rx_bytes_expected,&dummy)==3){
529             rx_in_prog = true;
530             rx_bytes_recvd = 1; //reset the RX counter so we know when we have all the data
531             return;
532         }
533         //check for link and unlink
534         action_buf = core_malloc(ACTION_BUF_SIZE);
535         if(sscanf(wifi_rx_buf, "%d,%s\r\n", &rx_chan, action_buf)==2){
536             if(strcmp(action_buf, "CONNECT")==0){
537                 core_wifi_link(rx_chan);
538                 wifi_rx_buf_idx=0;
539             }
540             if(strcmp(action_buf, "CLOSED")==0){
541                 core_wifi_unlink(rx_chan);
542                 wifi_rx_buf_idx=0;
543             }
544         }
545         //if wifi_rx_buf is longer than 20 chars we must be out of sync,
546         //look for \r\n sequence and flush the buffer
547         if(wifi_rx_buf_idx>20){
548             if(strcmp(&wifi_rx_buf[wifi_rx_buf_idx-2], "\r\n")==0){
549                 wifi_rx_buf_idx = 0;
550                 printf("flushed wifi_rx_buf, out of sync\n");
551             }
552         }
553         core_free(action_buf);
554         return;
555     }
556 }
557 }
558
559
560 /**NOTE: error when startup misses a character so wifi_rx_buf is out of sync**

```

Listing B-5: firmware/src/wemo.c: Driver for WEMO hardware. Solid state meter and relay logic

Git repository: <http://git.wattsworth.net/nilm/wemo-firmware>
 Filename: firmware/src/wemo.c
 Revision: master

```

1  #include <asf.h>
2  #include "string.h"
3  #include "wemo.h"
4  #include "monitor.h"
5
6  /*****
7  -----Theory of Operation-----
8  The server listens for relay commands over TCP
9  and pushes out updates from the power meter over TCP
10 -----Relay Commands-----
11 relay_on: turn the relay on
12 relay_off: turn the relay off
13 ----Power Meter-----
14 Read the wemo power chip every second. check if data in the power struct
15 is valid, if so send it //otherwise flush it and start listening to
16 the UART. The UART waits for the sync byte 0xAE then reads 29 more
17 bytes computes the checksum and parses the data into the power
18 struct if the checksum is good and everything is stored the valid

```

```

19  flag is set
20  *****/
21
22
23  power_sample wemo_sample;
24  uint8_t wemo_buffer [30];
25  //take 30 byte buffer from WEMO and fill power sample struct
26  uint8_t process_sample(uint8_t *buffer);
27
28  void wemo_init(void){
29      //allocate memory for the server buffer
30      //set up the power meter UART
31      static usart_serial_options_t usart_options = {
32          .baudrate = WEMO_UART_BAUDRATE,
33          .charlength = WEMO_UART_CHAR_LENGTH,
34          .paritytype = WEMO_UART_PARITY,
35          .stopbits = WEMO_UART_STOP_BITS
36      };
37      gpio_configure_pin(PIO_PB2_IDX, (PIO_PERIPH_A | PIO_DEFAULT));
38      pmc_enable_periph_clk(ID_WEMO_UART);
39      sysclk_enable_peripheral_clock(ID_WEMO_UART);
40      usart_serial_init(WEMO_UART,&usart_options);
41      NVIC_SetPriority(WEMO_UART_IRQ,4); //lowest priority
42      NVIC_EnableIRQ(WEMO_UART_IRQ);
43  };
44
45
46  uint8_t process_sample(uint8_t *buffer){
47      //process 30 byte data packet buffer
48      uint8_t checksum = 0;
49      uint8_t bytes[3];
50      int32_t vals[9];
51      int i;
52      //1.) check for header and length
53      if(buffer[0]!=0xAE || buffer[1]!=0x1E){
54          return false;
55      }
56      //2.) compute checksum
57      for(i=0;i<29;i++){
58          checksum += buffer[i];
59      }
60      checksum = (~checksum)+1;
61      if(checksum!=buffer[29]){
62          core_log("bad checksum");
63          return false;
64      }
65
66      //3.) Parse raw data into values
67      // Data is 3 byte signed LSB
68      for(i=0;i<9;i++){
69          bytes[0] = buffer[3*i+2];
70          bytes[1] = buffer[3*i+3];
71          bytes[2] = buffer[3*i+4];
72          vals[i] = bytes[0] | bytes[1]<<8 | bytes[2]<<16;
73          if((bytes[2]&0x80)==0x80){
74              vals[i] |= 0xFF << 24; //sign extend top byte
75          }
76      }
77      //4.) Populate the power struct
78      wemo_sample.vrms = vals[2];
79      wemo_sample.irms = vals[3];
80      wemo_sample.watts= vals[4];
81      wemo_sample.pavg = vals[5];
82      wemo_sample.pf = vals[6];
83      wemo_sample.freq = vals[7];
84      wemo_sample.kwh = vals[8];
85      //5.) Set the valid flag

```



```

86  wemo_sample.valid = true;
87  //all done
88  return true;
89  };
90
91  void wemo_read_power(void){
92  //start listening to the WEMO
93  wemo_sample.valid=false;
94  usart_enable_interrupt(WEMO_UART, US_IER_RXRDY);
95  }
96
97  ISR(UART1_Handler)
98  {
99  uint8_t tmp;
100  static uint8_t buf[30]; //30 byte packet
101  static uint8_t buf_idx=0;
102  usart_serial_getchar(WEMO_UART,&tmp);
103  switch(buf_idx){
104  case 0: //search for sync byte
105  if(tmp==0xAE){
106  //found sync byte, start capturing the packet
107  buf[buf_idx++]=tmp;
108  }
109  break;
110  case 30: //sample is full, read checksum and return data
111  if(process_sample(buf)){
112  //success, stop listening to the UART
113  usart_disable_interrupt(WEMO_UART, US_IER_RXRDY);
114  //reset the index
115  buf_idx=0;
116  } else { //failure, look for the next packet
117  buf_idx=0;
118  }
119  break;
120  case 1: //make sure the packet length is valid
121  if(tmp!=0x1E){
122  buf_idx = 0;
123  break;
124  } //else, add to buf
125  default: //reading packet
126  buf[buf_idx++]=tmp;
127  }
128  }

```

B.3 NILM Software Stack

The NILM software stack can run on a wide variety of hardware platforms from commodity desktop and laptop machines to low cost embedded single board computers (SBC). There are two main variants of the software stack: standalone and embedded. The standalone suite is designed for relatively high power desktop or laptop machines with a keyboard, mouse and monitor. This configuration provides a local distribution of NILM Manager for data visualization and control which means there is no dependency on an Internet connection.

The embedded version is a stripped down software stack optimized to run on

resource constrained embedded systems. This variant does not have a local distribution of NILM Manager and must have a network connection to a NILM Manager instance in order to run. Embedded systems usually do not have input/output devices like a mouse, keyboard, or monitor, so this stack uses a locally hosted web page for configuration. When the system is in configuration mode, users can connect to the “Wattsworth Config” ssid and visit www.wattsworth.net to view and edit the system settings. The flex sensor LED indicates the state of the system. The colors are as follows:

Solid Green Normal operation: System is in run mode, and securely connected to a NILM Manager server

Blinking Red Error: System is in run mode but has a configuration error

Blinking Orange Busy: System is booting or switching between modes

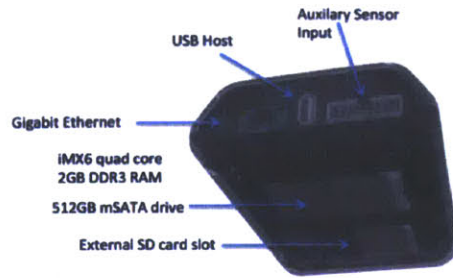
Solid Blue Configuration mode. Connect to “Wattsworth Config” WiFi network to manage the system.

The flex sensor button is used for basic input tasks. When the system is in run mode, press the button once to transition to configuration mode. Press and hold the button for three seconds to shut down the system. When the system is shutting down, the LED will blink red quickly. When the LED turns off it is safe to unplug the system.

The standalone software stack has been verified on Ubuntu 14 LTS. The core data capture, analysis and storage software has been verified on Ubuntu 15. Currently the NILM Manager software only runs on Ubuntu 14. The embedded stack is tightly coupled to the hardware platform and has been deployed on a custom built single board computer as well as the Raspberry Pi 2 (Figure B-9). The custom single board computer was designed as a reference embedded NILM, but as commercial single board computers have improved this has reduced the need for a custom design. The full schematics and bill of materials for the custom NILM Board are included at the end of this appendix for completeness.



(a) Raspberry Pi 2



(b) NILM Board (Appendix B.3.3)

Figure B-9: Hardware platforms supported by the NILM embedded software stack

B.3.1 Data Capture

The NILM software stack is primarily designed to capture and store data collected by contact or non-contact (flex sensor) power meters. Both the standalone and embedded stacks implement a similar data capture service illustrated in Figure B-10. The source code for all data capture processes can be found in `/opt/nilmcapture`. All file names listed in this section are relative to this folder. When the system boots, the OS spawns a `supervisor` process (`nilmcapture/supervisor.py`). The `supervisor` loads the system configuration from the `meters.yml` file, and spawns a `capture` process (`nilmcapture/captured.py`) for each meter that is present and calibrated. If any `capture` process stops due to an error, the `supervisor` tries to restart it. If all of the `capture` processes fail, the `supervisor` assumes the database is corrupt and runs `nilmdb-fsck` before restarting the `capture` processes.

B.3.1.1 Standalone Capture

The `capture` processes differ between the standalone and embedded implementations. The standalone software supports both contact and non-contact sensors and also allows storing more types of data than the embedded implementation. This section covers the standalone capture and the following section covers the embedded capture. Figure B-10 shows the standalone capture implementation. The capture process first

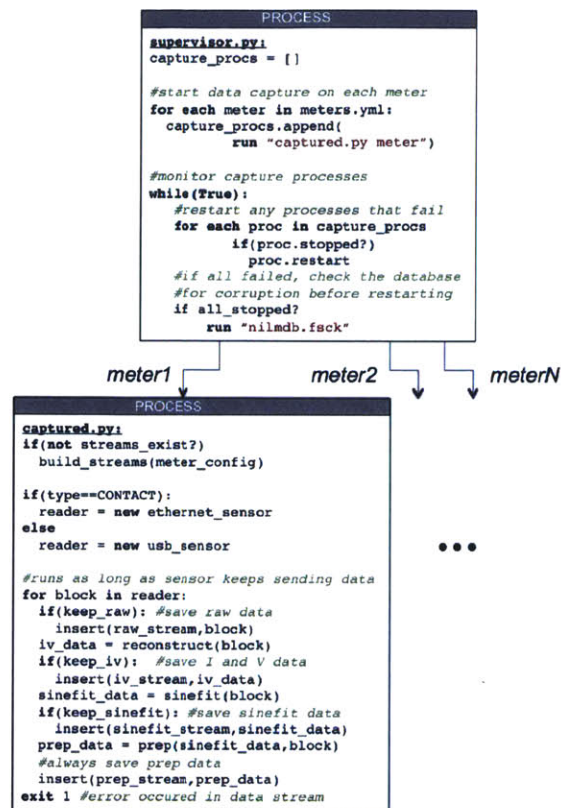


Figure B-10: NILM data capture. The supervisor process manages a capture process for each meter on the NILM

creates a data reader object. The class of this reader depends on whether the meter is contact (`ethernet_sensor`) or non-contact (`usb_sensor`). The capture process starts the reader as a new process and runs in a loop requesting blocks of data from the reader as it becomes available. Each block runs through several data processing stages. The raw (ADC counts) data is first reconstructed to current (amps) and voltage (volts). For a contact sensor the conversion values are specified in `meters.yml`, and for a non-contact sensor these values are calculated during calibration. The `reconstruct` function is implemented in `nilmcapture/reconstructor.py`. The data is then passed through `sinefit (/opt/nilmtools/nilmtools/sinefit.py)` to detect the line frequency and zero crossings. Both the reconstructed data and the `sinefit` data are then passed to `prep`. `Prep (nilmcapture/auto_prepper.py)` is implemented as a stateful object instead of a NILM filter to facilitate streaming decimation. The algorithm is identical to the NILM filter version found in `/opt/nilmtools/nilmtools/prepare.py`.

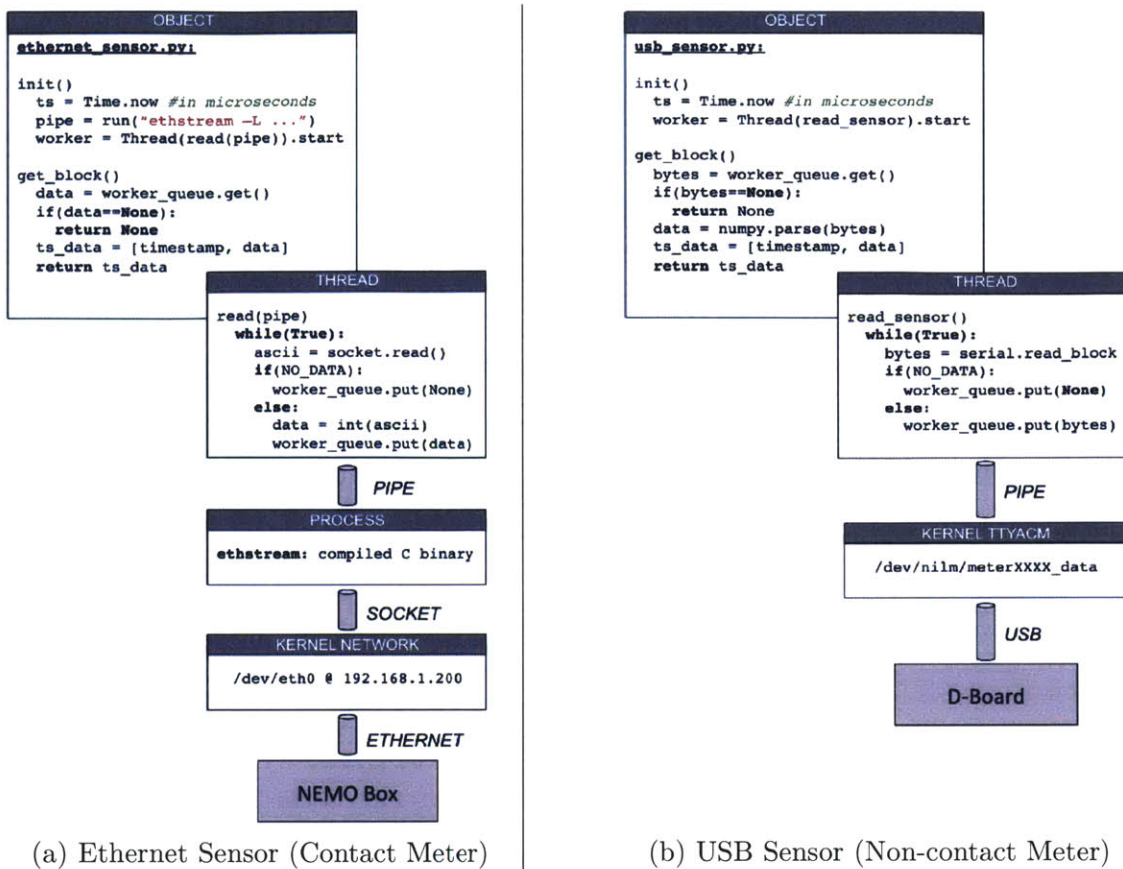


Figure B-11: Pseudo code for sensor reader processes

The result of each data processing step can be optionally inserted into the database depending on the settings in `meters.yml`.

On average, the capture process consumes data faster than it arrives (a requirement for realtime processing), but it only polls for new data once it has finished processing a block. Depending on the system architecture the kernel may not have sufficient internal buffers to store incoming sensor data while the capture process is working on a block. To prevent a buffer overflow in the kernel, `capture` relies on a `reader` object to continuously poll the kernel for sensor data. The `reader` has sufficient buffer space to hold the data until `capture` is ready for it. The reader implementations are discussed below.

Contact Sensor Figure B-11a illustrates the reader structure for contact sensors. The contact sensors use a LabJack UE9 data acquisition board. Following the diagram

from the bottom up, data acquisition board connects to the NILM via Ethernet. The kernel's network drivers expose the connection as a device node to user space. A pre-compiled binary, `ethstream`, connects to the board using a socket to the device node and streams data from it. The `reader` spawns a thread that continuously polls `ethstream` for new data. This data is accumulated in a thread-safe queue. `capture` retrieves data from the `reader` by calling the `get_block` function which dequeues a block of data. If a sensor error has occurred the `reader` returns a `None` element to `capture` which causes it to shut down. The `supervisor` detects this and restarts the `capture` process which in turn re-initializes the sensor to clear the error. The only error that is likely to occur is a buffer overrun which happens when the `reader` thread is not polling the sensor frequently enough. This indicates that the system has too many processes running. If this happens frequently, reduce the number of processes running (eg NILM filters or analyzers), or switch to a faster hardware platform.

Non-contact Sensor Figure B-11b illustrates the reader structure for non-contact sensors. The data flow is similar to the contact sensor reader described above. The main difference is that the non-contact sensors connect to the NILM by USB instead of Ethernet. Like Ethernet, the kernel drivers expose the connection as a device node to user space. However instead of a network connection, the device is a serial line (TTYACM). The `reader` polling thread connects to the device node using the `pySerial` module included in the default Python installation. The rest of the data flow is identical to the contact sensor.

B.3.1.2 Embedded Capture

The data capture process is structured differently in the embedded software stack because embedded systems do not have the processing power to run a full sinefit and prep toolchain in Python. The embedded stack uses a similar `supervisor` but has a different `capture` process as illustrated in Figure B-12. A separate compiled binary written by David Lawrence runs the prep algorithm directly on the sensor data. This “s-prep” data is fed into the `capture` process by block. Since the prep transform is

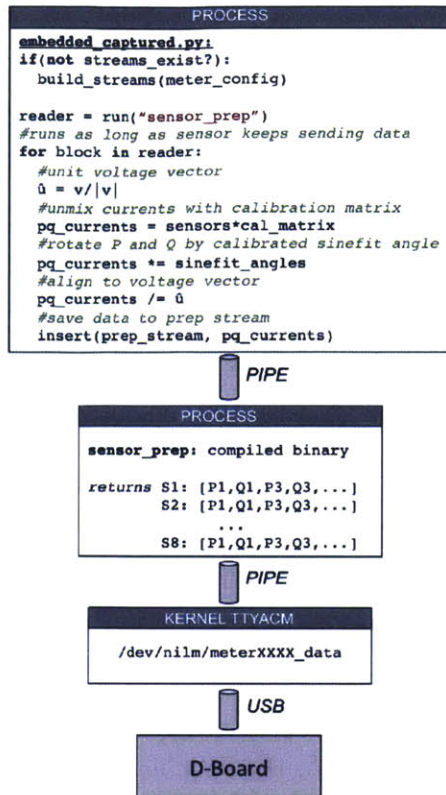


Figure B-12: Embedded data capture

linear, the calibration matrix can be applied directly to s-prep with the same result as if it was applied to the time domain sensor data. The advantage of applying the calibration to s-prep is that this stream arrives at a much lower data rate than the time domain data (60 Hz vs 3kHz). Two rotations are then applied. The ratio of P and Q for each phase are corrected by the sinefit rotation angle computed during calibration, and the ratio of P and Q is aligned to the electric field sensor. This pair of rotations is equivalent to running sinefit.

The prep output is stored in the database. Unlike the standalone software stack, there is no option to store intermediate data streams like raw or sinefit as these are never directly computed.

B.3.2 Embedded Management

The embedded stack runs a global management daemon which controls the system operation. This daemon is illustrated in Figure B-13. When the system boots this daemon is started automatically (in the standalone stack the `supervisor` is started automatically).

Configuration The daemon first checks to see if the system has been configured.

If not, it switches to configuration mode and starts a WiFi access point at “Wattsworth Config”. The user connects to the access point and configures the system with information about the wireless network to use (SSID and password), a name and description of the installation, and an e-mail to associate with the NILM owner. Once this information is entered, the daemon restarts the system.

Network Setup If the daemon detects that the configuration values are present and valid, it attempts to connect to the specified network. If this is unsuccessful, it sets the flex sensor LED to blinking red and waits for the user to press the flex sensor button to enter to configuration mode.

Registration If the network connection is successful, the daemon sets up an encrypted virtual private network (VPN) with the NILM Manager server. The keys for this connection are pre-installed on the system. Once it has joined the VPN, the daemon checks to see if the NILM has been registered with the manager. If this is the first time the NILM has connected to the manager, it registers by sending the e-mail address of the NILM owner as entered by the user during system configuration. The manager then associates the NILM with the user’s account. If the e-mail does not belong to an existing account, the manager sends the user an e-mail with instructions how to create an account and claim the NILM.

Calibration After connecting to the NILM Manager server, the daemon checks if its meters have been calibrated. If not, it starts a calibration server instance and waits for the user to connect the NILM Manager to begin the calibration

process. If the meters are calibrated, the daemon starts the **supervisor** process described in Appendix B.3.1.

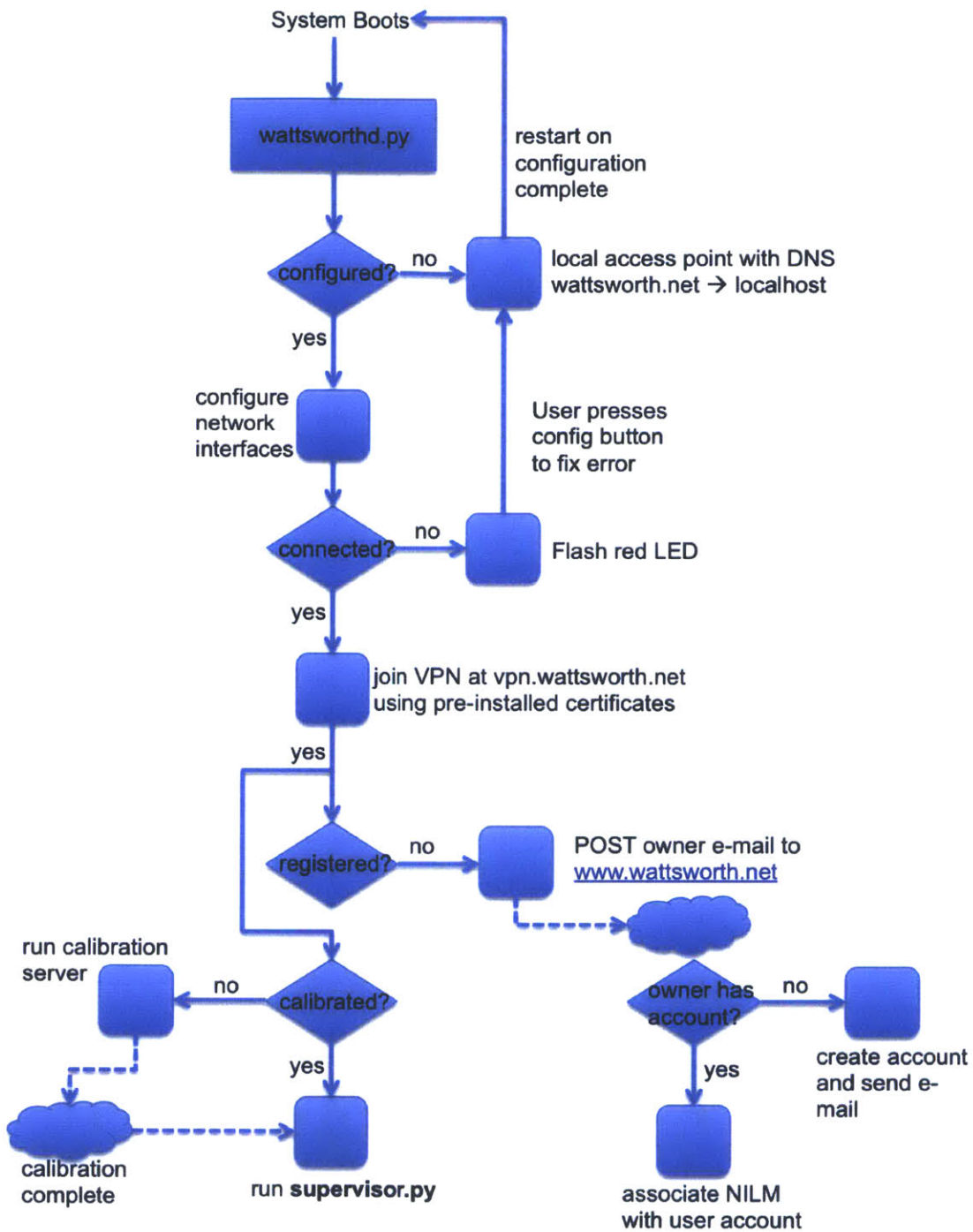


Figure B-13: Embedded system management

NILM Board

Variant: [No Variations]

4/22/2016
V112

DRAFT

327

Page	Index	Page	Index	Page	Index	Page	Index
1	COVER PAGE	11	MISC MODULES (CONT)	21	31
2	BLOCK DIAGRAM	12	REVISION HISTORY	22	32
3	iMX6 SOM	13	23	33
4	iMX6 SOM (continued)	14	24	34
5	STORAGE	15	25	35
6	NETWORK & USB	16	26	36
7	ANALOG and DEBUG UART	17	27	37
8	MICRO and SDRAM	18	28	38
9	STORAGE	19	29	39
10	MISC MODULES	20	30	40

DESIGN CONSIDERATIONS

DESIGN NOTE
Example text for informational design notes.

DESIGN NOTE:
Example text for critical design notes.

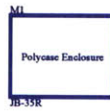
DESIGN NOTE
Example text for cautionary design notes.

LAYOUT NOTE:
Example text for critical layout guidelines.

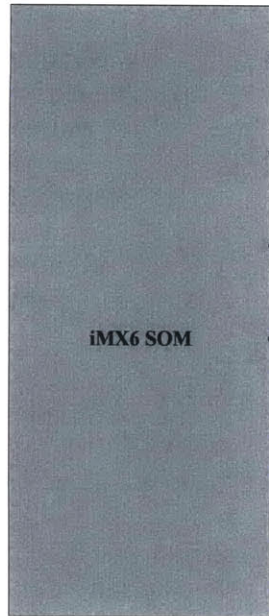
Waltworth		CONFIDENTIAL. Do not distribute.	
Title: NILM Board		Variant: [No Variations]	
Page Contents: [01] - COVER PAGE.SohDoo		Checked by:	
Size:	DWG NO:	Revision:	V112
Date:	4/22/2016	Sheet:	1 of 11

NILM Board

Page 4



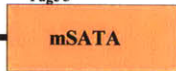
Page 3



Page 5



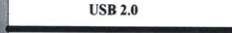
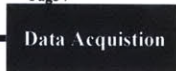
Page 5



Page 6



Page 7

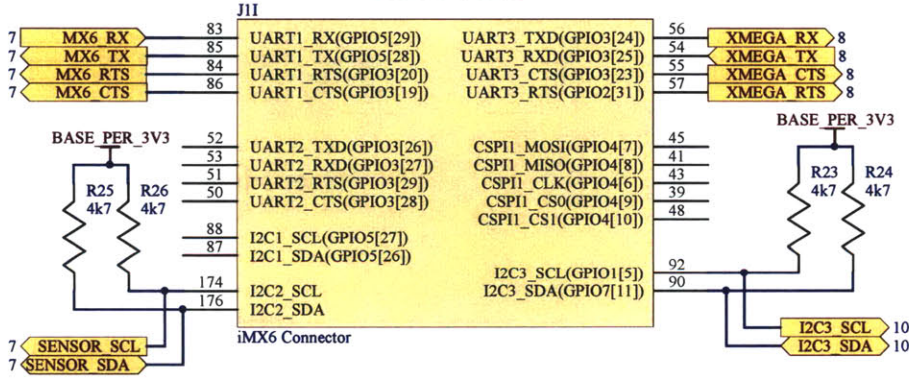


328

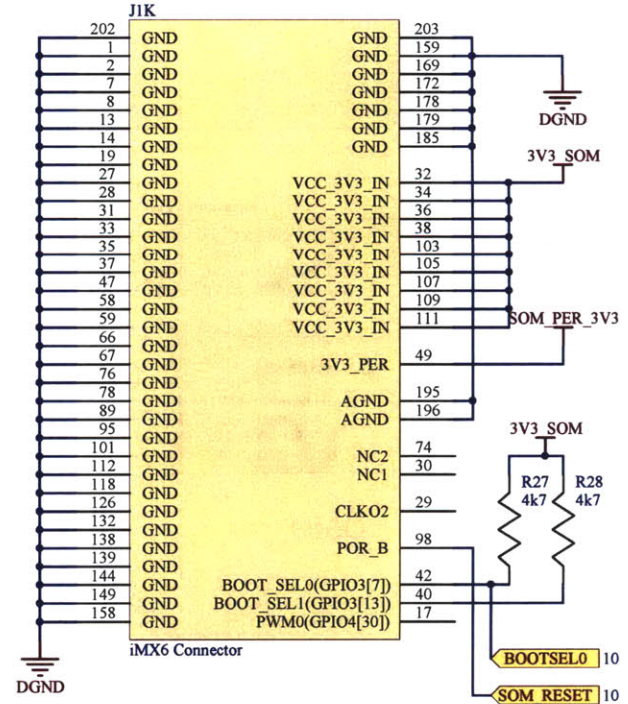
Waltworth		CONFIDENTIAL. Do not distribute.	
Title: NILM Board		Variant: [No Variations]	
Page Contents: [02] - BLOCK DIAGRAM.SchDoc		Checked by:	
Size:	DWG NO:	Revision: V112	
Date:	4/22/2016	Sheet 2 of 11	

iMX6 SOM

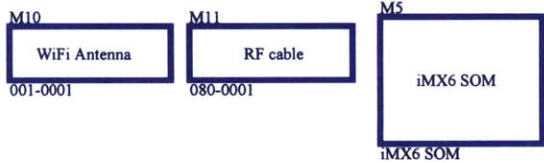
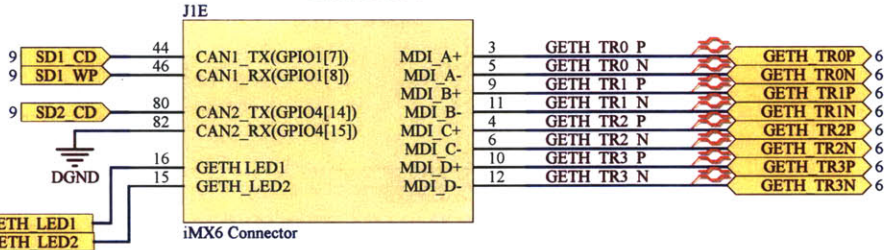
UART / I2C



Power



Ethernet

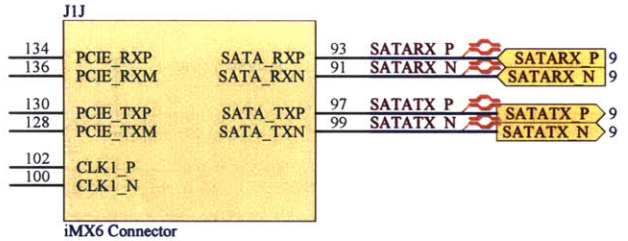


Title		
Size	Number	Revision
A		
Date:	4/22/2016	Sheet of
File:	C:\Users\...\[03]-SOM.SchDoc	Drawn By:

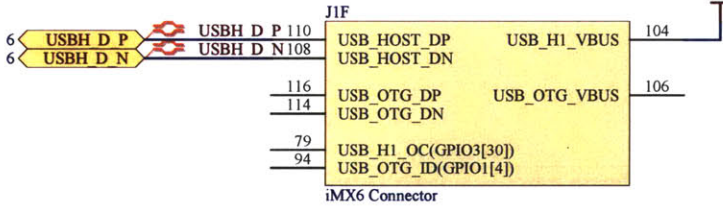
329

iMX6 SOM

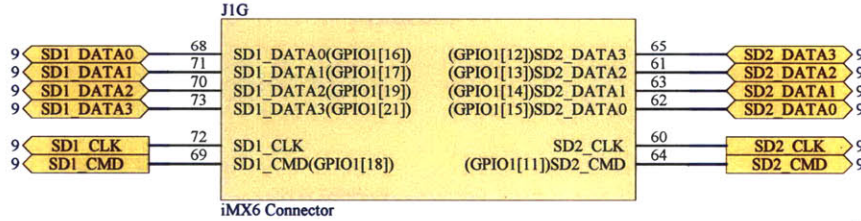
SATA



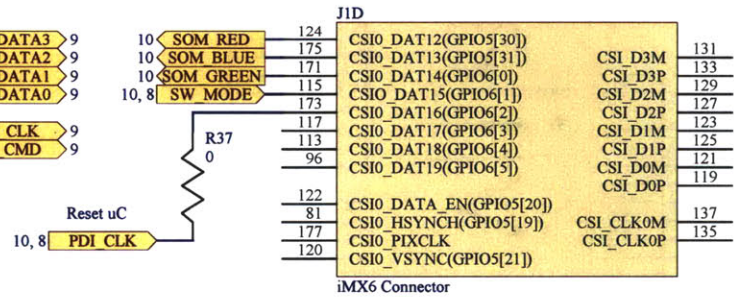
USB



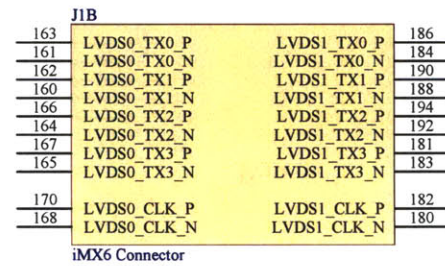
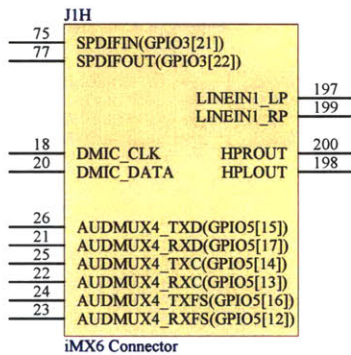
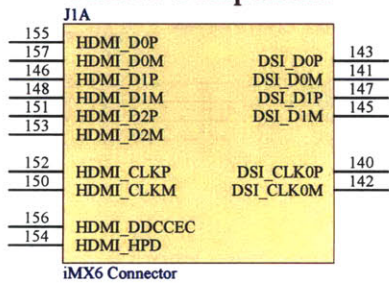
SD Cards



GPIO



Unused Components



1

2

3

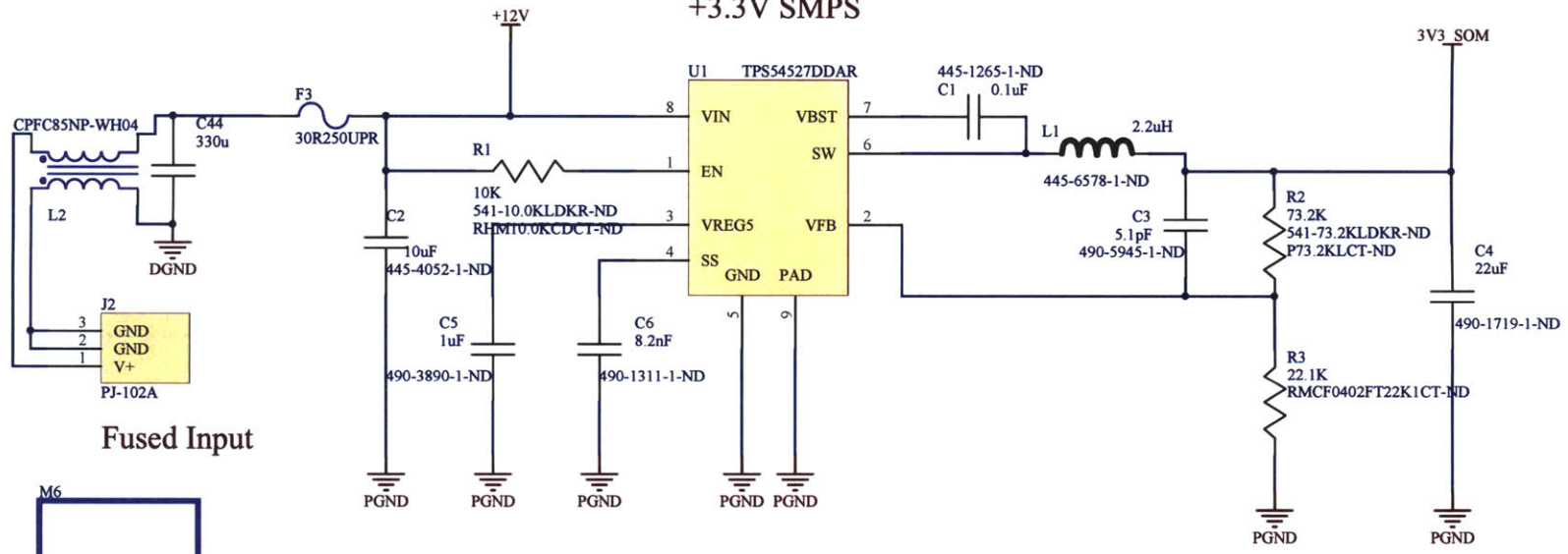
4

330

Title		
Size	Number	Revision
A		
Date:	4/22/2016	Sheet of
File:	C:\Users\...[04]-SOM2_SchDoc	Drawn By:

Power Supplies

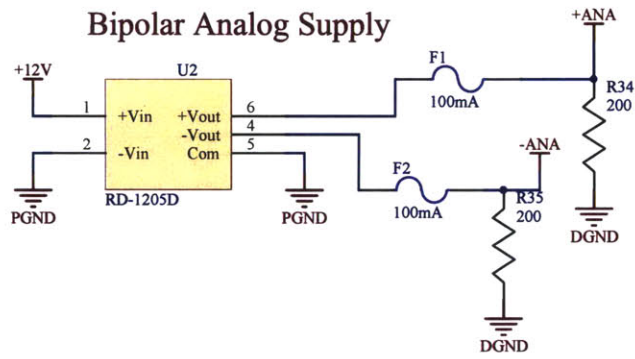
+3.3V SMPS



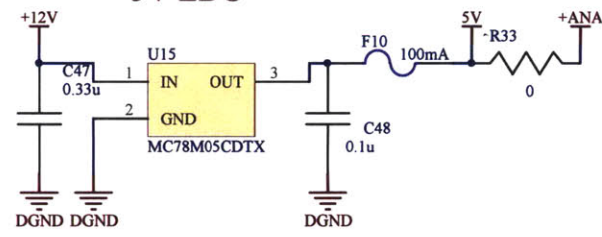
Fused Input



Bipolar Analog Supply



+5V LDO



Bypass Bipolar

Title		
Size	Number	Revision
A		
Date:	4/22/2016	Sheet of
File:	C:\Users\...\[05]-POWER.SchDoc	Drawn By:

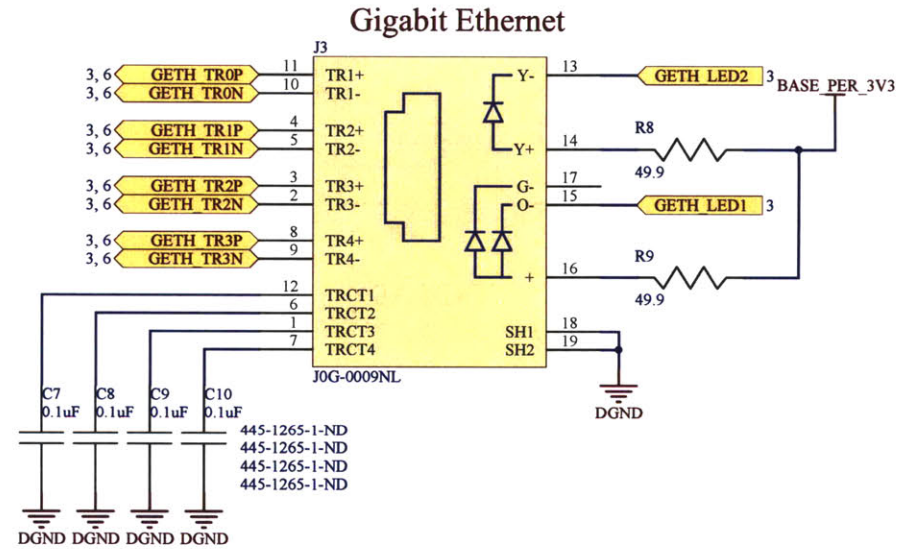
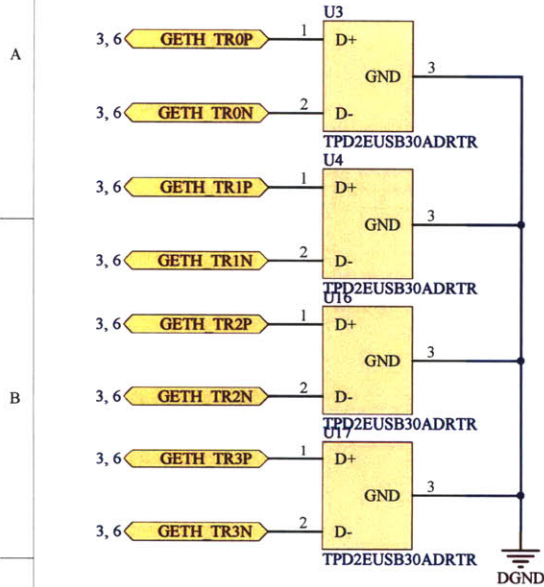
331

A
B
C
D

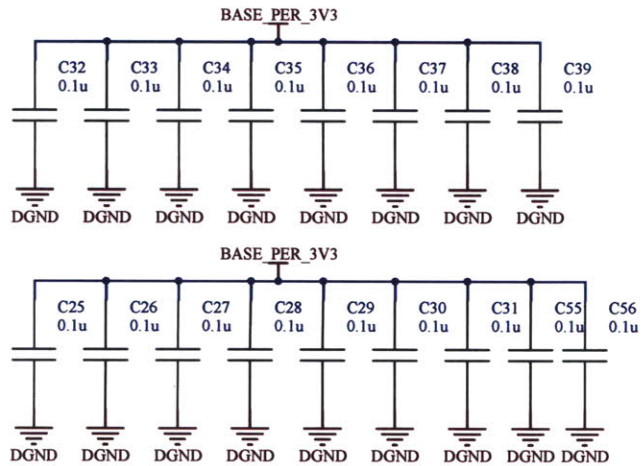
A
B
C
D

1 2 3 4

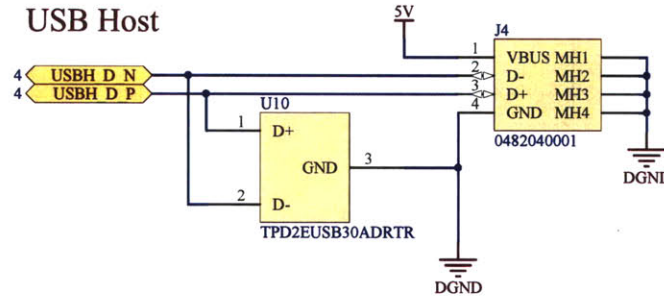
Network and USB



Bypass Caps for Micro and SDRAM



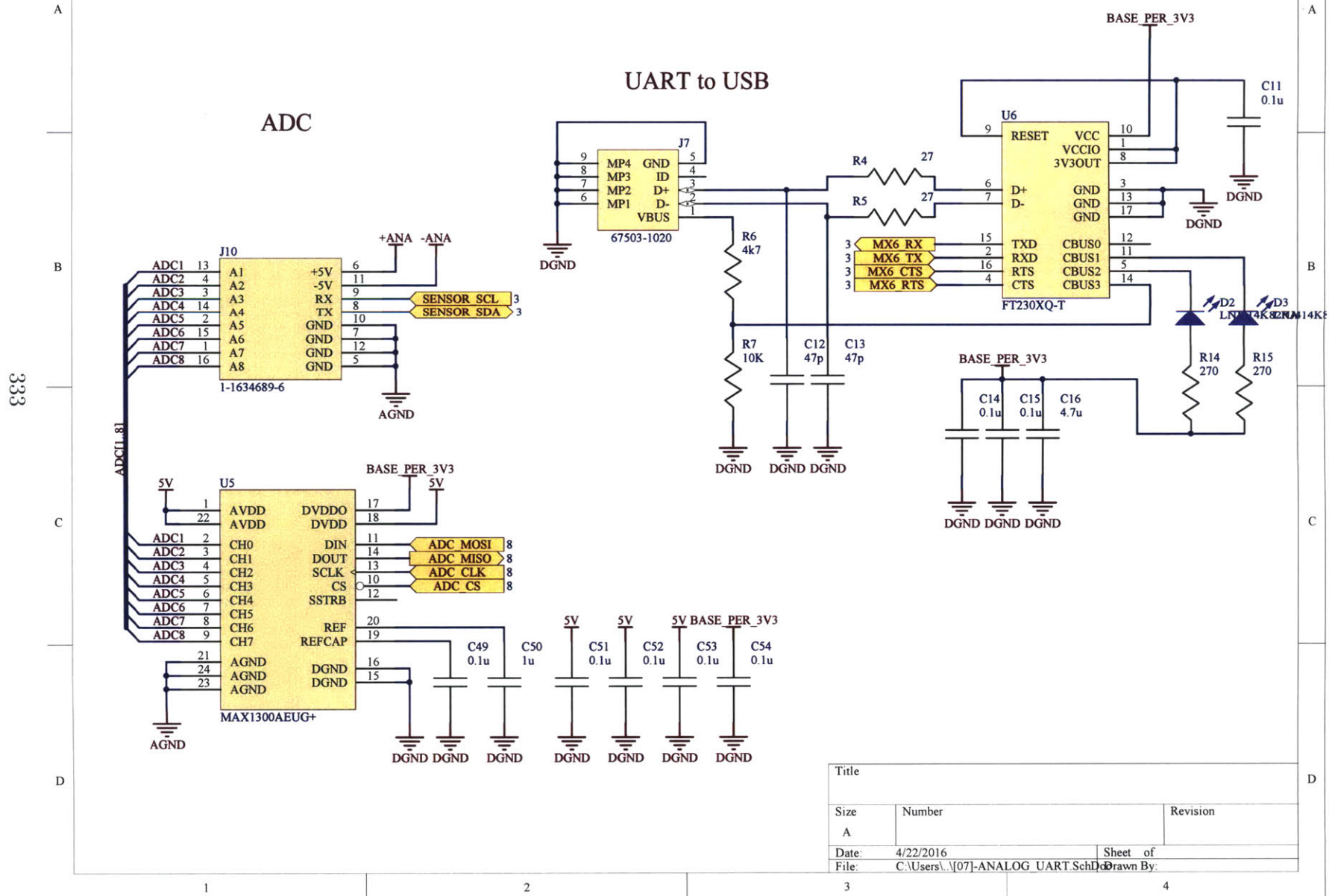
USB Host



Title		
Size	Number	Revision
A		
Date:	4/22/2016	Sheet of
File:	C:\Users\...[06]-NTWK_USB.SchDoc	Drawn By:

332

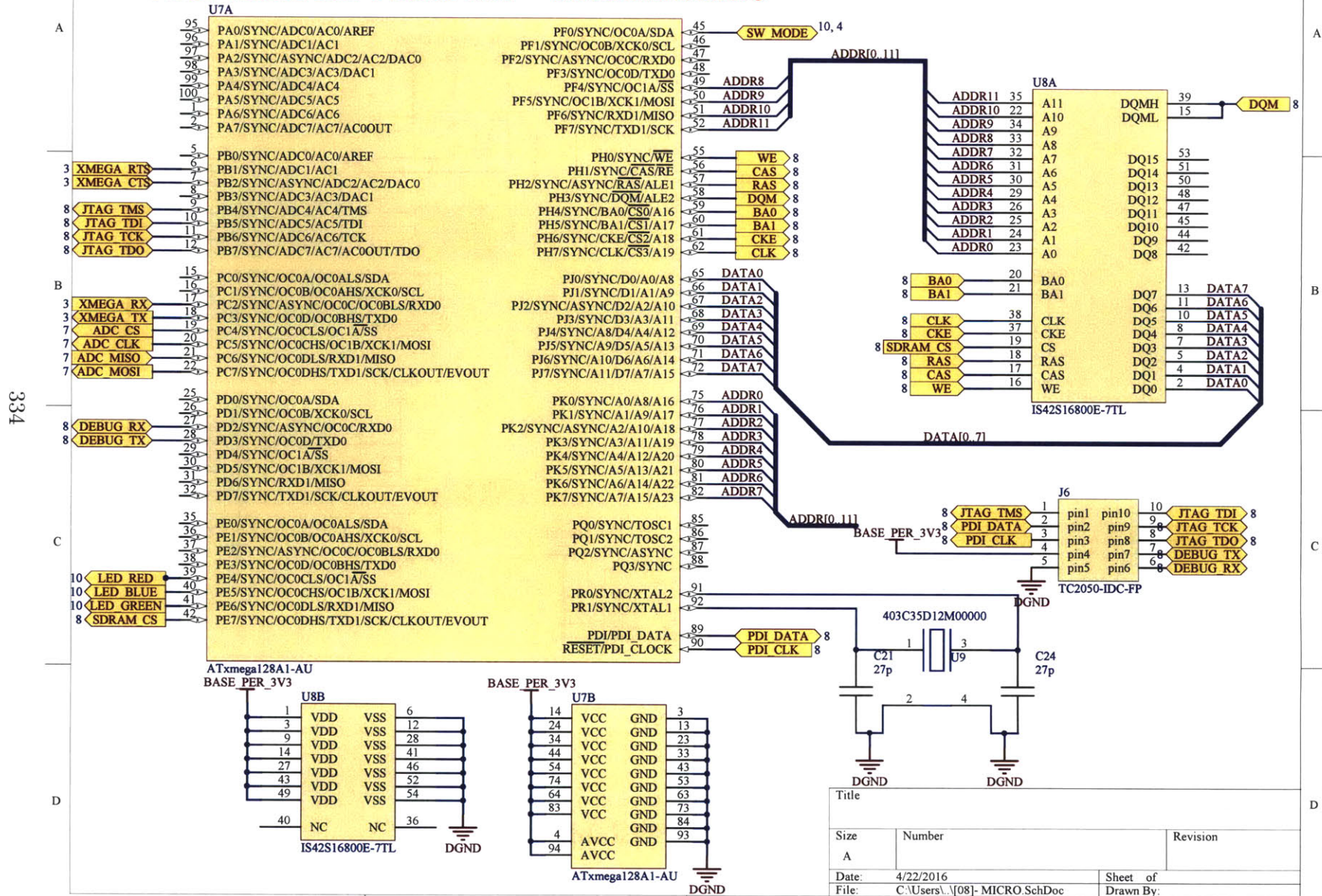
ADC INPUT and DEBUG UART



Title		
Size	Number	Revision
A		
Date:	4/22/2016	Sheet of
File:	C:\Users\...[07]-ANALOG_UART.SchDoc	Drawn By:

Micro and SDRAM

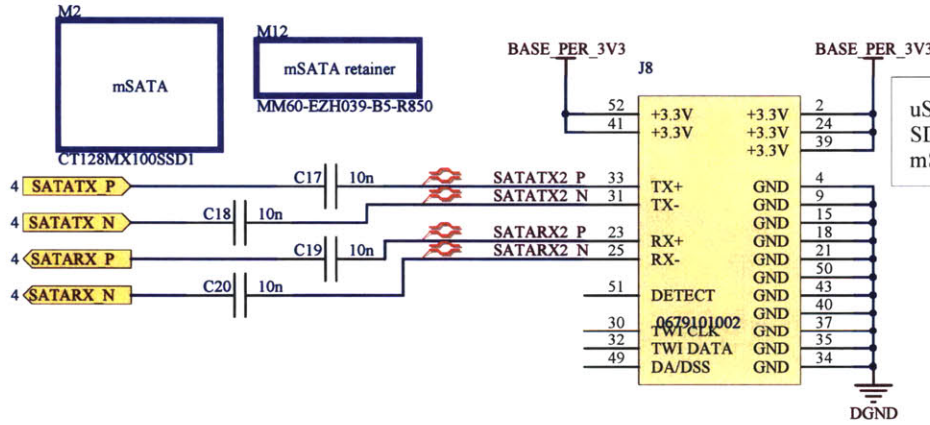
****NOTE THIS IS THE WRONG CHIP** Use XMEGA128A1U**



Title		
Size	Number	Revision
A		
Date:	4/22/2016	Sheet of
File:	C:\Users\...\[08]- MICRO.SchDoc	Drawn By:

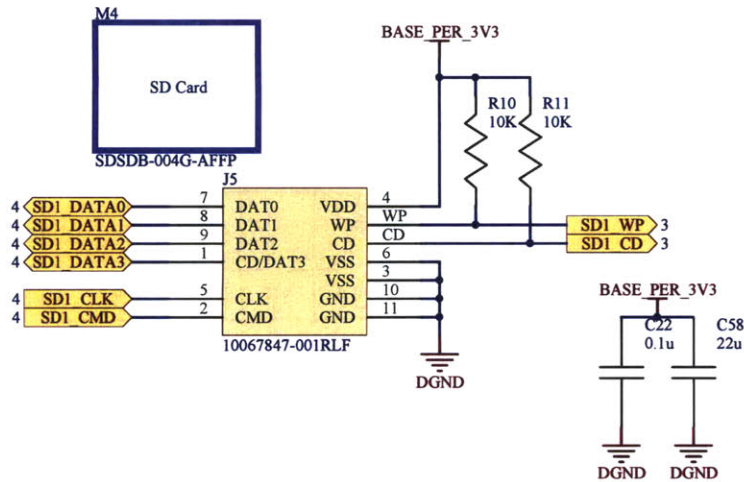
STORAGE

mSATA

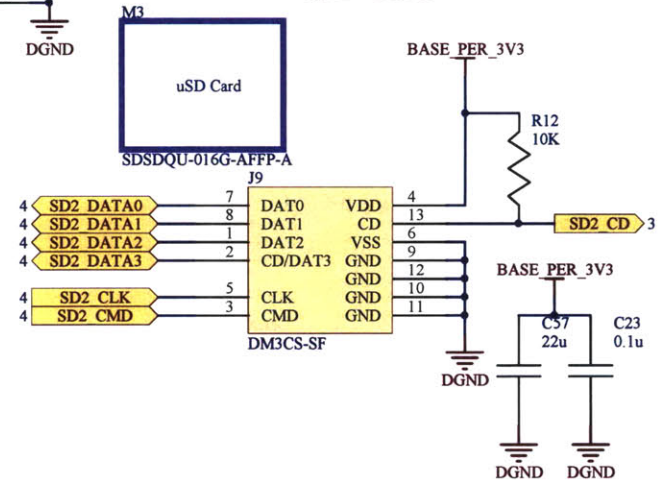


uSD Card: bootable volume, not user accessible
 SD Card: auxiliary storage, user accessible
 mSATA: auxiliary storage, not user accessible

SD Card



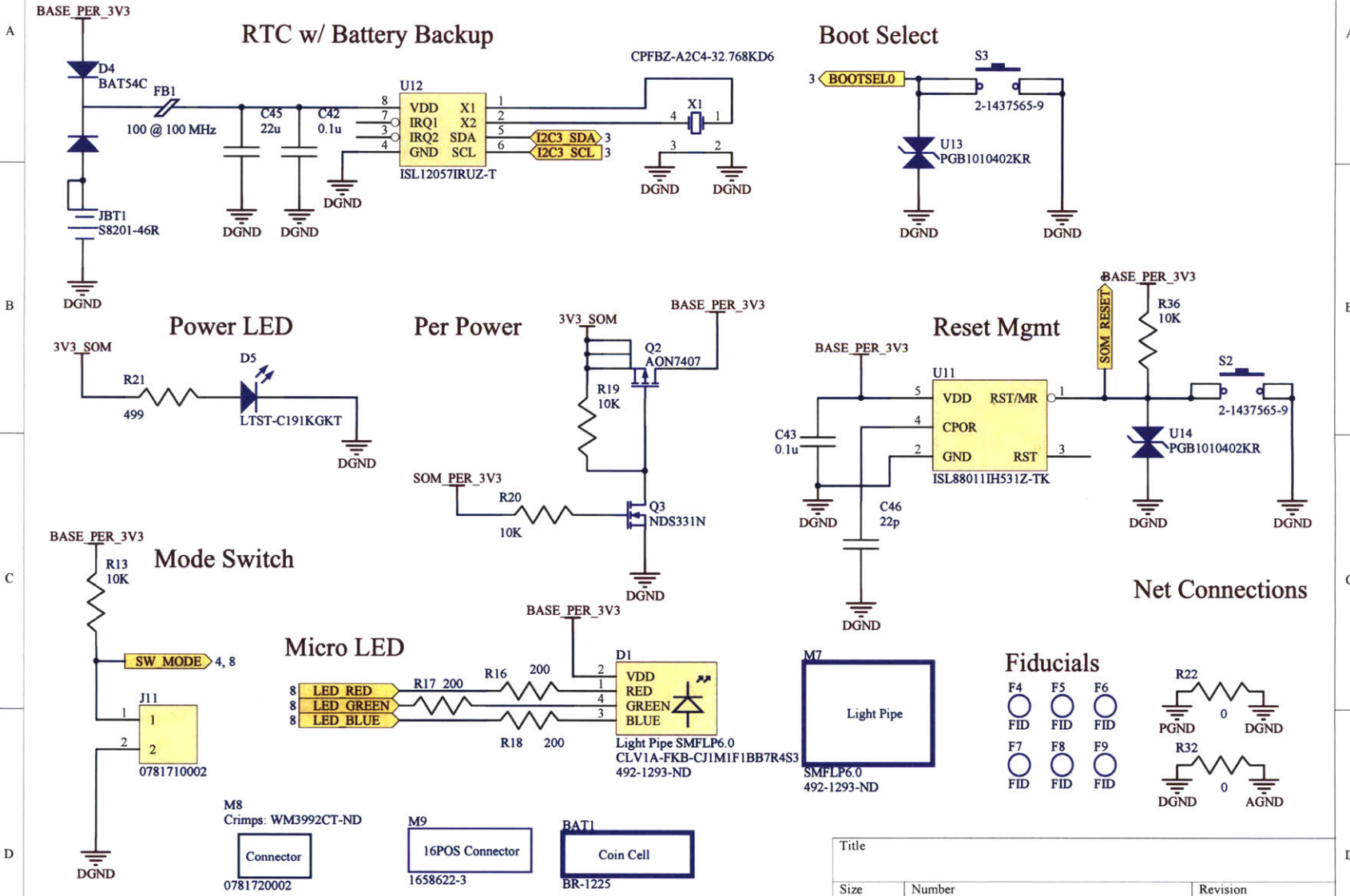
uSD Card



Title		
Size	Number	Revision
A		
Date:	4/22/2016	Sheet of
File:	C:\Users\...\[09] STORAGE.SchDoc	Drawn By:

335

MISCELLANEOUS



336

Title		
Size	Number	Revision
A		
Date:	4/22/2016	Sheet of
File:	C:\Users\...[10] - MISC.SchDoc	Drawn By:

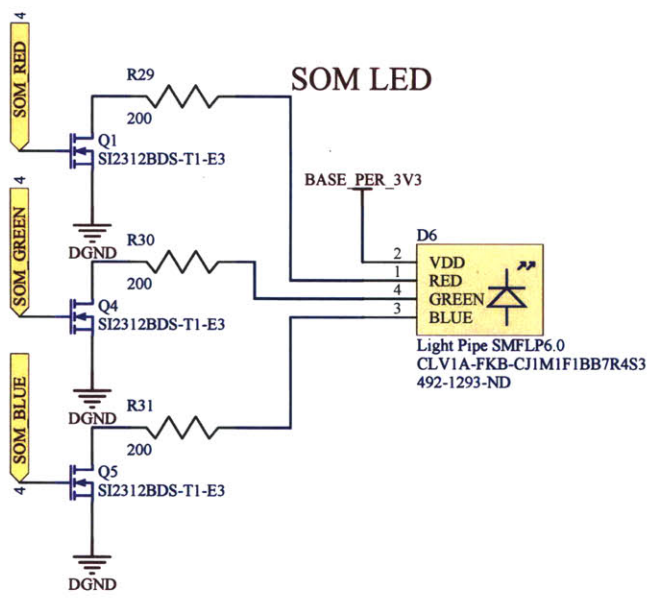
DOC: REVISION HISTORY

337

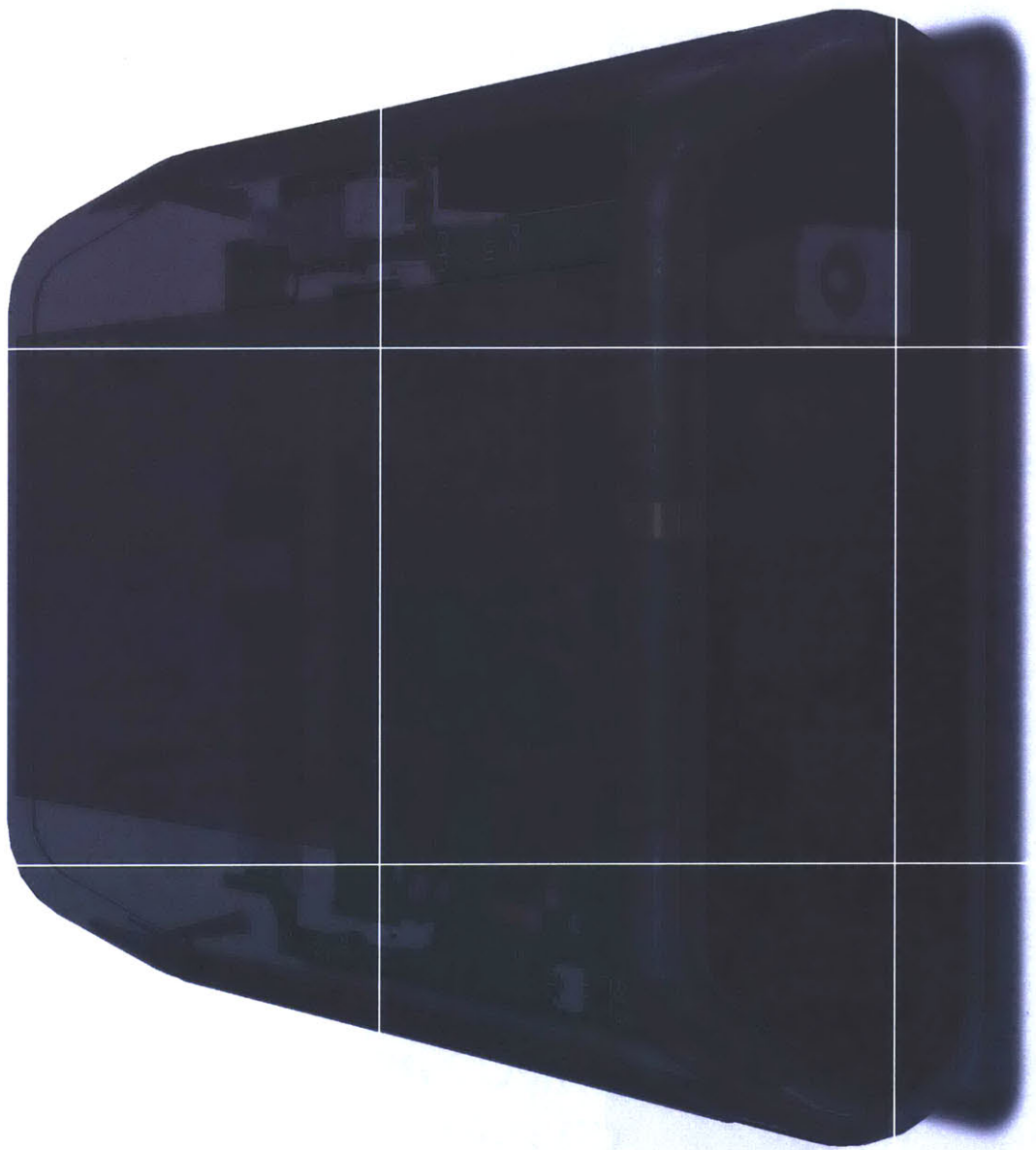
Wattsorth		CONFIDENTIAL. Do not distribute.	
Title	NILM Board		Variant: [No Variations]
Page Contents:	[12] - DOC REVISION HISTORY.SchDoc	Checked by	
Size:	DWG NO	Revision: V112	
Date:	4/22/2016	Sheet	11 of 11

MISCELLANEOUS (CONT)

338



Title		
Size	Number	Revision
A		
Date:	4/22/2016	Sheet of
File:	C:\Users\A.[11] MISC2.SchDoc	Drawn By:





B.4 Server Architecture

Creating a secure and reliable infrastructure to manage the remote energy monitors requires more than a web server with VPN software. NILM Manager is in fact a cluster of seven separate servers, illustrated on the following page. These servers work together to provide a complete suite of management tools. In our experimental implementation, these servers run as virtual machines (VM's) on two Dell R320 servers with Xen hypervisor. All VM's are paravirtualized Ubuntu 14.0 images. The servers are named according to their roles, described next.

B.4.1 Firewall

The firewall is the only machine with a public network connection. Incoming traffic is analyzed against a set of rules that determines whether the particular packet is allowed and where it should be routed. The firewall VM does not run any services itself making it easier to secure against attack.

B.4.2 Backbone

The backbone manages communication between servers in the management node and remote NILM's. NILM's authenticate with the backbone using SSL certificates. Certificates (unlike passwords) allow for two-way authentication meaning NILM's verify the identity of the management node and vice versa. This prevents impostor management nodes from accessing NILM's and rogue NILM's from accessing the management node. Once a NILM authenticates, the backbone assigns it an IP address and host-name which uniquely identifies it on the VPN. Other servers on the management node can access the NILM by requesting its IP address from the backbone using a domain name resolution service (DNS).

B.4.3 Web

The web server hosts the NILM Manager website. The firewall directs all inbound HTTP requests to this VM using port address translation. This protects the web server from unsolicited (and potentially malicious) traffic. One of the advantages to deploying the web server in a VM is that resources can be scaled with user demand. If web traffic increases, the Xen Hypervisor can reassign processor cores and memory to handle the additional load [81].

B.4.4 Metrics

Metrics runs Nagios [60] and Ganglia [82] monitoring services. Nagios periodically checks the health of remote NILM's and Ganglia provides a trending report of memory usage, CPU load, and other metrics for each NILM machine . This enables rapid detection and diagnosis of faults in deployed NILM systems. Additionally it provides profiling information that helps in designing hardware for future NILM's based on their real world usage.

B.4.5 Archive

Archive holds NILM data for long term storage. This server is used to backup valuable data sets collected by deployed NILM's. The archive server is useful for testing and evaluating different data processing techniques as the machine has significant hardware resources as well as a reliable network connection (neither of which can be assumed for remote NILM's).

B.4.6 Devops

Devops (a portmanteau of “development” and “operations”) provides configuration management for remote NILM's. All of the settings, packages, and scripts needed by a NILM are stored on this server using Puppet, an open source management tool [83]. Puppet automatically mirrors updates to these files to every NILM on the VPN ensuring they have consistent and up-to-date configurations. Without such a

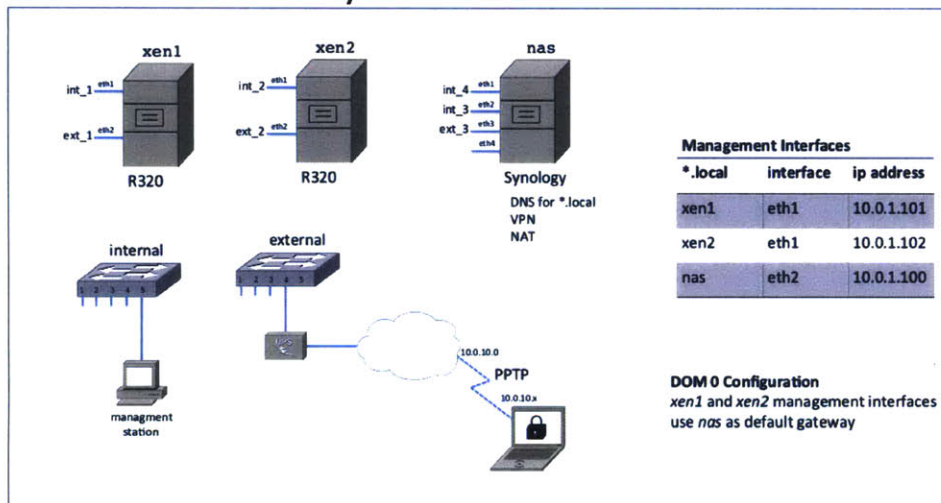
service any update to a setting or script would have to be manually applied to each NILM- a tedious and error prone process.

B.4.7 Git

This server hosts git repositories [84] for all the software developed for NILM's and NILM Manager. Git provides version controlled storage and enables collaborative work on the NILM code base.

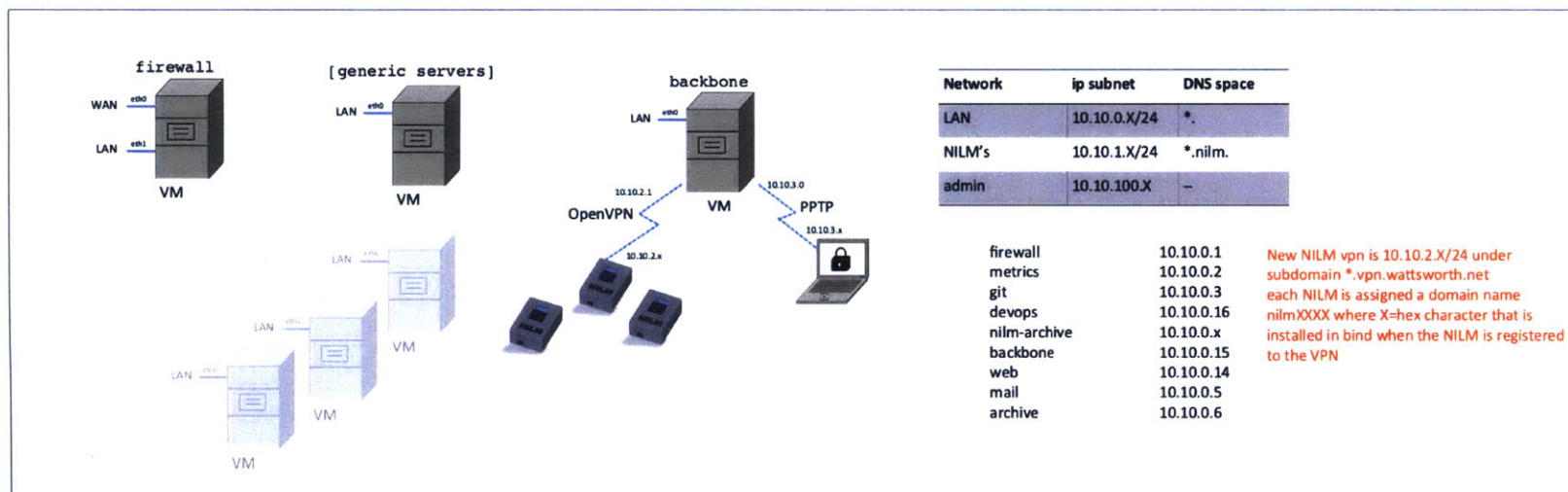
Together these servers create a reliable and secure infrastructure for managing NILM systems. They permit access to remote database storage located at different monitoring sites. Visualizations or other results of data analysis can be returned from a remote monitor. Python or Octave-style analysis code can be transmitted to remote monitors to provide new analytical capabilities or requests. In both directions, network bandwidth is minimized, as large data streams never have to be transmitted from the remote monitoring sites. The next two sections describe how this infrastructure makes it possible to view and process almost unlimited amounts of NILM data with very little exchange of information over a network.

Physical Hardware



Physical Servers implement a private cloud using XEN

Private Cloud



Bibliography

- [1] K. Fehrenbacher. 5 reasons why microsoft hohm didn't take off. *Gigaom*, July 2011.
- [2] J. S. Donnal and S. B. Leeb. Noncontact power meter. *IEEE Sensors Journal*, 15(2):1161–1169, Feb 2015.
- [3] C. Schantz, J. Donnal, B. Sennett, M. Gillman, S. Muller, and S. Leeb. Water nonintrusive load monitoring. *IEEE Sensors Journal*, 15(4):2177–2185, April 2015.
- [4] C. Schantz, J. Donnal, S. Leeb, P. N. Marimuthu, and S. Habib. Nwattsworth: Monitor electric power anywhere. *WIT Transactions on the Built Environment*, 139:125–136, 2014.
- [5] J. Donnal, U. Orji, C. Schantz, J. Moon, S. B. Leeb, J. Paris, A. Goshorn, K. Thomas, J. Dubinsky, R. Cox, and J. Moon. Vampire: Accessing a life-blood of information for maintenance and damage assessment. In *American Society of Naval Engineers Day*, pages 132–142, February 2012.
- [6] J. Moon, J. Donnal, J. Paris, and S. B. Leeb. Vampire: A magnetically self-powered sensor node capable of wireless transmission. In *Applied Power Electronics Conference and Exposition (APEC), 2013 Twenty-Eighth Annual IEEE*, pages 3151–3159, March 2013.

- [7] J. Donnal and S. B. Leeb. Wattsworth: Monitor electric power anywhere. In *Applied Power Electronics Conference and Exposition (APEC), 2014 Twenty-Ninth Annual IEEE*, pages 2223–2230, March 2014.
- [8] J. Paris, J. S. Donnal, and S. B. Leeb. Nilmdb: The non-intrusive load monitor database. *IEEE Transactions on Smart Grid*, 5(5):2459–2467, Sept 2014.
- [9] J. Paris, J. S. Donnal, Z. Remscrim, S. B. Leeb, and S. R. Shaw. The sinefit spectral envelope preprocessor. *IEEE Sensors Journal*, 14(12):4385–4394, Dec 2014.
- [10] John Donnal, Jim Paris, and Steve Leeb. Energy apps. Provisional US Patent 62/242618, October 2015.
- [11] D. Lawrence. Hardware and software architecture for non-contact, non-intrusive load monitoring. Master’s thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, February 2016.
- [12] Micro Magnetics. *STJ-340: Four Element Bridge Magnetic Sensor Data Sheet*, May 2010.
- [13] Allegro. *Allegro 1362 Data Sheet*, December 2013.
- [14] Melexis. *Melexis MLX91206 Data Sheet*, December 2013.
- [15] M. Julliere. Tunneling between ferromagnetic films. *Phys. Lett.*, 54A: 225226, 1975.
- [16] S. S. P. Parkin et al. Giant tunnelling magnetoresistance at room temperature with mgo (100) tunnel barriers. *Nat. Mat.*, 3 (12):862867, 2004.
- [17] S. Ikeda, J. Hayakawa, Y. Ashizawa, Y. M. Lee, K. Miura, H. Hasegawa, M. Tsunoda, F. Matsukura, and H. Ohno. Tunnel magnetoresistance of 604% at 300 K by suppression of Ta diffusion in CoFeB/MgO/CoFeB pseudo-spin-valves annealed at high temperature. *Applied Physics Letters*, 93(8):082508, August 2008.

- [18] B. Hoberman. *The Emergence of Practical MRAM*. Crocus Technologies, April 2009.
- [19] W. A. Zisman. A new method of measuring contact potential differences in metals. *Review of Scientific Instruments*, 3:367–370, March 1932.
- [20] W. E. Vosteen. A high speed electrostatic voltmeter technique. In *Industry Applications Society Annual Meeting*, pages 1617–1619, October 1988.
- [21] D. Grant, G. Hearn, W. Maggs, and I. Gonzalez. An electrostatic charge meter using a microcontroller offers advanced features and easier atex certification. *Journal of Electrostatics*, 67:473–476, May 2009.
- [22] L. Wu, P. Wouters, E. van Heesch, and E. Steennis. On-site voltage measurement with capacitive sensors on high voltage systems. In *IEEE Trondheim PowerTech*, pages 1–6, June 2011.
- [23] J. Bobowski, S. Ferdous, and T. Johnson. Calibrated single-contact voltage sensor for high-voltage monitoring applications. *IEEE Trans. Instrum. Meas.*, 64:923–934, April 2015.
- [24] K. M. Tsang and W. L. Chan. Dual capacitive sensors for non-contact ac voltage measurement. *Sensors and Actuators A: Physical*, 167:261–266, June 2011.
- [25] D. Balsamo, D. Porcarelli, L. Benini, and B. Davide. A new non-invasive voltage measurement method for wireless analysis of electrical parameters and power quality. In *IEEE Sensors*, pages 1–4, November 2013.
- [26] A. Parashar; M. Adler; K. E. Fleming; M. Pellauer; J. S. Emer. Leap: A virtual platform architecture for fpgas. In *CARL 2010: The 1st workshop on the Intersections of Computer Architecture and Reconfigurable Logic*, 2010.
- [27] T. Horie. Alpha blending tutorial, June 2002.

- [28] R. Zachar. Naval applications of enhanced temperature, vibration and power monitoring. Master's thesis, Massachusetts Institute of Technology, Department of Mechanical Engineering and Engineering Systems Division, June 2015.
- [29] R. Zachar, P. Lindahl, J. Donnal, W. Cotta, C. Schantz, and S. B. Leeb. Utilizing spin-down transients for vibration-based diagnostics of resiliently mounted machines. *IEEE Transactions on Instrumentation and Measurement*, PP(99):1–10, 2016.
- [30] J. Donnal. Home NILM: A Comprehensive Non-Intrusive Load Monitoring Toolkit. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, June 2013.
- [31] J. Paris. *A Comprehensive System for Non-Intrusive Load Monitoring and Diagnostics*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, September 2013.
- [32] Alan V. Oppenheim, Alan S. Willsky, and Ian T. Young. *Signal and Systems*. Prentice-Hall, 1997.
- [33] Alan V. Oppenheim and Ronald W. Schaffer. *Discrete Time Signal Processing*. Prentice-Hall, 2009.
- [34] S.K. Yang. A condition-based failure-prediction and processing-scheme for preventive maintenance. *Reliability, IEEE Transactions on*, 52(3):373–383, Sept 2003.
- [35] Andrew K.S. Jardine, Daming Lin, and Dragan Banjevic. A review on machinery diagnostics and prognostics implementing condition-based maintenance. *Mechanical Systems and Signal Processing*, 20(7):1483 – 1510, 2006.
- [36] Aiwin Heng, Sheng Zhang, Andy C.C. Tan, and Joseph Mathew. Rotating machinery prognostics: State of the art, challenges and opportunities. *Mechanical Systems and Signal Processing*, 23(3):724 – 739, 2009.

- [37] S. Nandi, H.A. Toliyat, and Xiaodong Li. Condition monitoring and fault diagnosis of electrical motors-a review. *Energy Conversion, IEEE Transactions on*, 20(4):719–729, Dec 2005.
- [38] International Organization for Standardization. ISO/TC 108 condition monitoring and diagnostics of machine systems.
- [39] U.S. Department of Defense. MIL-STD-167-1A: Department of defense test method standard: Mechanical vibration of shipboard equipment (type i environmental and type ii internationally excited), 2005.
- [40] J.R. Stack, R.G. Harley, and T.G. Habetler. An amplitude modulation detector for fault diagnosis in rolling element bearings. *Industrial Electronics, IEEE Transactions on*, 51(5):1097–1102, Oct 2004.
- [41] C. Bianchini, F. Immovilli, M. Cocconcelli, R. Rubini, and A. Bellini. Fault detection of linear bearings in brushless ac linear motors by vibration analysis. *Industrial Electronics, IEEE Transactions on*, 58(5):1684–1694, May 2011.
- [42] I. Bediaga, X. Mendizabal, A. Arnaiz, and J. Munoa. Ball bearing damage detection using traditional signal processing algorithms. *Instrumentation Measurement Magazine, IEEE*, 16(2):20–25, April 2013.
- [43] A. Soualhi, K. Medjaher, and N. Zerhouni. Bearing health monitoring based on hilbert-huang transform, support vector machine, and regression. *Instrumentation and Measurement, IEEE Transactions on*, 64(1):52–62, Jan 2015.
- [44] C. Concari, G. Franceschini, and C. Tassoni. Differential diagnosis based on multivariable monitoring to assess induction machine rotor conditions. *Industrial Electronics, IEEE Transactions on*, 55(12):4156–4166, Dec 2008.
- [45] V. Climente-Alarcon, J.A. Antonino-Daviu, F. Vedreno-Santos, and R. Puche-Panadero. Vibration transient detection of broken rotor bars by psh sidebands. *Industry Applications, IEEE Transactions on*, 49(6):2576–2582, Nov 2013.

- [46] Anders Brandt. *Noise and Vibration Analysis; Signal Analysis and Experimental Procedures*. John Wiley & Sons, Ltd, 2011.
- [47] Zhangjun Tang, P. Pillay, and A.M. Omekanda. Vibration prediction in switched reluctance motors with transfer function identification from shaker and force hammer tests. *Industry Applications, IEEE Transactions on*, 39(4):978–985, July 2003.
- [48] Leo L. Beranek and I. L. Vér, editors. *Noise and vibration control engineering: principles and applications*. Wiley, 1992.
- [49] Christopher James Schantz. *Methods for non-intrusive sensing and system monitoring*. PhD thesis, Massachusetts Institute of Technology, 2014.
- [50] Julius O. Smith III. *Mathematics of the Discrete Fourier Transform (DFT): with Audio Applications - Second Edition*. W3K Publishing, 2007.
- [51] D. Grillo, N. Pasquino, L. Angrisani, and R. Schiano Lo Moriello. An efficient extension of the zero-crossing technique to measure frequency of noisy signals. In *Instrumentation and Measurement Technology Conference, 2012 IEEE International*, pages 2706–2709, May 2012.
- [52] R.W. Wall. Simple methods for detecting zero crossing. In *Industrial Electronics Society, 2003. IECON '03. The 29th Annual Conference of the IEEE*, volume 3, pages 2477–2481 Vol.3, Nov 2003.
- [53] O. Vainio and S.J. Ovaska. Digital filtering for robust 50/60 hz zero-crossing detectors. *Instrumentation and Measurement, IEEE Transactions on*, 45(2):426–430, Apr 1996.
- [54] Peter D. Welch. The use of fast Fourier transform for the estimation of power spectra: A method based on time averaging over short, modified periodograms. *IEEE Transactions on Audio and Electroacoustics*, 15(2):70–73, June 1967.

- [55] Jinyeong Moon and S.B. Leeb. Analysis model for magnetic energy harvesters. *Power Electronics, IEEE Transactions on*, 30(8):4302–4311, Aug 2015.
- [56] J. Paris, Z. Remscrim, K. Douglas, S. B. Leeb, R. W. Cox, S. T. Gavin, S. G. Coe, J. R. Haag, and A. Goshorn. Scalability of non-intrusive load monitoring for shipboard applications. In *American Society of Naval Engineers Day 2009*, National Harbor, Maryland, April 2009.
- [57] N. Amirach, B. Xerri, B. Borloz, and C. Jauffret. A new approach for event detection and feature extraction for nilm. In *Electronics, Circuits and Systems (ICECS), 2014 21st IEEE International Conference on*, pages 287–290, Dec 2014.
- [58] A.N. Milioudis, G.T. Andreou, V.N. Katsanou, K.I. Sgouras, and D.P. Labridis. Event detection for load disaggregation in smart metering. In *Innovative Smart Grid Technologies Europe (ISGT EUROPE), 2013 4th IEEE/PES*, pages 1–5, Oct 2013.
- [59] Ming Dong, P.C.M. Meira, Wilsun Xu, and C.Y. Chung. Non-intrusive signature extraction for major residential loads. *Smart Grid, IEEE Transactions on*, 4(3):1421–1430, Sept 2013.
- [60] D. Josephsen. *Nagios: Building Enterprise-Grade Monitoring Infrastructures for Systems and Networks*. Prentice Hall, second edition, 2013.
- [61] Flot. Attractive javascript plotting for jquery. Available <http://www.flotcharts.org/>. Accessed 2015-01-29.
- [62] PG&E. Smartmeter network- how it works. Available <http://www.pge.com/en/myhome/customerservice/smartmeter/howitworks/index.page>. Accessed 2015-01-09.
- [63] ComEd. Smartmeters: Empowering you to save energy and money. Available <https://www.comed.com/Documents/technology/What%20is%20a%20Smart%20Meter.pdf>. Accessed 2015-01-09.

- [64] BGE. How smart meters work. Available <http://www.bge.com/smartenergy/smartgrid/smartmeters/Pages/How-Smart-Meters-Work.aspx>. Accessed 2015-01-09.
- [65] J. Paris, J. S. Donnal, R. Cox, and S. Leeb. Hunting cyclic energy wasters. *IEEE Transactions on Smart Grid*, 5(6):2777–2786, Nov 2014.
- [66] Belkin. Wemo insight switch. Available <http://www.belkin.com/us/p/P-F7C029/>. Accessed 2015-08-28.
- [67] DLink. Wi-fi smart plug. Available <http://us.dlink.com/products/connected-home/wi-fi-smart-plug/>. Accessed 2015-08-28.
- [68] M. Gillman. Interpreting human activity from electrical consumption data through non-intrusive load monitoring. Master’s thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, June 2014.
- [69] W. Cotta. Machinery diagnostics and characterization through electrical sensing. Master’s thesis, Massachusetts Institute of Technology, Department of Mechanical Engineering, June 2015.
- [70] Shipboard Automatic Watchstander. Greg bredariol, john donnal, william cotta, and steven leeb. In *American Society of Naval Engineers Day*, February 2016.
- [71] C. Laughman, Kwangduk Lee, R. Cox, S. Shaw, S. Leeb, L. Norford, and P. Armstrong. Power signature analysis. *IEEE Power and Energy Magazine*, 1(2):56–63, Mar 2003.
- [72] H. Friedman; A. Potter; T. Haasl; and D. Claridge. Persistence of benefits from new building commissioning. Technical report, California Energy Commission, October 2003.
- [73] M. Schwartz; K. Blakeley; R. O’Rourke. Department of defense energy initiatives: Background and issues for congress. Technical report, Congressional Research Service, December 2012.

- [74] Report of the defense science board task force on dod energy strategy, “more fight, less fuel”. Technical report, Office of the Under Secretary of Defense For Acquisition, Technology, and Logistics, February 2008.
- [75] Noblis. Sustainable forward operating bases. Technical report, Strategic Environmental Research and Development Program (SERDP), 2010.
- [76] Eady, D.S., Siegel, S.B., Bell, R.S., and Dicke, S. H. Sustain the mission project: Casualty factors for fuel and water resupply convoys. Technical report, Army Environmental Policy Institute, 2009.
- [77] International Energy Agency. Annex 47: Report 2 commissioning tools for existing and low energy buildigns. Technical report, Organisation for Economic Co-operation and Development, 2010.
- [78] Mark Augustin Piber. Improving shipboard maintenance practices using non-intrusive load monitoring. Master’s thesis, Massachusetts Institute of Technology, 2007.
- [79] COMDTINST. 3500.3 operational risk management, 1999.
- [80] John Pike. 270-foot medium endurance cutter (wmec) famous cutter class, November 2015.
- [81] Daniel Tosatto. *Citrix XenServer 6.0: Administration Essential Guide*. Packt Publishing, Birmingham, UK, 2012.
- [82] M. Massie, B. Li, B. Nicholes, V. Vuksan, R. Alexandar, et al. *Monitoring with Ganglia: Tracking Dynamic Host and Application Metrics at Scale*. O’Reilly Media, 2012.
- [83] S Krum, W.V. Hevelingen, B. Kero, J. Turnbull, and J. McCune. *Pro Puppet*. Apress, second edition, 2013.
- [84] S. Chacon. *Pro Git*. Apress, second edition, 2014.