UNIVERSITY OF CAPE TOWN

Department of Electrical Engineering

Design of a Power Profiler for Domestic Load Research Analysis

By

Grant Stowe

Dissertation presented for the Degree of Masters of Science in Engineering

in the Department of Electrical Engineering

University of Cape Town

November 2012

Declaration

I, Grant Stowe, hereby declare that this dissertation is my own, unaided work. I know the meaning of plagiarism and declare that all the work in this document, save for that which is properly acknowledged, is my own.

This work has not been submitted for any other degree in this or any other University.

I hereby grant the University free license to reproduce this dissertation in whole or in part for the purpose of research.

_____

Grant Richard Stowe

November, 2012

Abstract

Design of a Power Profiler for Domestic Load Research Analysis

Department of Electrical Engineering, University of Cape Town

Master of Science in Engineering

by Grant R Stowe

This dissertation presents the development of a proof-of-concept measurement instrument to satisfy the requirements of the University of Cape Town's Load Research Group. The new instrument needed to make use of current technologies to cost-effectively measure multiple residential households and provide remote communications capabilities for reliable remote data retrieval. Measurement of three-phase 4-wire configurations was necessary as well as the ability to create second-based profiles of consumer voltage, current and active, reactive and apparent powers for load identification purposes.

The main components of a measurement system were identified and prototype development was split into two stages: component integration testing and final product integration. The product was called the Power Profiler and two early prototypes were developed to test the main measurement and processing components and mechanical integration to create a compact versatile test instrument. Energy measurement ICs were used instead of discrete sampling of voltage and current to improve accuracy and allow the use of a low-cost and easy to program digital signal controller for measurement processing and storage. Several communication options were incorporated into the Power Profiler including remote GSM and local USB and Bluetooth communication.

The third prototype was subjected to full calibration testing with a Chauvin Arnoux power quality analyzer used as the reference meter. The Power Profiler proved to be a high accuracy instrument with a typical inaccuracy of 0.4% for RMS voltage (over a 20:1 range), 0% for RMS current, 1.1% for active power, 0.3% for reactive power and 0.1% apparent power versus the reference meter. To validate the Power Profiler measurement functionality it was connected to a residential household in conjunction with the reference meter and 4 hours of measurement data was captured for comparison. After a successful validation the Power Profiler was then left to capture another 43 hours of measurement data using a 5-second, 10-minute and 2-hour measurement profile. This data was then analyzed to see if non-intrusive load discrimination could be performed.

From the measurement and validation data several common household appliances were identified including the refrigerator, geyser, kettle, microwave and washing machine. The Power Profiler stores not only the average values measured between recording intervals but the maximum and minimum levels as well. This

data proved very valuable for the detection of voltage dips and swells that would otherwise have gone unreported by a conventional power meter. The maximum statistic also effectively showed the startup current of motor-based appliances such as the refrigerator and washing machine which greatly assisted with load discrimination.

The embedded software for the Power Profiler was split into several software tasks to simplify code maintenance and future development. A C# PC Control and Configuration application was developed to automate calibration as far as possible, and to allow an operator to configure and upload measurement data from the Power Profiler. Extensive use of object-oriented programming techniques was made to allow the reuse of the interface and control software in future applications by other developers.

Remote communications capabilities were demonstrated over GSM GPRS connections to allow the retrieval of measurement data from field-deployed Power Profilers. The ability of the Power Profiler to simultaneously record second-, minute- and hour-based profiles and to allow the operator to decide which of this measured data to upload is highly effective in managing remote communication bandwidth and reducing overall operating cost of the system.

The Power Profiler product proved to be a cost-effective replacement to the currently used logger and offered more channels with higher resolution measurement and profiling capabilities in a compact instrument. Remote communication capabilities reduces the overall operating cost of the system and simplified measurement retrieval. The ability to easily custom-develop software for both the Power Profiler and PC-based applications made it a flexible instrument valuable for research-orientated applications.

Acknowledgements

I would like to express my sincere gratitude to my supervisor Prof. T. Gaunt whose continued support allowed me to see this project to completion. I would also like to thank Prof. J. Tapson for presenting the project opportunity and for his technical guidance during the early stages of the development.

Thanks are also due to Chris Wosniak of the Power Engineering laboratory for the many times he set up the equipment necessary for calibration and testing of the field prototype.

Finally, special thanks need to go to my parents Peter and Erika and to Janine Pingo for their support, encouragement and belief that I would see the project through to completion.

Contents

Chapter 1

Introduction

"The construction and improvement of energy infrastructure forms an important element of government's focus on infrastructure development, particularly in rural areas. As such, better long-term planning of generation, distribution and maintenance is critical for the achievement of the 2014 goal of universal access to electricity." [National Planning Commission, 2010]

1.1. Background

The South African economy has grown substantially in the past 10 years and this has put significant pressure on State utility provider Eskom to meet the growing demand for electricity. In 2008 this increased pressure on an already strained National network became evident when electricity demand exceeded generation capacity and blackouts resulted across the Country. The solution at that time was to introduce "load shedding" until available generation capacity could be increased according to an ESKOM growth plan. The ability to forecast power usage is an essential part of proper network planning for power generation and distribution [Gaunt, et al., 2007].

In 2011 the situation became more manageable through the use of gas-powered generation plants to provide peak demand while more base-load plants can be constructed. This is however a very costly method of electricity generation and can only be seen as a short-term solution until sufficient base-load becomes available.

The South African National Planning Commission's National Development Plan framework economic infrastructure target list includes increasing the proportion of people with access to electricity from 70% in 2010, to 95% by 2030 [National Planning Commission, 2011]. The Development Indicators 2010 classifies 23.9% of the number of households in South Africa as an informal or traditional dwelling, with the planned building of almost four million houses for the poor [National Planning Commission, 2010].

Accurate electricity usage models assist network planners to better determine future generation and distribution requirements. To generate these models, electricity usage per household needs to be captured and analyzed to determine emerging trends in consumer usage. To determine what types of appliances a typical dwelling contains would require a survey to be conducted of the household, and the typical usage profile determined by analyzing the recorded current signature.

1.2. The Current Solution to Load Analysis

The University of Cape Town's Load Research Group utilizes a power logger that was designed in 1987 and subsequently updated by Eskom in approximately 1993. This product offers one voltage and up to seven current measurement channels.

This logger suffers from several restrictions including:

- No remote communications capability thus requiring service disruption to the area to allow the downloading of measured data via a serial port
- Frequent corruption of measured data either due to surge transients or communications errors
- Only one voltage channel and therefore not able to measure per-phase voltage in a three-phase four-wire (star) configuration

It must be mentioned that this logger was designed in a period when GSM communications and low-cost microprocessors were not available. The worldwide move by utility providers towards automated meter reading (AMR) and advanced metering infrastructure (AMI) gives them access to load profiling data in that a smart meter incorporates the two main requirements for this: energy measurement and remote communication ability.

One drawback of AMI products for use in load analysis is that they typically measure only the energy in one load (single- or three-phase), have limited measurement functionality and they often run proprietary communications protocols and interfaces. Since 2004 the use of GSM cellular communications for telemetry applications has increased considerably due to the reducing cost of embeddable GSM modems and the availability of packet-switched data (GPRS, EDGE and 3G) communications. This makes GSM a very cost-effective option for accessing remotely deployed measurement devices.

Research-orientated energy loggers are typically expensive and bulky products not suitable for field deployment and don't typically incorporate GSM remote communications capabilities. Products that do have remote communications are often aimed at energy metering for billing purposes and therefore do not allow flexibility in their function or user-access to their software to support research applications.

Outside the scope of load analysis applications there are several areas of research activity where a low-cost remotely-accessible data gathering device capable of measuring power could be useful. In addition to solving the current load analysis problems experienced by the Load Research Group it was hoped that the proposed solution could be a tool to benefit other research conducted at the University of Cape Town and other research institutions.

1.3. Research Question

This research topic came from the University of Cape Town's Load Research Group's requirement for a logger design capable of providing better quality domestic energy consumption data and which can take advantage of modern communication options available to achieve cost-effective measurement and remote data retrieval.

The Hypothesis is: can a cost-effective replacement for the existing logger be developed that will incorporate modern communications techniques in a flexible manner and aid in further research into power measurement and power analysis applications.

Requirements of the project were:

- To produce a cost-effective, compact hardware solution
- The capability to measure multiple consumers on single- or three-phase installations
- To measure RMS voltage and current as well as active, reactive and apparent power usage
- To time-stamp measurements with a suitable low-drift time source
- Flexible measurement intervals to support differing research applications
- Cost-effective remote and local data access options with modular communications interfaces to support future expansion
- The ability to function in urban and rural areas with or without remote communications access
- To offer a platform on which to support other research work at UCT

1.4. Research Proposal

To solve the research problem I proposed the design and manufacture of a laboratory prototype unit that could incorporate the project requirements into a field-testable solution and demonstrate key concepts of logger design.

To achieve this several key research areas were identified:

- Current energy measurement technologies and optimal solutions
- Measurement processing, accurate timekeeping and storage
- Communication options for local and remote data access
- Power supply design to allow a compact solution

These identified areas would be researched to find suitable solutions to the requirements and to select those which will best integrate into one product. From the design concepts a field-testable unit would be developed and measurement data gathered to determine if the instrument can be applied to non-intrusive load profiling.

1.5. Chapter Summary

The literature review in Chapter 2 considers the current approaches to load profiling and researches the necessary components required to achieve a laboratory prototype. The measurement standards required for load profiling and power quality analysis applications are discussed as well as the communications capabilities available for integration in the final field prototype.

Chapter 3: Power Profiler Concept Development uses the research findings of Chapter 2 to design the first prototype units which will be used for laboratory testing. For the first prototype mechanical integration is not considered, and the purpose is purely to test the functioning of the digital control and measurement circuitry. The second prototype presents the power supply design, mechanical integration and remote communications abilities. Basic hardware and software development and testing was performed during this prototype stage to verify correct circuit operation in preparation of the final field prototype development discussed in Chapter 4.

Chapter 4 presents the final Power Profiler field prototype design discussing the mechanical integration; control, power and communications board design; and assembly process. The costing of the final Power Profiler product is discussed to determine if it does achieve the objective of being a low-cost instrument for load and power quality analysis.

Chapter 5 presents the development of the embedded software in the Power Profiler and the PC-based Configuration and Control application. The PC application is developed in C# and makes extensive use of Object-Oriented Programming methodologies to support the re-use of developed control and interface classes in future applications.

The testing and calibration process is discussed in Chapter 6 where a Qualistar Power Analyser is used to calibrate and verify accuracy of the Power Profiler instrument. Once verified to be working correctly the Power Profiler is subjected to field-testing in Chapter 7 to further verify operational ability and to gather measurement data for load analysis. Several household appliances are identified from the measurement data recorded by the Power Profiler over a 43-hour period.

Chapter 8 presents the conclusions and discusses the future work to be undertaken to further develop the Power Profiler product.

Appendices A and B give the schematics and PCB design of the Power Profiler prototypes and field test units. The software listing of the embedded and C# developed code is given in Appendix C and the calibration and field testing results are available in Appendices D and E respectively.

Chapter 2

Literature Review

Energy profiling in the context of this dissertation is the measurement of voltage, current and power and representing it over a time period to create a high-resolution trend. Through analysis of the profiled power usage, it is possible to determine appliance types based on the power signature and from this consumer trends and future demand can be better predicted.

The use of energy profiling information to determine load types is termed Nonintrusive Load Monitoring (NLM) and was patented in August 1989 (patent number 4858141) by Hart, Kern and Schweppe of the Massachusetts Institute of Technology. Their invention describes an apparatus and method for monitoring the energy consumption of individual appliances within a residence without requiring intrusion into the residence [Hart, 1989].

The current generation of smart meters feature two-way communications with the utility provider allowing load management as well as energy usage profiling. These meters form part of an advanced metering infrastructure (AMI) solution that is being rolled out by utility providers in several countries but it appears that it is seen as more of an effective network management and billing tool, rather than a supporting platform for load research analysis. For example, Groenewald describes Eskom's approach to an integrated AMI solution with a focus on the management applications of AMI, but does not mention load profiling opportunities [Groenewald, 2008].

This lack of attention to load profiling is possibly because a metering system is designed for the purpose of providing billing information usually expressed in kilowatt-hours and measured over a fairly long interval. Load research analysis applications on the other hand, typically require higher-resolution measurement data captured over short time intervals which results in far higher data volumes to retrieve from the meter and process. This does not imply that a standard power meter with remote data retrieval could not be configured to produce higher-resolution data over time intervals down to one second, but these meters are usually part of a costly and complex measurement system that may only be practical for use by large utility companies.

At the heart of any AMI solution is the requirement for an instrument capable of energy measurement, data storage and remote communications for data retrieval. A back-end software system is required to retrieve and process the measured information for further analysis depending on the data application.

2.1. Energy Metering

The principle of AC energy measurement dates back to the 1800s with the first mass-produced ampere-hour meter designed by O.B. Shallenberger in 1888, with over 120 000 meters sold over the following 10 years [Dahle, n.d.]. Meter manufacturer Elster only stopped accepting orders for electro-mechanical three-phase meters as of 31 March 2008 and the replacement is an electronic meter that can still accommodate the same footprint but offer the advantage of supporting communication with AMI systems [Elster, 2012].

Modern energy meters use a solid-state metrology solution based on energy measurement ICs provided by several semiconductor manufacturers including Analog Devices [Analog Devices, 2012], Maxim/Teridian [Maxim, 2012] and Texas Instruments [Texas Instruments, 2012]. A single-phase solid state meter solution can be generalized as shown in Figure 2.1.

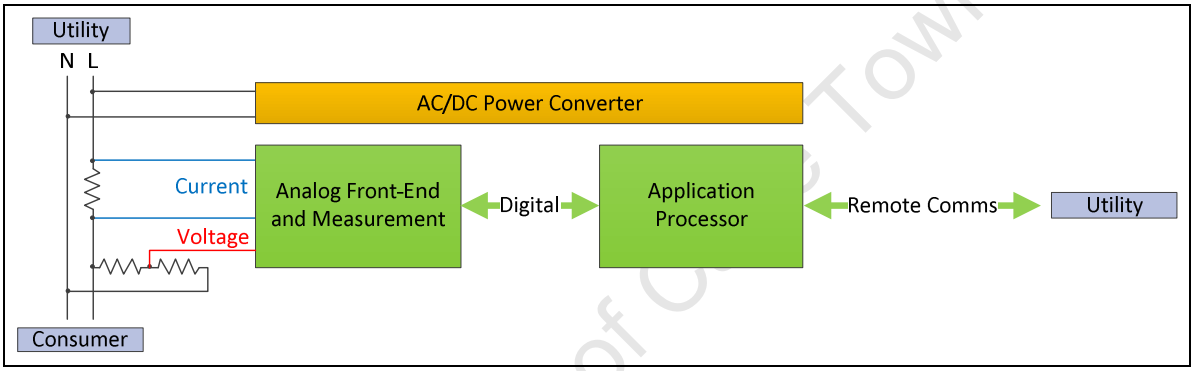


Figure 2.1 – Generalized Single-Phase Solid-State Meter Solution

In the analog front-end the current and voltage channels are simultaneously sampled with high-resolution ADCs and the RMS values calculated in software. Energy measurement ICs calculate instantaneous active, reactive and apparent power and accumulate these values in internal registers to represent accumulated energy [Analog Devices, 2011]. These measured values can be read by the application processor to update a LCD display or mechanical counter, or to report the data remotely to the utility provider.

A typical residential single-phase electronic energy meter such as the Itron ACE1000 282 measures active power only, with their more advanced products such as the ACE SL7000 measuring active, reactive and apparent power and supporting power quality monitoring [Itron, 2012]. While there are lower-cost meter solutions that can support remote communications, these products typically provide only limited power measurement and power quality information as they are intended for billing purposes.

The rising cost of electricity (particularly in South Africa) has created a market space where several companies have developed cost-effective energy profilers for domestic use. The intention is to make a household's residents aware of their energy usage habits and through this they can optimize their usage

to reduce their consumption and overall cost. An example of such a system is offered by Plogg and is designed to fit between the power socket and the appliance and will record the appliance's power usage for display on a computer [Plogg, 2010]. Google started their PowerMeter project in 2009 which used data supplied from utility companies and smart meter manufacturers to trend online consumer energy usage. This project was retired by Google in September 2011 after a trial period [Google, 2011].

2.2. Measuring Load Profile Data

Electrical loads have characteristic signatures which can be identified from the high-resolution power profile of a residential or industrial consumer. Power quality analyzers such as the Chauvin Arnoux C.A 8334B have the ability to measure power usage down to one-second resolution over long periods of time [Chauvin Arnoux, 2003]. Such a power quality analyzer is an expensive, bulky item, making it unsuitable for load profiling in security-risk areas and these units typically lack remote communication capabilities, making the data retrieval process labour-intensive.

An alternative to a dedicated power meter is to use a more general-purpose data logger such as the DaqPro 5300 [Fourier Systems, 2004] with the appropriate interfaces to allow it to measure AC voltage and current. The drawback with such an approach is that it would require very high sampling rates and large volumes of data to determine the power-factor of the load as these loggers typically do not measure phase information between channels. There are several power meter manufacturers that do produce metering products that could be adapted for load profile analysis applications, but they are generally expensive and limited in their user configurability.

The data logger currently in use for load profiling at UCT is the TRI ESS 5001 developed by Technology Services International, a subsidiary of Eskom. According to the Load Research Group this product has several drawbacks, including lack of remote communications ability, data corruption upon retrieval and the inability to measure three-phase four-wire distribution systems which are typically used for domestic power distribution. Although this logger has multiple current channels, only one voltage channel is measured making it unsuitable for power profiling when residences are supplied from a three-phase four-wire power distribution arrangement.

2.3. Measurement Standards

The IEC 61000-4-30 Testing and measurement techniques – Power quality measurement methods [IEC, 2003] specification defines several useful parameters that are relevant to a product such as the Power Profiler and are discussed below.

2.3.1. Classification and Measurement Accuracy

Measurement instruments are classified as either class A for precise measurement or as class B where low measurement uncertainty is not necessary. Measurements of the same signal with two different class A instruments will result in matching values within the specified uncertainty. Applications of class A instruments include verifying contractual obligations, verifying compliance with standards and resolving disputes. The measurement accuracy and aggregation methods of class B instruments are determined by the manufacturer and specified in the instrument documentation. Class B instruments are typically used for statistical surveys and trouble-shooting applications.

The basic measurement time interval for parameter magnitudes (supply voltage, harmonics, inter-harmonics and unbalance) is 200msec (10-cycles for 50Hz and 12-cycles for 60Hz power systems) for class A instruments. These measurement values are accumulated over three different time intervals:

- 3-s interval (150 cycles for 50Hz and 180 cycles for 60Hz)
- 10-min interval tagged with absolute time (e.g. 00:10:00) at the end of the aggregation interval
- 2-h interval aggregated from twelve 10-min intervals

Time clock uncertainty for class A performance devices should not exceed ±20ms for 50Hz or ±16,7ms for 60Hz and when external synchronization between instruments (e.g. through the use of GPS) is not available the time tagging tolerance must be better than 1 second per 24 hours.

Measurement accuracy is defined as uncertainty not exceeding:

- RMS voltage: ±0.1% of declared input voltage
- RMS current: ±0.1% of full-scale
- Frequency: ±0.01Hz

While the intention with this dissertation is not to produce a class A performance instrument, it was considered beneficial to apply the principles behind the IEC 61000-4-30 specification to the development.

2.3.2. Power Quality Measurement

Power quality measurement parameters are described in the IEC 61000-4-30 specification and while they may not be relevant in load profiling applications, they could form part of future power quality analysis applications.

Flicker

Recurrent variance of network supply voltage causes flickering of incandescent light sources which is visible to the human eye. The measured supply voltage fluctuations are processed using a model of the luminous flux versus voltage characteristic of the tungsten bulb and a model of the human reaction to

fluctuations of luminous flux, resulting in an instantaneous flicker measurement value [Hanzelka & Bien, 2005]. Flicker measurement in class A instruments is defined by the IEC 61000-4-15 specification.

Harmonics and inter-harmonics

Harmonic distortion of the voltage and current is a result of the operation of nonlinear loads and devices on the power system [Melhorn & McGranaghan, 1995]. The basic measurement of voltage and current harmonics and inter-harmonics is defined in the IEC 61000-4-7:2002 specification. Harmonics are measured in instruments such as the Qualistar C.A 8334B Power Quality Analyzer by sampling the voltage and current inputs and applying a Fast Fourier Transform to achieve harmonic binning [Chauvin Arnoux, 2003].

Transient voltages

The IEC 61000-4-30 specification defines a transient as a quantity which varies between two consecutive steady states during a short time interval (compared with the time-scale of interest), and a surge as a transient voltage wave propagating along a line characterized by a rapid increase followed by a slower decrease of the voltage. Transients are detected through several methods including comparative, envelope, sliding window, dv/dt threshold and RMS value via rapid sampling.

Surge protection devices are common in commercial and industrial products (including the measurement instruments themselves) and therefore measuring transient voltages in many environments is of limited use.

Voltage dips and swells

A voltage dip or swell is defined as beginning when the half-cycle RMS phase voltage ($U_{rms(1/2)}$) exceeds the dip threshold (typically defined as a percentage of the declared input voltage in low-voltage applications) and ends when the $U_{rms(1/2)}$ value is back within the dip threshold. The characterized value is the minimum or maximum $U_{rms(1/2)}$ value measured during the dip or swell and the duration of the dip or swell.

For poly-phase measurements the dip or swell can start on any one channel and end when all channels are back within threshold. Dip thresholds are typically in the range of 85% to 90% of the declared supply voltage for troubleshooting or statistical applications, and 70% for contractual applications. Swell thresholds are typically greater than 110% of the declared supply voltage.

Voltage Interruptions

A voltage interruption begins when any of the half-cycle RMS phase voltages ($U_{rms(1/2)}$) falls below a user-defined threshold and ends when all phase voltages return to within the threshold (with hysteresis).

The measured parameter is the duration of the voltage interruption with a measurement uncertainty of less than two cycles within the specified auxiliary power supply back-up time.

2.4. Communications

With the utility industry aiming for automated meter reading of electricity, gas and water usage, remote communication has become an essential part of the overall metering solution. Interfaces to energy meters are divided into two categories for discussion here: local access and remote access communications.

2.4.1. Local Access Communications

While several years ago RS-232 was the well-known choice for communications between computers and peripherals, USB has replaced this on modern consumer products. While USB does offer higher data speeds, it has the disadvantage of a complicated hardware and software interface. Semiconductor manufacturers Microchip and Silicon Labs (amongst others) manufacture self-contained USB to serial converter ICs with virtual com-port drivers allowing quick integration of USB into legacy products. USB has become so popular that many 16-bit microprocessors are now offered with integrated USB transceivers and supporting software libraries, making adding USB capability to any product simpler.

In applications where a multi-drop network is required RS-485 still offers a low cost solution that can support multiple nodes over long distances, and in very electrically noisy environments [National Semiconductor, 2002]. RS-485 is still widely used in industrial controllers, but is being replaced with newer technologies such as Industrial Ethernet, Control Area Network and Local Interconnect Network. RS-485's well-established use in industrial products, high data transfer rates (up to 10 mbps) and simple multi-drop capability still makes it a suitable choice for low-cost inter-device communication in industrial environments.

An infra-red interface is offered on a variety of energy meters. Entry-level meters such as the Itron ACE1000 282 have an IEC 62056-21 compliant infra-red interface [Itron, 2010]. This IEC specification defines the direct local data exchange protocol for meter reading, tariff and load control via optical and current-loop interfaces [IEC, 2002]. Optical interfaces have the benefit of providing electrical isolation from the measurement system for the operator and attached equipment.

Power-line communications is in use by several AMI solution providers to remotely access installed meters via a data concentrator. The European Telecommunications Standards Institute is developing (in conjunction with several AMI solution providers) the Open Smart Grid Protocol (OSGP) which will provide an open standard for metering integration across all utilities sectors [ETSI, 2012]. Echelon provides power-line based communication options and describes the three main challenges with this medium as attenuation, noise and channel distortion [Echelon, 2012]. To overcome these problems

complex software algorithms and spread-spectrum communication techniques are employed, but power-line modems still suffer from the limitation that their signals cannot pass through a power transformer.

Short-range RF solutions operating in the license-free ISM band are already used for local communication access to energy meters. Cooper Power Systems use a RF mesh-network operating between 902-928MHz which supports two-way communications for utility metering [Cooper Power Systems, n.d.]. Mesh networks offer the benefit of self-healing structures and can grow with the distribution network (e.g. as a new house is added the meter will act as a mesh node and extend the network). Digi International offers several mesh-capable ZigBee modules operating in the ISM band on 868, 900 MHz and 2.4 GHz frequencies along with a fully integrated smart management solution [Digi International, 2012].

The Bluetooth Special Interest Group is targeting Bluetooth for integration in the smart home market, touting it as the most successful short-range wireless technology used in the home with more than 2 billion Bluetooth enabled chips shipped every year. Applications for smart metering include user-awareness through display devices trending consumption and local access to meters by computers and tablets. The intention is to educate the homeowner as to their consumption habits in the hope that they will reduce their usage and ultimately save on their utility bill [Bluetooth SIG, 2011]. Bluetooth OEM modules are available from several manufacturers such as BlueGiga, KC Wirefree and Laird and offer embeddable devices capable of supporting serial data profile via a simple AT command UART interface.

WiFi communication is now replacing cabled Ethernet installations in many residential, office and industrial environments. Module manufacturer ConnectOne offers an integrated WiFi solution with UART, SPI or USB interfaces and supports an AT command interface for module control [ConnectOne, 2012]. This module also has a user-configurable web page which could be used to display measurement data to a homeowner when accessed through their residential WiFi access point. No commercial WiFi-based energy meters could be found, and this is possibly due to the more commercial nature of such a device and the previously high cost of WiFi modules.

2.4.2. Remote Access Communications

The public telephone network (PTN) is usable for remote access to an energy meter but is unsuitable for informal settlements or pole-top mounting where telephone lines may be unavailable. The householder might also not agree to the use of their telephone account for data retrieval. Embedded modems are available from companies such as TDK Semiconductor, Xecom, Zoom and Wintec [Microchip Technology Inc., 2005].

GSM data communications was first launched in 1994 with 2G data/fax capability and required a dial-up connection to be established in a similar manner to that of a landline telephone. With data rates of up to 9600 bps, and billing based on time connected and not actual data transferred it was a costly solution for

remote telemetry. With the first commercial release of GPRS services in 2000 packet-based data communications became available for telemetry applications [GSM World, 2011]. Data rates of up to 114 Kbps and billing based on data usage made GPRS a practical solution for telemetry applications. After GPRS followed the introduction of EDGE, 3G UMTS and now 4G LTE which offer significant improvements in data access speeds while still adopting a packet-based billing method.

GSM modem modules are typically used in embedded products because they offer compact size and integrated functionality suited for machine-to-machine (M2M) systems. Manufacturers such as Falcom, Sierra Wireless, Telit, u-Blox and Wavecom offer a variety of GSM modems ranging from GPRS to 3G speeds.

From the data sheets available from the GSM manufacturers, the communications interface to the modem is typically via a UART. The u-Blox LISA-U1 3.75G modem supports high-speed access via USB and SPI interfaces, but retains a UART for backward compatibility [u-Blox, 2010]. The control interface is typically via a standard AT command interface, with enhanced functionality used where necessary.

2.5. Conclusions

Although Nonintrusive Load Monitoring (NLM) has been around since 1989, utility providers today do not appear to emphasise the advantages that advanced metering infrastructure (AMI) and smart grid solutions can have for gathering effective load profile data. Modern energy meters are designed with solid-state metrology solutions and incorporate energy measurement ICs to achieve high accuracy measurement of voltage, current and active, reactive and apparent power. Two-way communication is often available and intended for integration with AMI systems, but usually these systems are costly and use proprietary protocols.

While general-purpose data loggers can be used for load profile capture, they often lack the functionality required for high-accuracy power measurement applications. The current profiling solution used by the UCT Load Research Group suffers from several problems which can now be addressed through the use of modern measurement solutions and technologies in a new measurement system. The specification of class A instruments described in IEC 61000-4-30 can also be implemented where possible in the new measurement system to create scope for future development work in power quality applications.

The wide deployment and cost-effective packet-based data access of GSM modems provides a suitable platform for remote access to load profiling devices. Short-range communications options such as Bluetooth, WiFi and ZigBee offer a good solution for low-cost short-range access to the measurement system while also providing electrical isolation to the user and their equipment.

Chapter 3
The Power Profiler Concept

This chapter takes the key concepts discussed in Chapter 2 and integrates them into a product called the Power Profiler which can be used to test the hypothesis: can a cost-effective replacement for the existing logger be developed that will incorporate newer communications techniques in a flexible manner and aid in further research into power measurement and power analysis.

To exceed the capabilities of the existing TSI logger the Power Profiler needed to offer:

- Measurement of more than eight current channels
- Measurement of more than one voltage channel
- Measurement of active, reactive and apparent power
- Measurement processing and reliable storage
- Reliable communication solutions
- The ability to support future research applications

The proposed Power Profiler hardware concept consisted of two main components: the Control Board with the measurement, processing and communications functions on it; and the Power Board containing the necessary AC/DC conversion and battery backup capabilities. The technologies identified in Chapter 2 must first be applied to defining the capabilities of the Power Profiler to achieve the project objectives.

3.1. Measurement Capabilities

Until the availability of energy measurement System-on-Chip (SoC) products a solid state energy meter or research logger (such as the currently used TSI device) would sample voltage and current using an ADC and process the measured waveform to provide the RMS values which could then be stored for profiling. A disadvantage of this approach is that it is very processor-intensive and is not easily scalable to accommodate more input channels.

Energy Measurement

As the SoC solution is a highly competitive market space there is little to distinguish the products between the different manufacturers. Analog Devices developed some of the earliest electronic energy metering SoC products and has already deployed solutions in over 75 million electricity meters [Wan, 2003]. Analog Devices were also one of the earlier companies to offer multi-phase energy measurement solutions in a single chip which would be beneficial to load profiling applications.

Analog Device's current range of poly-phase energy measurement products are shown in Table 3.1.

Table 3.1 – Analog Devices SoC Product Offering [Analog Devices, 2012]

Device	Measurement Parameters	Current Sensor Interface	Output Interface
ADE7880 ADE7854	Vrms , VA, VAR, Watt	CT, Rogowski Coil	HSDC, I^2C, Pulsed, SPI
ADE7858 ADE7868 ADE7878	Irms, Vrms, VA, VAR, Watt	CT, Rogowski Coil	HSDC, I^2C, Pulsed, SPI
ADE7752A ADE7752B ADE7762	Watt	CT	Pulsed
ADE7758	Irms, Vrms, VA, VAR, Watt	CT, Rogowski Coil	Pulsed, SPI
ADE7754	Irms, Vrms, VA, Watt	CT, shunt	Pulsed, SPI

At the time of designing the Power Profiler the devices shown in the table in red were the only suitable SoCs available from Analog Devices. The ADE7758 was selected because it provided all the measurement parameters required for load profiling and has an SPI interface for access to measurement and waveform data. Integrating four ADE7758 energy measurement ICs in the Power Profiler will allow the measurement of 12 channels of RMS current and voltage and their active, reactive and apparent power values.

Electrical Interface – Current

The ADE7758 has differential voltage inputs with a range of ±0.5Vpk for measuring current. Typical low-cost smart meters use a low-resistance shunt that is placed in-line to the load and has a fixed measurement range based on the resistance value [Analog Devices, 2011]. While this is a suitable low-cost solution for general energy measurement, it does limit the measurement range and is therefore unsuited to research-orientated applications.

Rogowski coil Hall-effect current sensors are designed for high-bandwidth measurements and use a measure of the magnetic field in a conductor to generate a relative output current [Shepard & Yauch, n.d.]. They are more expensive than a current shunt or current transformer (CT) and often require an external power source for their internal amplifiers. They do have the advantage of being lightweight as they have no magnetic core and are flexible enough to accommodate large diameter conductors or bus-bars. The ADE7758 energy measurement SoCs have an internal integrator for use with Rogowski coils.

CTs offer greater flexibility than a resistor shunt and lower cost than Rogowski coils. They are available in several primary/secondary turns ratios and by selecting a suitable burden resistor they can be matched to the current measurement range required. Clamp-on split-core CTs also allow for non-permanent installation, aiding measurement flexibility and can prevent unnecessary supply disruption during installation or replacement. Because of these reasons and their easy availability, CTs were selected for use with the Power Profiler.

Electrical Interface - Voltage

The ADE7758s have a single-ended input with ±0.5Vpk range for measuring voltage. A low-cost resistor voltage-divider attenuation network will be used as recommended in the Analog Devices ADE7758 application note [Analog Devices, 2011]. Voltage transformers were also considered but they would increase the overall solution cost and calibration complexity. The three voltage signals (L_1, L_2, L_3) will be attenuated via a resistor-divider and multiplexed to the four energy measurement ICs using Maxim MAX350 SPI-controlled analog multiplexers. The multiplexers will allow any of the three input voltages to be sampled together with any of the input currents to achieve the phase matching that is missing from the current TSI logger.

Industrial Inputs

To add measurement flexibility to the Power Profiler eight industrial input channels with a range of 0..15V were added to the concept design. These channels are intended for DC level measurement by the ADC in the control processor and will allow external parameters such as temperature or load state to be profiled along with the measured power data.

Temperature

The Power Profiler will have an internal digital temperature sensor to allow the measurement of ambient temperature which could be used to relate seasonal characteristics with measured profile data. A Maxim/Dallas DS1631A I^2C interface sensor with an accuracy of ±1°C over a range of -10°C to +85°C was selected [Maxim, n.d.].

Statistical Measurement Processing

A typical logger reports only the average value measured during the measurement interval. The Power Profiler will also record maximum and minimum measurement values for more advanced analysis. It is expected that with the maximum current and power statistic the in-rush current of electrical motors can be detected which will assist in load-profiling applications. Voltage swells and dips should also be detectable which will be beneficial to power quality analysis.

Timekeeping

The current time-of-day information used to time-stamp measurements will be maintained by a battery-backed real-time-clock. A Maxim/Dallas low-drift real-time-clock with internal crystal oscillator was chosen as the clock for use in the Power Profiler. This real-time-clock offers a high accuracy which will allow the Power Profiler to be operated in the field with minimal time drift between instruments. A GPS is also valuable to synchronize measurement between loggers deployed in the field, but this was not focused on at this stage due to the higher cost and the requirement for integration of a GPS antenna into the Power Profiler product.

3.2. Processing, Memory and Control

An advantage of using a SoC solution for calculating the RMS and power measurements is that the controlling processor for the Power Profiler need not be a high-speed DSP product and a more cost-effective microcontroller can be used. As the Power Profiler is intended as more of a general research-orientated tool, additional signal processing functionality such as single-cycle multiply-and-accumulate hardware may be beneficial.

Processor Options

When selecting a microprocessor the silicon cost must also be considered along with the cost of the software and hardware development tools. Analog Devices family of digital-signal processors was considered but they require a costly programming environment and hardware emulator for development. Their more complicated parallel-instruction processor core may also limit ease of programming for future research applications.

Microchip Technologies offers a 16-bit Digital Signal Controller (DSC) family aimed at mixed-signal applications such as power-factor correction, motor control, sensor systems and power electronics. This processor met the requirements for the Power Profiler application as it offered basic signal processing functionality within a 16-bit core that was simple to program and is supported by a free development environment, C compiler and application libraries. A low cost emulator/programmer was also available to assist with development. The dsPIC30F6013 was selected based on its 30 MIPS operation ability and integrated UARTs, SPI and I^2C interfaces. The internal Flash and EEPROM memory will also allow embedded code and configuration parameters to be stored in the processor, reducing overall complexity of the Power Profiler concept.

Memory Requirements

The Power Profiler will support the independent aggregation of measurements using a second-, minute- and hour-based measurement interval. By offering all three measurement intervals that can accumulate independently in one product it will be more beneficial to load analysis and power quality applications in that the operator can decide after the measurements have been gathered which of the three intervals to retrieve for analysis. For example, to analyze load behavior the second-based measurement aggregation method offers the highest resolution to make it easier to distinguish individual loads, whereas the hourly data is useful for determining long-term power trends. Once a residence is profiled the operator can decided not to upload the second- and minute-based measurement profiles which will substantially reduce the amount of data uploaded and reduce the GSM operating costs (24 hourly vs. 28 800 3-sec measurements in one day). Likewise, if an item of interest is identified in a minute- or hour-based profile the operator can then decide to download the higher resolution data for further analysis.

To determine the storage memory requirements the maximum measurement size needs to be calculated. Each stored measurement has an additional byte for flags (which could be used in future applications for event marking) and a packed timestamp of the measurement time-of-day from the real-time-clock. The ADE7758 ICs provide RMS with 24-bit resolution and power and frequency with 16-bit resolution so the maximum size per measurement record can be calculated as shown in Table 3.2.

Table 3.2 – Maximum Measurement Record Size

Parameter	Size (bytes)	Quantity	Total (bytes)
Measurement flags	1	1	1
Timestamp (packed)	4	1	4
Voltage RMS (3 channels)	3	9	27
Current RMS (12 channels)	3	36	108
Active Power (12 channels)	2	36	72
Reactive Power (12 channels)	2	36	72
Apparent Power (12 channels)	2	36	72
Frequency	2	3	6
Industrial Inputs (8 channels)	2	24	48
Temperature	1	3	3
Total bytes required per stored measurement			**413**

It was assumed that to perform a suitable load analysis, two-days of high-resolution 3-second data would be required. This would require:

$$\frac{60}{3} * 60 * 48 * 413 = \frac{23\ 788\ 800 \text{ bytes}}{1024 * 1024} = 22.687 \text{ MB}$$

Three Atmel AT45DB642 8MB flash memories resulting in 24 MB of storage memory would be capable of recording two days of 3-sec measurement data. The use of discrete Flash ICs over a MMC or SD storage card was done to reduce the risk of theft of the Power Profiler purely for removal of the memory card.

A 512KB SRAM memory is also provided for general processing of measurement data and any other future applications requiring high-speed memory access. The SRAM memory is interfaced with the dsPIC processor via an 8-bit interface with address decoding provided by three 74HC373 octal latches.

An additional two Microchip 512Kb I^2C EEPROMs are also added to the Power Profiler to ensure sufficient EEPROM memory for calibration and configuration parameter storage.

LCD Interface

A 4-line by 20-character LCD module will display basic measurement data to the operator to assist with verifying a valid field install. Voltage, current, power and system status information will be provided on several screens that the microprocessor will rotate through automatically.

3.3. Communication Interfaces

The literature review discussed several local and remote communications interfaces. An early concept design of the Power Profiler system consisted of several Power Profilers connecting via a short-range local RF link such as ZigBee to a local data gathering device called the Communicator. The Communicator would buffer the data and transmit it to the host server when necessary. This model is very similar to that of AMI systems using power-line communications and the intention was to reduce the number of GSM modems required to reduce overall unit and operating cost. The drawback of such an approach is that in a rural area where few Power Profilers are required with (possibly) large distances between them, each installation would require both a Power Profiler and a Communicator which would increase installation cost and maintenance requirements.

The solution to this problem was to incorporate the Communicator functionality into the Power Profiler and produce only one product that would include wired, short- and long-range RF interfaces. To increase flexibility and to support future communications technologies the RF interfaces will be implemented as plug-in modules. Standard serial UART interfaces will be provided between the microprocessor and RF boards to ensure simple integration and support of the common AT command standard over serial communication.

Wired Communications

USB offers the best solution to interfacing with modern desktop and portable computers as serial ports are typically not available on newer desktop and laptop computers. A Silicon Labs CP2101 USB to UART transceiver will be used to implement the interface as the dsPIC processor does not have an integrated USB interface. Virtual com-port drivers will be used on the host computer to simplify the programming interface to the Profiler.

To offer future support for intelligent sensors and to allow multiple Power Profilers to connect via a multi-drop network an RS-485 interface will also be provided.

Short-Range RF Communications

Although ZigBee is targeted at AMI applications, Bluetooth was considered a better option for the Power Profiler because the majority of laptop and cellphones available have integrated Bluetooth transceivers making interfacing with the Power Profiler simpler for the majority of operators.

The KC Wirefree KC-11 Bluetooth module was chosen because it has a line-of-sight range of up to 100 metres and implements the serial port profile (SPP) required for serial-cable replacement implementations [KC Wirefree, 2007].

Remote Communications

A Telit GC864 GPRS modem module was selected for the Power Profiler. GPRS communications was chosen over 3G because 3G modems are more costly and the higher data rate is not considered necessary for a telemetry application such as this. The GC864 modem offers a compact form-factor, board-to-board connector interface, and integrated TCP/IP stack and firewall which will simplify the software interface implementation [Telit, 2008]. It is worth mentioning that Telit also offer modems in a more compact ball-grid-array package but this was not chosen because the assembly process requires the use of specialist surface-mount assembly techniques and this type of device is more suited to mass production.

3.4. Power Regulation

There were three main power consumers anticipated in the Profiler design: measurement and processing components; communications interfaces; and the battery charger. It was decided to leave the battery charger and final power supply design to later in the prototyping process so that the digital and analogue components and software could be concentrated on. Point-of-load regulation is used to satisfy the different voltage supplies required for the Power Profiler components.

From the concepts described in this Chapter a guideline as to what the power requirements are can be determined. The operating voltages and the nominal and peak current values for the measurement and processing components required were taken from the manufacturer data sheets' and are summarized in Table 3.3.

Table 3.3 – Component Current Consumption (rounded to nearest mA)			
Digital Components (+3.3V)	**Supply Voltage**	**Current Nominal**	**Current Max**
dsPIC Processor (3.3V supply) 20 MIPS	3.3V	50mA	50mA
DataFlash memory	3.3V	1mA	25mA
External EEPROM	3.3V	1mA	5mA
Real-time-clock (DS1339-33)	3.3V	1mA	1mA
SRAM memory	3.3V	3mA	15mA
Temperature Sensor (DS1631A)	3.3V	2mA	2mA
Total current requirement: +3.3V Digital		**58mA**	**98mA**
Digital Components (+5.0V)	**Supply Voltage**	**Current Nominal**	**Current Max**
ADE7758 Energy Measurement IC (x4)	5.0V	36mA	52mA
LCD display - Controller	5.0V	2mA	2mA
LCD display - Backlight	5.0V	95mA	130mA
Total current requirement: +5.0V Digital		**133mA**	**184mA**
Analogue Components (+5.0V)	**Supply Voltage**	**Current Nominal**	**Current Max**
ADE7758 Energy Measurement IC (x4)	5.0V	20mA	32mA
MAX350 multiplexers (x6)	5.0V	1mA	1mA
Total current requirement: +5.0V Analogue		**21mA**	**33mA**
Analogue Components (-5.0V)	**Supply Voltage**	**Current Nominal**	**Current Max**
MAX350 multiplexers (x6)	5.0V	1mA	1mA
Total current requirement: -5.0V Analogue		**1mA**	**1mA**
Total current requirement: Control Board		**213mA**	**316mA**

It is clear that the main power consumer is the +5.0V digital rail at 184mA, mainly because of the LCD backlight. The analogue components are relatively low current, particularly the -5.0V rail which is only required for the multiplexers. The ON Semiconductor MC33275DT family of 300mA low-dropout linear regulators was chosen for use in the digital and analogue +3.3V and +5.0V rails. These regulators offer a low-dropout voltage of 260mV at 300mA and have internal current and thermal limiting [ON Semiconductor, 2005] which will provide circuit protection if component failures occur.

To supply the -5.0V rail for the multiplexers a National Semiconductor LM2660 switched capacitor voltage converter with an output current limit of 100mA was selected. The LM2660 offers a very compact solution requiring only an external stabilization capacitor to operate as a supply inverter [National Semiconductor, 1999].

Microchip MCP1726 adjustable low-dropout linear regulators were selected to power the communications interfaces. This device has the benefit of being able to supply up 3A loads for short durations (termed a soft overload) making it well suited to the burst requirements of RF devices. Other advantages are fast transient-response time and a low dropout voltage of 220mV at 1A, resulting in low power dissipation and longer operation when powered from a battery [Microchip Technology Inc., 2007].

The Telit GC864 GSM modem (as for several GSM modems reviewed) operates from a nominal supply of +3.8V with burst currents of up to 2A for short durations. During idle operation the modem will typically require 150 to 500mA depending on network signal strength and functional state [Telit, 2009]. These high-current bursts are usually supplied by bulk-storage capacitors near the modem's power pins. For the first prototype the power for the GSM modem will be provided by a laboratory power supply to allow the current usage to be monitored, after which the MCP1726 will be integrated.

The KC-11 Bluetooth module operates from a +3.3V supply and consumes 10 to 30mA while idle with bursts of up to 210mA during transmissions [KC-Wirefree, 2007]. A comparison with Bluetooth and WiFi modules from other manufacturers shows a similar operating voltage and current consumption not exceeding 300mA. The MCP1726 will be well suited to power the RF short-range interface as it can supply more than the required current and has an adjustable output.

3.5. Prototype One – Component Integration

The objective of the first prototype was to test the integration of the selected components for control, communication and measurement on one PCB. To reduce design risk and implementation time the switch-mode power supply and mechanical integration was not focused on at this stage. A block diagram of the Prototype One design is shown in Figure 3.1 with full schematics and PCB layouts given in Appendix A.

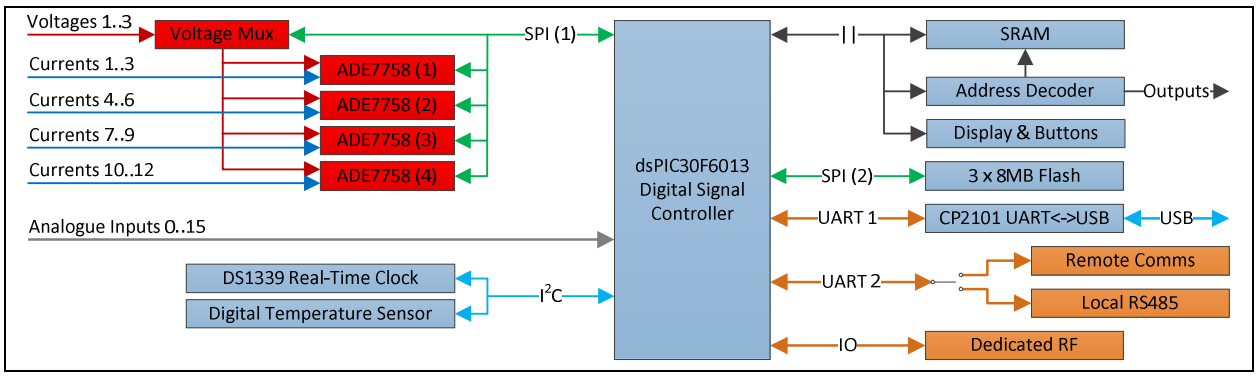


Figure 3.1 – Prototype One Block Diagram

The input voltage was regulated via several low-dropout linear regulators to +5V and +3.3V, and this voltage was then inverted to -5V and -3.3V for the analogue components using National Semiconductor LM2660 switched-capacitor voltage converters. This method of one supply voltage to the Control Board and localized regulation simplified the switch-mode power supply design requirements. To interface the dsPIC processor operating at +3.3V with the energy measurement ICs and voltage multiplexers operating at +5V, Maxim MAX3387 bi-directional level translators were used for the SPI bus, and 74LCX244 octal buffers with 5V-tolerant inputs were used for the interrupt signals.

The PCB for Prototype One was designed as a four-layer board with internal power and ground planes. A 3D rendering of the PCB is shown in Figure 3.2.

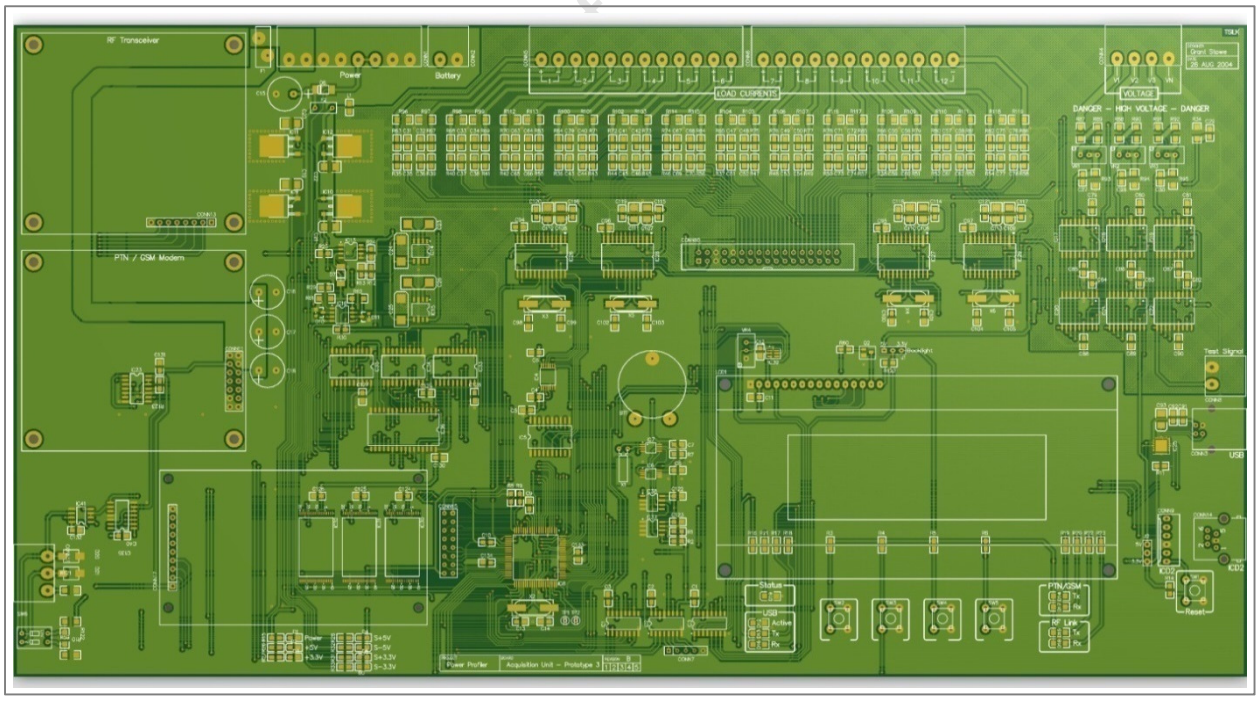


Figure 3.2 – Prototype One PCB 3D Rendering (Not To Scale)

The communications interfaces for remote and local RF are implemented as the two rectangular board interfaces on the top left of the PCB. Connectors are provided for DC power and the electrical signals.

Basic embedded software was developed to test the function of the components and their interfaces, not to necessarily implement their full functionality, but more to test that the interfaces would work. From the testing of this prototype the following were identified:

- The LCD display is a slow device to control requiring delays between display update commands which would impact on the measurement acquisition in the dsPIC unless a background updating process is implemented
- The LCD operates from +5V DC whereas the dsPIC is operating at +3.3V and a read from the LCD will exceed the maximum input voltage of the dsPIC I/O pins (although this should not damage the device, it is not correct operation and should be corrected through level translation)
- The burden resistors should be placed external to the Power Profiler to improve installation flexibility
- Bluetooth should be used for the short-range RF communications and will require a dedicated UART interface to the processor and not an I/O implementation as currently implemented
- The dsPIC processor, SRAM and DataFlash memories worked as expected

The circuit design was modified to accommodate the changes identified in the first prototype and this was used as the basis for the design of the second prototype.

3.6. Prototype Two – Product Integration

In the second prototype the design was split into two main components: the Control Board with the measurement and control functionality on it, and the Power Board with the AC/DC converter and backup supply components. To house the Power Profiler an OKW A 94 13 341 ABS plastic shell-type case was selected and is supplied as a split lid and base in several overall heights ranging from 45mm to 91mm. The Control Board and signal connectors would be housed in the larger lid-portion of the enclosure and the power supply in the base as shown in Figure 3.3.

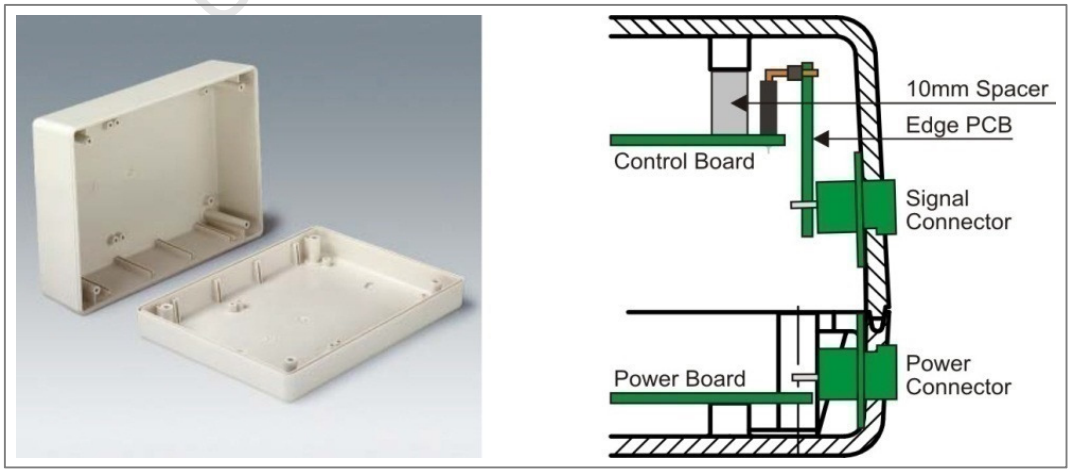


Figure 3.3 – OKW Enclosure and Connector Mounting

Having several base heights available would allow the overall solution height to be modified to accommodate the power supply design which at this stage was not complete. Changes to the power supply would have no effect on the Control Board and signal interfaces as these are mounted in the lid. The mechanical integration aspects of the second prototype are discussed in detail in Chapter 4: Profiler Design and Assembly as they did not change between this prototype and the final Power Profiler design.

An important circuit design change in prototype two was to replace the CP2101 USB-to-UART converter with a Microchip PIC18F4550 16-bit processor with integrated USB interface. This created additional I/O for measurement and control, and this processor now controls the LCD module to better optimize software operation of the dsPIC. The PIC18F processor operates from +5V and is interfaced with the +3.3V dsPIC via level shifters on the SPI interface. Direct sampling by the dsPIC of the voltage input channels was also added to support future development. The block diagram for Prototype Two is shown in Figure 3.4 below. Schematics and PCB layouts are very similar to the final Power Profiler design given in Appendix B and are therefore only referenced here for discussion. The final product design is discussed in detail in Chapter 4.

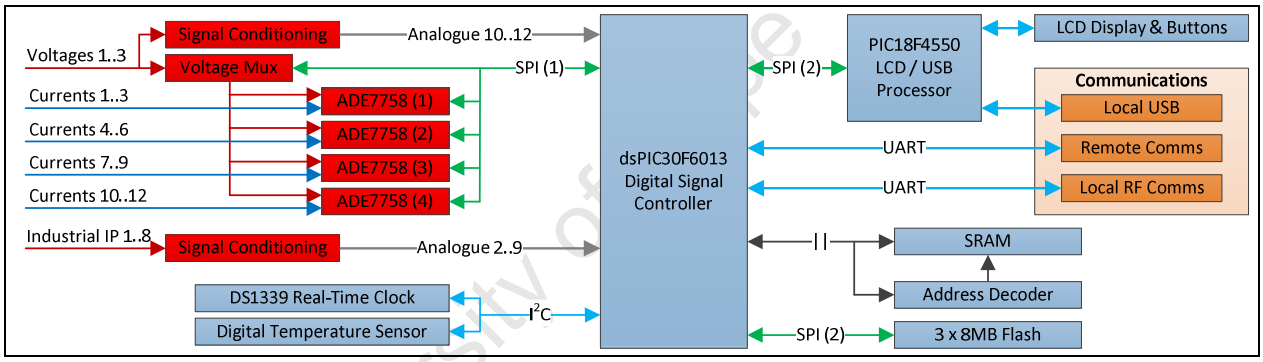


Figure 3.4 – Prototype Two Block Diagram

The embedded software developed for the first prototype was modified to test the additional functionality and new firmware was developed for the USB/LCD processor. The following design issues were identified from Prototype Two:

- Direct measurement of the current channels by the dsPIC is necessary and will be integrated in the final product design after testing on a breadboard
- There were four control buttons for the LCD display / menu which were seen as a possible tamper points and were requested to be removed
- Configuration will be done via an available communications interface and the LCD will provide measurement / state information only
- By powering the dsPIC from +5V instead of +3.3V the level-shifters can be removed thus reducing interface complexity and overall cost
- The current Dallas/Maxim DS1339 real-time-clock will be replaced with a Dallas/Maxim DS3231S with integrated temperature-compensated crystal with higher long-term accuracy

A prototype AC/DC switching power supply was designed for testing with Prototype Two. The SMPS design was implemented using an ON Semiconductor NCP1200A controller with final regulation provided by a SEPIC converter to provide a stable +5.5V output with 1.5A capability. A 9V rechargeable NiMh battery was used for power backup during dips and outages and charging was implemented using a Microchip PS200 battery charger IC.

This power supply prototype was discarded before manufacture in favor of a Power Integrations SMPS controller design and Lithium Polymer battery backup circuit described in Chapter 4. The decision for this was based on the Power Integrations products offering integrated high-voltage MOSFET switches which simplified the circuit design (and reduced the BOM) and excellent design software incorporating simulation and transformer design. Microchip discontinued the PS200 battery charger and it was replaced with better Lithium Polymer charge management controllers. The SEPIC converter was also replaced with a linear regulator with low dropout and the revised power supply solution resulted in a reduced overall height, simpler circuit design and smaller bill-of-materials.

3.7. Conclusions

In the first Power Profiler prototype the focus was on testing the control and measurement component interfacing and basic circuit operation. Point-of-load voltage regulation was used, supplied by a common 5.5V supply voltage which simplifies the requirements of the AC/DC power supply design. The testing of the first prototype verified the correct operation of the measurement and processing components and that the LCD display interface was slow and may have an impact on measurement processing ability of the dsPIC.

The second prototype allowed direct sampling of the voltage input channels by the dsPIC and included a secondary PIC18F4550 processor to control the LCD and implement the USB interface. Prototype Two also split the design into two PCBs: the Control Board and the Power Board. A prototype switch-mode power supply and battery backup circuit was designed for Prototype Two, but was replaced before testing with the final power solution to be described in Chapter 4.

Prototype Two was also the first design to be integrated into a plastic OKW enclosure. The Control Board and external signal connectors interface through three side-mounted PCBs integrated into the lid of the enclosure, and the Power Board is mounted into the base. This split design approach allowed the two main components of the Power Profiler to be independently developed without minor design changes impacting on each other.

Chapter 4

Power Profiler Design and Assembly

The Power Profiler design followed an iterative process starting with component integration testing, design into a suitable enclosure and finally circuit corrections to produce a concept product suitable for final field testing. The schematics, PCB layouts and bill-of-materials for the final Power Profiler design is given in Appendix B but extracts are shown in this Chapter to aid discussion.

4.1. Mechanical Integration

The Power Profiler prototype-two concept was the first design to be integrated into the selected OKW enclosure. The solution consisted of the Control Board, Power Board and several edge signal interface boards that link the external signal connectors to the Control Board as shown in Figure 4.1.

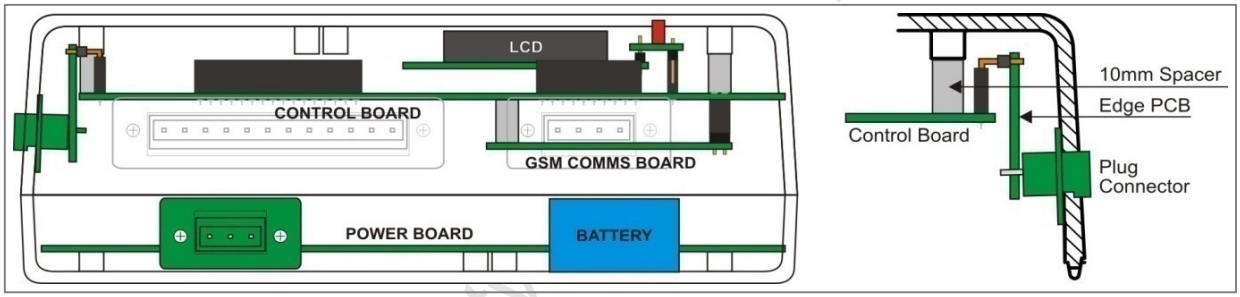


Figure 4.1 – Profiler Mechanical Overview

The Control Board contains the control and processing components and acts as the central interface between the edge signal interface boards, the communication boards, the LCD display and the LED status indicators. The signal interface boards also provide the surge protection circuitry to route surge currents away to the Earth instead of the sensitive electronics on the Control Board.

4.2. Control Board Design

A block diagram of the Control Board design (excluding point-of-load power) is shown in Figure 4.2. The main improvement over the prototype-two concept was that the dsPIC is now powered from +5V instead of +3.3V which simplified the design as the level shifters are no longer required. The DataFlash memories still operate from +3.3V, but they have 5V tolerant inputs allowing them to directly interface with the +5V SPI bus. The local RS485 interface was also added through the use of a Maxim MAX3100 SPI UART and an RS485 transceiver. Direct sampling of the current channels by the dsPIC is now also provided by the addition of amplifiers and multiplexers.

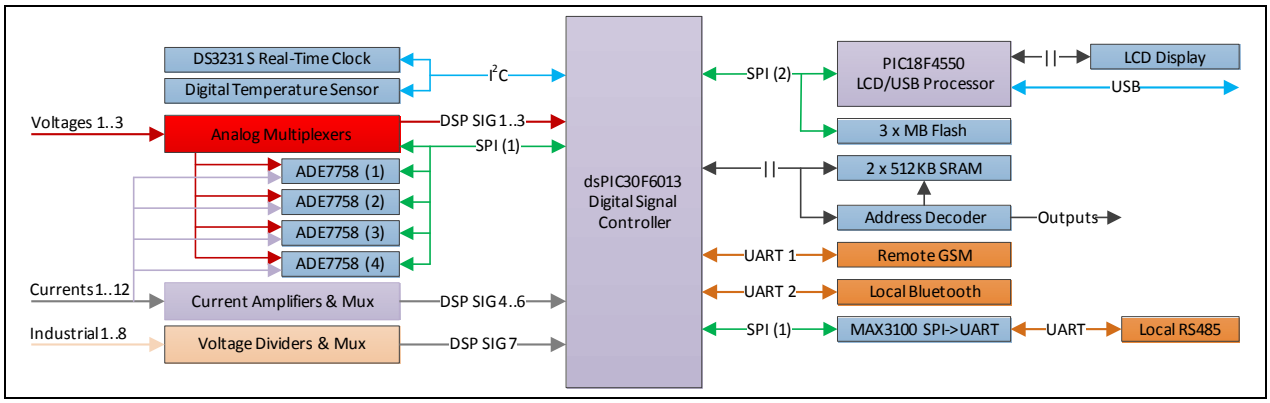


Figure 4.2 – Control Board Block Diagram

4.2.1. Analogue Signal Path

The external analogue signal interfaces to the Power Profiler consist of four voltages (L1, L2, L3 and N), twelve currents and eight industrial inputs. The voltage and current channels are not only routed to the energy measurement ICs, but are also conditioned and routed to the dsPIC for direct measurement and processing.

The recommended interface method for the ADE7758 energy measurement ICs taken from the Analog Devices evaluation board documentation is shown in Figure 4.3 [Analog Devices, 2003].

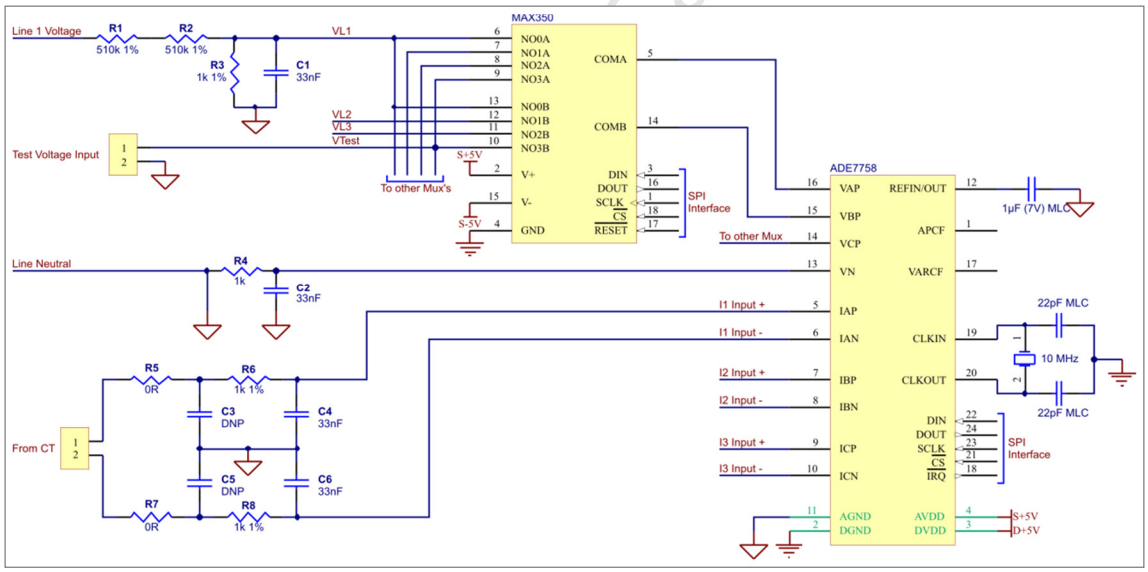


Figure 4.3 – Voltage and Current Channel Inputs

The three supply input voltages (L1, L2 and L3) are reduced by the voltage divider combination of R_1, R_2 and R_3 to the measurable range of the ADE7758 of ±0.5V. C_1 forms part of the anti-aliasing filter network recommended by Analog Devices and is necessary due to the sigma-delta ADCs used in the ADE7758.

The output voltage is calculated as follows:

$$V_{OUT} = V_{IN} * \frac{R_3}{R_1 + R_2 + R_3} = V_{IN} * \frac{1}{1021}$$

The use of two 510k 1% resistors in series instead of a single resistor provides protection in the case of a resistor failing closed-circuit which would result in the full line voltage potential becoming present on the MAX350 multiplexers. The output of the voltage divider is connected to inputs 1, 2 and 3 of the six MAX350 SPI-controlled multiplexers ($IC_{41..46}$), before connecting to the ADE7758 energy measurement ICs ($IC_{9..12}$). The MAX350s allow any input voltage to the Power Profiler to be connected to any input of the energy measurement ICs. The fourth common input to the multiplexers is connected to a test connector ($CONN_6$) for signal input during testing and calibration. The Neutral line is connected to analog ground and to the ADE7758 Neutral reference via an anti-aliasing filter (R_4 and C_2) as recommended by Analog Devices [Analog Devices, 2003].

To interface the voltage signals with the 12-bit ADC of the dsPIC, the voltage-divided signals must be amplified to maximize ADC resolution and offset by 2.5V DC. A Microchip MCP604 rail-to-rail operational amplifier is configured as a non-inverting summing amplifier with 3.3X gain as shown in Figure 4.4. The 2.5V offset voltage is provided by a Microchip MCP1525 high-precision voltage source.

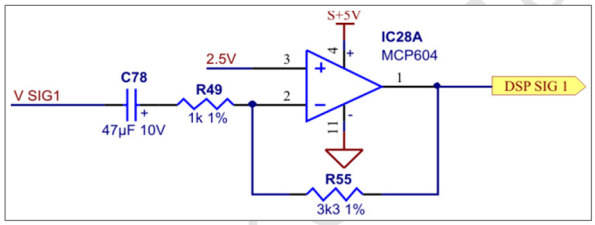


Figure 4.4 – Voltage Input Amplifier Circuit

The voltage inputs are protected against surge voltages with Littelfuse CH Series 275V surface-mount metal-oxide varistors (MOVs) and are rated capable of dissipating up to 8 Joules of energy [Littelfuse, 2009]. If additional voltage input protection is necessary it can be provided using external surge protection devices. Voltage transformers can also be used if the internal voltage dividers are adjusted to still achieve a full-range swing of ±0.5V on the input to the MAX350s.

The current channel inputs are designed according to the ADE7758 evaluation board documentation [Analog Devices, 2003] and are configured for current-transformer (CT) interfacing with a measurement range of ±0.5V. Current transformers are the preferred method of measuring current as they can be selected according to the optimal transfer ratio to maximize measurement resolution and are also not electrically coupled to the measurement source and therefore offer additional protection. $R_{6,7}$ and $C_{3,5}$ provide support for future interfacing with Rogowski coils for current measurement.

The current channels are differential inputs with a range of ±0.5V and after passing through an anti-aliasing filter (R_6, C_4 and R_8, C_6) are routed directly to the ADE7758 inputs. To interface the current inputs with the dsPIC the differential pair needs to be amplified and converted to a single-ended output biased to +2.5V

DC. This is achieved using an Analog Devices AD620 low-power instrumentation amplifier as shown in Figure 4.5.

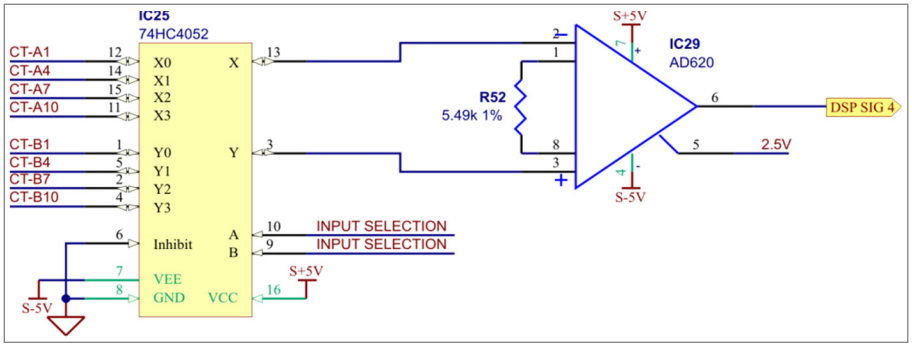


Figure 4.5 – Current Channel Amplifier Circuit

The gain is set by resistor R_{52} according to the equation [Analog Devices, 2004]:

$$G = \frac{49.9\text{k}\Omega}{\text{R}_{52}} + 1 = \frac{49.4\text{k}\Omega}{5.49\text{k}\Omega} + 1 = 9.998$$

The bias voltage of +2.5V offsets the amplified signal and the differential pair is selected through dual 4:1 multiplexers $IC_{25..,27}$.

The current channels use CTs with low-ohm burden resistors to convert measured current to voltage. To provide fast-response protection against transients coupled into the CT cables all inputs are fitted with 5V bi-directional Transient Voltage Suppressors (Tranzorbs) capable of dissipating up to 600W.

Please take note that a CT should **never** be left coupled to an active load without a burden resistor across the secondary. The potential voltage across an open-circuit secondary can reach very high levels which could result in electrical shock to the installer or damage to the measurement system. If the unit is to be temporarily disconnected and no burden resistor is available keep the CT secondary leads shorted.

The industrial inputs are voltage-divided to a +5V DC measurement level as shown in Figure 4.6. The dsPIC can measure one industrial input channel at a time using the 74HC4051 8:1 multiplexer.

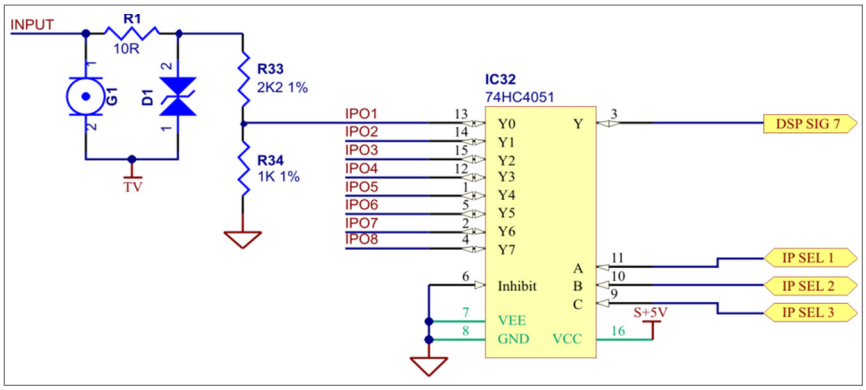


Figure 4.6 – Industrial Input Interface

The maximum measurable input voltage range (V_{IN}) is +15V DC determined using the voltage divider equation:

$$V_{OUT} = V_{IN} * \frac{R_{34}}{R_{33} + R_{34}} = V_{IN} * \frac{1100}{3300} = V_{IN} * 0.333$$

The Industrial Inputs can be subjected to high-voltage, high-energy transients because they may require long cable runs to connect the Power Profiler and sensor being measured. To protect against high-energy short duration and lower-energy long duration transients, both a bi-directional 16V Tranzorb (D_1) and a 75V gas arrestor (G_1) is used as shown in Figure 4.6. Gas arrestors are capable of dissipating large amounts of energy (2 500A with 8/20µs impulse discharge characteristic) but are not as quick to respond to transients as Tranzorbs. The inclusion of the 10Ω resistor increases the input impedance to the measurement circuit to attempt to 'force' the transient to Earth via G_1. Any resultant energy or fast transient that is too quick for the gas arrestor will be dissipated by D_1.

4.2.2. Digital Design

The digital components in the Power Profiler consists of a mix of high-speed digital processors, memory and communications interfaces and several mixed-signal devices such as the energy measurement ICs and multiplexers. This mix of analogue and digital components requires careful consideration to be taken during the PCB design stage to minimize interference from the noisy digital components coupling into the sensitive measurement circuits. To achieve this low-noise circuit design techniques are employed such as the use of a 4-layer PCB with ground and power planes, appropriate use of power supply decoupling capacitors, and by minimizing PCB trace lengths and ground loops.

Digital Signal Controller

At the heart of the Profiler is the Microchip dsPIC30F6013 digital signal controller. It operates using a 7.3728 MHz crystal and internal 16x PLL to achieve just short of 30 MIPS performance. It is powered by the D+5V rail and the internal ADC powered by the S+5V rail.

The dsPIC communicates with most external peripherals via an SPI interface capable of up to 10 Mbps speed depending on external device capabilities. SPI communications is split across two separate ports: SPI1 for measurement and sensor communications, and SPI2 for memory access and communication with the USB/LCD PIC18F4550 processor. This split-bus approach allows the measurement acquisition from the energy ICs to be able to interrupt the lower-priority operations such as DataFlash access and local USB communications. A I^2C interface is used to connect with the real-time-clock and digital temperature sensor.

The analog interfaces to the dsPIC's ADC (DSP SIG 1..7) connect with the voltage, current and industrial inputs through the relevant signal conditioning circuitry as described in section 4.2.1.

PIC18F4550 LCD/USB Interface

A PIC18F4550 USB microprocessor provides a cost-effective USB interface, controls the LCD module and provides several management functions for the Power Profiler. It operates at 12 MIPS using a 20 MHz crystal and is powered by the digital D+5V supply rail. The PIC18F4550 implements a USB V2.0 compliant Low (1.5 Mb/s) or Full (12 Mb/s) speed interface and has an onboard transceiver. Microchip's free USB software stack is used to implement a serial cable replacement profile for a PC to interface with the Power Profiler via a virtual serial (COM) port. The PIC18F4550 also performs some basic management tasks in the Power Profiler by controlling the power regulators, battery charger and monitoring key voltage levels. Through its onboard 10-bit ADC it monitors the point-of-load regulated voltages within the Profiler and can indicate fault conditions to the dsPIC when polled through the SPI interface.

A Crystal Clear Technologies CMC420L01 4 line by 20 character backlit LCD display with integrated controller displays basic measurement data and system state. The PIC18F4550 generates the measurement information screens and transfers them to the LCD module controller through a parallel 8-bit interface.

Real-Time-Clock

The Maxim/Dallas DS3231S real-time-clock (RTC) was selected because it has an internal temperature-compensated high accuracy 32.768 KHz crystal allowing it to achieve minimal drift over time. The RTC communicates with the dsPIC on I^2C address 104_d. Its square-wave output is configured for a 1Hz duty cycle which is used to synchronize a software-implemented time-of-day clock in the dsPIC. Power for the RTC is supplied from the D+5V supply rail and a 3V lithium battery provides a backup source to maintain accurate time-of-day when power to the Power Profiler is removed.

Digital Temperature Sensor

A Microchip TCN75-5.0 digital I^2C temperature sensor with a measurement range of -55°C to +125°C is used to provide temperature measurement to the dsPIC. It is interfaced to via the I^2C bus on address 72_d and operates from the D+5V supply rail. Measured temperature can be used to correlate demand usage with seasonal influences or extreme weather conditions.

Signal Selection and Energy Measurement

The ADE7758 energy measurement ICs and the MAX350 multiplexers are controlled via the dsPIC's SPI1 interface. The process of initializing and controlling the ADE7758s is described in Chapter 5: Embedded Software Development. The ADE7758s operate from both digital (D+5V) and analogue (S+5V) supply rails. The MAX350 multiplexers require a dual 5V analogue supply (S+5V and S-5V) which is provided using a LM2660 (IC_{17}) switched-capacitor voltage converter.

The MAX350 does not require a digital supply voltage, but does have individual analogue and digital grounds for correct signal referencing and noise isolation [Maxim, 1998].

Storage Memory Interface

The DataFlash devices offer both a serial SPI port capable of up to 20Mbps operational speed and an 8-bit parallel interface in the same package. To minimize PCB layout complexity the SPI interface was chosen for interfacing with the dsPIC. The DataFlash operates from the D+3V rail and has 5V tolerant interfaces allowing it to connect to the dsPIC's SPI2 bus without requiring level translation.

$CONN_{11}$ provides an expansion header to accommodate other memory types such as Multi-Media Card (MMC) or microSD cards if required in the future. During the Power Profiler development the DataFlash memory selected became more readily available in a smaller form-factor and to support this device, a small interface board was designed to interface via $CONN_{11}$ as shown in Figure 4.7.

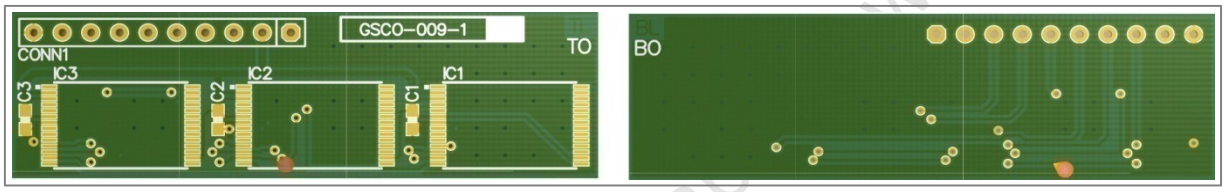


Figure 4.7 – DataFlash Memory Interface Board Top and Bottom Layout

SRAM Memory

Two BSI 512K x 8-bit SRAMs interface to the dsPIC via a parallel 8-bit interface with address decoding logic provided by three 74HC373 octal latches. Separate Chip Enable (CE) signals allow each SRAM to be accessed and their output enable (OE) and read/write (R/W) signals are shared to reduce the I/O requirements to interface to the dsPIC.

4.2.3. Communications Interfaces

The Power Profiler incorporates communication interfaces for local access using USB, RS485 and short-range RF (Bluetooth). Remote access is implemented on an interface board to support multiple communication interfaces through one interface. Remote communications is intended for GSM but Public Telephone Network (PTN) and multi-drop RS485 were also considered. Interface signals for PTN and RS485 access were routed from the front-panel connector ($CONN_5$) to the interface board via $CONN_3$.

USB Interface

The PIC18F4550 processor implements the USB interface and uses the Microchip USB software stack to implement a serial cable replacement USB profile. The software operation of the USB interface is described in detail in Chapter 5. A USB type B vertical connector is used to provide an interface on the front panel.

The Power Profiler does not optically isolate the USB interface and it is not recommended for use during measurements when the Power Profiler is connected to a high-voltage source. An external optically-isolated USB converter could be used to provide sufficient isolation.

RS-485 Interface

A RS485 multi-drop network is provided for the future connection of multiple Power Profilers and intelligent sensors. As the dsPIC does not have a spare UART available to drive the RS485 interface, a Maxim MAX3100 SPI to UART protocol converter is used with a SN65LBC184D RS485 transceiver, as shown in Figure 4.8.

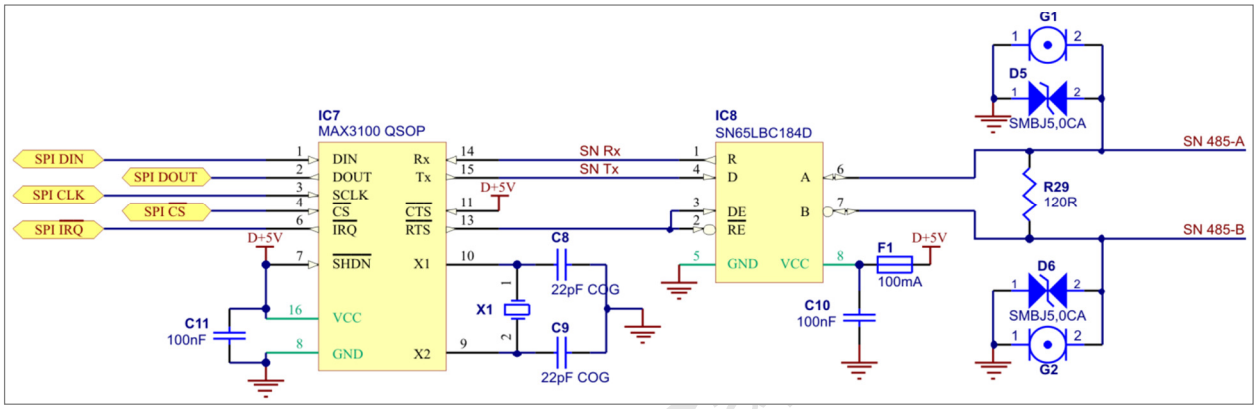


Figure 4.8 – RS485 Multi-Drop Interface

The MAX3100 provides SPI to UART conversion at a baud rate of up to 230 Kbps. Transient protection for the RS485 transceiver is provided by a 75V gas arrestor and 5V bi-directional Tranzorb. The 100mA fuse on the transceiver power rail prevents a surge if a transient protector shorts to the positive power rail via the transceivers' internal Schottky protection diodes.

Remote Communications Interface

$CONN_8$ provides a generic interface with serial UART and control signals to support a variety of remote communications interfaces such as GSM, PTN and RS485. The remote communications is implemented on a modular daughter-board using the interface connections shown in Figure 4.9.

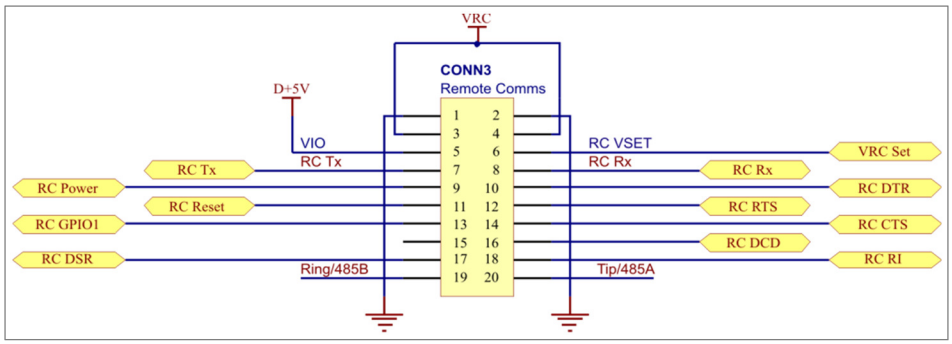


Figure 4.9 – Remote Communications Interface Connector

The standard RS232 interface signals provided from the dsPIC to the remote communications module are: Tx, Rx, DTR, CTS, DCD, RI and DSR. Support for PTN and RS485 connections is provided by routing the Tip/Ring and RS485 A/B signals from the front-panel connector to this interface. Power, reset and general purpose I/O lines are also provided. The 'VRC Set' signal allows a single resistor connected between this pin and ground to set the VRC supply voltage level of the MCP1726 linear regulator powering the remote communications interface board.

Wireless (RF) Communications

For the RF module interface the Tx, Rx, RTS and CTS signals are provided along with power control and reset signals as shown in Figure 4.11.

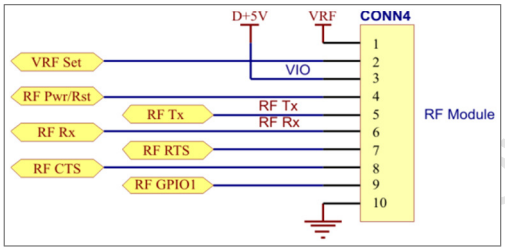


Figure 4.10 – Wireless (RF) Module Interface Connector

The 'VRF Set' signal allows a single resistor connected between this pin and ground to set the VRF supply voltage level for the RF modem as discussed in Section 4.2.4.

4.2.4. Localized Power

The input power to the Control Board consists of one +6V DC input supplied from the Power Board via connector $CONN_1$, a dual-in-line 10-way IDC boxed header. Using a single supply and localized power regulation allows regulators to be placed close to their loads to minimizing current loops and requires a simpler power supply design as only one main supply voltage is required. The power distribution arrangement of the Control Board is shown in Figure 4.11.

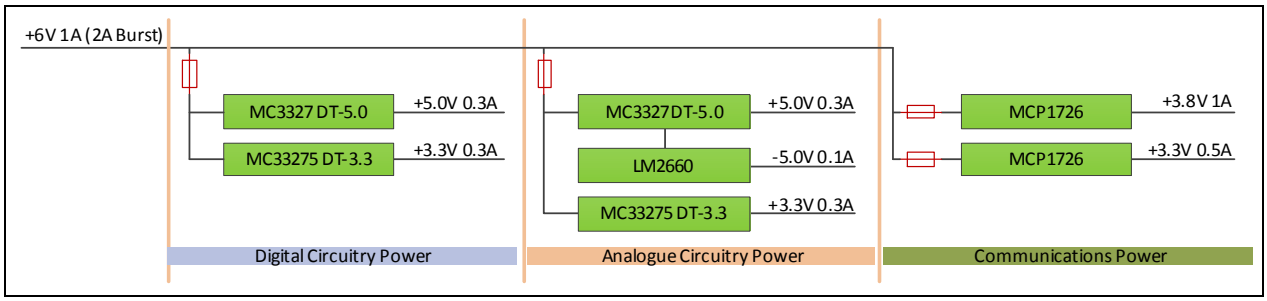


Figure 4.11 – Control Board Power Distribution

ON Semiconductor MC33275DT fixed-output 300mA low-dropout linear voltage regulators regulate the +6V input down to +5.0V and +3.3V for the digital components (D+5V and D+3.3V) and to +5.0V for the analogue components (S+5V). The MC33275DT is a low-dropout regulator (260mV at 300mA load)

resulting in less power dissipation and lower input voltage requirement which is useful to maximize battery capacity [ON Semiconductor, 2005].

The power for the Local RF and Remote Communications interfaces is supplied from Microchip MCP1726 adjustable 1A low-dropout regulators. The MCP1726 is capable of supplying up to 3A burst currents, making it well suited for powering products incorporating RF transmitters where high currents are present during transmission bursts. A single resistor on the interface card sets the output voltage of the regulator, making the supply flexible for new interface board designs.

Resettable fuses are used to supply the analogue, digital and communications interfaces. These voltage outputs are monitored by the processors to detect a fault condition and attempt to recover from it. The 10-bit ADC of the PIC18F processor measures the primary (V+), battery (VBATT), digital (VDIG), analogue (VSIG), remote communications (VRC) and RF communications (VRF) voltage levels. These values are monitored for out-of-range conditions and if a fault occurs, it is reported to the dsPIC which can disable measurements and if possible report the fault through remote communications to an operator.

4.2.5. PCB Design and Assembly

The Profiler Control Board was designed as a four-layer PCB with internal ground and power planes to minimize power loops and reduce radiated EMI. Extensive use was made of surface-mount devices to suit the design for future commercial production and to provide more optimal route layout as both sides of the PCB can be utilized for component placement. All ICs are also decoupled with 100nF ceramic capacitors to reduce EMI.

The Power Profiler required an extensive three-dimensional approach in designing the PCBs and defining the mechanical interfaces. In addition to selecting components based on functionality, mechanical constraints such as height needed to be considered. The bottom layer of the Control Board PCB houses the front-panel connectors, LCD and LED interfaces. An interface PCB board is used to connect the Control Board, LEDs and LCD module as they are assembled at differing heights as seen in Figure 4.12.

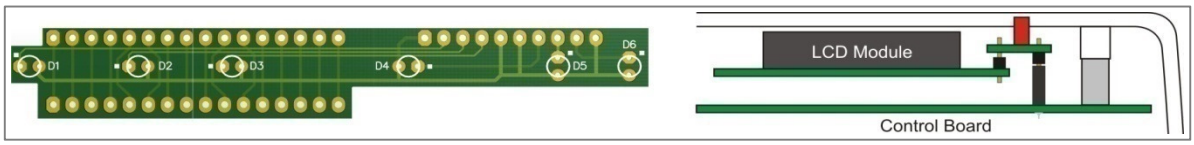


Figure 4.12 –LCD/LED Interface Board

The top layer of the Control Board (visible to the user when the enclosure is open) houses the components that need to be user-accessible such as the connectors for the remote and RF interface boards, power, test signal and ICD programming. The In-Circuit Debugger (ICD) connector is used for programming both the dsPIC and PIC18F processors.

The top and bottom layers of the control board are shown in Figure 4.13. The connectors shown in blue in Figure 4.13 connect the control and interface boards mounted to the sides of the enclosure lid.

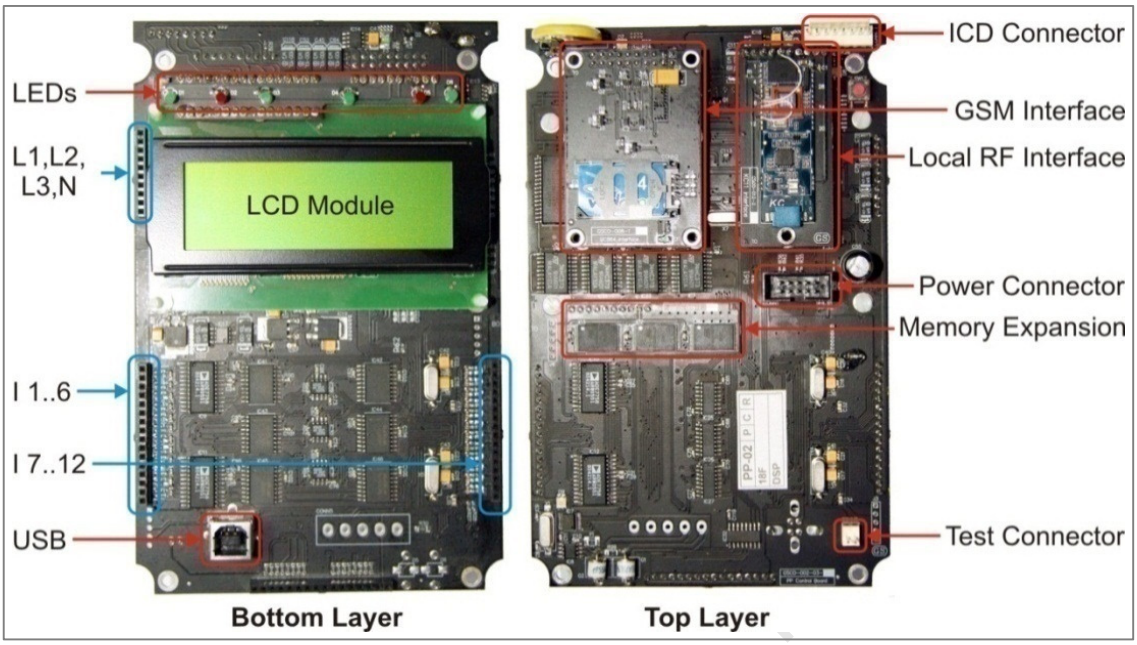


Figure 4.13 – Control Board Top and Bottom Layers

As the PCB is a mixed-signal design involving both high-speed digital circuits and low-amplitude analogue signals several guidelines were adopted for the PCB layout. Components were grouped as shown in the shaded areas of Figure 4.14: purple – power, red – communications, white – digital and yellow – signal measurement. Detailed designs are available in Appendix B.

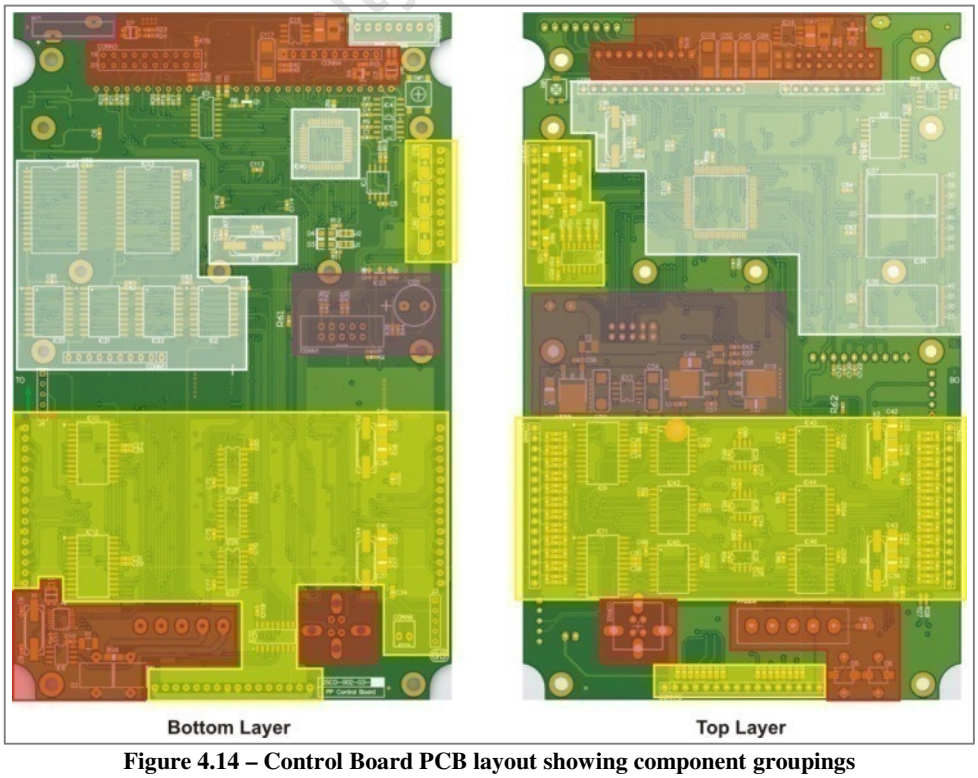


Figure 4.14 – Control Board PCB layout showing component groupings

Guidelines adopted for the PCB layout were:

- Separate (as much as possible) analogue and digital circuitry as is clearly visible in Figure 4.15 as the yellow and white separated areas of the PCB
- High-speed digital circuits are kept together: the processors and memory are contained within the top portion of the PCB away from the energy measurement components
- Communications interfaces are kept close to the processor to minimize trace lengths and therefore reduce radiated EMI
- The power components for the communications interfaces are kept as close to the connectors as possible to minimize power losses
- The main power regulation for the analogue and digital circuitry is kept close to the input connector to minimize power losses
- The inner layers (not shown in Figure 4.15) are used for power and ground distribution for both the analogue and digital power circuitry
- Analogue and digital ground planes were poured on all layers to minimize current loops to decrease noise and radiated EMI

The PCB design files were used to generate a bill-of-materials for the Control Board and the components purchased through local distributors. The PCB was hand-assembled as the low volumes required for the prototypes did not warrant the necessary cost to generate paste stencils for automated SMD assembly. The most complex components such as the dsPIC and other multi-pin high-density SMD ICs were assembled first, followed by the passive components and finally the through-hole components.

4.2.6. Control Board Testing

Basic testing was performed during assembly of the Control Board starting with verifying that the point-of-load regulators were supplying the correct voltage levels. As the embedded software was developed the board was further tested, culminating in the functional tests described in Chapter 6: Functional Testing and Calibration. The assembled PCB was visually inspected for solder shorts and cleaned with flux remover before embedded software development was started.

4.3. Power Supply Board

The main supply for the Power Profiler is an off-line AC switch-mode implementation using a Power Integrations TOP245P controller. Power can be supplied from either an 110V or 230V AC source and the SMPS produces two voltage outputs: one for powering the Control Board via the low-dropout linear regulator, and the other for powering the LiPolymer battery charger. The backup battery is a 7.4V 1850mAh Lithium Ion Polymer battery which can be enabled or disabled by the Control Board to prevent excessive discharge and subsequent battery damage.

The Power Supply Board block diagram is shown in Figure 4.15.

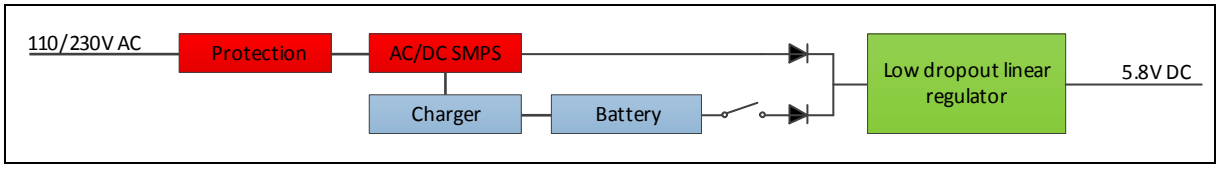


Figure 4.15 – Power Supply Block Diagram

4.3.1. Protection Circuitry

As the Power Profiler is installed in an environment where high-energy transients such as lightning can occur, power supply surge protection and EMI filtering is implemented as shown in Figure 4.16.

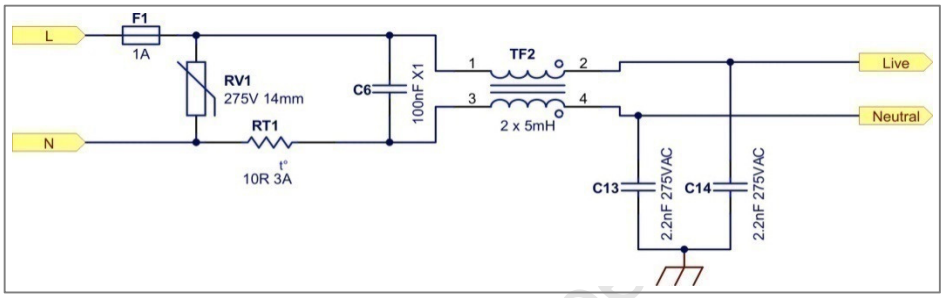


Figure 4.16 – Power Supply Input Protection

A 20mm 1A slow-blow glass fuse protects the utility supply in the case of a permanent fault condition in the Power Profiler. RV_1 is a $275V_{RMS}$ Metal-Oxide Varistor (MOV) that protects the input from surge voltages such as those from lightning or line faults. Thermistor RT_1 provides in-rush current protection for the fuse [Power Integrations, 2005]. C_6, TF_2, C_{13} and C_{14} form part of the EMI filter and the values chosen were recommended by the Power Integration design software. C_6 is termed an X capacitor and $C_{13,14}$ are Y capacitors according to IEC 60384-14. X and Y capacitors are connected directly to the line and are exposed to over-voltages and transients which can damage the capacitors resulting in electric shock or fire [EPCOS, 2009]. Because of this these capacitors are tested at manufacture and certified to comply with the necessary safety regulations. Y capacitors are typically only a few nano-Farad and the same value as C_5 in the SMPS design was selected to reduce the BOM.

4.3.2. Switch-Mode Power Supply

A Power Integrations (PI) TOP245 SMPS controller and custom transformer design converts the rectified AC input to a +6V 1A and +10V 1A outputs. The PI Expert Suite Software tool was used to design the SMPS circuit and select the necessary component values. The component values recommended by the design software were verified against the TOP245 data sheet and the circuit shown in Figure 4.17 was produced.

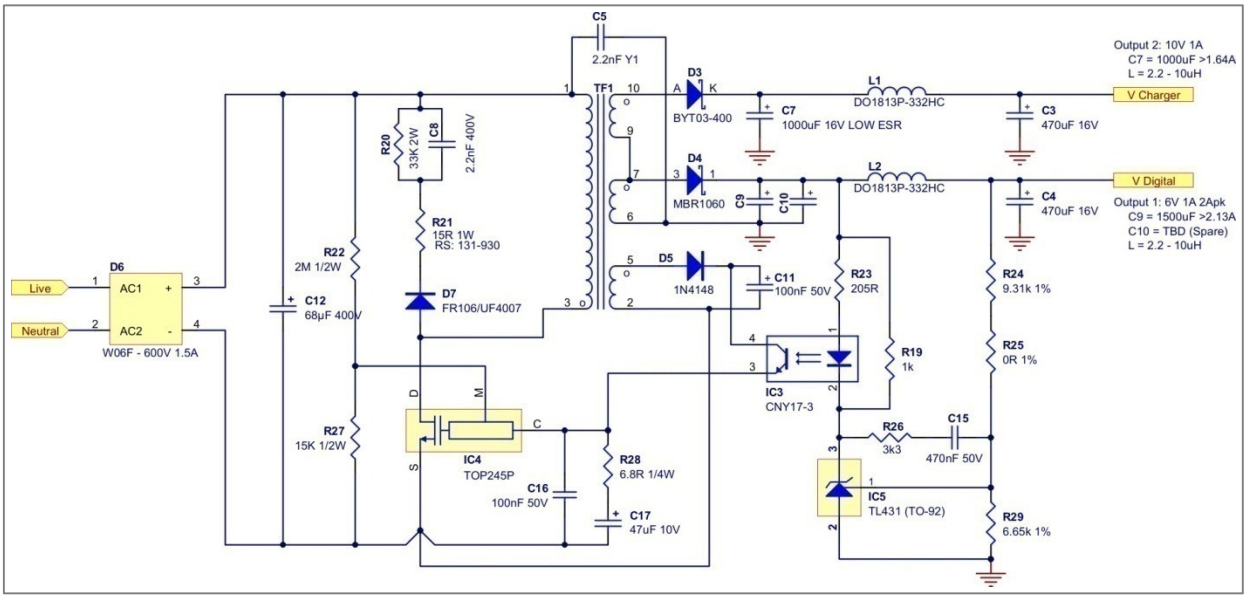


Figure 4.17 – Switch Mode Power Supply Circuit Diagram

The TOP245 operates at a switching frequency of 132KHz and provides over-voltage and under-voltage lockout, current limiting and uses frequency jittering to reduce EMI [Power Integrations, 2005]. The critical components of a SMPS design include the transformer design (TF_1), Schottky rectifier diode ($D_{3,4,5}$) current ratings, output inductor values ($L_{1,2}$) and the value and ripple current rating of the output filter capacitors ($C_{7,9,10}$). The PI design tool calculates the recommended values and allows the user to customize the design based on available items such as the transformer core, inductors and capacitors. Changes to these values update the design and a new simulation is generated. C_5 is used to reducing EMI and is a Y1-rated safety capacitor as the primary is at a high DC voltage and a capacitor failure would result in a dangerous primary/secondary shorting of the transformer.

In an effort to make the Power Profiler as compact as possible the selection of the transformer for the SMPS design was very important. The transformer core needed to be as shallow as possible to fit into the enclosure base, and based on the transformer designs recommended by the PI design software, a Ferroxcube E25/13/7 3C90 (EF25) E-core and 10-pin bobbin was selected. This bobbin and core provided a height solution of only 20mm.

R_{22} and R_{27} provide either line-sense or external current limiting for the TOP245 controller, dependant on which resistor is populated. For our application R_{22} is used to provide over-voltage and under-voltage detection with the 2MΩ value specifying an acceptable range of 100VDC to 450VDC.

$R_{20,21}$, C_8 and D_7 form a clamp to limit the peak drain to source voltage of the TOP245 to an (estimated) average of 174V. The bias secondary winding (TF_1 pins 2 and 5) produces a +12V half-wave rectified voltage source for the feedback opto-isolator IC_3. The PI design software calculates the necessary values for the output stage rectifier diode, inductor, and post-filter capacitors. Schottky diodes with an equivalent

(or better) recommended maximum peak inverse voltage and current ratings were selected. Capacitors were selected according to the same principle taking into account recommended capacitance and ripple current ratings. A Coilcraft 3.3µH inductor with a saturation current of 3A was selected for both the +6V and +10V outputs.

4.3.3. Battery and Charger

An EEMB LP103450P-2S 7.4V 1850mAh Lithium Ion Polymer battery pack is used for backup during power outages or dips. LiPolymer batteries have a fairly flat discharge curve (until shutoff) which allows longer operation of the Power Profiler when battery powered.

The chosen battery pack consists of two LiPolymer 3.7V cells connected in series to produce a nominal voltage of 7.4V and a fully discharged end voltage of 5.5V. The battery pack also includes an internal temperature sensor which is used by the charger IC to determine charge termination and to comply with the necessary safety protection recommended for Li-Poly batteries [EEMB Battery, n.d].

The charger circuit is based on a Microchip MCP73864 dual-cell Li-Polymer charge management controller with integrated pass transistor, current sensing and safety timers. The battery charger circuit diagram is shown in Figure 4.18.

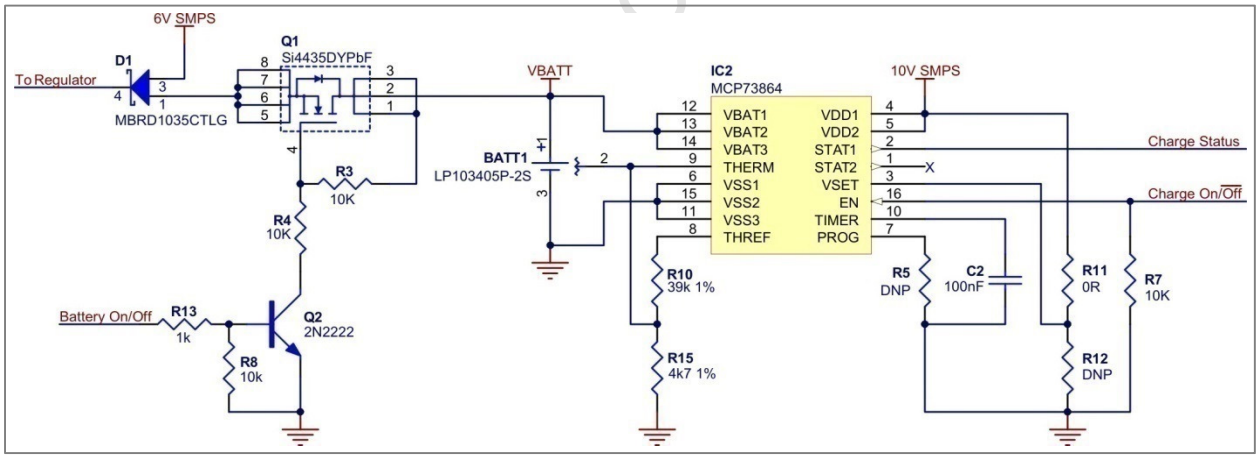


Figure 4.18 – Lithium Polymer Battery Charger Circuit

The MCP73864 is set to charge the battery with 8.4V (as recommended by the manufacturer) by V_{SET} being pulled up to VDD via R_{11}. R5 sets the charge current according to the formula shown below:

$$R_{PROG} = \frac{13.2 - 11 * I_{REG}}{12 * I_{REG} - 1.2}$$

For the Power Profiler implementation R_5 is not populated in order to default the charger to the maximum charge current of 100mA. It was decided to keep this charge current low to minimize the overall current draw of the Power Profiler as it will typically be connected to the utility side of the household being profiled.

Capacitor C_2 sets the safety timer for pre-conditioning, fast charge and termination. A value of 100nF was selected to set charge termination at 3 hours.

The output of the battery is connected to Q_1, a Si4435DY P-Channel 30V 8A MOSFET. The low gate threshold voltage of -1.0V allows it to be turned on with a simple NPN transistor (Q_2) pulling it to half its source voltage via divider R_3 and R_4. Schottky diode D_1 prevents current flowing through Q_1's blocking diode to the battery when the MOSFET is off. Schottky diodes are used instead of general purpose rectifier diodes throughout the power circuits because their lower forward-drop voltage results in less loss and therefore longer operation when battery powered. For example, a common 1N4007 rectifier diode has a forward voltage of 1.1V at 1A (at 25°C) [Fairchild Semiconductor, 2009] versus the MBRD1035 used here which has a forward voltage of 0.47V at 5A (at 25°C) [ON Semiconductor, 2006].

4.3.4. Final Regulation

A Micrel MIC29152 low-dropout linear voltage regulator is used to regulate the battery and SMPS main power to a regulated +5.6V at up to 1.5A. This Micrel device has a dropout voltage of 350mV at 1.5A, allowing it to (theoretically) operate from battery input voltage down to 5.95V, which would be approximately 90% capacity of a typical Li-Poly battery. As a comparison, a commonly-used linear regulator such as an LM317 can have a dropout voltage as high as 2.25V at 1.5A (at ambient temperature) [National Semiconductor, 2011], and therefore could not be used to regulate the battery power in our application. A 470µF 16V low-ESR capacitor is used on the output of the regulator to provide a good power source for the Control Board. The power is supplied to the control board through a 10-way dual-in-line IDC PCB header which also provides input voltage measurement and power control signals.

4.3.5. Power Management

The Control Board can enable the battery backup by switching MOSFET Q_1 on/off and also monitors and controls the battery charger. The battery voltage is monitored so that when operating on backup supply the battery can be disconnected and the Power Profiler turned off if the battery voltage drops below the manufacturer-specified end-discharge voltage. The primary power sense signal line (VPRIOn) indicates power on by voltage-dividing the +6V output of the SMPS down to a TTL-input compatible level of 0 to +5V DC.

4.3.6. PCB Design and Assembly

The Power Board was a challenge to integrate into the low-profile base of the Power Profiler enclosure primarily because of the large DC bulk-storage capacitors and the SMPS transformer required by the design. A low-profile Ferroxcube horizontal-mount transformer was selected for the SMPS design and axial PCB-mount electrolytic capacitors were placed horizontally in PCB cutouts to reduce their height as shown in Figure 4.19 (the height-constrained parts are marked in yellow). This solution of placing the capacitors and

also the battery in cut-outs proved to be very effective in reducing the overall height of the solution to less than 28mm.

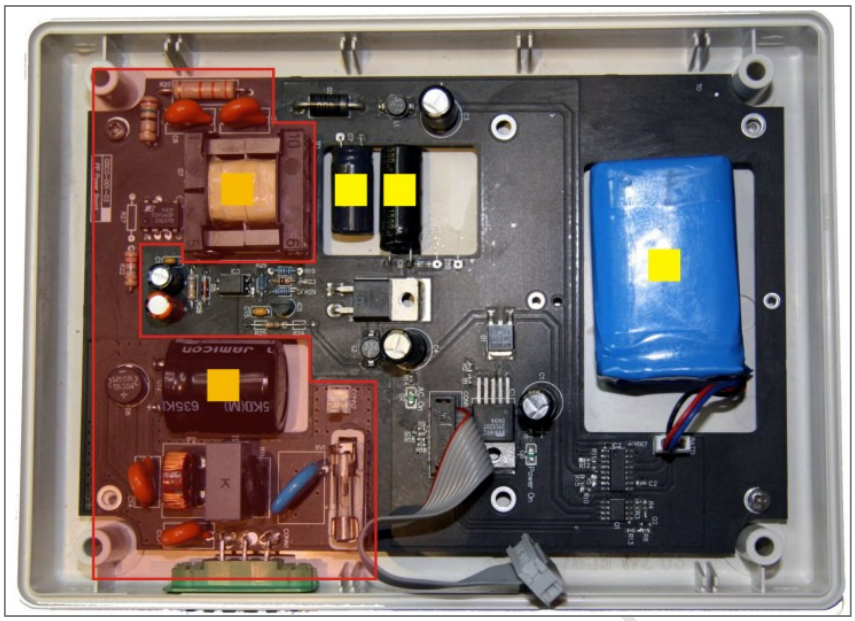


Figure 4.19 – Power Supply mounted in Enclosure Base

The area shaded in red is the high-voltage side of the switch-mode power supply and caution should be exercised when working in this area when the Power Profiler is powered and open. The battery charger and control circuitry is on the right side of the PCB with the localized power regulation in the middle near the Control Board ribbon connector. The Power Board was assembled by hand from the generated BOM and the completed PCB is screwed into the machined plastic base using four 2.9mm x 6.5mm self-tapping screws.

4.3.7. Power Supply Testing

To safely test the switch-mode-power-supply the input voltage was ramped up from 80V to 230V AC using a variable-output transformer which also verified that the SMPS would operate from 110V AC. The primary output voltage was measured as +6V DC and the secondary charger supply as +10.6V DC which was above the design value of +10V DC but still within an acceptable input level for the MCP73864 charger.

During normal mains-powered operation MOSFET Q_1 is off so that the battery is isolated from powering the linear regulator during charging. The problem this arrangement creates is that when power is lost, Q1 needs to be very quickly turned on to supply power from the battery to the final linear regulator. A more reliable means of switching to backup battery may be required and could possibly be implemented using a voltage comparator to directly activate Q_1.

The voltage divider used for measuring the battery voltage is also non-ideal in that it presents a constant discharge rate of approximately 2mA which over time will fully discharge the backup battery. A more appropriate solution to this would be to use either a voltage comparator with appropriately set trip voltage, or a high-impedance voltage-divider and op-amp buffer circuit to still provide a low-impedance measurement source as required for the PIC18F ADC.

4.4. Signal Interface Boards

The signal interface boards are used to connect the external signal connectors mounted on the sides of the enclosure lid to the Control Board. The surge protection components described in Section 4.2.1 are mounted on these boards and connected to the Earth connection on the Power Board. Any surge voltages will be routed away from the Control Board to external Earth to provide an additional means of protection to the Power Profiler input channel circuitry. The detailed designs of the interface boards as well as the mechanical integration details are given in Appendix B.

4.4.1. Voltage and Currents 1 to 6 Interface Board

The voltage interfaces are protected using surface mounted MOVs ($R_{1..4}$) and the current channels are protected using 5V Tranzorbs ($D_{1..12}$). These components are placed on the same side of the interface PCB as the signal connectors to make effective use of the 'dead space' between the enclosure and PCB, as well as to provide clearance for the inter-connection to the Control Board as shown in Figure 4.20.

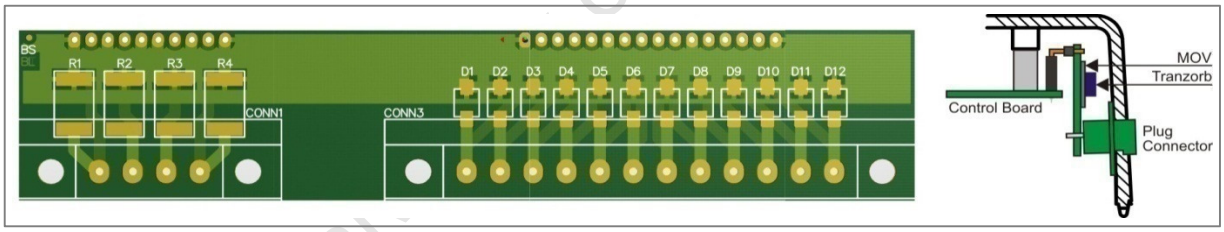


Figure 4.20 – Voltage and Currents 1 to 6 Interface Board

4.4.2. Currents 7 to 12 Interface Board

The interface board for current channels 7 to 12 is essentially the current portion of the Voltage/Current Interface board described in Section 4.4.1 as shown in Figure 4.21.

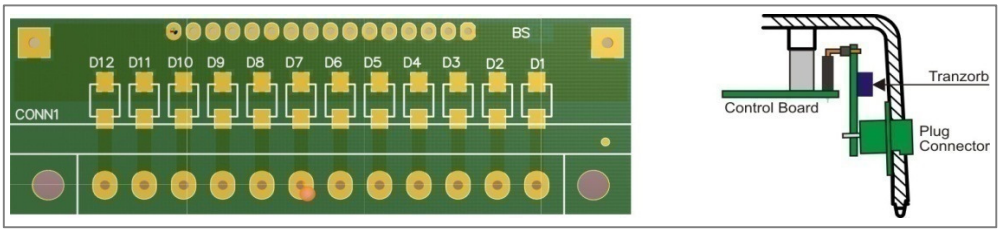


Figure 4.21 – Voltage and Currents 7 to 12 Interface Board

4.4.3. Industrial Inputs Interface Board

Circuit protection for the industrial inputs consists of gas arrestors and Tranzorbs which results in a higher PCB layout than the current and voltage Interface boards as shown in Figure 4.22. The gas arrestors are placed low down on the PCB to allow sufficient mechanical clearance to the Control Board.

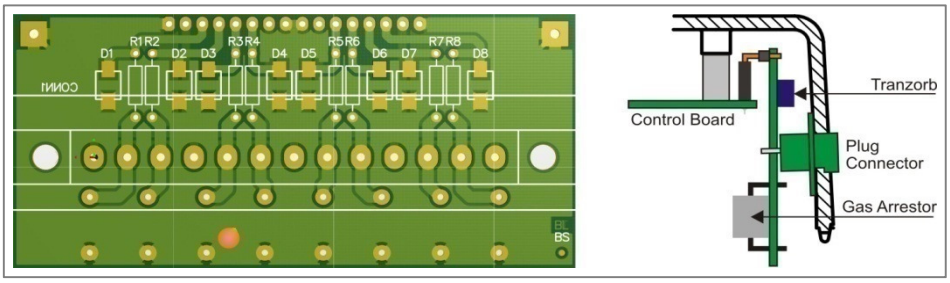


Figure 4.22 – Industrial Inputs Interface Board

4.5. GSM Remote Communications Interface Board

The GSM interface board provides the necessary interface between the dsPIC UART and the Telit GC864 GSM module. The GC864 is powered from a nominal +3.8V DC supply with I/O interfaces that are limited to +1.8V DC. These characteristics are typical of GSM modules from various manufacturers. The interface board also provides the SIM card holder ($CONN_2$) and a visual indicator (D_1) to show modem operational state.

4.5.1. Circuit Design

Through the interface connector $CONN_1$ the Control board provides regulated power, sets the I/O interface voltage, and provides the UART communication and device control signals (power and reset). Refer to Appendix B for detailed circuit diagrams.

Resistor R_2 sets the output voltage of the MCP1726 adjustable regulator when the interface board is plugged into the Control Board. The 10k 1% resistor (R_{78}) on the Control Board in conjunction with the R_2 value of 82k 1% sets the output voltage to approximately +3.8V according to the equation [Microchip Technology Inc., 2007]:

$$V_{RC} = 0.41 * \frac{R_2 + R_{78}}{R_2} = 0.41 * \frac{82k + 10k}{10k} = 3.78V$$

1% tolerance resistors are used to ensure tighter output voltage regulation. GSM modules produce high-current peak burst at up to 2A for short durations which are supplied by low-ESR bulk-storage capacitors mounted close to the modem power pins through wide power tracks to reduce series resistance.

The UART and I/O interface for the GC864 operate at a lower voltage than the dsPIC and require level shifting which is achieved using BCR22PN NPN/PNP transistor pairs. The BCR22PN include the necessary

bias resistors and is offered in a small SOT363 surface mount package [Siemens, 1996]. These transistor pairs are used as shown in Figure 4.23 to translate from VIO (+5.0V) to GSMIO (+1.8V) and vice versa.

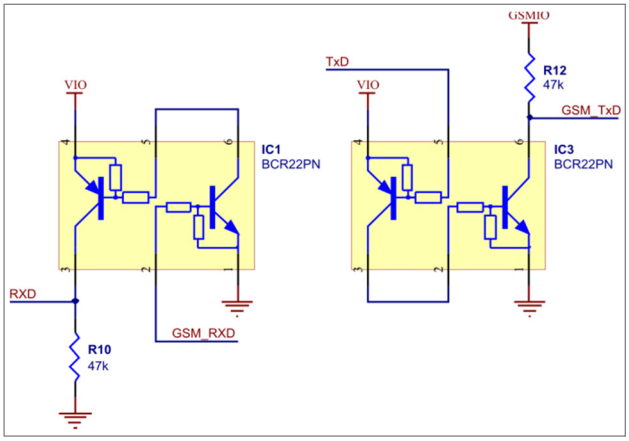


Figure 4.23 – I/O Interface Translation

The GC864 requires the power on/off signal to be pulled low for approximately one second as specified in the GC864 Hardware User Guide [Telit, 2009]. This is achieved using a 2N2222 NPN transistor (Q_2) to pull down the open collector input of the GC864. The module is reset by pulling low the reset pin in the same manner as the power pin and is provided for by Q_3.

4.5.2. PCB Design and Testing

The GSM interface board is designed as a double-sided PCB with maximized ground planes and thick tracks for the +3.8V power connection. Bulk storage capacitor C_2 is placed as close to the power pins of the 80-way GC864 module connector and multiple vias are used to connect the top and bottom traces together to minimize track resistance and inductance. The SIM card signal connectors are sensitive to noise from other traces and are routed to be as short as possible. The GC864 interface board layout is shown in Figure 4.24. The multiple-via thick track connection for the +3.8V power can be seen on the positive pin of C_2 on the top layer, and again on the bottom layer where it connects to the module itself.

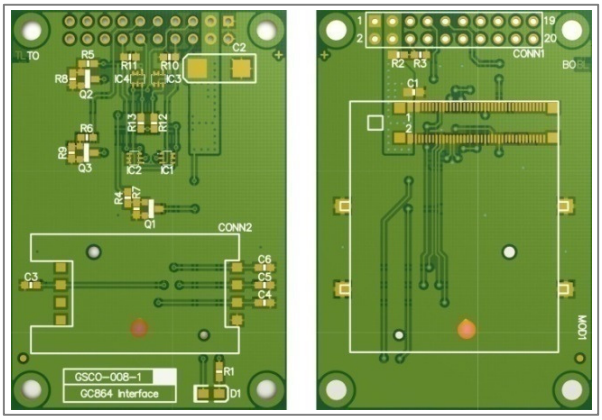


Figure 4.24 – GC864 Interface Board Top and Bottom

Board assembly was made more difficult by the high pin density of the 80-way Molex connector used for the module interface. Although not an easy process it was possible to hand-solder this connector effectively using good quality solder and flux. The external antenna directly connects to the module through an on-board U.FL RF connector. A Taoglas FXP-07 flexible PCB antenna was selected [Taoglas, 2012] as it can be mounted inside the plastic enclosure and therefore may offer some protection against vandalism. In cases where an external antenna is required a SMA bulkhead to U.FL connector could be used and mounted through the Power Profiler enclosure. Before the GC864 module was installed the correct supply voltage of +3.8V was checked. Further testing of the actual module required the embedded software interface to be implemented as discussed in Chapter 5.

4.6. Bluetooth Interface Board

The Bluetooth Interface board provides the necessary interface between the dsPIC and the KC Wirefree KC-11 Bluetooth module. As the KC-11 operates at +3.3V and the dsPIC at +5V, the same method of I/O level shifting and reset control is used as discussed in Section 4.5.1 for the GSM interface board design.

4.6.1. Circuit Design

Power for the RF interface board is supplied by a MCP1726 regulator on the Control Board. The Bluetooth interface board uses a 68k 1% and 2.4k 1% resistor in series to set the MCP1726 adjustable output at +3.3V. The KC-11 has relatively low burst currents of 210mA and bulk-storage is supplied from the output capacitors of the MCP1726 regulator.

4.6.2. PCB Design and Testing

The layout of the KC-11 Interface PCB is straight-forward and the module is supplied as a surface-mount assembly unit with integrated PCB chip antenna. The board layout is shown in Figure 4.25.

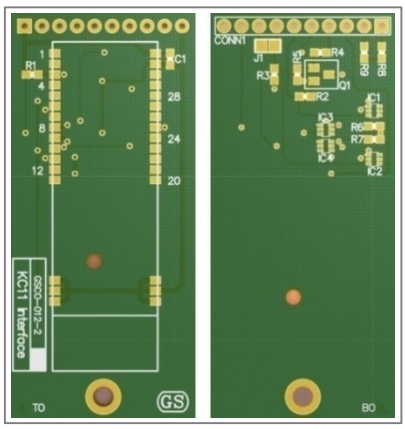


Figure 4.25 – KC-11 Bluetooth Interface Board Layout

The board is simple to assemble and before the KC-11 was populated the output voltage of the regulator was verified to be +3.3V. Further testing required the embedded software to be implemented.

4.7. Power Profiler Assembly

An assembled Power Profiler consists of the enclosure base with the Power Board mounted and the enclosure lid with the signal interface boards and the Control Board mounted. The Power and Control Boards are connected through a 10-way IDC ribbon cable that supplies regulated power and control signals for power supply management.

4.7.1. Base Assembly

The enclosure base needs to be machined as shown in Appendix B to allow for the three-way power connector to be fitted. The assembled Power Board is mounted into the base using four M2.5X5mm self-tapping screws. Once the PCB is fitted the power connector is soldered and the Lithium Polymer battery is mounted to the base with double-sided tape.

4.7.2. Lid Assembly

The enclosure lid needs to be machined as shown in Appendix B and its printed vinyl decal attached. The following process is followed to assemble the Control and interface boards into the lid:

- Step 1: Four threaded M3x10mm PCB spacers are inserted into the mounting holes of the enclosure lid. The mounting holes can be thread tapped or the spacers can be heated using a soldering iron and when hot enough, pushed in.
- Step 2: The plug connectors are screwed into the machined enclosure using stainless M3X8mm flat-head machine screws.
- Step 3: The LED/LCD interface board is plugged into the Control Board and secured using M3 nylon bolts, nuts and washers. Nylon bolts can be easily cut to length once secured.
- Step 4: The interface boards are plugged into the Control Board via the right-angle connectors and the entire assembly is placed in the enclosure. The interface boards can be pulled towards each other to get them over the plug connector pins.
- Step 5: Secure the Control board into the enclosure with four M3X8mm machine screws.
- Step 6: Once everything is suitably aligned the plug connectors are soldered to the interface boards. Locating the interface boards and connectors before soldering significantly improved the ease with which the Control Board could be removed and re-inserted into the enclosure. This more ‘natural’ alignment makes the interface connector pins align far better because the plug connectors are not actually perpendicular to the interface boards as the enclosure tapers.
- Step 7: The remote and RF communication boards can now be fitted to the Control Board and screwed in place using M3x8mm machine screws.

Figure 4.26 shows a Control Board ready to be installed and the completed Power Profiler lid assembly.

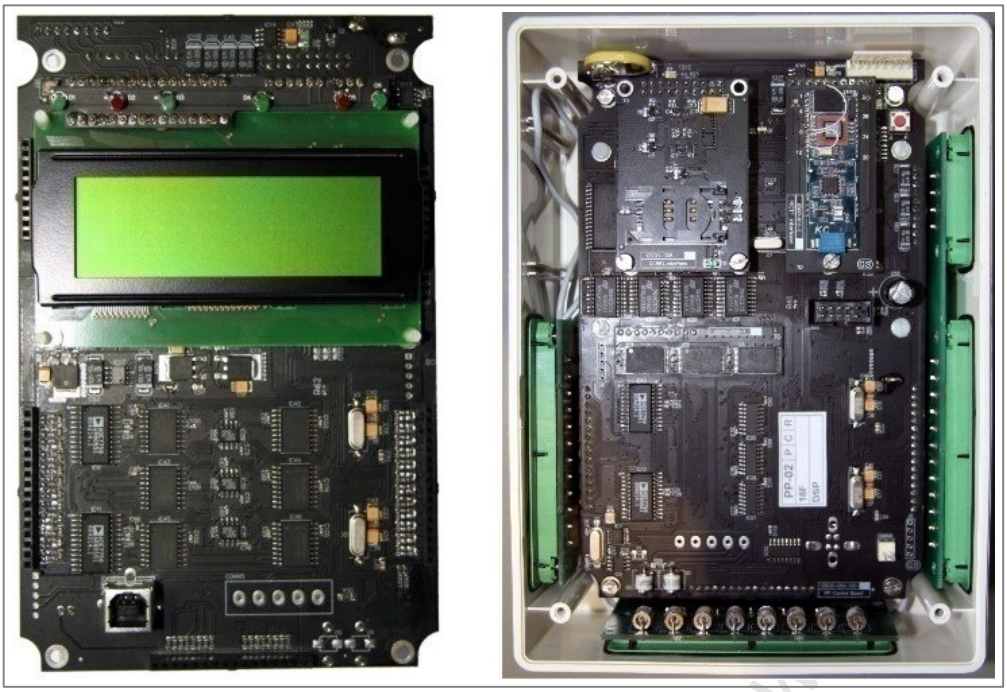


Figure 4.26 – Control Board and Assembled Power Profiler Lid

4.7.3. Unit Assembly

The base and lid are connected using the 10-way IDC cable and the Power Profiler is screwed closed using the four screws supplied with the enclosure. The completely assembled unit is shown in Figure 4.27. The front decal is printed using a digital production colour printer onto clear vinyl with an adhesive backing. The necessary cut-outs around the LCD and front-panel connectors are done using a digital cutter.

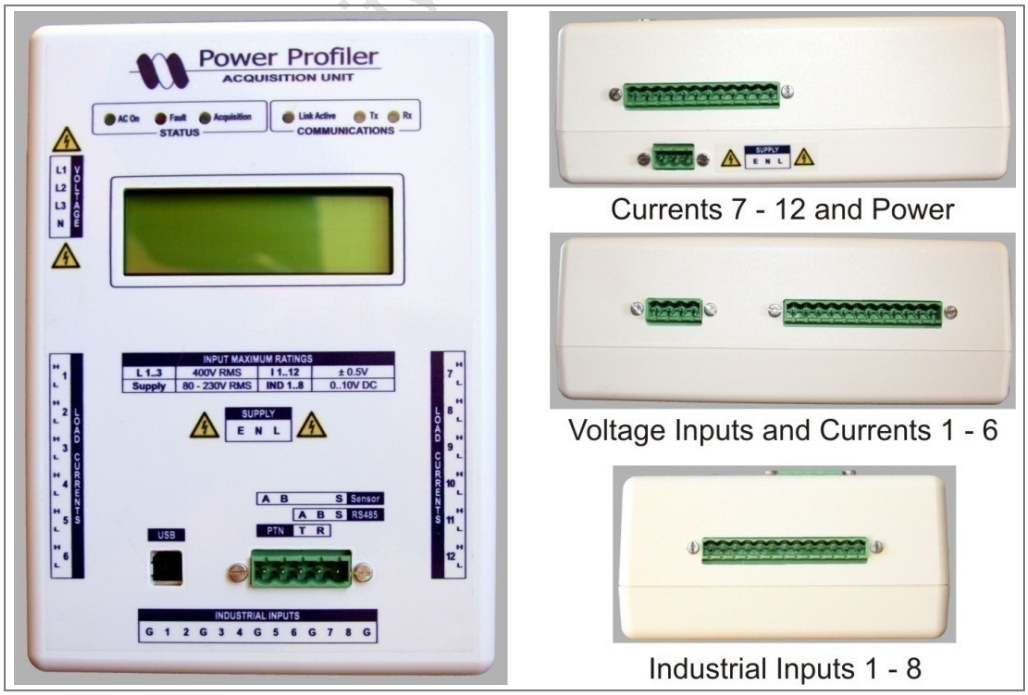


Figure 4.27 – Assembled Power Profiler

4.8. Profiler Costing

A requirement of the Power Profiler product is that it needs to provide a cost-effective solution for research-orientated domestic load measurement applications. To determine the cost-effectiveness the overall manufactured cost of the Power Profiler is calculated and compared against other currently available solutions.

The manufacturing cost of a Power Profiler is listed in Table 4.1. This cost is based on a small order quantity that is hand-assembled and is expected to decrease by a minimum of 25% with automated assembly and production volumes over 100 pieces.

Table 4.1 – Profiler Costing Summary		
Component Group	**Line Cost**	
Control board	R	1 587
LCD/LED display board	R	99
Currents 7 to 12 interface board	R	114
Industrial Inputs interface board	R	145
Voltage inputs and currents 1 to 6 interface board	R	158
Power Supply Board	R	594
Enclosure, machining and assembly parts	R	223
Total cost for Profiler (ex. VAT)	**R**	**2 920**
KC-11 Bluetooth interface board	R	286
GC864 GSM interface board	R	347
Total cost including GSM and Bluetooth (ex. VAT)	**R**	**3 553**

The first cost of R2 920 is for a Power Profiler without communications capabilities and can be compared with using discrete panel meters for measurement. The final cost of R3 553 includes the Bluetooth and GSM GPRS communications interfaces.

4.8.1. Comparison to Commercially Available Solutions

Entry-level commercially available single- or three-phase power meters (distribution board type) could be used for limited measurement gathering via their infra-red interface or by connecting them to an intelligent controller using a bus communications interface such as MODBUS. Online distributor RS Components offers an entry-level single-phase energy meter with infra-red interface such as the Socomec Countis E10 for R1 106 or a three-phase Hobut M850-MP1 power meter with RS485 interface at R1 544 [RS Components, 2012].

The most cost-effective equivalent to the Power Profiler would be to use four Hobut M850's connected via MODBUS to a controller for data collection and remote communications. Such a solution would cost over R6 000 just for the measurement units, and cost would increase considerably with the addition of a controller and communications solution. Based on this costing an integrated solution such as the Power Profiler remains a more cost-effective solution than using individual panel meters and a controller.

Power Quality Analysis Comparison

Other than the requirement of measuring (and profiling) energy usage, the Power Profiler offers direct access to measurement data which can be used for power quality analysis applications. A low-cost power quality analyzer such as the ISO-TECH IPM3005 is available from RS Components for R2 644, and provides three-phase voltage, current, active, reactive and apparent power measurements. More advanced instruments such as the Chauvin Arnoux CA8334 power quality analyzer retail at RS Components for R29 263 [RS Components, 2012].

Remote Communications Ability

Panel meter and power quality analyzers do not typically contain GSM modems to access measurement data. Since most meters have digital interfaces it would be possible to use a GSM modem to connect to them for data access, but typically these meters would not have the profiling capability to store measurement data.

4.8.2. Cost-Effectiveness Comparison

The Power Profiler offers a cost-effective solution that integrates remote and local access communications abilities with measurement storage and power quality analysis functions. The integrated nature of the product suits it well to research-orientated applications in that the software is accessible to be changed to support future applications. If the Power Profiler could provide even basic power quality analysis functionality, it would make it an even more cost-effective solution.

4.9. Conclusions

This chapter presented the design of the final Power Profiler prototype which is to be used for field testing. Circuit corrections and improvements identified from prototype two are included and a new low-profile AC/DC power supply and battery backup design is presented. The design of the GSM and Bluetooth RF interface modules is also given and basic testing performed to ensure correct circuit operation.

The production cost of the Power Profiler product was established at R3 553 including GSM and Bluetooth communications. To establish its cost-effectiveness it was compared to using off-the-shelf measurement instruments which would require four power meters, a controller and GSM module. Just the four power meters in such a configuration would cost in excess of R6 000 so the Power Profiler is confirmed as being a suitable cost-effective instrument for load-profiling applications.

Chapter 5

Software Development

The Power Profiler system consists of three software components viz. the dsPIC embedded software, PIC18F embedded software and the PC-based Configuration and Control application. This chapter describes the implementation of these software components and how they inter relate by discussing them as four functional areas:

- Hardware initialization and control,
- Measurement acquisition and profiling,
- Communications implementation between the PC and Power Profiler, and
- Configuration, control and retrieved data processing.

The embedded software was implemented in Microchip C using their free MPLAB integrated development environment and REAL-ICE and ICD2 in-circuit debugging tools. The PC-based software was written in Microsoft C# using .NET Framework 4 in Visual Studio 2010 Professional. The C# communications and device control software was implemented as re-usable classes to support the development of multiple applications that can interface with and control the Power Profiler. The source code listings are given in Appendix C.

5.1. Hardware Initialization and Control

The embedded software functionality in the Power Profiler is split into several modular components termed Tasks. Each Task consists of an initializer method which is called at processor start-up, and a processor method that is repetitively executed to perform the Task's functions, typically implemented as a state-machine. Certain high-level Tasks (such as communications) will call several sub-Tasks each with their own initializer and processor methods. The main tasks of the Power Profiler as shown in Figure 5.1.

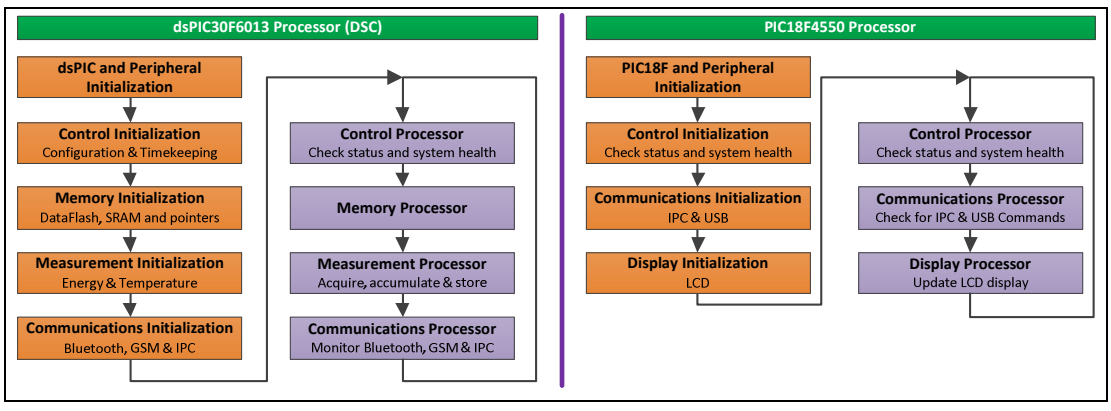


Figure 5.1 – Software Tasks per Processor

The dsPIC processor is the heart of the Power Profiler performing the data acquisition, measurement processing and storage and controlling the communications. The PIC18F processor is used as an I/O expander to control several peripherals and the LCD interface, and to implement the USB communications interface. The two processors communicate with each other via the SPI interface with the dsPIC processor acting as the Master. As processors operate independently and at different speeds a hand-shaking signal is used to synchronise data transfers.

5.1.1. Processor and Peripheral Initialization

Basic processor operation such as crystal selection and speed are defined as pre-processor definitions in the *Main.c* files for the respective processors. The processor initialization routines *dsPIC_Initialize* and *PIC18F_Initialize* configure the processor, I/O ports, and internal peripherals such as the ADC, communication interfaces and timers. The dsPIC processor communicates with several external peripherals through the I^2C, SPI and UART ports as shown in Figure 5.2.

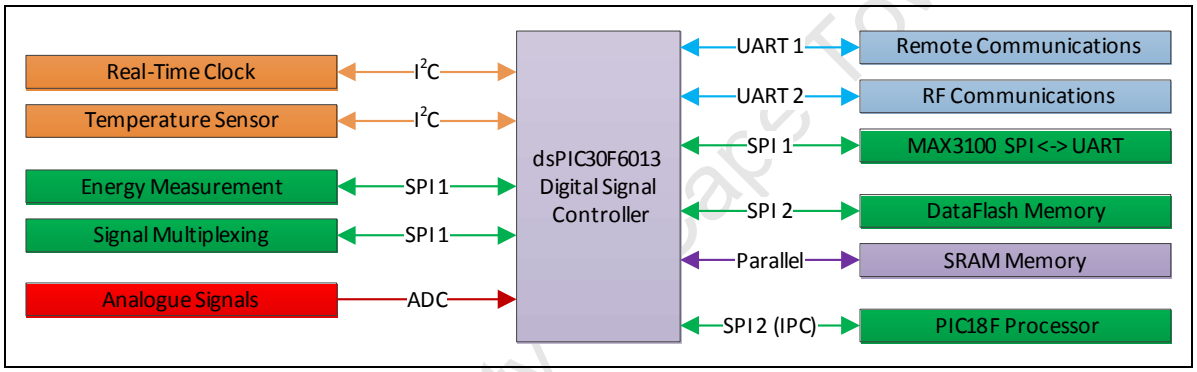


Figure 5.2 – dsPIC External Communication Interfaces

The parallel interface used to access the SRAM memory is implemented as discrete port operations using an 8-bit port and discrete I/O signals. The functionality of the I^2C, SPI and UART interfaces is described in the Microchip dsPIC30F Family Reference Manual [Microchip Technology Inc., 2005].

The I^2C interface is a serial 2-wire interface with data and clock signals. Each device is uniquely addressed on the bus and baud rate is limited to 400 kHz depending on the peripheral device capabilities. The SPI interface is a high-speed bi-directional serial interface that uses separate data in and out signals and individual chip-select lines for each device on the bus. An I^2C and SPI interface library is implemented in *I2CSPIComms.c* to configure the ports and communicate with the external devices.

The UART ports implement a hand-shake serial communications interface that typically operates at speeds up to 115 Kbps. The Power Profiler uses the UART ports to interface with the remote and RF communications modules using the Microchip UART library.

The Power Profiler uses several I/O lines to control peripheral devices. Devices that require fast access and control are driven directly from the I/O ports of the dsPIC and PIC18F processors. To increase the available I/O signals an external octal latch (IC_2) is used and is controlled using the *SelReg_Initialize*, *SelReg_Clear* and *SelReg_Set* methods described in *Main.c*. The PIC18F processor uses a serial-to-parallel 8-bit shift register (IC_1) to create additional I/O and is controlled using the *SelReg_SetValue* method to serially clock and latch the specified register value. *SelReg_Clear* and *SelReg_Set* methods allow individual pin control of the shift register.

5.1.2. Power Profiler Control

The Control Task is responsible for maintaining the real-time-clock, synchronizing time-based operations between the software Tasks and to monitor the hardware state to ensure correct operation of the Power Profiler.

Time-Keeping

The Profiler uses internal timers and the DS32331S high-accuracy RTC to maintain current time-of-day information for synchronizing the software tasks and for time-stamping measurements. There are several base time triggers used in the Power Profiler viz. 10msec generated by Timer 5, 100msec generated by Timer 4 and 1sec generated by an interrupt from the external RTC. The 100msec trigger is used by most of the software Tasks' state machines to trigger functions such as measurement acquisition and status reporting.

The DS3231S RTC is initialized using the *DS3231_Initialize* method to ensure its oscillator is started and to configure its square-wave output signal for a 1Hz duty cycle. This signal is connected to pin RA7/CN23 of the dsPIC and increments a software-implemented time-of-day clock *(currentTOD).* The software clock is synchronized with the external RTC at startup and on minute rollovers to ensure the date is correctly represented and that the two clocks are synchronized. To read or set the RTC's current time-of-day value the *DS3231_GetClock* and *DS3231_SetClock* methods are used with the value represented as a *tDateTime* type. This method of 'caching' the time-of-day value allows events to be time-stamped without requiring I2C bus access to the external RTC which speeds up and simplifies the process.

Configuration Settings

Power Profiler configuration parameters are stored in the dsPIC's internal EEPROM memory and are accessed during Task initialization using the *intEE_ReadRow* method. The configuration values are accessed in the embedded code as a data structure defined in *ConfigBlockDefns.h* and are written to EEPROM using the *intEE_WriteRow* method under the control of the Configuration and Control PC application. The dsPIC processor implements EEPROM memory access in rows of 32-bytes and so the configuration memory is allocated in blocks of rows as given in Table 5.1.

Table 5.1 - EEPROM Memory Utilization		
Position	**Blocks**	**Description**
0	1	Profiler Identity
32	2	Measurement configuration
96	1	Memory configuration
128	5	Channel calibration data (general)
288	2	Unallocated
352	2	Energy IC 1 calibration data
416	2	Energy IC 2 calibration data
480	2	Energy IC 3 calibration data
544	2	Energy IC 4 calibration data
608	7	1-Phase 2-Wire conversion calibration data
832	7	3-Phase 3-Wire conversion calibration data
1056	7	3-Phase 4-Wire conversion calibration data
1280	7	Custom setup conversion calibration data
1504	1	Communications configuration
1536	16	Unallocated

Erased processor memory is defaulted to 0xFF so after a successful configuration write the first two bytes are set to 0xAABB to indicate the configuration is now valid. During Control Task initialization the Profiler Identity block is retrieved to access the unique 32-bit serial number and 16-character password used for access authentication during remote communications. For a non-configured Power Profiler the default Device ID is 0 and the password is "" (blank). The password length is set as a maximum of 16 bytes to support the future use of an MD5 Hash sum representation [Rivest, 1992] if desired. The other configuration blocks are accessed by their respective Tasks during initialization.

System Status Monitoring

The Control Task regularly verifies the correct operation of the hardware and software Tasks and will report errors in the *procErrors* variable *(procControl.h)* which can be read by the Control and Configuration software. The front panel Error LED is also switched on to indicate to the user that a fault has occurred and that the Power Profiler may not be capable of measurement processing. When mains failure is detected the Control Task will disable the battery charger and enable the battery MOSFET to power the Profiler from the internal battery. If the battery voltage falls below the minimum recommended threshold the MOSFET will be opened to disconnect the battery and prevent excessive discharge and battery damage.

LCD Module

The 4 line by 20 character LCD module is controlled via the parallel interface of the PIC18F processor and is used to display the current measurement parameters. The measurement values are sent by the dsPIC processor every 800msec and displayed as a cycling screen of the voltage and current values. The LCD module uses commands to initialize the display and to control character placement using a cursor. The module's register select signal line is set low for command access and high for data buffer access for the display of characters. Initialization is performed by the *LCD_Initialize* method which issues commands to

configure the interface, clear the display and reset the cursor position to zero. Before characters are written to the display buffer the cursor needs to be moved to the correct column and row position using the *LCD_Move* method. Several methods have been written to write characters, strings and numbers to the display and can be found in *LCD.c*.

5.1.3. Memory Management

The procMemory Task initializer verifies the correct operation of the external DataFlash and SRAM memories using the *DataFLASH_DeviceOK* and *SRAM_CheckDevice* methods and any errors will be reported by the Control Task's *procErrors* structure. The *procMemory_Processor* method does not perform any functions at this time, but is intended for future use if managed memories such as an SD card were to be used.

The *procMemory.c* file includes the memory access routines used by the Measurement and Communications Tasks to store and retrieve measurements. To simplify the addressing of the DataFlash memory it is addressed as a contiguous 24MB block and is converted using the *StorMem_DecodeAddr* method from a 32-bit address to a device, page and page index representation. The storage memory map for the Power Profiler is shown in Table 5.3.

Table 5.3 – Storage Memory Map
Measurement pointers (1056 bytes)
Seconds profile measured data (user-defined size, can be wrapped)
Minutes profile measured data (user-defined size, can be wrapped)
Hours profile measured data (user-defined size, can be wrapped)
Event profile measured data (user-defined size, can be wrapped)
Event messages (0 bytes – unused at present)

The first page (1056 bytes) of DataFlash 0 is used to store the measurement read and write pointers so that if power is removed, the Power Profiler can resume after startup without overwriting previous measurements. These pointers are accessed as a *tDataFLASHMemoryPtr* structure using the *StorMem_GetMemPointers* and *StorMem_SetMemPointers* methods. Flash memory can perform a limited number of write operations to a cell before cell failure can occur. To increase the endurance of the flash page the measurement pointers are not updated with every measurement stored, but once per hour and immediately if the power is removed.

The memory used by the measurement and event profiling is determined during configuration of the Power Profiler. The user can enable or disable specific measurement channels (affecting the measurement size) and the ratio of memory allocated between the different profiles. The configuration software dynamically allocates memory based on the user's preferences and writes the settings to the Power Profiler's configuration memory at address 96.

The event messages memory space is currently not used but is intended to support future power-quality analysis flagging according to the IEC 61000-4-30 specification [IEC, 2003].

The SRAM memory uses *SRAM_SelectAddress* to latch the correct 32-bit address value onto the memory address selection latches. Read and write data access is provided by several *SRAM_Read* and *SRAM_Write* routines for different data sizes and for multi-byte access. The SRAM memory is provided for future applications where processing of raw signal inputs may be required.

5.1.4. Measurement Initialization

The measurement hardware such as the dsPIC ADC, external temperature sensor and energy measurement ICs are initialized by the *procMeasurement* Task. During initialization the communications with the external devices is verified and any errors are reported in the *procErrors* structure.

Analogue Measurement Signals

The ADC is configured (in the *ADC_Initialize* method) to scan and auto-convert the AN9 to AN15 inputs and generate an interrupt when the conversion of all 7 channels is completed. The results are available in the ADC's ADCBUF0..6 registers which are read during the interrupt routine and the process automatically restarts. The register values and their associated analogue signals are shown in Table 5.4.

Table 5.4 – ADC Register and Signal Relationship

Register	Description
ADCBUF0	Voltage input signal 1
ADCBUF1	Voltage input signal 2
ADCBUF2	Voltage input signal 3
ADCBUF3	Current signal inputs 1,4,6,10 multiplexed
ADCBUF4	Current signal inputs 2,5,8,11 multiplexed
ADCBUF5	Current signal inputs 3,6,9,12 multiplexed
ADCBUF6	Industrial inputs 1..8 multiplexed value

The voltage and current channel sampling is provided for future power quality analysis applications. When the interrupt routine is triggered the industrial input register (ADCBUF6) value is stored in the *currentMeasurements.IndustrialInputs* array and the next input is selected using *IndIP_SelectNextChannel*. When all 8 inputs have been sampled, the process re-starts with input 1.

TCN75 Temperature Sensor

The Microchip TCN75 digital temperature sensor is initialized using *I2CTemp_Initialize* and configured to auto-convert temperature. The measured temperature value is retrieved every second using the *I2CTemp_Temperature* method.

Energy Measurement

The ADE7758 energy measurement ICs are accessed on SPI port 1. Read and write operations to the ADE7758 control and measurement registers are performed by the *ADE7758_Read* and *ADE7758_Write* methods. Initializing the ADE7758 for operation requires the calibration and measurement configuration data to be retrieved from configuration memory and written to each energy IC.

5.2. Measurement Acquisition and Profiling

Measurement and profiling is implemented by the *procMeasurement* Task and is one of the main software components of the Power Profiler responsible for the acquisition, processing and storage of measured values. Task initialization starts by verifying the correct operation of the hardware and configuration parameters set by the Configuration and Control application. Hardware faults are reported in the *procMeas.HWFault* flag and configuration errors are flagged during the *procMeas_InitializeCapture* method.

5.2.1. Measurement Configuration

The *procMeas_InitializeCapture* method retrieves the configuration set by the user in the Configuration and Control application and downloaded to the configuration memory. Configuration block structures are used to simplify accessing the parameters and can be found in *configBlockDefns.h*.

Measurement configuration parameters are retrieved first and used to configure the voltage input configuration (1-phase, 3-phase or custom) and to enable or disable the individual voltage, current and industrial channels. The custom voltage profile is reserved for future applications where input voltages may be mixed across several energy measurement ICs. The voltage channel input mode is set by the *VMux_SetChannels* method and based on the voltage input mode selected the relevant channel calibration and ADE7758 control register settings are retrieved from memory by the *procMeas_SetEnergyICMode* method. The energy measurement IC calibration parameters remain the same independent of voltage measurement mode and are retrieved from configuration memory and downloaded to the ADE7758s.

The settings for the three profiles (seconds, minutes and hours) are stored as a *tProfileSettings* structure in the *secsProfile*, *minsProfile* and *hrsProfile* variables. Storage memory locations for each profile are retrieved during the *procMemory* initialization and stored in the *dataFLASHPtr* variable which is a *tDataFLASHMemoryPtr* structure. If the retrieved values have not been configured or the measurement profiles are being reset, these values are set from the retrieved configuration memory parameters and updated in DataFlash memory. This process ensures that measurement acquisition can continue if all power is lost to the Power Profiler.

5.2.2. Measurement Acquisition

To simplify the addition of new software functionality the measurement acquisition and profiling functions are isolated in two separate processes. At full-scale inputs the energy measurement ICs WATTHR and VARHR accumulated energy registers would overflow in 0.52 seconds [Analog Devices, 2011]. To prevent this overflow the RMS and power measurement values are read every 200msec using the *procMeas_Update* method. A global variable *currentMeasurements* maintains the latest measurement value and is updated with the RMS and energy measurements every 200msec, the current temperature reading every second and the industrial input values on each ADC interrupt (see 5.1.4).

Updating the measurement values independently of any post-processing functionality (such as profiling) makes software maintenance and development simpler as the developer just needs to access one variable to use the latest measurement values. This is also the reason for using the energy measurement ICs for RMS and energy calculation as continuous sampling of the analogue signals would add substantial processor overhead and complicate the development of future software functionality.

5.2.3. Measurement Profiling

The Power Profiler has three independent measurement profiles: seconds, minutes and hours. By operating these three independently the amount of data regularly uploaded can be better managed. For example: to identify appliance usage characteristics high-resolution second-based measurements would be useful, whereas longer-term demand forecasting can use hour-based measurement data. Seconds-based profiling generates large volumes of data so regular daily uploading would increase the operating cost of the Power Profiler solution. By only uploading hourly-profiled data the operating cost is significantly reduced and if needed the higher resolution data can still be uploaded as required.

For each profile the user is able to set the measurement storage interval, the memory usage (which determines the maximum number of samples stored) and whether the memory should wrap to the beginning when full to overwrite old measurement data. The hours profile is currently set to synchronize measurement storage with a roll-over of the clock hour, allowing hour-profile data to be synchronized between different deployed loggers.

Measurement Accumulation

Measurement values for each of the profile types are stored in an array of *tMeasAccumulation* type called *measAccumulation*. When the *currentMeasurements* value has been updated with a new measurement result, each profile will call the *procMeas_Accumulate* method to accumulate the measured value in the appropriate *measAccumulation* variable and to update the maximum and minimum values. Only channels that are enabled are processed and if this is the first measured value in an accumulation cycle, the maximum, minimum and accumulation fields are cleared.

Measurement Averaging and Storage

For the seconds and minutes profiles when the accumulated measurement count is equal to the specified profile's measurement total, the accumulated values are averaged and stored in the DataFlash using *procMeas_AverageAndStore*. This method creates a packed buffer of the measurement data starting with a measurement flags byte, a timestamp from the internal RTC, and the measurement values (maximum, average and minimum) for each enabled channel. The packed buffer is then added to storage memory by passing it to the *StorMem_AddProfileValue* method which will write it into the appropriate memory location for that profile and wrap the memory if necessary. Only channels that are enabled are included in the packed buffer to reduce the overall measurement size and to maximize memory usage. The hours profile averages and stores the measurement value on the hour-rollover of the real-time-clock to allow data to be correlated across multiple Power Profilers deployed in the field.

Once a profile's measurement has been stored the accumulation counter is set to zero which will trigger a reset of the accumulation fields upon the next *procMeas_Update*. Once per hour the *StorMem_SetMemPointers* method is called to commit the current measurement pointers to DataFlash memory.

5.3. Communications

The Power Profiler uses four communications types: inter-processor communications via SPI, USB for bulk data transfers, Bluetooth for short-range wireless communications and GSM GPRS for long-range wireless data access. The *procComms* Task is responsible for managing these interfaces and responding to commands from the Host system when connected.

Communication configuration settings are retrieved from the EEPROM memory during the Task initialization and set for each communications interface. The *procComms* Task treats each interface as a sub-task and calls the initializer and processor methods for each interface.

C# interface routines for the PC applications are implemented for host-side communications control using three namespaces:

- *Profiler.Comms.Interfaces* which implements the Bluetooth, GSM and USB interface classes that are used by the PC application to communicate with the Power Profiler
- *Profiler.Comms.Mediums* which implements the serial and TCP/IP interfaces used by the interface classes
- *Profiler.Comms.Protocols* which implements the Single-Point Error-Checked (SPECC) protocol used for ensuring communication integrity

The Communications process implements the communications buffer *(commsBuffer)* which is used by the Bluetooth, GSM and USB interfaces to buffer received SPECC packets. Received data is passed to the *Comms_RxByte* method where it is buffered until a full SPECC packet has been received. The data source is specified by the interface in use so that the Profiler can respond to the received command using the correct interface.

The *procComms_Processor* method calls each interface's task processor and waits for a SPECC command to be received in *commsBuffer*. Received commands are verified using a 16-bit CRC, decoded into a *tCommsPacket* structure and passed to *Cmd_Process* (see dsPIC:Commands.c) method for processing. A processed command responds to the Host using the *Comms_Repond* method and uses the *PacketSource* identifier specified in the packet to determine which interface to route the response through. The response could consist of a simple acknowledge (command), or return data (request) and uses a 16-bit CRC value to ensure data integrity.

Before access to data or configuration is allowed, the Host must login to the Profiler with the correct 32-bit ID and 16-byte password. A 16-byte password length was chosen to support the future use of an MD5 Hash algorithm to increase security as the password is not sent as an unencrypted plain byte field. If an incorrect ID or password is sent, the Profiler will ignore the Host (to improve security) so a 'ping' command is provided to verify communications and to determine the interface response time. The only commands supported without a successful login are: Ping, Login and Logoff.

5.3.1. Host Interface Protocol

The GSM, Bluetooth and USB communications interfaces use the same high-level protocol to interface between the Host and Power Profiler. The proprietary protocol is termed SPECC: Single-Point Error-Checked Communications and implemented on the host in *Profiler.Comms.Protocols.cs* and in the dsPIC in *Comms_IPC*. Data integrity is ensured using a 16-bit Cyclic Redundancy Check (CRC) implemented for transmission and reception on both the Host and Power Profiler.

Packet Structure

The SPECC packet structure is shown in Table 5.5.

Table 5.5 – SPECC packet structure

~	Packet Length	Transaction ID	Command	Parameters	CRC
1 byte	2 bytes	1 byte	1 byte	x bytes	2 bytes

The '~' character marks the start of a packet and the receiving methods will discard any characters received before the '~', synchronizing the data reception. The packet length is used to determine when the full packet has been received and is limited by the communications interface used. The transaction ID is incremented with each message sent to allow for duplicate messages sent during a re-send to be discarded. The command

byte specifies the action to be performed using the parameters provided (some commands do not require any parameters to be sent). The CRC appended to each packet is used to verify message integrity and if calculated over the entire packet received using the same CRC polynomial the result should be zero.

SPECC Protocol Usage

The *SPECCProtocol* class is instantiated within each communications interface class used by the Host: Bluetooth, GSM and USB. Communications between the Host and Power Profiler is performed using two methods: *Command* and *Request* which are exposed to the interface classes via the *SPECCProtocol* class.

Data to be transmitted is formed into the packet to be sent using the *Encode* method which also adds the CRC value. The constructed packet is sent using a *ProtocolTxEvent* delegate called in the parent interface class to send the data. Data received on the communications interface is passed on a byte-by-byte basis to the *ReceiveByte* method of the *SPECCProtocol* class. This method will receive the packet, wait for the start-of-transmission byte ('~') and will buffer the received data until the complete packet is received. The received byte array is passed to the *Decode* method where data integrity is verified using the CRC and if correct the packet is passed to the *Command* or *Request* in progress for processing.

The implementation of *Commands* and *Requests* within the *SPECCProtocol* class simplifies the implementation of interface classes as they only need to execute these methods to communicate with the Power Profiler.

Commands and Requests

The *Command* method in *Profiler.Comms.Protocols.cs* is called by higher-level interface classes with a command byte and any necessary parameters to instruct the Power Profiler to execute a task and acknowledge it if required. The method will wait up to *minResponseTime* milliseconds for the reply and if not received will report a communications error. The *Request* method issues the instruction to the Power Profiler in the same manner as a Command, but expects the Power Profiler to return data which is passed back to the calling interface for further processing.

5.3.2. Inter-Processor Communications

The dsPIC and PIC18F processors communicate with each other using SPI with the dsPIC acting as a master and the PIC18F as a slave. The physical interface consists of the standard SPI data, clock and chip select signals with the addition of a handshake signal from the PIC18F processor to synchronize the two processors during data transfers. The *IPC_Initialize* method initializes the software state machine and hardware interface signals for IPC on each processor. *IPC_Processor* is then called by the *procComms* Task to manage the messaging process.

Message Structure and Protocol

The basis of communications between the two processors consists of packets with the following structure:

- Command: one byte (0x00..0xFF) identifying the command
- Length: one byte specifying packet length of up to 200 bytes (determined by PIC18F processor memory limitations)
- Parameters: 0 to 196 bytes of command parameters
- Checksum: one byte of a Fletcher 8-bit checksum to verify packet integrity

A message transfer is implemented in the dsPIC using the *IPC_Message* method which starts by lowering the PIC18F Chip Select (CS) signal and waiting for the Hand-Shake (HS) signal to go high, indicating that the PIC18F processor is ready to receive data. Each byte of the message is sent with the PIC18F processor toggling the HS signal between bytes to maintain synchronization as shown in Figure 5.3.

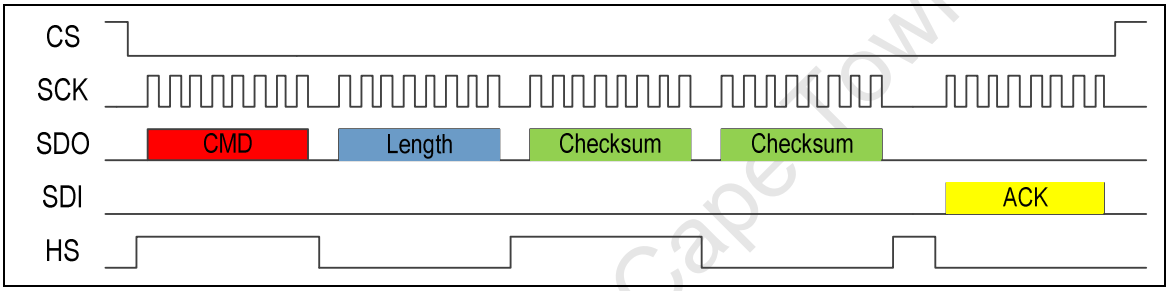


Figure 5.3 – Command message structure

The example above is a command without parameters or one expecting a response message from the PIC18F processor. All messages require an acknowledge (ACK) byte to be read by the dsPIC once the PIC18F processor has completed the executing command. The checksum of the received message is verified against a calculated checksum and if the command is successful the ACK bit in the returned byte is set. If the message is requesting data from the PIC18F processor the ACK byte contains a bit indicating this. The next byte read indicates the length of the message and the dsPIC reads these bytes into its internal buffer *ipcBuffer*. The returned bytes are verified using the checksum to ensure integrity before further processing. The message is ended by the dsPIC raising the CS signal.

When IPC is not in use, the HS signal is low. The PIC18F can indicate to the dsPIC processor that it has a message to send by raising the HS signal and initiating a status read. The *dsPIC:IPC_GetIPCStatus* method selects and transmits two zero bytes to the PIC18F process, reading in the *PIC18F:tipcStatusReg* value. This method does not require a full message to be sent and allows for a quick response to indicate if the PIC18F wants to send a USB packet or status message to the dsPIC.

IPC Commands

The dsPIC processor verifies communications with the PIC18F processor using a CMDIPC_PING command which responds with an ACK byte to indicate success. This command needs to be successful before the *IPC_Processor* will allow any higher-level functions to be performed.

The current measurement values are sent using the dsPIC*:IPC_UpdateMeasurements* method which sends the current time-of-day; Power Profiler configuration and status; RMS voltage and RMS current values to the PIC18F for display on the LCD. This message is sent every 800ms to maintain a smooth-running clock display on the LCD.

The dsPIC processor responds to message requests by the PIC18F processor to read status information (*dsPIC:IPC_GetPIC18FStatus*) and received USB packets (*dsPIC:IPC_GetUSBPacket*) when available, indicated in the IPC status register (*dsPIC:ipcStatus*).

Error Checking

IPC uses an 8-bit Fletcher checksum algorithm which is a simple and high-speed means of verifying data integrity [Fletcher, 1982]. CRC checking is not used as the possibility of data corruption is low because the processors are tightly coupled and it is a slower algorithm than the Fletcher checksum. The checksum algorithm can be seen in *PIC18F:IPC_CalculateChecksum*. The integrity of all communications between the Power Profiler and the Host is verified using Cyclic-Redundancy Checking (CRC) and it is only for communication between the two processors that the Fletcher checksum is used.

5.3.3. Bluetooth Communications

The Bluetooth interface to the Power Profiler is implemented in the *BluetoothInterface* class defined in *Profiler.Comms.Interfaces.cs*. The Bluetooth software stack running on the KC-Wirefree module and the PC USB Bluetooth interface uses the Serial-Port Profile (SPP) to establish a virtual serial-port link between these two devices. The *BluetoothInterface* class uses a *SerialCommsPort* class defined in *Profiler.Comms.Mediums.cs* to manage and transfer data over the Bluetooth SPP link.

Interface Initialization

The KC-Wirefree module in the Power Profiler is pre-configured to allow pairing (without requiring a PIN) and to support an SPP link. The *dsPIC:comms_Bluetooth* files implement the management and data transfer interface to the KC-Wirefree module. Data received by the dsPIC UART interrupt is passed to the *BT_CommsRx* method which checks the received string for an SPP connection or disconnection AT instruction. On the PC the *BluetoothInterface* class opens the serial port by calling the *SerialCommsPort:Open* method with the port name and baud rate parameters. This causes the Bluetooth software stack to open or close the SPP link which is detected by the dsPIC as described above. When the

PC application is closed the *SerialCommsPort:Close* method is called which disconnects the SPP link and places the KC-Wirefree interface back into AT command mode.

Data Transmission and Reception

Data is transmitted from the Host to the Profiler via a command or request and is received in the dsPIC UART interrupt handler. Received bytes are passed to the Communications Task's *Comms_RxByte* method to check for a SPECC packet.

Data received by the PC over the SPP link it is passed by the *SerialCommsPort:DataReceiveHandler* event to the *BluetoothInterface* class which in turn sends it for processing by its protocol handler as discussed in Section 5.6.2.

5.3.4. GSM Communications

The dsPIC processor interfaces with the Telit GC-864 GSM modem via serial UART 1 using an AT command interface [Telit, 2010]. The GSM modem interface is initialized in *GSM_Initialize* and the software task to control and send or receive data is implemented in *GSM_Processor* which is frequently called by the Communications Task.

The *GSM_Initialize* method initializes the GSM modem interface to 57600 baud and disables the data sending and socket timeouts to speed -up throughput when sending data to the Host. The interface to the GSM modem is either in command or data modes. The command mode is used to verify network registration, check modem state and to control the GPRS connection for TCP/IP data transfers. The Power Profiler creates a listening socket to listen for incoming TCP connections from the Host and once a connection is opened the modem enters data mode. Any data transmitted to or from the Power Profiler is sent as TCP packets by the GSM modem stack.

GSM Modem Control and State Checking

Every five seconds (specified by *gsmSecCount*) the *GSM_Processor* Task checks the state of the GSM module by calling the *GSM_CheckConnection* method which performs several tasks:

1. Call *GSM_CheckPIN* to verify that the PIN has been correctly entered. The PIN code is set in the Profiler Configuration and stored in the *gsmPIN* variable, but PIN code checking can be disabled on the SIM card.
2. The signal strength is checked by calling *GSM_CheckSignal* and is used for diagnostics reporting
3. The state of the GPRS connection is checked and if necessary is activated using *GSM_ActivateGPRS*

GPRS Context Activation

To establish a GPRS connection an Access Point Name (APN) needs to be specified when the GPRS context is opened to indicate to the Service Provider what data interface is permitted. The typical APN for Vodacom subscribers is 'internet' which allows Internet access but hides the modem behind a firewall allowing a connection to be initiated from the modem's side only. Vodacom provides private APNs with SIM cards configured with static IPs to allow any node to contact another node on an address that does not change. For the purpose of testing the Power Profiler GSM communications a private APN was used to allow the Host computer to contact the test Power Profiler and retrieve data.

The *GSM_ActivateGPRS* method issued an 'AT+CGDCONT' command to the modem, specifying the APN name retrieved from configuration memory *(apnName)* and that static IP addressing will be used. The Telit GSM modem's internal firewall is disabled using the 'AT#FRWL=1,"1.1.1.1","0.0.0.0"' command to allow unrestricted packet-based data access and the context is opened using the 'AT#GPRS=1' command. The command returns an 'OK' AT response if the context was activated successfully.

TCP Data Communications

After activating the GPRS context a TCP listening socket is opened by calling the *GSM_Listen* method using the port specified by *gsmListenPort*, which is retrieved from configuration memory. The PC Host implements a *GSMTCPInterface* class which uses the SPECC protocol over a TCP client to communicate with the Power Profiler.

A connection is opened using the *GSMTCPInterface:Open* method and specifying the IP address of the Power Profiler and the port to access. The method will return a value of true if the connection is established or false if the connection failed. Once the connection is established commands and requests can be performed in the same manner as via Bluetooth or USB interfaces. The TCP Client is disconnected using the *GSMTCPInterface:Disconnect* method which closes the TCP interface.

5.3.5. USB Communications

The Profiler's USB interface is implemented as communications device class device which allows the PC to interface with it using a virtual serial port driver which simplifies interface complexity and future software maintenance. *Comms_USB.c* implements the interface between the PIC18F's Communications Task and the Microchip Application Library's USB software stack which controls the processor's USB transceiver.

The USB interface is initialized by the Communications Task calling *USB_Initialize* which calls the Microchip stack initialization routine *USBDeviceInit*. *USB_Processor* is called by the Communications Task and calls the USB stack's *USBDeviceTasks* method to handle connection requests and data transfers.

USB Data Transfers

When the Profiler is connected to a USB Host a virtual COM port is created by the Windows operating system which can be used in the same manner as a serial port implemented for Bluetooth SPP profile. Data transmitted to or from the Power Profiler is managed by the USB stack and appears as a transparent serial data connection. A *USB_VCPInterface* class is used on the Host to implement the serial port interface and uses command and requests to manage the data transfers via the SPECC protocol.

The PIC18F's *USB_Processor* sub-task checks when data is available from the USB interface and moves it to the *USB_In_Buffer*. As all Host-Profiler communications are handled by the dsPIC processor, data received on the USB interface is passed to the dsPIC for processing. The *usbDataWaiting* flag is set to indicate to the Communications Task that USB data is available and the received packet is transferred using the Inter-Processor Communications (IPC) described in Section 5.3.2.

Data to be sent from the Profiler back to the Host is transferred from the dsPIC to the PIC18F using an IPC command. The IPC header and checksum is stripped from the IPC message and the USB data is transferred in blocks of up to 64-bytes long to the *USB_Out_Buffer* where the *USB_Processor* sends it to the Host for processing.

5.3.6. Sensor Network Communication

The MAX3100 SPI to UART protocol converter is intended to support a future intelligent sensor network interface. The MAX3100 is initialized into a shutdown mode at start-up to reduce power consumption.

5.4. Configuration and Control Application

A C# .Net 4 PC application was developed to calibrate, configure and test the functionality of the Power Profiler. The GUI was implemented as an MDI application to allow the loading of multiple graphing windows for future data analysis functionality and the underlying control interface to the Power Profiler is implemented as several object-oriented classes to support re-use in other applications.

The class implementation is split into five namespaces:

- *Profiler.Base* which contains the base classes all Power Profiler classes inherit from and adds diagnostics messaging capabilities
- *Profiler.Comms.Mediums* which provides the low-level communication interface classes for serial port and TCP/IP access
- *Profiler.Comms.Interfaces* which makes use of the *Profiler.Comms.Mediums* classes to provide higher-level managed communication interface such as Bluetooth, GSM and USB
- *Profiler.Comms.Protocols* which implements the interface protocol used for communication

- *Profiler.Devices* which contains the classes that interface with the Power Profiler embedded software at a device level

Splitting the software across several class implementations and the use of object-based programming techniques such as inheritance supports code-reusability and ongoing development of the Power Profiler product. The code listings are well commented and available in Appendix C.

5.4.1. The Power Profiler Class

The *Profiler_Rev1* class implements the full interface between the user application and the Power Profiler. It encapsulates the communication, control and calibrated measurement retrieval functionality and exposes several methods and parameters that the user application can access. To include a Power Profiler interface and initiate communication with it a user application can use the code shown below:

```
Profiler_Rev1 profiler = new Profiler_Rev1();     // Create the Power Profiler class
profiler.Connect_Bluetooth("COM1");               // Open a Bluetooth connection on COM1
if (profiler.Login(0xFFFFFFFF, ""))               // Log in with ID 0xFFFFFFFF and no password
  profiler.Initialize();                          // Synchronize the class and Power Profiler
```

The process above creates a *Profiler_Rev1* class, opens the communications interface when logged in with the correct ID and password the class is initialized to retrieve the calibration and configuration information from the connected Power Profiler. The user application is now able to interact with the class using method interfaces for device configuration and measurements access.

The main functionality of the *Profiler_Rev1* class is split into the functional areas of communication, control, configuration, calibration and measurement access. For the purposes of the subsequent discussion the *Profiler_Rev1* class will be shortened to the *Profiler* class.

Communication

The *Profiler* class uses a *CommsInterface* class to implement the communication interfaces described in Section 5.3. This class exposes the abstract methods Connect, Disconnect, Command and Request which are used to communicate with the Power Profiler irrespective of what descendent interface (Bluetooth, GSM or USB) is actually used. This results in simpler code implementation as the necessary protocols and hardware interfaces required are not exposed to the Profiler class.

Communication is based on the principle of commands and requests. Commands consist of an instruction with optional parameters and may or may not require a response from the device. A command returns a true or false Boolean response if the command was successful or unsuccessful. Requests issue an instruction with optional parameters and expect one or more bytes in response to be returned from the method. These methods are implemented as abstract classes which require their implementation in inherited classes with

the benefit being that any device classes using these communication interfaces are guaranteed to have a minimum functionality set, resulting in a simpler interface for future development.

Once communication is established with the Power Profiler using the appropriate Connect method the Profiler's *CommsInterface* class is then used for device access. Logging in with a valid ID and password is required before access to measurement data and control parameters is allowed. To verify communications a *Ping()* method is available, which can also be used to measure the latency of the communications link being used.

Configuration

Many operational settings of the Power Profiler are user-configurable and are accessed using several forms of the Configuration and Control application. Configuration and calibration memory access is also implemented as classes to simplify parameter encoding and decoding in the same way that configuration block definitions are used in the embedded code. Each configuration block has a defined interface class that accepts the parameters to be stored as public variables and implements a *Load* method to read and decode the configuration block and a *Store* method to encode and write the configuration block to the Power Profiler.

Calibration

The Power Profiler requires calibration against an accurate source or measurement meter before it can be used for data gathering. To calibration process is implemented as several method calls in the *Profiler* class. These methods are called by the Control and Configuration application and when completed, the calibration results are stored in the Power Profiler configuration memory. The calibration process is described in Chapter 6.

Control

To manage and initialize the Power Profiler several methods are implemented in the *Profiler* class which the user can use to determine the Power Profiler status and control the measurement process.

void GetStatus()

Issues a request to the Power Profiler and retrieves the current real-time-clock value and the memory pointers which are used to determine what measurement data can be uploaded.

bool SetClock(DateTime value)

Sets the Power Profiler's real-time-clock to the value specified which is normally the current GMT time.

bool ResetAcquisition()

Instructs the Power Profiler to reset all profile memory pointers and restart data acquisition.

bool StartAcquisition()

Starts acquisition after it has been stopped without affecting the profile memory pointers.

bool StopAcquisition()

Halts acquisition without affecting the memory pointers.

FLASHMemoryPointers GetFlashPointers()

Retrieves the Flash memory pointers and returns them in a data structure. This is used to determine what profile values are available for uploading.

bool GetCurrentMeasurements(ref Measurements value)

Retrieves the latest voltage, current, power, industrial inputs, temperature and frequency values from the Power Profiler and returns them as a *Measurements* data structure through the referenced 'value'. These values can be used for measurement display or for the calibration calculations.

5.4.2. New Profiler Initialization

The PIC18F and dsPIC30F software is downloaded to the Power Profiler using a Microchip Real-ICE Emulator or ICD2 programmer via the 8-pin Molex connector $CONN_{12}$. As the dsPIC's EEPROM settings are cleared upon initial programming, the Power Profiler will default to a device identity of zero with no password. These settings are updated through the Profiler class's *ProfilerIDCfg* parameter and written to the configuration memory.

5.4.3. Communication Settings

The communication settings are accessed through the Profiler class's *commsConfig* configuration block. The current parameters are retrieved and can be edited through the Configure Communications form shown in Figure 5.4.

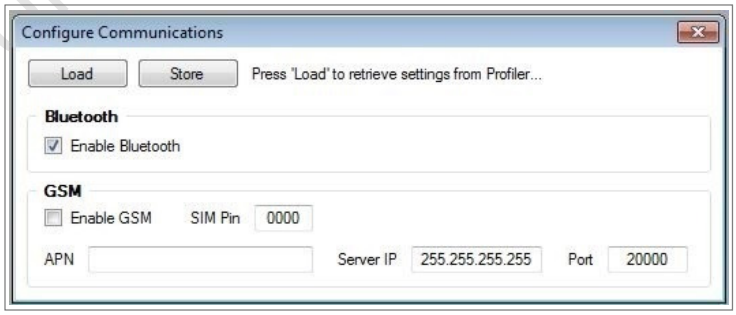


Figure 5.4 – Configure Communications form

Bluetooth is enabled by default and is the preferred local connection method as it offers electrical isolation between the Power Profiler and the user's system. To enable the GSM module (if it is installed in the Power Profiler) the user enters the SIM PIN code (for no pin it is 0000), Access-Point Name (APN) for connection,

server IP and server port. The configuration is updated in the Power Profiler by pressing the 'Store' button which updates the parameters in the *Profiler.commsConfig* class and updates the configuration memory.

5.4.4. Measurement Configuration

The measurement mode of the Power Profiler is set using the Configure Profiler form shown in Figure 5.5.

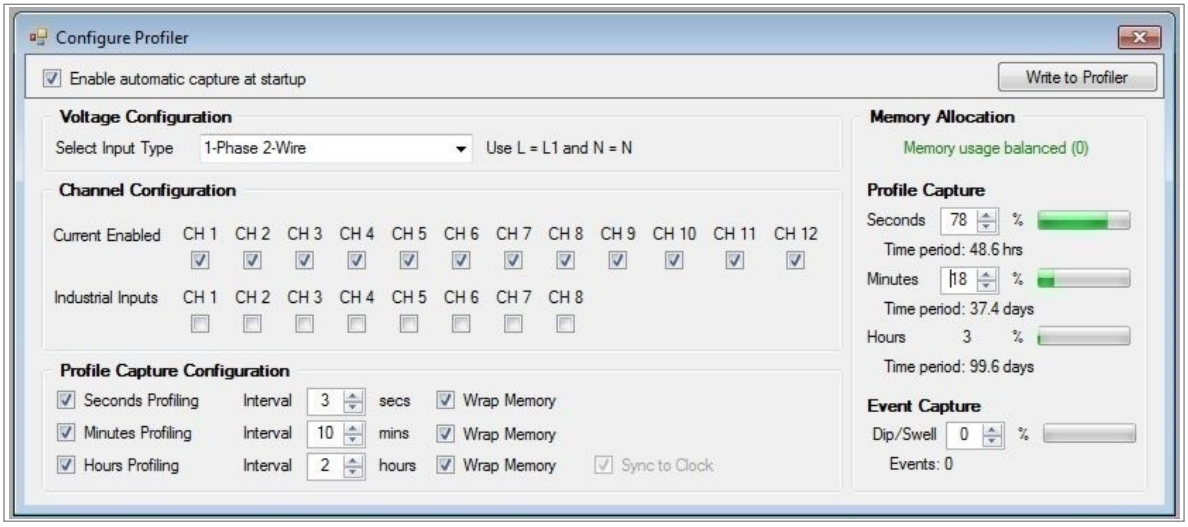


Figure 5.5 – Configure Profiler Form

The first checkbox sets the Power Profiler to automatically capture measurements when power is applied. In cases where more direct control is required (such as in a laboratory environment) this can be disabled to allow the user to start and stop measurement profiling through the Configuration and Control application's menus.

The voltage configuration setting allows the user to select between single-phase, 3-phase and custom configurations, specifying to the embedded software which calibration configuration to use for measurements. The channel configuration allows the user to specify which current and industrial channels are to be used. The seconds, minutes and hours profiles can be enabled or disabled, their storage interval rate set and whether to wrap the memory (return to the beginning and overwrite old measurements) when full or not. The hours profile will by default store measurements on the hour roll-over of the clock to synchronise multiple Power Profilers deployed in the field.

To maximize memory usage the memory allocation fields allow the user to specify the ratio of seconds, minutes and hours allocated to each profile and uses the configuration settings for voltage, channels and profiles to determine the memory requirement for each measurement stored. As the user adjusts the allocation percentage, the application will determine the total measurement period for each profile and display it. Events are not currently used in the Power Profiler and are therefore not allocated memory.

When the configuration is complete the user presses the 'Write to Profiler' button which updates the profiler class's measurement and memory configurations and updates the necessary configuration memory blocks in the Power Profiler. The configuration settings are read by the embedded software during the initialization of the *procMeasurement* software task.

5.4.5. Measurement Retrieval

Measurement data is uploaded using the Upload Measurements form shown in Figure 5.6.

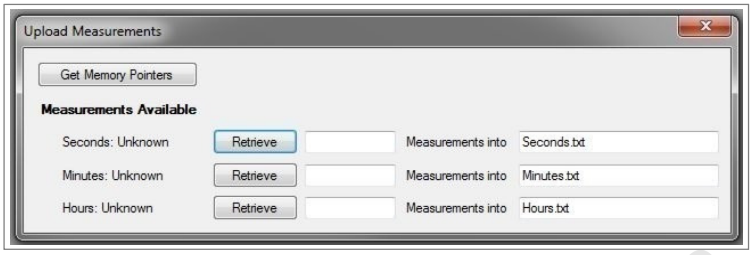


Figure 5.6 - Upload Measurements Form

The current memory pointers are retrieved using a call to the *Profiler.GetFlashPointers* method and determine the number of seconds, minutes and hours measurements values available for uploading. The user enters the number of measurements to retrieve in the text box next to the 'Retrieve' button and can specify a filename to store the CSV measurement data in. Pressing the 'Retrieve' button will upload the measurement data, convert it into actual RMS and power values using the stored calibration and conversion values and then write it into the specified text file as CSV lines. The stored CSV text file can be imported into an appropriate data analysis application (such as Microsoft Excel or MATLAB) and can be viewed in the Configuration and Control application.

A measurement is uploaded using a call to *profiler.GetNextMeasurement(<profile type>, ref <result>)* where *<profile type>* specifies which profile to access and *<result>* is the returned measurement value. The Power Profiler maintains a current read pointer for each profile and this is incremented to point to the next measurement when this request is issued. This makes measurement retrieval simple, but in applications where multiple systems may require measurement access more direct memory control will be required.

5.4.6. View Measurements

The File->Open Measurements menu item allows the user to load a CSV measurements file and display it in a chart window. A benefit of using an MDI GUI is that multiple measurement windows can be opened and the standard Windows tile and cascade functionality can be used as shown in Figure 5.7.

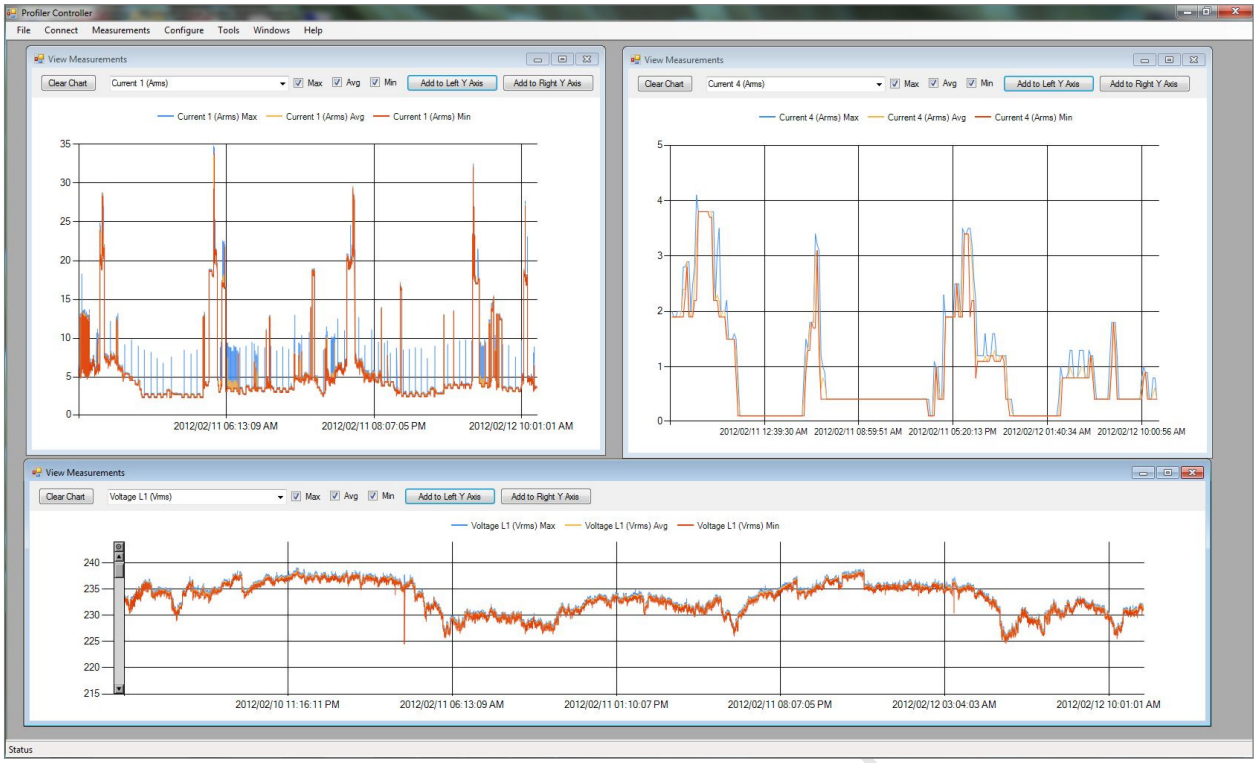


Figure 5.7 – View Measurements MDI Child Window

The measurement data is loaded from the selected CSV file and placed in a System.Data.DataTable class using the *LoadDataSet* method. The user is able to select the measurement channel from a drop-down box and whether the maximum, average and minimum values should be plotted on the left or right axis. Being able to plot on two different axis allows different measurement ranges (such as voltage and current) to be better represented. The chart interface supports zooming and can be cleared by pressing the 'Clear Chart' button.

5.5. Conclusions

This chapter presented the embedded and PC-based software for the Power Profiler. The embedded software is split into several Tasks making the code easier to maintain and to support future development. The PC software was implemented as C# classes that contain the communications, control and measurement retrieval and processing functionality. These classes are easy to use and will support future re-use in new applications developed for the Power Profiler.

The Control and Configuration application was also discussed which is integral to the Power Profiler configuration, calibration and measurement processing functions. Chapter 6 makes use of this application to test and calibrate the Power Profiler and in Chapter 7 field measurements are gathered for analysis and discussion.

Chapter 6

Testing and Calibration

The Power Profiler required functional testing, configuration and calibration of the measurement channels before field testing could commence. This chapter describes the calibration process and results obtained and in chapter 7 the Power Profiler will be field tested using the calibration settings obtained here. The calibration results are available in Appendix D.

6.1. Hardware Testing

During the software development stage the majority of the digital hardware was tested. Development of the prior prototypes resulted in the third integrated Power Profiler product having few design issues as the circuitry was designed from a known working base. The Control Board was tested using a laboratory DC power supply and signal generator. The switch-mode-power supply was tested by itself to confirm correct operation of the SMPS and battery charger before connecting it to the Control Board. Final calibration and verification was performed on a complete unit to verify integrated operation and to assess possible RF interference on the measurement circuitry from the switch-mode-power-supply.

6.1.1. Control Board

The first test of the assembled Control Board was to power it from a current-limited DC power supply to ensure the current consumption was within a normal operating range. High current draw would likely indicate a short-circuit fault, but this was not detected. The voltage outputs of the linear regulators were confirmed to be within specification and the micro-processors were connected to the Microchip In-Circuit Debugger (ICD) via $CONN_{12}$ to verify their correct operation. The pin-out of the ICD connector is given in Table 6.1.

Table 6.1 – $CONN_{12}$ ICD Pin Layout

Pin	Description
1	+5V digital power
2	Digital Ground
3	PIC18F Memory Clear (MCLR)
4	PIC18F Program Data (PGD)
5	PIC18F Program Clock (PGC)
6	dsPIC Memory Clear (MCLR)
7	dsPIC Program Data (PGD)
8	dsPIC Program Clock (PGC)

Further testing of the Control Board was done during development on a per-function basis: i.e. measurement, memory, communications and control.

6.1.2. Power Board

The power supply board was a first prototype and a few issues were found with its design. The SMPS was tested by connecting a variable AC transformer and slowly increasing the supply voltage up to the 230V AC operating level. During this test the output voltage was measured to ensure it was within the design specification of 6V for the Control Board supply and 10V for the battery charger. The voltage levels were correct and the battery power circuitry was tested next.

When the LiPolymer battery was connected to the charger, the charge/test cycles were clearly visible on the oscilloscope confirming the operation of the charger IC. The MCP73864 LiPolymer charger functions by applying a charge voltage for a period of time and then disconnecting it to measure the battery voltage. The charger will transition from fast- to trickle-charging based on the measured voltage level. With the battery charger operating at a current limit of 100mA the 10V charger supply was stable but if the current limit was increased towards the maximum of 1.2A a voltage dip of up to 1V during charge cycles was visible on the oscilloscope even though this output of the SMPS is designed for a load current of 1A. As the Power Profiler is connected to the primary side of the Utility connection (and not the Client's side) it is important to minimize current draw and therefore the charge current was kept at the minimum of 100mA. If more charging current is required the SMPS supply could be improved by increasing the winding size for the secondary winding of the transformer (to be confirmed in laboratory test).

Common-cathode diode D_1 was intended to supply power from the battery when the main supply failed but because the battery voltage of 7.4V is higher than the switch-mode supply voltage of 6V power for the Control Board would permanently be taken from the battery. The work-around during testing was to disable MOSFET Q_1 to disconnect the battery from diode D_1. The proposed solution to correct this is to use a comparator to maintain Q_2 low while sufficient voltage is available on the main power supply. The simplest approach would be to increase the output voltage of the SMPS to higher than the battery voltage but this would result in additional heat dissipation in regulator IC_1 during normal use and require changes to the custom-manufactured fly-back transformer.

6.2. Power Profiler Initialization

Before use a newly-assembled Power Profiler requires programming of the microprocessors and configuration and calibration using the PC Control and Configuration application. The latest version of the PIC18F and dsPIC firmware is downloaded to the microprocessors using a Microchip REAL-ICE, ICD2 or ICD3 programmer via the $CONN_{12}$ ICD connector. When the process is complete the power is cycled to reset the processors and the Power Profiler starts up in an initialized state waiting for a new configuration.

To facilitate a quick and simple configuration process a newly programmed Power Profiler has a device ID of '0', a blank password and Bluetooth communications is enabled by default. It is recommended that all configuration and calibration be done via the Bluetooth interface as this provides electrical isolation between the configuration computer and the Power Profiler. If it is necessary for the USB port to be used, an isolated USB hub is recommended to ensure electrical isolation for the computer.

After starting up the Control and Configuration software establish a Bluetooth connection to the Power Profiler by selecting the appropriate serial COM port as shown in Figure 6.1. The drop-down list is populated with all available COM ports at application startup, or by pressing the 'Refresh' menu item.

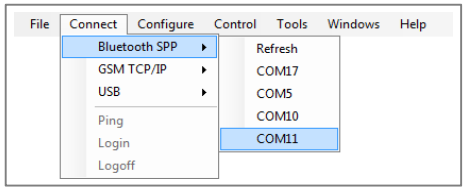


Figure 6.1 – Bluetooth Connection Menu Item

The status bar of the main form will indicate success or failure when opening the serial port. Once connected, communications can be verified by selecting the *Connect* > *Ping* function which will return the latency of the communications interface in the status bar. Select *Connect* > *Login* and enter the appropriate ID and password on the form that appears. Login success or failure will be indicated in the status bar. Only after a successful login can the user access the full functionality to perform calibration, configuration and to retrieve measurement data.

To set a new Power Profiler ID and password select the *Configure* > *Set Profiler ID and Password* function from the main menu and enter the new ID and password on the form that appears. This data will be saved in the configuration memory of the Power Profiler.

6.3. Calibration Process

The calibration process for the Power Profiler consists of the following:

1. Determining the voltage and current RMS offset values according to the process described in the ADE7758 data sheet [Analog Devices, 2003]
2. Calculating the conversion factors to convert the ADE7758 RMS byte representation into actual measured voltage and current values
3. Calculating the conversion factors to convert the ADE7758 energy measurement byte representation into the corresponding active, reactive and apparent power values
4. Calculate the industrial input channel conversion factors to convert the dsPIC byte representation into actual measured values

Calibration values are separated into those that are device-specific (such as the energy IC offset values) and those that are measurement-specific. To achieve this the Power Profiler implements four energy IC calibration blocks and four measurement calibration blocks for 1-Phase 2-Wire, 3-Phase 3-Wire, 3-Phase 4-Wire and a custom configuration. The configuration locations and class structures used to represent the contents is shown in Table 6.2.

Table 6.2 – Calibration Configuration Memory

Position	Blocks	Description	Class Structure (Profiler.Devices.cs)
128	5	Channel calibration data (general)	ChannelCalibrationConfig
352	2	Energy IC 1 calibration data	EnergyICConfig
416	2	Energy IC 2 calibration data	EnergyICConfig
480	2	Energy IC 3 calibration data	EnergyICConfig
544	2	Energy IC 4 calibration data	EnergyICConfig
608	7	1-Phase 2-Wire conversion calibration data	MeasurementCalibrationConfig
832	7	3-Phase 3-Wire conversion calibration data	MeasurementCalibrationConfig
1056	7	3-Phase 4-Wire conversion calibration data	MeasurementCalibrationConfig
1280	7	Custom setup conversion calibration data	Presently not used (not allocated)

Every 200msec the Power Profiler automatically updates an internal representation of the current measurement values (rms, power etc.) which are retrieved using the *Profiler.GetCurrentMeasurements* method. These values are compared against a known calibration source to determine the correct conversion factor per channel. The measurement process is mostly automated and performed by the 'Calibrate Power Profiler' (frmCalibrate.cs) form shown in Figure 6.2.

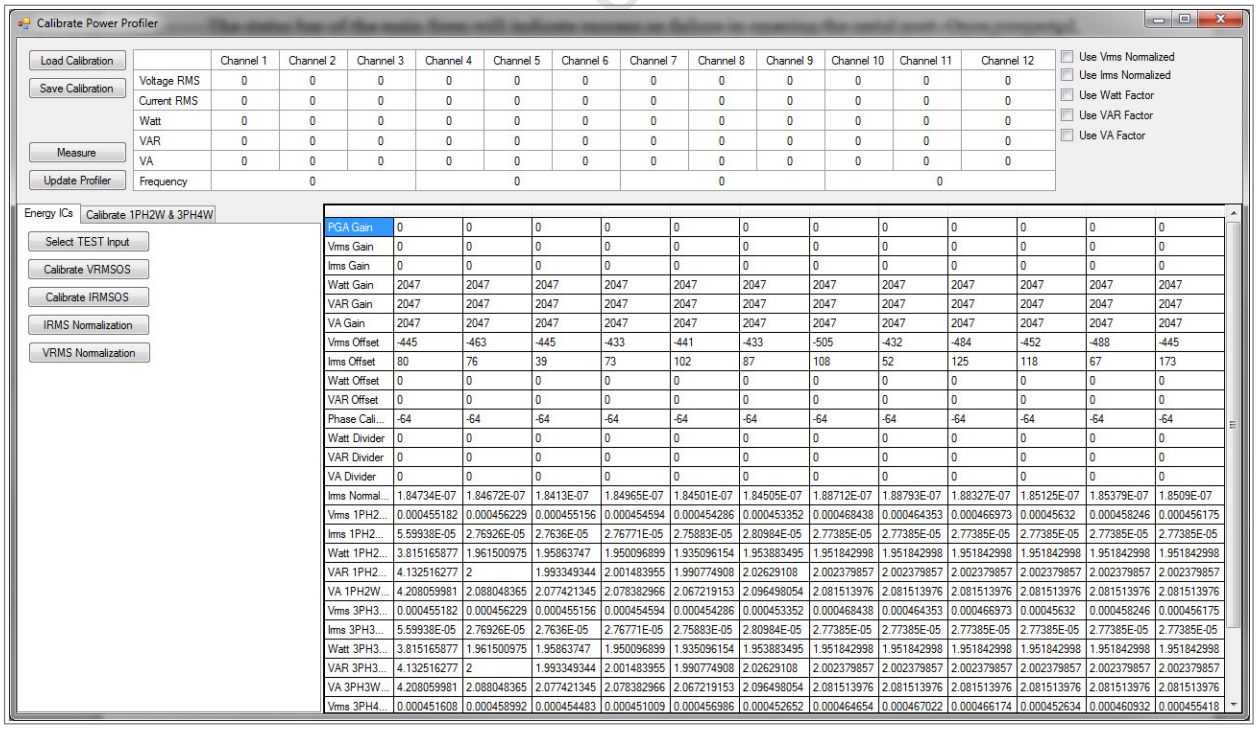


Figure 6.2 – Calibrate Power Profiler Form (frmCalibrate.cs)

The ‘Load’ and ‘Save’ calibration buttons allow the user to retrieve and store the current calibration parameters from the table into a text file. This is an extremely useful feature during development as it allowed the calibration process to be interrupted and later resumed without requiring a restart of the process. The ‘Measure’ button retrieves the current measurement values and displays them in the measurement grid (top). The checkboxes alongside the measurement grid allow the user to view measured data in raw or with post-conversion factors applied. Once all calibration is complete the ‘Update Profiler’ button will download the full calibration to the Power Profiler’s configuration memory. The calibration values for all 12 channels are accessible in the lower table and can be edited by the operator.

6.3.1. Energy IC Calibration

Analog Devices provides a recommended calibration process for the ADE7758 Energy Measurement ICs in the part’s data sheet [Analog Devices, 2011] and consists of RMS offset calculation followed by calibration of the energy measurement accumulation registers. The process is automated by the Calibration form and will prompt the operator to appropriately configure the signal source and will then execute the necessary calibration steps and update the parameter grid.

The calibration settings for each energy measurement IC consist of the following:

- Analog channel PGA, voltage RMS, current RMS, Watt, VAR and VA gains
- Voltage RMS, current RMS, Watt and VAR offset values
- Voltage channel phase calibration
- Watt, VAR and VA register divider settings

The voltage and current input channels are designed for the maximum input range of the ADE7758 so the PGA gain and RMS voltage and current factors are set to unity. The energy gain values are set at maximum (2047) as the data sheet states that at 50Hz with full voltage and current input signal levels the energy registers will overflow in 0.524 sec [Analog Devices, 2011]. As the dsPIC processor reads the accumulated energy values every 0.2 sec this gain setting maximizes the measurement range.

The RMS offset values (VRMSOS and IRMSOS) are used to adjust for part-to-part inaccuracies and are calculated according to the following equations [Analog Devices, 2011]:

$$xVRMSOS = \frac{1}{64} * \frac{V_{NOM} * VRMS_{VMIN} - V_{MIN} * VRMS_{VNOM}}{V_{MIN} - V_{NOM}}$$

Where:

V_{MIN} is the full scale voltage / 20 = 355 mV_{RMS} / 20 = 17.75 mV_{RMS} = 25 mV_{pk}

V_{NOM} is the nominal line voltage = 235 mV_{RMS} = 166 mV_{pk}

$$xIRMSOS = \frac{1}{16384} * \frac{({I_{TEST}}^2 * {IRMS_{IMIN}}^2) - ({I_{MIN}}^2 * {IRMS_{ITEST}}^2)}{{I_{MIN}}^2 - {I_{TEST}}^2}$$

Where:

I_{MIN} is the full scale current / 500 = 355 mV_{RMS} / 500 = 710 μV_{RMS} = 1 mV_{pk}

I_{TEST} is the test current = 10A = 47mV_{pk}

Calibration is performed using an Agilent 33220A 20MHz signal generator to provide an accurate and stable 50Hz signal source. The signal generator is connected to the Test voltage input on the Control Board ($CONN_6$) and via external connectors to all twelve current channels. The Test input allows the voltage calibration signal to bypass the voltage dividers on the L_1, L_2 and L_3 inputs and be directly connected to the energy measurement IC voltage inputs via the multiplexers. The offsets are calculated by averaging 10 samples at each specified amplitude level and using the above equations for VRMSOS and IRMSOS.

The calibration process is the same for both VRMS and IRMS offsets and is as follows:

1. Set the waveform generator to 50Hz sine-wave and connect to the Test voltage input and all twelve current channel inputs
2. Press the 'Select TEST Input' button to apply the test input voltage to all energy IC channels
3. Select the 'Calculate VRMSOS' or 'Calculate IRMSOS' button, depending on which calibration to perform
4. The operator is prompted to set the signal generator to the specified low voltage
5. The Calibration software will perform 10 samples of the lower amplitude level
6. The operator is prompted to set the signal generator to the specified high voltage
7. The Calibration software will perform 10 samples of the upper amplitude level
8. The VRMSOS or IRMSOS value will be calculated and the parameter grid will be updated

To improve the accuracy of the Power Profiler the current channels can have a normalizing factor calculated to reduce channel-to-channel inaccuracy. Normalizing the current channels is done by pressing the 'IRMS Normalization' button and using a signal generator source connected to all current channels at a high- and low-amplitude (as in the offset calculation) setting. The calculated normalization factors are updated in the parameter table and can be edited by the operator.

The voltage channels also support a normalizing factor but this is not currently used and is defaulted to 1.0. Voltage channel measurement in single-phase configuration is always taken from energy IC 1 channel 1, and a three-phase measurement is taken from energy IC 1 channels 1 to 3. Any inter-channel voltage inaccuracy is compensated for during the energy calibration process as the energy ICs use their respective voltage inputs to calculate accumulated energy.

6.3.2. RMS and Power Calibration

After the energy measurement ICs have been calibrated conversion factors need to be determined to relate the measured binary values to real-world RMS and energy measurements. As the Power Profiler is intended for single-phase and three-phase 4-wire configurations the process of calibrating the meter is the same for both measurement configurations.

A single-phase 230V AC supply, variable transformer, variable resistive load and variable inductor are used during calibration. The supply Live is connected to all three voltage channels (L_1, L_2 and L_3) of the Power Profiler and twelve split-core current transformers with burden resistors are connected to the current channel inputs. The test setup is configured as shown in Figure 6.3 below.

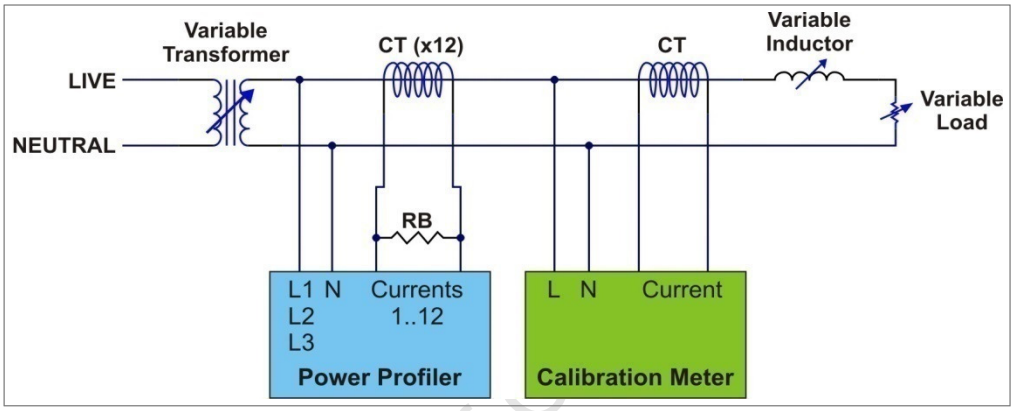


Figure 6.3 – Calibration and Test Setup

A Chauvin Arnoux Qualistar 8334B power quality analyser was used as the calibration meter reference for the Power Profiler. The 8334B is a high-accuracy 3-phase measurement unit capable of measuring and recording RMS voltage and current; active, reactive and apparent power; power factor and frequency. A comparison of the Qualistar versus the ADE7758 data sheet specified accuracy is given in Table 6.3.

Table 6.3 – Qualistar vs. Power Profiler Specified Accuracies [Chauvin Arnoux, 2003] [Analog Devices, 2011]

Measurement	Qualistar 8334B	ADE7758
RMS Voltage	±(0.5%+0.2V)	0.5% (20:1 range)
RMS Current (<1000A)	±(0.5%+0.2A)	0.5% (500:1 range) CT accuracy dependent
RMS Current (≥1000A)	±(0.5%+1A)	
Active Power	±(1%) for Cos Φ ≥ 0.8 ±(1.5%) for 0.5 ≤ Cos Φ ≥ 0.8	< 0.1% (1000:1 range) at 25°C
Reactive Power	±(1.5%) for Sin Φ ≥ 0.5 ±(2.5% + 20 pts) for 0.2 ≤ Sin Φ < 0.5	< 0.1% (1000:1 range) at 25°C
Apparent Power	±(1%)	< 0.1% (1000:1 range) at 25°C
Power Factor	±(1%) for Cos Φ ≥ 0.5 ±(1.5%+0.01) for 0.2 ≤ Cos Φ < 0.5	Not calculated on-chip
Frequency	±(0.01Hz)	0.0625 Hz per LSB

As the Qualistar is a power quality analyser it is expected that its accuracy would be higher than that of the ADE7758 which is a product aimed at metering applications. The ADE7758's application in metering is also the reason why its power measurement accuracy is very high as it requires accurate kWh measurement for billing purposes.

The Qualistar also uses high-accuracy Chauvin Arnoux MN93A current transformers intended for power quality analysis whereas the Power Profiler uses more general-purpose Taehwatrans TS10L CTs. The TS10L CTs are Class 1.0 accuracy rated and are intended for more general metering and data logging applications [Taehwatrans, 2012]. The specifications of these two CTs are given in Table 6.4.

Table 6.4 –Current Transformer Specifications [Chauvin Arnoux, 2003] [Taehwatrans, 2012]

Measurement	Chauvin Arnoux MN93A	Taehwatrans TS10L
Accuracy	≤ 0.7%	1% (Class 1)
Phase shift	≤ 0.7°	1.33° (80')
Frequency Range	48..65Hz	20..400Hz

A CT's phase shift property is particularly important as it has an impact on the accuracy of active and reactive power measurements. The MN93A probes use hall-effect sensors and have a lower phase-shift than the TS10L CTs. The ADE7758 does incorporate a phase shift compensation setting which can be used to apply a limited phase shift between the internal voltage and current waveform samples before they are used for power measurement calculations. Taehwatrans CTs are also available in higher accuracy ratings such as Class 0.1 which has a phase shift of 0.05° (3 minutes) [Taehwatrans, 2012] but these CTs are more costly and are typically intended for instrumentation applications.

Calibration Process

The calibration process is automated as far as possible and is accessed on the 'Calibrate 1PH2W & 3PH4W' tab of the Calibration form. The operator enters the reference meter values into the form and the calibration process performs 10 measurements, averages them and calculates the conversion factors which are updated in the parameter table. This process is used for both single-phase two-wire and three-phase four-wire setups. Due to fluctuating supply conditions calibration accuracy could be further improved using a stable speed-controlled motor/generator combination to produce a clean and predictable AC supply. Direct measurement from the calibration meter would also produce a closed-loop measurement environment which would result in higher measurement accuracy and a faster measurement process.

Voltage and Current RMS

To determine the voltage and current conversion factors the user sets the supply and load to a suitable voltage and current level at unity power factor and enters the measured values of the reference meter into the 'L' and 'I' text boxes on the Calibration form. The application then uses the average of 10 samples to determine the voltage and current conversion factors and updates the parameter table. The measured voltage

is applied to L_1, L_2 and L_3 making this calibration step applicable to both single- and three-phase measurements.

Active, Reactive and Apparent Power

The accumulated energy measurements are retrieved and reset from the energy measurement ICs every 200ms. These accumulated energy values need to be converted to appropriate 'Watt', 'VAR' and 'VA' measurements.

As with the RMS measurements, the operator sets the supply and load to a suitable measurement level at a power factor close to 0.5 inductive. The reference meter measurements are entered into the Calibration form and the calibration process is started by pressing the 'Calibrate Energy' button. Ten samples are averaged and the conversion factors are calculated and stored in the parameter table.

6.3.3. Calibration Results

During the calibration process the contents of the parameter table can be modified for experimentation or correction and when completed can be download to the Power Profiler configuration memory using the 'Update Profiler' button. The 'Measure' button can be used to display the latest measurement values with the parameter changes applied to verify correct calibration.

Calibrating the Power Profiler started as a very time consuming task but later evolved into a quick and flexible process through the ability to calibrate using automated steps, to store and retrieve calibration files, and being able to modify the parameter table before updating the Power Profiler. The final calibration values for the test Power Profiler are given in Table D.1 of Appendix D.

6.4. Verification Measurements

To determine the accuracy of the Power Profiler after calibration several measurements were taken and compared against the reference meter. To simplify this process a function was implemented on the Calibration form which allows the user to enter the voltage, current and power levels and to perform a measurement and store the results in a text file in comma-separated value (CSV) format. This text file could then be imported into Excel for measurement comparison and analysis. The tests were conducted at three differing power factors to effectively test reactive power measurements. For each test measurement five samples were taken and the values for all 12 channels (voltage, current and power) were written to the text file. These measurements were post-processed in Microsoft Excel to determine averages at each measurement level which could be compared against those performed at the differing power factors. In total 1440 measurements were processed per channel and these samples were rapidly gathered using the measurement file storage ability of the Calibration application.

The verification measurements were performed by setting the resistive load bank to create a load of approximately 9A at 230V. The variable transformer was adjusted to take several measurements down to 50V and the inductive load was adjusted to perform similar measurements at power factors of unity, 0.6 and 0.4 inductive. The measurements taken were compared to the Qualistar 8334B power meter which was used for calibration. The full results are available in Appendix D and are discussed below.

6.4.1. Voltage Channel Accuracy

The ADE7758 specifies an RMS inaccuracy of 0.5% for voltage channels over a 20:1 range from the calibrated nominal voltage of 230V. The average error at 230V was found to be 0.1% and over the 20:1 range approximately 0.4% which is well within the specified accuracy of the ADE7758. This error increases with decreasing voltage, becoming approximately 20% at 50V as shown in Figure 6.4. As expected, varying the power factor has minimal effect on the voltage measurement accuracy.

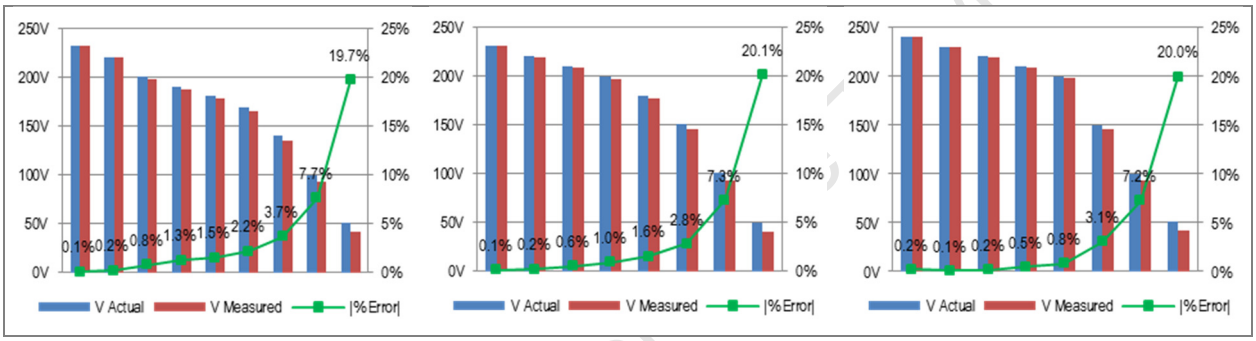


Figure 6.4 – Voltage RMS Verification Measurements at PF=1.0, 0.6 Inductive and 0.4 Inductive

6.4.2. Current Channel Accuracy

The Qualistar power meter reports current to a resolution of 0.1A whereas the Power Profiler reports it to 1mA. When rounded to one significant digit the Power Profiler data is generally consistent with that of the Qualistar meter, resulting in a zero percent error as shown in Figure 6.5. In cases where the percentage error is not zero percent, this is attributed to supply voltage variation and rounding inaccuracies. For example, the 1.9% error at 5.4A (PF=0.6) is reported by the Power Profiler as 5.331A versus the Qualistar's 5.4A measurement.

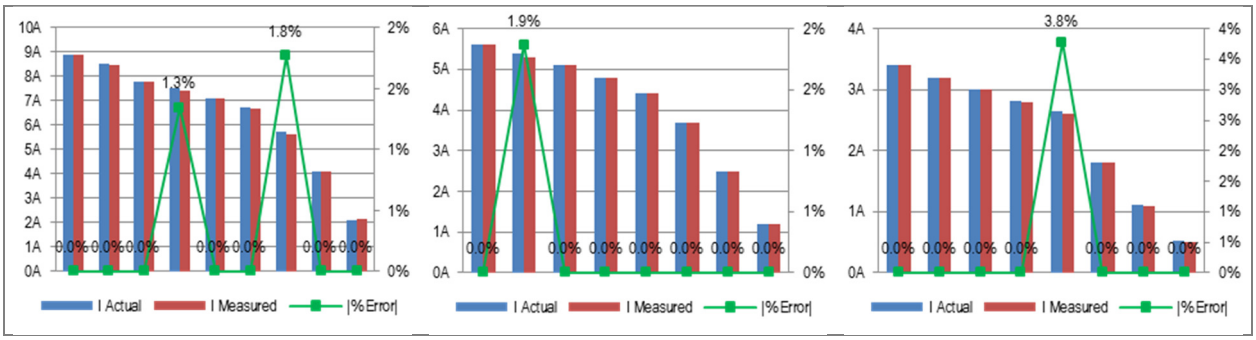


Figure 6.5 – Current RMS Verification Measurements at PF=1.0, 0.6 Inductive and 0.4 Inductive

6.4.3. Energy Calculation Accuracy

When evaluating the power measurement accuracy it is important to consider that the measured RMS voltage accuracy decreases significantly as the voltage level moves outside the 20:1 0.5% specified accuracy range of the ADE7758. A CT also has an inherent phase shift that impacts on the energy measurement accuracy, and even though this was compensated for by the ADE7758, higher accuracy will require higher-quality CTs or circuit modification of the Power Profiler voltage and current channel anti-aliasing filters.

The active power measurement profiles shown in Figure 6.6 show a measurement error of approximately 1.1% at the calibrated measurement voltage of 230V. This figure could possibly be improved through the use of higher-quality CTs to reduce the phase shift in the voltage and current measurement. An accuracy of 1.1% is sufficient for load discrimination purposes.

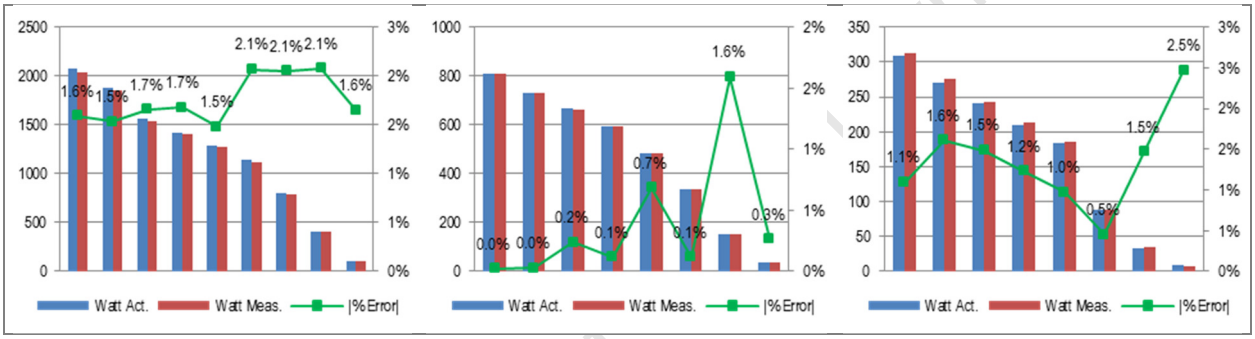


Figure 6.6 – Active Power Measurements at PF=1.0, PF=0.6 Inductive and PF=0.4 Inductive

A large discrepancy in the reactive power measurements (shown in Figure 6.7) between the Qualistar and Power Profiler at unity power factor is apparent. At unity power factor the reactive power should be zero but both instruments report an insignificant reactive power of ±15 VAR which is only 0.7% of the active power measurement of 2 100W. For this reason a percentage error calculation at unity power factor is not given. As the power factor moves from unity the reactive power measurement becomes relevant. At the nominal calibration voltage of 230V the reactive power error is 0.3% and is well within the accuracy required for load discrimination applications.

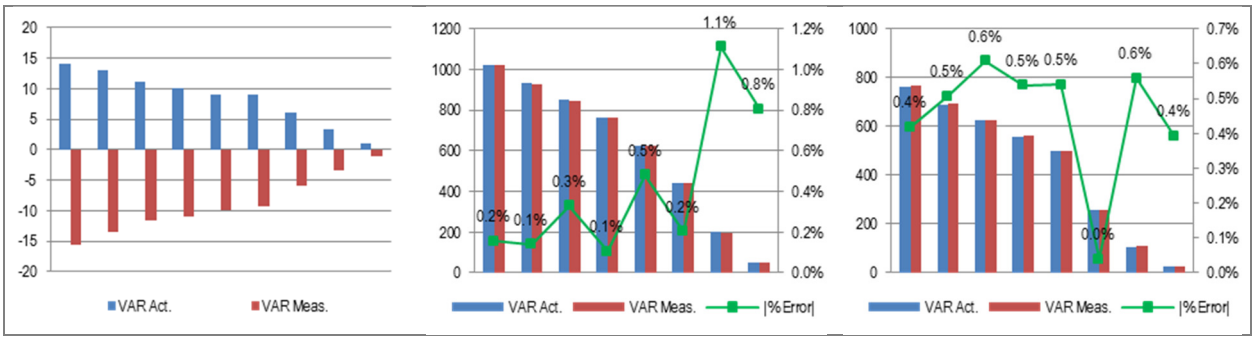


Figure 6.7 – Reactive Power Measurements at PF=1.0, PF=0.6 Inductive and PF=0.4 Inductive

The apparent power measurements are shown in Figure 6.8. At 230V the error percentage is calculated at less than 0.1% but as the voltage measurement level decreases, the error percentage increases. The phase-shift error of the CTs also has an insignificant impact on the apparent power measurement.

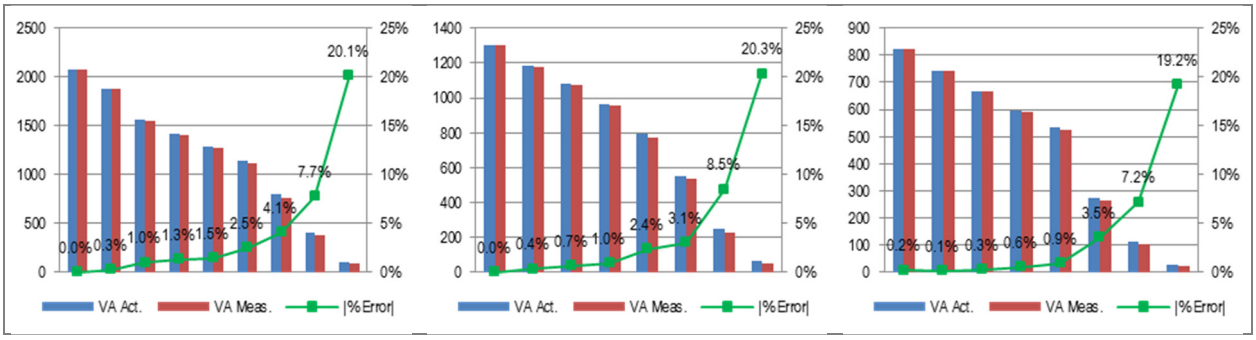


Figure 6.8 – Apparent Power Measurements at PF=1.0, PF=0.6 Inductive and PF=0.4 Inductive

6.5. Conclusions

The Power Profiler hardware was tested during the software development and calibration stages and found to be working correctly in order to achieve the objectives of the hypothesis. The battery management circuitry will need to be corrected in a future hardware release to improve charging current and mains-fail switch-over. The SMPS design was considered a high-risk item as it could only be verified for correct operation when a prototype was assembled and tested. Fortunately the SMPS design operated as required and the point-of-load regulation was within design limits and therefore no power issues were detected.

The automated calibration process simplified the testing and calibration of the Power Profiler and when you consider the accuracies achieved (versus the far more costly Qualistar power analyser) the Power Profiler performed very well. The voltage measurement inaccuracy at the nominal 230V was 0.1% and 0.4% over the full 20:1 range, well within the 0.5% specified inaccuracy of the ADE7758. Current channels typically showed a zero percentage error when compared against the Qualistar power meter, and where it was not zero it was attributed to supply variation and rounding errors due to the Qualistar reporting current at a lower resolution than the Power Profiler. Power measurement inaccuracy at nominal input voltage was found to be 1.1% for active power, 0.3% for reactive power and 0.1% for apparent power. The use of higher quality CTs such as Class 0.1 would produce less phase-shift in current measurement and result in far more accurate power measurements.

The ability of the digital signal controller and the energy measurement ICs to directly sample the voltage channel waveforms could also prove beneficial in power quality analysis applications where wide-range accuracy is desired. This confirms the dual-role of the Power Profiler as a cost-effective load analysis device with the ability to perform more advanced signal quality analysis processing through direct access of the raw input signals.

Chapter 7

Field Testing and Discussion

The calibration process in Chapter 6 determined the accuracy of the Power Profiler when compared to the Qualistar 8334B power meter. The purpose of field testing is to verify the correct operation of the Power Profiler's data gathering, storage, retrieval and post-analysis functionality. Field testing involved connecting the Power Profiler to a single-phase residential distribution board for several days to gather measurement data which was then uploaded for analysis.

To validate the Power Profiler measurement data the Qualistar power analyzer was simultaneously connected to the same residential load in order to capture several hours of measurements for comparison. As the two instruments are not time and measurement synchronized the measured data is graphically compared to see if any discrepancies occurred during the concurrent measurement process.

Once the Power Profiler was confirmed to be functioning correctly, a long-duration test was performed to gather substantially more data to allow for a more detailed analysis for residential load classification from the recorded power profiles.

7.1. Field-Testing Setup

The Power Profiler was configured to measure in 3-phase 4-wire mode with all live inputs connected to the single-phase 230V AC residential supply to simulate a three-phase test. Six current channels were enabled and Taehwatrans TS10L split-core CTs with a 3000:1 ratio were connected as listed in Table 7.1.

Table 7.1 – Power Profiler Current Channel Connection

Channel	House Load	Range	Burden Resistor
1	Main supply	106A	10Ω 1%
2	Main supply	53A	20Ω 1%
3	Oven circuit	53A	20Ω 1%
4	Lights circuit	53A	20Ω 1%
5	Geyser circuit	53A	20Ω 1%
6	Plugs circuit	53A	20Ω 1%
7..12	Unused		

The measurement of the main supply current using two CTs was to test the same CT at two different ranges and to compare their measured profiles. The 8334B power analyzer was connected in a single-phase configuration measuring the main supply voltage and load current. During measurement verification it was found that although current channels 1 and 2 produce similar current profiles, their reactive power is

different due to more phase-shift attributed to the differing burden resistor values. 20Ω is a recommended burden resistor value by the CT manufacturer and produces a better phase response to that of the Qualistar meter with the TS10L CT. For this reason all analysis is based on channel 2s current and power profiles.

7.2. Measurement Verification

Several hours of measurement data was captured using both the Qualistar 8334B power quality analyser and the Power Profiler both connected to the same supply and load. The Qualistar was configured to record on a 5-second interval and the Power Profiler on a 3-second measurement interval. Approximately 4 hours of measurement data was gathered for comparison, totaling 2 780 Qualistar and 4 710 Power Profiler measurements.

7.2.1. RMS Voltage and Current

From the Qualistar and Power Profiler RMS measurements shown in Figure 7.1 it is apparent that they are very similar in both amplitude and profile. This confirms that the software functionality of the Power Profiler is correctly gathering measurements and storing them in flash memory and the data retrieval process correctly retrieved and post-processed the measurements after applying the stored calibration constants. Slight variations in waveform profile are considered acceptable as the two instruments are measuring data over two non-synchronized time intervals with different measurement averaging periods.

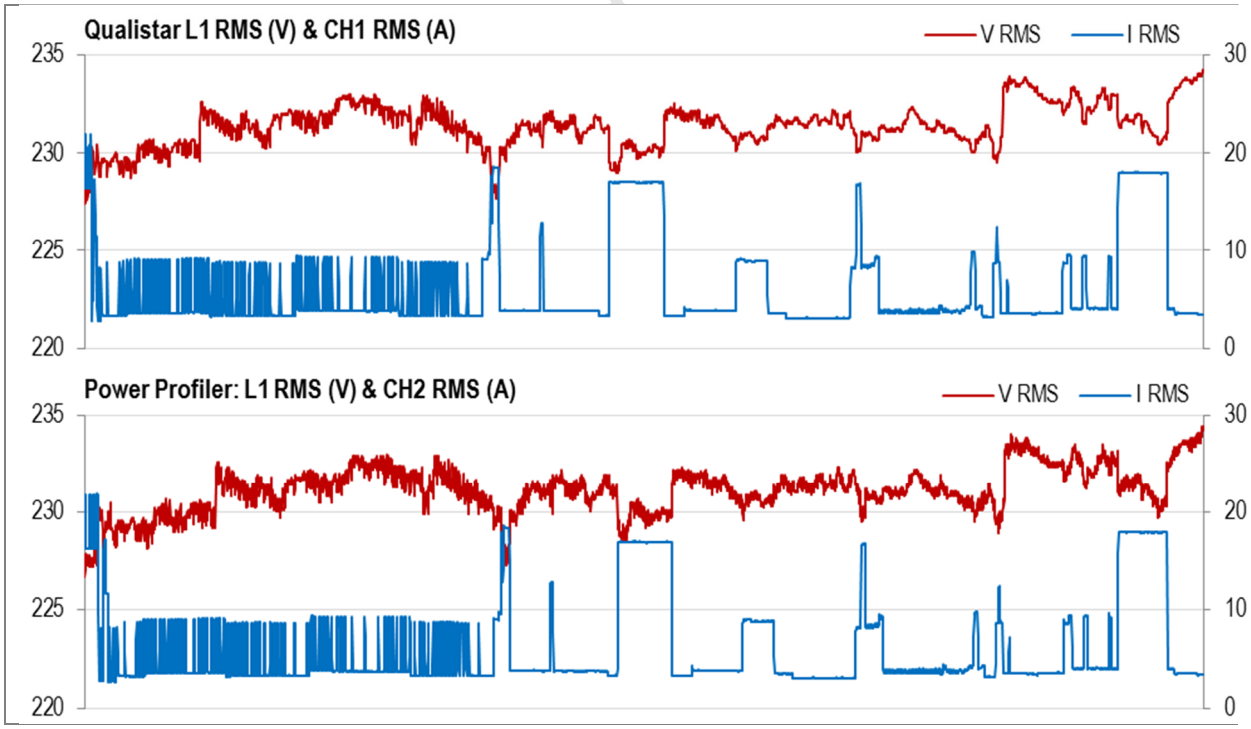


Figure 7.1 – Qualistar and Power Profiler Active, Reactive and Apparent Power Profiles

7.2.2. Power Measurements

Comparing the power measurement profiles shown in Figure 7.2 the active and apparent power profiles appear very similar between both instruments. Reactive power shows definite differences which are attributed to current transformer phase shift and accuracy as discussed in Chapter 6. The reactive difference is so small relative to the amplitude of the active power that it results in little impact on the active power measurement.

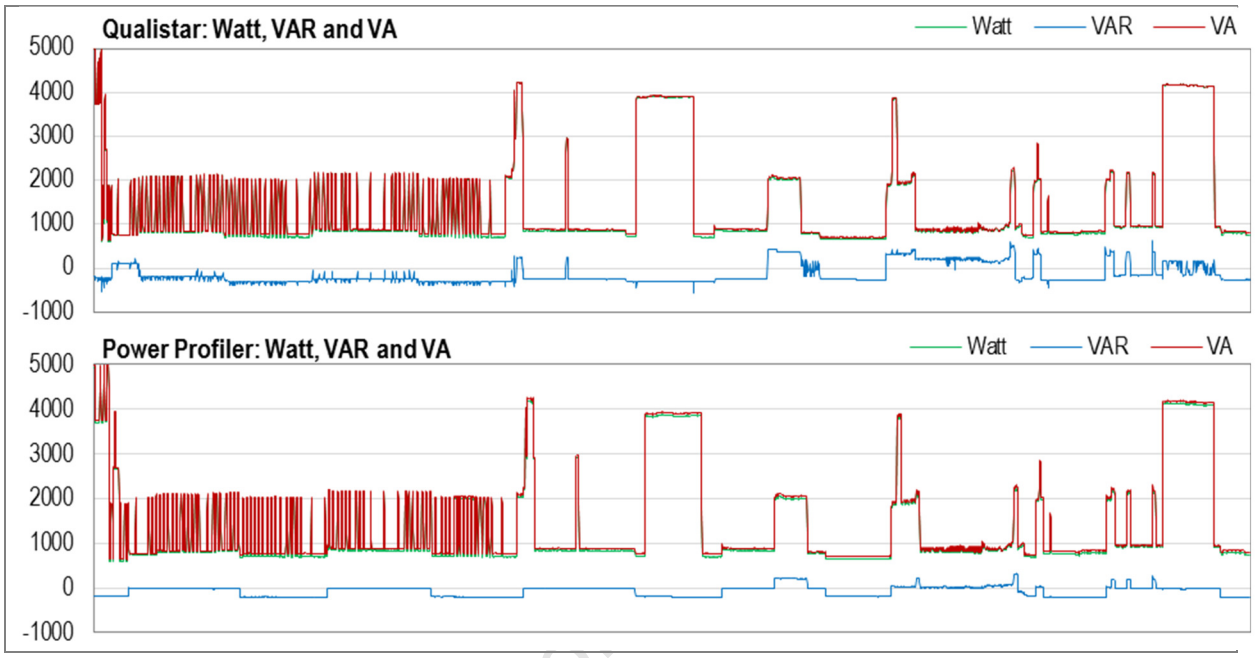


Figure 7.2 – Qualistar and Power Profiler Active, Reactive and Apparent Power Profiles

The impact of the phase shift is also evident when you consider the power factor profiles shown in Figure 7.3. The Qualistar reports the power factor as a measured value whereas the Power Profiler calculates it by dividing the active and apparent powers. Any phase shift or resolution error will therefore result in an error in the calculated power factor profile.

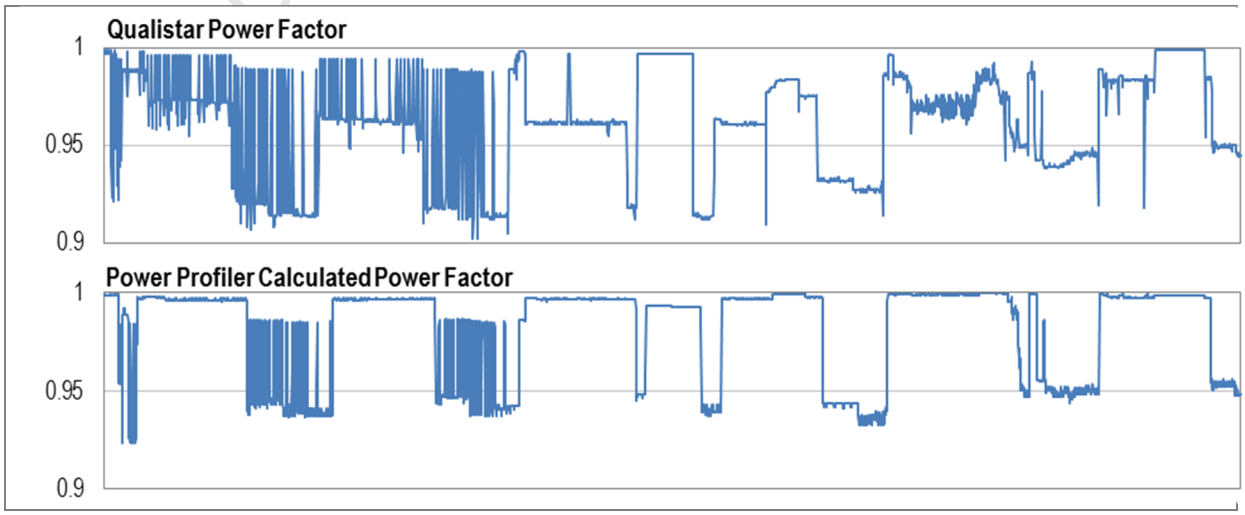


Figure 7.3 – Qualistar Measured and Power Profiler Calculated Power Factor Profiles

For power quality analysis applications the use of higher-quality measurement class CTs is recommended. For general load classification purposes the low-cost Taehwatrans CTs perform with sufficient accuracy to allow the operator to differentiate between resistive and non-resistive loads.

7.2.3. Frequency Measurements

The 8334B measures frequency with a resolution of 0.01Hz whereas the Power Profiler has a resolution of 0.1Hz. This results in the 8334B producing a smoother higher-resolution frequency profile than the Power Profiler as shown in Figure 7.4.

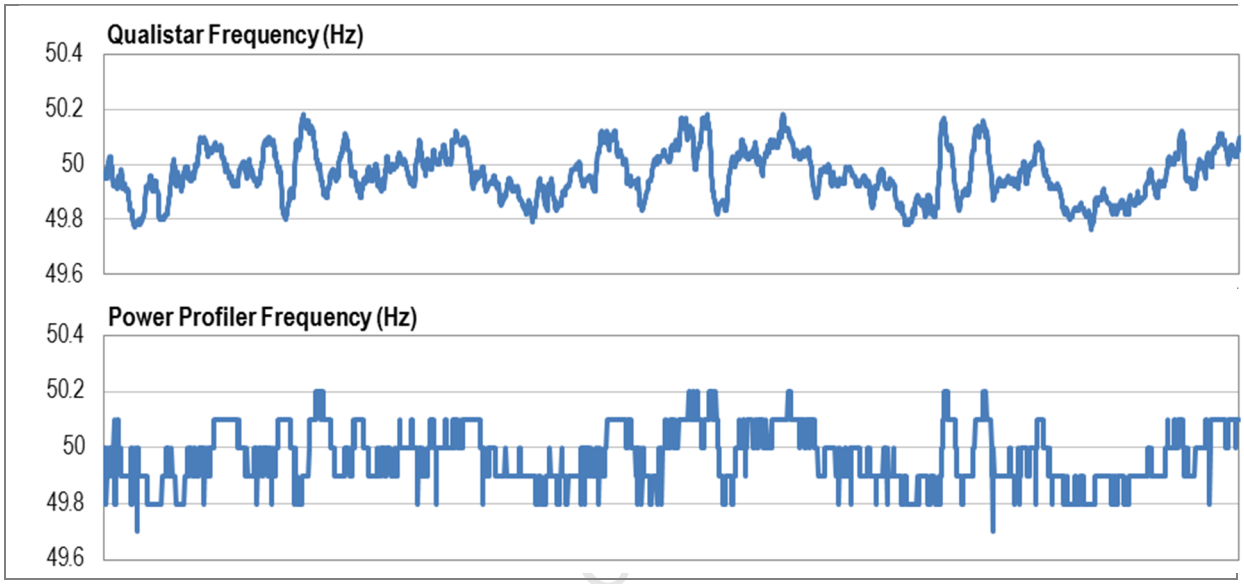


Figure 7.4 – Qualistar and Power Profiler Frequency Profiles

The achievable resolution for frequency measurement of the ADE7758 is 0.0625 Hz/LSB [Analog Devices, 2003]. For power quality analysis applications where higher accuracy frequency measurement may be required direct sampling of the voltage channel via the dsPIC or the ADE7758 is recommended.

7.2.4. Statistical Data

Most power quality measurement instruments record only the average value during the measurement interval whereas the Power Profiler records the maximum, average and minimum values. By including this additional statistical information it was hoped that it would add value to both power quality and load identification applications. By comparing the Qualistar and Power Profiler measurement including the statistical information, it is interesting to see what would be 'missed' using just an average measurement.

In the voltage channel profile shown in Figure 7.5 the small power dips and swells become evident (see 09:43 and 10:36) whereas they are not visible with just the average. This type of statistical data for the voltage supply is very useful for power quality analysis applications to detect supply disturbances.

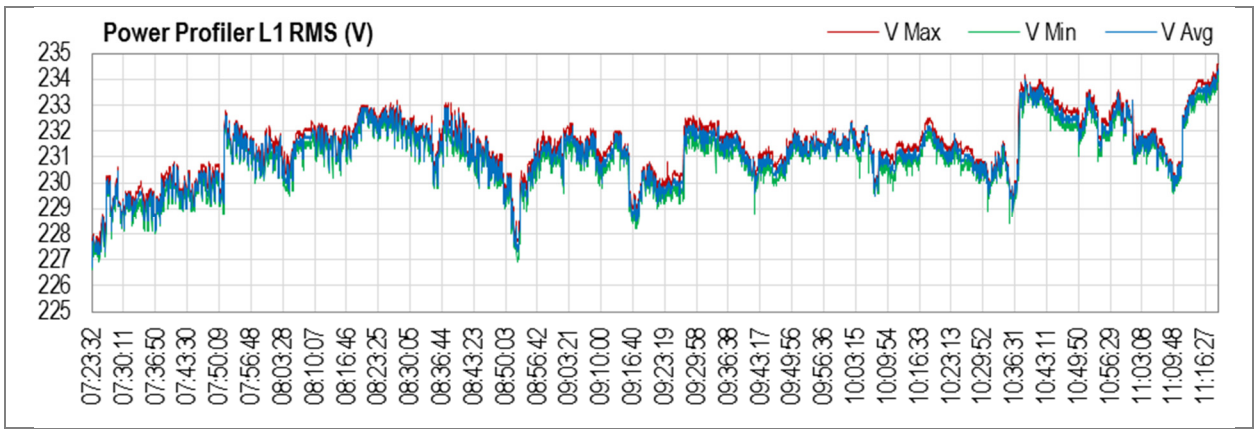


Figure 7.5 – Power Profiler Voltage RMS Profile with Maximum and Minimum

In the Power Profiler's current profile shown in Figure 7.6 the maximum current spikes are clearly visible and are typical of motor-based appliances due to their startup currents. For load identification purposes this maximum 'spike' allows the user to quickly identify appliances such as the fridge and washing machine from the current profile. At this 3 second measurement resolution the minimum statistical value offers little benefit to load identification and could possibly be disabled to better optimize memory usage in the Power Profiler.

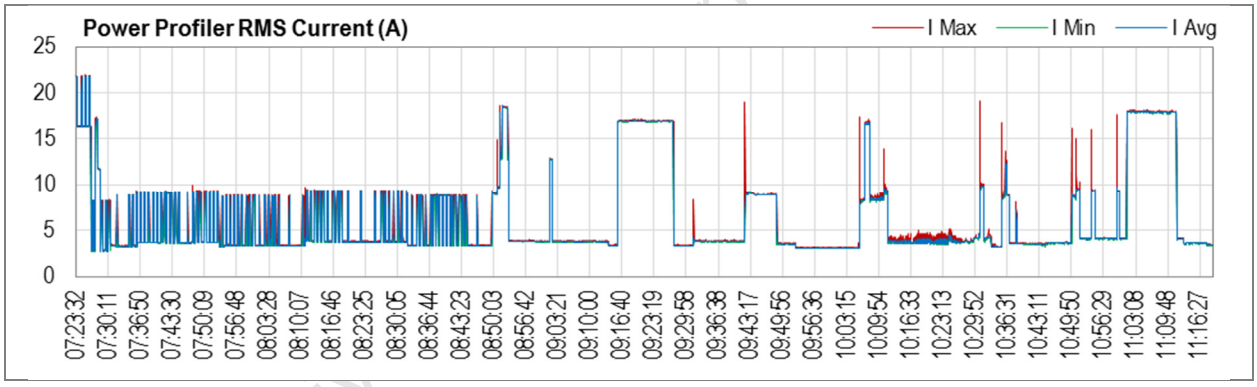


Figure 7.6 –Power Profiler Current RMS Profile with Maximum and Minimum

7.2.5. Verification Summary

The verification measurement tests prove that the Power Profiler software is correctly measuring, storing, retrieving and post-processing measurement data. The voltage and current measurements are very similar between both instruments but there is a visible difference between the two reactive power profiles which is attributed to the lower accuracy and phase shift characteristics of the current transformers used by the Power Profiler. As the active power is significantly larger than the reactive power this inaccuracy has little overall impact. These measurements are considered of a sufficient enough accuracy for load discrimination purposes but for power quality analysis applications the use of higher-quality CTs is recommended to reduce phase shift error and increase accuracy.

The inclusion of maximum and minimum data in the Power Profiler measurements is beneficial for both power quality and load analysis applications. Voltage spikes and dips become evident on the voltage profiles whereas they would be missed on the Qualistar data. The startup current of motor-based appliances is captured by the Power Profiler and allows the user to more easily identify these types of loads from the current and power profiles.

7.3. Field-Test Measurement

To gather sufficient data for load analysis the Power Profiler was connected to the residential distribution board and captured measurements for a 43 hour period using a 5-second, 10-minute and 2-hour measurement interval. The flexible measurement interval and the ability to capture all three measurement intervals at the same time gives the Power Profiler an advantage over conventional load monitoring and power quality analysis instruments which can typically only store one measurement interval. The second- and minute-based measurements are useful for both load identification and power quality analysis applications, and the hour-based measurements can be used for trending power usage over long periods.

The full measurement data gathered during field testing is available in Appendix E. A condensed version of the measurement data is presented here for discussion and is used for load determination in Section 7.4.

7.3.1. 5-Second Measurement Profile

Figure 7.7 shows the 5-second measurement data gathered on the main supply voltage channel during the test period. Because of the additional statistical measurement data the voltage dips at approximately 6am on the 11th and at 5am on the 12th are visible.

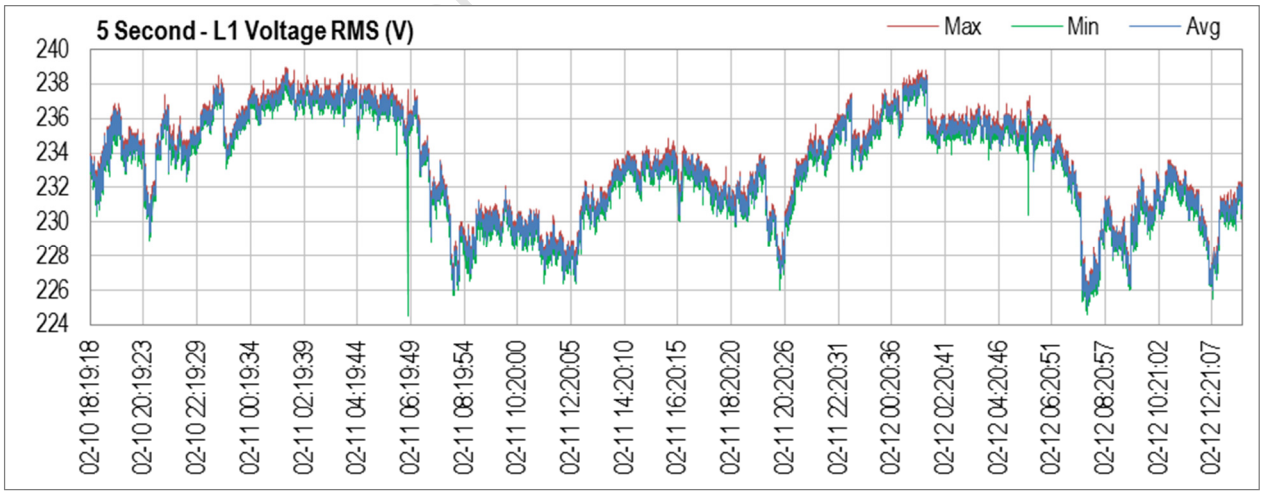


Figure 7.7 – 5 Second Main Supply Voltage Measurements

The average voltage supply over the measurement period was 233V_{RMS} with a recorded maximum of 239V_{RMS} and a minimum of 225V_{RMS}, giving a variation of approximately 3% from the average.

The voltage supply is also affected by the load current as can be seen in the voltage and current plot of Figure 7.8.

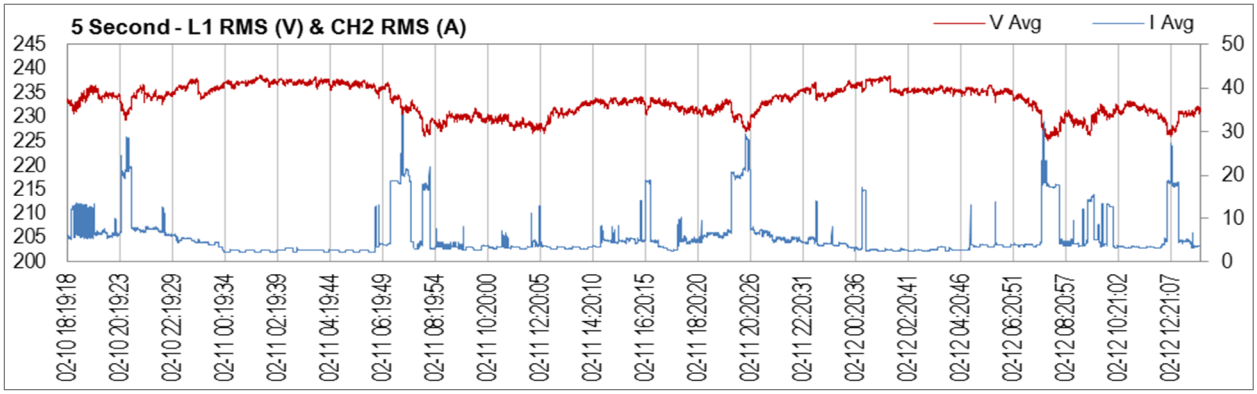


Figure 7.8 – 5 Second Main Supply Voltage and Current Measurements

During periods when the load current increases to approximately $20A_{RMS}$ the supply voltage drops by approximately $3V_{RMS}$. The residence used for the field test is an apartment and supply voltage fluctuations not related to high current usage can be attributed to the load of other apartments in the block and Utility supply variation.

When zoomed in on the two voltage dips and compared with the measured current profile as shown in Figure 7.9, we can see that the voltage dip is not related to the loads in the test residence and is most likely from the supply Utility or due to a fault condition in another residence connected to the complex's supply feed. It is interesting to see both a swell and dip at the same measurement point of the left plot. Given our measurement rate of 5 seconds and the sampling rate of 200msec, this dip and swell occurred during one of the 25 samples recorded during that measurement interval. The voltage drop due to increased load current within the residence is also visible in the left plot.

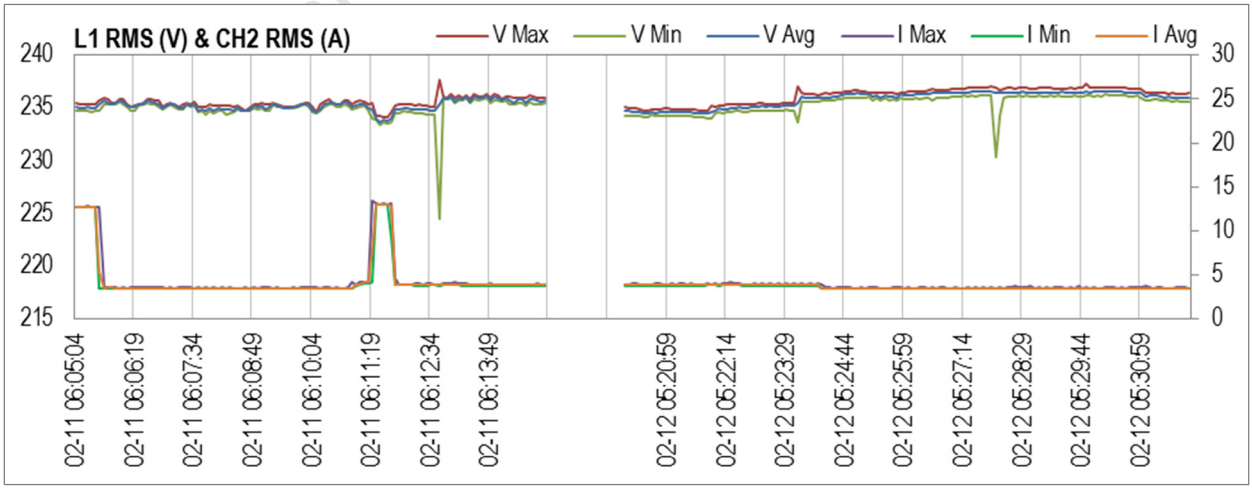


Figure 7.9 – Voltage Dips: Voltage and Current Plots

The value of the maximum and minimum voltage channel measurement to power quality analysis applications is evident from the fact that the two voltage events occurred during the test period would have otherwise gone unreported by a typical power meter.

The frequency plot for the test duration is shown in Figure 7.10. The supply frequency tolerance was typically less than 1% with several minimum values being reported down to 48Hz. These low frequency measurements are likely due to distortion of the supply waveform from switch-mode-powered loads such as computers, TVs and low-voltage lighting. The Qualistar was re-connected to the supply and a total harmonic distortion of 3% for voltage and 20% to 30% (load level dependent) for current was measured.

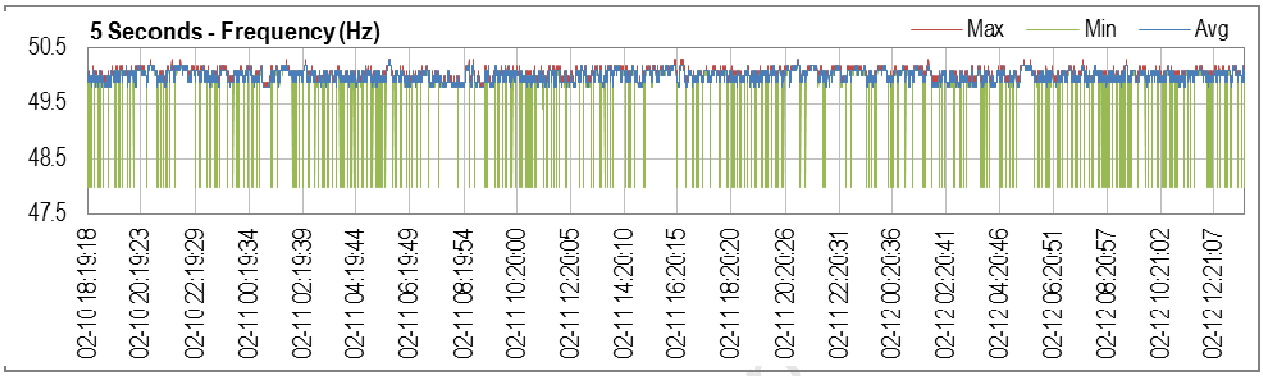


Figure 7.10 – 5-Second Frequency Profile

The full current measurement profile for the residence is shown in Figure 7.11. The maximum measurement value clearly identifies the startup of motor-based loads (in this case a refrigerator) and also assists with distinguishing resistive loads such as the geyser as there is no maximum current spike when these loads are switched on.

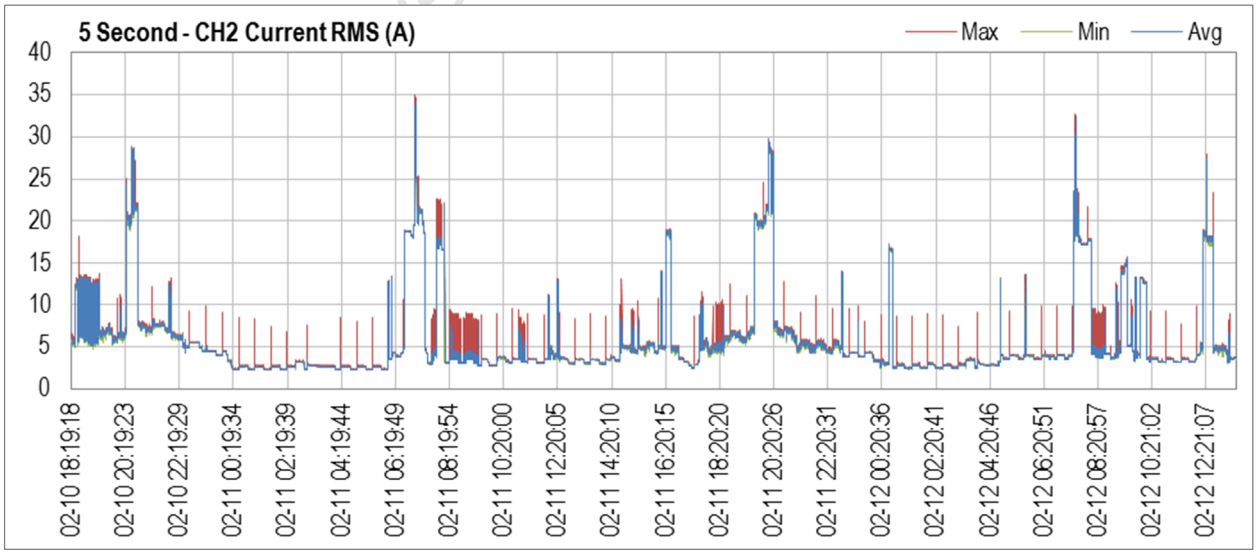


Figure 7.11 – 5-Second Current Profile

The active, reactive and apparent power measurement profiles are shown in Figure 7.12 on the following page. Apparent power is useful for determining the power rating of an appliance and to calculate the power factor profile. Using the reactive power the load can be classified as inductive, resistive or capacitive which is very useful to differentiate appliances in non-intrusive load monitoring applications where a residential survey has not been conducted. An appliance such as the fridge is easily identified from the apparent or active power profiles because of the high startup current spike of its compressor motor which can be seen in the maximum measurement.

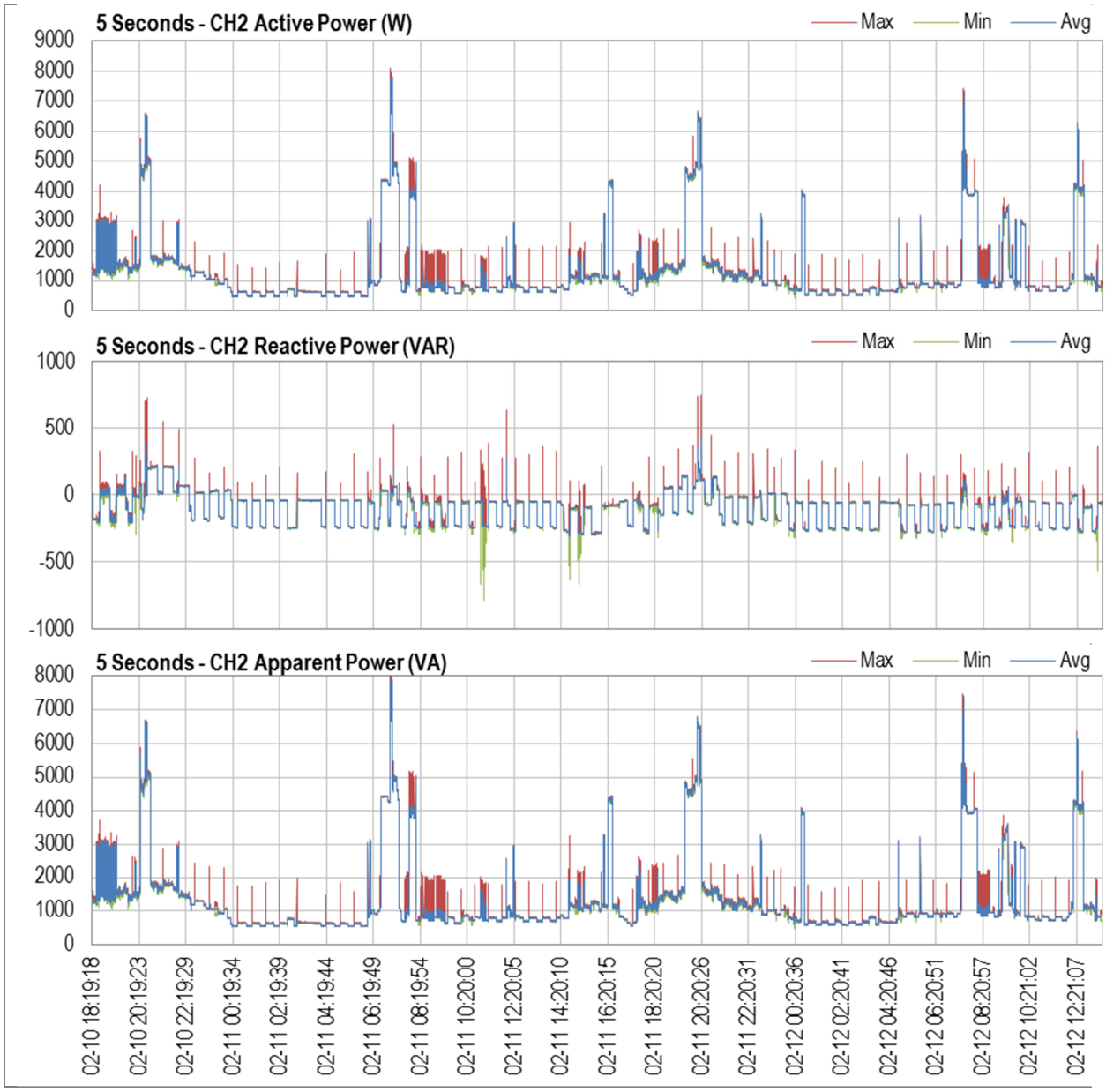


Figure 7.12 – 5-Second Active, Reactive and Apparent Power Profiles

The Power Profiler was installed indoors in summer and was not subjected to widely varying temperatures as shown in Figure 7.13. What is of interest is the gradual increase in temperature at the start of the testing period. Prior to the test the Power Profiler was not powered and was therefore settled at room temperature

of approximately 30°C. The rise in temperature is attributed to self-heating of the Power Profiler circuitry during operation.

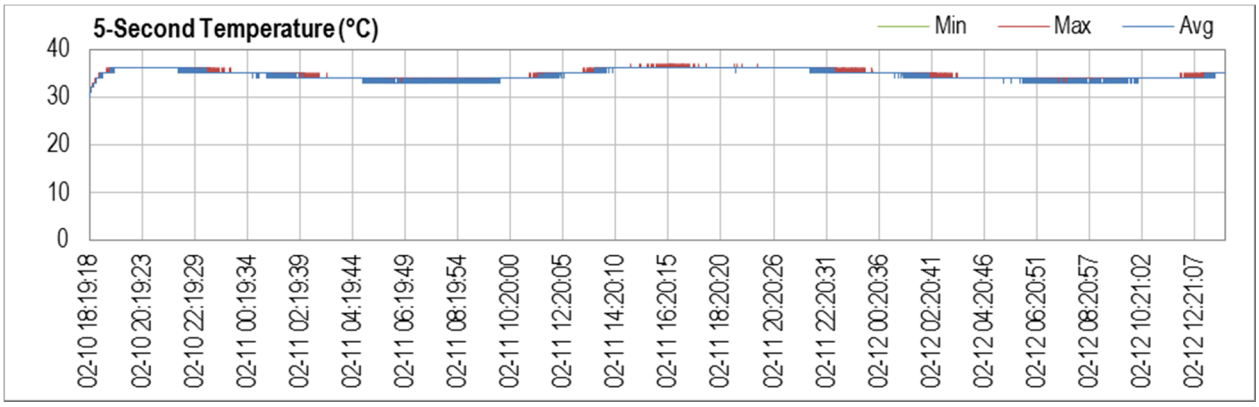


Figure 7.13 – 5-Second Temperature Profile

7.3.2. 10-Minute Measurement Profile

The 10-minute voltage and current measurement profile is shown in Figure 7.14 The maximum and minimum statistic for voltage is still useful as it shows the voltage dips at (approximately) 6am on the 11th and at 5am on the 12th. This data would still be beneficial in power quality analysis applications.

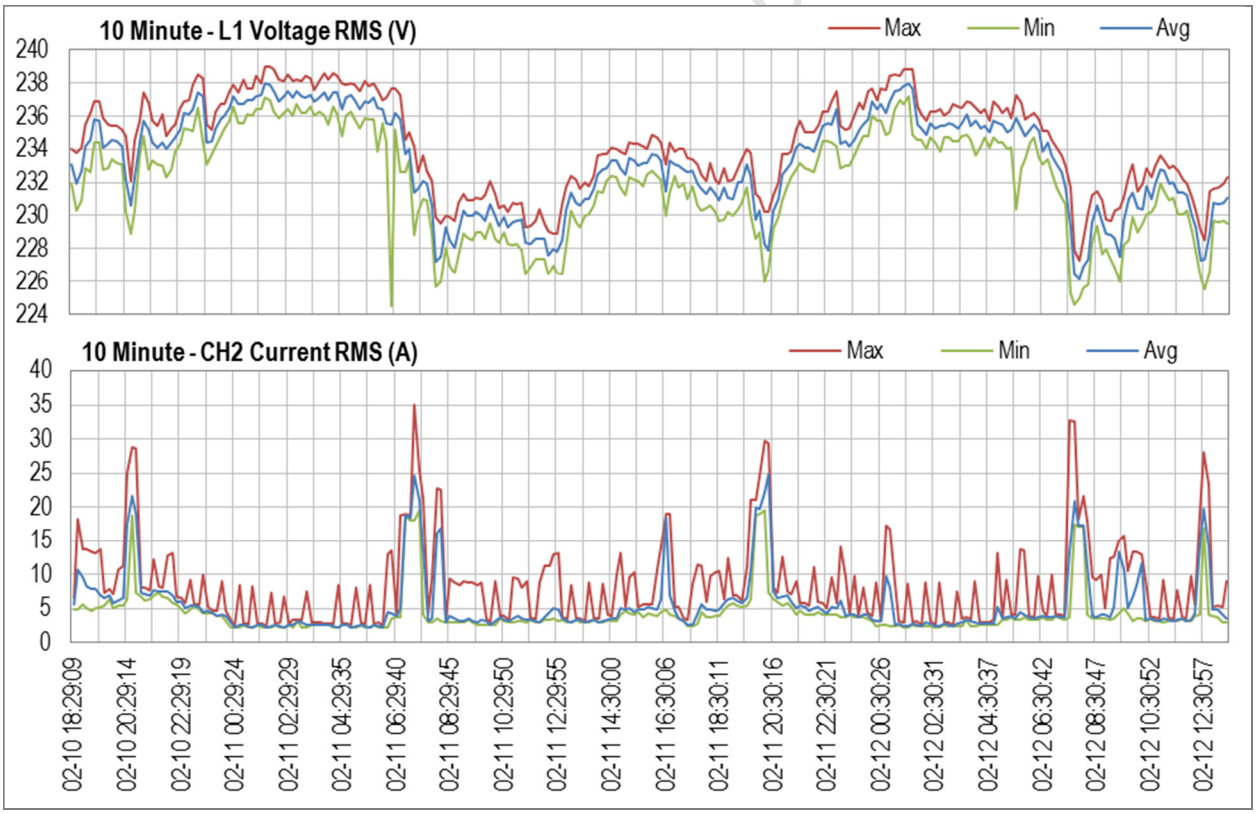


Figure 7.14 – 10-Minute Supply Voltage and Current Profile

The current profile plot still shows the basic appliance usage profile but the duration of appliance run is starting to become less evident. Large current consumers like the geyser are still determinable from the plot and the maximum measurement value allows appliances like the refrigerator to still be identified. The more

complex loads which were evident in the 5-second plot are now becoming less identifiable, but the statistical data at 6pm on the 10th still shows an interesting complex load. The ratio of the maximum, average and minimum can still be interpreted as a short-run-duration motor-based appliance because of the variance between the three measurement levels.

The active, reactive and apparent power profiles are shown in Figure 7.15. As with the current profile, large power consumers such as the geyser can still be determined from the power profiles, but the longer measurement interval makes differentiating complex loads difficult.

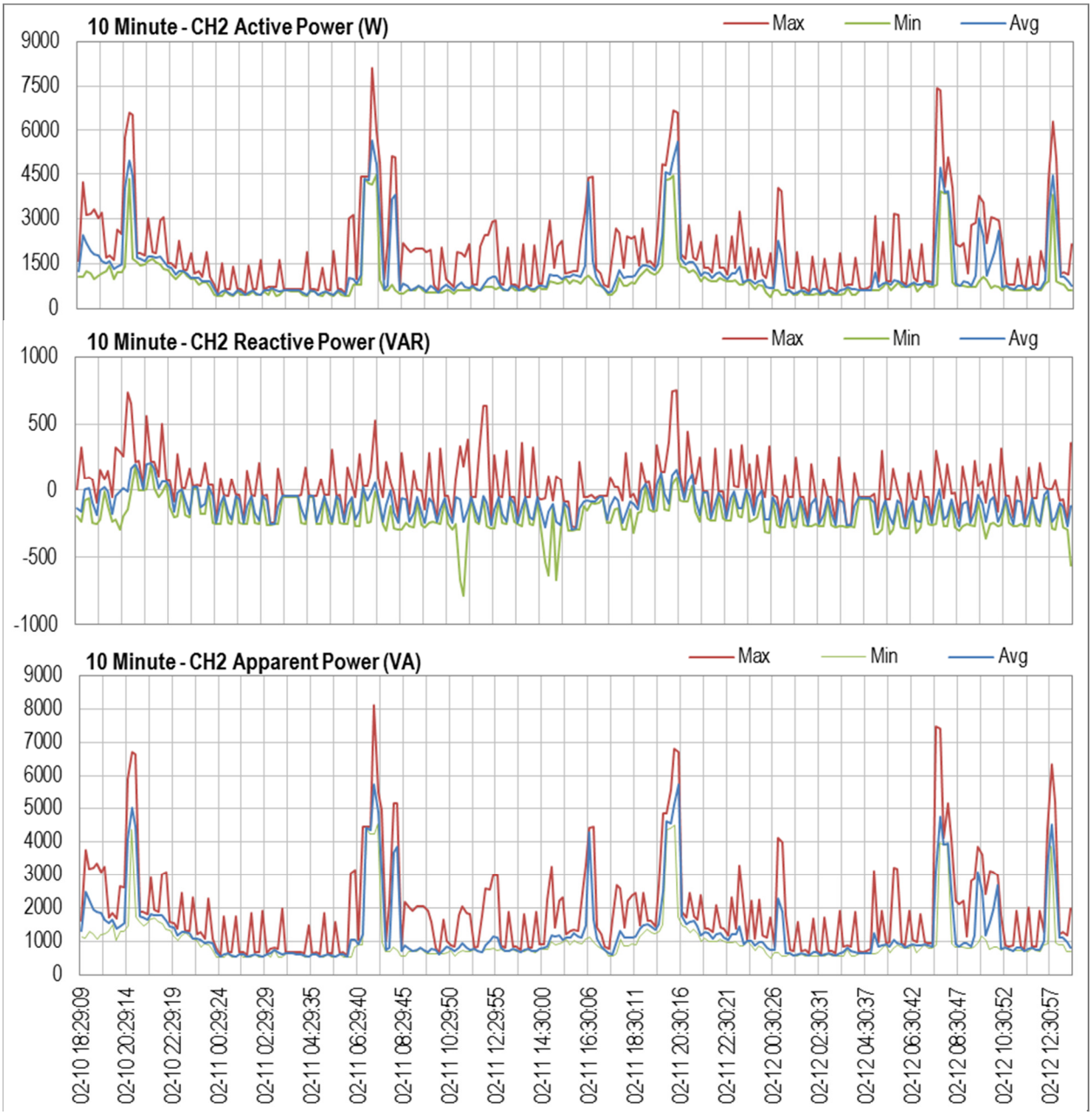


Figure 7.15 – 10-Minute Active, Reactive and Apparent Power Profiles

The frequency and temperature profile graphs are not presented here for discussion but are available in Appendix E.

7.3.3. 2-Hour Measurement Profile

The 2-hour measurement data is useful for long-term load trending but due to its low resolution is not practical for load identification purposes. The maximum and minimum voltage measurement values could be used in power quality monitoring applications to identify periods that contain out-of-range conditions for which a higher-resolution 5-sec profile could be uploaded. This is an advantage of the Power Profiler system: the second-, minute- and hour-based profiles are independently measured and retrieved. The bandwidth-intensive high-resolution second-based measurement data only needs to be uploaded for a relevant period which saves on the GSM operating costs.

In the 2-hour supply voltage profile shown in Figure 7.16 the two dips detected on the 5-second and 10-minute profiles are still evident from by their large deviation from the average value. The operator could upload the higher-resolution measurements for this period for further analysis.

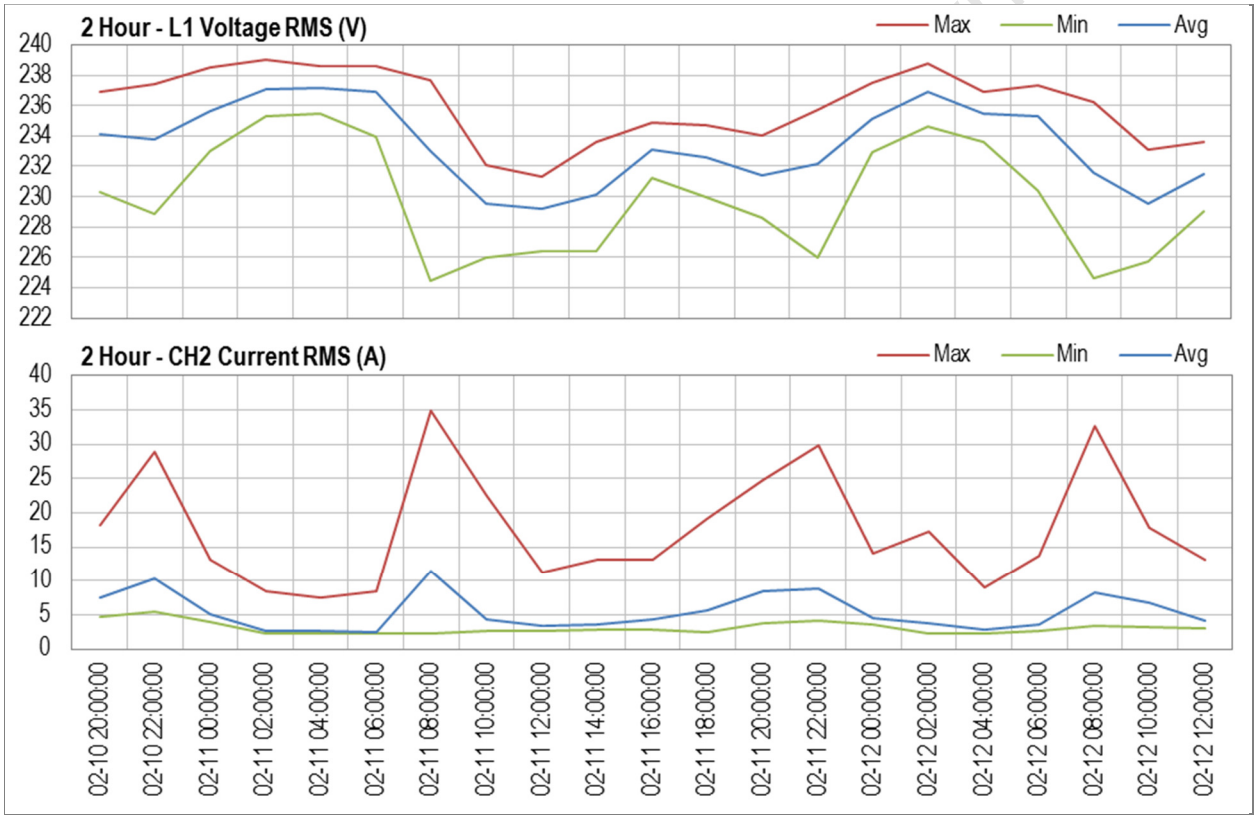


Figure 7.16 – 2-Hour Supply Voltage and Current Profile

The current profile offers little information due to its low resolution but the power profiles are far more useful to determine and trend average household power consumption as shown in Figure 7.17. The reactive power usage of the residence is also of interest as this is power that is lost by the Utility and must be taken into account when planning transmission and distribution networks.

By dividing the measured values by 2, the hourly average Wh, VARh and Vah measurements can be determined and profiled over time. For trending a measurement rate of 1-hour could be selected to simplify this process.

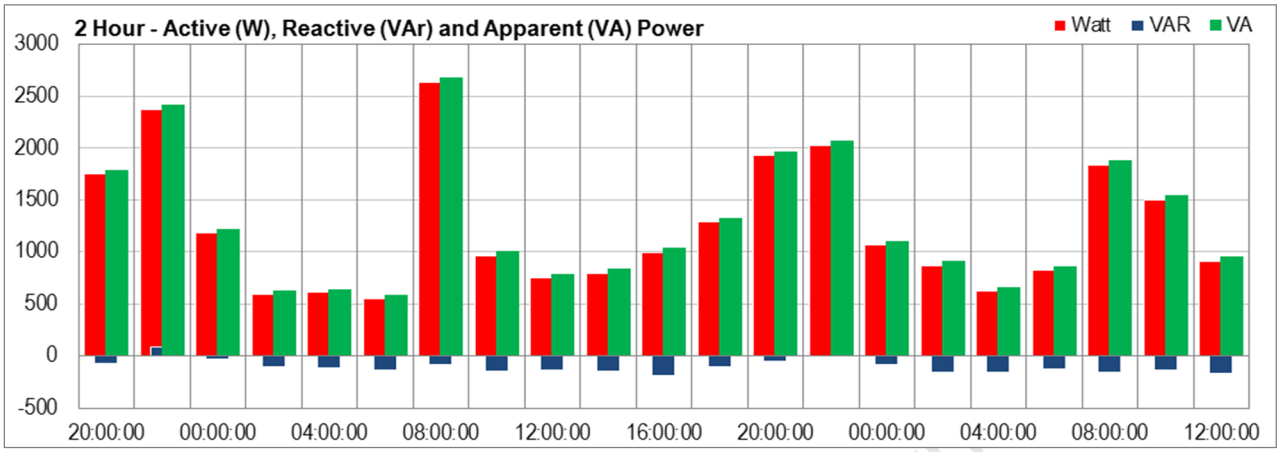


Figure 7.17 – 2-Hour Power Profile

7.3.4. Field Testing Summary

For load identification applications the 5-second measurement power profiles are very useful. As the measurement interval increases it becomes more difficult to differentiate loads, but the maximum measurement statistic still proves useful to identify motor-based appliances even at longer measurement intervals. The maximum and minimum voltage statistic is useful to determine swells and dips that are still evident even at long measurement intervals. The operator could then choose to download a higher resolution profile for this time period, reducing the operating cost over a GSM network. Hour-based measurement rates can be used for long-term trending of residential power demand.

7.4. Load Identification

From the field test measurement data several loads can be identified from the 5-second power profiles. A household survey of basic appliances was performed to confirm load identification as listed in Table 7.2.

Table 7.2 – Large Power Consumption Appliances

Appliance	Power	Load Type	Residence Circuit
Clothes Iron	1 200..1 440W	Resistive	Plugs
Fridge	150..250W	Inductive	Plugs
Geyser	3 000W	Resistive	Geyser
Kettle	2 200..2 400W	Resistive	Plugs
Oven / Microwave	2 100W Grill 1 900W Convection 1 650W Microwave	Complex	Oven
Washing Machine	150W	Complex (Inductive)	Plugs
Clothes Dryer	1 600..2 400W	Complex (Resistive)	Plugs

To identify the loads from the current and power profiles the following assumptions are used:

1. Resistive loads such as the clothes iron, geyser, kettle and oven will not consume reactive power
2. The fridge, washing machine and clothes dryer contain motors which will indicate a startup current and then a positive reactive power usage
3. Loads containing switch-mode power supplies (computers, television etc) will consume negative reactive power

As the Power Profiler was also connected to measure the geyser, lights, oven and plugs circuits these profiles can be used to confirm when certain appliances were operating. If you consider the combined active power profile of the verification and field testing period shown in Figure 7.18 the startup current for the inductive loads is clearly visible and the large step load is likely to be the geyser.

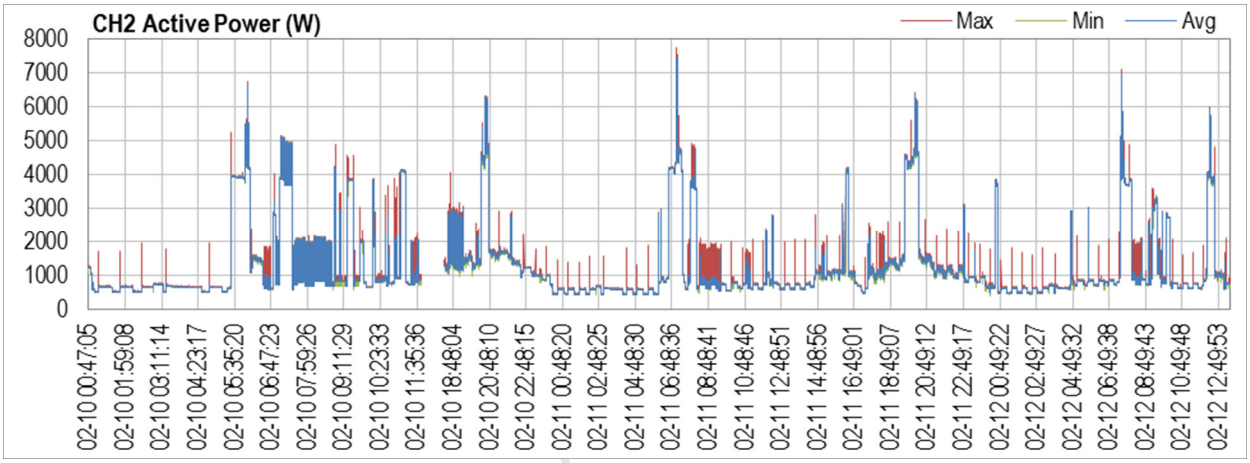


Figure 7.18 – Combined Verification and Field Test Active Power Measurements

From the active, reactive and apparent power profiles over the measurement period several loads were identified as discussed below.

Refrigerator

The refrigerator is a frequently cycling load which operates for approximately 20 minutes with a power profile as shown in Figure 7.19. A startup current of approximately 1 000W is followed by an operating power level of 200W and is confirmed by the load survey. The inductive load results in a positive increase in reactive power with the residence approaching unity power factor.

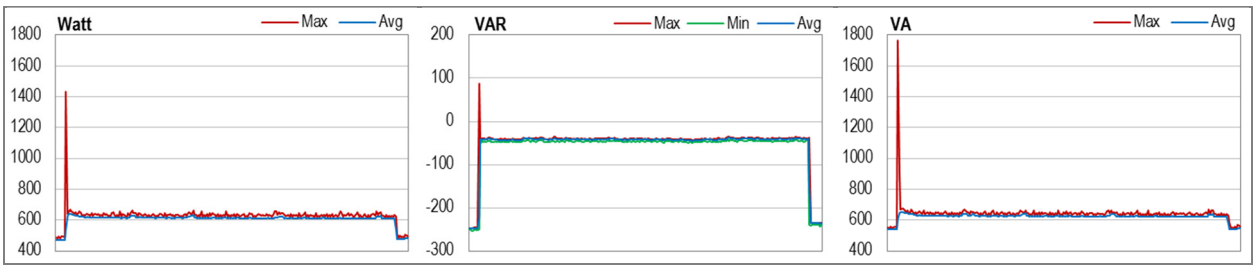


Figure 7.19 – Fridge Power Profile (02-11 01:07 to 01:29)

Geyser

The geyser is a large power consumer and a purely resistive load which should therefore result in no reactive power changes during use. The geyser is identified from the power profile as a high-power step load and is shown in Figure 7.20.

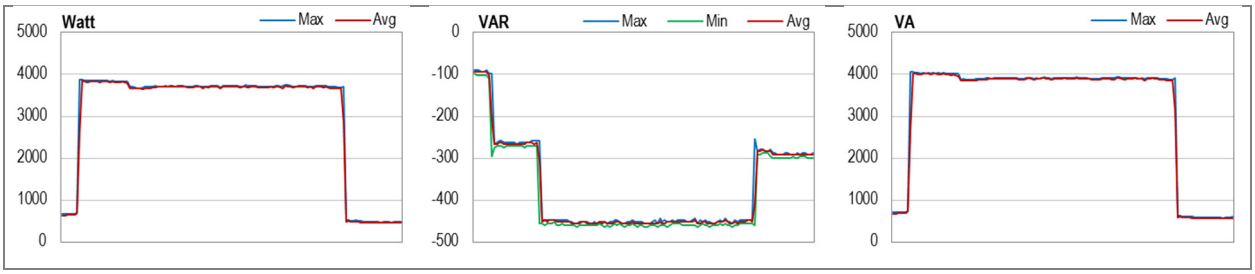


Figure 7.20 – Geyser Power Profile (02-12 00:35 to 00:46)

They geyser heated for approximately 8 minutes to just maintain the heat level. There is no step reactive power change coinciding with the geyser heating, confirming the purely resistive nature of this load.

Kettle

Like the geyser the kettle is also a large resistive power consumer but typically operates for short periods of time, making it a bit more difficult to identify from the power profile. The profile is shown in Figure 7.21 where it operated for approximately 2.5 minutes with a power consumption of approx. 2 200W. The purely resistive load is confirmed by a lack of coinciding step load in the reactive power profile.

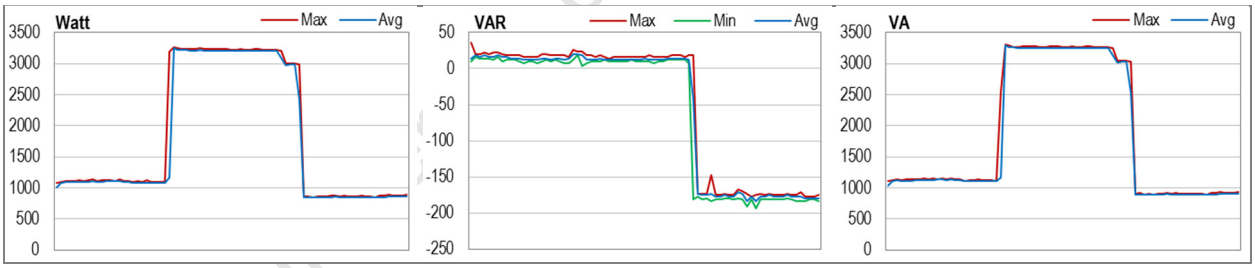


Figure 7.21 – Kettle Power Profile (02-11 22:48 to 22:55)

Clothes Iron

The identified profile for the clothes iron is shown in Figure 7.22. The cycling profile is due to the thermostat and the purely resistive load is evident from no coinciding stepped reactive power.

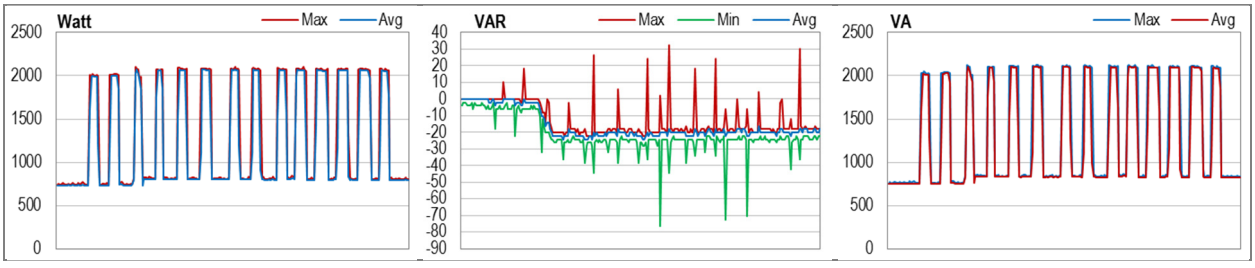


Figure 7.22 – Clothes Iron Power Profile (02-10 07:34 to 07:48)

Washing Machine & Clothes Dryer

The residence uses a LG combination washing machine and dryer. The clothes dryer was used for approximately one hour during the measurement period as shown in Figure 7.23. The step in the reactive power is likely due to the refrigerator turning on during the same period, and not related to the clothes dryer. The appliance uses a direct-drive motor system and therefore startup current is limited and is not as clearly visible as it is with the refrigerator.

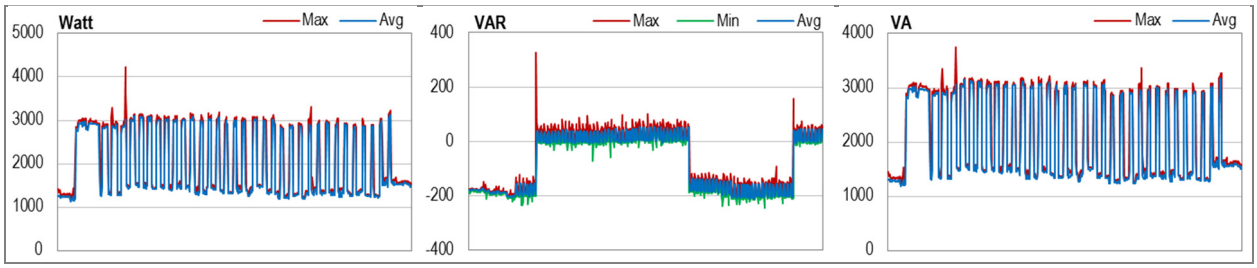


Figure 7.23 – LG Clothes Dryer Power Profile (02-11 18:26 to 19:25)

The initially long power consumption is likely due to the system heating up to the selected temperature and the subsequent cycling is due to the thermostat maintaining this temperature. The appliance rotates at a slow speed in one direction only during drying with the first small startup current spike possibly being the initial rotation of the dryer cycle once the machine is up to temperature.

Figure 7.24 shows a washing cycle of the appliance which lasted for approximately one hour. The cycling fridge is still evident in the reactive power profile, but the startup current of the washing machine motor is now also visible as the washing machine alternates rotation direction during the washing cycle. The motor is a direct drive system that uses a variable speed drive to control drum speed. The variable speed drive appears to ramp up the motor speed to wet the contents of the drum at the start of the wash cycle. The final high-speed spin is also shown as the last 'spike' on the profile.

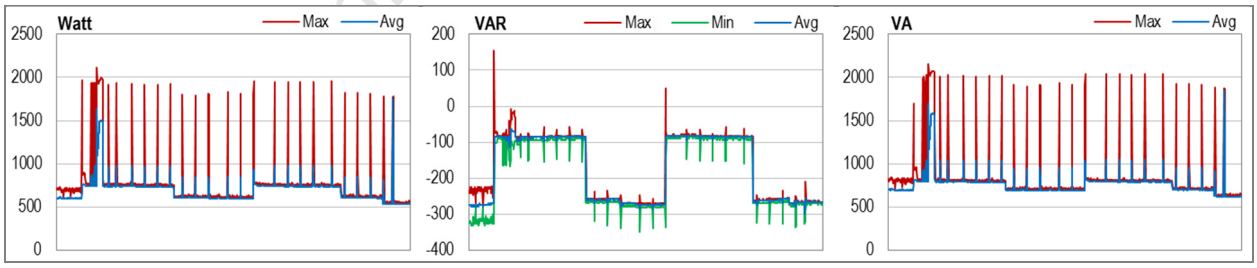


Figure 7.24 – LG Washing Machine Power Profile (02-11 08:15 to 09:17)

Microwave

The residence uses a LG combination microwave, oven and grill appliance. The oven function was not used during the measurement period but Figure 7.25 and Figure 7.26 respectively show the power profiles for the microwave and grill functions.

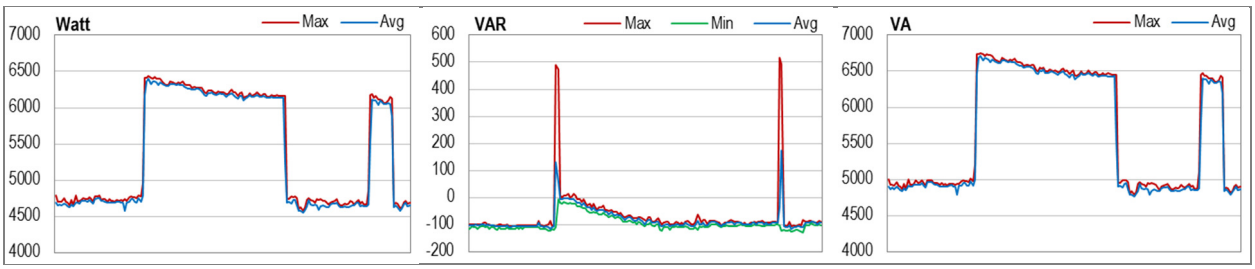


Figure 7.25 – LG 1 650W Microwave Power Profile (02-11 20:32 to 20:41)

The reactive power profile shows a large inductive spike probably due to the high-voltage step-up transformer used to drive the magnetron. As the power level to the magnetron decreases so does the reactive power consumption.

The grilling function power profiles are different to what one would expect from a conventional oven griller because the LG SolarDOM appliance uses high-power halogen lamps in conjunction with a conventional heater element for grilling. It is assumed that a control loop to establish a consistent grilling temperature results in the varying power profiles during use.

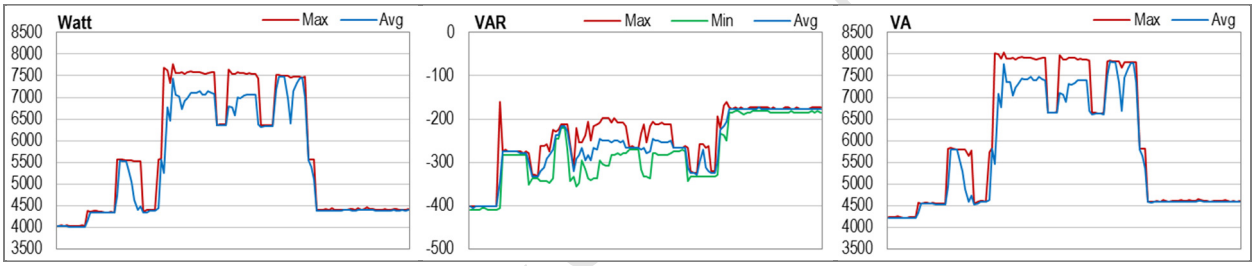


Figure 7.26 – LG 2 100W Grill Power Profile (02-11 07:00 to 07:10)

Idle Time

To determine the minimum load of the residence a period of minimum use is profiled and shown in Figure 7.27. This residence has several electronic products such as computers, chargers and appliances on standby resulting in a base load of approximately 440W with a reactive power of -265VAR. The capacitive reactive power is typical of the switch-mode power supplies typically used by computers, appliances and chargers.

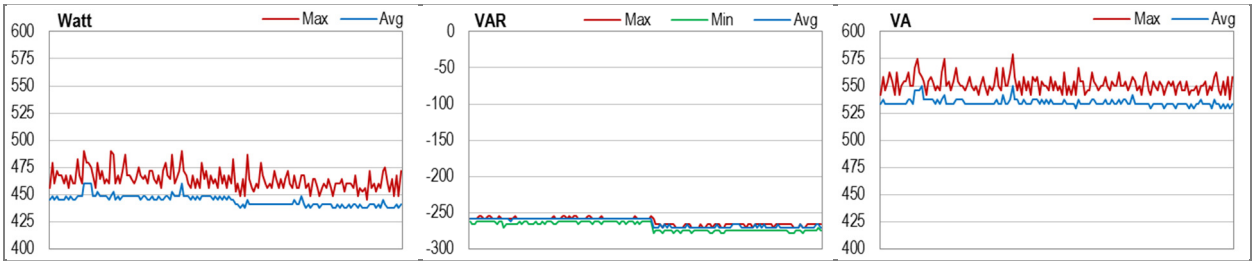


Figure 7.27 – Idle Time Power Profile (02-11 04:40 to 04:53)

7.5. Discussion

The measurement verification process compared the capture of 3-second measurements of the Power Profiler to 5-second measurements of the Qualistar 8334B power meter. Voltage and current RMS measurements were similar but phase-errors in the current transformers used by the Power Profiler resulted in less-accurate active and reactive power measurements. As discussed in the Chapter 6 this phase error can be corrected through modification of the anti-aliasing filters on the voltage and current channels, or through the use of higher quality CTs.

To perform an adequate field test of the Power Profiler it was connected to a residential load and left to gather data using a 5-second, 10-minute and 2-hour measurement storage rate as recommended in the IEC 61000-4-30 specification for power quality measurement [IEC, 2003]. The Power Profiler gathered data for 43 hours which was then uploaded using the Configuration and Control application via Bluetooth and exported in CSV format for further analysis and graphing in Microsoft Excel.

Typical power profiling devices store only the average level during a measurement period, whereas the Power Profiler stores the maximum, minimum and average values for all channels. Of particular interest during the software development stage was if the ability to capture more statistical data could make it easier to characterise load types and assist with non-intrusive load identification. This was confirmed with the maximum current statistic being very valuable in differentiating motor-based appliances such as the refrigerator. For voltage measurements the maximum and minimum measurements proved valuable in identifying supply line surges and dips as these values represent the extremes encountered during the measurement interval. For example for a 2-hour measurement interval the maximum and minimum samples represent the highest and lowest measurement values over 36 000 samples.

The 5-second current profile benefits significantly from the maximum measurement in that it clearly shows the start-up current of motor-based appliances. This benefit reduces with an increased measurement interval and while still useful at a 10-minute interval, it becomes impractical at higher intervals for load-characterisation. The maximum statistic could be very beneficial to automated load characterisation using the 5-second profile data.

Power measurement profiles provided an easy method of characterising loads based on their power usage. In the field test it was easy to identify the refrigerator of 150W and the geyser of 3 000W from the power plot. The measurement of active, reactive and apparent power and the ability to trend a consumer's hourly power usage can assist with supply demand modelling and network planning.

Chapter 8

Conclusions and Future Work

8.1. Conclusions

The purpose of this dissertation was to evaluate if a suitable cost-effective replacement for the existing logger used by the University of Cape Town's Load Research Group for residential load profiling could be developed to take advantage of newer technologies and incorporate reliable, remote data retrieval and processing. The design, implementation and testing of a new instrument to achieve this purpose has been successfully demonstrated.

During the literature study the key components of a measurement system intended for non-intrusive load monitoring were identified and researched. The requirements for power quality analysis were also discussed to determine if such functionality could also be supported by the solution. A concept product called the Power Profiler was proposed to test the hypothesis. The Power Profiler hardware was developed over three prototypes and incorporated energy measurement system-on-chips for high accuracy voltage, current and power measurements. Control and measurement processing was performed using a low-cost Microchip dsPIC processor and DataFlash memory was used for measurement storage. GSM GPRS communication provided remote access to the Power Profiler when field deployed and Bluetooth communication provided an electrically-isolated local interface to be used on-site without having to disconnect the supply to the monitored residences. All communications incorporate cyclic-redundancy checking algorithms to ensure reliable access for configuration and the retrieval of measurement data.

Software was developed in embedded C for the Power Profiler's dsPIC and PIC18F processors to implement the required control, measurement and communications functionality. A PC control and configuration application was developed in C# that makes use of an interface library to access and control the Power Profiler hardware. Extensive use was made of object-oriented programming techniques so as to easily support future code re-use in custom-developed applications.

The Power Profiler was successfully calibrated using a Qualistar 8334B power quality analyser as a reference meter. It was found that the Power Profiler exhibited an inaccuracy of 0.4% for RMS voltage (over a 20:1 range), typically 0% for RMS current, 1.1% for active power, 0.3% for reactive power and 0.1% for apparent power versus the reference meter. Phase error of the Taehwatrans TS10L CTs became apparent in the active and reactive power measurements and the use of higher-quality CTs was recommended for power quality analysis applications.

The cost-effectiveness of the Power Profiler was confirmed by comparing it against an equivalent commercially available solution consisting of four three-phase power meters connected to a controller and GSM modem. The estimated Power Profiler manufacturing cost including GSM and Bluetooth communications was found to be significantly less than the cost of the four power meters. Considering the high accuracy achieved during calibration versus the Qualistar power quality analyzer (a far more expensive instrument) and its possible use in power quality analysis applications it does offer a highly competitive and cost-effective solution for research and (in particular) for load monitoring applications.

The functional operation of the Power Profiler was validated by connecting both it and the Qualistar 8334B to a residential household in order to gather 4 hours of measurement data for comparison. All channels were confirmed to be working correctly by the amplitude and profile of the measurements being very similar between both instruments. Small discrepancies in active and reactive power were observed and attributed to the lower accuracy and higher phase shift error of the Taehwatrans CTs being used.

The Power Profiler was then used to measure the residence's power consumption for a period of 43 hours after which the gathered measurement data was analysed to determine if loads could be discriminated form the measured power profiles. The inclusion of the statistical maximum and minimum measurement values identified two voltage dips during the measurement period that would have otherwise gone unreported by a typical power meter. The maximum current statistic also effectively showed the start-up current of motor-based appliances which greatly assisted in load discrimination. From the measurement data common household appliances such as the refrigerator, geyser, kettle, clothes iron and washing machine were identified and their power profiles discussed.

It was shown that while the second- and minute-based profiles may be used for load identification, the use of longer measurement intervals made it more difficult to differentiate loads. Hour-based profile data is considered only practical for long-term power trending and for the detection of out-of-tolerance conditions such as voltage dips or swells. The ability of the Power Profiler to independently profile second-, minute-, and hour-based measurements and for the operator to select which data to upload allowed for better system management and reduced operating cost over a GSM network.

The development and testing has shown that the original hypothesis that "can a cost-effective replacement for the existing logger be developed that will incorporate modern communications techniques in a flexible manner and aid in further research into power measurement and power analysis applications" is valid.

8.2. Future Work

The Power Profiler hardware has been confirmed to correctly profile the data measured by the Analog Devices ADE7758 energy measurement ICs. Attention can be turned to the development of further software functionality to implement basic power quality analysis such as dip and swell profiling which were termed

'events'. The SRAM memory is provided for the purpose of multiple measurements to be buffered and when an event occurs the pre- and post-event measurement data can be stored in flash memory.

The battery backup functionality needs to be improved to ensure a seamless switchover upon main supply failure to ensure that long-term dips (or outages) may be profiled. The use of a comparator circuit is suggested and should be tested to confirm correct operation. Another option would be to increase the output supply voltage of the switch-mode-power converter to higher than that of the battery supply, but this will result in higher heat dissipation in the final voltage regulator.

The direct sampling of the voltage and current channels by the dsPIC processor opens up many possibilities for power quality analysis applications to be researched. The dsPIC can support digital signal processing of the measured signals and so measurement of power quality attributes such as flicker and harmonic distortion can be experimented with. Higher quality CTs should be also tested for use in such power quality analysis applications.

An automated data retrieval function using a database for measurement storage may also be implemented for server control of remotely deployed Power Profilers. Such software is made easier to implement by the object-based design approach of the C# Power Profiler interface components. It is also recommended that the USB port have some form of electrical isolation for operator protection and to allow for measurement of 3-phase delta configurations.

Glossary

3G	3rd Generation packet-based GSM communications
ADC	Analog-to-Digital Converter
AMI	Advanced Metering Infrastructure
AMR	Automated Meter Reading
BOM	Bill of Materials
CS	Chip Select
CT	Current Transformer
DSC	Digital Signal Controller
dsPIC	The Microchip dsPIC30F6013 digital signal controller used as the main processor
EDGE	Packet-based GSM communications
EMI	Electro-Magnetic Interference
GPRS	General Packet Radio Service (packet-based GSM communications)
GSM	Global System for Mobile Communications (cellular-based communications)
I^2C	Inter-Integrated Circuit interface that uses two wires for communication
IC	Integrated Circuit
I/O	Input/Output
LCD	Liquid Crystal Display
LSB	Least Significant Bit
MSB	Most Significant Bit
NLM	Non-intrusive Load Monitoring
PCB	Printed Circuit Board
PIC18F	The Microchip PIC18F4550 microcontroller
PTN	Public Telephone Network
RS232	Serial communications interface complying with the EIA-232 standard
RS485	Multi-drop serial communications complying with the EIA-485 standard
SMPS	Switch-Mode Power Supply
SoC	System-on-Chip
SPI	Serial Peripheral Interface
TCP	Transmission Control Protocol used in packet-based data transfers
UART	Universal Asynchronous Receiver Transmitter
USB	Universal Serial Bus
WiFi	Wireless packet-based communications protocol complying with the 802.11 standard

References

A Analog Devices. 2003. *Evaluation Board Documentation ADE7758 Energy metering IC.* s.l.: s.n.

Analog Devices. 2004. *AD620 Data Sheet.* s.l.: s.n.

Analog Devices. 2011. *ADE7758 Poly Phase Multifunction Energy Metering IC with Per Phase Information.* s.l:s.n.

Analog Devices. 2012. *Energy Measurement*. [Online]. Available: http://www.analog.com/ [28 Jan 2012]

B Bluetooth SIG. 2011. *The Official Bluetooth Technology Web Site.* [Online]. Available: http://www.bluetooth.com/Pages/Bluetooth-Home.aspx [23 Feb 2012]

C Chauvin Arnoux. 2003. *Three Phase Power Quality Analyser C.A 8334 User's Manual.* Paris: Chauvin Arnoux.

ConnectOne. 2012. *ConnectOne.* [Online]. Available: http://www.connectone.com [27 Feb 2012]

Cooper Power Systems. (n.d.). *RF AMI.* [Online]. Available: http://www.cooperindustries.com/ [27 Feb 2012]

D Dahle, D. (n.d). *Dave's old Watthour Meter webpage*. [Online]. Available: http://watthourmeters.com/history.html [26 Feb 2012]

Digi International. 2012. *Digi X-Grid.* [Online], Available: http://www.digi.com/solutions/digi-x-grid [27 Feb 2012]

E Echelon. 2012. *Challenges to reliable PLC.* [Online]. Available: http://www.echelon.com/technology/power-line/pl-challenges.htm [27 Feb 2012]

EEMB Battery. n.d. *LP103450 Data Sheet.* s.l.:s.n.

EPCOS. 2009. *EMI suppresion capacitors (MKP).* s.l.:s.n.

Elster. 2012. *Elster.* [Online]. Available: www.elstersolutions.com [27 Feb 2012]

ETSI. 2012. *ETSI GS OSG 001 V1.1.1.* s.l.:European Telecommunications Standards Institute.

F Fairchild Semiconductor. 2009. *1N4001 - 1N4007 Data Sheet.* s.l.:s.n.

Fletcher, J. 1982. *An Arithmetic Checksum for Serial Transmissions.* IEEE Transactions on Communication, vol. COM-30, no. 1, pp. 247-252.

Fourier Systems. 2004. *DaqPro.* [Online]. Available: http://www.fouriersystems.com/ [22 Feb 2012]

G Gaunt, Mostert & Geldenhuys. 2007. *Proof-of-Concept Data Logger for Power Quality Monitoring.* Vienna, CIRED.

Google Inc. 2011. *Google Powermeter.* [Online]. Available: http://www.google.com/powermeter/about/ [22 Feb 2012]

Groenewald, H. 2008. *NRS049 – Advanced Metering Infrastructure (AMI) for Residential and Commercial Customers.* [Online]. Available: http://www.ameu.co.za/ejournal/ [23 Feb 2012]

GSM World. 2011. *GSM World – History.* [Online]. Available: http://gsmastage.com/about-us/history.htm [24 Feb 2012]

H Hart, G. 2006. *Nonintrusive Appliance Load Monitoring.* [Online]. Available: http://georgehart.com/research/nalm.html [22 Feb 2012]

Hart, G; Kern, E & Schweppe, F. 1989. *US Patent 4858141.* [Online]. Available: http://www.patft.uspto.gov/ [22 Feb 2012]

Hanzelka, Z. & Bien, A. 2005. *Voltage Disturbances Flicker Measurement.* s.l.: Copper Development Association.

I IEC. 2003. *CEI IEC 61000-4-30 Part 4-30: Testing and measurement techniques – Power quality measurement methods.* Geneva, Switzerland: International Electrotechnical Commission.

IEC. 2002. *IEC 62056-21 Direct local data exchange.* s.l.:International Electrotechnical Commission.

Itron. 2010. *ACE1000 282 Brochure,* s.l.:Itron.

Itron. 2012. *Itron.* [Online]. Available: http://www.itron.com/ [27 Feb 2012]

K KC Wirefree. 2007. *KC-11 Datasheet. KC-Wirefree*, 2007.

L Littelfuse. 2009. *CH Series Varistor Products Datasheet,* s.l.: s.n.

M Maxim. n.d. *DS1631/DS1631A/DS1731 Data Sheet,* s.l.:s.n.

Maxim. 1998. *MAX350 Data Sheet (Rev 1),* s.l.: s.n.

Maxim. 2012. *Energy-Measurement and Metering SoCs.* [Online]. Available: http://www.maxim-ic.com/products/energy/ [28 Jan 2012]

Melhorn, C. & McGranaghan, M. 1995. Interpretation and analysis of power quality measurements. *IEEE Transactions on Industry Applications,* 31(6), pp. 1363-1370.

Microchip Technology Inc. 2005. *dsPIC30F Family Reference Manual.* s.l.: Microchip Technology Inc.

Microchip Technology Inc. 2005. *Introduction to Utility Metering Tutorial.* s.l.: Microchip Technology Inc.

Microchip Technology Inc. 2007. *MCP1726 Data Sheet.* s.l.:s.n.

N National Planning Commission. 2010. *Development Indicators 2010.* s.l.: National Planning Commission, RSA.

National Planning Commission. 2011. *National Development Plan Vision 2030.* s.l.: National Planning Commission, RSA.

National Semiconductor. 1999. *LM2660/LM2661 Switched Capacitor Voltage Converter.* s.l.:s.n.

National Semiconductor. 2002. *The Practical Limits of RS-485.* s.l.:National Semiconductor.

National Semiconductor. 2011. *LM117/LM317A/LM317 Data Sheet.* s.l.:s.n.

O ON Semiconductor. 2005. *MC33275 Data Sheet.* s.l.:s.n.

ON Semiconductor. 2006. *MBRD1035CTL Schottky Barrier Rectifier Data Sheet.* s.l.:s.n.

P Plogg. 2010. *Plogg.* [Online]. Available: http://www.plogginternational.com/ [28 Mar 2012]

Power Integrations. 2005. *TOP242-250 TopSwitch-GX Family Data Sheet.* s.l.:s.n.

R Rivest, R. 1992. *RFC1321: The MD5 Message-Digest Algorithm.*[Online]. Available: http://tools.ietf.org/html/rfc1321 [28 Mar 2012].

RS Components. 2012. RS Components. Available: http://za.rs-online.com/web/ [28 Mar 2012]

S Shepard, D. & Yauch, D. n.d. *An Overview of Rogowski Coil Current Sensing Technology.* Ohio, LEM DynAmp Inc.

Siemens. 1996. *BCR22PN Data Sheet.* s.l:s.n

T Taehwatrans. 2012. *Split Core (Clip-On) Current Transformer.* [Online]. Available: http://www.taehwatrans.com [11 October 2012]

Taoglas. 2012. *FXP07.07.0100A Data Sheet.* [Online]. Available: http://www.taoglas.com/antennas/Cellular-GSM/Internal_Flexible_PCB_antennas_-_FXP_Series/ [26 March 2012]

Texas Instruments. 2012. *Smart Meter.* [Online]. Available: http://www.ti.com/solution/smart_e_meter_amr_ami [30 Jan 2012]

Telit. 2008. *GC864-QUAD Data Sheet.* Italy: Telit Communications S.p.A

Telit. 2009. *GC864 Hardware User Guide Rev. 9.* s.l.:s.n.

Telit. 2010. *AT Commands Reference Guide Rev. 7.* s.l.:s.n.

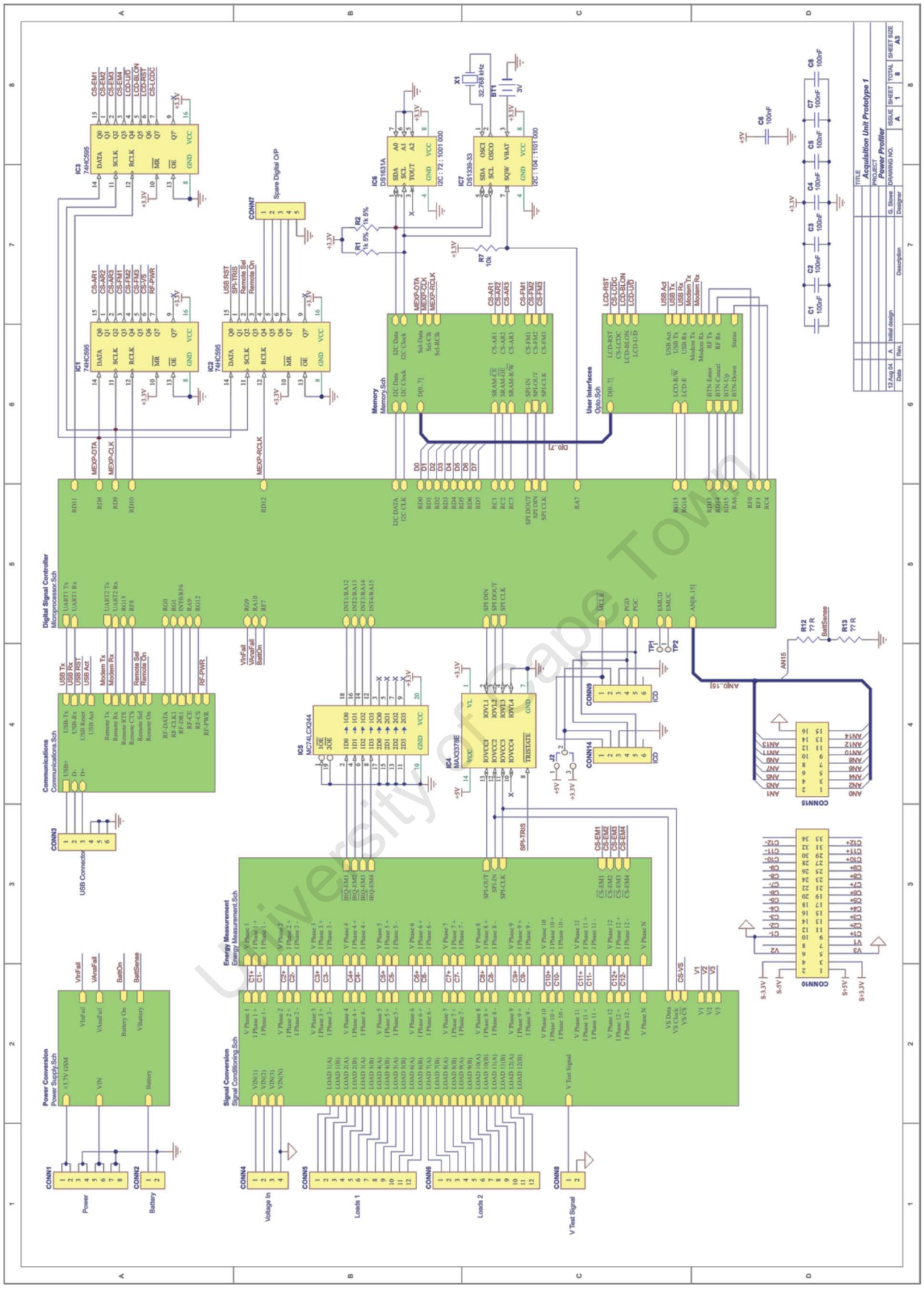
U u-Blox. 2010. *LISA-U1 series 3.75G HSPA Wireless Modules Data Sheet.* s.l.:u-Blox.

W Wan, N. 2003. Reactive Energy Measurement Made Simple. *Metering*, pp. 24-26.

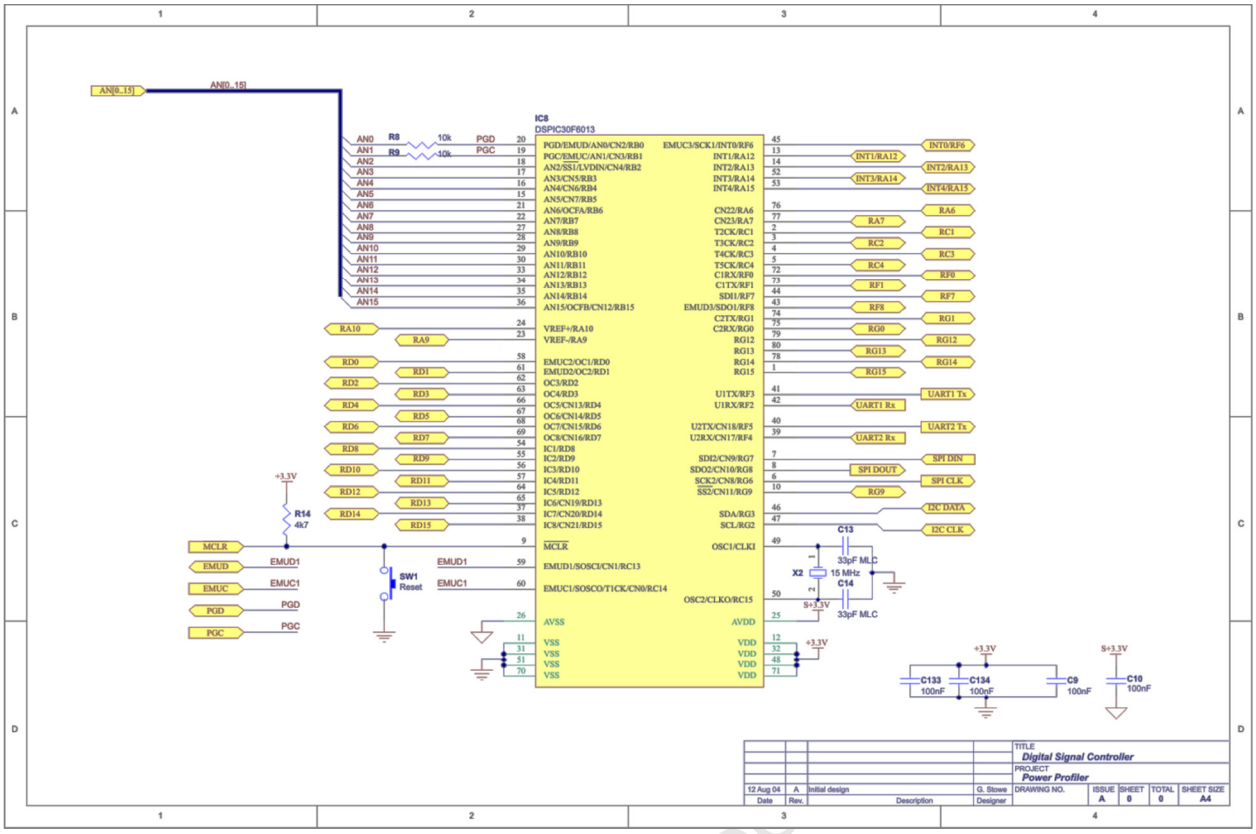
Appendix A

Concept Schematics and PCB Layouts

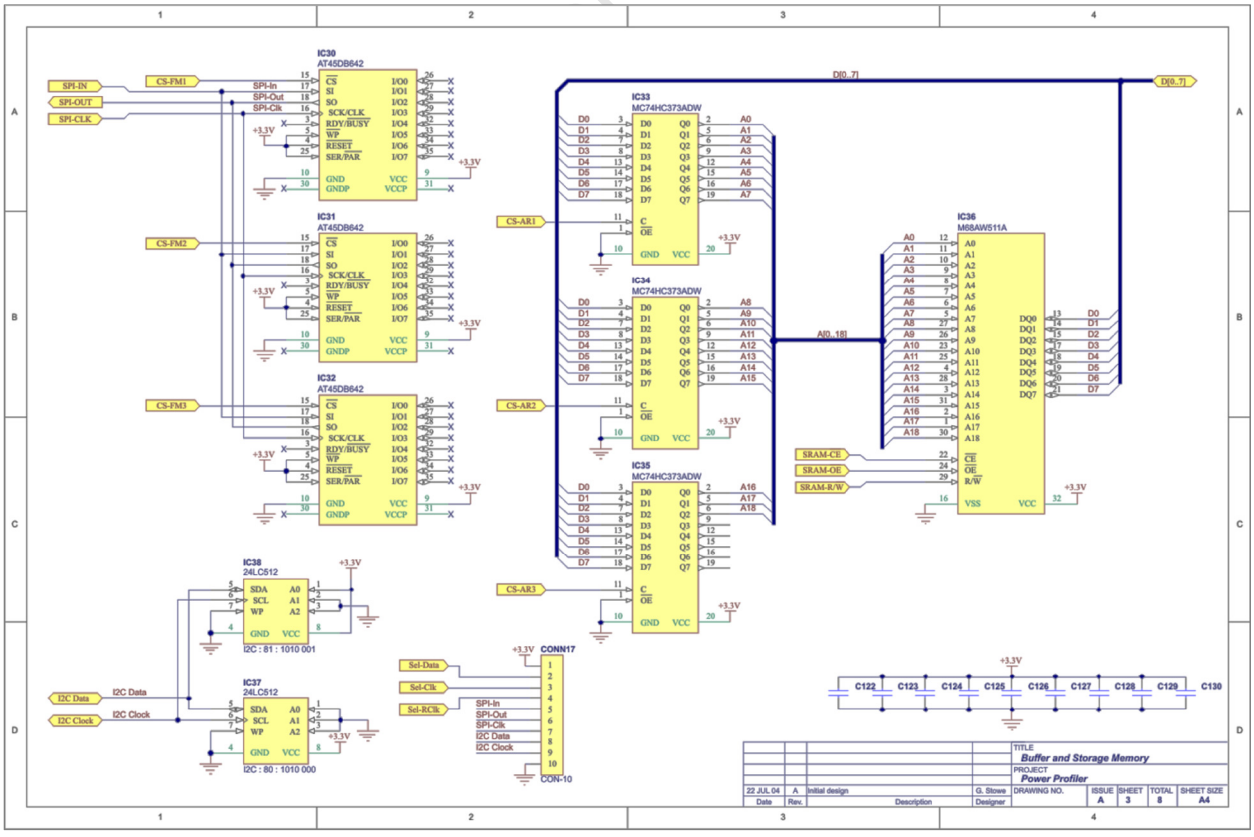
Prototype One



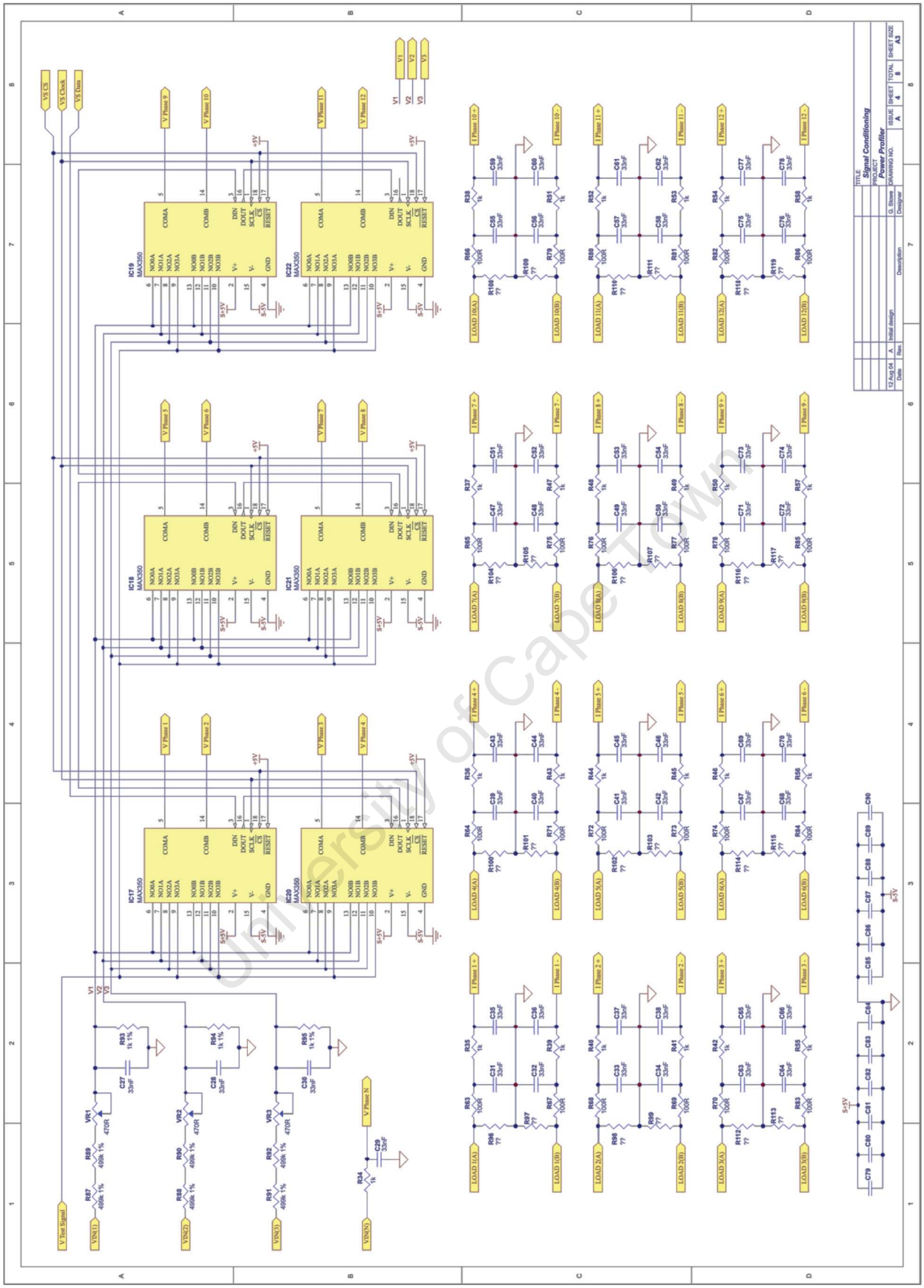
Prototype One: Schematic Overview



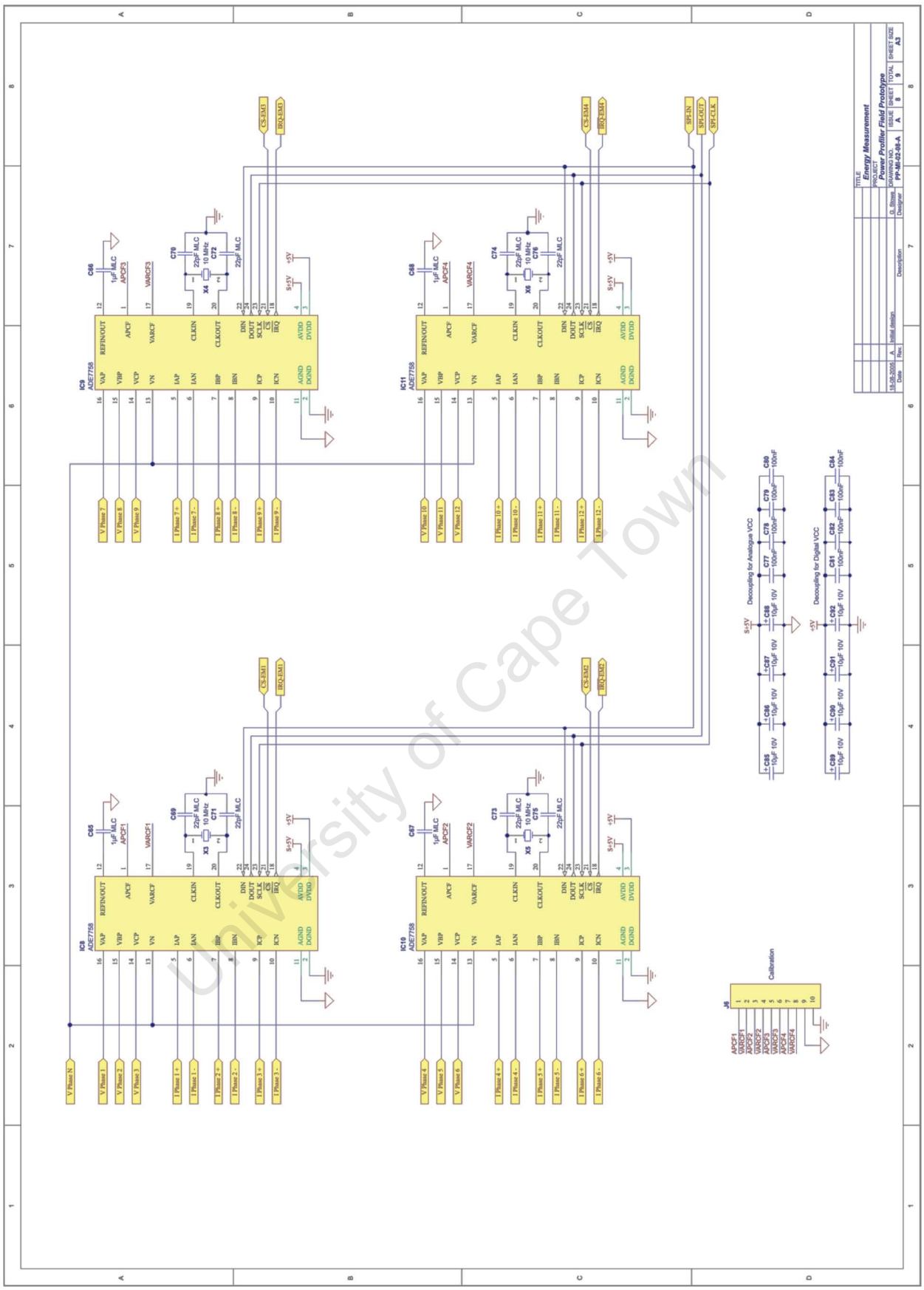
Prototype One: Digital Signal Controller



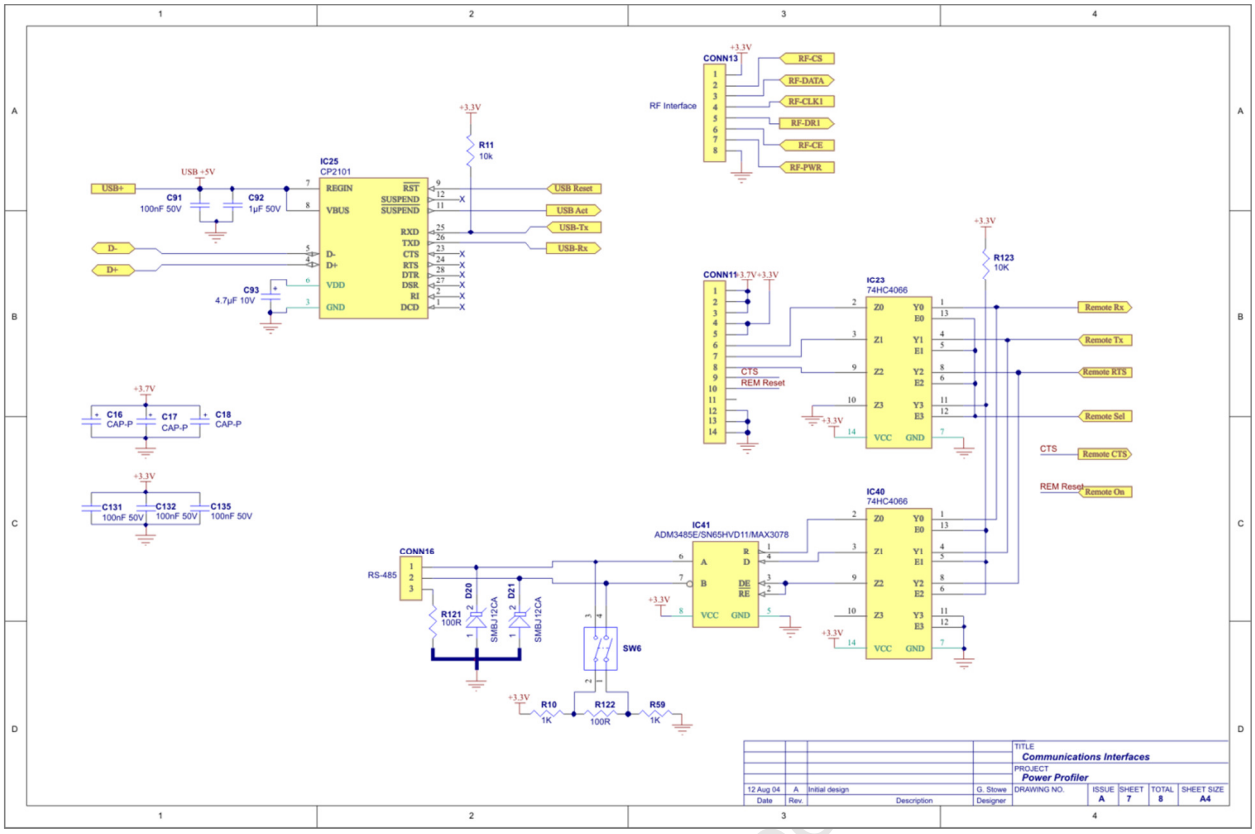
Prototype One: Buffer and Storage Memory



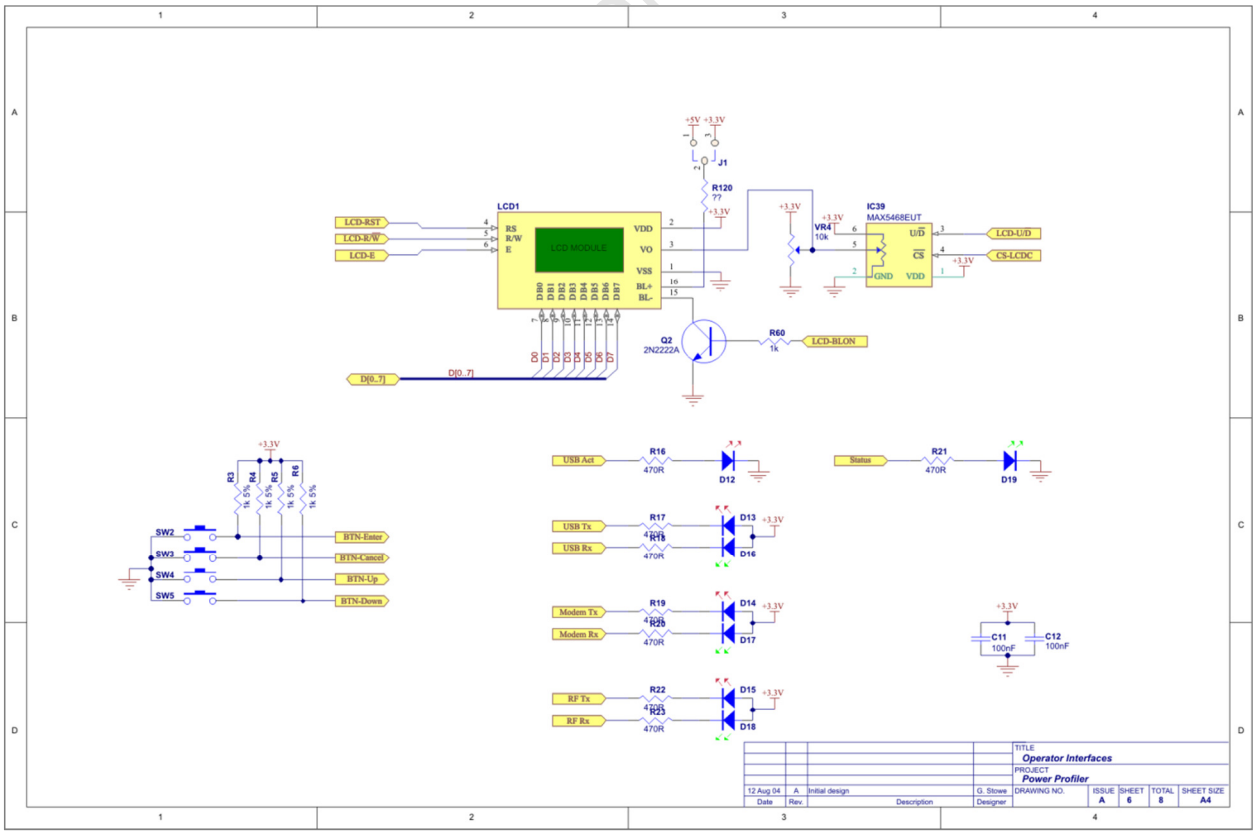
Prototype One: Signal Conditioning



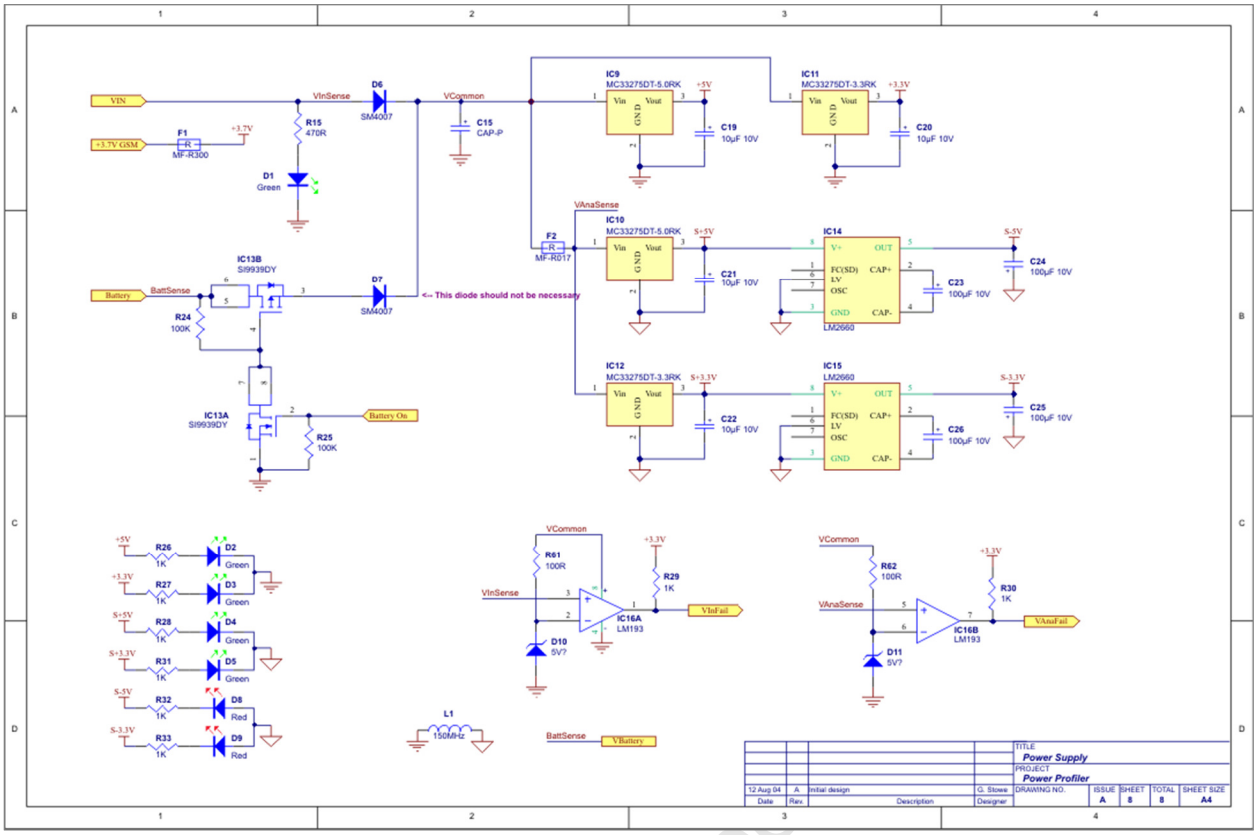
Prototype One: Energy Measurement



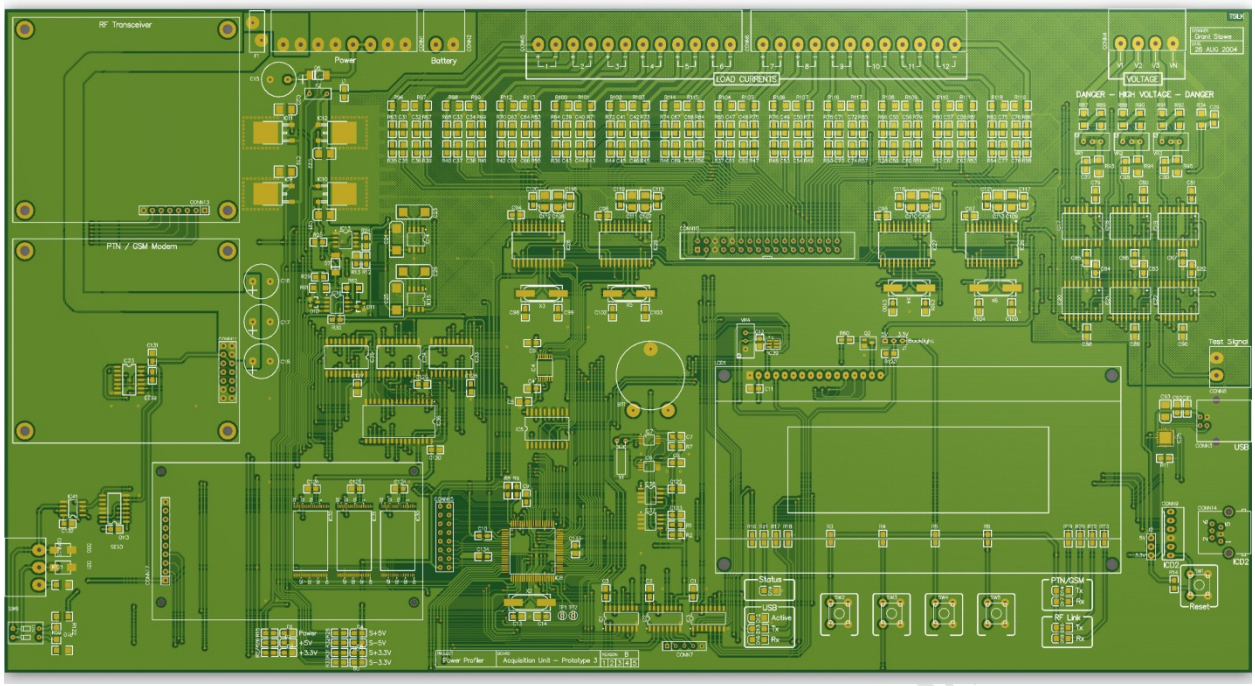
Prototype One: Communications Interfaces



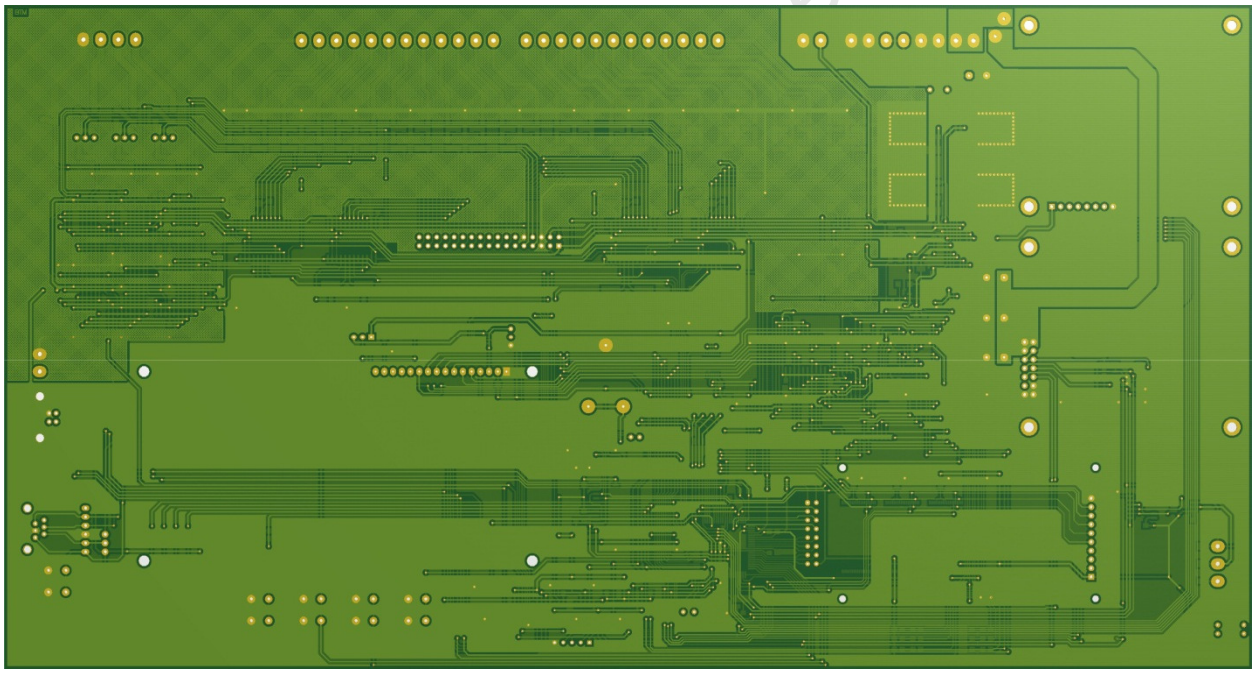
Protoype One: Operator Interfaces



Prototype One: Power Supply



Prototype 3 – Top PCB Layout Rendering (Not to scale)

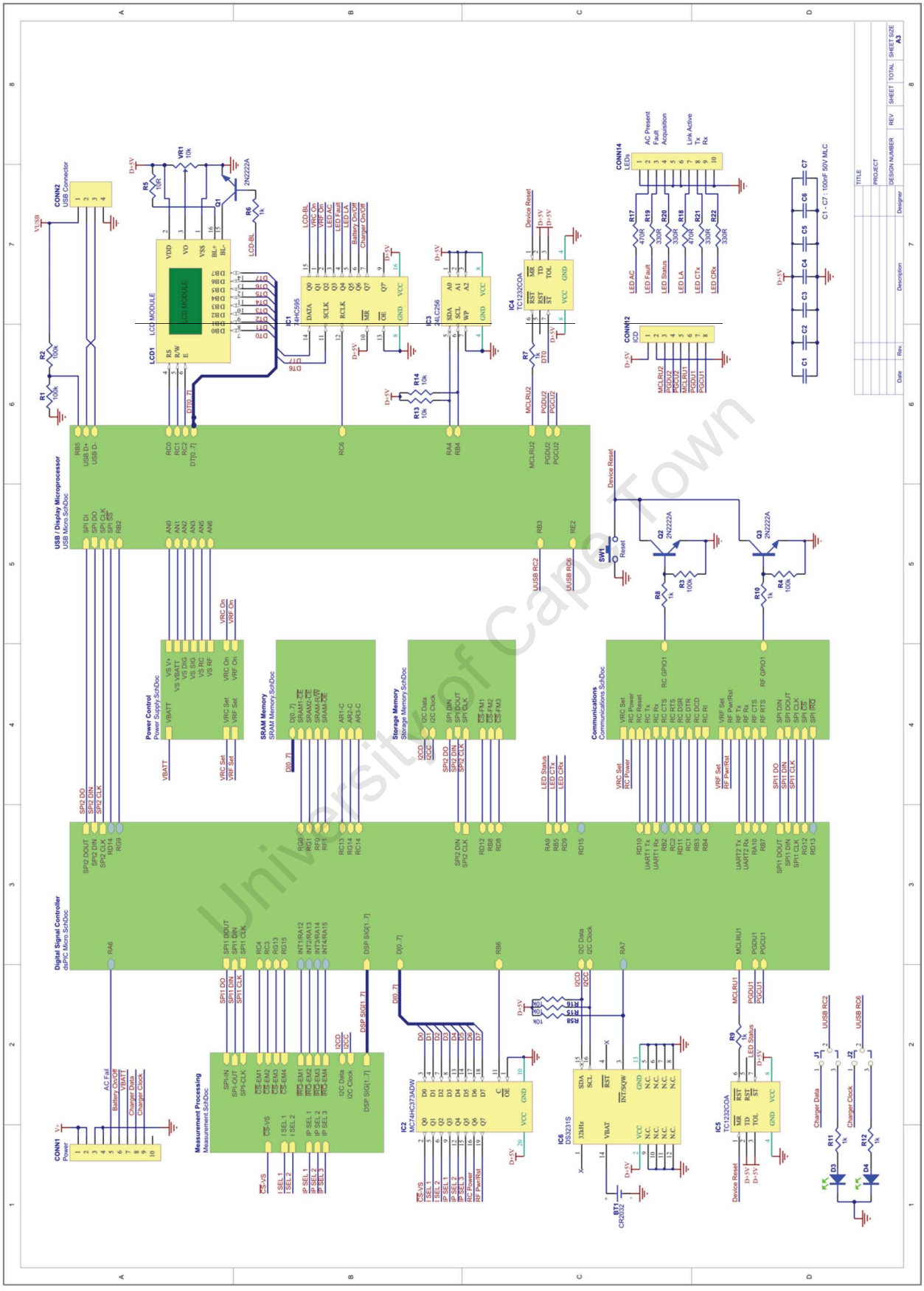


Prototype 3 – Bottom PCB Layout Rendering (Not to scale)

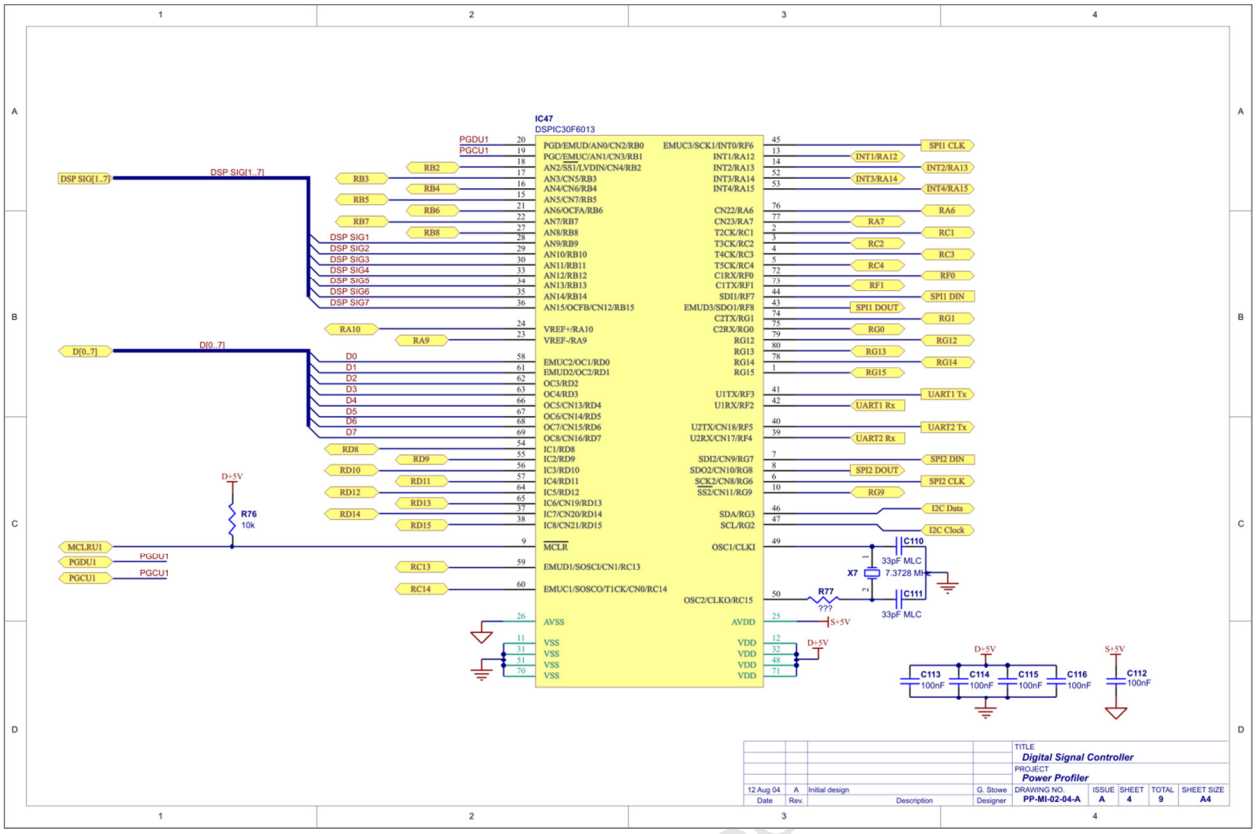
Appendix B

Power Profiler Product Design

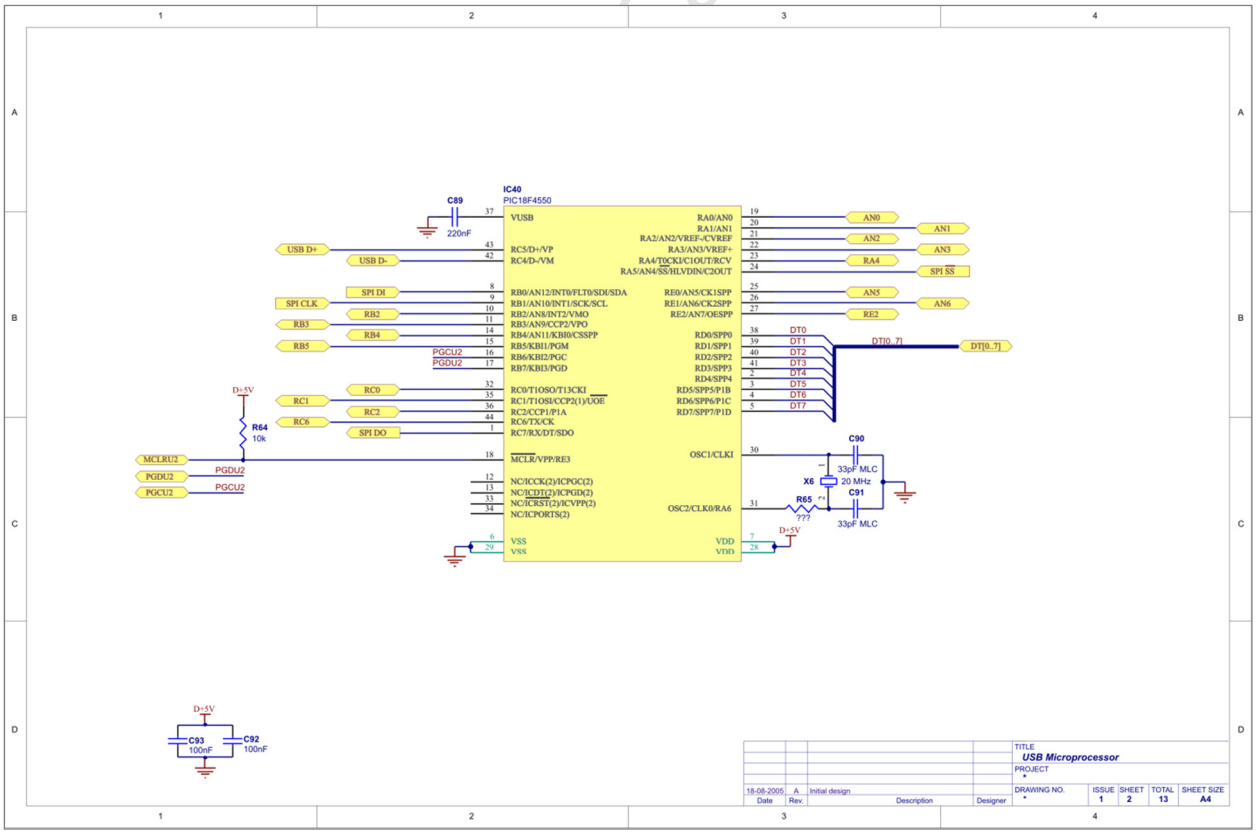
Control Board Design



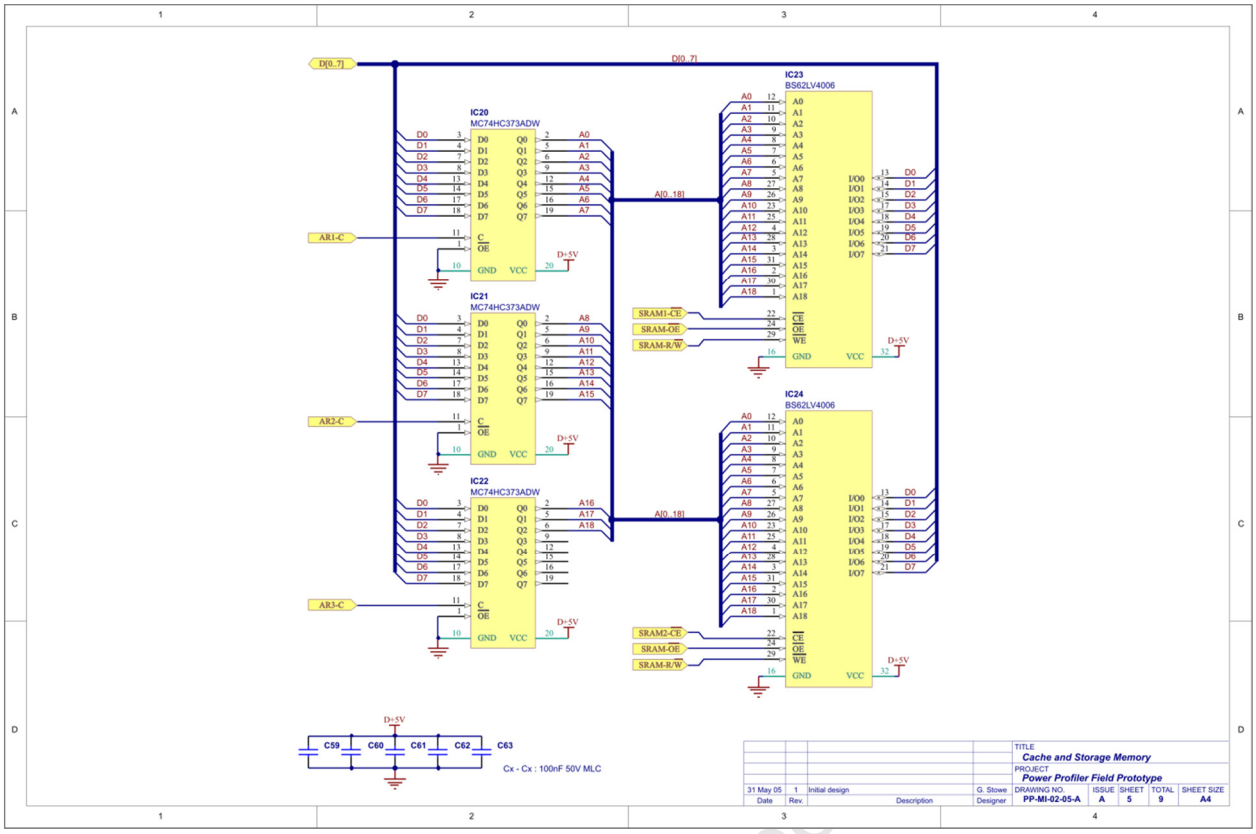
Control Board: Schematic Overview



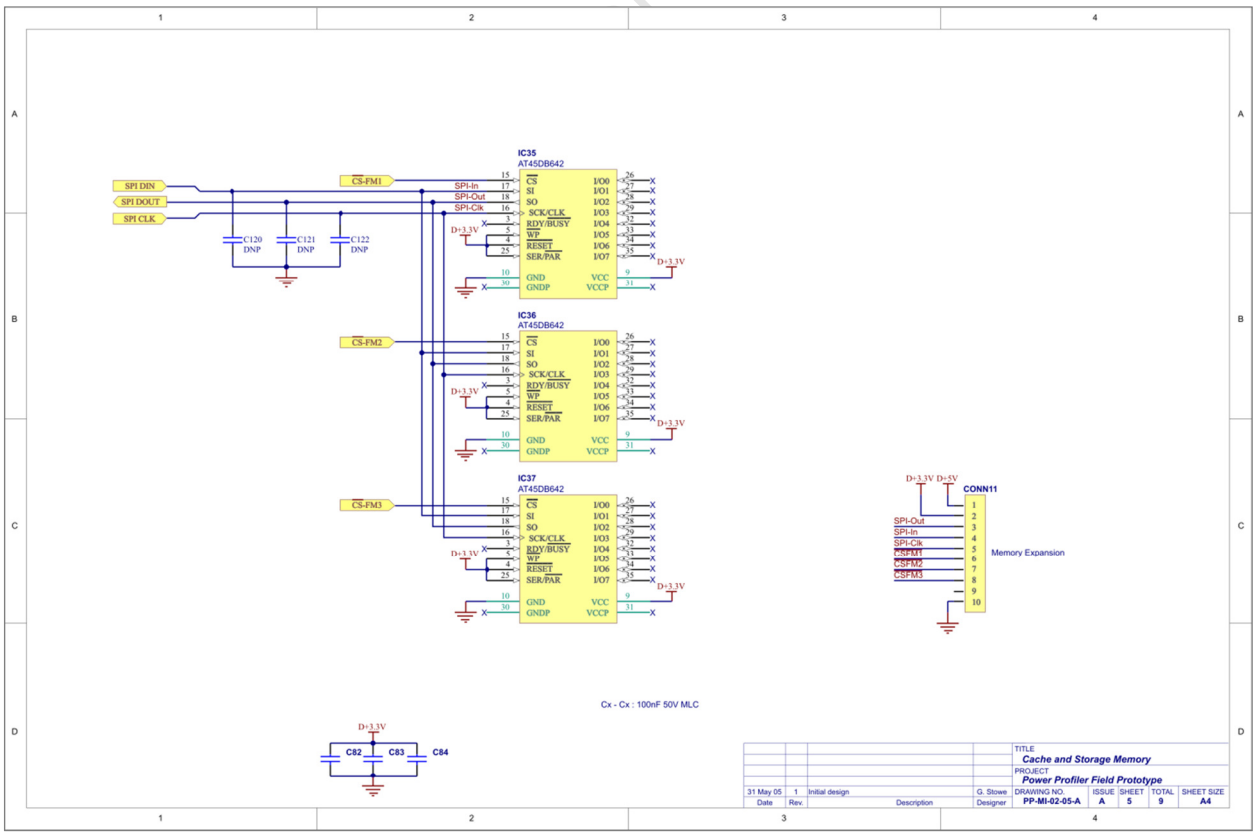
Control Board: Digital Signal Controller



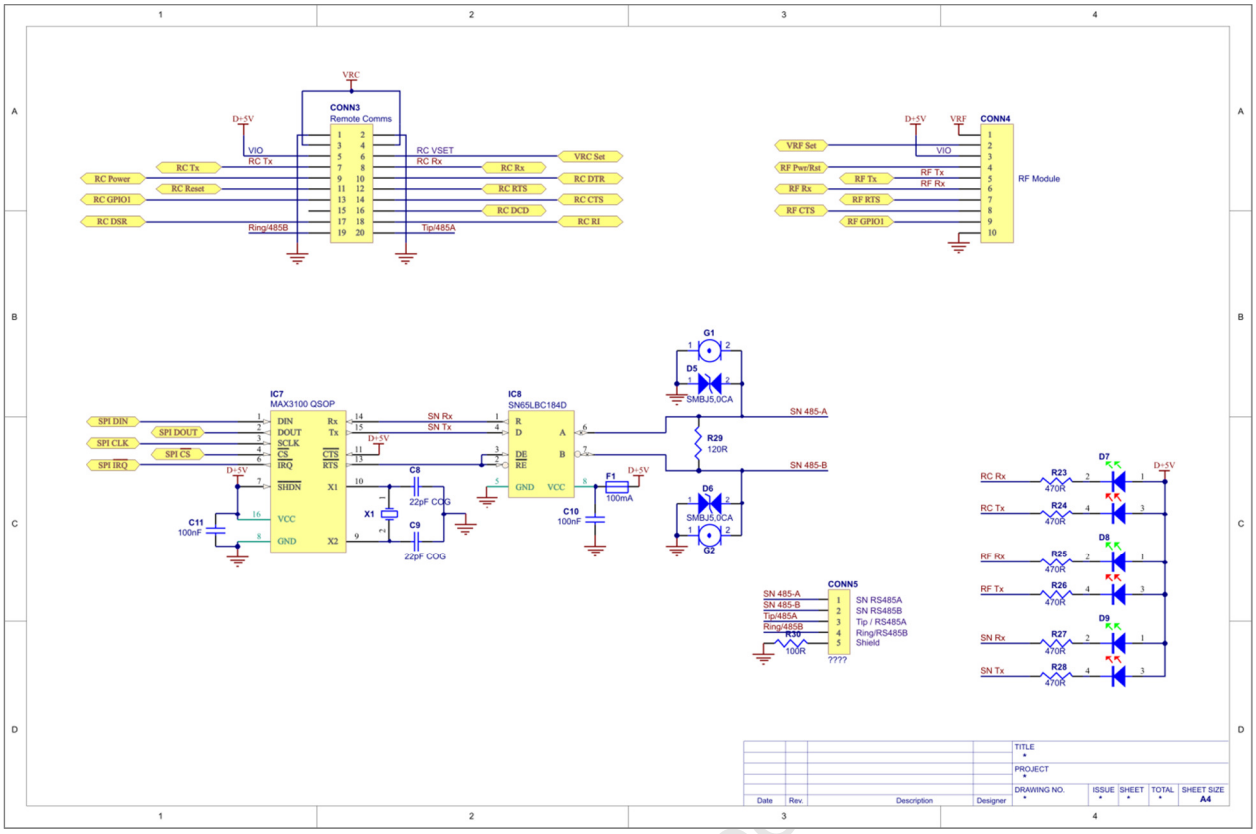
Control Board: USB Microprocessor



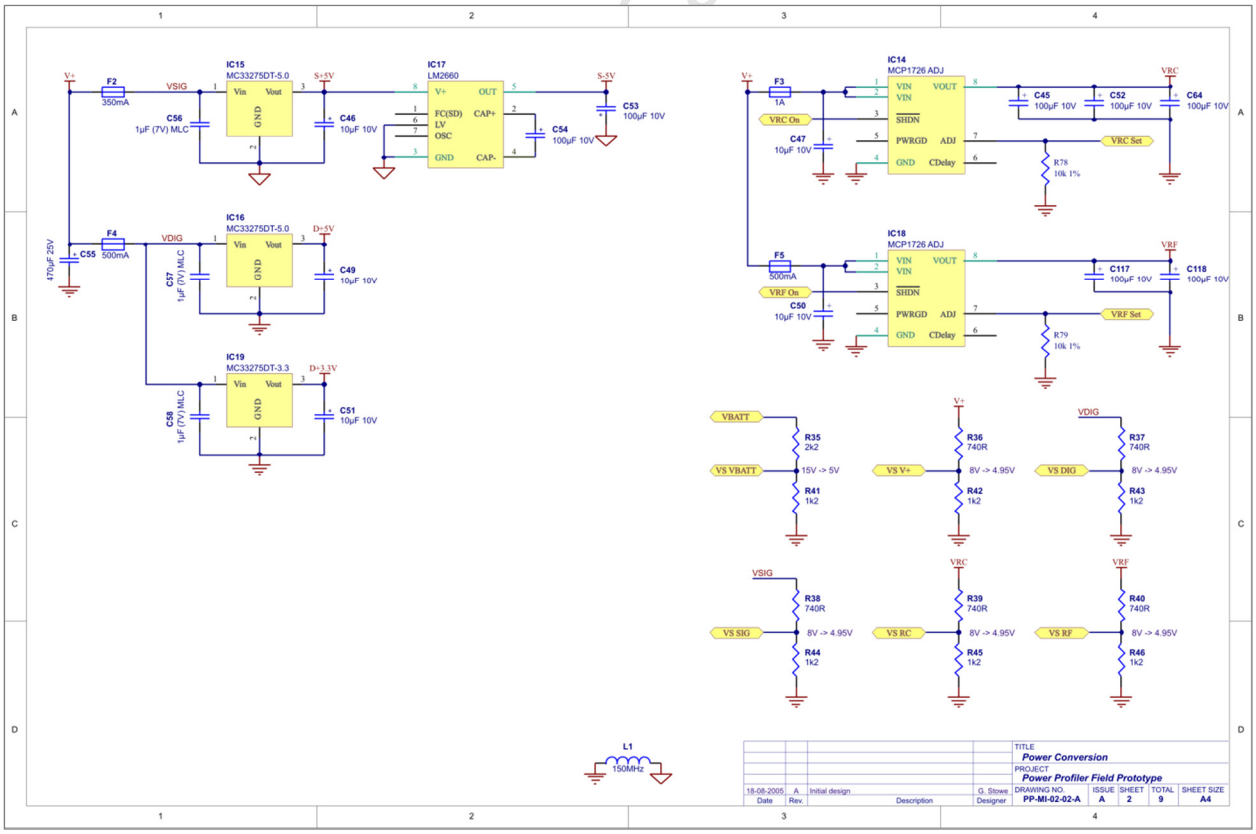
Control Board: Cache Memory



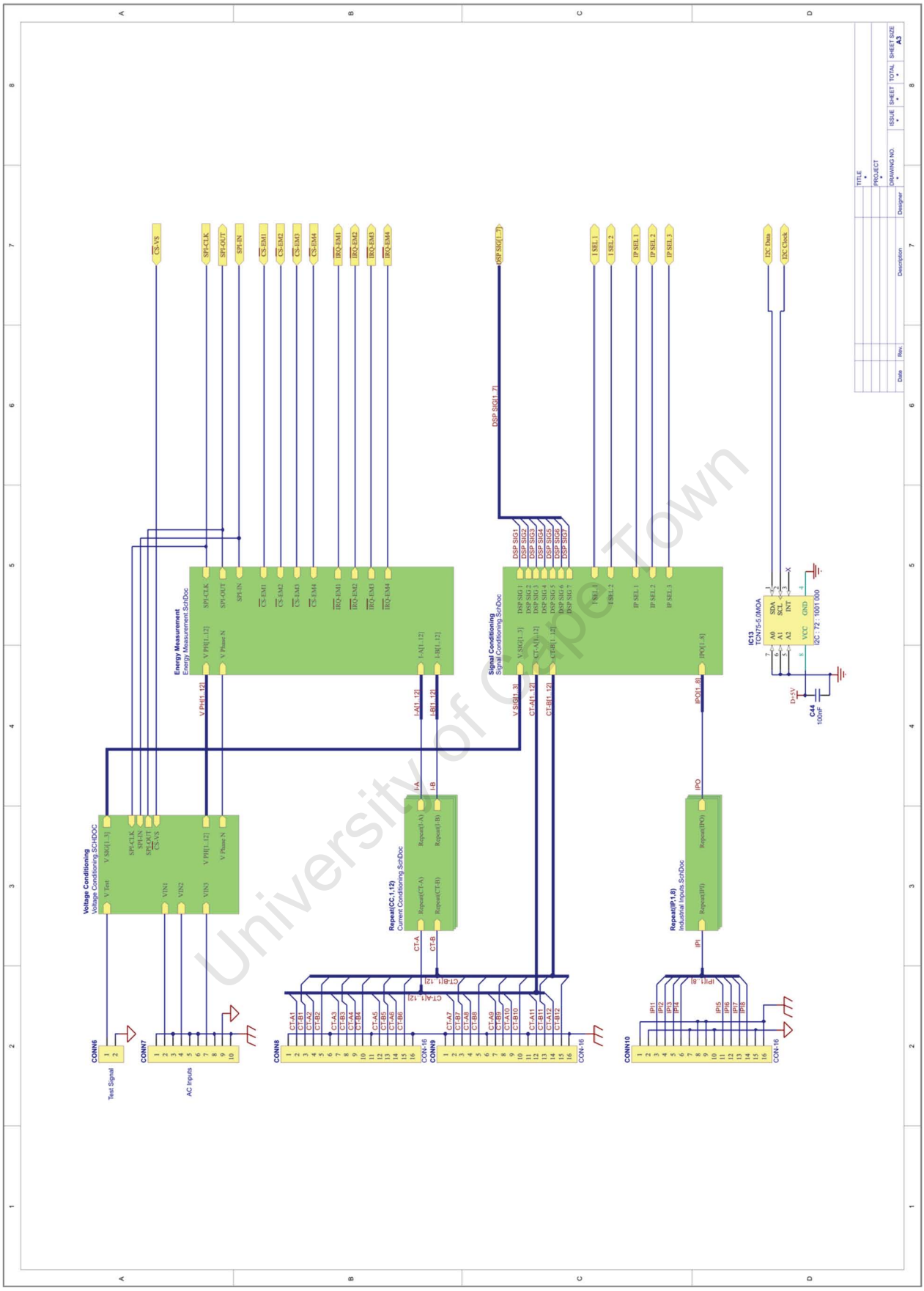
Control Board: Storage Memory



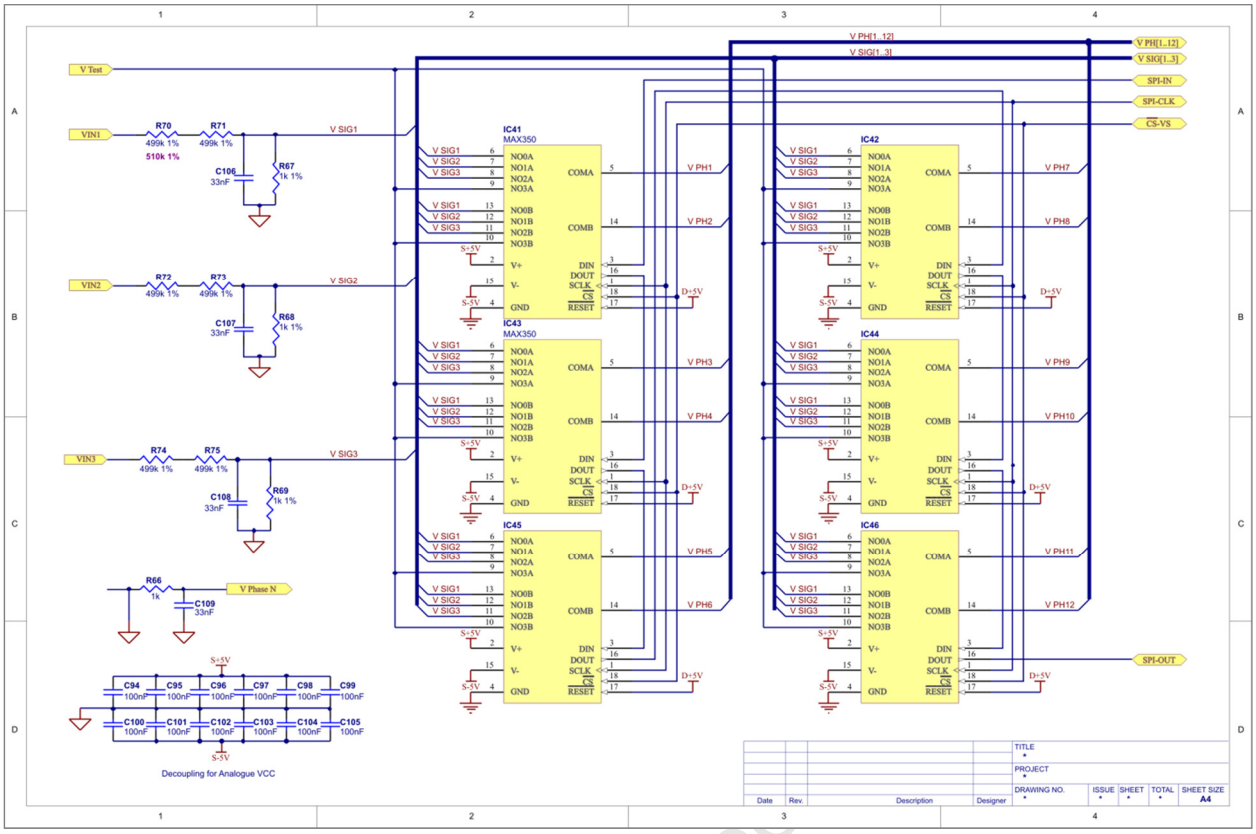
Control Board: Communications



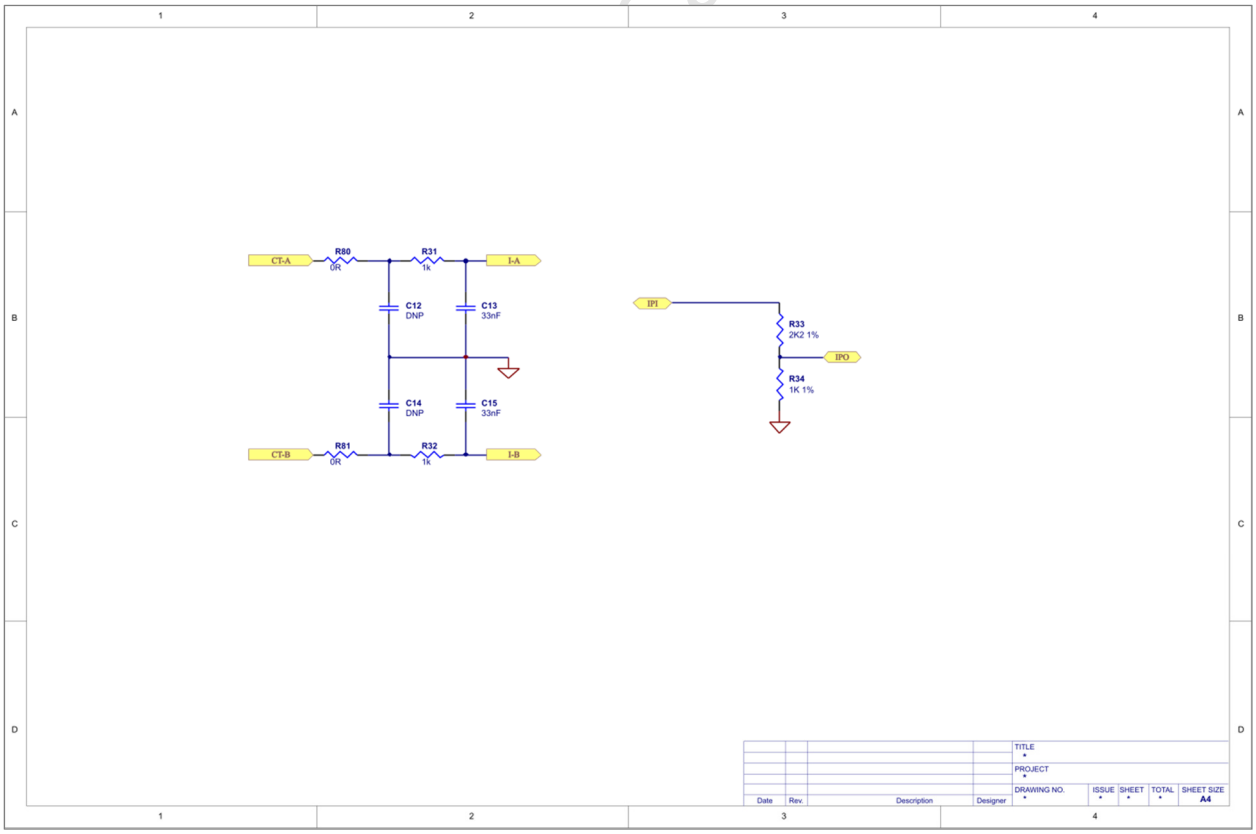
Control Board: Power Supply



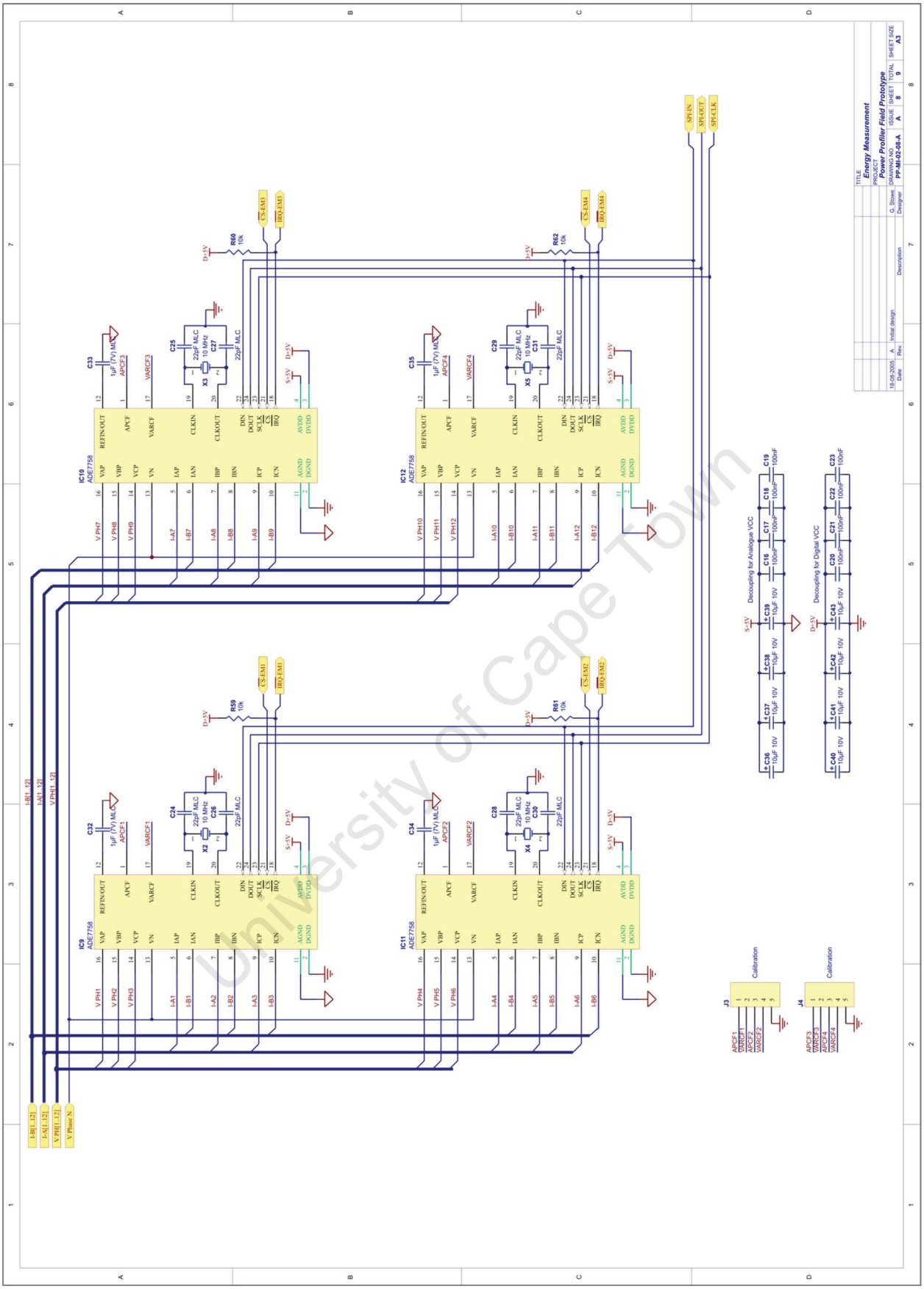
Control Board: Measurement



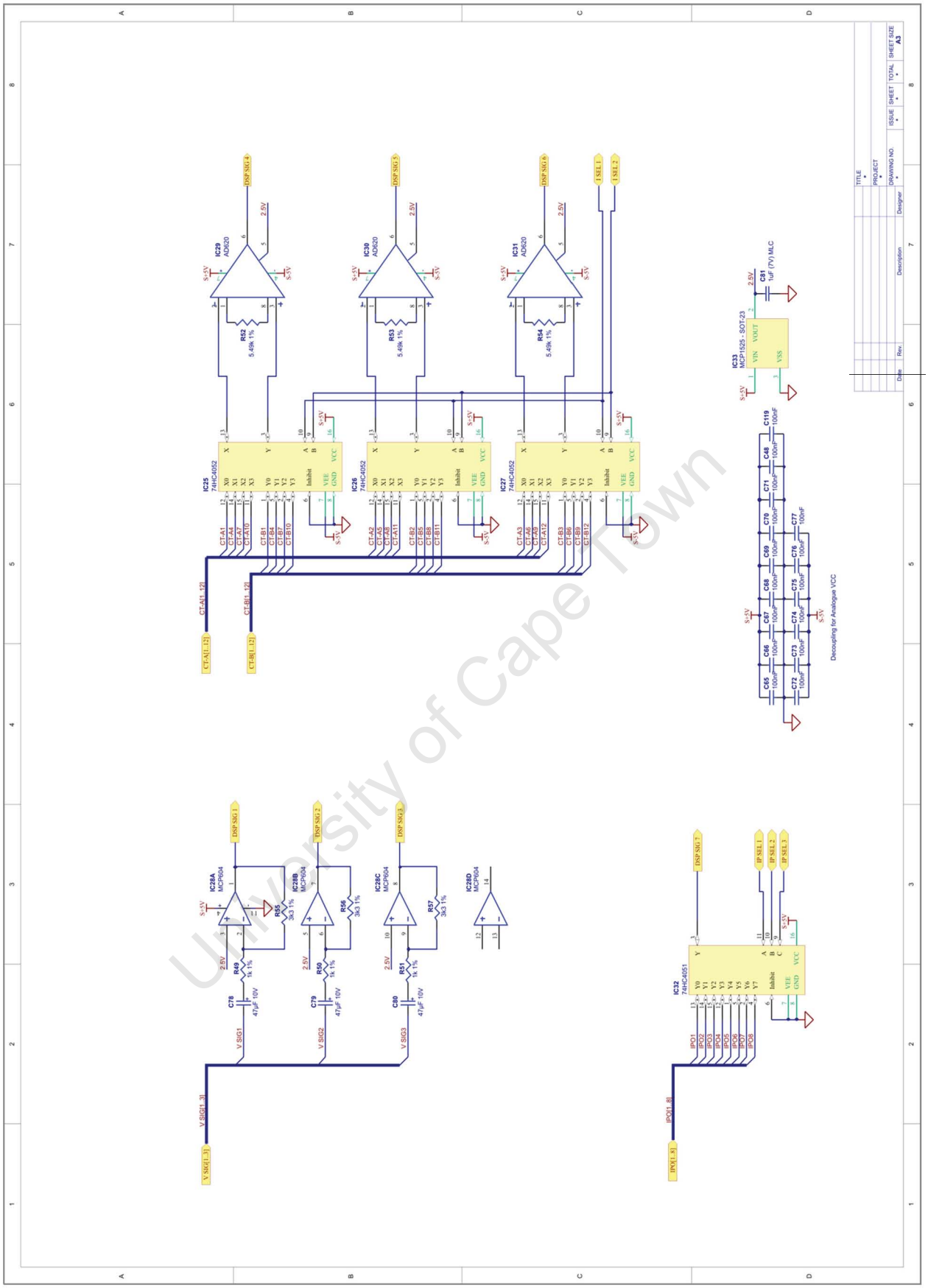
Control Board: Voltage Channel Conditioning



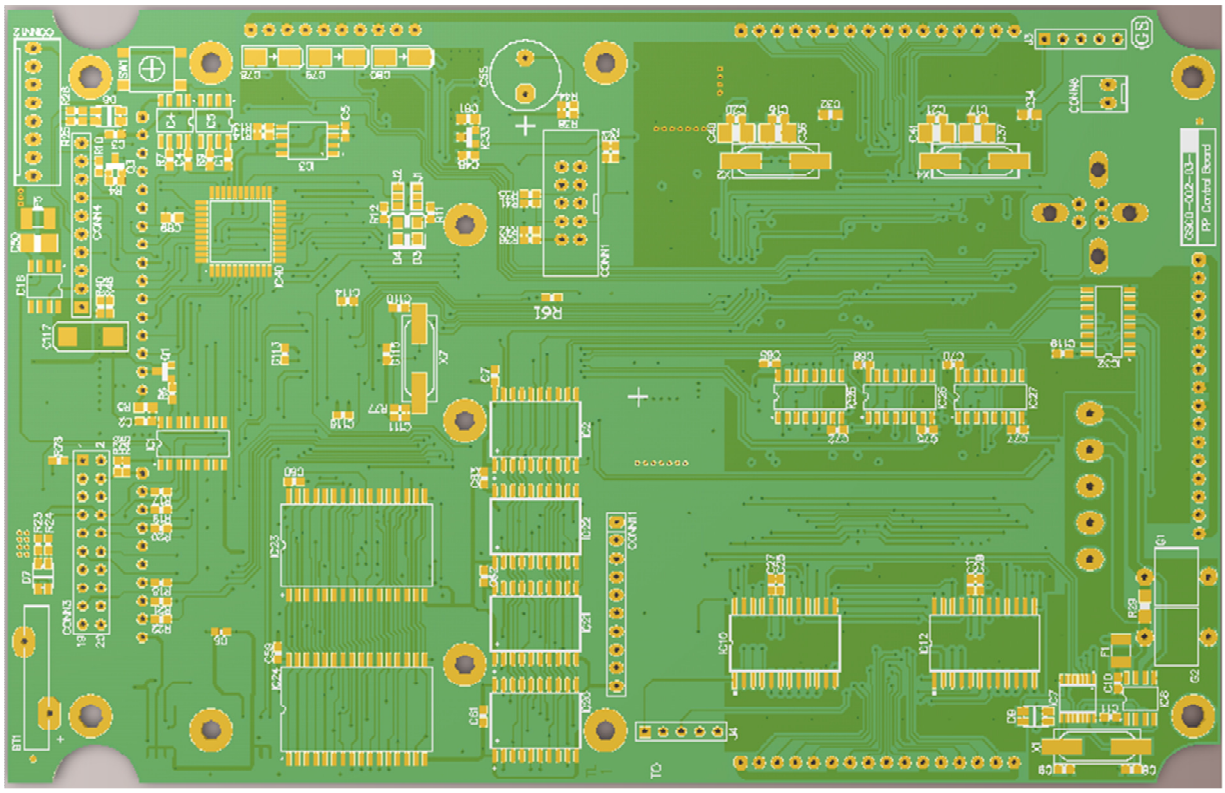
Control Board: Current and Industrial Input Channel Signal Conditioning



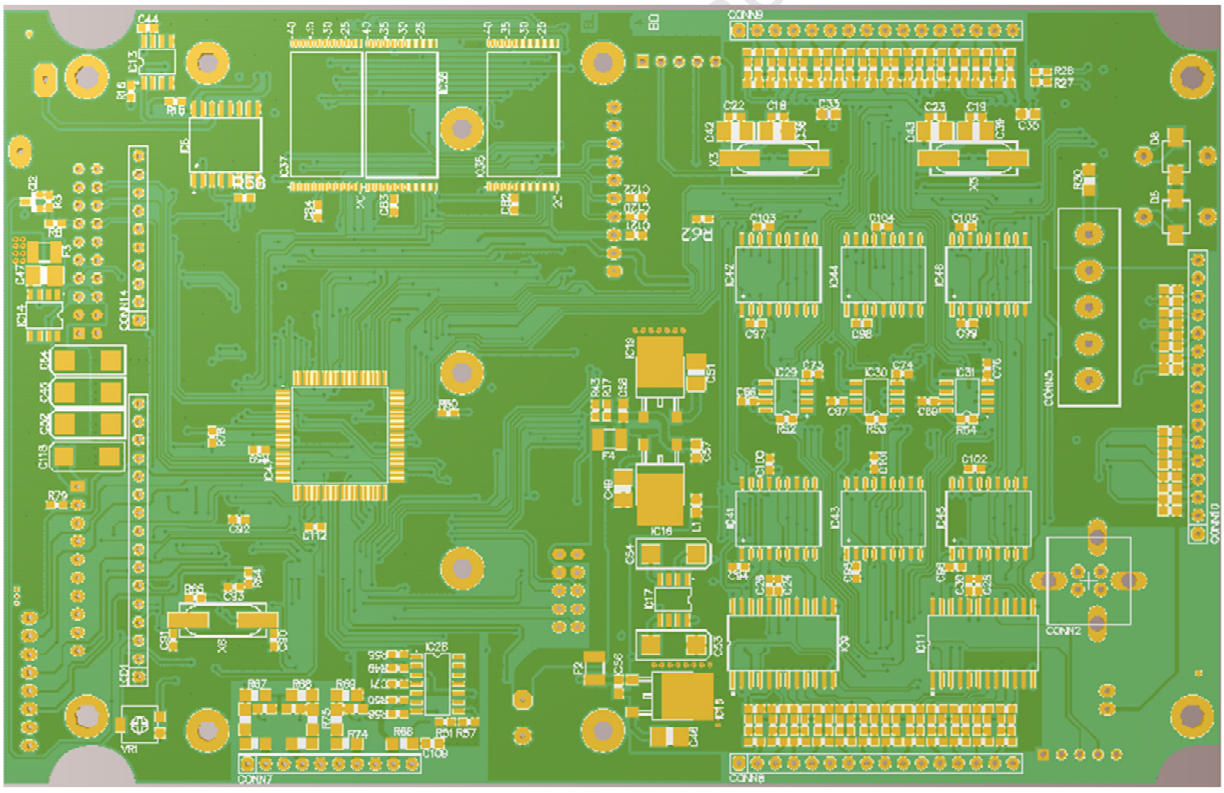
Control Board: Energy Measurement



Control Board: Analog Signal Conditioning for dsPIC Interface

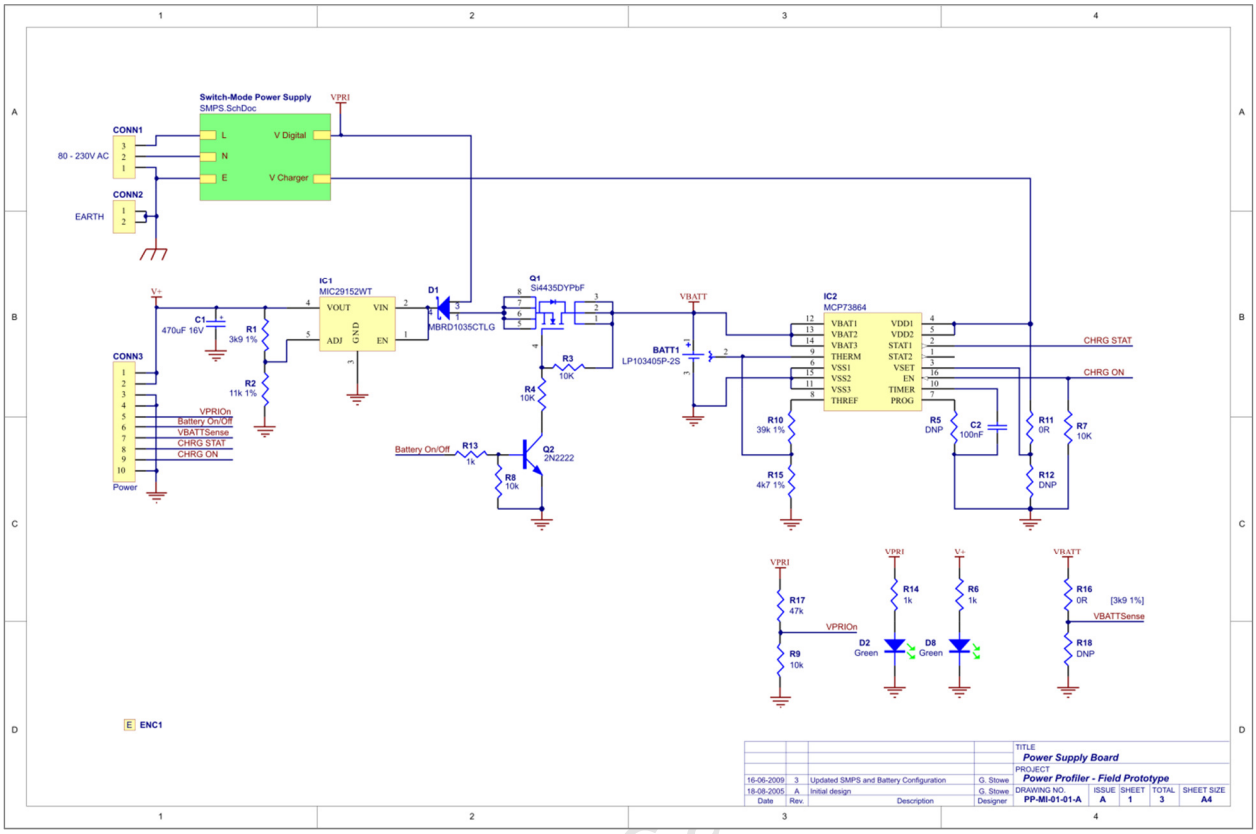


Control Board: Top Layer Render (Not to scale)

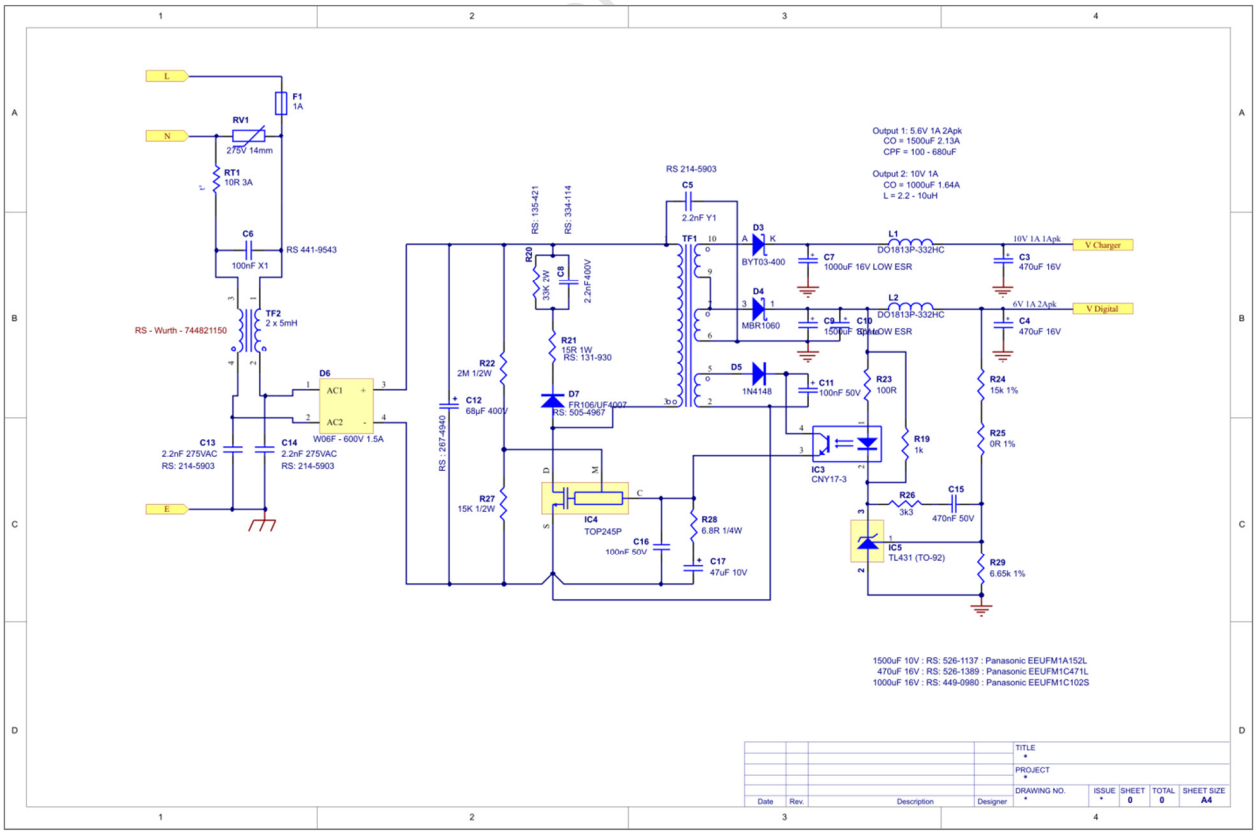


Control Board: Bottom Layer Render (Not to scale)

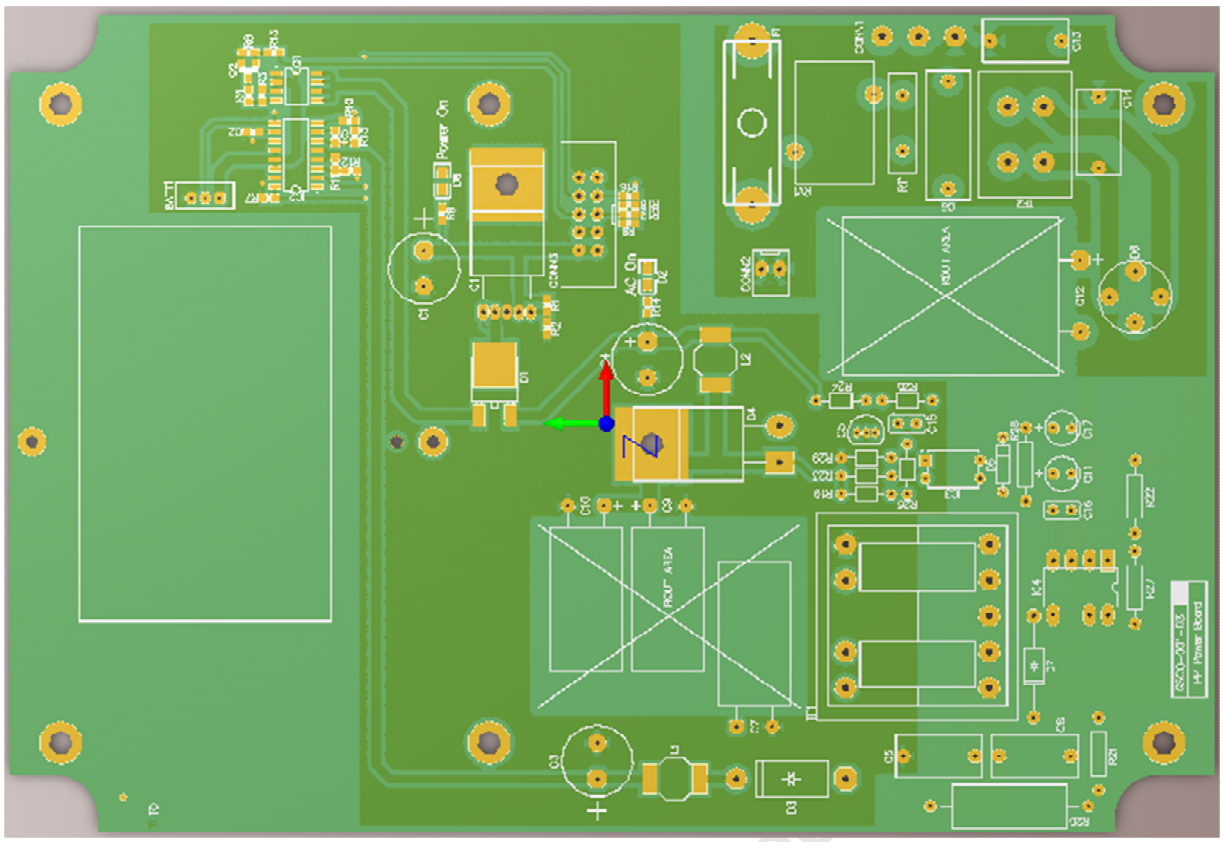
Power Board Design



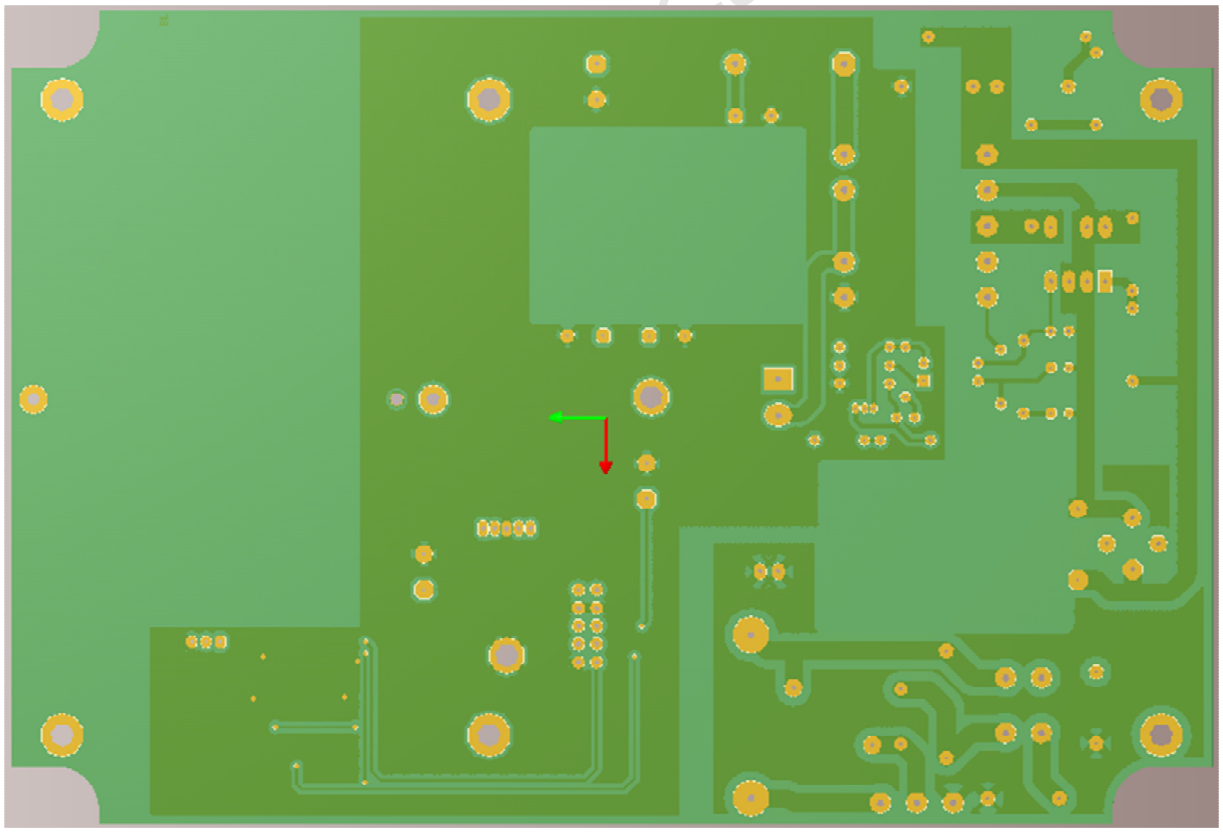
Power Supply Board: Power Supply Overview



Power Supply Board: Switch Mode Power Supply



Power Supply Board: Top Layer Render (Not to scale)

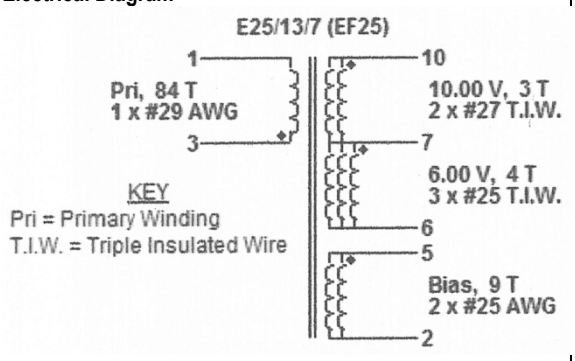


Power Supply Board: Bottom Layer Render (Not to scale)

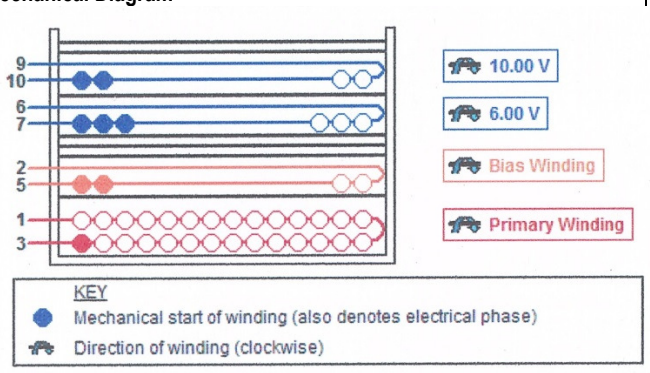
Switch-Mode Power Supply Transformer Construction

SWITCH-MODE POWER SUPPLY TRANSFORMER CONSTRUCTION

Electrical Diagram



Mechanical Diagram



Wiring Instructions

Primary Winding

Start on pin 3 and wind 84 turns (x1 filar) of item [5] in 2 layers from left to right. At the end of the 1st layer continue to wind the next layer from right to left. On the final layer spread the winding evenly across the entire bobbin. Finish this winding on pin 1.

Add 1 layer of tape [3] for insulation.

Bias Winding

Start on pin 5 and wind 9 turns (x2 filar) of item [6]. Wind in same rotational direction as the primary winding. Spread the winding evenly across the entire bobbin. Finish this winding on pin 2.

Add 3 layers of tape [3] for insulation.

Secondary Winding

Start on pin 7 and wind 4 turns (x3 filar) of item [7]. Spread the winding evenly across the entire bobbin. Wind in the same rotational direction as the primary winding. Finish this winding on pin 6.

Add 1 layer of tape [3] for insulation.

Start on pin 10 and wind 3 turns (x2 filar) of item [8]. Spread the winding evenly across the entire bobbin. Wind in the same rotational direction as the primary winding. Finish this winding on pin 9.

Add 2 layers of tape [3] for insulation.

Core Assembly

Assemble and secure the core halves (item [1]).

Varnish

Dip varnish uniformly in item [4]. Do not vacuum impregnate.

Comments

1. For non-margin wound transformers use triple insulated wire for all secondary windings.
2. For lowest EMI place diode in return leg of the secondary (diode cathode connected to finish / termination pin side of secondary).

Materials

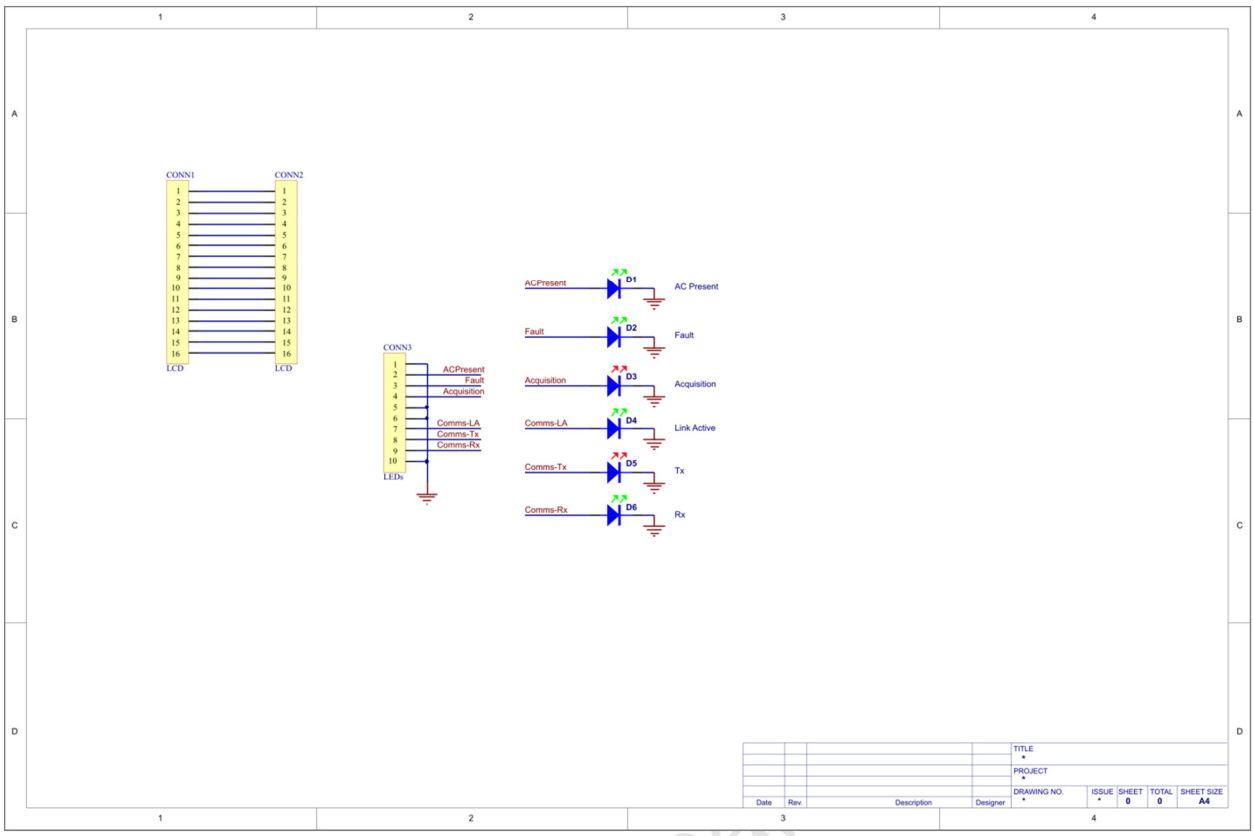
[1] Core: E25/13/7 (EF25), NC-2H (Nicera) or equivalent, gapped for ALG of 137 nH/t².
[2] Bobbin: Generic 5 pri. 5 sec.
[3] Barrier tape: Polyester film (1 mil base thickness), 15.30 mm wide.
[4] Varnish
[5] Magnet wire: 29 AWG, solderable double coated
[6] Magnet wire: 25 AWG, solderable double coated
[7] Triple insulated wire: 25 AWG
[8] Triple insulated wire: 27 AWG

Electrical Test Specifications

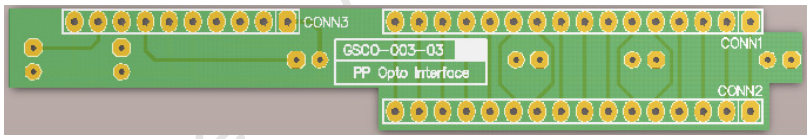
Electrical strength	60 Hz 1 second, from pins 1 – 5 to pins 6 – 10	3000 VAC
Nominal primary inductance	Measured at 1Vpk-pk, typ. switching frequency, pins 1 to 5 with all other windings open	1045 +/- 10%
Maximum primary leakage	Measured between pin 1 to pin 5, with all other windings shorted	31.4µH

Specified by Power Integrations Software for Custom SMPS Design

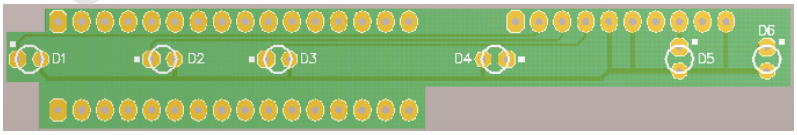
Display Board



Display Board: Schematic

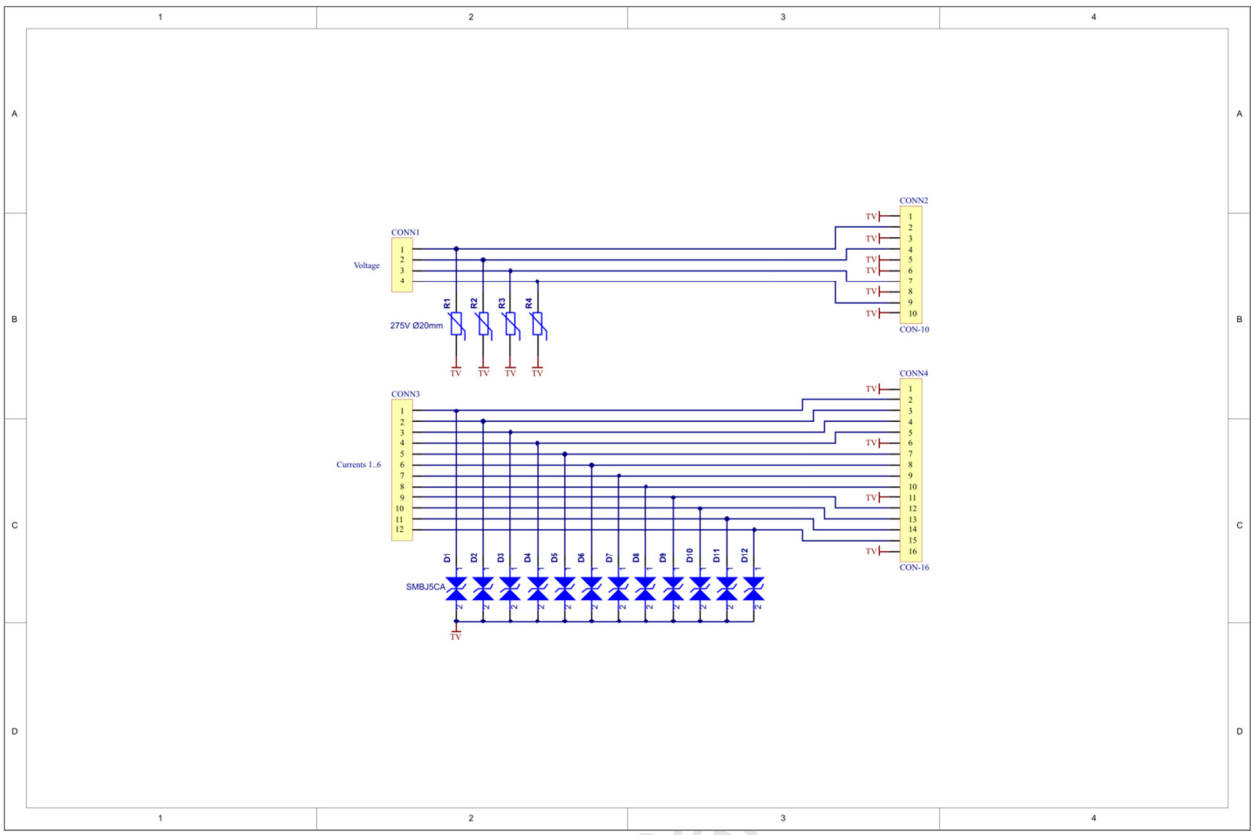


Display Board: Top Layer Render (Not to scale)

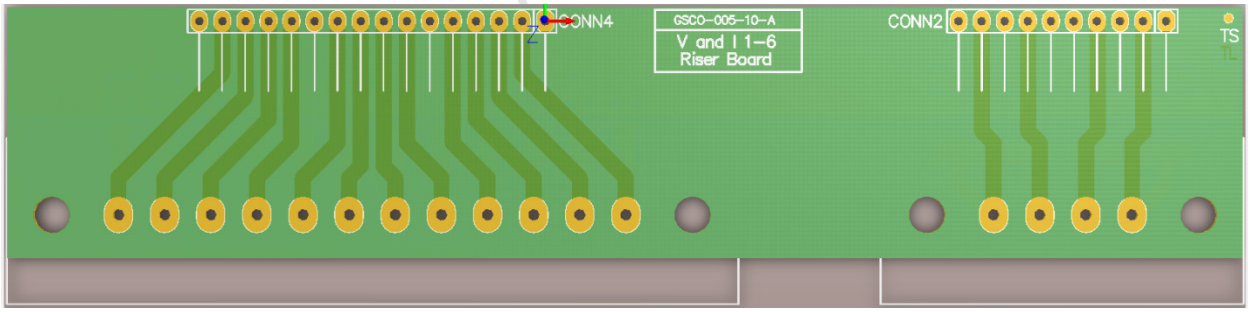


Display Board: Bottom Layer Render (Not to scale)

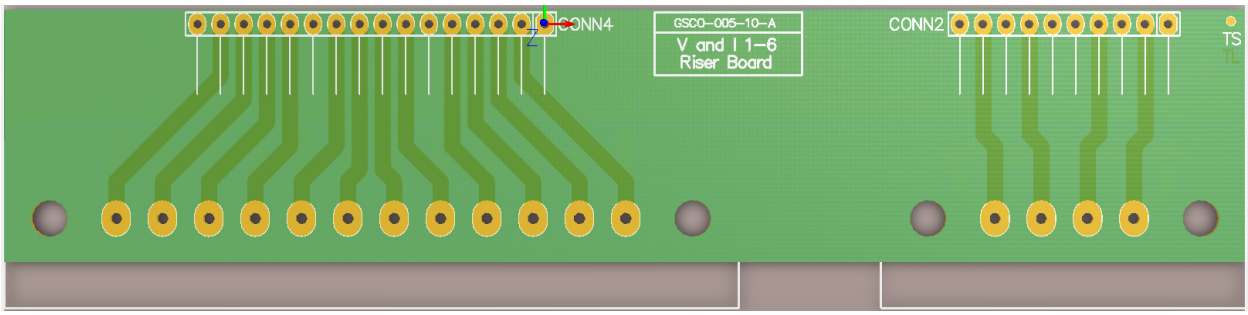
Voltage and Currents 1..6 Riser Board



Voltage and Current Riser Board: Schematic

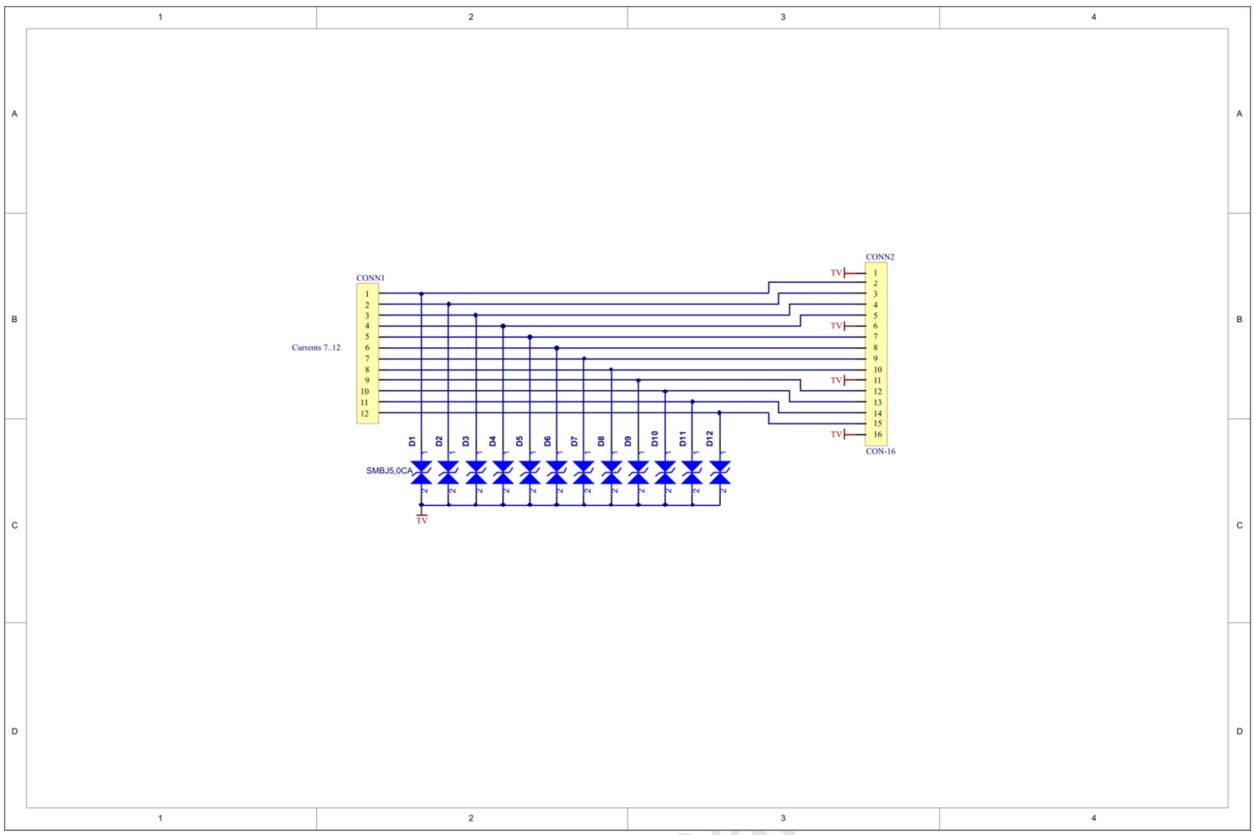


Voltage and Current Riser Board: Top Layer Render (Not to scale)

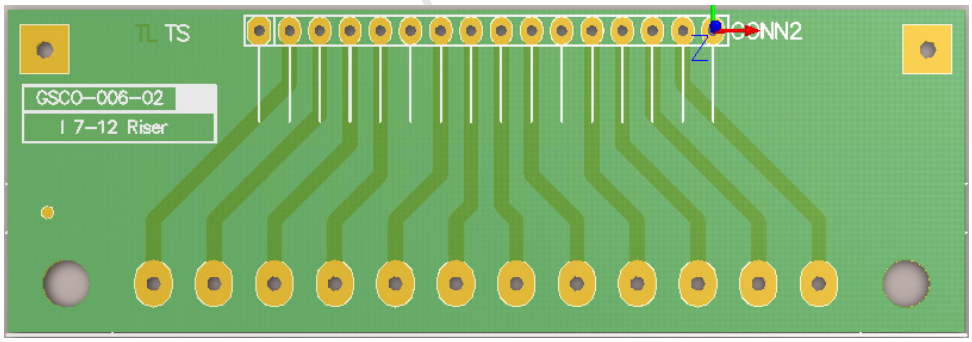


Voltage and Current Riser Board: Bottom Layer Render (Not to scale)

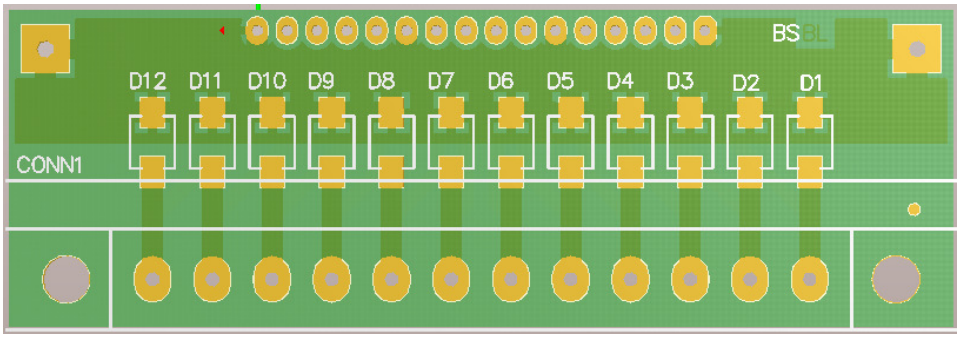
Currents 7..12 Riser Board



Currents 7..12 Riser Board: Schematic

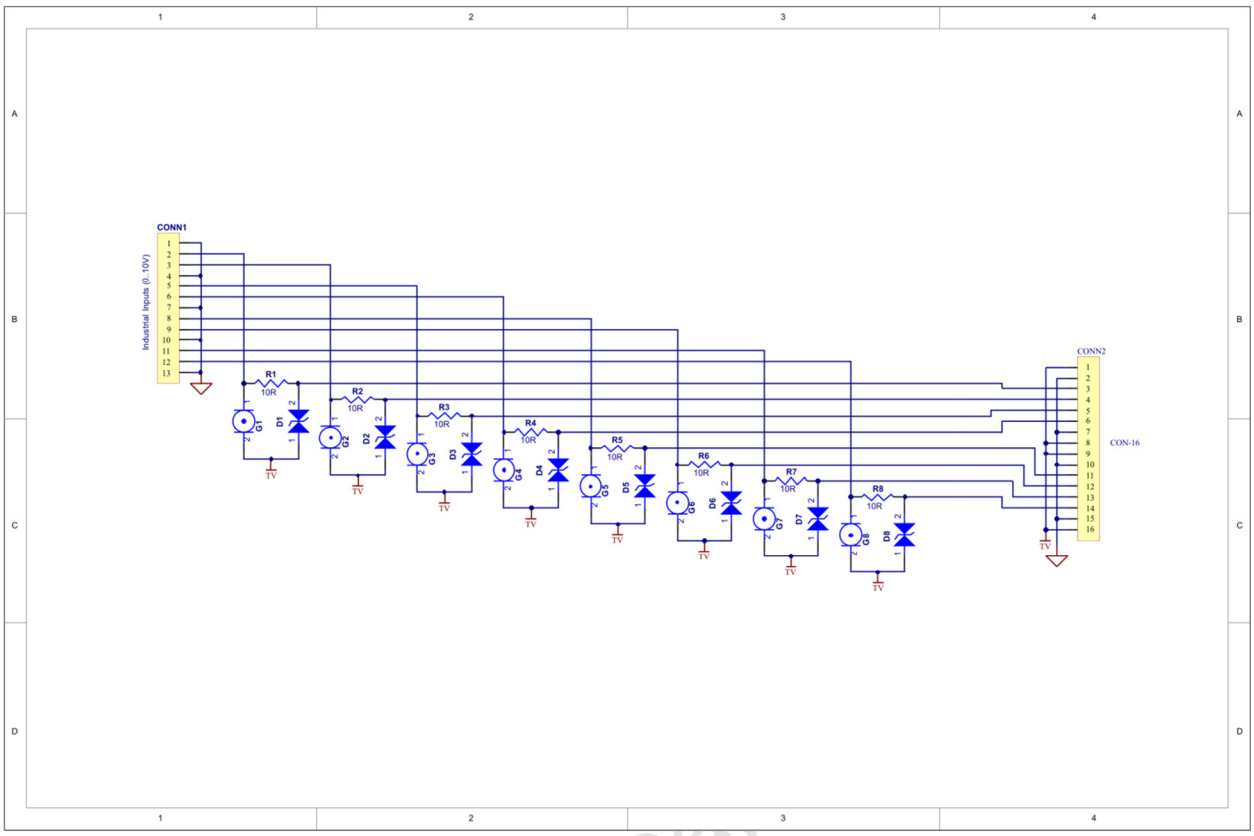


Currents 7..12 Riser Board: Top Layer Rendering (Not to scale)

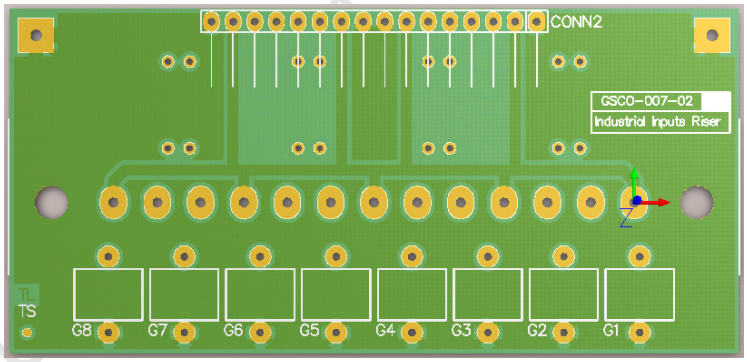


Currents 7..12 Riser Board: Bottom Layer Rendering (Not to scale)

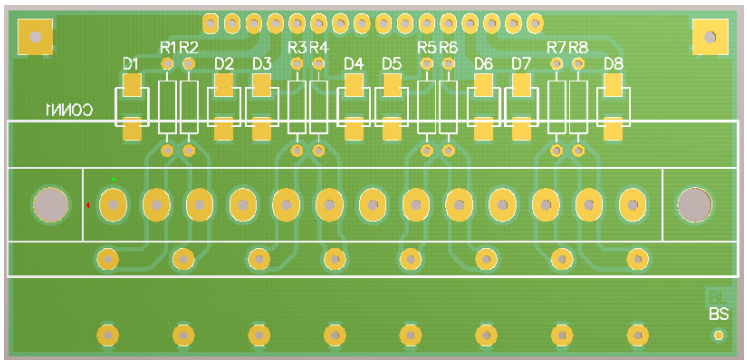
Industrial Inputs Riser



Industrial Inputs Riser Board: Schematic

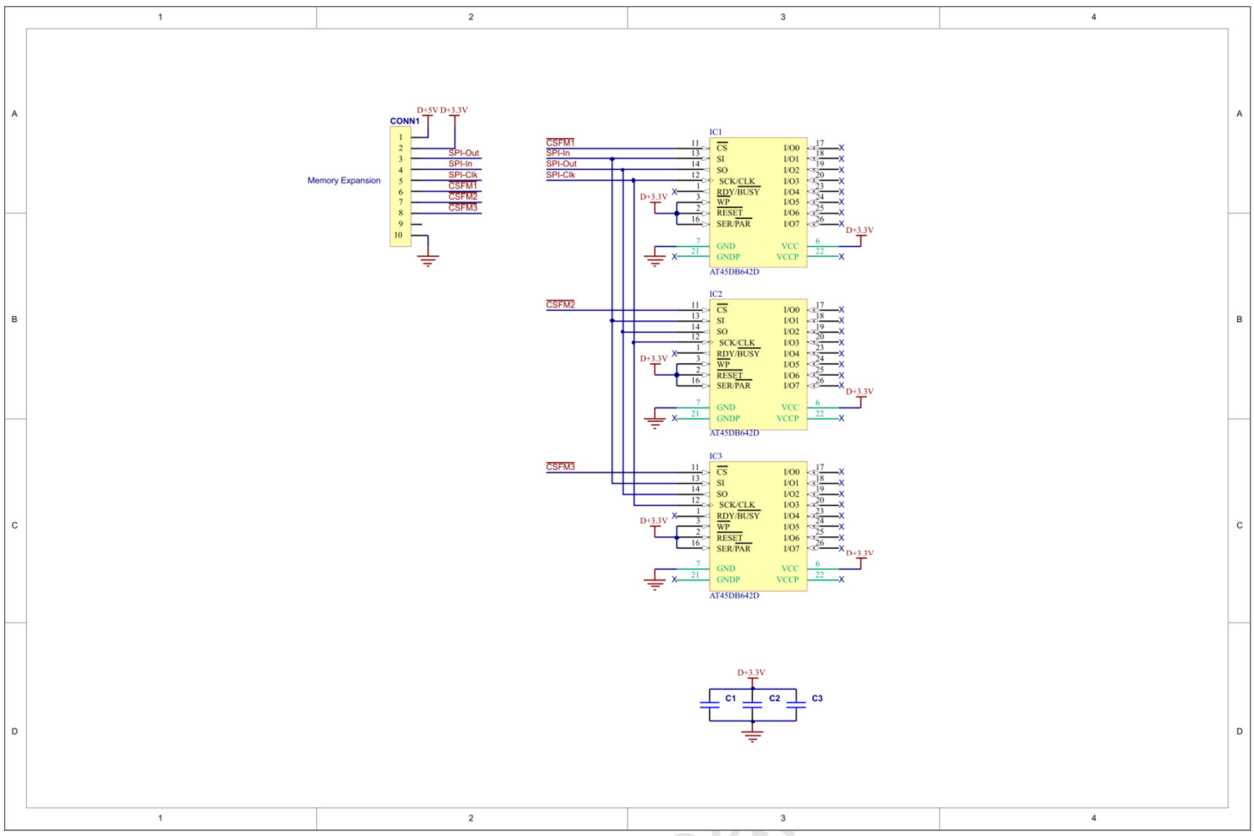


Industrial Inputs Riser Board: Top Layer Rendering (Not to scale)

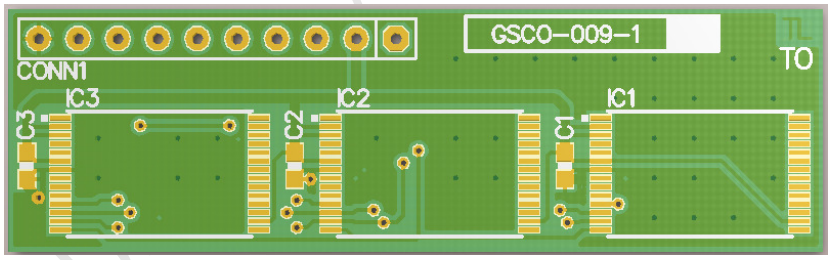


Industrial Inputs Riser Board: Bottom Layer Rendering (Not to scale)

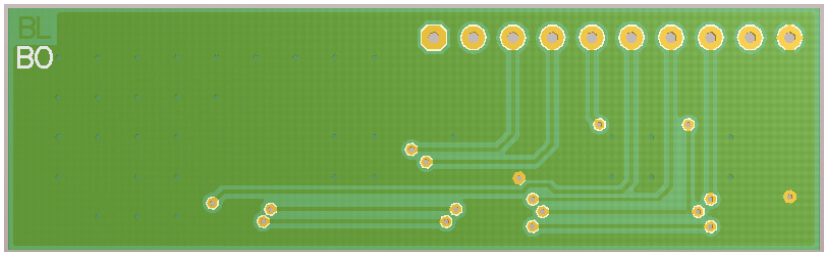
DataFlash Memory Interface Board



DataFlash Memory Interface Board: Schematic

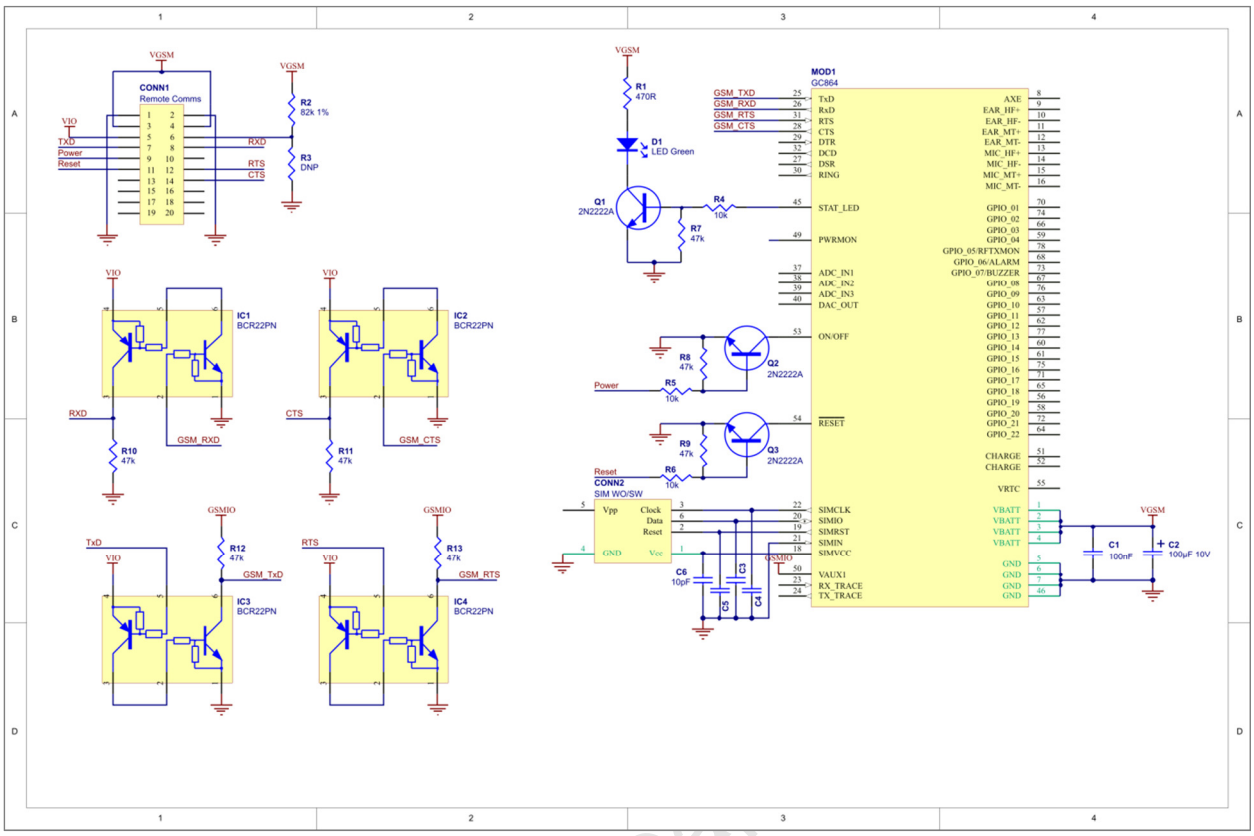


DataFlash Memory Interface Board: Top Layer Rendering (Not to scale)

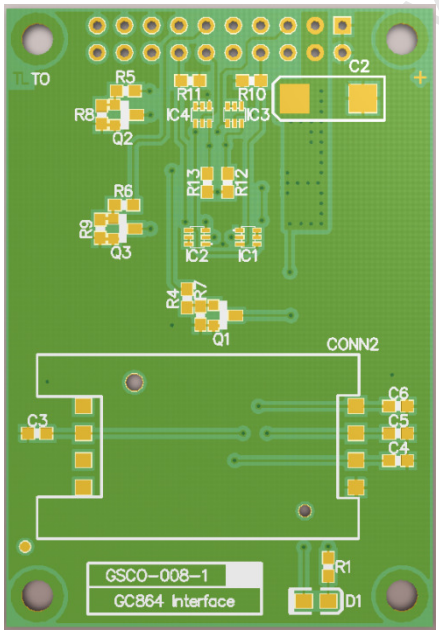


DataFlash Memory Interface Board: Bottom Layer Rendering (Not to scale)

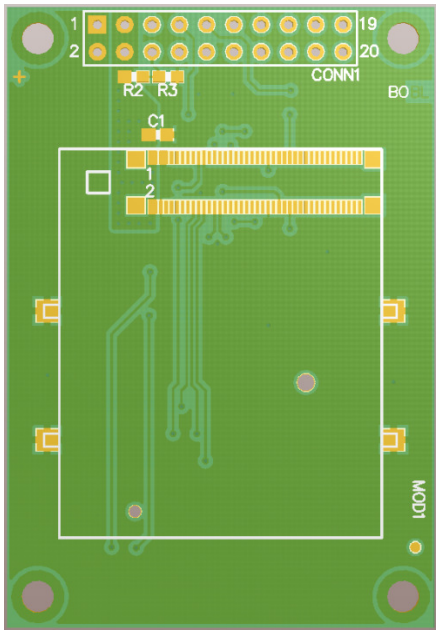
GSM Interface Board



GSM Interface Board: Schematic

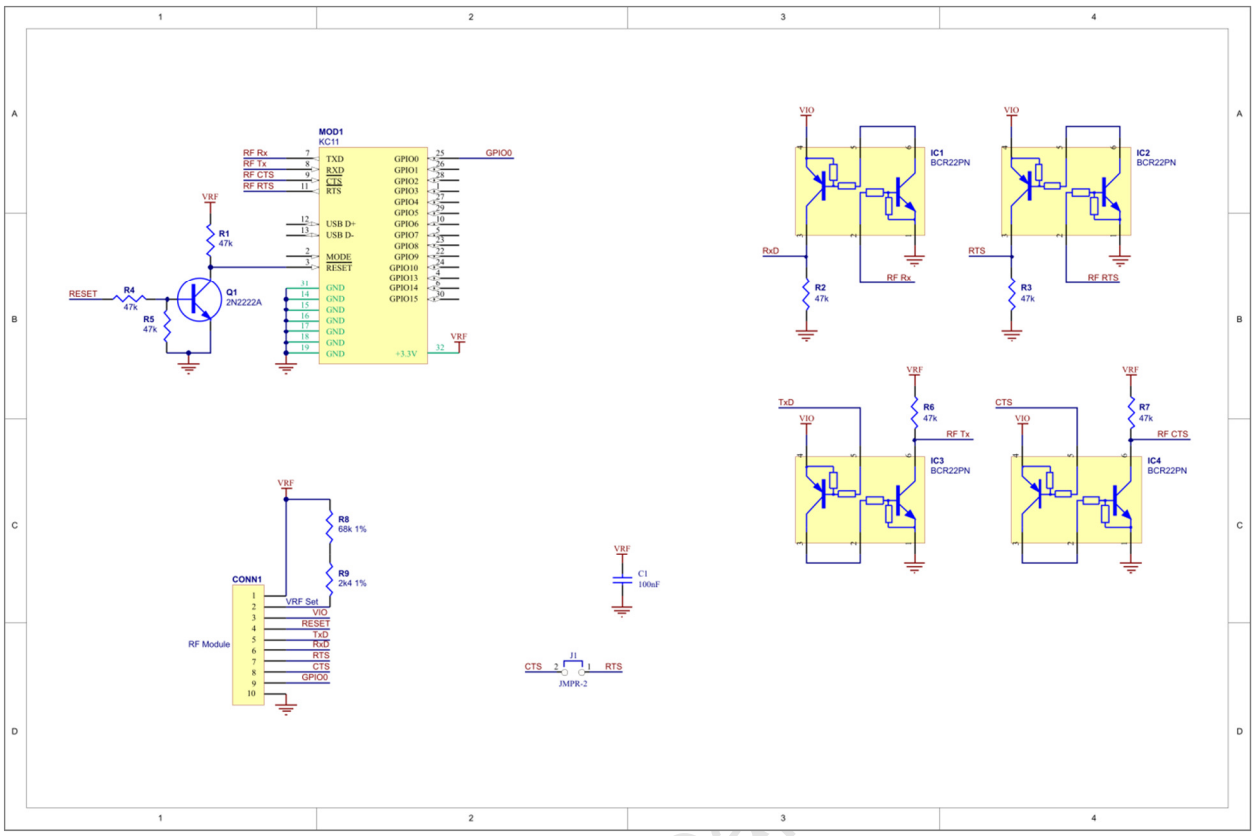


GSM Interface Board: Top Rendering (Not to scale)

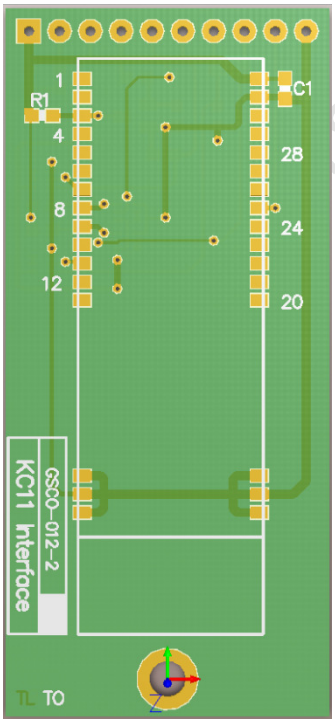


GSM Interface Board: Bottom Rendering (Not to scale)

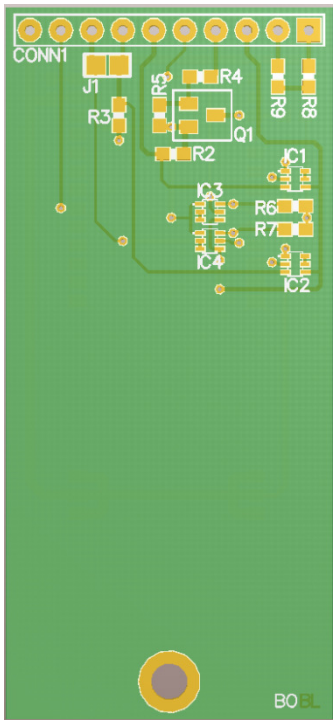
KC11 Bluetooth Interface Board



KC11 Bluetooth Interface Board: Schematic

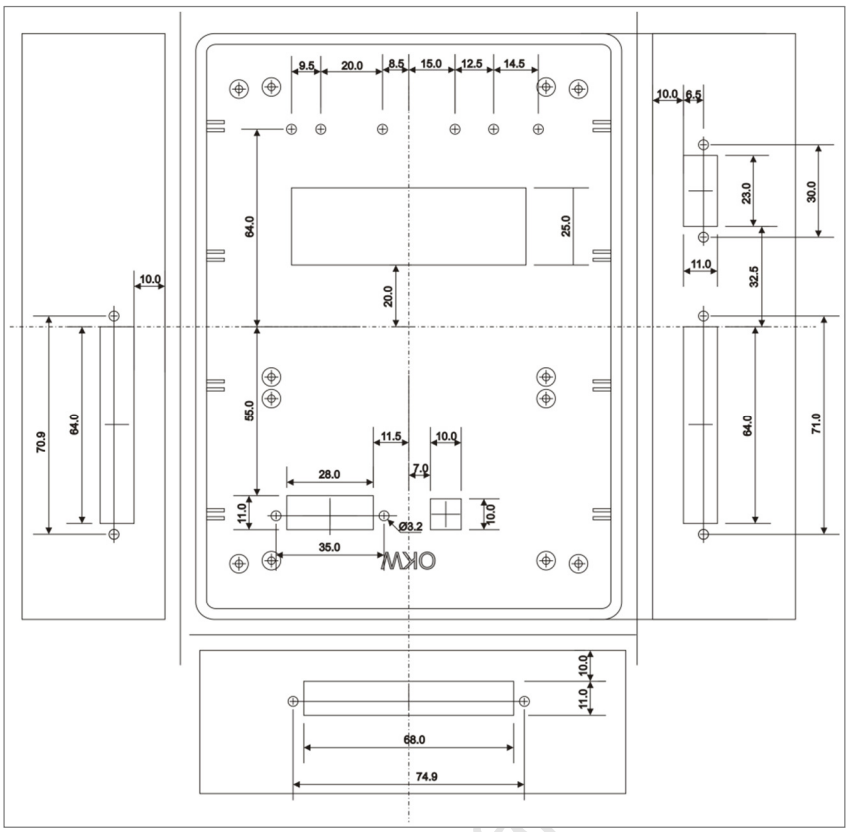


KC11 Bluetooth Interface Board: Top Layer Rendering (Not to scale)

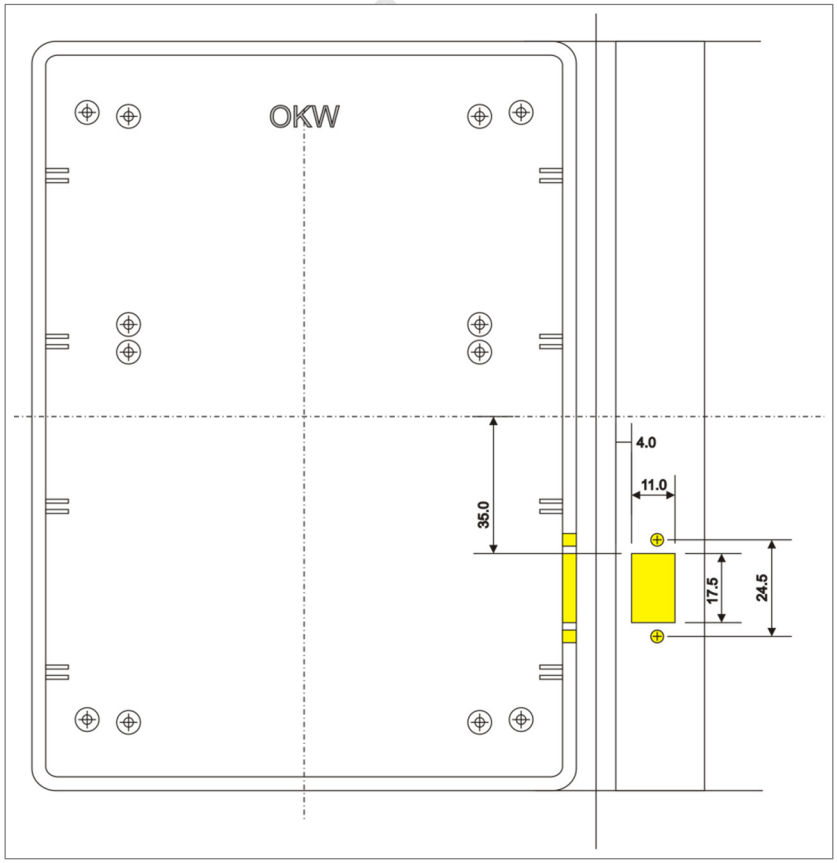


KC11 Bluetooth Interface Board: Bottom Layer Rendering (Not to scale)

Enclosure Machining Detail

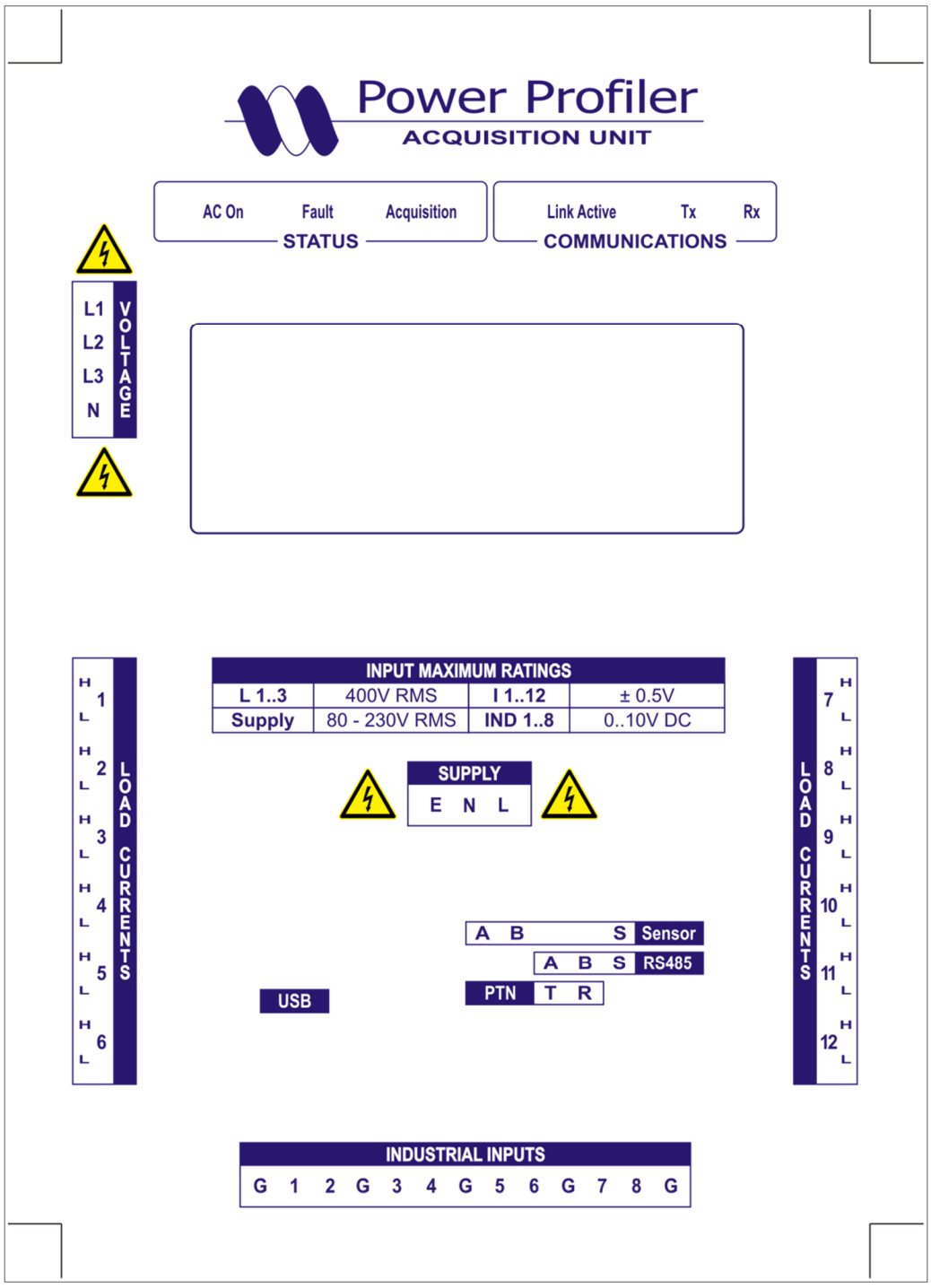


Machining Detail of OKW Enclosure Lid



Machining Detail of OKW Enclosure Base

Front Panel Decal Design



Front Panel Decal Printed On Adhesive Vinyl

Appendix C

Software Listings

The software developed by the author for the purposes of this dissertation is given in this Appendix. Support library functions supplied by Microchip for peripheral control have not been listed here but are accessible at www.microchip.com. The software has been split into the functional groups shown below and is listed alphabetically by filename for each group.

dsPIC30F6013 Embedded Software

Device Initialization and Control

Memory Control

Communications

Measurement Acquisition and Processing

```
/* ==  ADE7758.H & ADE7758.C ==========================================================================================
Interface routines for Analog Device ADE7758 Energy Measurement ICs.
==========================================================================================================================*/
#include "HardwareProfile.h"
// -- Data Type: tADE7758Calibration: Holds the calibration register values to configure and ADE7758. ----------------------
typedef union {
   struct __attribute__ ((packed)) {
      unsigned char PGAGain;
      S16 VRMSGain[3], IGain[3], WattGain[3], VARGain[3], VAGain[3];
      S16 VRMSOffset[3], IRMSOffset[3], WattOffset[3], VAROffset[3];
      char PhaseCalibration[3];
      unsigned char WattDivider, VARDivider, VADivider;
   } v;
   unsigned char b[61] __attribute__ ((packed));
} tADE7758Calibration;
// -- Data Type: tADE7758Measurement ----------------------------------------------------------------------------------------
// Holds the measurement data retrieved from the ADE7758. 24-bit voltage and current values are stored in sign-extended 32-bit variables for simpler processing.
typedef union {
   struct __attribute__ ((packed)) {
      S16 Watts[3], VAR[3], VA[3];
      S32 IRMS[3], VRMS[3];
      U16 Frequency;
      U8 Temperature;
   } v;
   unsigned char b[45] __attribute__ ((packed));
} tADE7758Measurement;

//==  ADE7758.C ==========================================================================================================
#include "ADE7758.h"
// == COMMUNICATIONS INTERFACE ============================================================================================
// -- ADE7758_SetChipSelect: Asserts or de-asserts the chip select line for the specified ADE7758. ----------------------
void ADE7758_SetChipSelect(U8 device, U8 state)
{
   switch (device)
   {
      case 1      : CS_EM1 = state; break;
      case 2      : CS_EM2 = state; break;
      case 3      : CS_EM3 = state; break;
      case 4      : CS_EM4 = state; break;
      case 255    : CS_EM1 = state; CS_EM2 = state; CS_EM3 = state; CS_EM4 = state; break;
   };
}
// -- ADE7758_Read: Reads multiple bytes from an Energy Measurement IC using the SPI port. ----------------------------------
void ADE7758_Read(U8 device, U8 reg, U8 length, U8 *buffer)
{
   unsigned char cntr1;
   ADE7758_SetChipSelect(device, 0);                                  // Select the device
   ADE7758_SPI_TxByte(reg & 0x7F);                                    // Specify the register to read from
   Time_Wait_us(); Time_Wait_us(); Time_Wait_us(); Time_Wait_us();     // Necessary delay for read access
   for (cntr1=0;cntr1<length;cntr1++) *buffer++ = ADE7758_SPI_TxByte(0x00);  // Read the data and place in buffer
   ADE7758_SetChipSelect(device, 1);                                  // De-select the device
}
// -- ADE7758_Write: Writes multiple bytes to the Energy Measurement IC using the SPI port. ----------------------------------
void ADE7758_Write(U8 device, U8 reg, U8 length, U8 *buffer)
{
   unsigned char cntr1;
   ADE7758_SetChipSelect(device, 0);                                  // Select the device
   ADE7758_SPI_TxByte(reg | 0x80);                                    // Specify a write to the register
   for (cntr1=0;cntr1<length;cntr1++) ADE7758_SPI_TxByte(*buffer++);  // Write the values from buffer to the ADE7758
   ADE7758_SetChipSelect(device, 1);                                  // De-select the device
}
// -- ADE7758_ReadChecksum: Reads the checksum value for the last communications transfer to/from the ADE7758. ------------
unsigned char ADE7758_ReadChecksum(U8 device)
{
   unsigned char result;
   ADE7758_Read(device, 0x7E, 1, &result);
   return result;
}
// == CONTROL AND CONFIGURATION ===========================================================================================
// -- ADE7758_ Check: Checks communications with an Energy Measurement IC by requesting the MMODE register and verifying the checksum. ----------
unsigned char ADE7758_Check(U8 device)
{
   unsigned char cntr, ones = 0, temp;
   ADE7758_Read(device, 0x14, 1, &temp);                              // Value should be 0xFC at reset
   for (cntr=0;cntr<8;cntr++)
   {
```

```
      temp <<= 1;
      if (SRbits.C == 1) ones++;                                    // Sum the number of 1s in the retrieved MMODE value
   }
   return (ADE7758_ReadChecksum(device) == ones);                   // Verify the retrieved checksum matches the calculated one
}
// -- ADE7758_Disable: Places the ADE7758 in a low-power state, terminating any measurements. ----------------------------------------------------------------------
void ADE7758_Disable(U8 device)
{
   unsigned char temp = 0x3C;
   ADE7758_Write(device, 0x13, 1, &temp);                           // Set the Operational Mode register (OPMODE) to disable the ADE7758
}
// -- ADE7758_SetControlRegisters: Writes the control register values from buffer to the ADE7758. ----------------------------------------------------------------
void ADE7758_SetControlRegisters(U8 device, U8 *buffer)
{
   unsigned char addr, i, idx;
   addr = 0x13;                                                     // OPMODE, MMODE, WAVMODE, COMPMODE, LCYCMODE registers
   for (i=0;i<5;i++) ADE7758_Write(device, addr++, 1, buffer++);
}
// -- ADE7758_GetCalibration: Retrieves the calibration parameters from the ADE7758 and places it in a tADE7758Calibration structure. ---------------------------
void ADE7758_GetCalibration(U8 device, tADE7758Calibration *result)
{
   unsigned char addr, i;
   unsigned char *ptr = &((*result).b[0]);
   addr = 0x23;
   ADE7758_Read(device, addr++, 1, ptr++);
   for (i=0;i<27;i++) { ADE7758_Read(device, addr++, 2, ptr); ptr += 2; }
   for (i=0;i<6;i++)  { ADE7758_Read(device, addr++, 1, ptr++); }
   return True;
}
// -- ADE7758_SetCalibration: Writes the calibration parameters from a tADE7758Calibration structure to the ADE7758. ----------------------------------------------
void ADE7758_SetCalibration(U8 device, tADE7758Calibration value)
{
   unsigned char addr, i, idx = 0;
   addr = 0x23;
   ADE7758_Write(device, addr++, 1, &value.b[idx++]);
   for (i=0;i<27;i++) { ADE7758_Write(device, addr++, 2, &value.b[idx]); idx += 2; }
   for (i=0;i<6;i++)  { ADE7758_Write(device, addr++, 1, &value.b[idx++]); }
   return True;
}
// == MEASUREMENT ACCESS ========================================================================================================================
// -- ADE7758_GetMeasurement: Retrieves the measurement values from the ADE7758 and places them in the tADE7758Measurement structure. ----------------------
void ADE7758_GetMeasurement(U8 device, tADE7758Measurement *result)
{
   unsigned char *buffptr, cntr, idx = 0, ptr, temp[3];
   ptr = 0x01;                                                      // This is the ADE7758 register pointer
   buffptr = &((*result).b[0]);                                     // Pointer to result structure buffer to make data access simpler
   // -- Read the Power values converting from MSB first to LSB first --
   for (cntr=0;cntr<9;cntr++)
   {
      ADE7758_Read(device, ptr++, 2, &temp[0]);                     // Read the 16-bit register value and reverse the order
      *buffptr++ = temp[1]; *buffptr++ = temp[0];
   }
   // -- Read the Current and Voltage RMS values converting from MSB first to LSB first --
   for (cntr=0;cntr<6;cntr++)
   {
      ADE7758_Read(device, ptr++, 3, &temp[0]);                     // Read the 24-bit register value and reverse the order
      *buffptr++ = temp[2]; *buffptr++ = temp[1]; *buffptr++ = temp[0];
      if ((temp[0] & 0x80) == 0x80)                                 // Values are stored in a 32-bit register so sign extension is necessary
         *buffptr++ = 0xFF;
      else
         *buffptr++ = 0x00;
   }
   // -- Read the Frequency value and convert from MSB first to LSB first --
   ADE7758_Read(device, ptr++, 2, &temp[0]);                        // Read the 16-bit register value and reverse the order
   *buffptr++ = temp[1]; *buffptr++ = temp[0];
   // -- Read the Temperature value --
   ADE7758_Read(device, ptr++, 1, buffptr);
}
// == INTERRUPT ROUTINES ========================================================================================================================
// -- INT1 : Energy Measurement device 1 (IC9) ----------------------------------------------------------------------------------------------------------------
void __attribute__((__interrupt__)) _INT1Interrupt(void) { IFS1bits.INT1IF = False; }
// -- INT2 : Energy Measurement device 2 (IC11) ---------------------------------------------------------------------------------------------------------------
void __attribute__((__interrupt__)) _INT2Interrupt(void) { IFS1bits.INT2IF = False; }
// -- INT3 : Energy Measurement device 3 (IC10) ---------------------------------------------------------------------------------------------------------------
void __attribute__((__interrupt__)) _INT3Interrupt(void) { IFS2bits.INT3IF = False; }
// -- INT4 : Energy Measurement device 4 (IC12) ---------------------------------------------------------------------------------------------------------------
```

```
void __attribute__((__interrupt__)) _INT4Interrupt(void) {  IFS2bits.INT4IF = False;  }
```

```
/*==  COMMANDS.H & COMMANDS.C  ================================================================================
Processes the commands issued by the Host to the Power Profiler.
============================================================================================================*/
#include "HardwareProfile.h"  #include "procComms.h"
// -- Externally Accessible Routines ----------------------------------------------------------------------------
extern void Cmd_Process(tCommsPacket *message);
#endif

// ==  COMMANDS.C  ============================================================================================
#include "Commands.h"  #include "ADE7758.h"  #include "I2CDevices.h"  #include "Timekeeping.h"  #include "procComms.h"  #include "procControl.h"
#include "procMeasurement.h"  #include "procMemory.h"
// == CONNECTION ==============================================================================================
// -- Cmd_Ping: Sends an acknowledge response to the Host to verify that communications is working. ------------------------------------------------------
void Cmd_Ping(tCommsPacket *packet)
{
   *(*packet).Params = 0x06;                                                    // Place ACK response in return buffer
   (*packet).MessLen = 1;                                                       // Specify one byte to return
   Comms_Respond(packet);                                                       // Send the response to the Host
}
// -- Cmd_Login: Verifies the ID and password supplied by the Host to allow access to data and config. commands. Incorrect login information is ignored. ----------
void Cmd_Login(tCommsPacket *packet)
{
   unsigned char i, *params, pwdOk; TMyU32 temp;
   params = (*packet).Params;                                                   // Pointer to the received parameters in the buffer
   for (i=0;i<4;i++) temp.Bytes[i] = *params++;                                 // Copy ID from buffer as it is sent before the password
   pwdOk = true;                                                                // Verify the password
   for (i=0;i<16;i++) if (*params++ != password[i]) { pwdOk = false; break; }   // If any byte of the password does not match return false
   if ((temp.Value == deviceID.Value) && (pwdOk))                               // Verify the Device ID
   {
      procComms.Connected = true;                                               // Set connection as active to allow access to commands
      *(*packet).Params = 0x06;                                                 // Place ACK response in return buffer
      (*packet).MessLen = 1;                                                    // Specify one byte to return
      Comms_Respond(packet);                                                    // Send the response to the Host
   }
}
// -- Cmd_Logoff: Disconnects the Host from the Profiler. ----------------------------------------------------------------------------------------------
void Cmd_Logoff(tCommsPacket *packet)
{
   procComms.Connected = false;                                                 // Set connection is inactive
   *(*packet).Params = 0x06;                                                    // Place ACK response in return buffer
   (*packet).MessLen = 1;                                                       // Specify one byte to return
   Comms_Respond(packet);                                                       // Send the response to the Host
}
// -- Cmd_Reset: Resets the processor upon instruction from the Host. Verifies the parameter matches the command before the reset. ----------------------------
void Cmd_Reset(tCommsPacket *packet)
{
   if (*(*packet).Params == 0x02)                                               // Verify the reset instruction is correct
   {
      *(*packet).Params = 0x06;                                                 // Place ACK response in return buffer
      (*packet).MessLen = 1;                                                    // Specify one byte to return
      Comms_Respond(packet);                                                    // Send the response to the Host
      asm volatile ("RESET");                                                   // Issue the assembler command to reset the processor
   }
}
// == DEVICE CONFIGURATION ====================================================================================
// -- Cmd_ReadConfiguration: Reads multiple blocks (32-bytes) of configuration EEPROM memory starting at address 'addr'. ----------------------------------
void Cmd_ReadConfiguration(tCommsPacket *packet)
{
   unsigned char i, blocks, *ptr; TMyU16 addr;
   addr.Bytes[0] = *((*packet).Params); addr.Bytes[1] = *((*packet).Params + 1);  // Starting address to read from
   ptr = (*packet).Response + 1;
   blocks = *((*packet).Params + 2);                                            // Number of 32-byte blocks to return
   for (i=0;i<blocks;i++)
   {
      intEE_ReadRow(addr.Value, ptr);                                           // Read from the EEPROM and place in the return buffer
      ptr += 32;                                                                // Increment the buffer pointer by 32-bytes
      addr.Value += 32;                                                         // Increment the EEPROM address by 32-bytes
   }
   *((*packet).Response) = 0x06;                                                // Acknowledge command successful
   (*packet).MessLen = (32 * blocks) + 1;                                       // Specify amount of data being returned
   Comms_Respond(packet);                                                       // Send the response to the Host
}
// -- Cmd_WriteConfiguration: Writes multiple blocks of Configuration data to EEPROM starting at address 'addr'. ---------------------------------------
```

```
void Cmd_WriteConfiguration(tCommsPacket *packet)
{
   unsigned char i, blocks, *ptr; TMyU16 addr;
   ptr = (*packet).Params;                                  // Pointer to the parameters from the Host
   addr.Bytes[0] = *ptr++; addr.Bytes[1] = *ptr++;          // Starting address to write to
   blocks = *ptr++;                                         // Number of blocks to write
   for (i=0;i<blocks;i++)
   {
      intEE_WriteRow(addr.Value, ptr);                     // Write to EEPROM from the buffer
      ptr += 32;                                           // Increment the parameter pointer by 32-bytes
      addr.Value += 32;                                    // Increment the EEPROM address by 32-bytes
   }
   *((*packet).Response) = 0x06;                            // Acknowledge command successful
   (*packet).MessLen = 1;                                   // Just the ACK byte to return
   Comms_Respond(packet);                                   // Send the response to the Host
}
// == STATUS / DEVICE INITIALIZATION ==========================================================================
// -- Cmd_ReadStatus: Returns the current Profiler status to the Host. ----------------------------------------
void Cmd_ReadStatus(tCommsPacket *packet)
{
   unsigned char *ptr, i, count = 0;
   ptr = (*packet).Response;
   *ptr++ = 0x06;                                           // ACK response
   count++;
   // -- Add clock value --
   for (i=0; i<7; i++) *ptr++ = currentTOD.b[i];            // Current time-of-day in byte array
   count += 7;
   // -- Add Memory pointers --
   for (i=0; i<70; i++) *ptr++ = dataFLASHPtr.b[i];
   count += 70;
   (*packet).MessLen = count;
   Comms_Respond(packet);
}
// -- Cmd_SetClock: Sets the internal clock and RTC to the specified time and date. ---------------------------
void Cmd_SetClock(tCommsPacket *packet)
{
   unsigned char *ptr, i; tDateTime value;
   ptr = (*packet).Params;
   for (i=0; i<7; i++) value.b[i] = *ptr++;                 // Copy the byte array into 'value' to set the clock
   if (Clock_SetTOD(value))
      (*packet).Response = 0x06;                            // Command was successful
   else
      (*packet).Response = 0x021;                           // Command was not successful
   (*packet).MessLen = 1;
   Comms_Respond(packet);
}
// == MEASUREMENT / CALIBRATION COMMANDS ======================================================================
// -- Cmd_GetCurrentMeasurements: Sends the current measurement values to the PC application. -----------------
void Cmd_GetCurrentMeasurements(tCommsPacket *packet)
{
   unsigned char i, *ptr;
   ptr = (*packet).Params;
   *ptr++ = 0x06;                                           // Command was successful
   for (i=0;i<195;i++) *ptr++ = currentMeasurements.Bytes[i]; // Copy the measurement data into the return array
   (*packet).MessLen = 196;
   Comms_Respond(packet);
}
// -- Cmd_SetVoltageChannels: Sets the voltage channel multiplexers to the specified values for calibration and testing. ----------
void Cmd_SetVoltageChannels(tCommsPacket *packet)
{
   unsigned char i, *ptr;
   ptr = (*packet).Params;
   VMux_SetChannels(ptr);                                   // Set the voltage multiplexers to the supplied values
   *ptr++ = 0x06;                                           // Acknowlege the command was successful
   (*packet).MessLen = 1;
   Comms_Respond(packet);
}
// -- Cmd_SetEnergyICMode: Sets the specified energy IC mode values for calibration or testing. ----------------
void Cmd_SetEnergyICMode(tCommsPacket *packet)
{
   unsigned char *resp, device, *params;
   resp = (*packet).Params;
   device = *((*packet).Params);                            // Specify the ADE7758 IC
   params = (*packet).Params + 1;                           // Pointer to the start of data
   ADE7758_SetControlRegisters(device, params);             // Set the ADE7758 registers from the rx packet
   *resp++ = 0x06;                                          // Acknowledge the command was processed
```

```
    (*packet).MessLen = 1;
    Comms_Respond(packet);
}
// -- Cmd_GetEngeryICCalibration: Retrieves the energy IC calibration parameters and sends them to the PC. ----------
void Cmd_GetEnergyICCalibration(tCommsPacket *packet)
{
    unsigned char *resp, device, *params, i; tADE7758Calibration value;
    resp = (*packet).Params;
    device = *((*packet).Params);                    // Specify which device to read from
    ADE7758_GetCalibration(device, &value);          // Retrieve the calibration values to the structure
    *resp++ = 0x06;                                  // Acknowledge the command was successful
    for (i=0;i<61;i++) *resp++ = value.b[i];         // Copy the calibration from the structure to the result buffer
    (*packet).MessLen = 62;                          // Return the values to the PC
    Comms_Respond(packet);
}
// -- Cmd_SetEnergyICCalibration: Sets the energy IC calibration values from the host PC. ----------
void Cmd_SetEnergyICCalibration(tCommsPacket *packet)
{
    unsigned char *resp, device, *params, i; tADE7758Calibration value;
    resp = (*packet).Params;
    device = *((*packet).Params);                    // Specify the device to set
    params = (*packet).Params + 1;
    for (i=0;i<61;i++) value.b[i] = *params++;       // Copy the parameters to the calibration structure
    ADE7758_SetCalibration(device, value);           // Set the calibration values from the structure
    *resp++ = 0x06;                                  // Acknowledge the command was successful
    (*packet).MessLen = 1;                           // return the ACK
    Comms_Respond(packet);
}
// -- Cmd_StartAcquisition: Initializes and starts data acquisition. ----------
void Cmd_StartAcquisition(tCommsPacket *packet)
{
    procMeas_InitializeCapture(false);               // Initialize capture
    procMeas_ResetAccumulation();                    // Reset the accumulation for each profile
    procMeas.DoAcquire = true;                       // Acquire and process measurements
    (*packet).Response = 0x06;                       // Acknowledge command success
    (*packet).MessLen = 1;                           // Return the ACK
    Comms_Respond(packet);
}
// -- Cmd_Stop Acquisition: Writes the current memory pointers to DataFlash and stops measurement acquisition. ----------
void Cmd_StopAcquisition(tCommsPacket *packet)
{
    StorMem_SetMemPointers();                        // Write memory pointers to DataFlash before stopping
    procMeas.DoAcquire = false;                      // Stop acquisition
    (*packet).Response = 0x06;                       // ACK that the command was performed
    (*packet).MessLen = 1;                           // Send ACK
    Comms_Respond(packet);
}
// -- Cmd_ResetAcquisition: Re-initializes measurement capture. ----------
void Cmd_ResetAcquisition(tCommsPacket *packet)
{
    procMeas_InitializeCapture(true);                // Initialize acquisition
    procMeas.DoAcquire = true;                       // Activate acquisition
    (*packet).Response = 0x06;                       // ACK command was successful
    (*packet).MessLen = 1;                           // Send the ACK
    Comms_Respond(packet);
}
// == DATA ACCESS ==========
// -- Cmd_ReadStorageParams: Returns to the Host the measurement size and storage memory pointer values for data uploading. ----------
void Cmd_ReadStorageParams(tCommsPacket *packet)
{
    unsigned char *ptr, i;
    ptr = (*packet).Response;
    *ptr++ = 0x06;                                   // ACK the command
    *ptr++ = (measurementSize & 0xFF);               // Measurement record size LSB
    *ptr++ = (measurementSize >> 8);                 // Measurement record size MSB
    for (i=0; i<70; i++) *ptr++ = dataFLASHPtr.b[i];  // Copy the current memory pointer values
    (*packet).MessLen = 73;                          // Return the values to the Host
    Comms_Respond(packet);
}
// -- Cmd_GetNextMeasurement: Retrieves the next profile measurement value from storage memory and sends to the Host. ----------
void Cmd_GetNextMeasurement(tCommsPacket *packet)
{
    unsigned char profile, *param, *resPtr; unsigned short length;
    param = (*packet).Params; resPtr = (*packet).Response;
    profile = *param++;                              // Which profile must the measurement be read from
    *resPtr++ = 0x06;                                // Add the ACK to the return values
```

```
   length = StorMem_ReadNextMeasurement(profile, resPtr);          // Read the value and place in the return buffer
   (*packet).MessLen = length + 1;                                 // Return the value to the Host
   Comms_Respond(packet);
}
// -- Cmd_ReadStorageMemory: Reads several bytes from the storage memory and returns them to the Host. ----------
void Cmd_ReadStorageMemory(tCommsPacket *packet)
{
   unsigned char device, *param, *resPtr; TMyU16 length; TMyU32 rdAddr;
   param = (*packet).Params; resPtr = (*packet).Response;
   device = *param++;                                              // Which DataFlash must be read from?
   rdAddr.Bytes[0] = *param++; rdAddr.Bytes[1] = *param++;          // 32-bit memory read location
   rdAddr.Bytes[2] = *param++; rdAddr.Bytes[3] = *param++;
   length.Bytes[0] = *param++; length.Bytes[1] = *param++;          // Amount of data to read
   *resPtr++ = 0x06;                                               // ACK result
   StorMem_ReadMemory(rdAddr.Value, length.Value, resPtr);         // Read the data and return to the Host
   (*packet).MessLen = length.Value + 1;
   Comms_Respond(packet);
}
// -- Cmd_UpdateReadPointers: Writes the current read pointer values to DataFlash to allow the read to continue at a later stage. ----------
void Cmd_UpdateReadPointers(tCommsPacket *packet)
{
   StorMem_UpdateReadPointers();                                   // Update the pointers in DataFlash memory
   *(*packet).Response = 0x06;                                     // ACK that the command was successful
   (*packet).MessLen = 1;                                          // Send the ACK
   Comms_Respond(packet);
}
// -- Cmd_CommitFLASHPointers: Writes the current memory pointer values to DataFlash and acknowledges it to the Host. ----------
void Cmd_CommitFLASHPointers(tCommsPacket *packet)
{
   StorMem_SetMemPointers();                                       // Store the pointers in DataFlash
   (*packet).Response = 0x06;                                      // ACK the command
   (*packet).MessLen = 1;                                          // Send the ACK to the Host
   Comms_Respond(packet);
}
// == COMMAND PROCESSING ==========================================================================================
void Cmd_Process(tCommsPacket *message)
{
   if (!procComms.Connected)                                       // If not connected then only basic commands can be processed
   {
      switch ((*message).Cmd) {
         case 0x01 : Cmd_Ping(message); break;
         case 0x03 : Cmd_Login(message); break;
      };
   }
   else
   {
      switch ((*message).Cmd) {
         case 0x01 : Cmd_Ping(message); break;
         case 0x02 : Cmd_Reset(message); break;
         case 0x03 : Cmd_Login(message); break;
         case 0x04 : Cmd_Logoff(message); break;
         // -- Device Configuration --
         case 0x10 : Cmd_ReadConfiguration(message); break;
         case 0x11 : Cmd_WriteConfiguration(message); break;
         // -- Calibration / Testing --
         case 0x20 : Cmd_GetCurrentMeasurements(message); break;
         case 0x21 : Cmd_SetVoltageChannels(message); break;
         case 0x22 : Cmd_SetEnergyICMode(message); break;
         case 0x23 : Cmd_GetEnergyICCalibration(message); break;
         case 0x24 : Cmd_SetEnergyICCalibration(message); break;
         // -- Status Checking and Device Initalization --
         case 0x30 : Cmd_ReadStatus(message); break;
         case 0x31 : Cmd_SetClock(message); break;
         case 0x32 : Cmd_ResetAcquisition(message); break;
         case 0x33 : Cmd_StartAcquisition(message); break;
         case 0x34 : Cmd_StopAcquisition(message); break;
         // -- Data uploading --
         case 0x40 : Cmd_ReadStorageParams(message); break;
         case 0x41 : Cmd_GetNextMeasurement(message); break;
         case 0x42 : Cmd_ReadStorageMemory(message); break;
         case 0x43 : Cmd_UpdateReadPointers(message); break;
         case 0x44 : Cmd_CommitFLASHPointers(message); break;
      }
   }
}
```

```
/*== COMMS_BLUETOOTH.H & COMMS_BLUETOOTH.C ==========================================================
Interface routines for the KC-Wirefree KC11 Bluetooth module.
====================================================================================================*/
#include "HardwareProfile.h"
// -- Constants and Type Definitions ------------------------------------------------------------------
#define BTBUFFERLEN                 100
// -- tBTStatus: Bluetooth processor control status and flags -----------------------------------------
typedef union {
   struct {
      unsigned Enabled              : 1;          unsigned Connected      : 1;
      unsigned PacketRx             : 1;          unsigned Unused         : 5;
   };
   unsigned char Value;
} tBTStatus;

// == COMMS_BLUETOOTH.C ============================================================================
#include "Comms_Bluetooth.h"  #include <uart.h>  #include "procComms.h"
// -- Global Variables --------------------------------------------------------------------------------
unsigned char btBuffer[BTBUFFERLEN], btLastByteRx = 0;
unsigned short btBufferPtr = 0, btPacketLength = 0;
tCommsPacket btPacket;
tBTStatus btStatus;
// -- BT_Command: Sends an AT command to the KC-11 bluetooth module and terminates it with a return and new-line. ----------
void BT_Command(char *s)
{
   while (*s != 0) BT_ByteTx(*s++);                                  // Send command character
   BT_ByteTx(13); BT_ByteTx(10);                                     // Send return and new line
}
// -- BT_Initialize: Initializes the serial interface. ------------------------------------------------
void BT_Initialize(unsigned char enabled)
{
   btStatus.Value = 0;
   btStatus.Enabled = enabled;
   RF_CTS_Dir = 1;                                                   // RTS and CTS are connected on the module so make CTS an input
   RF_RTS_Dir = 1;                                                   // RTS and CTS are connected on the module so make RTS an input
   U2MODE = 0;                                                       // 8-data bits, 1-stop bit, no parity
   U2STA  = 0x8400;                                                  // Interrupt on byte receive only
   U2BRG = ((FCY / 16) / 115200) - 1;                                // Set baud rate to 115200 bps
   U2MODEbits.UARTEN = 1;                                            // Enable UART for operation
   U2STAbits.UTXEN = 1;                                              // Enable transmit
   U2STAbits.URXISEL = 0;                                            // Enable interrupt on data receive
   IFS1bits.U2RXIF = false;                                          // Clear receive interrupt flags
   IPC6bits.U2RXIP = 7;                                              // Receive interrupt priority level 7
   IEC1bits.U2TXIE = false;                                          // Disable the transmit interrupt
   IEC1bits.U2RXIE = true;                                           // Enable the receive interrupt
   IPC6bits.U2RXIP = 7;                                              // Receive interrupt prioroty is 7
}
// -- BT_Processor: The main control routine for the Bluetooth module which is periodically called by the communications processor. ----------
void BT_Processor(void)
{
   if (!btStatus.Enabled) return;
}
// -- BT_CommsRx: When a byte is received it is procesed here to compile the received packet or to process the AT command result. ----------
void BT_CommsRx(unsigned char value)
{
   if (btBufferPtr >= BTBUFFERLEN) btBufferPtr = 0;                  // If the receive buffer is going to overflow, zero the pointer
   btBuffer[btBufferPtr++] = value;                                  // Add the received byte to the Bluetooth buffer
   if ((value == 0x0A) && (btLastByteRx == 0x0D))                    // Check for Command entry and exit responses
   {
      if ((btBufferPtr > 19) && (btBuffer[btBufferPtr - 20] == 'A') && (btBuffer[btBufferPtr - 3] == '-'))
      {
         btStatus.Connected = true;                                  // A data connection via SPP is active
         btPacketLength = BTBUFFERLEN;                               // This prevents a packet Rx being triggered after connection
         btBufferPtr = 0;                                            // Clear receive buffer pointer
      }
      if ((btBufferPtr > 14) && (btBuffer[btBufferPtr - 15] == '#') && (btBuffer[btBufferPtr - 3] == 'R'))
         btStatus.Connected = false;                                 // Interface is back in AT command mode
      if (!btStatus.Connected) btBufferPtr = 0;                      // Zero the buffer pointer
      LED_CommsRx = false;
   }
   btLastByteRx = value;
   if (!btStatus.Connected) return;                                  // If not in data mode return to caller
   if (Comms_RxByte(CMDSRC_BT, value))                               // Pass the received byte to the Communications Task for processing
      btBufferPtr = 0;                                               // Point back to first character since we are not in command mode
}
```

```
// -- BT_ByteTx: Transmits a byte to the Bluetooth module via UART 2. ----------
void BT_ByteTx(unsigned char value) { WriteUART2((unsigned int)value); while (BusyUART2()); }
// -- UART2 Rx Interrupt : Communications with RF module ----------
void __attribute__((__interrupt__)) _U2RXInterrupt(void) { BT_CommsRx(U2RXREG); IFS1bits.U2RXIF = 0; }
```

```
/*== COMMS_GSM.H & COMMS_GSM.C ==========================================================
GC864 GSM modem control code for TCP/IP communications.
==========================================================*/
#include "HardwareProfile.h"
// -- Definitions ----------
// -- User Pin Definitions for GSM Interface ----------
#define GSM_Reset_DIR       TRISDbits.TRISD10       #define GSM_Reset_PIN       LATDbits.LATD10
#define GSM_RTS_DIR         TRISCbits.TRISC2        #define GSM_RTS             LATCbits.LATC2
#define GSM_CTS_DIR         TRISBbits.TRISB2        #define GSM_CTS             PORTBbits.RB2
// -- Communications interface ----------
#define GSM_BUFFLEN         (200ul)                 #define ATCMDDELAY          5
#define APNNAMELENGTH       15
// -- Control status and flags ----------
typedef union {
   struct {
      unsigned Enabled      : 1;    unsigned Tick100ms    : 1;
      unsigned Tick1s       : 1;    unsigned DataMode     : 1;
      unsigned SIMError     : 1;    unsigned PINOk        : 1;
      unsigned DataReady    : 1;    unsigned Listening    : 1;
      unsigned PacketRx     : 1;    unsigned Unused       : 7;
   };
   unsigned short Value;
} tGSMStatus;
// -- AT command responses ----------
#define rNone               0       #define rATOK               1
#define rATERROR            2       #define rATConnected        3
#define rPINError           4
// -- PIN check responses ----------
#define pinrNone            0       #define pinrEnterPIN        1
#define pinrSIMError        2       #define pinrReady           3
#define pinrPINError        4

// == COMMS_GSM.C ========================================================================
#include "Comms_GSM.h"  #include <uart.h>  #include "procComms.h"
// -- Variable Definitions ----------
unsigned char gsmBuffer[GSM_BUFFLEN];
unsigned short gsmBuffPtr = 0, gsmCmdLen = 0, gsmSecCount = 5, gsmSecTick = 0, gsmSignalStrength, gsmListenPort;
char gsmPIN[5] = "0000", apnName[APNNAMELENGTH] = { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 };
tGSMStatus gsmStatus;

// == INITIALIZATION AND CONTROL =========================================================
// -- GSM_Initialize: Initializes the UART interface to the GC864 and verifies communication. ----------
void GSM_Initialize(unsigned char enabled)
{
   gsmStatus.Value = 0;                                     // Initialize the status flags
   gsmStatus.Enabled = enabled;
   // -- Initialize IO and UART --
   GSM_RTS_DIR = 0;                                         // Configure the request-to-send pin as an output
   GSM_CTS_DIR = 1;                                         // Configure the clear-to-send pin as an input
   GSM_RTS = 0;                                             // Default for RTS is low (ready to receive)
   GSM_Reset_DIR = 0;                                       // Reset pin is an output
   GSM_Reset_PIN = 0;                                       // Default pin state is off
   TRISFbits.TRISF3 = 0;                                    // UART port direction pins (Tx)
   TRISFbits.TRISF2 = 1;                                    // UART port direction pins (Rx)
   U1MODE = 0;                                              // 8-data bits, 1-stop bit, no parity
   U1STA  = 0x8400;                                         // Interrupt on byte receive only
   U1BRG = ((FCY / 16) / 9600) - 1;                         // Default GC864 baud rate is 9600bps
   U1MODEbits.UARTEN = 1;                                   // UART module enabled
   U1STAbits.UTXEN = 1;                                     // UART transmit enabled
   U1STAbits.URXISEL = 0;                                   // UART interrupt on byte receive
   IFS0bits.U1RXIF = false;                                 // Clear receive interrupt flags
   IPC2bits.U1RXIP = 7;                                     // Receive interrupt priority level 7
   // -- If the GSM module is not enabled then return here before setting the interrupt --
   If (!gsmStatus.Enabled) return;
   IEC0bits.U1RXIE = true;                                  // Enable receive interrupt
   GSM_TogglePower();                                       // Power the modem and verify communications
   if (GSM_SendATCmd("AT", 3, 0) == rATOK)                  // Verify communications with the modem
   {
      // -- Set Baud rate to 57600 bps --
      if (GSM_SendATCmd("AT+IPR=57600", 3, ATCMDDELAY) == rATOK)
```

```
            U1BRG = ((FCY / 16) / 57600) - 1;
        if (GSM_SendATCmd("AT", 3, ATCMDDELAY) == rATOK)                    // Verify communications at new baud rate
        {
            gsmStatus.Enabled = true;
            // -- Configure basic parameters for device control --
            GSM_SendATCmd("AT#DSTO=1", 3, ATCMDDELAY);                      // Data sending timeout set to 100ms
            GSM_SendATCmd("AT#SKTTO=0", 3, ATCMDDELAY);                     // No timeout on socket activity
        }
    }
}
// -- GSM_Processor: This routine acts as the state-machine managing the modem. ----------
void GSM_Processor(void)
{
    if (!gsmStatus.Enabled) return;                                         // If the GSM is disabled then return to the main program
    if (gsmStatus.Tick100ms) {  gsmStatus.Tick100ms = false;  }             // 100ms Tick timed events
    if (gsmStatus.Tick1s)                                                   // One second tick timed events
    {
        gsmSecTick++;
        gsmStatus.Tick1s = false;
    }
    // -- If the modem is not in data mode check its state every 'gsmSecCount' seconds. The GSM_CheckConnection routine will control the state of the modem. --
    if ((gsmStatus.DataMode == false) && (gsmSecTick > gsmSecCount))
    {
        GSM_CheckConnection();                                              // Verify modem operation
        if ((gsmStatus.DataReady) && (!gsmStatus.Listening))
            gsmStatus.Listening = GSM_Listen();                             // Activate the listening socket if the GPRS contect is active
        gsmSecTick = 0;
    }
}
// -- GSM_CheckSignal ----------
// Returns the signal strength for the GSM connection: 0 - -113dBm or less, 1 - -111dbm,  2..30 - -109dbm..-53dbm in 2dBm steps, 31 - -51dBm, 99 - Unknown.
unsigned char GSM_CheckSignal(void)
{
    unsigned char str[3];
    if (GSM_SendATCmd("AT+CSQ", 5, ATCMDDELAY) == rATOK)                    // Command to request signal strength
    {
        str[0] = gsmBuffer[16]; str[1] = gsmBuffer[17];
        if (str[1] == ',') str[1] = 0; else str[2] = 0;
        return atoi(str);                                                   // Convert CSQ result into a byte value
    }
    else
        return 99;                                                          // 99 signifies measurement error
}
// -- GSM_CheckPIN: Query PIN state and enter PIN if 'enterPIN' is true. ----------
unsigned char GSM_CheckPIN(unsigned char enterPIN)
{
    unsigned char atResp, result;
    char cmd[15] = "AT+CPIN=";
    result = pinrNone;
    if (GSM_SendATCmd("AT+CPIN?", 2, 1) == rATOK)                           // Request PIN state from the modem
    {
        if (gsmBuffer[24] == 'I')                                           // PIN code needs to be entered
        {
            result = pinrEnterPIN;
            if (enterPIN)
            {
                if (gsmPIN[0] == 0) return pinrPINError;                    // The PIN code has not been set in the Profiler so report an error
                strcat(cmd, gsmPIN);                                        // Copy the PIN code from gsmPIN to cmd
                if (GSM_SendATCmd(cmd, 100, 0) == rATOK)                    // Issue the PIN to the module
                    return pinrReady;                                       // If the PIN was accepted then return that the module is Ready
                else
                    return pinrSIMError;                                    // PIN was rejected so an error is returned
            }
        }
        if (gsmBuffer[19] == 'R') result = pinrReady;                       // PIN code is already entered or is disabled (not required)
    }
    else
    {
        GSM_SendATCmd("AT+CMEE=1", 2, ATCMDDELAY);                          // Disable verbose message mode to determine reason PIN request failed
        if (GSM_SendATCmd("AT+CPIN?", 2, 1) == rATOK)                       // Re-issue PIN request
            if (gsmBuffer[gsmBuffPtr - 4] == '1') result = pinrSIMError;    // Report a problem with the SIM card
        GSM_SendATCmd("AT+CMEE=0", 2, 1);                                   // Enable verbose message mode
    }
    return result;
}
// -- GSM_CheckConnection: Checks the modem state and activates the GPRS connection if ready. ----------
```

```
void GSM_CheckConnection(void)
{
   if (gsmStatus.DataMode) return;                                  // If the modem is processing data we cannot use AT commands
   if (!gsmStatus.PINOk)                                            // Check the PIN code if it hasn't been entered
   {
      switch (GSM_CheckPIN(true))
      {
         case pinrNone: break;                                      // Modem not powered?
         case pinrSIMError: gsmStatus.SIMError = true; break;       // Problem with SIM card
         case pinrReady: gsmStatus.PINOk = true; break;             // PIN Ready
         case pinrPINError: gsmStatus.SIMError = true; break;       // PIN Problem
      }
   }
   gsmSignalStrength = GSM_CheckSignal();                           // Check signal strength
   if (gsmStatus.PINOk == false) return;                            // If there is a PIN problem the GPRS context cannot be activated
   if (GSM_SendATCmd("AT#GPRS?", 2, 1) == rATOK)                    // Check the GPRS context is active
   {
      if (gsmBuffer[19] == '1')                                     // GPRS context is active
         gsmStatus.DataReady = true;
      else
         GSM_ActivateGPRS();                                        // Activate the GPRS context
   }
}
// == PACKET DATA ACCESS ========================================================================================================
// -- GSM_ActivateGPRS: Initializes the GPRS context. ---------------------------------------------------------------------------------
unsigned char GSM_ActivateGPRS(void)
{
   char cmd[35 + APNNAMELENGTH]; int charCnt;
   GSM_SendATCmd("AT#GPRS=0", 10, 5);                               // Close the current GPRS connection if it is already open
   // -- Construct the command to initialize the APN --
   charCnt = snprintf(&cmd[0], 35 + APNNAMELENGTH, "AT+CGDCONT=1,\"IP\",\"%s\",\"0.0.0.0\",0,0", &apnName[0]);
   cmd[charCnt] = 0;
   GSM_SendATCmd(cmd, 10, 5);                                       // Send the command to configure the APN
   GSM_SendATCmd("AT#FRWL=1,\"1.1.1.1\",\"0.0.0.0\"", 10, 5);       // Initialize the firewall to allow all received packets through
   gsmStatus.DataReady = (GSM_SendATCmd("AT#GPRS=1", 50, 10) == rATOK);  // Activate the GPRS connection
   return gsmStatus.DataReady;
}
// -- GSM_Listen: Opens a listening socket on port 'listenPort'. ----------------------------------------------------------------------
unsigned char GSM_Listen(void)
{
   int charCnt; char cmd[20];                                       // 15 command characters plus 5 for the port number
   if (GSM_SendATCmd("AT#SKTL?", 1, ATCMDDELAY) == rATOK)           // Check listening state and if active, return to main program
   {
      if (gsmBuffer[19] == '1') return true;                        // '1' - Already listening
      charCnt = snprintf(&cmd[0], 20, "AT#SKTL=1,0,%d,0", gsmListenPort);
      cmd[charCnt] = 0;
      return (GSM_SendATCmd(cmd, 50, 10) == rATOK);                 // Execute the command to activate the listening socket
   }
   return false;
}
// == HARDWARE-LEVEL COMMUNICATIONS =============================================================================================
// -- GSM_SendATCmd -------------------------------------------------------------------------------------------------------------------
// Sends an AT command to the modem and waits for a response. *cmd - pointer to string to send, timeout - number of 100ms ticks to wait for response,
// postDelay - number of 100ms ticks to wait after command is completed.
unsigned char GSM_SendATCmd(const char *cmd, unsigned char timeout, unsigned char postDelay)
{
   unsigned char result = rNone; unsigned short startTick;
   gsmBuffPtr = 0; gsmCmdLen = 0;
   while (*cmd != 0)
   {
      GSM_ByteTx(*cmd++);                                           // Transmit each byte of the cmd to the modem
      gsmCmdLen++;                                                  // gsmCmdLen is used to determine where the AT result starts
   }
   GSM_ByteTx(13); GSM_ByteTx(10);                                  // AT command terminator
   gsmCmdLen += 2;
   LED_CommsTx = false;
   // -- Wait for result from modem --
   result = rNone;
   startTick = timerTick100ms;
   while ((result == rNone) && (Time_Elapsed(startTick) < timeout))
   {
      if ((gsmBuffPtr > gsmCmdLen) && (gsmBuffer[gsmBuffPtr - 1] == 10))
      {
         if ((gsmBuffPtr > (gsmCmdLen + 3)) && (gsmBuffer[gsmBuffPtr - 3] == 'K'))
            result = rATOK;
         else
```

```
            if ((gsmBuffPtr > (gsmCmdLen + 6)) && (gsmBuffer[gsmBuffPtr - 4] == 'O'))
               result = rATERROR;
            else
               if ((gsmBuffPtr > (gsmCmdLen + 8)) && (gsmBuffer[gsmBuffPtr - 4] == 'C'))
                result = rATConnected;
      }
      TimeCriticalCode();                                    // Allows other routines to execute while waiting for a response
   }
   if (postDelay > 0)                                        // Delay after command to prevent back-on-back commands
   {
      startTick = timerTick100ms;
      while (Time_Elapsed(startTick) < postDelay) TimeCriticalCode();
   }
   gsmBuffPtr = 0;
   return result;
}
// -- GSM_ByteRx ------------------------------------------------------------------------------------------------------------------------------------------------------
// Called by the UART receive interrupt service routine when a byte of data is received from the modem. If data is received it is passed to the Comms Task.
void GSM_ByteRx(unsigned char value)
{
   gsmBuffer[gsmBuffPtr++] = value;                          // Add the received byte the to gsmBuffer
   if (gsmStatus.DataMode)                                   // In data mode the received data is passed to the Comms Processor
   {
      if (Comms_RxByte(CMDSRC_GSM, value))                   // Pass the byte to the Communications Processor
         gsmBuffPtr = 0;                                     // Clear the gsmBuffer pointer
      // -- If the client disconnects "NO CARRIER" is received --
      if ((gsmBuffPtr == 14) && (value == 10) && (gsmBuffer[gsmBuffPtr - 4] == 'E'))
      {
         gsmStatus.DataMode = false;                         // Modem is back in command mode
         gsmStatus.Listening = false;                        // Modem needs to be put back into listening mode
      }
   }
   else
   {
      // -- Check for a connection request --
      if ((gsmBuffPtr > 33) && (value == 10) && (gsmBuffer[gsmBuffPtr - 5] == 'E'))
      {
         gsmStatus.DataMode = true;                          // Modem is in data mode
         gsmBuffPtr = 0;                                     // Reset the gsmBuffer pointer
      }
   }
}
// -- GSM_ByteTx: Transmits a character to the GSM module via UART 1. -------------------------------------------------------------------------------------------
void GSM_SendChar(unsigned char value)
{
   WriteUART1((unsigned int)value);                          // Transmit the byte
   while (BusyUART1());}                                     // Wait until the buffer is empty before returning
}
// -- Interrupt: UART Transmit: Not used, clears the interrupt flag. --------------------------------------------------------------------------------------------
void __attribute__((__interrupt__)) _U1TXInterrupt(void) { IFS0bits.U1TXIF = 0; }
// -- Interrupt: UART Receive: Send each byte received to GSM_ByteRx. -------------------------------------------------------------------------------------------
void __attribute__((__interrupt__)) _U1RXInterrupt(void)
{
   while (DataRdyUART1()) GSM_ByteRx(ReadUART1());
   IFS0bits.U1RXIF = 0;
}
// == HARDWARE CONTROL ============================================================================================================================================
// -- GSM_TooglePower: Sets the GSM power (active low) for 1.2secs and then waits 0.5secs before returning to allow modem start. ----------------------------------
void GSM_TogglePower(void)
{
   unsigned char startTick;
   SelReg_Set(64);                                           // Activate pull-down transistor on GSM interface card
   startTick = timerTick100ms;
   while (Time_Elapsed(startTick) < 12) TimeCriticalCode();  // Wait 1.2 secs and process other Profiler functions
   SelReg_Clear(64);                                         // Deactivate pull-down transistor on GSM interface card
   startTick = timerTick100ms;
   while (Time_Elapsed(startTick) < 5) TimeCriticalCode();   // Wait 0.5 secs and process other Profiler functions
}
// -- GSM_Reset: Sets the GSM reset (active low) for 0.2secs and then waits 0.2secs before returning to allow modem reset. ----------------------------------------
void GSM_Reset(void)
{
   unsigned char startTick;
   GSM_Reset_PIN = 1;                                        // Activate pull-down transistor on GSM interface card
   startTick = timerTick100ms;
   while (Time_Elapsed(startTick) < 2) TimeCriticalCode();   // Wait 0.2 secs and process other Profiler functions
   GSM_Reset_PIN = 0;                                        // Deactivate pull-down transistor on GSM interface card
```

```
    startTick = timerTick100ms;
    while (Time_Elapsed(startTick) < 2) TimeCriticalCode();          // Wait 0.2 secs and process other Profiler functions
}
```

```
/*==  COMMS_IPC.C & COMMS_IPC.C  ================================================================================================
Inter-processor communications implementation between the dsPIC and PIC18F processors.
================================================================================================================================*/
#include "HardwareProfile.h"  #include "Timekeeping.h"  #include "procComms.h"
// -- Definitions and Data Types ---------------------------------------------------------------------------------------------------
#define IPC_BUFF_SIZE                   200
// -- IPC Interface Commands -------------------------------------------------------------------------------------------------------
#define IPCCMD_PING                     0x01        #define IPCCMD_MEASUREMENTS         0x10
#define IPCCMD_GETPIC18FSTATUS          0x11        #define IPCCMD_GETUSBPACKET         0x30
#define IPCCMD_SENDUSBPACKET            0x31
// -- tipcStatus: IPC Communications task state machine and flags. -----------------------------------------------------------------
typedef union {
    struct {
        unsigned msec10Tick             : 1;        unsigned Tick100ms          : 1;
        unsigned hsPinState             : 1;        unsigned commsError         : 1;
        unsigned SlaveReady             : 1;        unsigned Busy               : 1;
        unsigned UpdateMeas             : 1;        unsigned USBMessageRx       : 1;
        unsigned UpdateStatus           : 1;        unsigned                    : 9;
    };
    unsigned short value;
} tipcStatus;
// -- tPIC18F_IPCStatus: Reflects PIC18F processor status returned through an SPI read. --------------------------------------------
typedef union {
    struct {
        unsigned ACK                    : 1;        unsigned Fault              : 1;
        unsigned MsgWaiting             : 1;        unsigned USBMsgWaiting      : 1;
        unsigned Response               : 1;        unsigned Verifier           : 3;
    };
    unsigned char value;
} tPIC18F_IPCStatus;
// -- tIPCStatusMessage: Data structure representing the measurement information to be displayed on the LCD. The Flags registers specify the IO states. ----------
typedef union {
    struct {
        unsigned char Flags[2];
        tDateTime Clock;
        float VRMS[3], IRMS[12];
        signed char Temp;
        float Freq;
    };
    unsigned char Bytes[75];
} tIPCStatusMessage;

// ==  COMMS_IPC.C  ================================================================================================================
#include "Comms_IPC.h"  #include "procMeasurement.h"  #include "procComms.h"  #include "procControl.h"
// -- Global Variables -------------------------------------------------------------------------------------------------------------
unsigned char ipcBuffer[IPC_BUFF_SIZE];
unsigned char ipcBuffRxIdx = 0, ipcChkSumA, ipcChkSumB, ipcTimer_10ms = 0, ipcMeasTick = 0;
tipcStatus ipcStatus; tPIC18F_IPCStatus pic18fIPCStatus;
// -- IPC_Initialize: Initialize the port pins and registers for inter-processor communications. ----------------------------------
void IPC_Initialize(void)
{
    IPC_CS_Dir = 0;                                 // Set IPC chip select as an output
    IPC_CS_Pin = 1;                                 // Disable the IPC chip select
    IPC_HS_Dir = 1;                                 // Set the IPC hand-shaking signal as an input
    ipcStatus.value = 0;                            // Initialize the IPC Task status register
    ipcBuffRxIdx = 0;                               // Initailize the IPC receive buffer index
    pic18fIPCStatus.value = 0;                      // Initialize the PIC18F IPC status variabale
}
// -- IPC_WaitForResponse: Waits for a toggle of the handshaking signal to show that the PIC18 processor is ready for the next byte. ----------
bool IPC_WaitForResponse(unsigned char delay)
{
    if (ipcStatus.commsError) return false;         // Do not continue processing if an SPI error has been reported
    // -- Delay until the handshake is received or the process times out --
    ipcTimer_10ms = 0;
    while ((ipcStatus.hsPinState == IPC_HS_Pin) && (ipcTimer_10ms < delay ))
        TimeCriticalCode();                         // Execute other critical tasks while waiting
    ipcStatus.commsError = (ipcTimer_10ms >= delay);// Did we timeout?
    ipcStatus.hsPinState = IPC_HS_Pin;              // Copy the level of the hand-shake signal
    return ipcStatus.commsError;                    // Return the error state
}
```

```
// -- IPC_TxByte ------------------------------------------------------------------------------------------------------------------------------------------------------
// Transmit a byte to the PIC18F processor via the SPI port using the handshaking signal. Calculates the checksum for transmission at the end of the packet.
unsigned char IPC_TxByte(unsigned char value)
{
   unsigned char temp;
   ipcChkSumA += value; ipcChkSumB += ipcChkSumA;                    // Checksum calculation to verify data integrity
   temp = SPI2_TxByte(value);                                        // Transmit the value and return the received byte
   IPC_WaitForResponse(5);                                           // Wait up to 50ms for the handshaking to indicate the PIC18F is ready
   return temp;                                                      // Return the received SPI value during the transmit
}
// -- IPC_Message: Creates a message to send to the PIC18F processor and waits for a response of 'respLength' bytes. ----------------------------------------------
bool IPC_Message(unsigned char cmd, unsigned char *params, unsigned char length, unsigned char *respLength)
{
   bool success = false; U8 i, bytesRx = 0, chkSumA, chkSumB; tPIC18F_IPCStatus pic18fStatus;
   ipcStatus.Busy = true;
   ipcChkSumA = 0; ipcChkSumB = 0;                                   // Clear checksum which is added to in TxByte
   ipcStatus.commsError = false;                                     // Clear the error marker bit
   ipcStatus.hsPinState = IPC_HS_Pin;                                // Determine ready-state of handshaking signal
   IPC_CS_Pin = 0;                                                   // Select the PIC18F processor
   IPC_WaitForResponse(5);
   IPC_TxByte(cmd);                                                  // Send the command
   IPC_TxByte(length + 4);                                           // Packet is always <= 255 chars
   for (i=0;i<length;i++) IPC_TxByte(*params++);                     // Send the data to be transmitted
   if (ipcStatus.commsError)                                         // Disable communications and reset the interface if an error has occurred
   {
      ipcStatus.Busy = false;
      IPC_CS_Pin = 1;                                                // De-select the PIC18F processor
      return false;                                                  // Quit the message reporting an error
   }
   chkSumA = ipcChkSumA; chkSumB = ipcChkSumB;                       // Copy the checksum values because IPC_TxByte will change it
   IPC_TxByte(chkSumA); IPC_TxByte(chkSumB);                         // Transmit the checksum value
   IPC_WaitForResponse(5);                                           // Wait for toggle to indicate command processing
   pic18fStatus.value = IPC_TxByte(0x00);                            // Read and update the ipcStatus register
   if ((pic18fStatus.Verifier == 5) && (pic18fStatus.ACK == 1))      // Verify an ACK was received and then read the data
   {
      if (pic18fStatus.Response)                                     // Has the PIC18F processor indicated there is data to be received?
         {
         chkSumA = 0; chkSumB = 0;                                   // Clear the checksum to verify the received data
         bytesRx = IPC_TxByte(0x00);                                 // The first byte received is the Rx data count
         chkSumA += bytesRx;                                         // Add the received byte to the Checksum calculation
         chkSumB += chkSumA;
         for (i=0;i<bytesRx;i++)
         {
            ipcBuffer[i] = IPC_TxByte(0x00);                         // Copy the data from the PIC18F processor to ipcBuffer
            chkSumA += ipcBuffer[i]; chkSumB += chkSumA;             // Add the received byte to the Checksum calculation

         }
         // -- Read and verify the checksum values --
         success = (chkSumA == IPC_TxByte(0x00)) && (chkSumB == IPC_TxByte(0x00));
      }
      else
         success = true;                                             // No return data was expected so the ACK indicates success
   }
   ipcStatus.Busy = false;                                           // Indicate the end of data transfer and disable the chip select
   IPC_CS_Pin = 1;                                                   // De-select the Chip Select ending the message
   *respLength = bytesRx;                                            // Indicate the number of bytes received to the calling method
   return success;
}
// -- IPC_GetIPCStatus: Reads the current IPC status register from the PIC18F processor which is used to indicate the ACK respons and general status. ----------
void IPC_GetIPCStatus(void)
{
   tPIC18F_IPCStatus ipcTemp;                                        // Temporary structure to hold the IPC register values
   IPC_CS_Pin = 0;                                                   // Select the PIC18F processor
   ipcStatus.commsError = false;                                     // Clear communications errors
   IPC_TxByte(0x00);                                                 // Transmit the first byte to 'flush' the interface
   ipcTemp.value = IPC_TxByte(0x00);                                 // Second transmitted byte returns the IPC register value
   IPC_CS_Pin = 1;                                                   // De-select the PIC18F processor
   if (ipcTemp.Verifier == 5)                                        // Verify the returned byte is an IPC status
   {
      pic18fIPCStatus.value = ipcTemp.value;                         // Copy to the main pic18fIPCStatus variable
      if (pic18fIPCStatus.MsgWaiting) ipcStatus.UpdateStatus = true; // Is the PIC18F processor wanting to send a status update?
   }
}
// -- IPC_GetPIC18FStatus: Reads the status and supply voltage information from the PIC18F processor and passes it to the Control Task. ----------------------------
bool IPC_GetPIC18FStatus(void)
{
```

```
    unsigned char byteCnt;
    if (IPC_Message(IPCCMD_GETPIC18FSTATUS, NULL, 0, &byteCnt))
        return ProcCtrl_PIC18FStatusUpdate(&ipcBuffer[0], byteCnt);
    else
        return false;
}
// -- IPC_GetUSBPacket: Reads the USB packet from the PIC18F processor. Each byte is then passed to the Communications Task for processing. ----------
void IPC_GetUSBPacket(void)
{
    unsigned char byteCnt, i;
    if (IPC_Message(IPCCMD_GETUSBPACKET, NULL, 0, &byteCnt))
        for (i=0;i<byteCnt;i++) Comms_RxByte(CMDSRC_USB, ipcBuffer[i]);
}
// -- IPC_UpdateMeasurements: Sends the Profiler status and RMS voltage and current values to the PIC18F processor for display on the LCD. ----------
bool IPC_UpdateMeasurements(void)
{
    int i;
    tIPCStatusMessage statusMsg;
    if (!ipcStatus.Busy)
    {
        // -- Compile status information --
        statusMsg.Flags[0] = procControl.Bytes[2]; statusMsg.Flags[1] = procControl.Bytes[3];
        statusMsg.Clock = currentTOD;
        // -- Set VRMS measurements --
        for (i=0;i<3;i++) statusMsg.VRMS[i] = currentMeasurements.Voltage[voltageChannel[i]] * dispVRMS[i];
        // -- Set IRMS measurements --
        for (i=0;i<12;i++) statusMsg.IRMS[i] = currentMeasurements.Current[i] * dispIRMS[i];
        statusMsg.Temp = currentMeasurements.Temperature;
        statusMsg.Freq = currentMeasurements.Frequency[freqChannel] * dispFreq;
        return IPC_Message(IPCCMD_MEASUREMENTS, &statusMsg.Bytes[0], 75, 0);
    }
    return false;
}
// -- IPC_Processor: Performs the IPC software tasks. ----------
void IPC_Processor(void)
{
    // -- Functions processed every 100ms --
    if (ipcStatus.Tick100ms)
    {
        // -- Verify comms is working by pinging the slave --
        if (!ipcStatus.SlaveReady) ipcStatus.SlaveReady = IPC_Message(IPCCMD_PING, &ipcBuffer[0], 0, 0);
        // -- Every 800ms send a measurement update to the PIC18F processor --
        if (ipcMeasTick++ > 7)
        {
            ipcStatus.UpdateMeas = true;
            ipcMeasTick = 0;
        }
        ipcStatus.Tick100ms = false;
    }
    // -- Skip the following if the PIC18F processor is not ready yet --
    if (!ipcStatus.SlaveReady) return;
    // -- If the PIC18F processor has a USB packet to read, then get it --
    if (pic18fIPCStatus.USBMsgWaiting) IPC_GetUSBPacket();                  // Read the USB packet from the PIC18F processor
    // -- PIC18F processor is signalling that there is a message to be read --
    if ((IPC_CS_Pin == 1) && (IPC_HS_Pin == 1)) IPC_GetIPCStatus();         // Read the IPC status using an SSPBUF read
    // -- Update the measurements on the PIC18F processor --
    if (ipcStatus.UpdateMeas) ipcStatus.UpdateMeas = !IPC_UpdateMeasurements();  // Clear the measurement update flag when successful
    // -- Request the PIC18F status --
    if (ipcStatus.UpdateStatus) ipcStatus.UpdateStatus = !IPC_GetPIC18FStatus();  // Clear the status request flag when successful
}
```

```
/*==  COMMS_MAX3100.H & COMMS_MAX3100.C  ============================================================
Implements the SPI interface to the Maxim MAX3100 SPI/UART transceiver.
==========================================================================================================*/
#include "HardwareProfile.h"
// -- Constant and Type Definitions ----------
#define MAX3100_Shutdown        0x1000

// ==  COMMS_MAX3100.C  ==========================================================================
#include "Comms_MAX3100.h"
// -- MAX3100_Intialize: Writes the configuration value to the MAX3100 and verifies communications. ----------
unsigned char MAX3100_Initialize(unsigned short cfgValue)
{
    TMyU16 temp, resp;
    MAX3100_CS = 0;                                                         // Enable the MAX3100 for SPI communications
```

```
   cfgValue |= 0xC000;                                  // Specify a write
   temp.Value = cfgValue;
   resp.Bytes[1] = MAX3100_SPI_TxByte(temp.Bytes[1]);   // Send the MSB to the MAX3100
   resp.Bytes[0] = MAX3100_SPI_TxByte(temp.Bytes[0]);   // Send the LSB to the MAX3100
   MAX3100_CS = 1;                                      // Disable the MAX3100 SPI
   return (resp.Value == 0x4000);                       // Verifies communication with the MAX3100
}
// -- MAX3100_ReadConfig: Returns the value of the configuration register. ----------
unsigned short MAX3100_ReadConfig(void)
{
   TMyU16 resp;
   MAX3100_CS = 0;                                      // Enable the MAX3100 for SPI communications
   resp.Bytes[1] = MAX3100_SPI_TxByte(0x40);            // Specify a read of the configuration register
   resp.Bytes[0] = MAX3100_SPI_TxByte(0x00);
   MAX3100_CS = 1;                                      // Disable the MAX3100 SPI
   return resp.Value;                                   // Return the configuration word
}
// -- MAX3100_RTS: Sets the Request To Send signal output to 'value'. ----------
void MAX3100_RTS(unsigned char value)
{
   MAX3100_CS = 0;                                      // Select the MAX3100 SPI
   MAX3100_SPI_TxByte(0x84 + value);                    // Set the RTS state without tramsitting data
   MAX3100_SPI_TxByte(0x00);                            // LSB of RTS state word is a zero
   MAX3100_CS = 1;                                      // De-select the MAX3100
}
// -- MAX3100_Rx: Reads a byte from the receive buffer of the MAX3100. ----------
unsigned char MAX3100_Rx(void)
{
   TMyU16 resp;
   MAX3100_CS = 0;                                      // Select the MAX3100
   resp.Bytes[1] = MAX3100_SPI_TxByte(0x00);            // Specify a read of the receive buffer
   resp.Bytes[0] = MAX3100_SPI_TxByte(0x00);            // Dummy value to allow for SPI reception into 'resp'
   MAX3100_CS = 1;                                      // De-select the MAX3100
   return resp.Bytes[0];                                // Return the receive data byte
}
// -- MAX3100_Tx: Transmits 'value' over the UART. ----------
void MAX3100_Tx(unsigned char value)
{
   MAX3100_CS = 0;                                      // Select the MAX3100
   MAX3100_SPI_TxByte(0x82);                            // Specify a write to the transmission register
   MAX3100_SPI_TxByte(value);                           // Write the value out the serial port
   MAX3100_CS = 1;                                      // De-select the MAX3100
}
```

```
/*==  CONFIGBLOCKDEFNS.H ==========================================================
Defines data structures used when accessing the EEPROM configuration memory.
==============================================================================*/
// -- Definitions ----------
#define CFG_DEVICEID_ADDR          0       #define CFG_MEASCFG_ADDR           32
#define CFG_MEMORYCFG_ADDR         96      #define CFG_CHANNELCALIB_ADDR      128
#define CFG_ENERGYIC1REG_ADDR      352     #define CFG_ENERGYIC2REG_ADDR      416
#define CFG_ENERGYIC3REG_ADDR      480     #define CFG_ENERGYIC4REG_ADDR      544
#define CFG_1PH2WCALIB_ADDR        608     #define CFG_3PH3WCALIB_ADDR        832
#define CFG_3PH4WCALIB_ADDR        1056    #define CFG_CUSTOMCALIB_ADDR       1280
#define CFG_COMMS_ADDR             1504
// -- tcfgProfilerID ----------
typedef union {
   struct __attribute__ ((packed)) {
      unsigned short Verifier;
      unsigned long DeviceID;
      char Password[16];
      unsigned char ManufacturerID, TODManufactured[4];
      unsigned char Unused[5];
   } v;
   unsigned char b[32] __attribute__ ((packed));
} tcfgProfilerID;
// -- tcfgMeasurement ----------
typedef union {
   struct __attribute__ ((packed)) {
      unsigned short Verifier;
      unsigned long ConfigFlags;
      unsigned char CustomVoltageMux[6];
      unsigned char SecsProfileFlags, SecsProfileInterval, MinsProfileFlags, MinsProfileInterval, HrsProfileFlags, HrsProfileInterval;
      unsigned char Unused[14];
   } v;
```

```
    unsigned char b[32] __attribute__ ((packed));
} tcfgMeasurement;
// -- tcfgMemory ------------------------------------------------------------------------------------------------------------------------------------------------------
typedef union {
    struct __attribute__ ((packed)) {
        unsigned short Verifier;
        unsigned long SecsFLASHStart, MinsFLASHStart, HrsFLASHStart, EventsFLASHStart, DiagFLASHStart;
        unsigned char Unused[10];
    } v;
    unsigned char b[32] __attribute__ ((packed));
} tcfgMemory;
// -- tcfgEMICCalibration ---------------------------------------------------------------------------------------------------------------------------------------------
typedef union {
    struct __attribute__ ((packed)) {
        unsigned short Verifier;
        unsigned char PGAGain;
        unsigned short VGain[3], IGain[3], WGain[3], VARGain[3], VAGain[3], VRMSOffset[3], IRMSOffset[3], WattOffset[3], VAROffset[3];
        unsigned char PhaseCalib[3], WattDivider, VARDivider, VADivider;
        unsigned char Unused;
    } v;
    unsigned char b[64] __attribute__ ((packed));
} tcfgEMICCalibration;
// -- tcfgMeasurementCal ----------------------------------------------------------------------------------------------------------------------------------------------
typedef union {
    struct __attribute__ ((packed)) {
        unsigned short Verifier;
        float VRMS[3], IRMS[12], WPower[12], VARPower[12], VAPower[12], Frequency;
        unsigned char OPMODE, MMODE, WAVMODE, COMPMODE, LCYCMODE;
        unsigned char Unused[9];
    } v;
    unsigned char b[224] __attribute__ ((packed));
} tcfgMeasurementCal;
// -- tcfgCommunications ----------------------------------------------------------------------------------------------------------------------------------------------
typedef union {
    struct __attribute__ ((packed)) {
        unsigned char BluetoothFlags;
        unsigned char GSMFlags;
        unsigned char PIN[4], APNName[15];
        unsigned short ListenPort;
        unsigned char Unused[9];
    } v;
    unsigned char b[32] __attribute__ ((packed));
} tcfgCommunications;
```

```
/*==  DATAFLASH.H & DATAFLASH.C  ================================================================================================
Memory interface routines for Atmel AT45DB642 DataFlash memory.
==============================================================================================================================*/
#include "HardwareProfile.h"
// -- Definitions -----------------------------------------------------------------------------------------------------------------------------------------------
// -- User: SPI Interface Port ----------------------------------------------------------------------------------------------------------------------------------
#define DFLASH_SPI_TxByte          SPI2_TxByte
// -- Type definition: tDFAddress -------------------------------------------------------------------------------------------------------------------------------
typedef union {
    struct {
        U8 Device __attribute__ ((packed));
        U16 Page __attribute__ ((packed));
        U16 Address __attribute__ ((packed));
    };
    unsigned char Bytes[5];
} tDFAddress;

//==  DATAFLASH.C  ============================================================================================================
#include "DataFLASH.h"
// - DataFLASH_SetChipSelect: Sets the chip select line of the relevant DataFlash device to 'state'. ---------------------------------------------------------
void DataFLASH_SetChipSelect(U8 device, U8 state)
{
    switch (device) {
        case 1 : CS_FlashM1 = state; break;
        case 2 : CS_FlashM2 = state; break;
        case 3 : CS_FlashM3 = state; break;
    };
}
// - DataFLASH_GetStatus: Retrieves the Status register from DataFLASH 'device'. ----------------------------------------------------------------------------
U8 DataFLASH_GetStatus(U8 device)
{
```

```
   U8 temp;
   DataFLASH_SetChipSelect(device, 0);                    // Lower the Chip Select, select the DataFlash for SPI comms
   DFLASH_SPI_TxByte(0xD7);                               // Command byte for status access
   temp = DFLASH_SPI_TxByte(0x00);                        // Transmit zero to read the status into 'temp'
   DataFLASH_SetChipSelect(device, 1);                    // Raise the CS, de-selecting the DataFlash
   return temp;                                           // Return the status
}
// - DataFLASH_DeviceOK: Uses the DataFLASH Status register to determine if the device is operating correctly. ----------
U8 DataFLASH_DeviceOK(U8 device) {  return ((DataFLASH_GetStatus(device) & 0b00111100) == 0b00111100);  }
// - DataFLASH_ReadSRAMBuffer ----------
// Reads 'Length' bytes from the specified DataFLASH 'DeviceID' and 'SRAM' buffer starting at 'Address' and writing it to 'Buffer' at 'Offset'.
void DataFLASH_SRAM_Read(U8 device, U8 sram, U16 address, U8 *buffer, U16 length)
{
   U16 cntr1;
   TMyU16 temp;
   DataFLASH_SetChipSelect(device, 0);                    // Select the DataFlash for SPI comms
   if (sram == 1)                                         // Specify which SRAM to access
      DFLASH_SPI_TxByte(0xD4);
   else
      DFLASH_SPI_TxByte(0xD6);
   DFLASH_SPI_TxByte(0x00);                               // Upper address byte which is not used
   temp.Value = address;                                  // Copy the address into a 16-bit structure
   DFLASH_SPI_TxByte(temp.Bytes[1]); DFLASH_SPI_TxByte(temp.Bytes[0]);  // Transmit the address MSB, LSB
   DFLASH_SPI_TxByte(0x00);                               // Don't-care byte to initialize read operation
   for (cntr1=0;cntr1<length;cntr1++) *buffer++ = DFLASH_SPI_TxByte(0x00);  // Read the values from DataFlash SRAM into internal 'buffer'
   DataFLASH_SetChipSelect(device, 1);                    // De-select the DataFlash SPI
}
// - DataFLASH_WriteSRAMBuffer: Writes 'Length' bytes from 'Buffer' at 'Offset' to the specified 'DeviceID' DataFLASH 'SRAM' starting at 'Address'. ----------
void DataFLASH_SRAM_Write(U8 device, U8 sram, U16 address, U8 *buffer, U16 length)
{
   U16 cntr1; TMyU16 temp;
   DataFLASH_SetChipSelect(device, 0);                    // Select the DataFlash for SPI comms
   if (sram == 1)                                         // Specify which SRAM to access
      DFLASH_SPI_TxByte(0x84);
   else
      DFLASH_SPI_TxByte(0x87);
   DFLASH_SPI_TxByte(0x00);                               // Upper address byte which is not used
   temp.Value = address;                                  // Copy the address into a 16-bit structure
   DFLASH_SPI_TxByte(temp.Bytes[1]); DFLASH_SPI_TxByte(temp.Bytes[0]);  // Transmit the address MSB, LSB
   for (cntr1=0;cntr1<length;cntr1++) DFLASH_SPI_TxByte(*buffer++);  // Write the values from 'buffer' to DataFlash SRAM
   DataFLASH_SetChipSelect(device, 1);                    // De-select the DataFlash to disable communications
}
// - DataFLASH_PageToSRAM: Moves a page of data to the internal SRAM specified by 'sram'. ----------
void DataFLASH_PageToSRAM(U8 device, U16 page, U8 sram)
{
   TMyU32 temp;
   DataFLASH_SetChipSelect(device, 0);                    // Select the DataFlash for SPI comms
   if (sram == 1)                                         // Specify which SRAM to access
      DFLASH_SPI_TxByte(0x53);
   else
      DFLASH_SPI_TxByte(0x55);
   temp.Value = page; temp.Value <<= 11;                  // Specify the page address
   DFLASH_SPI_TxByte(temp.Bytes[2]); DFLASH_SPI_TxByte(temp.Bytes[1]); DFLASH_SPI_TxByte(temp.Bytes[0]);
   DataFLASH_SetChipSelect(device, 1);                    // De-select the DataFlash to disable communications
}
// - DataFLASH_SRAMToPage: Performs an internal erase and write of the internal SRAM to a specified page in Flash. ----------
void DataFLASH_SRAMToPage(U8 device, U8 sram, U16 page)
{
   TMyU32 temp;
   DataFLASH_SetChipSelect(device, 0);                    // Select the DataFlash for SPI comms
   if (sram == 1)                                         // Specify which SRAM to access
      DFLASH_SPI_TxByte(0x83);
   else
      DFLASH_SPI_TxByte(0x86);
   temp.Value = page; temp.Value <<= 11;                  // Specify the page and address to copy to
   DFLASH_SPI_TxByte(temp.Bytes[2]); DFLASH_SPI_TxByte(temp.Bytes[1]); DFLASH_SPI_TxByte(temp.Bytes[0]);
   DataFLASH_SetChipSelect(device, 1);                    // De-select the DataFlash to disable communications
}
// - DataFLASH_Idle: Returns 'True' if the DataFlash is not currently writing. ----------
U8 DataFLASH_Idle(U8 device) {  return ((DataFLASH_GetStatus(device) & 0x80) != 0);  }
// - DataFLASH_Read: Reads 'length' bytes from DataFlash memory and puts it in 'buffer'. ----------
void DataFLASH_Read(U8 device, U16 page, U16 address, U16 length, U8 *buffer)
{
   U8 i; TMyU32 temp;
   DataFLASH_SetChipSelect(device, 0);                    // Select the DataFlash for SPI comms
   DFLASH_SPI_TxByte(0xD2);                               // Command to read data from the DataFlash
```

```
   temp.Value = page; temp.Value <<= 11; temp.Value += address;                  // Specify the read page and address
   DFLASH_SPI_TxByte(temp.Bytes[2]); DFLASH_SPI_TxByte(temp.Bytes[1]); DFLASH_SPI_TxByte(temp.Bytes[0]);
   for (i=0;i<4;i++) DFLASH_SPI_TxByte(0x00);                                    // Transmit Don't Care bytes
   while (length-- > 0) *buffer++ = DFLASH_SPI_TxByte(0x00);                     // Read the values from DataFlash and place in 'buffer'
   DataFLASH_SetChipSelect(device, 1);                                           // De-select the DataFlash to disable communications
}
// - DataFLASH_Write_Buffer: Writes 'length' bytes to the SRAM page and internal Flash. ----------
void DataFLASH_Write_Buffer(tDFAddress *dest, U8 *buffer, U16 length)
{
   U8 dev; U16 addr, bytesInPage, pg;
   dev = dest->Device;                                                           // Get the device number from the tDFAddress data type
   pg = dest->Page;                                                              // Get the page from 'dest'
   addr = dest->Address;                                                         // Get the address from 'dest'
   while (length > 0)
   {
      bytesInPage = 1056 - addr;                                                 // Determine the number of bytes remaining in this page
      if (bytesInPage == 0)                                                      // Advance to the next page if there is no free space
      {
         if (pg < 8192)
            pg++;
         else
         {
            pg = 0;                                                             // If there are no more pages left, move to the next DataFlash device
            dev++;
         }
         bytesInPage = 1056;                                                    // A full page of 1056 bytes can be written
         addr = 0;                                                              // Point to the beginning of the page
      }
      if (length < bytesInPage) bytesInPage = length;                           // Restrict the write size to the number of bytes remaining
      DataFLASH_PageToSRAM(dev, pg, 1);                                         // Copy the DataFlash contents to SRAM for modfiication
      while (!DataFLASH_Idle(dev));                                             // Wait till the operation is complete
      DataFLASH_SRAM_Write(dev, 1, addr, buffer, bytesInPage);                  // Update the SRAM buffer with the new data
      DataFLASH_SRAMToPage(dev, 1, pg);                                         // Write the modified SRAM to the DataFlash memory
      while (!DataFLASH_Idle(dev));                                             // Wait till the operation is complete
      length -= bytesInPage;                                                    // Determine the data remaining
      addr += bytesInPage;                                                      // Increment the address pointer by the amount of data written
   }
   dest->Device = dev;                                                          // Update the 'dest' memory pointer
   dest->Page = pg;
   dest->Address = addr;
}
```

```
/*== DSPICSUPPORT.H & DSPICSUPPORT.C ==========================================================
Support routines for the dsPIC processor.
==========================================================================================*/
#include "HardwareProfile.h"

//== DSPICSUPPORT.C ============================================================================
#include "dsPICSupport.h"  #include <libpic30.h>
// -- dsPIC EEPROM_WriteRow: Writes a row (32 bytes) of data from memory pointer 'buffer' to EEPROM at 'address'. ----------
void intEE_WriteRow(unsigned short address, unsigned char *buffer)
{
   unsigned char i; unsigned short tempBuffer[16]; TMyU16 tempVal;
   _prog_addressT eeAddr;
   for (i=0;i<16;i++)
   {
      tempVal.Bytes[0] = *buffer++; tempVal.Bytes[1] = *buffer++;
      tempBuffer[i] = tempVal.Value;
   }
   eeAddr = 0x7FF800 + address;
  _erase_eedata(eeAddr, _EE_ROW);
  _wait_eedata();
  _write_eedata_row(eeAddr, &tempBuffer);
  _wait_eedata();
}
// -- dsPIC EEPROM_ReadRow: Reads a row (32-bytes) of data from 'address' and returns it in memory pointer 'buffer'. ----------
void intEE_ReadRow(unsigned short address, unsigned char *buffer)
{
   _prog_addressT eeAddr;
   eeAddr = 0x7FF800 + address;
  _memcpy_p2d16(buffer, eeAddr, _EE_ROW);
}
// -- U16BitTest: Returns true if the bit specified by 'bitValue' is set or not. ----------
unsigned char U16BitTest(unsigned short value, unsigned char bitValue)
{
```

```
    unsigned short temp = 1;
    while (bitValue-- > 0) temp <<= 1;
    return ((value & temp) == temp);
}
```

```
/*== HARDWAREPROFILE.H ==========================================================================================
Defines the interfaces specific for the Power Profiler.
==========================================================================================*/
#include <p30fxxxx.h>  #include <stdio.h>  #include <string.h>  #include "Types.h"  #include "ConfigBlockDefns.h"  #include "dsPICSupport.h"
// -- Crstal Oscillator Definition: Used to determine delay and communication baud rate values. ----------
#define FOSC 117964800                                  // 7.3728MHz crystal 16X PLL
#define FCY  (FOSC / 4)                                 // Quadrature clock is used in the dsPIC so the instruction cycle is FOSC/4
// -- General Timing Controls ----------
extern unsigned char timerTick100ms;
extern unsigned char Time_Elapsed(unsigned char startTime);      // Returns the difference between startTime and timerTick100ms
extern void TimeCriticalCode(void);                             // Used to execute important code during delays
// -- I/O Definitions ----------
#define LED_Acquisition             LATAbits.LATA9      #define LED_CommsRx             PORTDbits.RD9
#define LED_CommsTx                 PORTBbits.RB5       #define SPECC_RxLED             PORTDbits.RD9
#define SPECC_TxLED                 PORTBbits.RB5       #define SelReg_C                PORTBbits.RB6
#define CS_VMUX                     1
extern void SelReg_Set(unsigned char pin);
extern void SelReg_Clear(unsigned char pin);
// -- tSelRegValue ----------
typedef union {
    struct __attribute__ ((packed)) {
        unsigned CS_VS              : 1;    unsigned CurrSel        : 2;
        unsigned IndIPSel           : 3;    unsigned RCPower        : 1;
        unsigned RFPower            : 1;
    } v;
    unsigned char b __attribute__ ((packed));
} tSelRegValue;
// -- Communication Interfaces ----------
// -- RF Interface ----------
#define RF_CTS_Dir                  TRISAbits.TRISA10   #define RF_CTS                  PORTAbits.RA10
#define RF_RTS_Dir                  TRISBbits.TRISB7    #define ipRF_RTS                PORTBbits.RB7
#define ipRF_DCD                    PORTBbits.RB3
// -- GSM Interface ----------
#define RC_DTR                      PORTCbits.RC1       #define RC_RTS                  PORTCbits.RC2
#define RC_Reset                    PORTDbits.RD10      #define ipRC_RI                 PORTBbits.RB4
#define ipRC_DSR                    PORTDbits.RD11      #define ipRC_CTS                PORTBbits.RB2
// -- Sensor Network Interface ----------
#define INT_SensNWIRQ               PORTDbits.RD13      #define MAX3100_CS              PORTGbits.RG12
#define MAX3100_SPI_TxByte          SPI1_TxByte         #define MAX3100_XTAL            36864
// -- Inter-Processor Communications ----------
#define CS_IPC_Dir                  TRISGbits.TRISG9    #define CS_IPC                  LATGbits.LATG9
#define IPC_CS_Dir                  TRISGbits.TRISG9    #define IPC_CS_Pin              PORTGbits.RG9
#define IPC_HS_Dir                  TRISDbits.TRISD14   #define IPC_HS_Pin              PORTDbits.RD14
// -- Data Acquisition and Storage ----------
// - Energy Measurement ----------
#define CS_EM1                      PORTCbits.RC4       #define CS_EM2                  PORTCbits.RC3
#define CS_EM3                      PORTGbits.RG13      #define CS_EM4                  PORTGbits.RG15
#define INT_EM1                     PORTAbits.RA12      #define INT_EM2                 PORTAbits.RA13
#define INT_EM3                     PORTAbits.RA14      #define INT_EM4                 PORTAbits.RA15
#define ADE7758_SPI_TxByte          SPI1_TxByte
// - SRAM Memory ----------
#define SRAM_PortDir                TRISD               #define SRAM_Port               PORTD
#define SRAM_AR1                    PORTCbits.RC13      #define SRAM_AR2                PORTGbits.RG14
#define SRAM_AR3                    PORTCbits.RC14      #define SRAM1_CE                PORTGbits.RG0
#define SRAM2_CE                    PORTGbits.RG1       #define SRAM_WR                 PORTFbits.RF0
#define SRAM_OE                     PORTFbits.RF1
// - DataFLASH Memory ----------
#define CS_FlashM1                  PORTDbits.RD12      #define CS_FlashM2              PORTBbits.RB8
#define CS_FlashM3                  PORTDbits.RD8
```

```
/*== I2CDEVICES.H & I2CDEVICES.C ================================================================================
Control interfaces for the Microchip TCN75 Digital temperature sensor and Maxim DS3231S Real-Time-Clock.
==========================================================================================*/
#include "HardwareProfile.h"  #include "I2CSPIComms.h"
// == DS3231S: REAL-TIME CLOCK ====================================================================================
// -- Data Type: tDateTime: Represents the current time-of-day used in the Power Profiler. ----------
typedef union {
    struct __attribute__ ((packed)) {
        unsigned char Year, Month, Day, DayOfWeek;
```

```
    unsigned char Hrs, Min, Sec;
  };
  unsigned char b[7] __attribute__ ((packed));
} tDateTime;

//==  I2CDEVICES.C  ================================================================================================
#include "I2CDevices.h" #include "I2CSPIComms.h"
// == TCN75: TEMPERATURE SENSOR ================================================================================
// -- TCN75_Initialize: Initializes the temperature sensor for auto-conversion. Returns bool indicating that device is connected. ----------
bool I2CTemp_Initialize(void)
{
  bool ack;
  I2C_Start();                                  // Start I2C data transfer
  I2C_Write(0x90, true);                        // Specify the slave address: A2 = 0, A1 = 0, A0 = 0
  I2C_Write(0x01, true);                        // Write to the configuration register
  ack = I2C_Write(0x00, true);                  // Clear the configuration register to enable the device
  I2C_Stop();                                   // Stop I2C data transfer
  return ack;                                   // Return the ACK result to indicate successful comms
}
// -- TCN75_SetPointer: Selects the TCN75 register to access specified by devReg. Returns bool value indicating success. ----------
bool I2CTemp_SetPointer(U8 devReg)
{
  bool ack;
  I2C_Start();                                  // Start the I2C data transfer
  I2C_Write(0x90, true);                        // Specify the slave address
  ack = I2C_Write(devReg, true);                // Write the new pointer value
  I2C_Stop();                                   // Stop the I2C data transfer
  return ack;                                   // Return the ACK result to indicate successful comms
}
// -- TCN75_Temperature: Retrieves temperature value from sensor in 'result', routine returns bool value indicating success. ----------
bool I2CTemp_Temperature(TMyU16 *result)
{
  unsigned char ack;
  I2C_Start();                                  // Start the I2C data transfer
  ack = I2C_Write(0x91, true);                  // Specify a read
  (*result).Bytes[1] = I2C_Read(true);          // Read the first byte with an acknowledge
  (*result).Bytes[0] = I2C_Read(false);         // Read the second byte without an acknowledge
  I2C_Stop();                                   // Stop the I2C data transfer
  return ack;                                   // return the ACK result to indicate successful comms
}
// == DS3231S: REAL-TIME CLOCK ==================================================================================
// -- BCDToUC_Converts a Binary-Coded-Decimal value to a byte result. ----------
unsigned char BCDToUC(unsigned char value)
{
  unsigned char temp1, temp2;
  temp1 = ((value & 0xF0) >> 4) * 10; temp2 = value & 0x0F;
  temp1 += temp2;
  return temp1;
}
// -- UCToBCD_Converts a byte value to its Binary-Coded-Decimal representation. ----------
unsigned char UCToBCD(unsigned char value)
{
  unsigned char temp1, temp2;
  temp1 = value / 10; temp2 = value - (temp1 * 10);
  return ((temp1 << 4) + temp2);
}
// -- DS3231_Intialize:Initializes the RTC and enables the square wave output with 1Hz resolution. Returns true if connection was possible. ----------
bool DS3231_Initialize(void)
{
  bool ack;
  I2C_Start();                                  // Start the I2C communications
  I2C_Write(0xD0, true);                        // Specify a write to the DS3231
  I2C_Write(0x0E, true);                        // Write is to the Control register at address 0x0E
  ack = I2C_Write(0, true);                     // Enabled square-wave output with a 1Hz resolution. No alarms.
  I2C_Stop();                                   // Stop the I2C communications
  return ack;                                   // Return the ack to confirm communications
}
// -- DS3231_GetClock: Retrieves the current time-of-day from the RTC and returns it in result. Routine returns the clock validity. ----------
bool DS3231_GetClock(tDateTime *result)
{
  bool ack, timeValid = false; unsigned char cntr, rtcBuffer[16];
  I2C_Start();                                  // Start the I2C communications
  I2C_Write(0xD0, gcExpectACK);                 // Specify a write to the DS3231
  I2C_Write(0x00, gcExpectACK);                 // Set the pointer to the seconds register at address 0x00
  I2C_RStart();                                 // Issue a repeated start command
  ack = I2C_Write(0xD1, gcExpectACK);           // Specify a read from the current pointer location
```

```
    for (cntr=0; cntr<15; cntr++) rtcBuffer[cntr] = I2C_Read(gcI2CSendACK);   // Read the registers, the DS3231 will auto-increment the address pointer
    rtcBuffer[15] = I2C_Read(gcI2CSendNACK);                                  // Send a non-acknowledge for the last byte
    I2C_Stop();                                                               // Stop the I2C communications
    if (!i2cCommsError)                                                       // Check for an I2C communications error
    {
        timeValid = ((rtcBuffer[15] & 0x80) != 0x80);                         // Is the RTC time valid?
        (*result).Sec = BCDToUC(rtcBuffer[0]);                                // Seconds converted from BCD to byte
        (*result).Min = BCDToUC(rtcBuffer[1]);                                // Minutes converted from BCD to byte
        (*result).Hrs = BCDToUC(rtcBuffer[2] & 0x3F);                         // Hours converted from BCD to byte
        (*result).DayOfWeek = rtcBuffer[3];                                   // Day of week
        (*result).Day = BCDToUC(rtcBuffer[4]);                                // Day of month converted from BCD to byte
        (*result).Month = BCDToUC(rtcBuffer[5] & 0x1F);                       // Month converted from BCD to byte
        (*result).Year = BCDToUC(rtcBuffer[6]);                               // Year converted from BCD to byte
    }
    return timeValid;
}
// -- DS3231_SetClock: Sets the current time-of-day in the RTC to 'value' and returns true if successful. ----------------------------------------------------------
bool DS3231_SetClock(tDateTime value)
{
    bool ack;
    // - Clear the Oscillator Stopped flag bit if it is set -
    I2C_Start();                                                              // Start the I2C communications
    I2C_Write(0xD0, gcExpectACK);                                             // Specify a write to the DS3231 to set the register pointer
    I2C_Write(0x0F, gcExpectACK);                                             // Write will be to the Control/Status register
    ack = I2C_Write(0x00, gcExpectACK);                                       // Clear the register to initialize the oscillator stopped flag bit
    I2C_Stop();                                                               // Stop the I2C communications
    if (!ack) return false;                                                   // If there was no response from the DS3231 then return an error
    I2C_Start();                                                              // Start the I2C communications
    I2C_Write(0xD0, gcExpectACK);                                             // Specify a write to the DS3231 to set the register pointer
    I2C_Write(0x00, gcExpectACK);                                             // Set the memory pointer to start with the Seconds register
    I2C_Write(UCToBCD(value.Sec), gcExpectACK);                               // Write the Seconds value, converting it to BCD
    I2C_Write(UCToBCD(value.Min), gcExpectACK);                               // Write the Minutes value, converting it to BCD
    I2C_Write(UCToBCD(value.Hrs), gcExpectACK);                               // Write the Hours value, converting it to BCD
    I2C_Write(value.DayOfWeek, gcExpectACK);                                  // Write the Day of week value
    I2C_Write(UCToBCD(value.Day), gcExpectACK);                               // Write the Day of Month value, converting it to BCD
    I2C_Write(UCToBCD(value.Month), gcExpectACK);                             // Write the Month value, converting it to BCD
    ack = I2C_Write(UCToBCD(value.Year), gcExpectACK);                        // Write the Year value, converting it to BCD
    I2C_Stop();                                                               // Stop the I2C communications
    return ack;                                                               // Return true if comms was successful
}
```

```
/*==  I2CSPICOMMS.H & I2CSPICOMMS.C  ========================================================================================
Interface routines for I2C and SPI communications on the dsPIC30F processor.
==========================================================================================================================*/
#include "HardwareProfile.h"
// == I2C INTERFACE ROUTINES ==============================================================================================
// -- Definitions ---------------------------------------------------------------------------------------------------------
#define gcI2CSendACK        0               #define gcI2CSendNACK       1
#define gcNoACK             0               #define gcExpectACK         1
// -- tI2C_Status: Indicates the status of the I2C interface. -------------------------------------------------------------
typedef union {
    struct {
        unsigned WriteError     : 1;        unsigned ReadError      : 1;
        unsigned DeviceError    : 1;        unsigned Unused         : 5;
    };
    unsigned char Byte;
} tI2C_Status;
// == SPI INTERFACE ROUTINES ==============================================================================================
// -- Definitions ---------------------------------------------------------------------------------------------------------
#define spiStopIdle             0x2000      #define spiFramed               0x4000
#define spiSampleEnd            0x0200      #define spiSampleMid            0
#define spiDOutActiveToIdle     0x0100      #define spiDOutIdleToActive     0
#define spiSSEnabled            0x0080
#define spiClkIdleHigh          0x0040      #define spiClkIdleLow           0
#define spiMaster               0x0020      #define spiSlave                0
#define spiSecPresc_1_1         0b0000000000011100  #define spiSecPresc_2_1     0b0000000000011000
#define spiSecPresc_3_1         0b0000000000010100  #define spiSecPresc_4_1     0b0000000000010000
#define spiSecPresc_5_1         0b0000000000001100  #define spiSecPresc_6_1     0b0000000000001000
#define spiSecPresc_7_1         0b0000000000000100  #define spiSecPresc_8_1     0b0000000000000000
#define spiPriPresc_1_1         0b0000000000000011  #define spiPriPresc_4_1     0b0000000000000010
#define spiPriPresc_16_1        0b0000000000000001  #define spiPriPresc_64_1    0b0000000000000000

/*==  I2CSPICOMMS.C  =========================================================================================================
#include "I2CSPIComms.h"
// == I2C INTERFACE ROUTINES ==============================================================================================
```

```
// -- Variable Definitions ------------------------------------------------------------------------------------------------------------------------------------
bool i2cCommsError;
// -- I2C_Configure: Initializes the I2C interface for the selected baud rate. --------------------------------------------------------------------------------
void I2C_Configure(U32 baudRate, U8 slewRateOff)
{
   I2CCON = 0;                                                    // Clear the I2C Control register
   I2CBRG = ((FCY / baudRate) - (FCY/1111111)) - 1;               // Calculate the Baud rate register value
   I2CCONbits.DISSLW = slewRateOff;                               // Set the Slew Rate parameter
   I2CSTAT = 0;                                                   // Clear the I2C Status register
   I2CCONbits.I2CEN = true;                                       // Enable the I2C interface
}
// -- I2C_Start: Sets an I2C Start condition and waits till completed. ---------------------------------------------------------------------------------------
void I2C_Start(void)
{
   I2CCONbits.SEN = 1;                                            // Set an I2C Start condition
   while (I2CCONbits.SEN);                                        // Wait till the condition is completed
   i2cCommsError = false;                                         // Clear the I2C error flag
}
// -- I2C_Stop: Sets an I2C Stop condition and waits till completed. ----------------------------------------------------------------------------------------
void I2C_Stop(void) { I2CCONbits.PEN = 1; while (I2CCONbits.PEN); }
// -- I2C_RStart: Sets an I2C Repeates Start condition and waits till completed. ----------------------------------------------------------------------------
void I2C_RStart(void) { I2CCONbits.RSEN = 1; while (I2CCONbits.RSEN); }
// -- I2C_Read: Initiates an I2C Read and sends an acknowledge if 'sendACK' is sent. The read byte is returned. ----------------------------------------------
unsigned char I2C_Read(unsigned char sendACK)
{
   // -- Wait until the I2C interface is free --
   while(I2CCONbits.SEN || I2CCONbits.PEN || I2CCONbits.RCEN || I2CCONbits.ACKEN || I2CSTATbits.TRSTAT);
   I2CCONbits.RCEN = true;                                        // Enable an I2C Read
   while (!I2CSTATbits.RBF);                                      // Wait till the read is complete
   I2CCONbits.ACKDT = sendACK;                                    // Send the ACK if required
   I2CCONbits.ACKEN = true;                                       // Enable the I2C ACK/NACK sending
   while (I2CCONbits.ACKEN);                                      // Wait till the ACK has been sent
   return I2CRCV;                                                 // Return the received byte
}
// -- I2C_Write: Initiates an I2C Write of 'data' and if an acknowledge is expected from the slave it is returned. -------------------------------------------
unsigned char I2C_Write(unsigned char data, unsigned char ackExpected)
{
   IFS0bits.MI2CIF = false;                                       // Clear the I2C interrupt flag
   I2CTRN = data;                                                 // Set the byte to send
   while (!IFS0bits.MI2CIF)                                       // Wait till the send is complete
      if (I2CSTATbits.BCL)
      {
         i2cCommsError = true;                                    // If a bus collision occurs mark the error flag
         return false;                                            // Write failed so return to calling method
      }
   IFS0bits.MI2CIF = false;                                       // Clear the I2C interrupt flag
   if ((ackExpected) && (I2CSTATbits.ACKSTAT != 0))               // If an ACK was expected but not received return a false
      return false;
   return true;                                                   // Return that the command executed correctly
}
// == SPI INTERFACE ROUTINES ==================================================================================================================================
// -- SPI1_Configure:  Configures SPI 1 with the STAT and CONTROL registers specified. ----------------------------------------------------------------------
void SPI1_Configure(unsigned short spiStat, unsigned short spiCON)
{
   IEC0bits.SPI1IE = 0;                                           // Disable the SPI 1 interrupt
   SPI1STAT = spiStat;                                            // Configure the SPI 1 status register
   SPI1CON = spiCON;                                              // Configure the SPI 1 control register
   SPI1STAT |= 0x8000;                                            // Enable the SPI port
   SPI1STAT &= 0xFFBF;                                            // Clear any errors
}
// -- SPI1_TxByte:  Transmits 'value' on SPI 1 and returns the character received during the transmit. ------------------------------------------------------
unsigned char SPI1_TxByte(unsigned char value)
{
   while (SPI1STATbits.SPIRBF);                                   // Wait until a reception is completed if in progress
   SPI1STAT &= 0xA003;                                            // Clears the SPI error flags
   SPI1BUF = value;                                               // Loads the byte to send
   while (SPI1STATbits.SPITBF);                                   // Wait until data has been transmitted
   while (!SPI1STATbits.SPIRBF);                                  // Wait until data has been received (at the same time as the transmit)
   return (SPI1BUF & 0xFF);                                       // Return the received value
}
// -- SPI1_DeselectAll ----------------------------------------------------------------------------------------------------------------------------------------
// Deselects all devices that share the SPI 1 port.
void SPI1_DeselectAll(void)
{
   CS_EM1 = 1; CS_EM2 = 1; CS_EM3 = 1; CS_EM4 = 1;                // Deselect Energy Measurement ICs
   MAX3100_CS = 1;                                                // Deselect Sensor Network UART
```

```
}
// -- SPI2_Configure:  Configures SPI 2 with the STAT and CONTROL registers specified. ----------
void SPI2_Configure(unsigned short spiStat, unsigned short spiCON)
{
   IEC1bits.SPI2IE = 0;                                    // Disable the SPI 2 interrupt
   SPI2STAT = spiStat;                                     // Configure the SPI 2 status register
   SPI2CON = spiCON;                                       // Configure the SPI 2 control register
   SPI2STAT |= 0x8000;                                     // Enable the SPI port
   SPI2STAT &= 0xFFBF;                                     // Clear any errors
}
// -- SPI2_TxByte: Transmits 'value' on SPI 2 and returns the character received during the transmit. ----------
unsigned char SPI2_TxByte(unsigned char value)
{
   while (SPI2STATbits.SPIRBF);                            // Wait until a reception is completed if in progress
   SPI2STAT &= 0xA003;                                     // Clears the SPI error flags
   SPI2BUF = value;                                        // Load the byte to send
   while (SPI2STATbits.SPITBF);                            // Wait until data has been transmitted
   while (!SPI2STATbits.SPIRBF);                           // Wait until data has been received to return
   return (SPI2BUF & 0xFF);                                // Return the received value
}
// -- SPI2_DeselectAll: Deselects all devices that share the SPI 2 port. ----------
void SPI2_DeselectAll(void)
{
   CS_FlashM1 = 1; CS_FlashM2 = 1; CS_FlashM3 = 1;         // Disable the DataFlash memories
   CS_IPC = 1;                                             // Disable the Inter-Processor Communications (PIC18F CS pin)
}
```

```
/*==  MAIN.C  ==========================================================================================
Main routines for the control of the Power Profiler.
==========================================================================================================*/
#include "HardwareProfile.h"  #include <Timer.h>  #include "I2CDevices.h"  #include "Timekeeping.h"  #include "Comms_IPC.h"  #include "procMeasurement.h"
#include "procComms.h"  #include "procControl.h"
// -- dsPIC Processor Configuration Settings  ----------
_FOSC( CSW_FSCM_OFF & XT_PLL16 );                          _FWDT( WDT_OFF );
_FBORPOR( PBOR_OFF & BORV_45 & MCLR_EN );
// -- Local Variables ----------
bool LastRA7Val;          tSelRegValue selRegValue;
// == TIMEKEEPING / TIMING CONTROL ==========
unsigned char timerTick100ms = 0;
// -- Time_Elapsed: Returns the number of 100ms ticks that have elapsed since 'startTime'. ----------
unsigned char Time_Elapsed(unsigned char startTime)
{
   if (startTime <= timerTick100ms)
      return (timerTick100ms - startTime);
   else
      return ((256 - startTime) + timerTick100ms);
}
// -- Time_Wait_ms: Delays the specified number of milliseconds before returning. ----------
void Time_Wait_ms(unsigned short delay)
{
   unsigned short cntr1, cntr2;
   for (cntr1=0;cntr1<delay;cntr1++)
      for (cntr2=0;cntr2<2946;cntr2++)
         Nop();
}
// -- Time_Wait_us: Delays the specified number of microseconds before returning. ----------
void Time_Wait_us(void)
{
   unsigned char cntr;
   for (cntr=0;cntr<2;cntr++)  {  Nop(); Nop();  }
}

// == PERIPHERAL CONTROL ==========
// -- SelReg_Initialize: Sets the default value for the Octal Buffer (IC2) without effecting the other bits on PORTD. ----------
void SelReg_Initialize(void)
{
   TRISD &= 0xFF00;
   PORTD &= 0xFF00;
   PORTD |= 0xFF01;
   SelReg_C = true; SelReg_C = false;
}
// -- SelReg_Set: ORs the specified Pin with the lower byte of PORTD and updates the Octal Buffer (IC2) without affecting the other bits on PORTD. ----------
void SelReg_Set(unsigned char pin)
{
   TMyU16 tempPort;
```

```
    tempPort.Value = PORTD;
    tempPort.Bytes[0] |= pin;
    TRISD &= 0xFF00;
    LATD = tempPort.Value;
    SelReg_C = true; SelReg_C = false;
}
// -- SelReg_Clear: ANDs the specified Pin with the lower byte of PORTD and updates the Octal Buffer (IC2) without affecting the other bits on PORTD. ----------
void SelReg_Clear(unsigned char pin)
{
    TMyU16 tempPort;
    pin = 0xFF - pin;
    tempPort.Value = PORTD;
    tempPort.Bytes[0] &= pin;
    TRISD &= 0xFF00;
    LATD = tempPort.Value;
    SelReg_C = true; SelReg_C = false;
}
// -- dsPIC_Initialize: Initalizes the peripheral interfaces and modules in the dsPIC processor. ----------
void dsPIC_Initialize(void)
{
    // -- Set port directions and default output values --
    TRISA = 0xF9FF; PORTA = 0x0200;                 TRISB = 0xFE9F; PORTB = 0x0120;
    TRISC = 0x9FE1; PORTC = 0x001E;                 TRISD = 0b1110100011111111; PORTD = 0b0001001100000000;
    TRISF = 0b111010100; PORTF = 0b000000010;       TRISG = 0b0000000010111100; PORTG = 0xF003;
    // -- Initialize interrupts --
    IFS0 = 0x0000; IEC0 = 0x0000; IEC1 = 0x0000;
    // -- Initialize I2C Interface --
    I2C_Configure(400000, false);
    // -- Configure SPI Interface --
    SPI1_DeselectAll(); SPI1_Configure(0, spiMaster + spiDOutIdleToActive + spiSecPresc_3_1 + spiPriPresc_4_1);
    SPI2_DeselectAll(); SPI2CON = 0b0000000000111110;                   // Clock idle low, data o/p on transition Idle to Active, mid-data sampling
    SPI2STAT &= 0xFFBF; SPI2STATbits.SPIEN = true;
    // -- Configure Timer 4 for 100msec --
    OpenTimer4(T4_ON & T4_IDLE_CON & T4_PS_1_256 & T4_SOURCE_INT & T4_GATE_OFF, 11528);
    ConfigIntTimer4(T4_INT_PRIOR_1 & T4_INT_ON);
    EnableIntT4;
    // -- Configure Timer 5 for 10 msec --
    OpenTimer5(T5_ON & T5_IDLE_CON & T5_PS_1_256 & T5_SOURCE_INT & T5_GATE_OFF, 1150);
    ConfigIntTimer5(T5_INT_PRIOR_3 & T5_INT_ON);
    EnableIntT5;
}
// == MAIN ROUTINE ==========================================================================================================
int main (int argc, char * argv[])
{
    dsPIC_Initialize();                                 // Configure dsPIC internal registers
    SelReg_Initialize();                                // Set the IO control IC (IC2) to default values
    Time_Wait_ms(1000);                                 // Allow peripherals to 'settle' after power-up
    ProcCtrl_Initialize();                              // Initialize the Controller Task
    procMemory_Initialize();                            // Configure external memory access
    procMeas_Initialize();                              // Configure measurement control
    procComms_Initialize();                             // Configure communications
    // -- Enable interrupts that are needed --
    IEC1bits.INT1IE = true; IEC1bits.INT2IE = true; IEC2bits.INT3IE = true; IEC2bits.INT4IE = true;
    CNPU2bits.CN23PUE = true; CNEN2bits.CN23IE = true; IFS0bits.CNIF = false; IEC0bits.CNIE = true;
    // -- Start processing software tasks --
    while (1)
    {
        procMeas_Processor();                           // Measurements processor
        ProcCtrl_Processor();                           // Control Processor
        procMeas_Processor();                           // Measurements processor (repeated to ensure timely response)
        procComms_Processor();                          // Communications Processor
    }
    return -1;
}
// -- Time Critical Code: This routine is called during loops to ensure that control and measuremnet functions are executed. ----------
void TimeCriticalCode(void)
{
    ProcCtrl_Processor();                               // Control Processor
    procMeas_Processor();                               // Measurements Processor
}
// -- Interrupt : Timer 4: Called every 100ms and is used for sample acquisition signalling and for internal time delay. ----------
void __attribute__((__interrupt__)) _T4Interrupt(void)
{
    timerTick100ms++;
    procComms.Tick100ms = true;
    procControl.Tick100ms = true;
```

```
    procMeas.Tick100ms = true;
    IFS1bits.T4IF = 0;
}
// -- Interrupt: Timer 5: Occurs every 10ms and is used to increment the Inter-Processor Communications message timeout counter. ----------
void __attribute__((__interrupt__)) _T5Interrupt(void)
{
    WriteTimer5(0);
    ipcTimer_10ms++;
    IFS1bits.T5IF = false;
}
// -- Interrupt : Pin Change Notification ----------
void __attribute__((__interrupt__)) _CNInterrupt(void)
{
    // -- Check for a toggle of the RTC second tick --
    if (PORTAbits.RA7 != LastRA7Val)
    {
        LastRA7Val = PORTAbits.RA7;
        if (PORTAbits.RA7 == 0)
        {
            procComms.Tick1s = true;
            procControl.SecTick = true;
        }
    }
    IFS0bits.CNIF = 0;
}
```

```
/*== PROCCOMMS.H & PROCCOMMS.C ==========
Initializes and processes the communications sub-tasks.
==========*/
#include "HardwareProfile.h"
// -- Constant and Type Definitions ----------
#define COMMS_BUFF_SIZE             550         #define CRC_POLYNOMIAL16        0x8005
#define CMDSRC_BT                   0           #define CMDSRC_GSM              1
#define CMDSRC_USB                  2           #define CMDSRC_SENSNW           3
// -- tCommsStatus: Communications Task status register. ----------
typedef union {
    struct {
        unsigned HWFault            : 1;        unsigned CfgFault           : 1;
        unsigned Tick100ms          : 1;        unsigned Tick1s             : 1;
        unsigned CmdRx              : 1;        unsigned CmdSource          : 2;
        unsigned SensNW_Enabled     : 1;        unsigned SensNW_HW_Ok       : 1;
        unsigned Connected          : 1;        unsigned BT_Enabled         : 1;
        unsigned GSM_Enabled        : 1;        unsigned Unused             : 4;
    };
    unsigned short Value;
} tprocCommsStatus;
// -- tCommsPacket: Represents a packet structure as defined for use in the Power Profiler. ----------
typedef struct {
    U8  TID, Cmd;
    U16 PktNum, MessLen;
    U8  *Params, *Response;
    U16 RespLength;
    U8   PacketSource;
    U8  *BufferAddr;
} tCommsPacket;

// == PROCCOMMS.C ==========
#include "procComms.h"  #include "Commands.h"  #include "Comms_GSM.h"  #include "Comms_Bluetooth.h"  #include "Comms_IPC.h"
#include "Comms_MAX3100.h"  #include "ConfigBlockDefns.h"
// -- Global Variables ----------
unsigned char commsBuffer[COMMS_BUFF_SIZE];
int commsBufferPtr = 0, commsPacketLength = 0;
tprocCommsStatus procComms;
// -- Comms Task: Initialize: Initializes the Communications Task: IPC, Bluetooth, GSM and Sensor Network interfaces. ----------
void procComms_Initialize(void)
{
    procComms.Value = 0;
    LED_CommsTx = False; LED_CommsRx = False;
    IPC_Initialize();                                   // Initialize the inter-processor communications
    CfgMem_Read(CFG_COMMS_ADDR, 1, &cfgComms.b[0]);     // Load the configuration values from EEPROM
    if (cfgComms.v.Verifier == 0xBBAA)                  // Verify configuration is valid
    {
        procComms.SensNW_Enabled = ((cfgComms.v.SensNWFlags & 1) == 1);
        procComms.BT_Enabled = ((cfgComms.v.BluetoothFlags & 1) == 1);
        // -- Set the GSM parameters --
```

```
      procComms.GSM_Enabled = ((cfgComms.v.GSMFlags & 1) == 1);
      for (i=0;i<4;i++) gsmPIN[i] = cfgComms.v.PIN[i];
      gsmPIN[4] = 0;
      for (i=0;i<APNNAMELENGTH;i++) apnName[i] = cfgComms.v.APNName[i];
      gsmListenPort = cfgComms.v.ListenPort;
   }
   else
   {
      procComms.BT_Enabled = true;                                  // Bluetooth is always enabled by default
      procComms.CfgFault = true;                                    // Mark the configuration fault flag
   }
   if (procComms.SensNW_Enabled)                                    // Initialize the Sensor Network Interface (NOT CURRENTLY USED)
   {
      // The initialization sequence for the sensor network processor would be called here.
   }
   else
      procComms.SensNW_HW_Ok = MAX3100_Initialize(MAX3100_Shutdown);  // Disable the MAX3100
   BT_Initialize(true);                                             // Initialize Bluetooth communications
   GSM_Initialize(true);                                            // Initialize GSM communications
}
// -- Comms Task: Processor ---------------------------------------------------------------------------------------------------------------------------------------------------------------
// Executes the sub-task communications processors and if a command is received, it is checked for errors and then processed.
void procComms_Processor(void)
{
   tCommsPacket cmdPacket;
   if (procComms.Tick100ms)                                         // Tasks executed every 100ms
   {
      gsmStatus.Tick100ms = true;
      ipcStatus.Tick100ms = true;
      procComms.Tick100ms = false;
   }
   if (procComms.Tick1s)                                            // Tasks executed every second
   {
      gsmStatus.Tick1s = true;
      procComms.Tick1s = false;
   }
   // -- Run the Communication Device Processors --
   BT_Processor();
   GSM_Processor();
   IPC_Processor();
   // -- Process received commands --
   if (procComms.CmdRx)
   {
      // -- Decode and check the packet for errors --
      if (Comms_CalculateCRC_16Bit(&commsBuffer[0], commsPacketLength, CRC_POLYNOMIAL16) == 0)
      {
         Comms_DecodePacket(&commsBuffer[0], &cmdPacket);
         cmdPacket.PacketSource = procComms.CmdSource;
         cmdPacket.Response = &commsBuffer[5];
         Cmd_Process(&cmdPacket);                                   // Process the received command (refer to 'Commands.c')
      }
      commsBufferPtr = 0;                                           // Point received data back to the start of the buffer
      procComms.CmdRx = false;                                      // Clear the command received flag so the next cmd can be processed
   }
}
// -- Comms: RxByte -----------------------------------------------------------------------------------------------------------------------------------------------------------------
// Called when a data byte 'value' is received via the communications interface 'source'. When a full packet is received the Comms Task is notified.
bool Comms_RxByte(unsigned char source, unsigned char value)
{
   TMyU16 temp;
   if ((commsBufferPtr == 0) & (value != 0x7E)) return false;       // Wait for a start of packet character ('~')
   if (commsBufferPtr >= COMMS_BUFF_SIZE)                           // Ensure we do not overflow the data buffer
   {
      commsBufferPtr = 0;
      return false;
   }
   LED_CommsRx = true;                                              // Indicate data transfer activity
   commsBuffer[commsBufferPtr++] = value;                           // Add the received byte to the data buffer
   if (commsBufferPtr == 3)
   {
      temp.Bytes[0] = commsBuffer[commsBufferPtr - 2]; temp.Bytes[1] = commsBuffer[commsBufferPtr - 1];
      commsPacketLength = temp.Value;                               // Specify the packet length based on received data structure
   }
   if ((commsBufferPtr > 2) && (commsBufferPtr >= commsPacketLength))
   {
      procComms.CmdSource = source;                                 // Specify where the data is coming from
```

```
      procComms.CmdRx = true;                                   // Indicate that a command packet has been received
      LED_CommsRx = false;                                      // Disable LED to show activity is complete
      return true;
   }
   return false;
}
// -- Comms: Respond: Returns a data packet to the source specified in 'packet'. A CRC is calculated and appended to the packet. ----------
void Comms_Respond(tCommsPacket *packet)
{
   unsigned char *buffPtr; unsigned short i; TMyU16 temp;
   LED_CommsTx = true;
   temp.Value = (*packet).MessLen + 7;                          // Determine packet size including header information
   *((*packet).BufferAddr + 1) = temp.Bytes[0];                 // The total packet size is placed in the return buffer
   *((*packet).BufferAddr + 2) = temp.Bytes[1];
   buffPtr = (*packet).BufferAddr;
   // -- Calculate the 16-bit CRC value for later sending with the packet --
   temp.Value = Comms_CalculateCRC_16Bit(buffPtr, (*packet).MessLen + 5, CRC_POLYNOMIAL16);
   switch ((*packet).PacketSource) {                            // Return the result to the source specified in packet
      case CMDSRC_BT:                                           // Source: Bluetooth
         for (i=0;i<(*packet).MessLen + 5;i++) BT_ByteTx(*buffPtr++);   // Transmit packet
         BT_ByteTx(temp.Bytes[1]); BT_ByteTx(temp.Bytes[0]);    // Transmit CRC
      break;
      case CMDSRC_GSM:                                          // Source: GSM
         for (i=0;i<(*packet).MessLen + 5;i++) GSM_ByteTx(*buffPtr++);  // Transmit packet
         GSM_ByteTx(temp.Bytes[1]); GSM_ByteTx(temp.Bytes[0]);  // Transmit CRC
      break;
      case CMDSRC_USB:                                          // Source : USB
         *(buffPtr + (*packet).MessLen + 5) = temp.Bytes[1];    // Append CRC to packet
         *(buffPtr + (*packet).MessLen + 6) = temp.Bytes[0];
         IPC_Message(IPCCMD_SENDUSBPACKET, buffPtr, (*packet).MessLen + 7, 0);          // Send to PIC18F processor as a command
      break;
   };
   LED_CommsTx = false;
}
// -- Comms: CalculateCRC_16Bit: Calculates a 16-bit CRC value using 'polynomial'. ----------
unsigned short Comms_CalculateCRC_16Bit(unsigned char *buffer, unsigned short length, unsigned short polynomial)
{
   unsigned char crcHigh, crcLow, carry, crcDat1; unsigned short a, b; TMyU16 crcPolynomial;
   crcPolynomial.Value = polynomial;
   crcHigh = *buffer;
   if (length > 1) crcLow = *(buffer+1); else crcLow = 0;
   for (a=1;a<(length+1);a++)
   {
      if ((length > 2) && (a < (length - 1))) crcDat1 = *(buffer+a+1); else crcDat1 = 0;
      for (b=0;b<8;b++)
      {
         carry = (crcHigh & 0x80) >> 7;
         if (carry > 0) {  crcHigh ^= crcPolynomial.Bytes[1]; crcLow ^= crcPolynomial.Bytes[0];  }
         crcHigh <<= 1; crcHigh += (crcLow & 0x80) >> 7;
         crcLow <<= 1; crcLow += (crcDat1 & 0x80) >> 7;
         crcDat1 <<= 1;
      }
   }
   a = crcHigh; a <<= 8; a += crcLow;
   return a;
}
// -- Comms: DecodePacket: Represents the received data buffer in a tCommsPacket structure for simpler data access. ----------
void Comms_DecodePacket(unsigned char *buffer, tCommsPacket *result)
{
   TMyU16 temp;
   (*result).BufferAddr = buffer;
   buffer++;
   temp.Bytes[0] = *buffer++; temp.Bytes[1] = *buffer++;
   (*result).MessLen = temp.Value - 7;                          // Message length
   (*result).TID = *buffer++;                                   // Transaction ID
   (*result).Cmd = *buffer++;                                   // Command
   (*result).Params = buffer;                                   // Pointer to Parameters in received data
}
```

```
/*==  PROCCONTROL.H & PROCCONTROL.C  ==========================================================================
Implements the control functionality of the Power Profiler.
=============================================================================================================*/
#include "HardwareProfile.h"            #include "Timekeeping.h"
// -- Data Type: tprocControl: Holds the flags representing the state of the Power Profiler Control Task ----------
typedef union {
```

```
    struct __attribute__ ((packed)) {
        unsigned Tick100ms      : 1;    unsigned SecTick        : 1;    unsigned CheckStatus    : 1;
        unsigned ShutdownSet    : 1;    unsigned Unused1        : 4;    unsigned Unused2        : 7;
        unsigned Unused3        : 1;    unsigned ProfilerMode   : 2;    unsigned VoltageMode    : 4;
        unsigned TODValid       : 1;    unsigned Unused4        : 1;    unsigned LCDBacklightOn : 1;
        unsigned RCPowerOn      : 1;    unsigned RFPowerOn      : 1;    unsigned LED_ACOn       : 1;
        unsigned LED_Fault      : 1;    unsigned LED_LinkActive : 1;    unsigned BatteryOn      : 1;
        unsigned ChargerOn      : 1;
    };
    unsigned char Bytes[4];
    unsigned long Value;
} tprocControl;
extern tprocControl procControl;
// -- Data Type: tprocErrors: Hardware and software error checking flags ----------
typedef union {
    struct __attribute__ ((packed)) {
        struct __attribute__ ((packed)) {
            unsigned ClockFault     : 1;    unsigned CommsFault     : 1;    unsigned MeasFault      : 1;
            unsigned MemFault       : 1;    unsigned VRegFault      : 1;    unsigned VRCFault       : 1;
            unsigned VRFFault       : 1;    unsigned SupplyLow      : 1;
    } HW;
        struct __attribute__ ((packed)) {
            unsigned DevCfgInvalid  : 1;    unsigned RTCLostTime    : 1;    unsigned CommsCfgFault  : 1;
            unsigned MeasCfgFault   : 1;    unsigned Unused         : 4;
        } SW;
    };
    unsigned short Value;
} tprocErrors;
extern tprocErrors procErrors;
// -- Data Type: tPIC18FControlStatus: Represents the state of the PIC18F processor status flags ----------
typedef union {
    struct {
        struct  __attribute__ ((packed)) {
            unsigned Fault          : 1;    unsigned Tick100ms      : 1;    unsigned VSupplyLow     : 1;
            unsigned VBattLow       : 1;    unsigned VRegFault      : 1;    unsigned VRCFault       : 1;
            unsigned VRFFault       : 1;    unsigned Updated        : 1;
        };
        unsigned char Unused;
        float VoltageSupply, BatteryVoltage;
    };
    unsigned char Bytes[10];
} tPIC18FControlStatus;
extern TMyU32 deviceID;
extern unsigned char password[16];

//== PROCCONTROL.C ============================================================================================
#include "procControl.h"        #include "procMeasurement.h"    #include "Timekeeping.h"       #include "procMemory.h"
#include "Comms_IPC.h"          #include "Comms_GSM.h"          #include "Comms_Bluetooth.h"
unsigned char password[16], statusTick = 0;
float vSupply, vBattery, vBattThreshold;
TMyU32 deviceID;
tprocControl procControl;
tprocErrors procErrors;
// -- ProcControl_Initialize: Retrieves the configuration information and Profiler ID and configures the Real-Time-Clock ----------
void ProcCtrl_Initialize(void)
{
    unsigned char j = 0;
    procControl.Value = 0;          procErrors.Value = 0;
    vBattThreshold = 6.1;
    tcfgProfilerID idBlock;                                     // -- Retrieve Device Identity Block. If not set report an error. --
    CfgMem_Read(CFG_DEVICEID_ADDR, 1, &idBlock.b[0]);
    if (idBlock.v.Verifier == 0xBBAA)
    {
        deviceID.Value = idBlock.v.DeviceID;
        for (j=0;j<16;j++) password[j] = idBlock.v.Password[j];
    }
    else
    {
        deviceID.Value = 0;
        for (j=0;j<16;j++) password[j] = 0;
        procErrors.SW.DevCfgInvalid = true;
    }
    procErrors.HW.ClockFault = (Clock_Initialize() == false);
    procControl.LCDBacklightOn = true;
}
```

```
// -- ProcControl_Processor: Updates the timing ticks and checks the state of the Power Profiler. ----------
void ProcCtrl_Processor(void)
{
  if (procControl.Tick100ms)
  {
    // -- Update the measurement values on the PIC18F processor every 800ms --
    if (statusTick++ > 7)
    {
      ipcStatus.UpdateMeas = true;
      procControl.CheckStatus = true;
      statusTick = 0;
    }
    procControl.Tick100ms = false;
  }
  if (procControl.SecTick)
  {
    Clock_Increment();
    procControl.SecTick = False;
  }
  if (clockStatus.HourPassed)
  {
    procMeas.HourTick = true;
    clockStatus.HourPassed = false;
  }
  if (procControl.CheckStatus)
  {
    // -- Update error flags --
    procErrors.HW.CommsFault = procComms.HWFault;          procErrors.SW.CommsCfgFault = procComms.CfgFault;
    procErrors.HW.MeasFault = procMeas.HWFault;            procErrors.SW.MeasCfgFault = procMeas.CfgFault;
    procErrors.HW.MemFault = memoryStatus.HWFault;
    // -- Update Profiler status --
    procControl.TODValid = clockStatus.TODValid;           procControl.LED_ACOn = PORTAbits.RA6;
    procControl.LED_Fault = procErrors.Value > 0;          procControl.LED_LinkActive = (btStatus.Connected | gsmStatus.DataMode);
    procControl.ProfilerMode = 0;                          // Not currently used
    procControl.VoltageMode = procMeasCfg.v.VoltageInput;  procControl.RCPowerOn = gsmStatus.Enabled;
    procControl.RFPowerOn = btStatus.Enabled;              procControl.LED_ACOn = true;
    // -- Update power control --
    if (procControl.LED_ACOn)
    {
      procControl.BatteryOn = false;
      procControl.ChargerOn = true;
    }
    else
    {
      procControl.ChargerOn = false;
      procControl.BatteryOn = true;
      if ((procErrors.HW.SupplyLow) && (!procControl.ShutdownSet))
      {
        StorMem_SetMemPointers();                          // Store measurement parameters and disconnect the battery to shutdown the Profiler
        procControl.BatteryOn = false;
        procControl.ShutdownSet = true;
      }
    }
    procControl.CheckStatus = false;
  }
}
// -- ProcControl_PIC18FStatusUpdate: Decodes the PIC18F status retrieved from an IPC request and checks for reported errors. ----------
bool ProcCtrl_PIC18FStatusUpdate(unsigned char *buffer, unsigned char length)
{
  int i;
  tPIC18FControlStatus temp;
  if (length != 10) return false;
  for (i=0;i<10;i++) temp.Bytes[i] = *buffer++;
  temp.Updated = true;
  procErrors.HW.VRegFault = temp.VRegFault;              procErrors.HW.VRCFault = temp.VRCFault;
  procErrors.HW.VRFFault = temp.VRFFault;                procErrors.HW.SupplyLow = temp.VSupplyLow;
  vSupply = temp.VoltageSupply;                          vBattery = temp.BatteryVoltage;
  return true;
}
```

```
/*==  PROCMEASUREMENT.H & PROCMEASUREMENT.C  ==========================================================
Measurement management software task.
=============================================================================================================*/
#include "HardwareProfile.h"
// -- Constant and Type Definitions ----------
```

```
// -- tprocMeasStatus: State information for the Measurement Task. ----------
typedef union {
   struct __attribute__ ((packed)) {
      unsigned HWFault          : 1;      unsigned CfgFault         : 1;
      unsigned CalibFault       : 1;      unsigned EM1OK            : 1;
      unsigned EM2OK            : 1;      unsigned EM3OK            : 1;
      unsigned EM4OK            : 1;      unsigned EM1Config        : 1;
      unsigned EM2Config        : 1;      unsigned EM3Config        : 1;
      unsigned EM4Config        : 1;      unsigned VMuxConfig       : 1;
      unsigned TempSensOk       : 1;      unsigned Tick100ms        : 1;
      unsigned SecTick          : 1;      unsigned HourTick         : 1;
      unsigned DoMeasure        : 1;      unsigned DoAccumulate     : 1;
      unsigned UpdateTemp       : 1;      unsigned DoAcquire        : 1;
      unsigned Unused           : 12;
   };
   unsigned long Value;
} tprocMeasStatus;
// -- tprocMeasConfig: Process measurement configuration information decoded from configuration memory. ----------
typedef union {
   struct __attribute__ ((packed)) {
      unsigned ConfigValid      : 1;      unsigned EnergyIC1Enabled : 1;
      unsigned EnergyIC2Enabled : 1;      unsigned EnergyIC3Enabled : 1;
      unsigned EnergyIC4Enabled : 1;      unsigned VoltageInput     : 2;
      unsigned ChannelsEnabled  : 12;     unsigned IndustrialInputs : 8;
      unsigned AutoCapture      : 1;      unsigned Unused           : 4;
 } v;
 unsigned long l __attribute__ ((packed));
 unsigned char b[4] __attribute__ ((packed));
} tprocMeasConfig;
// -- Data Type: tcurrentMeasurements: Current measurement values returned from all four energy measurement ICs and the industrial input channels. ----------
typedef union {
   struct __attribute__ ((packed)) {
      signed long Voltage[12], Current[12];     signed short Watts[12], VAR[12], VA[12];
      unsigned char Temperature;                unsigned short Frequency[4];
      unsigned short IndustrialInputs[8];
   };
   unsigned char Bytes[195] __attribute__ ((packed));
} tcurrentMeasurements;
// -- Data Type: tMeasAccumulation: Used to accumulate and calculate max and min for measurement samples. ----------
typedef union {
   struct {
      unsigned short Measurements;              signed long VoltageMax[3], VoltageMin[3];
      signed long long VoltageAvg[3];           signed long CurrentMax[12], CurrentMin[12];
      signed long long CurrentAvg[12];          signed short WattsMax[12], WattsMin[12];
      signed long WattsAvg[12];                 signed short VARMax[12], VARMin[12];
      signed long VARAvg[12];                   signed short VAMax[12], VAMin[12];
      signed long VAAvg[12];                    unsigned char TempMax, TempMin;
      unsigned short TempAvg;                   unsigned short FreqMax, FreqMin;
      unsigned long FreqAvg;                    unsigned short IndustrialIPMax[8], IndustrialIPMin[8];
      unsigned long IndustrialIPAvg[8];
   } v;
   unsigned char bytes[598];
} tMeasAccumulation;
// -- Data Type: tProfileSettings: Contains secs, mins, hrs profile settings retrieved from configuration memory. ----------
typedef union {
   struct  __attribute__ ((packed)) {
      unsigned Enabled          : 1;      unsigned SyncTOD          : 1;
      unsigned WrapMemory       : 1;      unsigned TODTrigger       : 1;
      unsigned Unused           : 4;      unsigned char Interval, IntervalCount;
      unsigned long MeasTotal, MeasCount;
   } v;
   unsigned char b[11]  __attribute__ ((packed));
} tProfileSettings;

// ==  PROCMEASUREMENT.C  ==========================================================================================
#include "procMeasurement.h"  #include "I2CSPIComms.h"  #include "ADE7758.h"  #include "dsPICSupport.h"  #include "I2CDevices.h"
#include "procMemory.h"  #include "procControl.h"  #include "Timekeeping.h"
// -- Global Variables ----------
bool energyICEnabled[4] = { 0, 0, 0, 0 }; bool channelEnabled[12] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }; bool indIPEnabled[8] = { 0, 0, 0, 0, 0, 0, 0, 0 };
bool wrapMemory[3] = { 1, 1, 1 };
unsigned char measureTick = 0, secTickCount = 0, indipChannel = 0;
unsigned char voltageChannel[3] = { 0, 1, 2 };
unsigned short measurementSize = 0;
tprocMeasStatus procMeas;
tprocMeasConfig procMeasCfg;
tProfileSettings secsProfile, minsProfile, hrsProfile;
```

```
tMeasAccumulation measAccumulation[3];
// ==  TASK: MEASUREMENT  ============================================================================================
// -- ProcMeasurement: Initialize: Initializes the measurement devices and state machine. ----------------------------------------------------------------------
unsigned char procMeas_Initialize(void)
{
   unsigned char i;
   procMeas.Value = 0; procMeasCfg.I = 0;                                  // Defaults for process measurement configuration
   ADC_Initialize();                                                       // Initialize the Analogue Sampling
   procMeas.TempSensOk = I2CTemp_Initialize(0x90);                         // Initialize Temperature Sensor
   I2CTemp_SetPointer(0);
   // -- Initialize Energy Measurement ICs --
   procMeas.EM1OK = ADE7758_Check(1); procMeas.EM2OK = ADE7758_Check(2);
   procMeas.EM3OK = ADE7758_Check(3); procMeas.EM4OK = ADE7758_Check(4);
   procMeas_InitializeCapture(false);
   procMeas.HWFault = !(procMeas.EM1OK & procMeas.EM2OK & procMeas.EM3OK & procMeas.EM4OK & procMeas.TempSensOk);
   return procMeas.HWFault;
}
// -- ProcMeasurement: Processor: Executes software tasks to implement the Measurement state machine. ----------------------------------------------------------
void procMeas_Processor(void)
{
   unsigned char i, *Buffer; TMyU16 tempValue;
   // -- Check to process measurement data on every loop ---------------------------------------------------------------------------------------------------------
   if (procMeas.DoAcquire)
   {
      if (procMeas.DoAccumulate)
      {
         LED_Acquisition = true;
         if (secsProfile.v.Enabled)                                        // -- Seconds Profiling --
         {
            procMeas_Accumulate(0);
            if (measAccumulation[0].v.Measurements >= secsProfile.v.MeasTotal)
               procMeas_AverageAndStore(0);
         }
         if (minsProfile.v.Enabled)                                        // -- Minutes Profiling --
         {
            procMeas_Accumulate(1);
            if (measAccumulation[1].v.Measurements >= minsProfile.v.MeasTotal)
               procMeas_AverageAndStore(1);
         }
         if (hrsProfile.v.Enabled)                                         // -- Hours Profiling --
         {
            procMeas_Accumulate(2);
            if (!hrsProfile.v.SyncTOD)
            {
               if (measAccumulation[2].v.Measurements >= hrsProfile.v.MeasTotal)
                  procMeas_AverageAndStore(2);
            }
         }
         LED_Acquisition = false;
         procMeas.DoAccumulate = false;
      }
      if (procMeas.HourTick)                                               // The hour profiling can can be synchronized with the on-hour roll-over
      {
         if (hrsProfile.v.Enabled && hrsProfile.v.SyncTOD)                 // If enabled, synchronize the hours capture to an hour rollover of the RTC
         {
            hrsProfile.v.IntervalCount++;
            if (hrsProfile.v.IntervalCount >= hrsProfile.v.Interval)
            {
               LED_Acquisition = true;
               hrsProfile.v.IntervalCount = 0;
               procMeas_AverageAndStore(2);
               LED_Acquisition = false;
            }
         }
         StorMem_SetMemPointers();                                         // -- Update dataFLASH pointers in FLASH --
         procMeas.HourTick = false;
      }
   }
   // -- 100ms Tasks ---------------------------------------------------------------------------------------------------------------------------------------------
   if (procMeas.Tick100ms)
   {
      if (measureTick++ == 1)
      {
         // -- Acquire the data every 200ms --
         procMeas_Update();
         procMeas.DoAccumulate = true;
```

```
            measureTick = 0;
        }
        if (secTickCount++ == 9)
        {
            procMeas.SecTick = true;
            secTickCount = 0;
        }
        procMeas.Tick100ms = false;
    }
    // -- One Second Tasks ------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
    if (procMeas.SecTick)
    {
        procMeas.UpdateTemp = true;
        procMeas.SecTick = false;
    }
    // -- The temperature value is retrieved from the TCN75 every second --------------------------------------------------------------------------------------------------------------------------------
    if (procMeas.UpdateTemp)
    {
        if (I2CTemp_Temperature(&tempValue)) currentMeasurements.Temperature = tempValue.Bytes[1];
        procMeas.UpdateTemp = false;
    }
}
// -- procMeasurement: InitializeCapture: Retrieves the settings from configuration memory for measurement control. ------------------------------------------------------------------------
void procMeas_InitializeCapture(bool restartCapture)
{
    unsigned char i, vMux[6];  unsigned short channelValue;  tcfgMeasurement cfgMeas;  tcfgMemory cfgMemory;
    // -- Initialize measurement profiles --
    secsProfile.v.Enabled = false; measAccumulation[0].v.Measurements = 0;
    minsProfile.v.Enabled = false; measAccumulation[1].v.Measurements = 0;
    hrsProfile.v.Enabled = false; measAccumulation[2].v.Measurements = 0;
    // -- Load configuration from memory --
    CfgMem_Read(CFG_MEASCFG_ADDR, 1, &cfgMeas.b[0]);
    if (cfgMeas.v.Verifier == 0xBBAA)
    {
        procMeasCfg.l = cfgMeas.v.ConfigFlags;
        // -- Configure voltage inputs --
        switch (procMeasCfg.v.VoltageInput) {
            case 0: // 1 Phase 2-Wire 1, 2, 3 = L1, N = N
                    vMux[0] = 0x81; vMux[1] = 0x81; vMux[2] = 0x81; vMux[3] = 0x81; vMux[4] = 0x81; vMux[5] = 0x81;
                    VMux_SetChannels(&vMux[0]);
                    break;
            case 1: // 3-Phase 3-Wire (Delta) 1 = L1, 2 = L2, 3 = L3, N = N
                    vMux[0] = 0x22; vMux[1] = 0x84; vMux[2] = 0x41; vMux[3] = 0x22; vMux[4] = 0x84; vMux[5] = 0x41;
                    VMux_SetChannels(&vMux[0]);
                    break;
            case 2: // 3-Phase 4-Wire (Star) 1 = L1, 2 = L2, 3 = L3, N = N
                    vMux[0] = 0x22; vMux[1] = 0x84; vMux[2] = 0x41; vMux[3] = 0x22; vMux[4] = 0x84; vMux[5] = 0x41;
                    VMux_SetChannels(&vMux[0]);
                    break;
            case 3: // Custom configuration
                    VMux_SetChannels(&cfgMeas.v.CustomVoltageMux[0]);
                    break;
        };
        // -- Update channel enabling for quick access during accumulation --
        channelValue = procMeasCfg.v.ChannelsEnabled;
        for (i=0; i<12; i++) channelEnabled[i] = U16BitTest(channelValue, i);
        channelValue = procMeasCfg.v.IndustrialInputs;
        for (i=0; i<8; i++) indIPEnabled[i] = U16BitTest(channelValue, i);
        procMeasCfg.v.ConfigValid = procMeas_SetEnergyICMode(procMeasCfg.v.VoltageInput);
        // -- Update profile capture settings --
        secsProfile.b[0] = cfgMeas.v.SecsProfileFlags;
        secsProfile.v.Interval = cfgMeas.v.SecsProfileInterval;
        secsProfile.v.MeasTotal = secsProfile.v.Interval * 5;                    // There are 5 measurements captured in one second
        secsProfile.v.MeasCount = 0;
        wrapMemory[0] = secsProfile.v.WrapMemory;
        minsProfile.b[0] = cfgMeas.v.MinsProfileFlags;
        minsProfile.v.Interval = cfgMeas.v.MinsProfileInterval;
        minsProfile.v.MeasTotal = minsProfile.v.Interval * 300;                  // There are 300 measurements captured in one minute
        wrapMemory[1] = minsProfile.v.WrapMemory;
        hrsProfile.b[0] = cfgMeas.v.HrsProfileFlags;
        hrsProfile.v.Interval = cfgMeas.v.HrsProfileInterval;
        wrapMemory[2] = hrsProfile.v.WrapMemory;
        if (hrsProfile.v.SyncTOD) hrsProfile.v.IntervalCount = 0;
        procMeas.UpdateTemp = true;
        procMeas.DoAcquire = procMeasCfg.v.AutoCapture;
        procMeas.CfgFault = procMeasCfg.v.ConfigValid;
```

```
    }
    else
    {
        procMeasCfg.v.ConfigValid = false;
        procMeas.CfgFault = true;
    }
    if (procMeasCfg.v.EnergyIC1Enabled)
    {
        procMeas.EM1Config = procMeas_ConfigureEnergyIC(1, CFG_ENERGYIC1REG_ADDR);
        energyICEnabled[0] = true;
    }
    else
        ADE7758_Disable(1);
    if (procMeasCfg.v.EnergyIC2Enabled)
    {
        procMeas.EM1Config = procMeas_ConfigureEnergyIC(2, CFG_ENERGYIC2REG_ADDR);
        energyICEnabled[1] = true;
    }
    else
        ADE7758_Disable(2);
    if (procMeasCfg.v.EnergyIC3Enabled)
    {
        procMeas.EM1Config = procMeas_ConfigureEnergyIC(3, CFG_ENERGYIC3REG_ADDR);
        energyICEnabled[2] = true;
    }
    else
        ADE7758_Disable(3);
    if (procMeasCfg.v.EnergyIC4Enabled)
    {
        procMeas.EM1Config = procMeas_ConfigureEnergyIC(4, CFG_ENERGYIC4REG_ADDR);
        energyICEnabled[3] = true;
    }
    else
        ADE7758_Disable(4);
    // -- Load memory mapping --
    CfgMem_Read(CFG_MEMORYCFG_ADDR, 1, &cfgMemory.b[0]);
    if (cfgMemory.v.Verifier == 0xBBAA)
    {
        if ((dataFLASHPtr.v.Verifier != 0xBBAA) || (restartCapture))
        {
            // Initialize the memory registers
            dataFLASHPtr.v.TOD = 0;
            // Seconds
            dataFLASHPtr.v.SecsStartAddr = cfgMemory.v.SecsFLASHStart; dataFLASHPtr.v.SecsStopAddr = cfgMemory.v.MinsFLASHStart - 1;
            dataFLASHPtr.v.SecsWrPtr = dataFLASHPtr.v.SecsStartAddr; dataFLASHPtr.v.SecsRdPtr = dataFLASHPtr.v.SecsStartAddr;
            dataFLASHPtr.v.SecsMemFree = cfgMemory.v.MinsFLASHStart - cfgMemory.v.SecsFLASHStart;
            // Minutes
            dataFLASHPtr.v.MinsStartAddr = cfgMemory.v.MinsFLASHStart; dataFLASHPtr.v.MinsStopAddr = cfgMemory.v.HrsFLASHStart - 1;
            dataFLASHPtr.v.MinsWrPtr = dataFLASHPtr.v.MinsStartAddr; dataFLASHPtr.v.MinsRdPtr = dataFLASHPtr.v.MinsStartAddr;
            dataFLASHPtr.v.MinsMemFree = cfgMemory.v.HrsFLASHStart - cfgMemory.v.MinsFLASHStart;
            // Hours
            dataFLASHPtr.v.HrsStartAddr = cfgMemory.v.HrsFLASHStart; dataFLASHPtr.v.HrsStopAddr = cfgMemory.v.EventsFLASHStart - 1;
            dataFLASHPtr.v.HrsWrPtr = dataFLASHPtr.v.HrsStartAddr; dataFLASHPtr.v.HrsRdPtr = dataFLASHPtr.v.HrsStartAddr;
            dataFLASHPtr.v.HrsMemFree = cfgMemory.v.EventsFLASHStart - cfgMemory.v.HrsFLASHStart;
            StorMem_SetMemPointers();
        }
        currRdPtr_Secs = dataFLASHPtr.v.SecsRdPtr; currRdPtr_Mins = dataFLASHPtr.v.MinsRdPtr; currRdPtr_Hrs = dataFLASHPtr.v.HrsRdPtr;
    }
    else
    {
        procMeas.CfgFault = true;                                                    // -- Mark memory problem and disable storage --
    }
}
// -- procMeas: SetEnergyICMode: Retrieves and sets the ADE7758 operational mode registers from configuration memory. ----------------------------------------------
unsigned char procMeas_SetEnergyICMode(unsigned char mode)
{
    int i;  tcfgMeasurementCal calib;
    switch (mode) {
        case 0 : CfgMem_Read(CFG_1PH2WCALIB_ADDR, 7, &calib.b[0]); break;
        case 1 : CfgMem_Read(CFG_3PH3WCALIB_ADDR, 7, &calib.b[0]); break;
        case 2 : CfgMem_Read(CFG_3PH4WCALIB_ADDR, 7, &calib.b[0]); break;
        case 3 : CfgMem_Read(CFG_CUSTOMCALIB_ADDR, 7, &calib.b[0]); break;
    };
    if (calib.v.Verifier == 0xBBAA)
    {
        for (i=0; i<4; i++) ADE7758_SetControlRegisters(i, &calib.b[210]);
```

```
      return true;
   }
   else
      procMeas.CalibFault = true;
   return false;
}
// -- procMeas: ConfigureEnergyIC: Retrieves the calibration parameters from configuration memory and sets the specified ADE7758. ----------
unsigned char procMeas_ConfigureEnergyIC(unsigned char device, int eeAddr)
{
   unsigned char i;  tcfgEMICCalibration emICCalib;  tADE7758Calibration emCal;
   // -- To configure an energy measurement IC we need to retrieve the config parameters from internal EEPROM and then program the internal registers. --
   CfgMem_Read(eeAddr, 2, &emICCalib.b[0]);
   if (emICCalib.v.Verifier == 0xBBAA)
   {
      for (i=0;i<61;i++) emCal.b[i] = emICCalib.b[i + 2];
      ADE7758_SetCalibration(device, emCal);
      return true;
   }
   else
      procMeas.CalibFault = true;
   return false;
}
// -- Voltage Multiplexer : SetChannels: Passes the 6 channel control values specified in 'Buffer' to the MAX350s. ----------
void VMux_SetChannels(unsigned char *buffer)
{
   unsigned char i;
   SPI1_DeselectAll();                                                    // Disable all SPI devices and configure SPI port
   SPI1_Configure(0, spiMaster + spiDOutActiveToIdle + spiSecPresc_3_1 + spiPriPresc_4_1, 0);
   SelReg_Clear(CS_VMUX);                                                 // Enable Chip Select for MAX350s
   for (i=0;i<6;i++) SPI1_TxByte(*buffer++);                              // Send each channel value to the MAX350
   SelReg_Set(CS_VMUX);
   SPI1_DeselectAll();                                                    // Disable all SPI devices and configure SPI port
   SPI1_Configure(0, spiMaster + spiDOutIdleToActive + spiSecPresc_3_1 + spiPriPresc_4_1, 0);
   SPI1_TxByte(0xFF);                                                     // Dummy byte to get port working
}
// == MEASUREMENT ACQUISITION ==========================================================================================
// -- Global Variables ----------------------------------------------------------------------------------------------------
tcurrentMeasurements currentMeasurements;
unsigned char FreqEMIC = 0;
// -- procMeas_Update: Reads the measurement values from the ADE7758s and updates currentMeasurements. ----------
void procMeas_Update(void)
{
   unsigned char i, j, idx = 0;  tADE7758Measurement emValue;
   for (i=0; i<4; i++)                                                    // -- Acquire the data --
   {
      if (energyICEnabled[i])
      {
         ADE7758_GetMeasurement(i + 1, &emValue);                         // -- Get measurement values and update currentMeasurements --
         for (j=0; j<3; j++)
         {
            currentMeasurements.Voltage[idx] = emValue.v.VRMS[j];
            currentMeasurements.Current[idx] = emValue.v.IRMS[j];
            currentMeasurements.Watts[idx] = emValue.v.Watts[j];
            currentMeasurements.VAR[idx] = emValue.v.VAR[j];
            currentMeasurements.VA[idx] = emValue.v.VA[j];
            idx++;
         }
         currentMeasurements.Frequency[i] = emValue.v.Frequency;
      }
      else
         idx += 3;
   }
}
// -- procMeas_ResetAccumulation: Resets the measurement count so that on the next procMeas_Accumulate the values will be initialized. ----------
void procMeas_ResetAccumulation(void)
{  measAccumulation[0].v.Measurements = 0; measAccumulation[1].v.Measurements = 0; measAccumulation[2].v.Measurements = 0;  }
// -- MeasAccum_S32: Updates the max, min and accumulation registers for a signed 32-bit number 'value'. ----------
void MeasAccum_S32(signed long value, signed long *min, signed long long *avg, signed long *max)
{  *avg += value;  if (value > *max) *max = value;  if (value < *min) *min = value;  }
// -- MeasAccum_S16: Updates the max, min and accumulation registers for a signed 16-bit number 'value'. ----------
void MeasAccum_S16(signed short value, signed short *min, signed long *avg, signed short *max)
{  *avg += value;  if (value > *max) *max = value;  if (value < *min) *min = value;  }
// -- MeasAccum_U16: Updates the max, min and accumulation registers for a unsigned 16-bit number 'value'. ----------
void MeasAccum_U16(unsigned short value, unsigned short *min, unsigned long *avg, unsigned short *max)
{  *avg += value;  if (value > *max) *max = value;  if (value < *min) *min = value;  }
// -- MeasAccum_U8: Updates the max, min and accumulation registers for a unsigned byte 'value'. ----------
```

```
void MeasAccum_U8(unsigned char value, unsigned char *min, unsigned short *avg, unsigned char *max)
{ *avg += value;  if (value > *max) *max = value;  if (value < *min) *min = value;  }
// -- procMeas_Accumulate ----------
// Initializes the max, min and avg values for the profile 'value' if measurement count is zero, otherwise update with the value in 'currentMeasurements'.
void procMeas_Accumulate(unsigned char value)
{
   unsigned char i;
   if (measAccumulation[value].v.Measurements == 0)
   {
      // -- Initialize the voltage channel Max, Min, Avg ----------
      for (i = 0; i < 3; i++)
      {
         measAccumulation[value].v.VoltageMax[i] = currentMeasurements.Voltage[voltageChannel[i]];
         measAccumulation[value].v.VoltageMin[i] = currentMeasurements.Voltage[voltageChannel[i]];
         measAccumulation[value].v.VoltageAvg[i] = currentMeasurements.Voltage[voltageChannel[i]];
      }
      // -- Initialize the current channel Max, Min, Avg ----------
      for (i=0;i<12;i++)
         if (channelEnabled[i])
         {
            measAccumulation[value].v.CurrentMax[i] = currentMeasurements.Current[i];
            measAccumulation[value].v.CurrentMin[i] = currentMeasurements.Current[i];
            measAccumulation[value].v.CurrentAvg[i] = currentMeasurements.Current[i];
            measAccumulation[value].v.WattsMax[i] = currentMeasurements.Watts[i];
            measAccumulation[value].v.WattsMin[i] = currentMeasurements.Watts[i];
            measAccumulation[value].v.WattsAvg[i] = currentMeasurements.Watts[i];
            measAccumulation[value].v.VARMax[i] = currentMeasurements.VAR[i];
            measAccumulation[value].v.VARMin[i] = currentMeasurements.VAR[i];
            measAccumulation[value].v.VARAvg[i] = currentMeasurements.VAR[i];
            measAccumulation[value].v.VAMax[i] = currentMeasurements.VA[i];
            measAccumulation[value].v.VAMin[i] = currentMeasurements.VA[i];
            measAccumulation[value].v.VAAvg[i] = currentMeasurements.VA[i];
         }
      // -- Initialize the industrial input channels ----------
      for (i=0;i<8;i++)
         if (indIPEnabled[i])
         {
            measAccumulation[value].v.IndustrialIPMax[i] = currentMeasurements.IndustrialInputs[i];
            measAccumulation[value].v.IndustrialIPMin[i] = currentMeasurements.IndustrialInputs[i];
            measAccumulation[value].v.IndustrialIPAvg[i] = currentMeasurements.IndustrialInputs[i];
         }
      // -- Initialize the frequency channel ----------
      measAccumulation[value].v.FreqMax = currentMeasurements.Frequency[0];
      measAccumulation[value].v.FreqMin = currentMeasurements.Frequency[0];
      measAccumulation[value].v.FreqAvg = currentMeasurements.Frequency[0];
      // -- Initialize the temperature channel ----------
      measAccumulation[value].v.TempMax = currentMeasurements.Temperature;
      measAccumulation[value].v.TempMin = currentMeasurements.Temperature;
      measAccumulation[value].v.TempAvg = currentMeasurements.Temperature;
   }
   else
   {
      // -- Update voltage channel measurements ----------
      switch (procMeasCfg.v.VoltageInput)
      {
         case 0: // 1 Phase 2-Wire 1, 2, 3 = L1, N = N
                  MeasAccum_S32(currentMeasurements.Voltage[0], &measAccumulation[value].v.VoltageMin[0],
                     &measAccumulation[value].v.VoltageAvg[0], &measAccumulation[value].v.VoltageMax[0]);
                  break;
         case 1: // 3-Phase 3-Wire (Delta) 1 = L1, 2 = L2, 3 = L3, N = N
                  for (i=0;i<2;i++)
                     MeasAccum_S32(currentMeasurements.Voltage[i], &measAccumulation[value].v.VoltageMin[i],
                           &measAccumulation[value].v.VoltageAvg[i], &measAccumulation[value].v.VoltageMax[i]);
                  break;
         case 2: // 3-Phase 4-Wire (Star) 1 = L1, 2 = L2, 3 = L3, N = N
                  for (i=0;i<3;i++)
                     MeasAccum_S32(currentMeasurements.Voltage[i], &measAccumulation[value].v.VoltageMin[i],
                           &measAccumulation[value].v.VoltageAvg[i], &measAccumulation[value].v.VoltageMax[i]);
                  break;
         case 3: // Custom configuration
                  for (i=0;i<3;i++)
                     MeasAccum_S32(currentMeasurements.Voltage[i], &measAccumulation[value].v.VoltageMin[i],
                           &measAccumulation[value].v.VoltageAvg[i], &measAccumulation[value].v.VoltageMax[i]);
                  break;
      };
      // -- Update current channel measurements ----------
```

```
    for (i=0;i<12;i++)
    {
      if (channelEnabled[i])
      {
        MeasAccum_S32(currentMeasurements.Current[i], &measAccumulation[value].v.CurrentMin[i],
          &measAccumulation[value].v.CurrentAvg[i], &measAccumulation[value].v.CurrentMax[i]);
        MeasAccum_S16(currentMeasurements.Watts[i], &measAccumulation[value].v.WattsMin[i],
          &measAccumulation[value].v.WattsAvg[i], &measAccumulation[value].v.WattsMax[i]);
        MeasAccum_S16(currentMeasurements.VAR[i], &measAccumulation[value].v.VARMin[i],
          &measAccumulation[value].v.VARAvg[i], &measAccumulation[value].v.VARMax[i]);
        MeasAccum_S16(currentMeasurements.VA[i], &measAccumulation[value].v.VAMin[i],
          &measAccumulation[value].v.VAAvg[i], &measAccumulation[value].v.VAMax[i]);
      }
    }
    // -- Update the industrial input channel measurement ----------
    for (i=0;i<8;i++)
    {
      if (indIPEnabled[i])
        MeasAccum_U16(currentMeasurements.IndustrialInputs[i],
          &measAccumulation[value].v.IndustrialIPMin[i], &measAccumulation[value].v.IndustrialIPAvg[i],
          &measAccumulation[value].v.IndustrialIPMax[i]);
    }
    // -- Update the frequency channel measurement ----------
    MeasAccum_U16(currentMeasurements.Frequency[0], &measAccumulation[value].v.FreqMin,
      &measAccumulation[value].v.FreqAvg, &measAccumulation[value].v.FreqMax);
    // -- Update the temperature channel measurement ----------
    MeasAccum_U8(currentMeasurements.Temperature, &measAccumulation[value].v.TempMin,
      &measAccumulation[value].v.TempAvg, &measAccumulation[value].v.TempMax);
  }
  measAccumulation[value].v.Measurements++;
}
// -- Add24BitToBuffer: Packs the 32-bit value into 'buffer' as a 24-bit representation for optimal memory usage. ----------
void Add24BitToBuffer(U8 *buffer, U16 *idx, U32 maxValue, U32 avgValue, U32 minValue)
{
  unsigned char i;  TMyU32 temp;
  buffer += *idx;
  temp.Value = maxValue; for (i=0; i<3; i++) *buffer++ = temp.Bytes[i];
  temp.Value = avgValue; for (i=0; i<3; i++) *buffer++ = temp.Bytes[i];
  temp.Value = minValue; for (i=0; i<3; i++) *buffer++ = temp.Bytes[i];
  *idx += 9;
}
// -- Add16BitToBuffer: Packs the 16-bit value into 'buffer'. ----------
void Add16BitToBuffer(U8 *buffer, U16 *idx, U16 maxValue, U16 avgValue, U16 minValue)
{
  unsigned char i; TMyU16 temp;
  buffer += *idx;
  temp.Value = maxValue; *buffer++ = temp.Bytes[0]; *buffer++ = temp.Bytes[1];
  temp.Value = avgValue; *buffer++ = temp.Bytes[0]; *buffer++ = temp.Bytes[1];
  temp.Value = minValue; *buffer++ = temp.Bytes[0]; *buffer++ = temp.Bytes[1];
  *idx += 6;
}
// -- procMeas: AverageAndStore:  Adds a time-of-day stamp to the measurement and packs all values into 'measBuffer' which is then written to storage memory.
void procMeas_AverageAndStore(unsigned char value)
{
  unsigned char i, measBuffer[413];  unsigned short idx;  TMyU32 tod;
  idx = 0;
  // -- Measurement Flags (currently not used) ----------
  measBuffer[idx++] = 0;
  // -- Current Time of Day ----------
  tod.Value = Clock_PackDateTime(currentTOD);
  measBuffer[idx++] = tod.Bytes[0]; measBuffer[idx++] = tod.Bytes[1]; measBuffer[idx++] = tod.Bytes[2]; measBuffer[idx++] = tod.Bytes[3];
  // -- Voltage measurements ----------
  switch (procMeasCfg.v.VoltageInput)
  {
    case 0: // 1 Phase 2-Wire 1, 2, 3 = L1, N = N
            measAccumulation[value].v.VoltageAvg[0] /= measAccumulation[value].v.Measurements;
            Add24BitToBuffer(&measBuffer[0], &idx, measAccumulation[value].v.VoltageMax[0],
              measAccumulation[value].v.VoltageAvg[0], measAccumulation[value].v.VoltageMin[0]);
            break;
    case 1: // 3-Phase 3-Wire (Delta) 1 = L1, 2 = L2, 3 = L3, N = N
            for (i=0;i<2;i++)
            {
              measAccumulation[value].v.VoltageAvg[i] /= measAccumulation[value].v.Measurements;
              Add24BitToBuffer(&measBuffer[0], &idx, measAccumulation[value].v.VoltageMax[i],
                measAccumulation[value].v.VoltageAvg[i], measAccumulation[value].v.VoltageMin[i]);
            }
```

```
                break;
        case 2: // 3-Phase 4-Wire (Star) 1 = L1, 2 = L2, 3 = L3, N = N
                for (i=0;i<3;i++)
                {
                   measAccumulation[value].v.VoltageAvg[i] /= measAccumulation[value].v.Measurements;
                   Add24BitToBuffer(&measBuffer[0], &idx, measAccumulation[value].v.VoltageMax[i],
                      measAccumulation[value].v.VoltageAvg[i], measAccumulation[value].v.VoltageMin[i]);
                }
                break;
        case 3: // Custom configuration
                for (i=0;i<3;i++)
                {
                   measAccumulation[value].v.VoltageAvg[i] /= measAccumulation[value].v.Measurements;
                   Add24BitToBuffer(&measBuffer[0], &idx, measAccumulation[value].v.VoltageMax[i],
                      measAccumulation[value].v.VoltageAvg[i], measAccumulation[value].v.VoltageMin[i]);
                }
                break;
    };
    // -- Calculate Current and Power average values ----------
    for (i=0;i<12;i++)
    {
       if (channelEnabled[i])
       {
          // -- RMS Current (24-bit) ----------
          measAccumulation[value].v.CurrentAvg[i] /= measAccumulation[value].v.Measurements;
          Add24BitToBuffer(&measBuffer[0], &idx, measAccumulation[value].v.CurrentMax[i],
             measAccumulation[value].v.CurrentAvg[i], measAccumulation[value].v.CurrentMin[i]);
          // -- Power: Watts (16-bit) ----------
          measAccumulation[value].v.WattsAvg[i] /= measAccumulation[value].v.Measurements;
          Add16BitToBuffer(&measBuffer[0], &idx, measAccumulation[value].v.WattsMax[i],
             measAccumulation[value].v.WattsAvg[i], measAccumulation[value].v.WattsMin[i]);
          // -- Power: VAR (16-bit) ----------
          measAccumulation[value].v.VARAvg[i] /= measAccumulation[value].v.Measurements;
          Add16BitToBuffer(&measBuffer[0], &idx, measAccumulation[value].v.VARMax[i],
             measAccumulation[value].v.VARAvg[i], measAccumulation[value].v.VARMin[i]);
          // -- Power: VA (16-bit) ----------
          measAccumulation[value].v.VAAvg[i] /= measAccumulation[value].v.Measurements;
          Add16BitToBuffer(&measBuffer[0], &idx, measAccumulation[value].v.VAMax[i],
             measAccumulation[value].v.VAAvg[i], measAccumulation[value].v.VAMin[i]);
       }
    }
    // -- Frequency ----------
    measAccumulation[value].v.FreqAvg /= measAccumulation[value].v.Measurements;
    Add16BitToBuffer(&measBuffer[0], &idx, measAccumulation[value].v.FreqMax, measAccumulation[value].v.FreqAvg, measAccumulation[value].v.FreqMin);
    // -- Temperature ----------
    measAccumulation[value].v.TempAvg /= measAccumulation[value].v.Measurements;
    measBuffer[idx++] = measAccumulation[value].v.TempMax; measBuffer[idx++] = measAccumulation[value].v.TempAvg;
    measBuffer[idx++] = measAccumulation[value].v.TempMin;
    // -- Industrial Inputs ----------
    for (i=0;i<8;i++)
       if (indIPEnabled[i])
       {
          measAccumulation[value].v.IndustrialIPAvg[i] /= measAccumulation[value].v.Measurements;
          Add16BitToBuffer(&measBuffer[0], &idx, measAccumulation[value].v.IndustrialIPMax[i],
             measAccumulation[value].v.IndustrialIPAvg[i], measAccumulation[value].v.IndustrialIPMin[i]);
       }
    // -- Store the data in FLASH ----------
    measurementSize = idx;
    StorMem_AddProfileValue(value, wrapMemory[value], &measBuffer[0], idx);
    // -- Clear the counters ----------
    measAccumulation[value].v.Measurements = 0;
}
// == ANALOGUE SIGNAL SAMPLING ==========
// -- ADC_Initialize: Initialize the ADC for internal RC clock, auto-conversion and scanning of inputs AN9..15. ----------
void ADC_Initialize(void)
{
    ADPCFG = 0b0000000111111111;
    ADCON1 = 0b0000000011100100;                    // A/D module disabled, integer output, auto-convert
    ADCON2 = 0b0000010000000000;                    // AVdd and AVss references, scan input, 16-word buffer, use Mux A
    ADCON2bits.SMPI = 6;                            // Generate interrupt after 7 samples are completed
    ADCON3 = 0b0000000010000000;                    // Use internal RC clock
    ADCHS  = 0b0000000000000000;
    ADCSSL = 0b1111111000000000;
    IFS0bits.ADIF = false;                          // Clear the ADC interrupt flag
    IEC0bits.ADIE = true;                           // Enable the interrupt
    ADCON1bits.ADON = true; ADCON1bits.SAMP = true; // Enable the ADC
```

```
    IndIP_Initialize();                                                 // Initialize the Industrial Inputs
}
// Interrupt: ADC: ADC interrupt called when seven sample/conversions have been completed. Results are ADCBUF0..7 for AN9..15. ----------
void __attribute__((__interrupt__)) _ADCInterrupt(void)
{
    // -- ADCBUF 0,1,2 are the voltage inputs and are not used presently --
    // -- ADCBUF 3,4,5,6 are the current channel inputs and are not used presently --
    // -- ADCBUF 7 is the multiplexed industrial input which must be stored and the next channel selected --
    currentMeasurements.IndustrialInputs[indipChannel] = ADCBUF7;
    IndIP_SelectNextChannel();
    IFS0bits.ADIF = 0;                                                  // Clear the interrupt flag and return
}
// -- IndIP_SelectInput: Sets the industrial input multiplexer to the channel specified by 'value' using the octal latch. ----------
void IndIP_SelectInput(unsigned char value)
{
    TMyU16 tempPort;
    tempPort.Value = PORTD;
    selRegValue.b = (PORTD & 0xFF);
    selRegValue.v.IndIPSel = value;
    TRISD &= 0xFF00;
    LATD = selRegValue.b;
    SelReg_C = true; SelReg_C = false;
}
// -- IndIP_SelectNextChannel: Increments the industrial input channel pointer and wraps it to zero if necessary. ----------
void IndIP_SelectNextChannel(void)
{
    if (indipChannel++ > 7) indipChannel = 0;
    IndIP_SelectInput(indipChannel);
}
// -- IndIP_Initialize: Set the industrial input multiplexer to the first input channel. ----------
void IndIP_Initialize()
{
    indipChannel = 0;
    IndIP_SelectInput(indipChannel);
}
```

```
/*==  PROCMEMORY.H & PROCMEMORY.C  ==========
Initializes the SRAM and DataFlash memories. Contains the routines for data storage and retrieval.
==========*/
#include "HardwareProfile.h"
// -- Definitions ----------
// -- tMemoryStatus ----------
typedef union {
    struct __attribute__ ((packed)) {
        unsigned HWFault            : 1;        unsigned DataFLASH1Ok       : 1;
        unsigned DataFLASH2Ok       : 1;        unsigned DataFLASH3Ok       : 1;
        unsigned SRAM1Ok            : 1;        unsigned SRAM2Ok            : 1;
        unsigned Unused             : 2;
    };
    unsigned char Value;
} tMemoryStatus;
// -- tDataFLASHMemoryPtr ----------
typedef union {
    struct __attribute__ ((packed)) {
        unsigned short Verifier;
        unsigned long UpdateCount, TOD;
        unsigned long SecsStartAddr, SecsStopAddr, SecsRdPtr, SecsWrPtr, SecsMemFree;
        unsigned long MinsStartAddr, MinsStopAddr, MinsRdPtr, MinsWrPtr, MinsMemFree;
        unsigned long HrsStartAddr, HrsStopAddr, HrsRdPtr, HrsWrPtr, HrsMemFree;
    } v;
    unsigned char b[70] __attribute__ ((packed));
} tDataFLASHMemoryPtr;

// ==  PROCMEMORY.C  ==========
#include "procMemory.h"  #include "DataFLASH.h"  #include "procControl.h"  #include "procMeasurement.h"
// -- Global Variables ----------
tMemoryStatus memoryStatus;
tDataFLASHMemoryPtr dataFLASHPtr;
extern unsigned short measurementSize;
U32 currRdPtr_Secs, currRdPtr_Mins, currRdPtr_Hrs;
// -- procMemory_Initialize: Initalizes the SRAM and DataFlash memories, verifies their operation and retrieves the current memory pointers. ----------
unsigned char procMemory_Initialize(void)
{
    unsigned char device; unsigned short pg, pgidx;
    memoryStatus.Value = 0;                                             // Initialize the memory status flags
```

```
  SPI2_DeselectAll();                                   // Ensure all SPI 2 devices are de-selected
  memoryStatus.DataFLASH1Ok = DataFLASH_DeviceOK(1);    // Verify operation of DataFlash 1
  memoryStatus.DataFLASH2Ok = DataFLASH_DeviceOK(2);    // Verify operation of DataFlash 2
  memoryStatus.DataFLASH3Ok = DataFLASH_DeviceOK(3);    // Verify operation of DataFlash 3
  SRAM_Initialize();                                    // Initialize the SRAM interface
  memoryStatus.SRAM1Ok = SRAM_CheckDevice(1);           // Verify operation of SRAM 1
  memoryStatus.SRAM2Ok = SRAM_CheckDevice(2);           // Verify operation of SRAM 2
  if (memoryStatus.DataFLASH1Ok) StorMem_GetMemPointers();  // Retrieve the current memory pointers
  memoryStatus.HWFault = (!memoryStatus.DataFLASH1Ok || !memoryStatus.DataFLASH2Ok || !memoryStatus.DataFLASH3Ok ||
    !memoryStatus.SRAM1Ok || !memoryStatus.SRAM2Ok);    // Mark a fault condition if there is a memory problem
}
// -- procMemory_Processor: Not current used so immediately returns to calling method. ----------
void procMemory_Processor(void) {  // -- No tasks  }
// -- StorMem_GetMemPointers: Retrieves the current memory pointers for FLASH access from DataFlash 1 address 0. ----------
void StorMem_GetMemPointers(void) {  DataFLASH_Read(1, 0, 0, 70, &dataFLASHPtr.b[0]);  }
// -- StorMem_SetMemPointers: Stores the current memory pointers in DataFlash 1 at address 0. ----------
void StorMem_SetMemPointers(void)
{
  dataFLASHPtr.v.Verifier = 0xBBAA;                     // Memory verifier to confirm that data is correct when read
  dataFLASHPtr.v.UpdateCount++;                         // Increment the pointer update counter (to ensure we do not 'kill' the cell)
  while (!DataFLASH_Idle(1));                           // Wait until the DataFlash is idle
  DataFLASH_PageToSRAM(1, 0, 1);                        // Copy page to SRAM for modifying
  while (!DataFLASH_Idle(1));                           // Wait until the DataFlash is idle
  DataFLASH_SRAM_Write(1, 1, 0, &dataFLASHPtr.b[0], 70);  // Write the pointers to DataFlash
  while (!DataFLASH_Idle(1));                           // Wait until the DataFlash is idle
  DataFLASH_SRAMToPage(1, 1, 0);                        // Write the changed SRAM page to FLASH
}
// -- StorMem_DecodeAddr: Determines which device, page and index a 32-bit address falls on. ----------
void StorMem_DecodeAddr(U32 value, U8 *device, U16 *page, U16 *pgIdx)
{
  *page = value / 1056;                                 // Calculate the page from 'value'
  *pgIdx = value - (*page * 1056);                      // Calculate the position within the page
  *device = *page / 8192;                               // Which device does it fall on?
  *page -= (*device * 8192);                            // If it's the second device we need to adjust the page index
  *device += 1;                                         // Devices are indexed from 1, not 0
}
// -- StorMem_AddProfileValue ----------
// Stores a secs, mins or hours measurement value in DataFlash and updates the memory write pointers. To speed up memory access data is written as two
SRAM pages (if necessary) in sequence. ie. while one is being copied to FLASH the processor is updating the second one.
void StorMem_AddProfileValue(U8 profile, bool wrapMemory, U8 *buffer, U16 length)
{
  unsigned char device, flashSRAM = 1; unsigned short bytesThisPass, page, pgIdx;
  U32 *memStart, *memStop, *memWrPtr, *memFree;
  switch (profile)
  {
    case 0 :                                            // SECONDS measurement profile
      memStart = &dataFLASHPtr.v.SecsStartAddr;         // Start address of memory buffer in Flash
      memStop = &dataFLASHPtr.v.SecsStopAddr;           // Stop address of memory buffer in Flash
      memWrPtr = &dataFLASHPtr.v.SecsWrPtr;             // The current Write pointer to support circular buffering
      memFree = &dataFLASHPtr.v.SecsMemFree;            // The amount of data free in the Seconds buffer
    break;
    case 1 :                                            // MINUTES measurement profile
      memStart = &dataFLASHPtr.v.MinsStartAddr;         // Start address of memory buffer in Flash
      memStop = &dataFLASHPtr.v.MinsStopAddr;           // Stop address of memory buffer in Flash
      memWrPtr = &dataFLASHPtr.v.MinsWrPtr;             // The current Write pointer to support circular buffering
      memFree = &dataFLASHPtr.v.MinsMemFree;            // The amount of data free in the Minutes buffer
    break;
    case 2 :                                            // HOURS measurement profile
      memStart = &dataFLASHPtr.v.HrsStartAddr;          // Start address of memory buffer in Flash
      memStop = &dataFLASHPtr.v.HrsStopAddr;            // Stop address of memory buffer in Flash
      memWrPtr = &dataFLASHPtr.v.HrsWrPtr;              // The current Write pointer to support circular buffering
      memFree = &dataFLASHPtr.v.HrsMemFree;             // The amount of data free in the Minutes buffer
    break;
  };
  if ((*memWrPtr + length) > *memStop)                  // Do we have space to write without wrapping?
  {
    *memFree = 0;
    if (wrapMemory)                                     // If wrapping is enabled then wrap to the beginning of the buffer
      *memWrPtr = *memStart;
    else
      return;
  }
  while (length > 0)                                    // Write the data to DataFlash and update the memory write pointers
  {
    StorMem_DecodeAddr(*memWrPtr, &device, &page, &pgIdx);  // Decode address for device access
    bytesThisPass = 1056 - pgIdx;                       // A page is 1056 bytes long
```

```
    if (bytesThisPass > length) bytesThisPass = length;
    DataFLASH_PageToSRAM(device, page, flashSRAM);                  // Move the page to be accessed to SRAM for changing
    while (!DataFLASH_Idle(device));                                // Wait until command has been processed
    DataFLASH_SRAM_Write(device, flashSRAM, pgIdx, buffer, bytesThisPass);  // Modify the SRAM contents as required
    while (!DataFLASH_Idle(device));                                // Wait until command has been processed
    DataFLASH_SRAMToPage(device, flashSRAM, page);                  // Write the SRAM page back to Flash
    while (!DataFLASH_Idle(device));                                // Wait until command has been processed
    buffer += bytesThisPass; length -= bytesThisPass; *memWrPtr += bytesThisPass;  // Update memory pointers
    if (*memFree != 0) *memFree -= bytesThisPass;
    if (flashSRAM == 1) flashSRAM = 2; else flashSRAM = 1;          // Select next SRAM page
  }
}
// -- StorMem_ReadNextMeasurement ---------------------------------------------------------------------------------------------------------------------------------------
// Reads the next measurement of 'profile' from Flash memory and places it in 'result'. The read pointers are updated.
U16 StorMem_ReadNextMeasurement(U8 profile, U8 *result)
{
  unsigned char device;  unsigned short bytesThisPass, measLength, page, pgIdx;  U32 *memStart, *memStop, *memRdPtr, *memFree;
  measLength = measurementSize;
  switch (profile)
  {
    case 0 :                                                        // Seconds profile
      memStart = &dataFLASHPtr.v.SecsStartAddr; memStop = &dataFLASHPtr.v.SecsStopAddr; memRdPtr = &dataFLASHPtr.v.SecsRdPtr;
      memFree = &dataFLASHPtr.v.SecsMemFree;
    break;
    case 1 :                                                        // Minutes profile
      memStart = &dataFLASHPtr.v.MinsStartAddr; memStop = &dataFLASHPtr.v.MinsStopAddr; memRdPtr = &dataFLASHPtr.v.MinsRdPtr;
      memFree = &dataFLASHPtr.v.MinsMemFree;
    break;
    case 2 :                                                        // Hours profile
      memStart = &dataFLASHPtr.v.HrsStartAddr; memStop = &dataFLASHPtr.v.HrsStopAddr; memRdPtr = &dataFLASHPtr.v.HrsRdPtr;
      memFree = &dataFLASHPtr.v.HrsMemFree;
    break;
  };
  if ((*memRdPtr + measLength) > *memStop) *memRdPtr = *memStart;   // Wrap round to the beginning of the storage buffer
  while (measLength > 0)
  {
    StorMem_DecodeAddr(*memRdPtr, &device, &page, &pgIdx);          // Decode address for device access
    bytesThisPass = 1056 - pgIdx;
    if (bytesThisPass > measLength) bytesThisPass = measLength;
    DataFLASH_Read(device, page, pgIdx, bytesThisPass, result);     // Read from the Flash into result
    // -- Update the profile read pointers --
    result += bytesThisPass; measLength -= bytesThisPass; *memRdPtr += bytesThisPass; *memFree += bytesThisPass;
  }
  return measurementSize;
}
// -- StorMem_UpdateReadPointers: Commits the profile read pointers to Flash memory so if power is removed the Profiler can continue from where it left off. ----
void StorMem_UpdateReadPointers(void)
{
  dataFLASHPtr.v.SecsRdPtr = currRdPtr_Secs; dataFLASHPtr.v.MinsRdPtr = currRdPtr_Mins; dataFLASHPtr.v.HrsRdPtr = currRdPtr_Hrs;
  StorMem_SetMemPointers();                                         // Write the values to DataFlash
}
// -- StorMem_ReadMemory: Reads 'length' bytes from Flash memory at 'address' into 'result'. ----------------------------------------------------------------------
void StorMem_ReadMemory(U32 address, U16 length, U8 *result)
{
  unsigned char device; unsigned short bytesThisPass, page, pgIdx;
  while (length > 0)
  {
    StorMem_DecodeAddr(address, &device, &page, &pgIdx);            // Decode address for device access
    bytesThisPass = 1056 - pgIdx;
    if (bytesThisPass > length) bytesThisPass = length;
    DataFLASH_Read(device, page, pgIdx, bytesThisPass, result);     // Read from the Flash and place in 'result'
    result += bytesThisPass; length -= bytesThisPass; address += bytesThisPass;
  }
}
// ==  CONFIGURATION MEMORY ACCESS  ============================================================================================================================
// -- Configuration Memory: Read: Reads multiple blocks of memory from the DSC internal EEPROM to 'resp'. ---------------------------------------------------------
void CfgMem_Read(int eeAddr, unsigned char blocks, unsigned char *resp)
{
  int i;
  for (i=0; i<blocks; i++)
  {
    intEE_ReadRow(eeAddr, resp);                                    // Read each row placing the result at pointer 'resp'
    eeAddr += 32; resp += 32;                                       // Increment the read address and buffer pointer to the next row
  }
}
// -- Configuration Memory Write: Writes multiple blocks from pointer 'values' to the DSC internal EEPROM. ----------------------------------------------------------
```

```
void CfgMem_Write(int eeAddr, unsigned char blocks, unsigned char *values)
{
   int i;
   for (i=0;i<blocks;i++)
   {
      intEE_WriteRow(eeAddr, values);                    // Write the row starting at pointer 'values'
      eeAddr += 32; values += 32;                        // Increment the write address and buffer pointer to the next row
   }
}
```

```
/*== SRAM.H & SRAM.C ==========================================================================================
SRAM memory control and interface routines.
==========================================================================================*/

//== SRAM.C ====================================================================================================
 #include "SRAM.h"
// -- SRAM_Initialize: Initializes the port control pins for the external SRAM and address selection latches. ----------
void SRAM_Initialize(void)
{
   SRAM_OE = 1;                                       // Disable the SRAM
   SRAM1_CE = 1; SRAM2_CE = 1;                        // Clear the output enables for the SRAMs
   SRAM_AR1 = 0; SRAM_AR2 = 0; SRAM_AR3 = 0;          // Disable the Address Register Latches
}
// -- SRAM_CheckDevice: Performs a write/read verification test of SRAM operation. ----------
unsigned char SRAM_CheckDevice(U8 device)
{
   unsigned char i, tempBuffer[10];
   for (i=0;i<10;i++) tempBuffer[i] = i;              // Assign a 0..9 value to the 10 byte test variable
   SRAM_Write_Buffer(device, (U32)0, &tempBuffer[0], (U16)10);  // Write the value to the SRAM
   SRAM_Read_Buffer(device, (U32)0, &tempBuffer[0], (U16)10);   // Read the value back from the SRAM to tempBuffer
   for (i=0;i<10;i++) if (tempBuffer[i] != i) return false;     // Check each byte and if one does not match return a failure
   return true;                                       // The test was successful
}
// -- SRAM_SelectAddress: Latches the 'Address' value on the three registers used for addressing the external SRAM. ----------
void SRAM_SelectAddress(U32 address)
{
   TMyU32 temp;
   temp.Value = address;                              // Convert the 32-bit address value to a byte-accessible strucure.
   SRAM_PortDir &= 0xFF00;                            // Specify the port as an output
   SRAM_AR1 = 0; SRAM_AR2 = 0; SRAM_AR3 = 0;          // Clear the address register latches
   SRAM_Port &= 0xFF00;                               // Clear the port value
   SRAM_Port |= temp.Bytes[2];                        // Write the address value to the port
   SRAM_AR3 = 1; SRAM_AR3 = 0;                        // Toggle AR3 to latch address value
   SRAM_Port &= 0xFF00;                               // Clear the port value
   SRAM_Port |= temp.Bytes[1];                        // Write the address value to the port
   SRAM_AR2 = 1; SRAM_AR2 = 0;                        // Toggle AR2 to latch address value
   SRAM_Port &= 0xFF00;                               // Clear the port value
   SRAM_Port |= temp.Bytes[0];                        // Write the address value to the port
   SRAM_AR1 = 1; SRAM_AR1 = 0;                        // Toggle AR1 to latch address value
}
// -- SRAM_Read: Selects the external SRAM ('device') and reads a byte from 'address'. ----------
U8 SRAM_Read_U8(U8 device, U32 address)
{
   TMyU16 result;
   SRAM_WR = 1;                                       // Set the SRAM to a read state
   SRAM_OE = 1;                                       // Place the SRAM into a tri-state output mode
   SRAM_SelectAddress(address);                       // Select the address to read from
   if (device == 1) SRAM1_CE = 0; else SRAM2_CE = 0;  // Select the appropriate device to read from
   SRAM_PortDir |= 0x00FF;                            // Set the port as an input
   SRAM_OE = 0;                                       // Enable the SRAM outputs
   Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop();  // Wait for the data to latch onto the port
   result.Value = SRAM_Port;                          // Read the SRAM value from the port
   if (device == 1) SRAM1_CE = 1; else SRAM2_CE = 1;  // Disable the SRAM device
   return result.Bytes[0];                            // Return the read value
}
// -- SRAM_Read_U16: Selects the external SRAM ('device') and reads a 16-bit value from 'address'. ----------
void SRAM_Read_U16(U8 device, U32 address, U16 *result)
{
   TMyU16 temp;
   temp.Bytes[0] = SRAM_Read_U8(device, address++); temp.Bytes[1] = SRAM_Read_U8(device, address);
   *result = temp.Value;
}
// -- SRAM_Read_U32:  Selects the external SRAM ('device') and reads a 32-bit value from 'address'. ----------
void SRAM_Read_U32(U8 device, U32 address, U32 *result)
{
```

```
    U8 i; TMyU32 temp;
    for (i=0;i<4;i++) temp.Bytes[i] = SRAM_Read_U8(device, address++);
    *result = temp.Value;
}
// -- SRAM_Read_Buffer: Selects the external SRAM ('device') and reads 'length' bytes from 'address' and placing the result in memory position pointer 'buffer'. --
void SRAM_Read_Buffer(U8 device, U32 address, U8 *buffer, U16 length)
{  while (length-- > 0) *buffer++ = SRAM_Read_U8(device, address++);  }
// -- SRAM: Write_U8: Selects the external SRAM ('device') and writes a byte value to 'address'. ----------------------------------------------------------------------------------------
void SRAM_Write_U8(U8 device, U32 address, U8 value)
{
    SRAM_OE = 1;                                                    // Disable the SRAM outputs
    SRAM_PortDir &= 0xFF00;                                         // Specify the port as an output
    if (device == 1) SRAM1_CE = 1; else SRAM2_CE = 1;               // Select the appropriate SRAM device
    SRAM_SelectAddress(address);                                    // Set the address to write to
    SRAM_Port &= 0xFF00;                                            // Clear the port value
    SRAM_Port |= value;                                             // Set the value to write
    if (device == 1) SRAM1_CE = 0; else SRAM2_CE = 0;               // Enable the SRAM device
    SRAM_WR = 0;                                                    // Specify a write
    Nop(); Nop(); Nop(); Nop(); Nop();Nop(); Nop(); Nop(); Nop(); Nop();   // Necessary delay for the data to be written in SRAM
    SRAM_WR = 1;                                                    // Clear the write flag
    if (device == 1) SRAM1_CE = 1; else SRAM2_CE = 1;               // De-select the device
}
// -- SRAM_Write_U32: Selects the external SRAM ('device') and writes a 32-bit value to 'address'. ----------------------------------------------------------------------------------
void SRAM_Write_U32(U8 device, U32 address, U32 value)
{
    U8 i; TMyU32 temp;
    temp.Value = value;
    for (i=0;i<4;i++) SRAM_Write_U8(device, address++, temp.Bytes[i]);
}
// -- SRAM_Write_Buffer: Selects the external SRAM ('device') and writes 'length' bytes from memory pointer 'buffer' to SRAM, starting at 'address'. ------------------
void SRAM_Write_Buffer(U8 device, U32 address, U8 *buffer, U16 length) { while (length-- > 0) SRAM_Write_U8(device, address++, *buffer++);  }
```

```
/*==  TIMEKEEPING.H & TIMEKEEPING.C  ================================================================================
Implements the internal real-time-clock and synchronises with the external RTC.
================================================================================================================================*/
#include "HardwareProfile.h"  #include "I2CDevices.h"
// -- Definitions ---------------------------------------------------------------------------------------------------------------------------------------------
// -- tClockStatus --------------------------------------------------------------------------------------------------------------------------------------------
typedef union {
    struct __attribute__ ((packed)) {
        unsigned RTCOk          : 1;        unsigned TODValid       : 1;
        unsigned SyncClock      : 1;        unsigned SecPassed      : 1;
        unsigned HourPassed     : 1;        unsigned Unused         : 3;
    };
    unsigned char Value;
} tClockStatus;
// -- tPacketTOD ----------------------------------------------------------------------------------------------------------------------------------------------
typedef union {
    struct __attribute__ ((packed)) {
        unsigned Sec            : 6 __attribute__ ((packed));   unsigned Min    : 6 __attribute__ ((packed));
        unsigned Hr             : 5 __attribute__ ((packed));   unsigned Day    : 5 __attribute__ ((packed));
        unsigned Month          : 4 __attribute__ ((packed));   unsigned Year   : 6 __attribute__ ((packed));
    };
    unsigned char b[4] __attribute__ ((packed));
    unsigned long l __attribute__ ((packed));
} tPackedTOD;

// ==  TIMEKEEPING.C  ============================================================================================================
#include "Timekeeping.h"  #include "procMeasurement.h"
// -- Global Variables ----------------------------------------------------------------------------------------------------------------------------------------
tClockStatus clockStatus;  tDateTime currentTOD;
// -- Clock_Initialize: Initializes the external DS3231S RTC and if the clock is valid the internal currentTOD will be updated. ----------------------------------------
unsigned char Clock_Initialize(void)
{
    clockStatus.Value = 0;                                          // Initialize the clock status flags
    clockStatus.RTCOk = DS3231_Initialize();                        // Initialize the external DS3231 RTC
  if (clockStatus.RTCOk)
  {
      clockStatus.TODValid = DS3231_GetClock(&currentTOD);          // Update the internal clock with the RTC value
      CNEN2bits.CN23IE = true;                                      // Enabled the interrupt used for 1-sec duty cycle output of RTC
    }
    return clockStatus.RTCOk;
}
// -- Clock_Increment -----------------------------------------------------------------------------------------------------------------------------------------
// Called every second to update the internal clock. If a minute rollover is going to occur the internal clock will be re-synchronised with the RTC.
```

```
void Clock_Increment(void)
{
   if (clockStatus.TODValid)
   {
      if (currentTOD.Sec < 59)                          // Increment the second counter of the internal RTC
         currentTOD.Sec++;                              // if the value is less than a minute rollover.
      else
      {
         clockStatus.TODValid = DS3231_GetClock(&currentTOD);   // Synchronise with the external clock every minute
         if ((currentTOD.Min == 0) && (currentTOD.Sec == 0))    // Set the HourPassed bit every hour
            clockStatus.HourPassed = true;
      }
   }
   else
      clockStatus.TODValid = DS3231_GetClock(&currentTOD);    // Update the clock if the internal RTC is invalid
}
// -- Clock_PackDateTime ---------------------------------------------------------------------------------------------------------------------------------------------
// Packs a tDateTime value into four bytes which are returned from the routine. Used for time-stamping measurement values.
U32 Clock_PackDateTime(tDateTime value)
{
   tPackedTOD tod;                                   // Packed data structure to convert from a tDateTime to a 32-bit
   tod.Year = value.Year; tod.Month = value.Month; tod.Day = value.Day;   // Copy in each value to the structure
   tod.Hr = value.Hrs; tod.Min = value.Min; tod.Sec = value.Sec;
   return tod.l;                                     // Return the 32-bit value for the packed clock
}
// -- Clock_SetTOD: Updates the external RTC to the specified tDateTime value and re-synchronizes the internal clock if successful. ----------------------------
bool Clock_SetTOD(tDateTime value)
{
   bool resp;
   if (DS3231_SetClock(value))                       // Set the external RTC and synchronise with the internal clock
   {
      currentTOD = value;                            // Update the internal software clock to 'value'
      clockStatus.TODValid = true;                   // Clock is valid so can be used in measurements
   }
}
```

```
/*==  TYPES.H  ==================================================================================================================================================
Defines data types used in the Power Profiler code.
==========================================================================================================================================================*/
// -- Data types -----------------------------------------------------------------------------------------------------------------------------------------------
#define bool          unsigned char          #define S8          signed char
#define S16           signed short           #define S32         signed long
#define U8            unsigned char          #define U16         unsigned short
#define U32           unsigned long
// -- Logical Conditions ---------------------------------------------------------------------------------------------------------------------------------------
#define False         0                      #define false       0
#define True          1                      #define true        1
// -- Composite Data Types: simplify access to byte values -----------------------------------------------------------------------------------------------------
typedef union TMyU16 {  unsigned short Value;  unsigned char Bytes[2];  } TMyU16;
typedef union TMyU32 {  unsigned long Value;  unsigned char Bytes[4];  } TMyU32;
```

PIC18F Embedded Software

Device Initialization and Control

Communications

```
/*== HARDWAREPROFILE.H ========================================================================================
  Hardware-specific definitions for the Power Profiler board.
============================================================================================================*/
#include <P18F4550.h>  #include "Types.h"

// -- Definitions ----------------------------------------------------------------------------------------------
#define intLowPriority                  0                       #define intHighPriority             1
// -- Oscillator Configuration ---------------------------------------------------------------------------------
#define FOSC                            48000000                #define FCY                          FOSC/4
#define CLOCK_FREQ                      48000000                #define GetSystemClock()             CLOCK_FREQ
// - USB Parameters -----------------------------------------------------------------------------------------------
#define USE_USB_BUS_SENSE_IO            1                       #define USB_BUS_SENSE                PORTBbits.RB5
#define USB_Connected                   PORTBbits.RB5           #define self_power                   1
// - Busses -------------------------------------------------------------------------------------------------------
#define DataBus_Outputs                 TRISD = 0x00            #define DataBus_Inputs               TRISD = 0xFF
#define ddOutput                        0                       #define ddInput                      1
// - Communications -----------------------------------------------------------------------------------------------
#define SPIComms_LED                    PORTBbits.RB3           #define IPC_DOUT_Dir                 TRISCbits.TRISC7
#define IPC_SS_Dir                      TRISBbits.TRISB2        #define IPC_SS_Pin                   PORTBbits.RB2
#define IPC_HS_Dir                      TRISAbits.TRISA5        #define IPC_HS_Pin                   LATAbits.LATA5
// - Control ------------------------------------------------------------------------------------------------------
#define SelReg_RCLK                     PORTCbits.RC6           #define SelReg_Data                  PORTDbits.RD7
#define SelReg_DataDir                  TRISDbits.TRISD7        #define SelReg_Clk                   PORTDbits.RD6
#define SelReg_ClkDir                   TRISDbits.TRISD6        #define BATTCHG_EN_TRIS              TRISEbits.TRISE2
#define BATTCHG_EN_Pin                  LATEbits.LATE2          #define BATTCHG_Status_TRIS          TRISBbits.TRISB3
#define BATTCHG_Status_Pin              PORTBbits.RB3
// - Optoelectronics ----------------------------------------------------------------------------------------------
#define LCD_RS                          PORTCbits.RC0           #define LCD_RW                       PORTCbits.RC1
#define LCD_E                           PORTCbits.RC2           #define LCD_DataDir                  TRISD
#define LCD_Data                        PORTD
```

```
/*== MAIN.C ===================================================================================================
  Main file for program entry for PIC18F4550 processor code.
============================================================================================================*/
// == CONFIGURATION BITS ======================================================================================
#pragma config PLLDIV = 5           // 20 MHz crystal           #pragma config CPUDIV = OSC1_PLL2          // PLL divided by 2
#pragma config USBDIV = 2           // Clock source from 96MHz PLL/2  #pragma config IESO = OFF
#pragma config FOSC = HSPLL_HS      // High-speed oscillator with PLL  #pragma config FCMEN = OFF
#pragma config PWRT = OFF           // Power-up timer disabled
#pragma config BOR = ON             // Brown-out reset on       #pragma config BORV = 3
#pragma config VREGEN = ON          // USB Voltage reg on
#pragma config WDT = OFF            // Watchdog timer disabled  #pragma config WDTPS = 32768
#pragma config MCLRE = ON                                    #pragma config LPT1OSC = OFF
#pragma config PBADEN = OFF                                  #pragma config STVREN = ON
#pragma config XINST = OFF          // Extended Instruction Set disabled
#pragma config LVP = OFF
#pragma config CP0 = OFF            #pragma config CP1 = OFF     #pragma config CPB = OFF
#pragma config WRT0 = OFF           #pragma config WRT1 = OFF    #pragma config WRTB = OFF      #pragma config WRTC = OFF
#pragma config EBTR0 = OFF          #pragma config EBTR1 = OFF   #pragma config EBTRB = OFF
// -- File Includes -------------------------------------------------------------------------------------------
#include "HardwareProfile.h"  #include "GenericTypeDefs.h"  #include "Compiler.h"  #include "Timers.h"  #include "procComms.h"  #include "Comms_IPC.h"
#include "procControl.h"  #include "procDisp.h"
// -- Global Variables ----------------------------------------------------------------------------------------
unsigned char t3Counts = 0, timerTick100ms = 0;
// -- Processor Initialization: Initailizes the internal registers of the PIC18F processor. -------------------
void PIC18F_Initialize(void)
{
  // -- IO Port direction and default levels --
  TRISA = 0b1011111; PORTA = 0xFF; TRISB = 0b11101111; PORTB = 0x10; TRISC = 0xB8; PORTC = 0x00; TRISD = 0xFF; TRISE = 0xFF;
  // -- ADC Configuration --
  ADCON0 = 0;                                   // Disable the ADC until settings are done
  ADCON1 = 0b00001000;                          // AN0 to AN6 enabled. VREF- = VSS VREF+ = VDD
  ADCON2 = 0b10111110;                          // Result is right-justified, TAD = 20, FOSC/64
  ADCON0bits.ADON = true;
  T3CON = 0b10011000;                           // Configure Timer 3 for the 100ms tick timer
  TMR3H = 0; TMR3L = 0;
  PIE2bits.TMR3IE = true;                       // Enable Timer 3's interrupt
  IPR2bits.TMR3IP = intLowPriority;
  PIR2bits.TMR3IF = false;
  // -- Initialize SPI port pins --
  TRISCbits.TRISC7 = 1;                         // SPI SDO is set as input otherwise bus contention would occur
  TRISBbits.TRISB3 = 0;                         // SPI SDI
  IPC_HS_Dir = 0;                               // Handshaking signal is an output
  IPC_HS_Pin = 0;                               // Default signal state is low
  IPC_SS_Dir = 1;                               // SPI Slave select signal is an input
```

```
    // -- Initialize SPI interface --
    SSPCON1 = 0b00000101;
    SSPSTAT = 0b00000000;
    SSPCON1bits.SSPEN = true;
    // - Set up the SPI SS pin since it is using normal IO -
    INTCON2bits.INTEDG2 = 0;                                    // Trigger on falling edge of SPI Slave Select
    INTCON3bits.INT2IP = 1;
    INTCON3bits.INT2IE = 1;
    INTCON3bits.INT2IF = 0;
}
// -- ADC: MeasureChannel: Selects and samples the specified ADC channel and returns the result. ----------
unsigned short ADC_MeasureChannel(unsigned char channel)
{
    unsigned char i;
    channel <<= 2;
    ADCON0 = channel + 1;                                       // Select the channel to measure
    for (i=0;i<24;i++) Nop();                                   // Delay for ADC sampling to settle
    ADCON0bits.GO_DONE = true;                                  // Start the conversion
    while (ADCON0bits.GO_DONE == true);                         // Wait till conversion is complete
    return ADRES;                                               // Return the ADC conversion result
}
// == MAIN PROGRAM ROUTINE ====================================================================================
void main(void)
{
    PIC18F_Initialize();                                        // Initialize the PIC18F processor, IO interfaces and peripherals
    ProcControl_Initialize();                                   // Initialize the Control Task
    ProcComms_Initialize();                                     // Initialize the Communications Task
    ProcDisp_Initialize();                                      // Initialize the LCD display updating Task
    T3CONbits.TMR3ON = True;                                    // Enable Timer 3 to generate the 100ms tick
    INTCONbits.GIEH = 1; INTCONbits.GIEL = 1;                   // Enable the interrupts
    while (1)                                                   // Continuously process the Tasks
    {
        ProcControl_Processor();
        ProcComms_Processor();
        ProcDisp_Processor();
    }
}
// == INTERRUPT VECTORS =======================================================================================
#pragma code InterruptVectorHigh = 0x08
void InterruptVectorHigh(void) { _asm GOTO InterruptHandlerLow _endasm }
#pragma code
#pragma code InterruptVectorLow = 0x18
void InterruptVectorLow(void) { _asm GOTO InterruptHandlerLow _endasm }
#pragma code
#pragma interruptlow InterruptHandlerLow
void InterruptHandlerLow(void)
{
    U8 temp;
    // -- Timer 3 : Used as an increment for the 100ms Timer Tick --
    if (PIR2bits.TMR3IF)
    {
        if (t3Counts++ == 4)
        {
            timerTick100ms++;
            procDisp.Tick100ms = true;
            procControl.Tick100ms = true;
            t3Counts = 0;
        }
        TMR3H = 0x3C; TMR3L = 0xC5;
        PIR2bits.TMR3IF = False;
    }
    // -- Interrupt pin 2 is used for the SPI slave select --
    if (INTCON3bits.INT2IF == True)
    {
        if (INTCON2bits.INTEDG2 == 0)                           // Triggered on falling edge - start of transmission
        {
            IPC_MessageBegin ();                                // Start of SPI packet
            INTCON2bits.INTEDG2 = 1;                            // Configure for rising-edge detection
        }
        else
        {
            IPC_MessageEnd();                                   // End of SPI packet
            INTCON2bits.INTEDG2 = 0;                            // Trigger on falling edge
        }
        INTCON3bits.INT2IF = False;
    }
```

```
    // -- SPI Port receive interrupt --
    if (PIR1bits.SSPIF == True)
    {
        temp = SSPBUF;                          // Always read the SSPBUF register to prevent an overwrite fault flag
        if (IPC_SS_Pin == 0) IPC_RxByte(temp);  // If this device is selected then pass the received byte on to IPC_RxByte
        PIR1bits.SSPIF = False;                 // Clear the SSP receive interrupt flag
    }
    INTCONbits.GIEL = 1;  INTCONbits.GIEH = 1;  // Clear the interrupt flag bits and return
}
#pragma interrupt InterruptHandlerHigh
void InterruptHandlerHigh(void)
{
    // -- Not used, all interrupts default to using the low-priority interrupt handler --
    INTCONbits.GIEH = 1;
}
#pragma code
```

```
/*==  COMMS_IPC.H & COMMS_IPC.C  ============================================================================
  Inter-processor communications implementation between the dsPIC30F6013 and PIC18F4550 processors.
==========================================================================================================*/
#include "HardwareProfile.h"
// -- Constants and Type Definitions ---------------------------------------------------------------------------
#define IPC_BUFF_SIZE                   200
// -- IPC Interface Commands -----------------------------------------------------------------------------------
#define IPCCMD_PING                     0x01    #define IPCCMD_MEASUREMENTS     0x10
#define IPCCMD_GETPIC18FSTATUS          0x11    #define IPCCMD_GETUSBPACKET     0x30
#define IPCCMD_SENDUSBPACKET            0x31
// -- tipcStatus: Manages the IPC communication functionality. -------------------------------------------------
typedef union {
    struct {
        unsigned Active                 : 1;    unsigned MsgResponse        : 1;
        unsigned DSCSignalled           : 1;    unsigned USBMsgWaiting      : 1;
        unsigned MsgRx                  : 1;    unsigned MsgSendDone        : 1;
        unsigned DSCUpdate              : 1;
    };
    unsigned char value;
} tipcStatus;
// -- tipcStateReg: Status information reported back to the dsPIC processor during a data transfer. ------------
typedef union {
    struct {
        unsigned ACK                    : 1;    unsigned Fault              : 1;
        unsigned MsgWaiting             : 1;    unsigned USBMsgWaiting      : 1;
        unsigned Response               : 1;    unsigned Verifier           : 3;
    };
    unsigned char value;
} tipcStateReg;

// ==  COMMS_IPC.C  ========================================================================================
#include "Comms_IPC.h"  #include "procControl.h"  #include "Comms_USB.h"
// -- Global Variables -----------------------------------------------------------------------------------------
unsigned char ipcBuffRxIdx = 0, ipcBuffTxIdx = 0, ipcBuffTxLength = 0, ipcChkSumA, ipcChkSumB;
unsigned char ipcBuffer[IPC_BUFF_SIZE];
tipcStatus ipcStatus;  tipcStateReg ipcStatusReg;
// -- IPC_Initialize: Initializes the inter-processor communications. ------------------------------------------
void IPC_Initialize(void)
{
    ipcStatusReg.value = 0;                 // Clears the status register flags
    ipcStatusReg.Verifier = 5;              // Indicates to the DSC that the register is valid
    ipcStatus.value = 0;                    // Initialize the IPC task status register
    IPC_DOUT_Dir = 1;                       // Specify the SPI Data Out signal as an input
    IPC_HS_Pin = 0;                         // SPI hand-shaking default level is low
    SSPBUF = ipcStatusReg.value;            // Initialize the SPI buffer to contain the status register
}
// -- IPC_MessageBegin: Called by the interrupt routine when a falling edge on SPI Slave Select is detected. ---
void IPC_MessageBegin(void)
{
    ipcStatus.Active = true;                // Set flag to show a data transfer is in progress
    IPC_DOUT_Dir = 0;                       // Set SPI DOUT pin for output
    ipcBuffRxIdx = 0;                       // Initialize the data receive buffer
    ipcStatus.MsgResponse = false;          // Clear the message response flag so data is received into ipcBuffer
    IPC_HS_Pin = !IPC_HS_Pin;               // Toggle the handshake to show ready for data transfer
}
// -- IPC_MessageEnd: Called by the interrupt routine when a falling edge on SPI Slave Select is detected. -----
void IPC_MessageEnd(void)
{
```

```
    IPC_DOUT_Dir = 1;                                   // Set SPI DOUT pin as an input to prevent clashes with other SPI devices
    IPC_HS_Pin = 0;                                     // Set the hand-shake signal low
    ipcStatus.Active = false;                           // Set flag to show a data transfer is not in progress
}
// -- IPC_RxByte: Called by the interrupt routine when a byte is received on the SPI port. ----------
void IPC_RxByte(unsigned char value)
{
    // -- If we are currently sending data then load the next value into SSPBUF from ipcBuffer --
    if (ipcStatus.MsgResponse)
    {
        SSPBUF = ipcBuffer[ipcBuffTxIdx++];             // Send the data sitting in ipcBuffer and increment the pointer
        if (ipcBuffTxIdx == ipcBuffTxLength)
            ipcStatus.MsgSendDone = true;               // If the message is fully sent then indicate to the IPC Task Processor
    }
    else
    {
        // -- If the first byte received is a 0x00 return the status register into SSPBUF --
        if ((ipcBuffRxIdx == 0) && (value == 0x00))
            SSPBUF = ipcStatusReg.value;
        else
        {
            if (ipcBuffRxIdx < IPC_BUFF_SIZE)           // Check for overflow condition
                ipcBuffer[ipcBuffRxIdx++] = value;      // Add received data to ipcBuffer
            else
                ipcBuffRxIdx = 0;
            // -- Check if a full command packet has been received --
            if ((ipcBuffRxIdx > 2) && (ipcBuffRxIdx == ipcBuffer[1]))
                ipcStatus.MsgRx = true;
        }
    }
    IPC_HS_Pin = !IPC_HS_Pin;                           // Toggle the handshake to show that the data has been received
}
// -- IPC_CalculateChecksum: Calculates the 8-bit Fletcher Checksum value for 'buffer' of 'length'. ----------
void IPC_CalculateCheckSum(unsigned char *buffer, unsigned char length, unsigned char *chkSumA, unsigned char *chkSumB)
{
    unsigned char i;
    *chkSumA = 0; *chkSumB = 0;                         // Initialize the checksum return values
    for (i=0;i<length;i++) {  *chkSumA += *buffer++;  *chkSumB += *chkSumA;  }   // Update the checksum results for each element in buffer
}
// -- IPC_VerifyChecksum: Verifies the checksum for the received packet to ensure integrity. ----------
bool IPC_VerifyCheckSum(unsigned char *buffer, unsigned char length)
{
    unsigned char chkSumA, chkSumB;
    IPC_CalculateCheckSum(buffer, length - 2, &chkSumA, &chkSumB);  // Calculate the checksum for the received packet
    buffer += (length - 2);                             // Point the buffer to the checksum portion of the packet
    return ((*buffer++ == chkSumA) && (*buffer == chkSumB));    // Do the two checksums match?
}
// == IPC COMMAND PROCESSING ==========================================================================
// -- IPC Command: SendACK: Loads the SSPBUF register with the status (with ACK set) and toggles the handshake to start a transfer. ----------
void IPCCmd_SendACK(void)
{
    ipcStatusReg.ACK = true;                            // Signal the ACK response
    SSPBUF = ipcStatusReg.value;                        // Copy the IPC status into SSPBUF so it can be read by the dsPIC
    IPC_HS_Pin = !IPC_HS_Pin;                           // Toggle SRDY pin to show that data has been received
}
// -- IPC Command: UpdateStatus: Receives a measurement and status flags update from the dsPIC processor. ----------
void IPCCmd_UpdateStatus(void)
{
    int i;
    for (i=0;i<75;i++) currMeasurements.Bytes[i] = ipcBuffer[i + 2];    // Copy the received status data into the currMeasurements structure
    ProcControl_UpdateIOFlags(currMeasurements.Bytes[1]);   // Update the external IO flags
    ipcStatusReg.Response = false;                      // There are no parameters to send back to the dsPIC
    IPCCmd_SendACK();                                   // Acknowledge the received command
}
// -- IPC Command: GetProfilerStatus: Returns the Power Profiler status to the dsPIC processor. ----------
void IPCCmd_GetProfilerStatus(void)
{
    int i, idx; TMyFloat temp;
    idx = 0;
    ipcBuffer[idx++] = 10;                              // Specify the data length in ipcBuffer[0]
    ipcBuffer[idx++] = procControl.Value;               // Copy the Control processor status bits
    ipcBuffer[idx++] = 0;                               // Flags (not used presently but gives space for expansion)
    temp.Value = pwrVS;                                 // Copy the supply voltage into a temporary structure for byte access
    for (i=0;i<4;i++) ipcBuffer[idx++] = temp.Bytes[i];  // Copy the byte representation to the ipcBuffer
    temp.Value = pwrVBatt;                              // Copy the battery voltage into a temporary structure for byte access
    for (i=0;i<4;i++) ipcBuffer[idx++] = temp.Bytes[i];  // Copy the byte representation to the ipcBuffer
```

```
    IPC_CalculateCheckSum(&ipcBuffer[0], 11, &ipcBuffer[11], &ipcBuffer[12]);      // Calculate the checksum for the data to send back
    ipcBuffTxLength = 13;                                                          // Set the transmission length
    ipcBuffTxIdx = 0;                                                              // Point to the first character to send in ipcBuffer
    ipcStatusReg.Response = false;                                                 // Indicate to the dsPIC that there are more bytes to be read
    ipcStatus.MsgResponse = true;                                                  // Signal the next data is in response to a request
    IPCCmd_SendACK();                                                              // ACK the dsPIC indicating the transfer can continue
}
// -- IPC Command: GetUSBData: Returns the received USB packet data to the dsPIC processor by placing it in ipcBuffer. ----------
void IPCCmd_GetUSBData(void)
{
    int i, idx = 1;
    ipcBuffer[0] = usbRxCount;                                                     // Specify the data length in ipcBuffer[0]
    for (i=0;i<usbRxCount;i++) ipcBuffer[idx++] = USB_In_Buffer[i];                // Copy the USB packet to ipcBuffer to send to the dsPIC
    IPC_CalculateCheckSum(&ipcBuffer[0], usbRxCount + 1, &ipcBuffer[idx], &ipcBuffer[idx + 1]);
    ipcBuffTxLength = usbRxCount + 3;                                              // Set the transmission length
    ipcBuffTxIdx = 0;                                                              // Point to the first character to send in ipcBuffer
    ipcStatusReg.Response = false;                                                 // Indicate to the dsPIC that there are more bytes to be read
    ipcStatus.MsgResponse = true;                                                  // Signal the next data is in response to a request
    IPCCmd_SendACK();                                                              // ACK the dsPIC indicating the transfer can continue
}
// -- IPC Command: SendUSBData ----------
// Sends the data received from the dsPIC to the PC via the USB port. A max of 64 bytes at a time can be sent so multiple transfers are performed if required.
void IPCCmd_SendUSBData(void)
{
    unsigned char bytesTx, i;  unsigned short txPtr;  TMyU16 pktRemaining;
    pktRemaining.Bytes[0] = ipcBuffer[3]; pktRemaining.Bytes[1] = ipcBuffer[4];    // Determine the amount of data to send
    txPtr = 2;                                                                     // Point to the first data byte in the stream
    while (pktRemaining.Value > 0)                                                 // Loop until all data has been sent
    {
        if (pktRemaining.Value > 64)                                               // A maximum of 64 bytes can be sent per transfer
            bytesTx = 64;
        else
            bytesTx = pktRemaining.Value;
        for (i=0;i<bytesTx;i++) USB_Out_Buffer[i] = ipcBuffer[txPtr++];            // Write the data to the USB out buffer
        usbTxCount = bytesTx;                                                      // Signal to the USB task the amount of data to send
        while (usbTxCount > 0) USB_Processor();                                    // Wait for the data to be sent by the USB stack
        pktRemaining.Value -= bytesTx;                                             // Determine how many bytes still need to be sent
    }
}
// -- IPC: CmdReceived: Verifies the integrity of a command received from the dsPIC and processes it if valid. ----------
void IPCCmd_CmdReceived(void)
{
    if (IPC_VerifyCheckSum(&ipcBuffer[0], ipcBuffRxIdx) == true)
    {
        switch (ipcBuffer[0]) {
            // -- Communications verification --
            case IPCCMD_PING                : IPCCmd_SendACK(); break;
            // -- Power Profiler Status --
            case IPCCMD_MEASUREMENTS        : IPCCmd_UpdateStatus(); break;
            case IPCCMD_GETPIC18FSTATUS     : IPCCmd_GetProfilerStatus(); break;
            // -- USB data access --
            case IPCCMD_GETUSBPACKET        : IPCCmd_GetUSBData(); break;
            case IPCCMD_SENDUSBPACKET       : IPCCmd_SendUSBData(); break;
        };
    }
}
// == IPC TASK PROCESSOR ==========================================================================================
// -- SPI: CheckErrors: Checks the SPI control and status registers for errors and takes necessary action. ----------
void IPC_CheckSPIErrors(void)
{
    char temp;
    if (SSPCON1bits.SSPOV) SSPCON1bits.SSPOV = False;                              // Has a receive overflow occurred?
    if (SSPCON1bits.WCOL)                                                          // Has a write collision occurred?
    {
        temp = SSPBUF;                                                             // Dummy read to clear register
        SSPCON1bits.WCOL = false;
    }
}
// -- IPC: Processor ----------
// Inter-Processor Communications Task Processor. Checks for errors, processes received commands and notifies the dsPIC if data is waiting.
void IPC_Processor(void)
{
    IPC_CheckSPIErrors();                                                          // Check for errors on the SPI interface
    if (ipcStatus.MsgRx)                                                           // Check if a command from the dsPIC has been received
    {
        IPCCmd_CmdReceived();
```

```
        ipcStatus.MsgRx = false;
    }
    if (IPC_SS_Pin == 1)                                    // Verify operation of IPC interface when de-selected
    {
        if (ipcStatus.Active) IPC_MessageEnd();             // The IPC cannot be active if the PIC18F is not selected
        if ((ipcStatus.USBMsgWaiting) && (IPC_HS_Pin == 0))  // Must the dsPIC be signalled that USB data is ready for it?
        {
            ipcStatusReg.USBMsgWaiting = true;              // Mark that a USB message is waiting
            SSPBUF = ipcStatusReg.value;                    // Copy the IPC status register into SSPBUF
            IPC_HS_Pin = 1;                                 // Signal the dsPIC that there is a status change
        }
    }
    if (ipcStatus.MsgSendDone)                              // If a transfer to the dsPIC is complete then initialize the
    {                                                       // necessary flags
        ipcStatusReg.USBMsgWaiting = false;
        ipcStatus.USBMsgWaiting = false;
        ipcStatus.DSCSignalled = false;
        usbDataWaiting = false;
        ipcStatus.MsgSendDone = false;
    }
}
```

```
/*== COMMS_USB.H & COMMS_USB.C ==========================================================================
  Uses the Microchip Application Library (2010-10-18) USB CDC function to implement a virtual serial port interface to the host PC.
==========================================================================================================*/

/*== COMMS_USB.C ========================================================================================
#include "Comms_USB.h" #include "./USB/usb.h" #include "./USB/usb_function_cdc.h" #include "usb_config.h" #include "USB/usb_device.h"
#include "USB/usb.h"
// -- Global Variables ---------------------------------------------------------------------------------------------------------------------------------------------------------
#pragma udata
char USB_In_Buffer[64], USB_Out_Buffer[64];  unsigned char usbRxCount = 0, usbTxCount = 0;  bool usbDataWaiting = false;
// -- USB: Initialize: Calls the USB initialization tasks as required by the Microchip USB stack. ------------------------------------------------------------------------------
void USB_Initialize(void)  {  USBDeviceInit();  }
// -- USB: Processor: Executes the USB stack tasks and checks for data to transmit or receive. ----------------------------------------------------------------------------------
void USB_Processor(void)
{
   USBDeviceTasks();
   // -- If the USB port is not connected then return to normal program --
   if ((USBDeviceState < CONFIGURED_STATE) || (USBSuspendControl == 1)) return;
   // -- Check for received data on the USB CDC interface --
   if (usbDataWaiting == false)
   {
      usbRxCount = getsUSBUSART(USB_In_Buffer,64);
      if (usbRxCount > 0)
         usbDataWaiting = true;
   }
   // -- Check for data to transmit --
   if ((USBUSARTIsTxTrfReady()) && (usbTxCount > 0))
   {
      putUSBUSART(&USB_Out_Buffer[0], usbTxCount);
      usbTxCount = 0;
   }
   CDCTxService();
}
// == Microchip Stack Support Routines ===================================================================
void USBCBSuspend(void) { /* Not used */ }
void USBCBWakeFromSuspend(void) { /* Not used */ }
void USBCB_SOF_Handler(void) { /* Not used */ }
void USBCBErrorHandler(void) { /* Not used */ }
void USBCBCheckOtherReq(void) { /* Not used */ }
void USBCBStdSetDscHandler(void) { /* Not used */ }
void USBCBInitEP(void)  {  CDCInitEP();  }
void USBCBSendResume(void)
{
   static WORD delay_count;
   if(USBGetRemoteWakeupStatus() == TRUE)
   {
      // Verify that the USB bus is in fact suspended, before we send remote wakeup signalling.
      if(USBIsBusSuspended() == TRUE)
      {
         USBMaskInterrupts();
         // Clock switch to settings consistent with normal USB operation.
         USBCBWakeFromSuspend();
         USBSuspendControl = 0;
```

```
            USBBusIsSuspended = FALSE;                  // So we don't execute this code again, until a new suspend condition
            delay_count = 3600U;                        // 5ms delay for USB 2.0 specification compliance
            do { delay_count--; }while(delay_count);
            // Now drive the resume K-state signalling onto the USB bus.
            USBResumeControl = 1;                       // Start RESUME signaling
            delay_count = 1800U;                        // Set RESUME line for 1-13 ms
            do { delay_count--; }while(delay_count);
            USBResumeControl = 0;                       // Finished driving resume signalling
            USBUnmaskInterrupts();
        }
    }
}
#if defined(ENABLE_EP0_DATA_RECEIVED_CALLBACK)
void USBCBEP0DataReceived(void) { }
#endif
BOOL USER_USB_CALLBACK_EVENT_HANDLER(USB_EVENT event, void *pdata, WORD size)
{
    switch(event)
    {
        case EVENT_TRANSFER: break;
        case EVENT_SOF: USBCB_SOF_Handler(); break;
        case EVENT_SUSPEND: USBCBSuspend(); break;
        case EVENT_RESUME: USBCBWakeFromSuspend(); break;
        case EVENT_CONFIGURED:  USBCBInitEP(); break;
        case EVENT_SET_DESCRIPTOR: USBCBStdSetDscHandler(); break;
        case EVENT_EP0_REQUEST: USBCBCheckOtherReq(); break;
        case EVENT_BUS_ERROR: USBCBErrorHandler(); break;
        case EVENT_TRANSFER_TERMINATED: break;
        default: break;
    }
    return TRUE;
}
```

```
/*==  LCD.H & LCD.C  ============================================================================================
  Interface implementation for a 4x20 LCD module.
==============================================================================================================*/

// ==  LCD.C  ================================================================================================
#include "LCD.h"  #include <math.h>  #include <stdio.h>
// -- LCD_Initialize: Initialize the LCD for operation.  ---------------------------------------------------------
void LCD_Initialize(void)
{
    unsigned long temp;
    LCD_Command(0x38, False);                   // Set interface length to 8-bits, 4 lines, 5x8 dots
    for (temp=0;temp<65535;temp++) Nop();       // Necessary delay for command to be processed
    LCD_Command(0x0C, True);                    // Set display on, cursor off, blink off
    for (temp=0;temp<65535;temp++) Nop();       // Necessary delay for command to be processed
    LCD_Command(0x06, True);                    // Set entry mode to Increment, no shift of display
    for (temp=0;temp<65535;temp++) Nop();       // Necessary delay for command to be processed
    LCD_Command(0x01, True);                    // Clear display and return cursor to posn 0
    for (temp=0;temp<65535;temp++) Nop();       // Necessary delay for command to be processed
}
// -- LCD_Read: Reads a byte from the LCD module.  ---------------------------------------------------------------
unsigned char LCD_Read(void)
{
    LCD_DataDir = 0xFF;                         // Specify parallel port as an input
    LCD_RS = false;                             // Select the control register, not data buffer
    LCD_RW = true;                              // Specify read operation
    Nop(); Nop();                               // Necessary delay for LCD module
    LCD_E = true;                               // Enabled LCD interface
    Nop(); Nop(); Nop(); Nop();                 // Necessary delay for LCD module
    LCD_E = false;                              // Disable LCD interface
    Nop(); Nop();                               // Necessary delay for LCD module
    return LCD_Data;                            // Read the value from the LCD
}
// -- LCD_WaitForIdle: Reads the status register of the LCD module and returns the state of the busy flag. -------
void LCD_WaitForIdle(void)  {  while ((LCD_Read() & 0x80) == 0x80);  }
// -- LCD_Command: Writes a command to the LCD module.  ----------------------------------------------------------
void LCD_Command(U8 cmd, U8 checkBusy)
{
    if (checkBusy) LCD_WaitForIdle();
    LCD_RS = false;                             // Select the command register
    LCD_DataDir = 0x00;                         // Parallel port is an output
    LCD_Data = cmd;                             // Write the command
    LCD_RW = false;                             // Specify a write operation
```

```
    // Delay for operation to take place
    Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop();
    Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop();
    LCD_E = true;                                                   // Select the LCD module
    // Delay for operation to take place
    Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop();
    Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop();
    LCD_E = false;                                                  // De-select the LCD module
}
// -- LCD_Move: Specifies the display write position. ----------
void LCD_Move(U8 line, U8 posn)
{
    U8 cursor;
    switch (line)                                                   // Select the correct register position for the specified line
    {
        case 1 : cursor = 0x00; break;    case 2 : cursor = 0x40; break;
        case 3 : cursor = 0x14; break;    case 4 : cursor = 0x54; break;
    }
    cursor += (posn - 1);
    LCD_Command(0x80 + cursor, true);                               // Command the LCD module to move its cursor
}
// -- LCD_Char: Writes a character to the LCD display. The position has already been set using LCD_Move. ----------
void LCD_Char(U8 value)
{
    LCD_WaitForIdle();
    LCD_RS = true;                                                  // Select the data register (display)
    LCD_DataDir = 0x00;                                             // Parallel port is an output
    LCD_RW = false;                                                 // Specify a write
    // Delay for operation to take place
    Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop();
    Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop();
    LCD_Data = value;                                               // Output the value to the LCD module
    LCD_E = true;                                                   // Select the LCD module
    // Delay for operation to take place
    Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop();
    Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop();
     LCD_E = false;                                                 // De-select the LCD module
}
// -- LCD_PutString: Writes a string 'buffer' to the LCD display at 'line, posn'. ----------
void LCD_PutString(U8 line, U8 posn, const char rom *buffer, U8 clearLine)
{
    U8 cntr;
    if (clearLine)
    {
        LCD_Move(line, 1);                                          // Move the LCD cursor to the start of the specified line
        for (cntr=1;cntr<posn;cntr++) LCD_Char(' ');                // Write the specified number of blanks to clear the chars
    }
    cntr = posn;
    while(*buffer != 0) {  LCD_Char(*buffer);  buffer++;  cntr++;  }    // Write the string characters to the LCD module
    if (clearLine) while (cntr < 21) {  LCD_Char(' ');  cntr++;  }      // Clear any remaining characters left on the line
}
// -- LCD_PutInt: Displays an integer value on the LCD display. ----------
unsigned char LCD_PutInt(U8 line, U8 posn, U16 value, U8 padDigits)
{
    U8 i, data[5], visibleDigits = 1;
    for (i=0;i<4;i++) data[i] = 0;
    if (value >= 10000) data[0] = value / 10000;
    if (value >= 1000) data[1] = (value - (data[0] * 10000)) / 1000;
    if (value >= 100) data[2] = (value - (data[0] * 10000) - (data[1] * 1000)) / 100;
    if (value >= 10) data[3] = (value - (data[0] * 10000) - (data[1] * 1000) - (data[2] * 100)) / 10;
    data[4] = (value - (data[0] * 10000) - (data[1] * 1000) - (data[2] * 100) - (data[3] * 10));
    for (i=0;i<4;i++) if (data[i] != 0) break;
    visibleDigits = 5 - i;
    if (visibleDigits < padDigits) visibleDigits = padDigits;
    LCD_Move(line, posn);
    i = 5 - visibleDigits;
    while (i < 5) LCD_Char(0x30 + data[i++]);
    return (posn + visibleDigits);
}
// -- LCD_PutInt_8bit: Displays an 8-bit value on the LCD display. ----------
unsigned char LCDPutInt_8bit(U8 line, U8 posn, U8 value, U8 padDigits)
{
    unsigned char data[3], visibleDigits = 1;
    data[0] = value / 100;  data[1] = (value - (data[0] * 100)) / 10;  data[2] = value - (data[0] * 100) - (data[1] * 10);
    LCD_Move(line, posn);
    if (data[0] != 0)
```

```
    visibleDigits = 3;
  else
    if (data[1] != 0)
      visibleDigits = 2;
  if (visibleDigits < padDigits) visibleDigits = padDigits;
  switch (visibleDigits)
  {
    case 2 : LCD_Char(0x30 + data[1]); break;
    case 3 : LCD_Char(0x30 + data[0]); LCD_Char(0x30 + data[1]); break;
  }
  LCD_Char(0x30 + data[2]);
  return (posn + visibleDigits);
}
// -- LCD_PutFloat: Displays a floating point value on the LCD module. ----------
void LCD_PutFloat(float value, U8 line, U8 posn, U8 intChars, U8 precision)
{
  char tempStr[10]; U8 cntr; float fint, ffrac;
  ffrac = modf(value, &fint);
  for (cntr=0;cntr<precision;cntr++) ffrac *= 10;
  LCD_PutInt(line, posn, (unsigned short)fint, intChars);          // Write integer portion to display
  LCD_Move(line, posn + intChars);
  LCD_Char('.');                                                   // Write fractional portion to display
  LCD_PutInt(line, posn + intChars + 1, (unsigned short)ffrac, precision);
}
```

```
/*== PROCCOMMS.H & PROCCOMMS.C ==========================================================
  Software task that control communications. Initializes the inter-processor SPI communications and USB interfaces and then executes their processors.
==========================================================*/

// == PROCCOMMS.C ========================================================================
#include "procComms.h"  #include "Comms_IPC.h"  #include "Comms_USB.h"
// -- Communications: Initialize: Initializes the communications interface sub-tasks. ----------
void ProcComms_Initialize(void)
{
  IPC_Initialize();                                                // SPI communications with the dsPIC processor
  USB_Initialize();                                                // USB port as a serial device emulator
}
// -- Communications: Processor: Calls the sub-tasks and checks for USB data packets that need to be sent to the dsPIC processor. ----------
void ProcComms_Processor(void)
{
  IPC_Processor();
  USB_Processor();
  if (usbDataWaiting) ipcStatus.USBMsgWaiting = true;              // Is there USB data to send to the dsPIC?
}
```

```
/*== PROCCONTROL.H & PROCCONTROL..C ======================================================
  Software task for power control, IO and status checking.
==========================================================*/
#include "HardwareProfile.h"
// -- Constants and Type Definitions ----------
// -- tprocControl: Control Task state and flags used for status reporting to the dsPIC processor. ----------
typedef union {
  struct {
    unsigned Fault          : 1;     unsigned Tick100ms      : 1;
    unsigned VSupplyLow     : 1;     unsigned VBattLow       : 1;
    unsigned VRegFault      : 1;     unsigned VRCFault       : 1;
    unsigned VRFFault       : 1;     unsigned Updated        : 1;
  };
  unsigned char Value;
} tprocControl;
// -- tDateTime: Represents the current time-of-day used in the Power Profiler. ----------
typedef union {
  struct { unsigned char Year, Month, Day, DayOfWeek, Hrs, Min, Sec; };
  unsigned char b[7];
} tDateTime;
// -- tIOFlags: The LSB is the output of the external shift register and the MSB contains the update flag to signal a change. ----------
typedef union {
  struct {
    unsigned LCDBacklightOn : 1;     unsigned RCPowerOn      : 1;
    unsigned RFPowerOn      : 1;     unsigned LEDACOn        : 1;
    unsigned LEDFault       : 1;     unsigned LEDLinkActive  : 1;
    unsigned BatteryOn      : 1;     unsigned BattChargeOn   : 1;
    unsigned Updated        : 1;     unsigned Unused         : 7;
  };
```

```
    unsigned char Bytes[2];
    unsigned short Value;
} tIOFlags;
// -- tMeasurements: Represents the latest status, time-of-day and measurement values sent from the dsPIC processor. ----------
typedef union {
    struct {
        struct {
            unsigned ProfilerMode       : 2;        unsigned VoltageMode        : 4;
            unsigned TODValid           : 1;        unsigned Unused             : 1;
            unsigned LCDBacklightOn     : 1;        unsigned RCPowerOn          : 1;
            unsigned RFPowerOn          : 1;        unsigned LED_ACOn           : 1;
            unsigned LED_Fault          : 1;        unsigned LED_LinkActive             : 1;
            unsigned BatteryOn          : 1;        unsigned ChargerOn          : 1;
        } Flags;
        tDateTime Clock;
        unsigned char Unused;
        float VRMS[3], IRMS[12];
        signed char Temp;
        float Freq;
    };
    unsigned char Bytes[75];
} tMeasurements;

// ==  PROCCONTROL.C  ================================================================================================
#include "procControl.h"
// -- Global Variables ---------------------------------------------------------------------------------------------------
float pwrVS, pwrVBatt, pwrVDig, pwrVSig, pwrVRC, pwrVRF;
float pwrVSThreshold, pwrVBattThreshold, pwrVDigThreshold, pwrVSigThreshold, pwrVRCThreshold, pwrVRFThreshold;
tIOFlags ioFlags;  tMeasurements currMeasurements;  tprocControl procControl;
// -- Task: Control: Initialize: Initializes the Task and IO status registers and configures the battery charger interface pins. ----------
void ProcControl_Initialize(void)
{
    // -- Set the default power threshold levels --
    pwrVSThreshold = 5.3;
    pwrVBattThreshold = 6.1;
    pwrVDigThreshold = 5.2;
    pwrVSigThreshold = 5.2;
    pwrVRCThreshold = 3.7;
    pwrVRFThreshold = 3.2;
    // -- Initialize the Task and IO variables --
    procControl.Value = 0;
    ioFlags.Value = 0;
    ioFlags.Updated = 1;
    // -- Initialize port pins for battery charger control --
    BATTCHG_EN_TRIS = 0;
    BATTCHG_Status_TRIS = 1;
}
// -- Task: Control: Processor: Checks the Power Profiler 'health' and updates the IO control when required. ----------
void ProcControl_Processor(void)
{
    if (procControl.Tick100ms)
    {
        ProcControl_CheckVoltages();                    // Check the operational voltages are ok
        procControl.Tick100ms = false;
    }
    if (ioFlags.Updated)
    {
        ProcControl_SetIORegister(ioFlags.Bytes[0]);    // Update the external IO
        BATTCHG_EN_Pin = ioFlags.BattChargeOn;          // Enable the battery charger if set
        ioFlags.Updated = false;
    }
}
// -- ProcControl: UpdateIOFlags: If the IO 'value' is different to the currently used value, the 'Updated' flag is set to trigger an update. ----------
void ProcControl_UpdateIOFlags(unsigned char value)
{
    if (ioFlags.Bytes[0] != value)
    {
        ioFlags.Bytes[0] = value;
        ioFlags.Updated = true;
    }
}
// -- ProcControl: CheckVoltages: Measures the operational voltages of the Power Profiler and compares against the preset thresholds. ----------
void ProcControl_CheckVoltages(void)
{
    pwrVS = ADC_MeasureChannel(0) * 0.0079444;      // Main Supply Voltage
    procControl.VSupplyLow = (pwrVS < pwrVSThreshold);
```

```
    pwrVBatt = ADC_MeasureChannel(1) * 0.01348624;                    // Battery voltage
    procControl.VBattLow = (pwrVBatt < pwrVBattThreshold);
    pwrVDig = ADC_MeasureChannel(2) * 0.0079444;                      // Digital regulator
    pwrVSig = ADC_MeasureChannel(3) * 0.0079444;                      // Analogue regulator
    procControl.VRegFault = ((pwrVDig < pwrVDigThreshold) || (pwrVSig < pwrVSigThreshold));
    pwrVRC = ADC_MeasureChannel(5) * 0.0079444;                       // Remote Communications regulator
    procControl.VRCFault = (pwrVRC < pwrVRCThreshold);
    pwrVRF = ADC_MeasureChannel(6) * 0.0079444;                       // RF Communications regulator
    procControl.VRFFault = (pwrVRF < pwrVRFThreshold);
}
// -- ProcControl: SetIORegister: Clocks 'value' out to the external shift register and latches the output when done. ----------
void ProcControl_SetIORegister(unsigned char value)
{
    unsigned char cntr;
    SelReg_DataDir = ddOutput;
    SelReg_ClkDir = ddOutput;
    for (cntr=0;cntr<8;cntr++)
    {
        SelReg_Data = ((value & 0x80) == 0x80); value <<= 1;          // Output the MSB
        SelReg_Clk = 1; SelReg_Clk = 0;                               // Toggle the clock to clock in the new bit
    }
    SelReg_RCLK = 1; SelReg_RCLK = 0;                                 // Toggle the ripple clock to latch the values
}
```

```
/*==  PROCDISP.H & PROCDISP.C ==========================================================================================
   Manages the LCD disply and cycles the current measurement values.
==========================================================================================================================*/
// -- Constants and Type Definitions ----------------------------------------------------------------------------------------
// -- Type definition: tProcControlStatus: General task flags structure. ----------------------------------------------------
typedef union {
    struct {
        unsigned Fault              : 1;        unsigned Tick100ms      : 1;        unsigned Tick1sec       : 1;
    };
    unsigned char value;
} tprocDispStatus;
extern tprocDispStatus procDisp;

/*==  PROCDISP.C ======================================================================================================*/
#include "procDisp.h"               #include "LCD.h"                  #include "procControl.h"
tprocDispStatus procDisp;
unsigned char dispTick100ms = 0, unsigned char secTickCount = 0, unsigned char lcdScreen = 0;
// -- ProcDisp_Initialize: Initializes the display task Type definition: tProcControlStatus: General task flags structure. ----
void ProcDisp_Initialize(void) {    LCD_Initialize();           }
// -- ProcDisp_Processor: Main task routine for updating the LCD display. ---------------------------------------------------
void ProcDisp_Processor(void)
{
    if (procDisp.Tick100ms)
    {
        if (dispTick100ms++ > 10)
        {
            procDisp.Tick1sec = true;
            dispTick100ms = 0;
        }
        if (dispTick100ms == 7) Display_UpdateClock();
        procDisp.Tick100ms = 0;
    }
    if (procDisp.Tick1sec)
    {
        if (secTickCount++ > 3)
        {
            Display_Update();
            secTickCount = 0;
        }
        procDisp.Tick1sec = false;
    }
}
// -- Display_Update: Updates the display page of the LCD screen with the relevant messages. The 'lcdScreen' integer specified which screen to display. ----------
void Display_Update(void)
{
    switch (lcdScreen++) {
        case 0: LCD_PutString(2, 1, "None 1", True);
                LCD_PutString(3, 1, "None 2", True);
                LCD_PutString(4, 1, "None 3", True);
                break;
        case 1: LCD_PutString(2, 1, "L1=   . V  f=  .  Hz", True);
```

```
            LCD_PutString(3, 1, "L2=   . V", True);
            LCD_PutString(4, 1, "L3=   . V  Temp=  °C", True);
            LCD_PutFloat(currMeasurements.VRMS[0], 2, 4, 3, 1);
            LCD_PutFloat(currMeasurements.VRMS[1], 3, 4, 3, 1);
            LCD_PutFloat(currMeasurements.VRMS[2], 4, 4, 3, 1);
            LCD_PutFloat(currMeasurements.Freq, 2, 14, 2, 1);
            LCDPutInt_8bit(4, 17, currMeasurements.Temp, 2);
            break;
      case 2: LCD_PutString(2, 1, "I1=   . A  I4=   . A", True);
            LCD_PutString(3, 1, "I2=   . A  I5=   . A", True);
            LCD_PutString(4, 1, "I3=   . A  I6=   . A", True);
            LCD_PutFloat(currMeasurements.IRMS[0], 2, 4, 3, 1);
            LCD_PutFloat(currMeasurements.IRMS[1], 3, 4, 3, 1);
            LCD_PutFloat(currMeasurements.IRMS[2], 4, 4, 3, 1);
            LCD_PutFloat(currMeasurements.IRMS[3], 2, 15, 3, 1);
            LCD_PutFloat(currMeasurements.IRMS[4], 3, 15, 3, 1);
            LCD_PutFloat(currMeasurements.IRMS[5], 4, 15, 3, 1);
            //LCD_PutString(4, 1, "  -  -       :  :  ", True);
            break;
      case 3: LCD_PutString(2, 1, "I7=   . A I10=   . A", True);
            LCD_PutString(3, 1, "I8=   . A I11=   . A", True);
            LCD_PutString(4, 1, "I9=   . A I12=   . A", True);
            LCD_PutFloat(currMeasurements.IRMS[6], 2, 4, 3, 1);
            LCD_PutFloat(currMeasurements.IRMS[7], 3, 4, 3, 1);
            LCD_PutFloat(currMeasurements.IRMS[8], 4, 4, 3, 1);
            LCD_PutFloat(currMeasurements.IRMS[9], 2, 15, 3, 1);
            LCD_PutFloat(currMeasurements.IRMS[10], 3, 15, 3, 1);
            LCD_PutFloat(currMeasurements.IRMS[11], 4, 15, 3, 1);
            break;
   };
   if (lcdScreen == 4) lcdScreen = 0;
}
// -- Display_UpdateClock: Updates the clock on the current display screen. ----------
void Display_UpdateClock(void)
{
   // - Update clock on bottom line -
   LCD_PutInt(1, 1, currMeasurements.Clock.Day, 2);
   LCD_PutString(1, 3, "/", False);
   LCDPutInt_8bit(1, 4, currMeasurements.Clock.Month, 2);
   LCD_PutString(1, 6, "/", False);
   LCD_PutInt(1, 7, 2000 + currMeasurements.Clock.Year, 4);
   LCD_PutString(1, 11, "  ", False);
   LCD_PutInt(1, 13, currMeasurements.Clock.Hrs, 2);
   LCD_PutString(1, 15, ":", False);
   LCD_PutInt(1, 16, currMeasurements.Clock.Min, 2);
   LCD_PutString(1, 18, ":", False);
   LCD_PutInt(1, 19, currMeasurements.Clock.Sec, 2);
}
```

```
/*== TYPES.H ==========================================================================================
  Type definitions for use in PIC18F implementation.
==========================================================================================*/
// -- Constants and Type Definitions ----------
#define S8          signed char                 #define S16         signed short
#define S32         signed long                 #define U8          unsigned char
#define U16         unsigned short              #define U32         unsigned long
#define U64         unsigned long long          #define bool        unsigned char

typedef union TMyU16 { unsigned short Value; unsigned char Bytes[2]; } TMyU16;
typedef union TMyU32 { unsigned long Value; unsigned char Bytes[4]; } TMyU32;
typedef union TMyS32 { long Value; unsigned char Bytes[4]; } TMyS32;
typedef union TMyFloat { float Value; unsigned char Bytes[4]; } TMyFloat;
// -- Logical Conditions ----------
#define false       0                           #define False       0
#define true        1                           #define True        1
```

C# Classes and Configuration and Control Application

```
/* ==  PROFILER.BASE.CS  ========================================================================================
Implements the base class that integrates diagnostics messaging and exception generation.
=============================================================================================================*/
using System;
namespace Profiler
{
   // -- MessageType: Defines the message type and is useful for filtering messages. ----------
   public enum MessageType { Information, Warning, Error };
   // -- DiagMessage: Delegate used in parent application to process diagnostics messages. ----------
   public delegate void DiagMessage(string sender, MessageType messageType, string message);
   // -- Class: ProfilerBaseClass: Allows all inherited classes to support basic diagnostics message reporting ----------
   public abstract class ProfilerBaseClass
   {
      public string ClassName = "Undefined";
      public event DiagMessage DiagMessageEvent;
      protected void OnDiagMessageEvent(MessageType messageType, string message)
      { if (DiagMessageEvent != null) DiagMessageEvent(ClassName, messageType, message); }
      protected void OnDiagMessageEvent(string Value) { OnDiagMessageEvent(MessageType.Information, Value); }
   }
   // -- Class: BaseException: All exceptions used in the Power Profiler software will inherit from this exception ----------
   public class BaseException : Exception
   {
      public BaseException(string s)
         : base(s)
      { }
   }
}
```

```
/* ==  PROFILER.COMMS.INTERFACES.CS  ============================================================================
Implements the Bluetooth, GSM and USB interfaces and incorporates their protocol and medium handler.
=============================================================================================================*/
using System;
namespace Profiler.Comms.Interfaces
{
   // -- CommsInterface: Base class for all interface classes (Bluteooth, GSM etc) ----------
   public abstract class CommsInterface : ProfilerBaseClass
   {
      public abstract bool Command(byte cmd, byte[] payload, bool expectACK);
      public abstract bool Connect();
      public abstract bool Disconnect();
      public abstract byte[] Request(byte cmd, byte[] payload, int retnBytes);
   }
   // -- BluetoothInterface: A Bluetooth Serial Port Profile (SPP) implementation ----------
   public class BluetoothInterface : CommsInterface
   {
      protected SerialCommsPort commsPort = new SerialCommsPort();              // The serial port used for the Bluetooth SPP
      protected SPECCProtocol commsProtocol = new SPECCProtocol();              // The protocol required for comms with the Power Profiler
      public bool PortOpen { get { return commsPort.IsOpen; } }                 // Returns the state of the communications interface to the application
      // -- Constructor: Names the interface class for diagnostics messaging ----------
      public BluetoothInterface() { ClassName = "BTComms"; }
      // -- Destructor: Closes the communications port before freeing the class ----------
      ~BluetoothInterface() { commsPort.Close(); }
      // -- Connect: The Bluetooth connection is handled by the Bluetooth stack so a connection is not required. ----------
      public override bool Connect() { throw new Exception("Still to be implemented"); }
      // -- Disconnect: Closes the serial port. ----------
      public override bool Disconnect() { commsPort.Close(); return true; }
      // -- Open: Opens the SPP profile on port 'portName' at the specified baud rate. ----------
      public bool Open(string portName, int baudRate)
      {
         commsPort.Close();                                                     // Close the comm port in case it is already open
         commsProtocol.ProtocolTxEvent += ProtocolTx;                           // Call ProtocolTx when the protocol wants to transmit over the interface
         commsProtocol.CRCPolynomial = 0x8005;                                  // Specify the CRC polynomial uses by the SPECC protocol
         commsPort.CommsRxEvent += CommsPortRx;                                 // Call CommsPortRx on data receive
         commsPort.Open(portName, baudRate);                                    // Open the communications port
         return PortOpen;
      }
      // -- CommsPortRx: Called by the comms port when data is received. ----------
      protected void CommsPortRx(byte value)
      { commsProtocol.ReceiveByte(value); }                                     // Pass the received data to the protocol for processing
      // -- ProtocolTx: Event called by the protocol to transmit data over the comms port. ----------
      protected void ProtocolTx(byte[] value)
      { commsPort.Transmit(value); }                                            // Transmit the data over the serial port
      // -- Command: Issues a command to the Power Profiler using the protocol. ----------
      public override bool Command(byte cmd, byte[] payload, bool expectACK)
      { return commsProtocol.Command(cmd, payload, expectACK); }
```

```
    // -- Request: Issues a request to the Power Profiler using the protocol. ----------
    public override byte[] Request(byte cmd, byte[] payload, int retnBytes)
    {  return commsProtocol.Request(cmd, payload, retnBytes);  }
}
// -- GSMTCPInterface: Interface for GSM TCP/IP communications using the SPECC protocol. ----------
public class GSMTCPInterface : CommsInterface
{
    SPECCProtocol commsProtocol = new SPECCProtocol();                  // SPECCProtocol is used for Profiler communications
    TCPCommsPort tcpPort = new TCPCommsPort();                          // The physical interface is via a TCP Client
    public bool PortOpen { get { return tcpPort.IsOpen; } }             // Returns the state of the comms interface
    // -- GSMTCPInterface Constructor ----------
    public GSMTCPInterface()
    {
        ClassName = "GSMTCPComms";                                      // Name the class for diagnostics messaging
        commsProtocol.ResponseTime = 10000;                             // Default protocol response time for GSM is 10 seconds
    }
    // -- Connect: The GSM connection is handled by the Open command so this overridden command is not used. ----------
    public override bool Connect() { throw new Exception("Still to be implemented"); }
    // -- Disconnect: Close the TCP Client interface. ----------
    public override bool Disconnect() {  tcpPort.Close(); return true;  }
    // -- Open: Opens a TCP Client connection to the specified IP address and port. ----------
    public bool Open(string serverIP, int port)
    {
        tcpPort.Open(serverIP, port);                                   // Open the TCP connection on the specified IP and port
        commsProtocol.ProtocolTxEvent += ProtocolTx;                    // Specify ProtocolTx for data transmission handling
        tcpPort.CommsRxEvent += CommsPortRx;                            // Specify CommsPortRx for data reception handling
        commsProtocol.CRCPolynomial = 0x8005;                           // Specify the CRC Polynomial for SPECC protocol
        return PortOpen;
    }
    // -- ProtocolRx: Called when the TCP Client receives data. ----------
    protected void CommsPortRx(byte value)
    {  commsProtocol.ReceiveByte(value);  }                             // Pass the received data to the SPECC protocol
    // -- ProtocolTx: Called when the SPECC protocol wants to send data over the TCP link. ----------
    protected void ProtocolTx(byte[] value)
    {  tcpPort.Transmit(value);  }                                      // Send the data using the TCP Client interface
    // -- Command: Issue a command to the Profiler using the SPECC protocol. ----------
    public override bool Command(byte cmd, byte[] payload, bool expectACK)
    {  return commsProtocol.Command(cmd, payload, expectACK);  }
    // -- Request: Issue a request to the Profiler using the SPECC protocol. ----------
    public override byte[] Request(byte cmd, byte[] payload, int retnBytes)
    {  return commsProtocol.Request(cmd, payload, retnBytes);  }
}
// -- USB_VCPInterface: A USB Virtual-COM Port Communications Device Class implementation ----------
public class USB_VCPInterface : CommsInterface
{
    protected SerialCommsPort commsPort = new SerialCommsPort();        // The serial port used for the virtual COM port
    protected SPECCProtocol commsProtocol = new SPECCProtocol();        // The protocol required for comms with the Power Profiler
    // -- Externally accessible variables ----------
    public bool PortOpen { get { return commsPort.IsOpen; } }           // Returns the state of the communications interface to the application
    // -- Constructor: Names the class for diagnostics messaging ----------
    public USB_VCPInterface() {  ClassName = "USB_VCPComms";  }         // Names the class for diagnostics messaging
    // -- Destructor: Close the communications port before freeing the class ----------
    ~USB_VCPInterface() {  commsPort.Close();  }
    // -- Connect: Not required for the USB interface so an error is returned if called. ----------
    public override bool Connect() {  throw new Exception("Still to be implemented");  }
    // -- Open: Opens the virtual COM port 'portName' at the specified baud rate. ----------
    public bool Open(string portName, int baudRate)
    {
        commsPort.Close();                                              // Close the comm port in case it is already open
        commsProtocol.ProtocolTxEvent += ProtocolTx;                    // Call ProtocolTx when the protocol wants to transmit over the interface
        commsProtocol.CRCPolynomial = 0x8005;                           // Specify the CRC polynomial uses by the SPECC protocol
        commsPort.CommsRxEvent += CommsPortRx;                          // Call CommsPortRx on data receive
        commsPort.Open(portName, baudRate);                             // Open the communications port
        return PortOpen;
    }
    // -- CommsPortRx: Called by the comms port when data is received. ----------
    protected void CommsPortRx(byte value) {  commsProtocol.ReceiveByte(value);  }
    // -- ProtocolTx: Event called by the protocol to transmit data over the comms port. ----------
    protected void ProtocolTx(byte[] value)  {  commsPort.Transmit(value);  }   // Transmit the data over the serial port
    // -- Command: Issues a command to the Power Profiler using the protocol. ----------
    public override bool Command(byte cmd, byte[] payload, bool expectACK)
    {  return commsProtocol.Command(cmd, payload, expectACK);  }
    // -- Request: Issues a request to the Power Profiler using the protocol. ----------
    public override byte[] Request(byte cmd, byte[] payload, int retnBytes)
    {  return commsProtocol.Request(cmd, payload, retnBytes);  }
    // -- Disconnect: Close the serial port. ----------
```

```
        public override bool Disconnect() {  commsPort.Close(); return true;  }
    }
}
```

```
/* ==  PROFILER.COMMS.MEDIUMS.CS  ================================================================================
Implements the phsical interfaces such as serial port and TCP/IP Client access.
================================================================================================================*/
using System.Threading; using System.Net.Sockets; using System.IO.Ports; using Profiler.Base;
namespace Profiler.Comms.Mediums
{
    // -- onCommsRx: Delegate used in the parent class to process received data. --
    public delegate void onCommsRx(byte value);
    // -- SerialCommsPort: Implements a serial port interface. ----------------------------------------------------------
    public class SerialCommsPort : ProfilerBaseClass
    {
        static SerialPort serialPort;                                   // Serial port used for communications
        public event onCommsRx CommsRxEvent;                            // Event to call when data is received
        public bool IsOpen { get { return serialPort.IsOpen; } }        // Returns the serial port communications state
        // -- SerialPort: Constructor which creates the serial port and assigns the data receive handler. ---------------
        public SerialCommsPort()
        {
            serialPort = new SerialPort();
            serialPort.DataReceived += new SerialDataReceivedEventHandler(DataReceiveHandler);
        }
        // -- Open: Opens the specified serial port and configures its parameters. -------------------------------------
        public void Open(string portName, int baudRate)
        {
            serialPort.PortName = portName;                             // Specify which port to use based on user-specified value
            serialPort.BaudRate = baudRate;                             // Set the interface baud rate from the user-specified value
            serialPort.Parity = Parity.None;                            // Parity bits are not used
            serialPort.DataBits = 8;                                    // 8 data bits are used
            serialPort.StopBits = StopBits.One;                         // One stop bit
            serialPort.Handshake = Handshake.None;                      // RTS and CTS handshaking is not used
            serialPort.ReadTimeout = 500;                               // Set receive timeout
            serialPort.WriteTimeout = 500;                              // Set transmit timeout
            if (serialPort.IsOpen) serialPort.Close();                  // Close the serial port if it is already open
            serialPort.Open();                                          // Open the serial port using the new parameters
        }
        // -- Close: Closes the serial port if it is already open. ----------------------------------------------------
        public void Close() {  if (serialPort.IsOpen) serialPort.Close();  }
        // -- Transmit: Transmits the byte array over the serial port if open, otherwise raises an Exception to be handled by the host. ----------
        public void Transmit(byte[] value)
        {
            if (!serialPort.IsOpen) throw new BaseException("Serial port not open for transmission");
            serialPort.Write(value, 0, value.Length);
        }
        // -- DataReceiveHandler: Called by the serial port when data is ready to be read. The received bytes are passed on to the parent interface handler. ----------
        protected void DataReceiveHandler(object sender, SerialDataReceivedEventArgs e)
        {
            SerialPort sp = (SerialPort)sender;                         // Cast a serial port class to the sender for processing the received data
            while (sp.BytesToRead > 0)                                  // Read multiple bytes and pass each to the CommsRxEvent handler.
            {
                try
                {
                    if (CommsRxEvent != null)                           // Only call CommsRxEvent if it has actually been assigned
                    CommsRxEvent((byte)sp.ReadByte());                  // Read the value from the serial port and pass to the rx event handler
                }
                catch
                {
                    // Ignore errors for now as the protocol will support error detection at a packet level.
                }
            }
        }
    }
    // -- TCPCommsPort: Implements a TCP Client interface for communications with remote GSM modules. ----------------------
    public class TCPCommsPort : ProfilerBaseClass
    {
        bool enabled = true;                                            // Indicates to the listening thread if the class is still enabled
        TcpClient tcpClient = new TcpClient();                          // The .NET TcpClient class
        NetworkStream ns = null;                                        // Network streams are used to handle the transfer of data on a TcpClient
        public bool IsOpen { get { return tcpClient.Connected; } }      // Returns the state of the TCP Client connection
        public event onCommsRx CommsRxEvent;                            // Data receive handler to pass received bytes to the upper interface
        // -- Open: Opens a TCP Client connection with the remote Power Profiler on the specified IP and port. ---------------
        public void Open(string serverIP, int port)
        {
```

```
            tcpClient = new TcpClient(serverIP, port);                    // Create the TCP Client and open its connection
            ns = tcpClient.GetStream();                                   // Cast 'ns' as the TCP stream for data transmission / reception
            Thread listenThread = new Thread(new ThreadStart(ListenThread));  // Activate 'listenThread' which handles data reception from the stream.
            listenThread.Start();                                         // Start listenThread to handle received data
        }
        // -- Close: Closes the TCP Connection, network stream and terminates the listenThread by setting enabled as false. ----------
        public void Close()
        {
            enabled = false;                                              // Terminates the listenThread
            if (tcpClient.Connected)                                      // Only attempt to close the port if it is already open
        {
                ns.Close();                                               // Closes the network stream
                tcpClient.Close();                                        // Closes the TCP connection
            }
        }
        // -- ListenThread: A seperate thread to handle received data on the network stream associated with the TCP Client in use. ----------
        private void ListenThread()
        {
            while (enabled)                                               // Terminate the thread if it is no longer required
            {
                if (ns.DataAvailable)                                     // If data is available read from the network stream
                {
                    if (CommsRxEvent != null)                             // Only send the data to the receive event handler if it assigned
                        CommsRxEvent((byte)ns.ReadByte());                // Read a byte from the TCP/IP interface and send to the rx event handler
                }
                Thread.Sleep(1);                                          // Thread delay to minimize processor utilization
            }
        }
        // -- Transmit: Called by the parent interface class to transmit data over the network stream ----------
        public void Transmit(byte[] value)
        {
            if (!tcpClient.Connected) throw new BaseException("TCP connection not open for transmission");
            ns.Write(value, 0, value.Length);                             // Write the data to the network stream
        }
    }
}
```

```
/* ==  PROFILER.COMMS.PROTOCOLS.CS  ========================================================================================
Implementst the Single-Point Error-Checked (SPECC) protocol used to interface with the Power Profiler.
=============================================================================================================================*/
using System; using System.Collections.Generic; using System.Threading; using System.Net.Sockets; using Profiler.Base;
namespace Profiler.Comms.Procotols
{
    // -- Structure: SPECCPacket: Contains the members of the protocol for easy passing between methods. ----------
    public struct SPECCPacket
    {
        public byte TID;
        public byte Cmd;
        public byte[] Params;
    }
    // -- Delegate: ProtocolTx: Called in the interface class for data tranmission. ----------
    public delegate void ProtocolTx(byte[] Value);
    // -- Class: SPECCProtocol: Implements the error-checked communications protocol. ----------
    public class SPECCProtocol : ProfilerBaseClass
    {
        // -- Internal Variables ----------
        Boolean pktReceived = false;
        ushort crcPolynomial;
        protected int minResponseTime = 5000;
        protected List<byte> rxData = new List<byte>();
        SPECCPacket rxPacket = new SPECCPacket();
        private byte transactionID = 0;
        public int ResponseTime { set { minResponseTime = value; } get { return minResponseTime; } }
        public ushort CRCPolynomial { set { crcPolynomial = value; } }
        // -- Externally defined events ----------
        public event ProtocolTx ProtocolTxEvent;
        // -- Method: OnProtocolTxEvent: Calls the ProtocolTxEvent delegate if defined. ----------
        public void OnProtocolTxEvent(byte[] Value)
        { if (ProtocolTxEvent != null) ProtocolTxEvent(Value); else throw new Exception("Protocol Tx method not defined."); }
        // -- Method: ReceiveByte: Called by the interface class when a byte is received. ----------
        public Boolean ReceiveByte(byte value)
        {
            if ((rxData.Count == 0) && (value != 0x7E))
            {
                // Rubbish data received. Perhaps notify in a diagnostic.
```

```
    }
    else
        rxData.Add(value);
    if (rxData.Count > 2)
    {
        if (rxData.Count == (rxData[1] + rxData[2] * 256))                  // Has a full packet been received?
        {
            byte[] pkt = new byte[rxData.Count];                            // Construct a packet byte array of the required length
            for (int i = 0; i < pkt.Length; i++) pkt[i] = rxData[i];        // Copy the received packet from the List structure to pkt
            if (DecodePacket(pkt, out rxPacket)) pktReceived = true;        // Decode the received packet, verifying the CRC
            rxData.Clear();                                                 // Re-initialize the List structure so new packets can be received
            this.OnDiagMessageEvent("Packet received");                     // Send diagnostics message to show data was received
        }
    }
    return true;                                                            // Return value is used for hand-shaking, SPECC always returns true
}
// -- Method: CalculateCRC16: Calculates a 16-bit CRC on 'data' using 'polynomial' ---------------------------------------------------------------------
// This routine is a duplicate of that used in the dsPIC processor to ensure compatibility.
protected ushort CalculateCRC16(byte[] data, ushort polynomial)
{
    if (data.Length == 0) throw new BaseException("CRC packet too small");
    byte CRCH = (byte)(polynomial >> 8), CRCL = (byte)(polynomial & 0x0FF), CRCHigh = data[0], CRCLow, CRCDat1, Carry;
    if (data.Length < 2) CRCLow = 0; else CRCLow = data[1];
    for (int i = 1; i < (data.Length + 1); i++)
    {
        if ((data.Length > 2) && (i < (data.Length - 1))) CRCDat1 = data[i + 1]; else CRCDat1 = 0;
        for (int j = 0; j < 8; j++)
        {
            Carry = (byte)((CRCHigh & 0x80) >> 7);
            if (Carry > 0) {  CRCHigh = (byte)(CRCHigh ^ CRCH); CRCLow = (byte)(CRCLow ^ CRCL);  }
            CRCHigh = (byte)(CRCHigh << 1) + (byte)((CRCLow & 0x80) >> 7); CRCLow = (byte)(CRCLow << 1) + (byte)((CRCDat1 & 0x80) >> 7);
            CRCDat1 = (byte)(CRCDat1 << 1);
        }
    }
    return (ushort)((CRCHigh << 8) + CRCLow);
}
// -- Method: DecodePacket: Converts the received byte array into a SPECCPacket and verifies its CRC. ---------------------------------------------------
protected bool DecodePacket(byte[] pkt, out SPECCPacket value)
{
    value.Params = null; value.TID = 0; value.Cmd = 0;
    try
    {
        if (CalculateCRC16(pkt, crcPolynomial) != 0) return false;          // Verify CRC using internally set polynomial
        int idx = 0;
        if (pkt[idx++] != 0x7E) return false;                               // The first byte of the packet must be a '~'
        ushort pktLen = (ushort)((pkt[idx++] + pkt[idx++] * 256) - 7);      // Determine packet length
        Array.Resize(ref value.Params, pktLen);                             // Resize array
        value.TID = pkt[idx++];                                             // Transaction ID
        value.Cmd = pkt[idx++];                                             // Command
        Array.Copy(pkt, idx, value.Params, 0, value.Params.Length);         // Copy received parameters
    }
    catch {  return false;  }                                               // Return false on any error
    return true;
}
// -- Method: EncodePacket: Takes the SPECCPacket and returns a byte array to be sent via the interface class. ---------------------------------------
protected byte[] EncodePacket(SPECCPacket value)
{
    byte[] temp = new byte[value.Params.Length + 5];
    int idx = 0;
    temp[idx++] = 0x7E;                                                     // Start of transmission character
    Array.Copy(BitConverter.GetBytes(value.Params.Length + 7), 0, temp, idx, 2);
    idx += 2;                                                               // Packet length, LSB first
    temp[idx++] = value.TID;                                                // Transaction ID
    temp[idx++] = value.Cmd;                                                // Command
    Array.Copy(value.Params, 0, temp, idx, value.Params.Length);
    ushort crc = CalculateCRC16(temp, crcPolynomial);                       // Calculate CRC
    Array.Resize(ref temp, temp.Length + 2);                                // Add space for the generated CRC
    idx += value.Params.Length;
    temp[idx++] = (byte)(crc >> 8); temp[idx] = (byte)crc;                  // CRC must be sent MSB first
    return temp;
}
// -- Method: Command: Issues an instruction to the Power Profiler and will await an ACK if expected. ---------------------------------------------------
public bool Command(byte cmd, byte[] payload, bool expectACK)
{
    bool returnValue = false;
    SPECCPacket pkt = new SPECCPacket();                                    // Create a new SPECCPacket type to hold the command
```

```
            pkt.Cmd = cmd;                                          // Specify the command byte
            pkt.Params = payload;                                   // Copy the parameters to send into the packet
            pkt.TID = transactionID++;                              // Increment the transaction ID
            pktReceived = false;
            OnProtocolTxEvent(EncodePacket(pkt));                   // Transmit the packet using the interface class
            if (expectACK)
            {
                DateTime startTime = DateTime.Now;
                bool timed_out = false;
                while (!pktReceived && !timed_out)                  // Wait until the packet is received or the timeout occurs
                {
                    Thread.Sleep(20);                               // Sleep the thread while waiting to free up the processor
                    timed_out = ((DateTime.Now - startTime).TotalMilliseconds > minResponseTime);
                }
                if (pktReceived)                                    // If the packet is received check that the result is an ACK
                    returnValue = ((rxPacket.Cmd == cmd) && (rxPacket.Params[0] == 0x06));
            }
            return returnValue;
        }
        // -- Method: Request: Issues an instruction to the Power Profiler and waits for a response to process. ----------
        public byte[] Request(byte cmd, byte[] payload, int returnBytes)
        {
            SPECCPacket pkt = new SPECCPacket();                    // Create a new SPECCPacket type to hold the command
            pkt.Cmd = cmd;                                          // Specify the command byte
            pkt.Params = payload;                                   // Copy the parameters to send into the packet
            pkt.TID = transactionID++;                              // Increment the transaction ID
            pktReceived = false;
            OnProtocolTxEvent(EncodePacket(pkt));                   // Transmit the packet using the interface class
            DateTime startTime = DateTime.Now;
            bool timed_out = false;
            while (!pktReceived && !timed_out)                      // Wait until the packet is received or the timeout occurs
            {
                Thread.Sleep(20);                                   // Sleep the thread while waiting to free up the processor
                timed_out = ((DateTime.Now - startTime).TotalMilliseconds > minResponseTime);
            }
            if (pktReceived && (rxPacket.Cmd == cmd))               // If the packet is received verify the command and return the parameters
                return rxPacket.Params;
            else
                return null;
        }
    }
}
```

```
/* ==  PROFILER.DEVICES.CS  ================================================================================================
Implements the class interface for a Power Profiler Revision 1 device.
=======================================================================================================================*/
using System; using Profiler.Base; using Profiler.Comms.Interfaces;
namespace Profiler.Devices
{
    // -- Profiler parent class constructor ----------------------------------------------------------------------------------
    public class ProfilerDevice : ProfilerBaseClass { }
    // -- Profiler Revision 1 class ------------------------------------------------------------------------------------------
    public class Profiler_Rev1 : ProfilerDevice
    {
        // -- Class Constructor ----------------------------------------------------------------------------------------------
        public Profiler_Rev1()
        {
            MemoryCfg = new MemoryConfig(this);
            MeasConfig = new ProfileMeasurementConfig(this);
            ChannelCalCfg = new ChannelCalibrationConfig(this);
            EnergyICCfg[0] = new EnergyICConfig(this, CFG_ENIC1CAL_ADDR);           // Energy measurement IC 1 calibration data
            EnergyICCfg[1] = new EnergyICConfig(this, CFG_ENIC2CAL_ADDR);           // Energy measurement IC 2 calibration data
            EnergyICCfg[2] = new EnergyICConfig(this, CFG_ENIC3CAL_ADDR);           // Energy measurement IC 3 calibration data
            EnergyICCfg[3] = new EnergyICConfig(this, CFG_ENIC4CAL_ADDR);           // Energy measurement IC 4 calibration data
            MeasCalCfg_1PH2W = new MeasurementCalibrationConfig(this, CFG_MEAS1PH2W_ADDR);  // Create the measurement calibration variable
            MeasCalCfg_3PH3W = new MeasurementCalibrationConfig(this, CFG_MEAS3PH3W_ADDR);  // Create the measurement calibration variable
            MeasCalCfg_3PH4W = new MeasurementCalibrationConfig(this, CFG_MEAS3PH4W_ADDR);  // Create the measurement calibration variable
            commsConfig = new CommsConfig(this);
        }
        // -- Class Destructor -----------------------------------------------------------------------------------------------
        ~Profiler_Rev1() { // Not used at present }
        // == COMMUNICATION ===============================================================================================
        CommsInterface commsInterface = null;
        BluetoothInterface btInterface = new BluetoothInterface();
        GSMTCPInterface gsmInterface = new GSMTCPInterface();
```

```
USB_VCPInterface usbInterface = new USB_VCPInterface();
// -- Communication Interface Command Definitions ---------------------------------------------------------------------------------------------------------------------------
const byte CMD_PING = 0x01, CMD_RESET = 0x02, CMD_LOGIN = 0x03, CMD_LOGOFF = 0x04;
const byte CMD_GETCONFIG = 0x10, CMD_SETCONFIG = 0x11;
const byte CMD_GETCURRMEASUREMENTS = 0x20, CMD_SETVOLTAGECHANNELS = 0x21, CMD_SET_EM_MODES = 0x22;
const byte CMD_GET_EM_CALIBRATION = 0x23, CMD_SET_EM_CALIBRATION = 0x24;
const byte CMD_GETSTATUS = 0x30;
const byte CMD_SETCLOCK = 0x31;
const byte CMD_RESETACQUISITION = 0x32, CMD_STARTACQUISITION = 0x33, CMD_STOPACQUISITION = 0x34;
const byte CMD_GETMEMPTRS = 0x40;
const byte CMD_GETNEXTMEASUREMENT = 0x41;
// -- Communications Configuration Block ------------------------------------------------------------------------------------------------------------------------------------
public class CommsConfig : ConfigurationBlock
{
   public bool Bluetooth_Enabled, GSM_Enabled;
   public char[] GSM_APNName, GSM_SIMPIN;
   public UInt16 GSM_ListenPort;
   // -- Constructor: Initalizes the configuration class ----------------------------------------------------------------------------------------------------------------------
   public CommsConfig(Profiler_Rev1 parent) { profiler = parent; GSM_APNName = new char[15]; GSM_SIMPIN = new char[4]; }
   // -- Load: Retrieves the configuration from the connected Power Profiler ---------------------------------------------------------------------------------------------------
   public bool Load()
   {
      loaded = false; configValid = false;
      byte[] value = profiler.GetConfig(CFG_COMMS_ADDR, 1);                  // Read the configuration from the Profiler as a byte array
      loaded = true;
      int idx = 1;
      if ((value[idx++] == 0xAA) && (value[idx++] == 0xBB))                  // Check the the memory contents are valid
      {
         loaded = true; configValid = true;
         idx++;                                                              // Skip the first byte because it is for the sensor network (future use)
         Bluetooth_Enabled = ((value[idx] & 1) == 1); idx++;                 // Is the Bluetooth interface enabled
         GSM_Enabled = ((value[idx] & 1) == 1); idx++;                       // Is the GSM interface enabled
         for (int i = 0; i < 4; i++) GSM_SIMPIN[i] = (char)value[idx++];     // Get the SIM PIN from the value array
         for (int i = 0; i < 15; i++) GSM_APNName[i] = (char)value[idx++];   // Get the APN name from the value array
         GSM_ListenPort = (ushort)((value[idx++] << 8) + value[idx]);        // Get the listening port from the array
      }
      return loaded;
   }
   // -- Store: Creates the configuration block and writes it to the Power Profiler --------------------------------------------------------------------------------------------
   public bool Store()
   {
      byte[] cfgBlock = new byte[32];
      for (int i = 0; i < cfgBlock.Length; i++) cfgBlock[i] = 0;             // Initialize the configuation block
      int idx = 0;
      cfgBlock[idx++] = 0xAA; cfgBlock[idx++] = 0xBB;                        // Set the memory verifier
      // -- Copy the parameters to the configuration block array --
      cfgBlock[idx++] = 0;                                                   // Disable sensor network (future use)
      if (Bluetooth_Enabled) cfgBlock[idx++] = 1;                            // Bluetooth enabled flag
      if (GSM_Enabled) cfgBlock[idx++] = 1;                                  // GSM enabled flag
      for (int i = 0; i < 4; i++) cfgBlock[idx++] = (byte)GSM_SIMPIN[i];     // GSM SIM PIN code
      for (int i = 0; i < 15; i++) cfgBlock[idx++] = (byte)GSM_APNName[i];   // GSM APN name
      cfgBlock[idx++] = (byte)(GSM_ListenPort >> 8);
      cfgBlock[idx++] = (byte)(GSM_ListenPort & 0xFF);
      return profiler.SetConfig(CFG_COMMS_ADDR, 1, cfgBlock);                // Write the configuration to the Profiler
   }
}
// -- Connect_Bluetooth: Bluetooth Interface ---------------------------------------------------------------------------------------------------------------------------------
public void Connect_Bluetooth(string portName)
{
   if (btInterface.Open(portName, 115200))
      commsInterface = btInterface;
   else
      throw new Exception("Could not connect via Bluetooth");
}
// -- Connect_GSM: GSM TCP/IP Client Interface -------------------------------------------------------------------------------------------------------------------------------
public void Connect_GSM(string profilerIP, int port)
{
   if (gsmInterface.Open(profilerIP, port))
      commsInterface = gsmInterface;
   else
      throw new Exception("Could not connect via GSM TCP/IP");
}
// -- Connect_USB: USB Virtual COM Port Interface ----------------------------------------------------------------------------------------------------------------------------
public void Connect_USB(string portName)
{
   if (usbInterface.Open(portName, 115200))
```

```
            commsInterface = usbInterface;
        else
            throw new Exception("Could not connect via USB Virtual COM Port");
    }
    // -- Comms: Disconnct: Closes the current interface. ----------
    public void Disconnect() { commsInterface.Disconnect(); }
    // -- Comms: Login: Attempts to login to the Profiler using the specified ID and password. ----------
    public bool Login(UInt32 id, string pwd)
    {
        byte[] cmdparam = new byte[4 + 16];
        Array.Clear(cmdparam, 0, cmdparam.Length);
        Array.Copy(BitConverter.GetBytes(id), 0, cmdparam, 0, 4);
        int idx = 4;
        foreach (char c in pwd) cmdparam[idx++] = (byte)c;
        if (commsInterface.Command(CMD_LOGIN, cmdparam, true)) {  deviceID = id;  return true;  }
        return false;
    }
    // -- Logoff: Logs off the currently connected Power Profiler ----------
    public bool Logoff() { return commsInterface.Command(CMD_LOGOFF, null, true); }
    // -- Ping: Sends a ping request to the Profiler over the opened comms interface. ----------
    public bool Ping() { return commsInterface.Command(CMD_PING, new byte[0], true); }
    // == CONFIGURATION: DEVICE ==========
    UInt32 deviceID = 0;
    // -- Memory Definitions ----------
    private const int flashMarkerArea = 1056;
    private const int flashDiagnosticsSize = 1056 * 16;                                 // Allows: 512 x 33 byte messages
    private const int flashMemorySize = 3 * 8192 * 1056;                                // Each memory consists of 8192 pages of 1056 bytes
    private const int flashMemoryAvailable = flashMemorySize - flashDiagnosticsSize - flashMarkerArea;
    public int FLASHMemorySize { get { return flashMemorySize; } }
    public int FLASHMemoryAvailable { get { return flashMemoryAvailable; } }
    public int FLASHDiagnosticsSize { get { return flashDiagnosticsSize; } }
    public int FLASHMarkerSize { get { return flashMarkerArea; } }
    // -- Configuration block memory addresses and public access variables ----------
    const ushort CFG_PROFID_ADDR = 0, CFG_PROFMEAS_ADDR = 32, CFG_MEMORY_ADDR = 96, CFG_CHANCALIB_ADDR = 128;
    const ushort CFG_ENIC1CAL_ADDR = 352, CFG_ENIC2CAL_ADDR = 416, CFG_ENIC3CAL_ADDR = 480, CFG_ENIC4CAL_ADDR = 544;
    const ushort CFG_MEAS1PH2W_ADDR = 608, CFG_MEAS3PH3W_ADDR = 832, CFG_MEAS3PH4W_ADDR = 1056;
    const ushort CFG_MEASCUSTOM_ADDR = 1280;
    const ushort CFG_COMMS_ADDR = 1504;
    const ushort CFG_UNUSED_ADDR = 1536;
    public ProfileMeasurementConfig MeasConfig;
    public MemoryConfig MemoryCfg;
    public CommsConfig commsConfig;
    // -- GetConfig: Returns a byte array of the requested configuration block(s) from the connected Power Profiler ----------
    protected byte[] GetConfig(UInt16 memAddr, int blockCount)
    {
        byte[] param = new byte[3];
        Array.Copy(BitConverter.GetBytes(memAddr), 0, param, 0, 2);
        param[2] = (byte)blockCount;
        return commsInterface.Request(CMD_GETCONFIG, param, blockCount * 32);
    }
    // -- SetConfig: Writes the byte array 'values' to the specified memory address of the connected Power Profiler ----------
    protected bool SetConfig(UInt16 memAddr, int blockCount, byte[] values)
    {
        byte[] param = new byte[3 + values.Length];
        Array.Copy(BitConverter.GetBytes(memAddr), 0, param, 0, 2);
        param[2] = (byte)blockCount;
        Array.Copy(values, 0, param, 3, values.Length);
        return commsInterface.Command(CMD_SETCONFIG, param, true);
    }
    // -- ProfilerConfiguration class: Parent class for all configurations ----------
    public class ConfigurationBlock
    {
        protected Profiler_Rev1 profiler;
        protected bool loaded = false, configValid = false;
        public bool Loaded { get { return loaded; } }
        public bool Valid { get { return configValid; } }
    }
    // -- Profiler Configuration block implementation ----------
    // -- Profiler ID class: Contains the fields specified in the configuration block ----------
    public class ProfilerID : ConfigurationBlock
    {
        public UInt32 DeviceID;                             public char[] Password;
        public byte ManufacturerID;                         public UInt32 TODManufactured;
        // -- Constructor: Defines the parent Profiler ----------
        public ProfilerID(Profiler_Rev1 parent) { profiler = parent; }
        // -- Load: Retrieves the current configuration from the connected Power Profiler ----------
```

```
    public bool Load()
    {
        loaded = false; configValid = false;
        byte[] value = profiler.GetConfig(CFG_PROFID_ADDR, 1);                  // Read the configuration from the Profiler as a byte array
        loaded = true;  int idx = 1;
        if ((value[idx++] == 0xAA) && (value[idx++] == 0xBB))                   // Check the the memory contents are valid
        {
            DeviceID = BitConverter.ToUInt32(ReverseBytes(value, idx, 4), 0); idx += 4;
            for (int i = 0; i < 15; i++) Password[i] = (char)value[idx++];
            ManufacturerID = value[idx++];
            TODManufactured = BitConverter.ToUInt32(ReverseBytes(value, idx, 4), 0); idx += 4;
            configValid = true;
        }
        return loaded;
    }
    // -- Store: Constructs the configuration block and writes it to the Power Profiler ---------------------------------------------------------------------------------
    public bool Store()
    {
        byte[] cfgBlock = new byte[32];
        for (int i = 0; i < cfgBlock.Length; i++) cfgBlock[i] = 0;              // Initialize the configuation block
        int idx = 0; cfgBlock[idx++] = 0xAA; cfgBlock[idx++] = 0xBB;            // Set the memory verifier
        // -- Copy the parameters to the configuration block array --
        Array.Copy(BitConverter.GetBytes(DeviceID), 0, cfgBlock, idx, 4); idx += 4;
        for (int i = 0; i < 15; i++) cfgBlock[idx++] = (byte)Password[i];
        cfgBlock[idx++] = ManufacturerID;
        Array.Copy(BitConverter.GetBytes(TODManufactured), 0, cfgBlock, idx, 4); idx += 4;
        configValid = profiler.SetConfig(CFG_PROFID_ADDR, 1, cfgBlock);         // Write the configuration to the Profiler memory
        return configValid;
    }
}
// == CALIBRATION ==========================================================================================================================================
public EnergyICConfig[] EnergyICCfg = { null, null, null, null };
public ChannelCalibrationConfig ChannelCalCfg;
public MeasurementCalibrationConfig MeasCalCfg_1PH2W, MeasCalCfg_3PH3W, MeasCalCfg_3PH4W, measCalibrationCfg;
// -- Measurements data type: Used by the calibration routines to represent the current measurement values ------------------------------------------------------
public struct Measurements
{
    public double[] Voltage, Current, WattHr, VARHr, VAHr;
    public int Temperature;
    public float[] Frequency, IndustrialInputs;
}
// -- GetCurrentMeasurements: Retrieves the current measurement values from the Profiler and returns them via 'value' ------------------------------------------
public bool GetCurrentMeasurements(ref Measurements value)
{
    byte[] pkt = commsInterface.Request(CMD_GETCURRMEASUREMENTS, new byte[0], 196);          // Issue the request for the measurements
    if ((pkt.Length == 196) && (pkt[0] == 0x06))                                // Verify command was a success and that the packet is the correct length
    {
        int idx = 1;
        value.Voltage = new double[12]; for (int i = 0; i < 12; i++) { value.Voltage[i] = BitConverter.ToInt32(pkt, idx); idx += 4; }
        value.Current = new double[12]; for (int i = 0; i < 12; i++) { value.Current[i] = BitConverter.ToInt32(pkt, idx); idx += 4; }
        value.WattHr = new double[12]; for (int i = 0; i < 12; i++) { value.WattHr[i] = BitConverter.ToInt16(pkt, idx); idx += 2; }
        value.VARHr = new double[12]; for (int i = 0; i < 12; i++) { value.VARHr[i] = BitConverter.ToInt16(pkt, idx); idx += 2; }
        value.VAHr = new double[12]; for (int i = 0; i < 12; i++) { value.VAHr[i] = BitConverter.ToInt16(pkt, idx); idx += 2; }
        value.Temperature = pkt[idx++];
        value.Frequency = new float[4];
        for (int i = 0; i < 4; i++) { value.Frequency[i] = (float)(BitConverter.ToInt16(pkt, idx) * (50.0 / 800.0)); idx += 2; }
        value.IndustrialInputs = new float[8]; for (int i = 0; i < 8; i++) { value.IndustrialInputs[i] = BitConverter.ToInt16(pkt, idx); idx += 2; }
        return true;
    }
    return false;
}
// -- Enumerates the voltage input type used during calibration ---------------------------------------------------------------------------------------------------
public enum voltageInput { L1, L2, L3, Test };
// -- SelectTestInput: Sets the voltage inputs of all energy measurement ICs to the TEST input -------------------------------------------------------------------
public bool SelectTestInput()
{
    voltageInput[] voltageInputs = new voltageInput[12];
    for (int i = 0; i < 12; i++) voltageInputs[i] = voltageInput.Test;
    return SelectVoltageChannels(voltageInputs);
}
// -- SelectVoltageChannels: Configures the voltage channel multiplexers to select the correct input configuration ------------------------------------------------
public bool SelectVoltageChannels(voltageInput[] values)
{
    byte[] vreg = new byte[6];
    int idx = 5;
    for (int i = 0; i < 6; i++) vreg[i] = 0;
```

```
    for (int i = 0; i < 12; i++)
        if (((i + 1) % 2) == 0)          // Is this an even channel?
            vreg[idx--] |= (byte)Math.Pow(2.0, 7.0 - (double)values[i]);
        else
            vreg[idx] |= (byte)Math.Pow(2.0, (double)values[i]);
    return commsInterface.Command(CMD_SETVOLTAGECHANNELS, vreg, true);
}
// -- SetEnergyICMode: Sets the operational mode registers of the specified Energy Measurement IC ----------
public bool SetEnergyICMode(byte[] values) { return commsInterface.Command(CMD_SET_EM_MODES, values, true); }
// -- ChannelCalibrationConfig: Specifies the conversion values for the current channels (not currently used) and the industrial inputs ----------
public class ChannelCalibrationConfig: ConfigurationBlock
{
    public double[] CurrentInputNormalizer = new double[12]; public float[] IndustrialInputs = new Single[8];
    // -- Constructor: Assigns the parent Profiler class ----------
    public ChannelCalibrationConfig(Profiler_Rev1 parent) { profiler = parent; }
    // -- Load: Retrieves the configuration from the connected Power Profiler memory ----------
    public bool Load()
    {
        loaded = false; configValid = false;
        byte[] value = profiler.GetConfig(CFG_CHANCALIB_ADDR, 5);                    // Read the configuration block
        loaded = true;  int idx = 1;
        if ((value[idx++] == 0xAA) && (value[idx++] == 0xBB))
        {
            for (int i = 0; i < 12; i++) { CurrentInputNormalizer[i] = BitConverter.ToDouble(value, idx); idx += 8; }
            for (int i = 0; i < 8; i++) { IndustrialInputs[i] = BitConverter.ToSingle(value, idx); idx += 4; }
            configValid = true;
        }
        return loaded;
    }
    // -- Store: Writes the configuration to the connected Power Profiler memory ----------
    public bool Store()
    {
        byte[] cfgBlock = new byte[32 * 5]; Array.Clear(cfgBlock, 0, cfgBlock.Length);// Initialize the array
        cfgBlock[0] = 0xAA; cfgBlock[1] = 0xBB;                                      // Assign the memory verifier
        int idx = 2;
        for (int i = 0; i < 12; i++) { Array.Copy(BitConverter.GetBytes(CurrentInputNormalizer[i]), 0, cfgBlock, idx, 8); idx += 8; }
        for (int i = 0; i < 8; i++) { Array.Copy(BitConverter.GetBytes(IndustrialInputs[i]), 0, cfgBlock, idx, 4); idx += 4; }
        return profiler.SetConfig(CFG_CHANCALIB_ADDR, 5, cfgBlock);                  // Write the configuration block
    }
}
// -- MeasurementCalibrationConfig: Represents a configuration block for single- or three-phase configurations ----------
public class MeasurementCalibrationConfig : ConfigurationBlock
{
    ushort configAddress;
    public Single[] VRMS = new Single[3], IRMS = new Single[12], WattPower = new Single[12], VARPower = new Single[12], VAPower = new Single[12];
    public Single Frequency = 1.0F;
    public byte OPMODE, MMODE, WAVMODE, COMPMODE, LCYCMODE;
    // -- Constructor: Assigns parent Profiler class reference and memory position ----------
    public MeasurementCalibrationConfig(Profiler_Rev1 parent, ushort cfgAddr) { profiler = parent; configAddress = cfgAddr; }
    // -- Load: Retrieves the configuration block from the connected Power Profiler and decodes it ----------
    public bool Load()
    {
        loaded = false; configValid = false;
        byte[] value = profiler.GetConfig(configAddress, 7);                         // Retrieve the configuration
        loaded = true;  int idx = 1;
        if ((value[idx++] == 0xAA) && (value[idx++] == 0xBB))                        // Verify the contents are programmed
        {
            // -- Measurement conversion values --
            for (int i = 0; i < 3; i++) { VRMS[i] = BitConverter.ToSingle(value, idx); idx += 4; }
            for (int i = 0; i < 12; i++) { IRMS[i] = BitConverter.ToSingle(value, idx); idx += 4; }
            for (int i = 0; i < 12; i++) { WattPower[i] = BitConverter.ToSingle(value, idx); idx += 4; }
            for (int i = 0; i < 12; i++) { VARPower[i] = BitConverter.ToSingle(value, idx); idx += 4; }
            for (int i = 0; i < 12; i++) { VAPower[i] = BitConverter.ToSingle(value, idx); idx += 4; }
            Frequency = BitConverter.ToSingle(value, idx);
            idx += 4;
            // -- Energy measurement IC configuration register settings --
            OPMODE = value[idx++]; MMODE = value[idx++]; WAVMODE = value[idx++]; COMPMODE = value[idx++]; LCYCMODE = value[idx++];
            configValid = true;
        }
        return loaded;
    }
    // -- Store: Create the configuration block and write to the connected Power Profiler ----------
    public bool Store()
    {
        byte[] cfgBlock = new byte[32 * 7]; Array.Clear(cfgBlock, 0, cfgBlock.Length);// Initialize the array
        int idx = 0; cfgBlock[idx++] = 0xAA; cfgBlock[idx++] = 0xBB;                 // Set the memory verifier
```

```
        // -- Add the measurement conversion values --
        for (int i = 0; i < 3; i++) { Array.Copy(BitConverter.GetBytes(VRMS[i]), 0, cfgBlock, idx, 4); idx += 4; }
        for (int i = 0; i < 12; i++) { Array.Copy(BitConverter.GetBytes(IRMS[i]), 0, cfgBlock, idx, 4); idx += 4; }
        for (int i = 0; i < 12; i++) { Array.Copy(BitConverter.GetBytes(WattPower[i]), 0, cfgBlock, idx, 4); idx += 4; }
        for (int i = 0; i < 12; i++) { Array.Copy(BitConverter.GetBytes(VARPower[i]), 0, cfgBlock, idx, 4); idx += 4; }
        for (int i = 0; i < 12; i++) { Array.Copy(BitConverter.GetBytes(VAPower[i]), 0, cfgBlock, idx, 4); idx += 4; }
        Array.Copy(BitConverter.GetBytes(Frequency), 0, cfgBlock, idx, 4); idx += 4;
        // -- Add the energy measurement IC configuration register settings --
        cfgBlock[idx++] = OPMODE; cfgBlock[idx++] = MMODE; cfgBlock[idx++] = WAVMODE; cfgBlock[idx++] = COMPMODE;
        cfgBlock[idx++] = LCYCMODE;
        return profiler.SetConfig(configAddress, 7, cfgBlock);                  // Write the configuration to the connected Profiler
    }
}
// -- EnergyICConfig: Defines the calibration values for each energy measurement IC ----------
public class EnergyICConfig : ConfigurationBlock
{
    ushort configAddress;
    public byte PGAGain;
    public Int16[] VRMSGain = new Int16[3], IRMSGain = new Int16[3], WattGain = new Int16[3], VARGain = new Int16[3], VAGain = new Int16[3];
    public Int16[] VRMSOffset = new Int16[3], IRMSOffset = new Int16[3], WattOffset = new Int16[3], VAROffset = new Int16[3];
    public sbyte[] PhaseCalibration = new sbyte[3];
    public byte WDivider, VARDivider, VADivider;
    // -- Constructor: Assigns the parent Profiler class ----------
    public EnergyICConfig(Profiler_Rev1 parent, ushort configAddr) { profiler = parent; configAddress = configAddr; }
    // -- Load: Retrieves the configuration block from the connected Power Profiler and updates the internal variables ----------
    public bool Load()
    {
        loaded = false; configValid = false;
        byte[] value = profiler.GetConfig(configAddress, 2);                    // Retrieve the configuration
        int idx = 1; loaded = true;
        if ((value[idx++] == 0xAA) && (value[idx++] == 0xBB))                   // Verify the contents are valid
        {
            PGAGain = value[idx++];
            for (int i = 0; i < 3; i++) {  VRMSGain[i] = BitConverter.ToInt16(ReverseBytes(value, idx, 2), 0); idx += 2; }
            for (int i = 0; i < 3; i++) { IRMSGain[i] = BitConverter.ToInt16(ReverseBytes(value, idx, 2), 0); idx += 2; }
            for (int i = 0; i < 3; i++) { WattGain[i] = BitConverter.ToInt16(ReverseBytes(value, idx, 2), 0); idx += 2; }
            for (int i = 0; i < 3; i++) { VARGain[i] = BitConverter.ToInt16(ReverseBytes(value, idx, 2), 0); idx += 2; }
            for (int i = 0; i < 3; i++) { VAGain[i] = BitConverter.ToInt16(ReverseBytes(value, idx, 2), 0); idx += 2; }
            for (int i = 0; i < 3; i++) { VRMSOffset[i] = BitConverter.ToInt16(ReverseBytes(value, idx, 2), 0); idx += 2; }
            for (int i = 0; i < 3; i++) { IRMSOffset[i] = BitConverter.ToInt16(ReverseBytes(value, idx, 2), 0); idx += 2; }
            for (int i = 0; i < 3; i++) { WattOffset[i] = BitConverter.ToInt16(ReverseBytes(value, idx, 2), 0); idx += 2; }
            for (int i = 0; i < 3; i++) { VAROffset[i] = BitConverter.ToInt16(ReverseBytes(value, idx, 2), 0); idx += 2; }
            for (int i = 0; i < 3; i++) { PhaseCalibration[i] = (sbyte)value[idx++]; }
            WDivider = value[idx++]; VARDivider = value[idx++]; VADivider = value[idx++];
            configValid = true;
        }
        return loaded;
    }
    // -- Store: Creates the configuration block and writes it to the connected Power Profiler ----------
    public bool Store()
    {
        byte[] cfgBlock = new byte[32 * 2]; Array.Clear(cfgBlock, 0, cfgBlock.Length);          // Initialize the array
        int idx = 0; cfgBlock[idx++] = 0xAA; cfgBlock[idx++] = 0xBB;                            // Initialize the memory verifier
        cfgBlock[idx++] = PGAGain;
        for (int i = 0; i < 3; i++) { Array.Copy(ReverseBytes(BitConverter.GetBytes(VRMSGain[i]), 0, 2), 0, cfgBlock, idx, 2); idx += 2; }
        for (int i = 0; i < 3; i++) { Array.Copy(ReverseBytes(BitConverter.GetBytes(IRMSGain[i]), 0, 2), 0, cfgBlock, idx, 2); idx += 2; }
        for (int i = 0; i < 3; i++) { Array.Copy(ReverseBytes(BitConverter.GetBytes(WattGain[i]), 0, 2), 0, cfgBlock, idx, 2); idx += 2; }
        for (int i = 0; i < 3; i++) { Array.Copy(ReverseBytes(BitConverter.GetBytes(VARGain[i]), 0, 2), 0, cfgBlock, idx, 2); idx += 2; }
        for (int i = 0; i < 3; i++) { Array.Copy(ReverseBytes(BitConverter.GetBytes(VAGain[i]), 0, 2), 0, cfgBlock, idx, 2); idx += 2; }
        for (int i = 0; i < 3; i++) { Array.Copy(ReverseBytes(BitConverter.GetBytes(VRMSOffset[i]), 0, 2), 0, cfgBlock, idx, 2); idx += 2; }
        for (int i = 0; i < 3; i++) { Array.Copy(ReverseBytes(BitConverter.GetBytes(IRMSOffset[i]), 0, 2), 0, cfgBlock, idx, 2); idx += 2; }
        for (int i = 0; i < 3; i++) { Array.Copy(ReverseBytes(BitConverter.GetBytes(WattOffset[i]), 0, 2), 0, cfgBlock, idx, 2); idx += 2; }
        for (int i = 0; i < 3; i++) { Array.Copy(ReverseBytes(BitConverter.GetBytes(VAROffset[i]), 0, 2), 0, cfgBlock, idx, 2); idx += 2; }
        for (int i = 0; i < 3; i++) { cfgBlock[idx++] = (byte)PhaseCalibration[i]; }
        cfgBlock[idx++] = WDivider; cfgBlock[idx++] = VARDivider; cfgBlock[idx++] = VADivider;
        return profiler.SetConfig(configAddress, 2, cfgBlock);                                  // Write the configuration block to the connected device
    }
}
// == MEASUREMENT ========================================================================================
// -- Measurement Configuration ----------
// -- voltageMode: Enumerates the four voltage configurations allowed ----------
public enum voltageMode { V1PH2W, V3PH3W, V3PH4W, Custom };
// -- ProfileMeasurementConfig: Defines the measurement configuration block ----------
public class ProfileMeasurementConfig : ConfigurationBlock
{
    public bool[] EnergyICEnabled = new bool[4], ChannelsEnabled = new bool[12], IndustrialInputsEnabled = new bool[8];
```

```
    public voltageMode VoltageMux;
    public bool AutoCapture, SecsProfileEnabled, MinsProfileEnabled, HrsProfileEnabled, SecsSyncTOD, MinsSyncTOD, HrsSyncTOD;
    public bool SecsWrapMem, MinsWrapMem, HrsWrapMem;
    public byte SecsInterval, MinsInterval, HrsInterval;
    // -- Constructor: Assigns the parent Profiler class ------------------------------------------------------------------------------------------------------------------------------------------------------
    public ProfileMeasurementConfig(Profiler_Rev1 parent) { profiler = parent; }
    // -- Load: Retrieves the configruation from the Profiler and populates the class variables ---------------------------------------------------------------------------------------------------------
    public bool Load()
    {
      loaded = false; configValid = false;
      byte[] value = profiler.GetConfig(CFG_PROFMEAS_ADDR, 1);                    // Retrieve the values as a byte array
      loaded = true;  int idx = 1;
      if ((value[idx++] == 0xAA) && (value[idx++] == 0xBB))                       // Verify the contents are valid
      {
        int cfgBits = BitConverter.ToInt32(value, idx); idx += 4;
        int bitidx = 0;
        configValid = GetBitState(cfgBits, bitidx++);
        for (int i = 0; i < 4; i++) EnergyICEnabled[i] = GetBitState(cfgBits, bitidx++);   // Which energy ICs are enabled
        int vmux = 0;
        if (GetBitState(cfgBits, bitidx++)) vmux += 1;
        if (GetBitState(cfgBits, bitidx++)) vmux += 2;
        switch (vmux)
        {                                                                         // Determine the Voltage Multiplexer mode
          case 0: VoltageMux = voltageMode.V1PH2W; break;
          case 1: VoltageMux = voltageMode.V3PH3W; break;
          case 2: VoltageMux = voltageMode.V3PH4W; break;
        };
        for (int i = 0; i < 12; i++) ChannelsEnabled[i] = GetBitState(cfgBits, bitidx++);   // Which channels are enabled?
        for (int i = 0; i < 8; i++) IndustrialInputsEnabled[i] = GetBitState(cfgBits, bitidx++);   // Which of the industrial inputs are enabled
        AutoCapture = GetBitState(cfgBits, bitidx++);
        configValid = true;
      }
      return loaded;
    }
    // -- Store: Constructs the configuration block and writes it to the Profiler -----------------------------------------------------------------------------------------------------------------------
    public bool Store()
    {
      int idx = 0;
      byte[] cfgBlock = new byte[32]; Array.Clear(cfgBlock, 0, cfgBlock.Length);   // Initialize the array
      cfgBlock[idx++] = 0xAA; cfgBlock[idx++] = 0xBB;                             // Set the memory verifier to show the contents are valid
      UInt32 cfgBits = 0;                                                         // Store the configuration bits in a 32-bit value
      if (configValid) cfgBits += 1;
      for (int i = 0; i < 4; i++) if (EnergyICEnabled[i]) cfgBits += (UInt32)(1 << i + 1);   // Specify which energy ICs are enabled
      switch (VoltageMux)                                                         // Set the voltage multiplexer mode
      {
        case voltageMode.V1PH2W: cfgBits += (0 << 5); break;
        case voltageMode.V3PH3W: cfgBits += (1 << 5); break;
        case voltageMode.V3PH4W: cfgBits += (2 << 5); break;
        case voltageMode.Custom: cfgBits += (3 << 5); break;
      }
      for (int i = 0; i < 12; i++) if (ChannelsEnabled[i]) cfgBits += (UInt32)(1 << i + 7);   // Specify which channels are enabled
      for (int i = 0; i < 8; i++) if (IndustrialInputsEnabled[i]) cfgBits += (UInt32)(1 << i + 19);   // Specify which industrial inputs are enabled
      if (AutoCapture) cfgBits += (UInt32)(1 << 27);                              // Set Auto Capture at power-up
      Array.Copy(BitConverter.GetBytes(cfgBits), 0, cfgBlock, idx, 4);  idx += 4;   // Copy the configuration bits into the config block array
      idx += 6;                                                                   // Custom VMux configuration is not used at this stage
      byte profCfgBits = 0;
      if (SecsProfileEnabled) profCfgBits += 1;                                   // Set the parameters for Seconds profiling
      if (SecsSyncTOD) profCfgBits += 2;
      if (SecsWrapMem) profCfgBits += 4;
      cfgBlock[idx++] = profCfgBits; cfgBlock[idx++] = SecsInterval;
      profCfgBits = 0;
      if (MinsProfileEnabled) profCfgBits += 1;                                   // Set the parameters for Minutes profiling
      if (MinsSyncTOD) profCfgBits += 2;
      if (MinsWrapMem) profCfgBits += 4;
      cfgBlock[idx++] = profCfgBits; cfgBlock[idx++] = MinsInterval;
      profCfgBits = 0;
      if (HrsProfileEnabled) profCfgBits += 1;                                    // Set the parameters for Hours profiling
      if (HrsSyncTOD) profCfgBits += 2;
      if (HrsWrapMem) profCfgBits += 4;
      cfgBlock[idx++] = profCfgBits; cfgBlock[idx++] = HrsInterval;
      configValid = profiler.SetConfig(CFG_PROFMEAS_ADDR, 1, cfgBlock);           // Write the configuration to the Profiler memory
      return configValid;
    }
  }
  // -- Memory Configuration ---------------------------------------------------------------------------------------------------------------------------------------------------------------------------
  public class MemoryConfig : ConfigurationBlock
```

```
{
    public UInt32 SecsProfileAddr, MinsProfileAddr, HrsProfileAddr, EventsAddr, DiagAddr;
    // -- Constructor: Assigns the parent Profiler class ----------
    public MemoryConfig(Profiler_Rev1 parent) { profiler = parent; }
    // -- Load: Retrieve the configuration block from the connected Power Profiler and upate the parameters ----------
    public bool Load()
    {
        loaded = false; configValid = false;
        byte[] value = profiler.GetConfig(CFG_MEMORY_ADDR, 1);                  // Read the configuration from the Profiler as a byte array
        loaded = true;  int idx = 1;
        if ((value[idx++] == 0xAA) && (value[idx++] == 0xBB))                   // Check the the memory contents are valid
        {
            SecsProfileAddr = BitConverter.ToUInt32(ReverseBytes(value, idx, 4), 0); idx += 4;
            MinsProfileAddr = BitConverter.ToUInt32(ReverseBytes(value, idx, 4), 0); idx += 4;
            HrsProfileAddr = BitConverter.ToUInt32(ReverseBytes(value, idx, 4), 0); idx += 4;
            EventsAddr = BitConverter.ToUInt32(ReverseBytes(value, idx, 4), 0); idx += 4;
            DiagAddr = BitConverter.ToUInt32(ReverseBytes(value, idx, 4), 0); idx += 4;
            configValid = true;
        }
        return loaded;
    }
    // -- Store: Create the configuration block and write to the connected Power Profiler ----------
    public bool Store()
    {
        byte[] cfgBlock = new byte[32]; Array.Clear(cfgBlock, 0, cfgBlock.Length);  // Initialize the array
        int idx = 0; cfgBlock[idx++] = 0xAA; cfgBlock[idx++] = 0xBB;               // Set the memory verifier
        // -- Copy the parameters to the configuration block array --
        Array.Copy(BitConverter.GetBytes(SecsProfileAddr), 0, cfgBlock, idx, 4); idx += 4;
        Array.Copy(BitConverter.GetBytes(MinsProfileAddr), 0, cfgBlock, idx, 4); idx += 4;
        Array.Copy(BitConverter.GetBytes(HrsProfileAddr), 0, cfgBlock, idx, 4); idx += 4;
        Array.Copy(BitConverter.GetBytes(EventsAddr), 0, cfgBlock, idx, 4); idx += 4;
        Array.Copy(BitConverter.GetBytes(DiagAddr), 0, cfgBlock, idx, 4); idx += 4;
        return profiler.SetConfig(CFG_MEMORY_ADDR, 1, cfgBlock);                   // Write the configuration to the Profiler
    }
}
ChannelCalibrationConfig channelCalibration;
// -- FLASHMemoryPointers structure: Holds the current memory pointers retrieved from the connected Power Profiler ----------
public struct FLASHMemoryPointers
{
    public UInt32 UpdateCount, LastUpdateTOD;
    public UInt32 Secs_StartAddr, Secs_StopAddr, Secs_RdPtr, Secs_WrPtr, Secs_MemFree;
    public UInt32 Mins_StartAddr, Mins_StopAddr, Mins_RdPtr, Mins_WrPtr, Mins_MemFree;
    public UInt32 Hrs_StartAddr, Hrs_StopAddr, Hrs_RdPtr, Hrs_WrPtr, Hrs_MemFree;
}
public FLASHMemoryPointers FlashMemPtrs;
// -- Commands to Reset, Start and Stop acqusition. The Profiler will ACK if successful. ----------
public bool ResetAcquisition() { return commsInterface.Command(CMD_RESETACQUISITION, new byte[0], true); }
public bool StartAcquisition() { return commsInterface.Command(CMD_STARTACQUISITION, new byte[0], true); }
public bool StopAcquisition() { return commsInterface.Command(CMD_STOPACQUISITION, new byte[0], true); }
// -- GetFlashPointers: Retrieves the profile memory pointers by first updating the current status and then returning the values. ----------
public FLASHMemoryPointers GetFlashPointers() { GetStatus(); return FlashMemPtrs; }
// -- Measurement Retrieval and Processing ----------
// -- StatValue: Represents a statistical measurement value of maximum, minimum and average ----------
public struct StatValue { public double Max, Min, Avg; }
// -- Profiler_Rev1_MeasurementValue: Represents a measurement value implemented as a class to support further processing if necessary ----------
public class Profiler_Rev1_MeasurementValue
{
    public voltageMode VoltageMode;
    public bool[] ChannelsEnabled = new bool[3], IndustrialIPEnabled = new bool[8];
    public byte Flags;
    public DateTime Timestamp;
    public StatValue[] VRMS = new StatValue[3], IRMS = new StatValue[12];
    public StatValue[] Power_Watts = new StatValue[12], Power_VAR = new StatValue[12], Power_VA = new StatValue[12];
    public StatValue Frequency, Temperature;
    public StatValue[] IndustrialInputs = new StatValue[8];
}
// -- UnpackTOD: Decodes a 32-bit TOD value representation into a DateTime type ----------
public DateTime UnpackTOD(UInt32 value)
{
    DateTime result = new DateTime();
    result = result.AddSeconds((double)(value & 0x3F));
    result = result.AddMinutes((double)((value >> 6) & 0x3F));
    result = result.AddHours((double)((value >> 12) & 0x1F));
    result = result.AddDays((double)((value >> 17) & 0x1F) - 1);
    result = result.AddMonths((int)((value >> 22) & 0x0F) - 1);
    result = result.AddYears((int)((value >> 26) & 0x3F) - 1 + 2000);
```

```
    return result;
}
// -- RMSToUInt32: Reads a 24-bit RMS value from the measurement buffer and converts it to a 32-bit representation ----------
public UInt32 RMSToUInt32(byte[] value, int idx) { return (UInt32)(value[idx++] + (value[idx++] << 8) + (value[idx++] << 16)); }
// -- GetNextMeasurement: Requests the next measurement value for hte specified profile from the Power Profiler memory ----------
public bool GetNextMeasurement(int profile, ref Profiler_Rev1_MeasurementValue result)
{
    byte[] param = new byte[1];
    param[0] = (byte)profile;
    byte[] pkt = commsInterface.Request(CMD_GETNEXTMEASUREMENT, param, 100);
    if (pkt[0] == 0x06)
    {
        // Specify the channnels in use in the result based on the Measurement configuration
        result.VoltageMode = MeasConfig.VoltageMux;
        result.ChannelsEnabled = MeasConfig.ChannelsEnabled;
        result.IndustrialIPEnabled = MeasConfig.IndustrialInputsEnabled;
        // Populate the values from the retrieved measurement
        int idx = 1;
        result.Flags = pkt[idx++];
        result.Timestamp = UnpackTOD(BitConverter.ToUInt32(pkt, idx)); idx += 4;
        int voltageCount = 0;
        switch (result.VoltageMode)
        {
            case voltageMode.V1PH2W: voltageCount = 1; break;
            case voltageMode.V3PH3W: voltageCount = 2; break;
            case voltageMode.V3PH4W: voltageCount = 3; break;
        };
        for (int i = 0; i < voltageCount; i++)
        {
            result.VRMS[i].Max = RMSToUInt32(pkt, idx) * measCalibrationCfg.VRMS[i]; idx += 3;
            result.VRMS[i].Avg = RMSToUInt32(pkt, idx) * measCalibrationCfg.VRMS[i]; idx += 3;
            result.VRMS[i].Min = RMSToUInt32(pkt, idx) * measCalibrationCfg.VRMS[i]; idx += 3;
        }
        for (int i = 0; i < 12; i++)
            if (result.ChannelsEnabled[i])
            {
                result.IRMS[i].Max = RMSToUInt32(pkt, idx) * measCalibrationCfg.IRMS[i]; idx += 3;
                result.IRMS[i].Avg = RMSToUInt32(pkt, idx) * measCalibrationCfg.IRMS[i]; idx += 3;
                result.IRMS[i].Min = RMSToUInt32(pkt, idx) * measCalibrationCfg.IRMS[i]; idx += 3;
                result.Power_Watts[i].Max = BitConverter.ToInt16(pkt, idx) * measCalibrationCfg.WattPower[i]; idx += 2;
                result.Power_Watts[i].Avg = BitConverter.ToInt16(pkt, idx) * measCalibrationCfg.WattPower[i]; idx += 2;
                result.Power_Watts[i].Min = BitConverter.ToInt16(pkt, idx) * measCalibrationCfg.WattPower[i]; idx += 2;
                result.Power_VAR[i].Max = BitConverter.ToInt16(pkt, idx) * measCalibrationCfg.VARPower[i]; idx += 2;
                result.Power_VAR[i].Avg = BitConverter.ToInt16(pkt, idx) * measCalibrationCfg.VARPower[i]; idx += 2;
                result.Power_VAR[i].Min = BitConverter.ToInt16(pkt, idx) * measCalibrationCfg.VARPower[i]; idx += 2;
                result.Power_VA[i].Max = BitConverter.ToInt16(pkt, idx) * measCalibrationCfg.VAPower[i]; idx += 2;
                result.Power_VA[i].Avg = BitConverter.ToInt16(pkt, idx) * measCalibrationCfg.VAPower[i]; idx += 2;
                result.Power_VA[i].Min = BitConverter.ToInt16(pkt, idx) * measCalibrationCfg.VAPower[i]; idx += 2;
            }
        // -- Frequency --
        result.Frequency.Max = BitConverter.ToUInt16(pkt, idx) * measCalibrationCfg.Frequency; idx += 2;
        result.Frequency.Avg = BitConverter.ToUInt16(pkt, idx) * measCalibrationCfg.Frequency; idx += 2;
        result.Frequency.Min = BitConverter.ToUInt16(pkt, idx) * measCalibrationCfg.Frequency; idx += 2;
        // -- Temperature --
        result.Temperature.Max = pkt[idx++] * 1.0; result.Temperature.Avg = pkt[idx++] * 1.0; result.Temperature.Min = pkt[idx++] * 1.0;
        // -- Industrial Inputs --
        for (int i = 0; i < 8; i++)
            if (result.IndustrialIPEnabled[i])
            {
                result.IndustrialInputs[i].Max = BitConverter.ToInt16(pkt, idx) * channelCalibration.IndustrialInputs[i]; idx += 2;
                result.IndustrialInputs[i].Avg = BitConverter.ToInt16(pkt, idx) * channelCalibration.IndustrialInputs[i]; idx += 2;
                result.IndustrialInputs[i].Min = BitConverter.ToInt16(pkt, idx) * channelCalibration.IndustrialInputs[i]; idx += 2;
            }
        return true;
    }
    return false;
}
// == CONTROL / MANAGEMENT FUNCTIONALITY ===========================================================================
public bool TODValid;  public DateTime TOD;
// -- GetStatus: Retrieves the current RTC and profile memory pointers and updates the internal variables. ----------
public void GetStatus()
{
    byte[] param = new byte[0];
    byte[] result = commsInterface.Request(CMD_GETSTATUS, new byte[0], 77);
    int idx = 1;
    TOD = new DateTime(result[idx++] + 2000, result[idx++], result[idx++], result[idx++ + 1], result[idx++ + 1], result[idx++ + 1]);
```

```
            idx++;
            idx += 2; // skip verifiers
            FlashMemPtrs.UpdateCount = BitConverter.ToUInt32(result, idx); idx += 4;
            FlashMemPtrs.LastUpdateTOD = BitConverter.ToUInt32(result, idx); idx += 4;
            FlashMemPtrs.Secs_StartAddr = BitConverter.ToUInt32(result, idx); idx += 4;
            FlashMemPtrs.Secs_StopAddr = BitConverter.ToUInt32(result, idx); idx += 4;
            FlashMemPtrs.Secs_RdPtr = BitConverter.ToUInt32(result, idx); idx += 4;
            FlashMemPtrs.Secs_WrPtr = BitConverter.ToUInt32(result, idx); idx += 4;
            FlashMemPtrs.Secs_MemFree = BitConverter.ToUInt32(result, idx); idx += 4;
            FlashMemPtrs.Mins_StartAddr = BitConverter.ToUInt32(result, idx); idx += 4;
            FlashMemPtrs.Mins_StopAddr = BitConverter.ToUInt32(result, idx); idx += 4;
            FlashMemPtrs.Mins_RdPtr = BitConverter.ToUInt32(result, idx); idx += 4;
            FlashMemPtrs.Mins_WrPtr = BitConverter.ToUInt32(result, idx); idx += 4;
            FlashMemPtrs.Mins_MemFree = BitConverter.ToUInt32(result, idx); idx += 4;
            FlashMemPtrs.Hrs_StartAddr = BitConverter.ToUInt32(result, idx); idx += 4;
            FlashMemPtrs.Hrs_StopAddr = BitConverter.ToUInt32(result, idx); idx += 4;
            FlashMemPtrs.Hrs_RdPtr = BitConverter.ToUInt32(result, idx); idx += 4;
            FlashMemPtrs.Hrs_WrPtr = BitConverter.ToUInt32(result, idx); idx += 4;
            FlashMemPtrs.Hrs_MemFree = BitConverter.ToUInt32(result, idx); idx += 4;
        }
        // -- Initialize: Retrieves the channel and measurement configurations and updates interval variables. ----------------------------------------------------------
        public void Initialize()
        {
            channelCalibration = new ChannelCalibrationConfig(this);
            channelCalibration.Load();                                      // Retrieve the calibration values from the Profiler
            if (!MeasConfig.Loaded) MeasConfig.Load();                      // Retrieve the measurement configuration from the Profiler
            switch (MeasConfig.VoltageMux)                                  // Retrieve the relevant channel calibration data based on voltage mode
            {
                case voltageMode.V1PH2W: measCalibrationCfg = MeasCalCfg_1PH2W; break;
                case voltageMode.V3PH3W: measCalibrationCfg = MeasCalCfg_3PH3W; break;
                case voltageMode.V3PH4W: measCalibrationCfg = MeasCalCfg_3PH4W; break;
            };
            measCalibrationCfg.Load();                                      // Retrieve channel calibration configration from the Profiler
        }
        // -- Reset: Instructs the Power Profiler to perform a system reset ---------------------------------------------------------------------------------------------
        public bool Reset()
        {
            byte[] param = new byte[1];
            param[0] = 0x02;
            return commsInterface.Command(CMD_RESET, param, true);
        }
        // -- SetClock: Sets the Profiler's RTC to the value specified. -------------------------------------------------------------------------------------------------
        public bool SetClock(DateTime value)
        {
            byte[] param = new byte[7];
            int idx = 0;
            param[idx++] = (byte)(value.Year - 2000); param[idx++] = (byte)value.Month; param[idx++] = (byte)value.Day; param[idx++] = (byte)value.DayOfWeek;
            param[idx++] = (byte)value.Hour; param[idx++] = (byte)value.Minute; param[idx++] = (byte)value.Second;
            return commsInterface.Command(CMD_SETCLOCK, param, true);
        }
        // == SUPPORT ROUTINES ===========================================================================================================================
        // -- GetBitState: Returns whether the specified bit value is true or false -------------------------------------------------------------------------------------
        public static bool GetBitState(int value, int bit) { return ((value & (int)Math.Pow(2, bit)) == Math.Pow(2, bit)); }
        // -- ReverseBytes: Reverses the byte order of the supplied byte array ------------------------------------------------------------------------------------------
        private static byte[] ReverseBytes(byte[] buffer, int idx, int length)
        {
            byte[] temp = new byte[length];
            for (int i = 0; i < length; i++) temp[i] = buffer[idx + length - i - 1];
            return temp;
        }
    }
}
```

```
/* ==  MDIPARENT.CS  ================================================================================================
The MDI application used to Control and Configure the Power Profiler.
================================================================================================================*/
using System; using System.Windows.Forms; using Profiler.Comms.Mediums; using Profiler.Comms.Procotols; using Profiler.Comms.Interfaces;
using Profiler.Devices; using System.IO.Ports;
namespace ProfilerCalibrator
{
   public partial class MDIParent : Form
   {
      Profiler_Rev1 profiler = new Profiler_Rev1();
      // -- Constructor ----------------------------------------------------------------------------------------------------------------------------------------------
      public MDIParent() { InitializeComponent(); }
      // == MDI FORM CONTROL ========================================================================================
      // -- MDIParent_Load: Called after form creation and initializes the serial port list in the menus ------------------------------------------------------
      private void MDIParent_Load(object sender, EventArgs e)
      {
         string[] ports = SerialPort.GetPortNames();                              // Get the current list of serial ports available
         foreach (string port in ports)
         {
            bluetoothToolStripMenuItem.DropDownItems.Add(port, null, comBTSelectClick);
            uSBToolStripMenuItem.DropDownItems.Add(port, null, comUSBSelectClick);
         }
      }
      // -- Exit: Exits the application ------------------------------------------------------------------------------------------------------------------------
      private void ExitToolsStripMenuItem_Click(object sender, EventArgs e) { this.Close(); }
      // -- MDI child window control: Cascade, Tile, Arrange and Close All functions ---------------------------------------------------------------------------
      private void CascadeToolStripMenuItem_Click(object sender, EventArgs e)  { LayoutMdi(MdiLayout.Cascade); }
      private void TileVerticalToolStripMenuItem_Click(object sender, EventArgs e) { LayoutMdi(MdiLayout.TileVertical); }
      private void TileHorizontalToolStripMenuItem_Click(object sender, EventArgs e) { LayoutMdi(MdiLayout.TileHorizontal); }
      private void ArrangeIconsToolStripMenuItem_Click(object sender, EventArgs e) { LayoutMdi(MdiLayout.ArrangeIcons); }
      private void CloseAllToolStripMenuItem_Click(object sender, EventArgs e) { foreach (Form childForm in MdiChildren) { childForm.Close(); } }
      // == COMMUNICATIONS ==========================================================================================
      // -- comBTSelectClick: Menu item to connect via Bluetooth SPP -------------------------------------------------------------------------------------------
      private void comBTSelectClick(object sender, EventArgs e)
      {
         try
         {
            profiler.Connect_Bluetooth(((ToolStripMenuItem)sender).ToString());
            Status.Text = "Connected via Bluetooth";
            tsPing.Enabled = true; tsLogin.Enabled = true;
         }
         catch  {  Status.Text = "Error during Bluteooth connection";  }
      }
      // -- tsGSMConnect_Click: Menut item to connect via GSM --------------------------------------------------------------------------------------------------
      private void tsGSMConnect_Click(object sender, EventArgs e)
      {
         try
         {
            profiler.Connect_GSM(tsTBGSMIP.Text, 20001);
            Status.Text = "Connected via GSM TCP/IP to " + tsTBGSMIP.Text;
            tsPing.Enabled = true; tsLogin.Enabled = true;
         }
         catch {  Status.Text = "Error during GSM TCP/IP connection to " + tsTBGSMIP.Text;  }
      }
      // -- comUSBSelectClick: Menu item to connect via USB SPP ------------------------------------------------------------------------------------------------
      private void comUSBSelectClick(object sender, EventArgs e)
      {
         try
         {
            profiler.Connect_USB(((ToolStripMenuItem)sender).ToString());
            Status.Text = "Connected via USB Virtual COM Port";
            tsPing.Enabled = true; tsLogin.Enabled = true;
         }
         catch {  Status.Text = "Error during USB connection";  }
      }
      // -- tsPing_Click: Ping the connected Profiler to verify communications ---------------------------------------------------------------------------------
      private void tsPing_Click(object sender, EventArgs e)
      {
         DateTime Start = DateTime.Now;
         Status.Text = "Sending Ping";
         if (profiler.Ping())
            Status.Text = "Ping successful: " + (DateTime.Now - Start).TotalMilliseconds.ToString() + " ms";
         else
            Status.Text = "Ping failed";
      }
      // -- tsLogin_Click: Show the login dialog to allow the user to enter the login ID and password ----------
```

```
    private void tsLogin_Click(object sender, EventArgs e)
    {
        frmLogin login = new frmLogin();
        if (login.ShowDialog(this) == System.Windows.Forms.DialogResult.OK)
        {
            if (profiler.Login(Convert.ToUInt32(login.deviceID), login.password))
            {
                Status.Text = "Login to " + login.deviceID + " successful";
                profiler.Initialize();
            }
            else
                Status.Text = "Login to " + login.deviceID + " rejected";
        }
        else
            Status.Text = "Login cancelled by user";
    }
    // == CONFIGURATION AND CONTROL ===========================================================================
    // -- miSynchronizeRTC: Synchronizes the Power Profiler clock with the PC's real-time-clock ----------
    private void miSynchronizeRTC(object sender, EventArgs e) { profiler.SetClock(DateTime.UtcNow); }
    // -- Menu items to reset, start and stop Power Profiler acquisition ----------
    private void miResetAcq_Click(object sender, EventArgs e) { profiler.ResetAcquisition(); }
    private void miStartAcq_Click(object sender, EventArgs e) { profiler.StartAcquisition(); }
    private void miStopAcq_Click(object sender, EventArgs e) { profiler.StopAcquisition(); }
    // -- Menu items to create the calibration, measurement and communications settings forms and to set the Profiler ID and password ----------
    private void miCalibration_Click(object sender, EventArgs e)
        { frmCalibrate frm = new frmCalibrate(); frm.status = Status; frm.profiler = profiler; frm.Show(); }
    private void miMeasSettings_Click(object sender, EventArgs e)
        { frmConfigure frm = new frmConfigure(); frm.profiler = profiler; frm.MdiParent = this; frm.Show(); }
    private void miCommsSettings_Click(object sender, EventArgs e)
        { frmConfigureComms frm = new frmConfigureComms(); frm.status = Status; frm.profiler = profiler; frm.Show(); }
    private void miSetProfilerID_Click(object sender, EventArgs e)
        { frmChangeID frm = new frmChangeID(); frm.status = Status; frm.profiler = profiler; frm.Show(); }

    // == MEASUREMENT ACCESS ==================================================================================
    // -- OpenFile: Menu item to open a measurement CSV file ----------
    private void OpenFile(object sender, EventArgs e)
    {
        OpenFileDialog openFileDialog = new OpenFileDialog();
        openFileDialog.Filter = "Text Files (*.txt)|*.txt|All Files (*.*)|*.*";
        if (openFileDialog.ShowDialog(this) == DialogResult.OK)
        {
            string FileName = openFileDialog.FileName;
            frmViewMeas frm = new frmViewMeas();                    // Create the view measurement form
            frm.LoadDataSet(FileName);                              // Load the measurement data
            frm.MdiParent = this;
            frm.Show();                                             // Show the MDI child window
        }
    }
    // -- Upload: Creates the measurement upload form ----------
    private void uploadToolStripMenuItem_Click(object sender, EventArgs e)
    { frmUpload frm = new frmUpload(); frm.profiler = profiler; frm.status = Status; frm.Show(); }
  }
}
```

```
/* ==  FRMCONFIGURE.CS  ======================================================================================
Configures the Power Profiler measurement parameters and writes them to configuration memory.
=========================================================================================================*/
using System; using System.Collections.Generic; using System.ComponentModel; using System.Data; using System.Drawing; using System.Linq;
using System.Text; using System.Windows.Forms; using Profiler.Devices; using System.Threading;
namespace ProfilerCalibrator
{
  public partial class frmConfigure : Form
  {
    public Profiler_Rev1 profiler;
    public ToolStripStatusLabel status;
    CheckBox[] channelEnabled = new CheckBox[12];  CheckBox[] industrialIPEnabled = new CheckBox[12];
    int eventMemoryRequired, measMemoryRequired, flashAddrSecs, flashAddrMins, flashAddrHrs, flashAddrEvents, flashAddrDiag;
    // -- Constructor: points the indexed checkbox arrays to the individual controls on the form to simplify access ----------
    public frmConfigure()
    {
        InitializeComponent();
        int idx = 0;
        channelEnabled[idx++] = cbCh1; channelEnabled[idx++] = cbCh2; channelEnabled[idx++] = cbCh3; channelEnabled[idx++] = cbCh4;
        channelEnabled[idx++] = cbCh5; channelEnabled[idx++] = cbCh6; channelEnabled[idx++] = cbCh7; channelEnabled[idx++] = cbCh8;
        channelEnabled[idx++] = cbCh9; channelEnabled[idx++] = cbCh10; channelEnabled[idx++] = cbCh11; channelEnabled[idx++] = cbCh12;
```

```
    idx = 0;
    industrialIPEnabled[idx++] = cbIP1; industrialIPEnabled[idx++] = cbIP2; industrialIPEnabled[idx++] = cbIP3; industrialIPEnabled[idx++] = cbIP4;
    industrialIPEnabled[idx++] = cbIP5; industrialIPEnabled[idx++] = cbIP6; industrialIPEnabled[idx++] = cbIP7; industrialIPEnabled[idx++] = cbIP8;
}
// -- frmConfigure_Load: Called upon form creation and resets the voltage type select combobox ------------------------------------------------------------
private void frmConfigure_Load(object sender, EventArgs e)
{ cobVoltageType.SelectedIndex = 0; }
// -- SecsToMeasTime: Converts and displays profile periods in a more readable manner ------------------------------------------------------------------------
private string SecsToMeasTime(int secsValue)
{
    double mins, hrs, days;
    string s = "Time period: ";
    mins = secsValue / 60; hrs = mins / 60; days = hrs / 60;
    if (days >= 1)
        s = s + days.ToString("F1") + " days";
    else
        if (hrs >= 1)
            s = s + hrs.ToString("F1") + " hrs";
        else
            if (mins >= 1)
                s = s + mins.ToString("F1") + " mins";
            else
                s = s + secsValue.ToString("F0") + " secs";
    return s;
}
// -- UpdateMemoryUsage: Aligns the profile and event memories to block boundaries and allocates the memory to each profile ----------------------------------
private void UpdateMemoryUsage()
{
    eventMemoryRequired = 0;                                    // Events are not currently used and therefore not allocated memory
    measMemoryRequired = ProfileMemoryUsage();
    // Memories must be aligned to 1056 byte pages to reduce the number of page writes required
    int totalBlocks = (int)(profiler.FLASHMemoryAvailable / 1056);
    int memoryAvailable = profiler.FLASHMemoryAvailable;
    int eventCount = (int)Math.Truncate((nudEventMemory.Value / 100) * profiler.FLASHMemoryAvailable / eventMemoryRequired);
    int eventBlocks = (int)Math.Ceiling(((double)(eventCount * eventMemoryRequired) / 1056));
    memoryAvailable -= eventBlocks * 1056;
    pbEvents.Value = ((int)(100 * ((double)(eventBlocks * 1056) / profiler.FLASHMemoryAvailable)));
    lblEventCount.Text = "Events: " + eventCount.ToString();
    // -- Seconds Measurement Profiling --
    flashAddrSecs = profiler.FLASHMarkerSize;
    int secsCount = 0, secsBlocks = 0;
    if (cbSecsEnabled.Checked)
    {
        secsCount = (int)Math.Truncate((nudSecsMemory.Value / 100) * profiler.FLASHMemoryAvailable / measMemoryRequired);
        secsBlocks = (int)Math.Ceiling(((double)(secsCount * measMemoryRequired) / 1056));
    }
    memoryAvailable -= secsBlocks * 1056;
    lSecsMeasTime.Text = SecsToMeasTime((int)(secsCount * nudSecsInterval.Value));
    pbSecs.Value = ((int)(100 * ((double)(secsBlocks * 1056) / profiler.FLASHMemoryAvailable)));
    // -- Minutes Measurement Profiling --
    flashAddrMins = flashAddrSecs + secsBlocks * 1056;
    int minsCount = 0, minsBlocks = 0;
    if (cbMinsEnabled.Checked)
    {
        minsCount = (int)Math.Truncate((nudMinsMemory.Value / 100) * profiler.FLASHMemoryAvailable / measMemoryRequired);
        minsBlocks = (int)Math.Ceiling(((double)(minsCount * measMemoryRequired) / 1056));
    }
    memoryAvailable -= minsBlocks * 1056;
    lMinsMeasTime.Text = SecsToMeasTime((int)(minsCount * nudMinsInterval.Value * 60));
    pbMins.Value = ((int)(100 * ((double)(minsBlocks * 1056) / profiler.FLASHMemoryAvailable)));
    // -- Hours Measurement Profiling - Uses the remaining memory available, constrained by block size --
    flashAddrHrs = flashAddrMins + minsBlocks * 1056;
    int hrsBlocks = 0, hrsCount = 0;
    if (cbHrsEnabled.Checked)
    {
        hrsBlocks = (int)Math.Truncate((double)(memoryAvailable / 1056));
        hrsCount = (int)Math.Truncate((double)(hrsBlocks * 1056 / measMemoryRequired));
    }
    memoryAvailable -= hrsBlocks * 1056;
    lHrsMeasTime.Text = SecsToMeasTime((int)(hrsCount * nudHrsInterval.Value * 60 * 60));
    if (hrsCount > 0) pbHrs.Value = ((int)(100 * ((double)(hrsBlocks * 1056) / profiler.FLASHMemoryAvailable)));
    lHrsMemPerc.Text = pbHrs.Value.ToString();
    // -- Events memory allocation --
    flashAddrEvents = flashAddrHrs + hrsBlocks * 1056;
    flashAddrDiag = flashAddrEvents + eventBlocks * 1056;
    // -- Check memory is balanced correctly --
```

```
    if (memoryAvailable < 0)
    {
        lMemAllocated.ForeColor = Color.Red;
        lMemAllocated.Text = "Memory usage unbalanced (" + memoryAvailable.ToString() + ")";
    }
    else
    {
        lMemAllocated.Text = "Memory usage balanced (" + memoryAvailable.ToString() + ")";
        lMemAllocated.ForeColor = Color.Green;
    }
}
// -- btnStore_Click: Write the configuration to the profiler class and update the Configuration memory ----------
private void btnStore_Click(object sender, EventArgs e)
{
    switch (cobVoltageType.SelectedIndex)
    {
        case 0: profiler.MeasConfig.VoltageMux = Profiler_Rev1.voltageMode.V1PH2W; break;
        case 1: profiler.MeasConfig.VoltageMux = Profiler_Rev1.voltageMode.V3PH3W; break;
        case 2: profiler.MeasConfig.VoltageMux = Profiler_Rev1.voltageMode.V3PH4W; break;
    };
    for (int i = 0; i < 12; i++)
        profiler.MeasConfig.ChannelsEnabled[i] = channelEnabled[i].Checked;        // Update the profiler class's channel enabled
    int idx = 0;
    for (int i = 0; i < 4; i++)                                                    // Any measurement ICs that are not required are disabled to save power
    {
        profiler.MeasConfig.EnergyICEnabled[i] = ((channelEnabled[idx].Checked || channelEnabled[idx + 1].Checked || channelEnabled[idx + 2].Checked));
        idx += 3;
    }
    profiler.MeasConfig.AutoCapture = cbAutoCapture.Checked;                       // Enable automatic capture at startup
    // -- Update profile configuration in the profiler class --
    profiler.MeasConfig.SecsProfileEnabled = cbSecsEnabled.Checked;
    profiler.MeasConfig.SecsWrapMem = cbSecsWrap.Checked;
    profiler.MeasConfig.SecsSyncTOD = cbSecTODSync.Checked;
    profiler.MeasConfig.SecsInterval = (byte)nudSecsInterval.Value;
    profiler.MeasConfig.MinsProfileEnabled = cbMinsEnabled.Checked;
    profiler.MeasConfig.MinsWrapMem = cbMinsWrap.Checked;
    profiler.MeasConfig.MinsSyncTOD = cbMinsTODSync.Checked;
    profiler.MeasConfig.MinsInterval = (byte)nudMinsInterval.Value;
    profiler.MeasConfig.HrsProfileEnabled = cbHrsEnabled.Checked;
    profiler.MeasConfig.HrsWrapMem = cbHrsWrap.Checked;
    profiler.MeasConfig.HrsSyncTOD = cbHrsTODSync.Checked;
    profiler.MeasConfig.HrsInterval = (byte)nudHrsInterval.Value;
    profiler.MeasConfig.Store();                                                   // Write the configuration to the Profiler
    // -- Update memory addresses in the profiler class --
    profiler.MemoryCfg.SecsProfileAddr = (uint)flashAddrSecs;
    profiler.MemoryCfg.MinsProfileAddr = (uint)flashAddrMins;
    profiler.MemoryCfg.HrsProfileAddr = (uint)flashAddrHrs;
    profiler.MemoryCfg.EventsAddr = (uint)flashAddrEvents;
    profiler.MemoryCfg.DiagAddr = (uint)flashAddrDiag;
    profiler.MemoryCfg.Store();                                                    // Write the configuration to the Profiler
}
// -- cobVoltageType_SelectedIndexChanged: Updates the memory usage and instruction text based on the voltage input type selected ----------
private void cobVoltageType_SelectedIndexChanged(object sender, EventArgs e)
{
    switch (cobVoltageType.SelectedIndex)
    {
        case 0: lblVInInstruction.Text = "Use L = L1 and N = N"; break;
        case 1: lblVInInstruction.Text = "Use L1 = L1, L2 = L2 and L3 = N"; break;
        case 2: lblVInInstruction.Text = "Use L1 = L1, L2 = L2, L3 = L3 and N = N"; break;
    }
    UpdateMemoryUsage();
}
// -- ProfileMemoryUsage: Returns the memory size required for the user-defined profiles ----------
private int ProfileMemoryUsage()
{
    int usage = 0;
    usage++;                                                                       // Flags for each measurement
    usage += 4;                                                                    // Time of Day
    switch (cobVoltageType.SelectedIndex)                                          // Voltage channel requirements
    {
        case 0:                                                                    // 0 : Single Phase 2-Wire
            usage += 1 * 3 * 3;                                                    // 1 channel of 24-bit max, min, avg
            for (int i = 0; i < 12; i++)
                if (channelEnabled[i].Checked)
                    usage += (1 * 3 * 3) + (3 * 2 * 3);                            // 1ch 24-bit current (max, min, avg) + 3ch 16-bit power (max, min avg)
            break;
```

```
            case 1: usage += 2 * 3 * 3; break;                                  // 1 : Three Phase 3-Wire: 2ch 24-bit max, min, avg
            case 2: usage += 3 * 3 * 3;                                         // 2: Three Phase 4-Wire: 3ch 24-bit max, min, avg
               for (int i = 0; i < 12; i++)
                  if (channelEnabled[i].Checked) usage += (1 * 3 * 3) + (3 * 2 * 3); break;  // 1ch 24-bit current (max, min, avg) + 3ch 16-bit power (max, min avg)
         };
         usage += 1 * 2 * 3;                                                    // Frequency: 1ch 12-bit max, min, avg
         usage += 1 * 1 * 3;                                                    // Temperature: 1ch 8-bit max, min, avg
         for (int i = 0; i < 8; i++)                                            // Industrial Inputs
            if (industrialIPEnabled[i].Checked) usage += 1 * 2 * 3;             // 1ch 10-bit max, min, avg
         return usage;
      }
      // -- Check and value change events which trigger an update of memory usage --
      private void cbProfileEnabled_CheckedChanged(object sender, EventArgs e) { UpdateMemoryUsage(); }
      private void nudSecsInterval_ValueChanged(object sender, EventArgs e) { UpdateMemoryUsage(); }
      private void nudSecsMemory_ValueChanged_1(object sender, EventArgs e) { UpdateMemoryUsage(); }
      private void cbChannel_CheckStateChanged(object sender, EventArgs e) { UpdateMemoryUsage(); }
   }
}
```

```
/* ==  FRMLOGIN.CS  ========================================================================================================
Implements a user dialog form to specify the Device ID and Password. These fields are saved to two public variables which are accessed by the MDI application
upon a 'OK' dialog result.
========================================================================================================================*/
using System; using System.Windows.Forms;
namespace ProfilerCalibrator
{
   public partial class frmLogin : Form
   {
      public string deviceID, password;
      public frmLogin() { InitializeComponent(); }
      private void btnCancel_Click(object sender, EventArgs e) {  DialogResult = DialogResult.Cancel;  }
      private void btnLogin_Click(object sender, EventArgs e)
      {
         deviceID = tbID.Text;
         password = tbPwd.Text;
         DialogResult = DialogResult.OK;
      }
   }
}
```

```
/* ==  FRMUPLOAD.CS  =======================================================================================================
Uploads measurement data from the Power Profiler and stores it in CSV format in a text file.
========================================================================================================================*/
using System; using System.Windows.Forms; using Profiler.Devices;
namespace ProfilerCalibrator
{
   public partial class frmUpload : Form
   {
      public Profiler_Rev1 profiler;  public ToolStripStatusLabel status;
      int secsMeasurements, minsMeasurements, hrsMeasurements;
      // -- frmUpload: Form constructor ---------------------------------------------------------------------------------------------------------------------------------------------
      public frmUpload() {  InitializeComponent();  }
      // -- btnGetPtrs_Click: Retrieves the current secs, mins and hrs memory pointers and calculates the number of records available ------------------------------------
      private void btnGetPtrs_Click(object sender, EventArgs e)
      {
         Profiler_Rev1.FLASHMemoryPointers flashPtrs = profiler.GetFlashPointers();
         if (flashPtrs.Secs_WrPtr >= flashPtrs.Secs_RdPtr) secsMeasurements = (int)Math.Truncate((flashPtrs.Secs_WrPtr - flashPtrs.Secs_RdPtr) / 365.0);
         if (flashPtrs.Mins_WrPtr >= flashPtrs.Mins_RdPtr) minsMeasurements = (int)Math.Truncate((flashPtrs.Mins_WrPtr - flashPtrs.Mins_RdPtr) / 365.0);
         if (flashPtrs.Hrs_WrPtr >= flashPtrs.Hrs_RdPtr) hrsMeasurements = (int)Math.Truncate((flashPtrs.Hrs_WrPtr - flashPtrs.Hrs_RdPtr) / 365.0);
         lSecs.Text = "Seconds: " + secsMeasurements.ToString();
         lMins.Text = "Minutes: " + minsMeasurements.ToString();
         lHrs.Text = "Hours: " + hrsMeasurements.ToString();
      }
      // -- CreateMeasLine: Converts a measurement value into a comma-seperated string and returns the string result ----------------------------------------------------
      private string CreateMeasLine(Profiler_Rev1.Profiler_Rev1_MeasurementValue meas)
      {
         string s = meas.Timestamp.ToString() + ",";
         for (int i = 0; i < 3; i++) s = s + String.Format("{0:F1},{1:F1},{2:F1},", meas.VRMS[i].Max, meas.VRMS[i].Avg, meas.VRMS[i].Min);
         for (int i = 0; i < 12; i++)
            if (meas.ChannelsEnabled[i])
            {
               s = s + String.Format("{0:F1},{1:F1},{2:F1},", meas.IRMS[i].Max, meas.IRMS[i].Avg, meas.IRMS[i].Min);
               s = s + String.Format("{0:F1},{1:F1},{2:F1},", meas.Power_Watts[i].Max, meas.Power_Watts[i].Avg, meas.Power_Watts[i].Min);
```

```
            s = s + String.Format("{0:F1},{1:F1},{2:F1},", meas.Power_VAR[i].Max, meas.Power_VAR[i].Avg, meas.Power_VAR[i].Min);
            s = s + String.Format("{0:F1},{1:F1},{2:F1},", meas.Power_VA[i].Max, meas.Power_VA[i].Avg, meas.Power_VA[i].Min);
        }
        s = s + String.Format("{0:F1},{1:F1},{2:F1},", meas.Frequency.Max, meas.Frequency.Avg, meas.Frequency.Min);
        s = s + String.Format("{0:F0},{1:F0},{2:F0}", meas.Temperature.Max, meas.Temperature.Avg, meas.Temperature.Min);
        return s;
    }
    // -- btnGetSecs_Click: Retrieves the number of measurements specified in tbSecsLoad and stores it in the specified file as CSV strings ----------
    private void btnGetSecs_Click(object sender, EventArgs e)
    {
        status.Text = "Starting upload";
        int secsLoad = Convert.ToInt32(tbSecsLoad.Text);
        using (System.IO.StreamWriter file = System.IO.File.AppendText(tbSecsFile.Text))
        {
            // -- Upload seconds and store in file --
            for (int j = 0; j < secsLoad; j++)
            {
                Profiler_Rev1.Profiler_Rev1_MeasurementValue result = new Profiler_Rev1.Profiler_Rev1_MeasurementValue();
                profiler.GetNextMeasurement(0, ref result);
                string s = CreateMeasLine(result);
                file.WriteLine(s);
            }
            file.Close();
        }
        status.Text = "Upload complete";
    }
    // -- btnGetMins_Click: Retrieves the number of measurements specified in tbMinsLoad and stores it in the specified file as CSV strings ----------
    private void btnGetMins_Click(object sender, EventArgs e)
    {
        int minsLoad = Convert.ToInt32(tbMinsLoad.Text);
        using (System.IO.StreamWriter file = System.IO.File.AppendText(tbMinsFile.Text))
        {
            // -- Upload minutes and store in file --
            for (int j = 0; j < minsLoad; j++)
            {
                Profiler_Rev1.Profiler_Rev1_MeasurementValue result = new Profiler_Rev1.Profiler_Rev1_MeasurementValue();
                profiler.GetNextMeasurement(1, ref result);
                string s = CreateMeasLine(result);
                file.WriteLine(s);
            }
            file.Close();
        }
    }
    // -- btnGetHrs_Click: Retrieves the number of measurements specified in tbHrsLoad and stores it in the specified file as CSV strings ----------
    private void btnGetHrs_Click(object sender, EventArgs e)
    {
        int hrsLoad = Convert.ToInt32(tbHrsLoad.Text);
        using (System.IO.StreamWriter file = System.IO.File.AppendText(tbHrsFile.Text))
        {
            // -- Upload hours and store in file --
            for (int j = 0; j < hrsLoad; j++)
            {
                Profiler_Rev1.Profiler_Rev1_MeasurementValue result = new Profiler_Rev1.Profiler_Rev1_MeasurementValue();
                profiler.GetNextMeasurement(2, ref result);
                string s = CreateMeasLine(result);
                file.WriteLine(s);
            }
            file.Close();
        }
    }
  }
}
```

```
/* ==  FRMVIEWMEAS.CS  ========================================================================================================
Loads a CSV text file of measurement data and graphs it.
=========================================================================================================================*/
using System; using System.Collections.Generic; using System.ComponentModel; using System.Linq;using System.Windows.Forms;
using System.Windows.Forms.DataVisualization.Charting;
namespace ProfilerCalibrator
{
    // -- tStatDataValue: Structure used to hold a measurement value retrieved form the CSV file. ----------
    public struct tStatDataValue {  public double Max, Min, Avg;  }
    // -- tProfileDataElement: Holds an entire CSV line of measurement data. ----------
    public struct tProfileDataElement
    {
```

```
    public string Timestamp;
    public tStatDataValue[] VoltageRMS, CurrentRMS, ActivePower, ReactivePower, ApparentPower;
    public tStatDataValue Frequency, Temperature;
}
public partial class frmViewMeas : Form
{
    System.Data.DataTable profileData = new System.Data.DataTable("Profile Data");              // Data table housing the retrieved measurement values
    // -- Form constructor ------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
    public frmViewMeas() {  InitializeComponent();  }
    // -- StatValueFromArray: Reads the statistical measurement value from the array of CSV strings and increments the array index. ----------------------------------
    public tStatDataValue StatValueFromArray(string[] data, ref int idx)
    {
        tStatDataValue temp = new tStatDataValue();
        temp.Max = Convert.ToDouble(data[idx++]); temp.Avg = Convert.ToDouble(data[idx++]); temp.Min = Convert.ToDouble(data[idx++]);
        return temp;
    }
    // -- LoadDataSet: Reads the measurement data from 'filename' and places it in the profileData data table. ------------------------------------------------------
    public void LoadDataSet(string filename)
    {
        int valueIdx = 0;
        profileData.Clear();
        // -- Construct the data table columns --
        profileData.Columns.Add("Timestamp", typeof(string));
        // -- Voltage RMS --
        for (int i=0;i<3;i++) profileData.Columns.Add("Voltage L" + (i + 1) + " (Vrms)", typeof(tStatDataValue));
        // -- Current RMS --
        for (int i = 0; i < 12; i++) profileData.Columns.Add("Current " + (i + 1) + " (Arms)", typeof(tStatDataValue));
        // -- Active Power --
        for (int i = 0; i < 12; i++) profileData.Columns.Add("Active Power " + (i + 1) + " (W)", typeof(tStatDataValue));
        // -- Reactive Power --
        for (int i = 0; i < 12; i++) profileData.Columns.Add("Reactive Power " + (i + 1) + " (VAR)", typeof(tStatDataValue));
        // -- Reactive Power --
        for (int i = 0; i < 12; i++) profileData.Columns.Add("Apparent Power " + (i + 1) + " (VA)", typeof(tStatDataValue));
        // -- Frequency and Temperature --
        profileData.Columns.Add("Frequency (Hz)", typeof(tStatDataValue));
        profileData.Columns.Add("Temperature (°C)", typeof(tStatDataValue));
        // -- Update channel list box --
        cobLeftAxis.Items.Clear();
        for (int i = 1; i < profileData.Columns.Count; i++) cobLeftAxis.Items.Add(profileData.Columns[i].Caption);
        // -- Read the data from the CSV file line-by-line and place in the data table --
        using (System.IO.StreamReader file = new System.IO.StreamReader(filename))
        {
            string line;
            while ((line = file.ReadLine()) != null)
            {
                tProfileDataElement element = new tProfileDataElement();
                element.VoltageRMS = new tStatDataValue[3]; element.CurrentRMS = new tStatDataValue[12];
                element.ActivePower = new tStatDataValue[12]; element.ReactivePower = new tStatDataValue[12];
                element.ApparentPower = new tStatDataValue[12];
                string[] values = line.Split(',');
                valueIdx = 0;
                element.Timestamp = values[valueIdx++];
                for (int i = 0; i < 3; i++) element.VoltageRMS[i] = StatValueFromArray(values, ref valueIdx);
                for (int i = 0; i < 12; i++)
                {
                    element.CurrentRMS[i] = StatValueFromArray(values, ref valueIdx);
                    element.ActivePower[i] = StatValueFromArray(values, ref valueIdx);
                    element.ReactivePower[i] = StatValueFromArray(values, ref valueIdx);
                    element.ApparentPower[i] = StatValueFromArray(values, ref valueIdx);
                }
                element.Frequency = StatValueFromArray(values, ref valueIdx);
                element.Temperature = StatValueFromArray(values, ref valueIdx);
                // -- Add a row of measurement data to the table --
                profileData.Rows.Add(new object[] { element.Timestamp,
                    element.VoltageRMS[0], element.VoltageRMS[1], element.VoltageRMS[2],
                    element.CurrentRMS[0], element.CurrentRMS[1], element.CurrentRMS[2], element.CurrentRMS[3], element.CurrentRMS[4],
                    element.CurrentRMS[5], element.CurrentRMS[6], element.CurrentRMS[7], element.CurrentRMS[8], element.CurrentRMS[9],
                    element.CurrentRMS[10], element.CurrentRMS[11],
                    element.ActivePower[0], element.ActivePower[1], element.ActivePower[2], element.ActivePower[3], element.ActivePower[4],
                    element.ActivePower[5], element.ActivePower[6], element.ActivePower[7], element.ActivePower[8], element.ActivePower[9],
                    element.ActivePower[10], element.ActivePower[11],
                    element.ReactivePower[0], element.ReactivePower[1], element.ReactivePower[2], element.ReactivePower[3], element.ReactivePower[4],
                    element.ReactivePower[5], element.ReactivePower[6], element.ReactivePower[7], element.ReactivePower[8], element.ReactivePower[9],
                    element.ReactivePower[10], element.ReactivePower[11],
                    element.ApparentPower[0], element.ApparentPower[1], element.ApparentPower[2], element.ApparentPower[3], element.ApparentPower[4],
                    element.ApparentPower[5], element.ApparentPower[6], element.ApparentPower[7], element.ApparentPower[8], element.ApparentPower[9],
```

```
                element.ApparentPower[10], element.ApparentPower[11],
                element.Frequency, element.Temperature });
            }
        }
    }
    // -- AddDataToChart: Creates a new fastline series and creates the data points as specified by 'xvalues' and 'values'. ----------
    private void AddDataToChart(Chart chart, string[] xValues, double[] values, string dataTitle, bool leftAxis)
    {
        chart.Series.Add(dataTitle);                                    // Add the new series
        chart.Series[chart.Series.Count() - 1].ChartType = SeriesChartType.FastLine;    // Specify the series is a fastline
        chart.Series[chart.Series.Count() - 1].Name = dataTitle;        // Title the series so it is displayed in the legend
        if (!leftAxis) chart.Series[chart.Series.Count() - 1].YAxisType = AxisType.Secondary;    // Specify if series uses the left or right axis
        for (int i = 0; i < values.Count(); i++)                        // Add the X and Y data points
            chart.Series[chart.Series.Count() - 1].Points.AddXY(xValues[i], values[i]);
    }
    // -- AddPlot: Extracts the chart data from the data table and adds the series to the chart using the correct axis as reference. ----------
    private void AddPlot(bool leftYAxis)
    {
        tStatDataValue[] values = new tStatDataValue[profileData.Rows.Count];
        for (int i = 0; i < values.Count(); i++) values[i] = (tStatDataValue)profileData.Rows[i][cobLeftAxis.SelectedIndex + 1];
        string[] xValues = new string[profileData.Rows.Count];
        for (int i = 0; i < xValues.Count(); i++) xValues[i] = (string)profileData.Rows[i][0];
        // -- Plot maximum values for data channel --
        if (cbLeftMax.Checked)
        {
            double[] measValue = new double[values.Count()];
            for (int i = 0; i < measValue.Count(); i++) measValue[i] = values[i].Max;
            AddDataToChart(chart1, xValues, measValue, cobLeftAxis.Text + " Max", leftYAxis);
        }
        // -- Plot average values for data channel --
        if (cbLeftAvg.Checked)
        {
            double[] measValue = new double[values.Count()];
            for (int i = 0; i < measValue.Count(); i++) measValue[i] = values[i].Avg;
            AddDataToChart(chart1, xValues, measValue, cobLeftAxis.Text + " Avg", leftYAxis);
        }
        // -- Plot minimum values for data channel --
        if (cbLeftMin.Checked)
        {
            double[] measValue = new double[values.Count()];
            for (int i = 0; i < measValue.Count(); i++) measValue[i] = values[i].Min;
            AddDataToChart(chart1, xValues, measValue, cobLeftAxis.Text + " Min", leftYAxis);
        }
    }
    // -- btnAddLeftY_Click: Add the specified data channel to the left axis of the chart. ----------
    private void btnAddLeftY_Click(object sender, EventArgs e) {  AddPlot(true);  }
    // -- btnAddRightY_Click: Add the specified data channel to the right axis of the chart. ----------
    private void btnAddRightY_Click(object sender, EventArgs e) {  AddPlot(false); }
    // -- btnClearChart_Click: Clear the chart data. ----------
    private void btnClearChart_Click(object sender, EventArgs e) {  chart1.Series.Clear(); }
  }
}
```

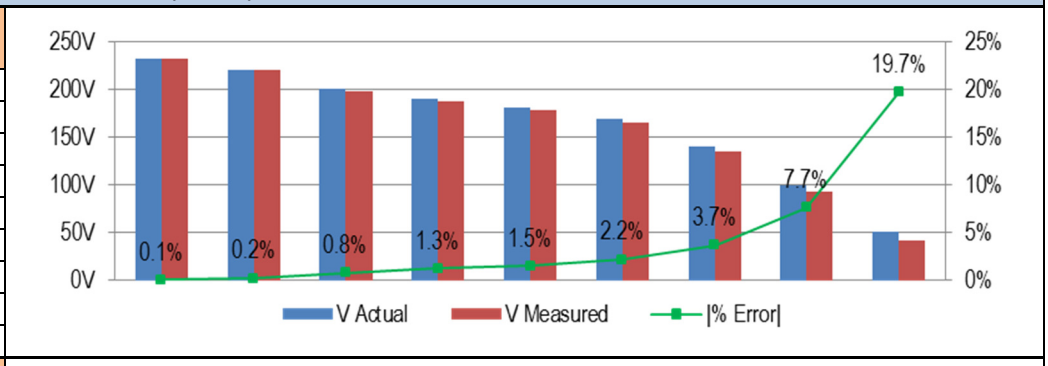
Appendix D

Calibration Results

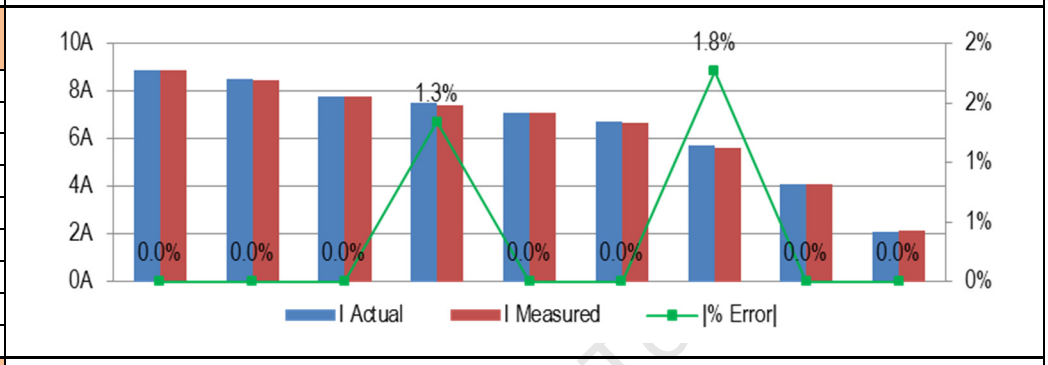
Table D.1 – Calibration Results

Parameter	Channel 1	Channel 2	Channel 3	Channel 4	Channel 5	Channel 6	Channel 7	Channel 8	Channel 9	Channel 10	Channel 11	Channel 12
PGA Gain	0	0	0	0	0	0	0	0	0	0	0	0
Vrms Gain	0	0	0	0	0	0	0	0	0	0	0	0
Irms Gain	0	0	0	0	0	0	0	0	0	0	0	0
Watt Gain	2047	2047	2047	2047	2047	2047	2047	2047	2047	2047	2047	2047
VAR Gain	2047	2047	2047	2047	2047	2047	2047	2047	2047	2047	2047	2047
VA Gain	2047	2047	2047	2047	2047	2047	2047	2047	2047	2047	2047	2047
Vrms Offset	-445	-463	-445	-433	-441	-433	-505	-432	-484	-452	-488	-445
Irms Offset	80	76	39	73	102	87	108	52	125	118	67	173
Watt Offset	0	0	0	0	0	0	0	0	0	0	0	0
VAR Offset	0	0	0	0	0	0	0	0	0	0	0	0
Phase Calibration	-64	-64	-64	-64	-64	-64	-64	-64	-64	-64	-64	-64
Watt Divider	0	0	0	0	0	0	0	0	0	0	0	0
VAR Divider	0	0	0	0	0	0	0	0	0	0	0	0
VA Divider	0	0	0	0	0	0	0	0	0	0	0	0
Irms Normalize	1.85E-07	1.85E-07	1.84E-07	1.85E-07	1.85E-07	1.85E-07	1.89E-07	1.89E-07	1.88E-07	1.85E-07	1.85E-07	1.85E-07
Vrms 1PH2W CF	0.000455182	0.000456229	0.000455156	0.000454594	0.000454286	0.000453352	0.000468438	0.000464353	0.000466973	0.00045632	0.000458246	0.000456175
Irms 1PH2W CF	5.60E-05	2.77E-05	2.76E-05	2.77E-05	2.76E-05	2.81E-05	2.77E-05	2.77E-05	2.77E-05	2.77E-05	2.77E-05	2.77E-05
Watt 1PH2W CF	3.815165877	1.961500975	1.95863747	1.950096899	1.935096154	1.953883495	1.951842998	1.951842998	1.951842998	1.951842998	1.951842998	1.951842998
VAR 1PH2W CF	4.132516277	2	1.993349344	2.001483955	1.990774908	2.02629108	2.002379857	2.002379857	2.002379857	2.002379857	2.002379857	2.002379857
VA 1PH2W CF	4.208059981	2.088048365	2.077421345	2.078382966	2.067219153	2.096498054	2.081513976	2.081513976	2.081513976	2.081513976	2.081513976	2.081513976
Vrms 3PH3W CF	0.000455182	0.000456229	0.000455156	0.000454594	0.000454286	0.000453352	0.000468438	0.000464353	0.000466973	0.00045632	0.000458246	0.000456175
Irms 3PH3W CF	5.60E-05	2.77E-05	2.76E-05	2.77E-05	2.76E-05	2.81E-05	2.77E-05	2.77E-05	2.77E-05	2.77E-05	2.77E-05	2.77E-05
Watt 3PH3W CF	3.815165877	1.961500975	1.95863747	1.950096899	1.935096154	1.953883495	1.951842998	1.951842998	1.951842998	1.951842998	1.951842998	1.951842998
VAR 3PH3W CF	4.132516277	2	1.993349344	2.001483955	1.990774908	2.02629108	2.002379857	2.002379857	2.002379857	2.002379857	2.002379857	2.002379857
VA 3PH3W CF	4.208059981	2.088048365	2.077421345	2.078382966	2.067219153	2.096498054	2.081513976	2.081513976	2.081513976	2.081513976	2.081513976	2.081513976
Vrms 3PH4W CF	0.000451608	0.000458992	0.000454483	0.000451009	0.000456986	0.000452652	0.000464654	0.000467022	0.000466174	0.000452634	0.000460932	0.000455418
Irms 3PH4W CF	5.59E-05	2.76E-05	2.76E-05	2.77E-05	2.76E-05	2.81E-05	2.77E-05	2.77E-05	2.77E-05	2.77E-05	2.77E-05	2.77E-05
Watt 3PH4W CF	3.801768264	1.973906741	1.959232614	1.934185606	1.948485571	1.955949246	1.954351956	1.954351956	1.954351956	1.954351956	1.954351956	1.954351956
VAR 3PH4W CF	4.104092173	2.017184144	1.992669753	1.990749663	2.006994366	2.02747792	2.007015169	2.007015169	2.007015169	2.007015169	2.007015169	2.007015169
VA 3PH4W CF	4.169040836	2.101483964	2.073689183	2.060387985	2.079254815	2.093134139	2.081590017	2.081590017	2.081590017	2.081590017	2.081590017	2.081590017

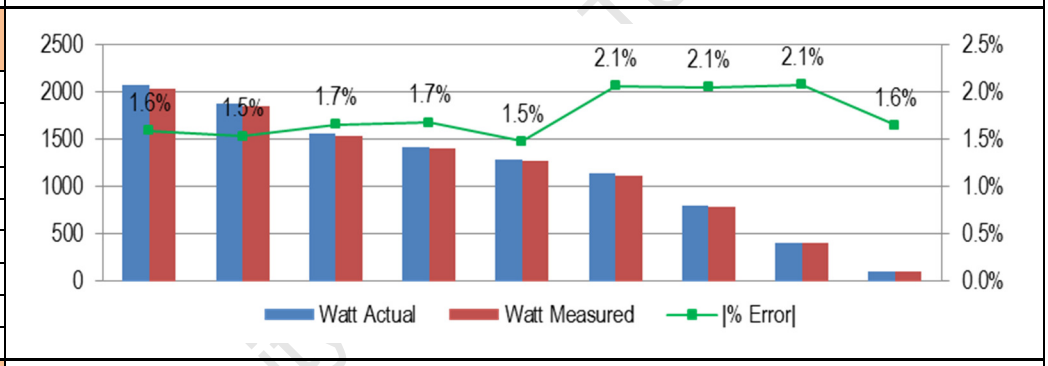
Table D.2 – Calibration Verification (PF=1.0)	
V Actual	**V Meas.**
232.3	232.5
220.5	220.1
200.3	198.7
190.5	188.1
180.8	178.0
169.5	165.8
140.5	135.4
100.3	92.9
51.2	42.0



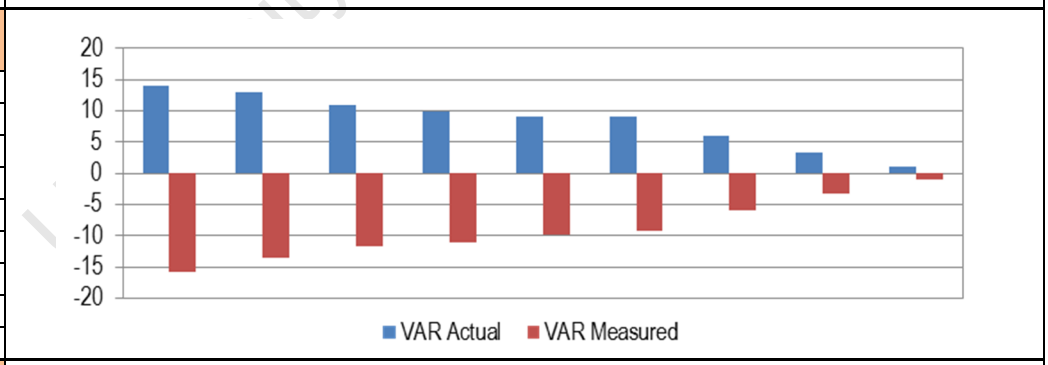
I Actual	**I Meas.**
8.9	8.866
8.5	8.471
7.8	7.778
7.5	7.434
7.1	7.093
6.7	6.684
5.7	5.630
4.1	4.091
2.1	2.135



Watt Actual	**Watt Meas.**
2072.0	2029.4
1878.0	1840.3
1568.0	1534.4
1424.0	1393.2
1287.0	1261.8
1143.0	1113.9
798.0	777.7
413.0	402.5
110.0	107.7



VAR Actual	**VAR Meas.**
14.0	-24.4
13.0	-21.5
11.0	-18.3
10.0	-17
9.0	-15.3
9.0	-14.1
6.0	-9.2
3.4	-5.1
1.1	-1.3



VA Actual	**VA Meas.**
2072.0	2071.1
1878.0	1872.5
1568.0	1552.4
1424.0	1405.1
1287.0	1268.3
1143.0	1114.3
798.0	766.1
413.0	382.4
110.0	90.0

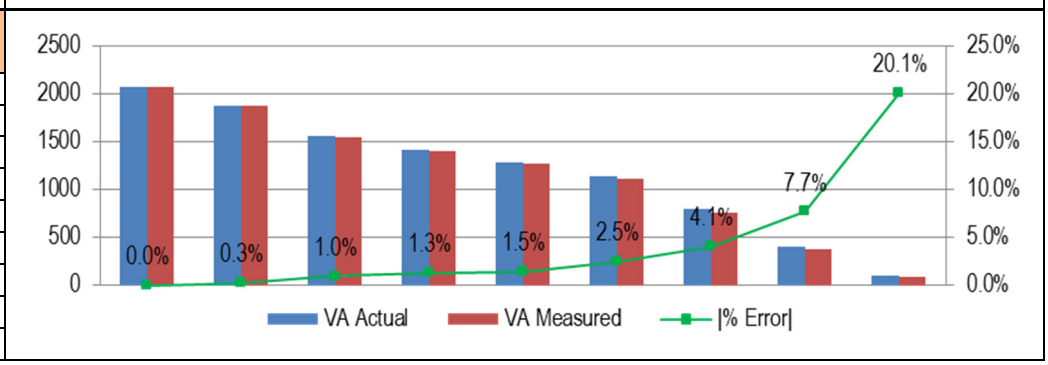
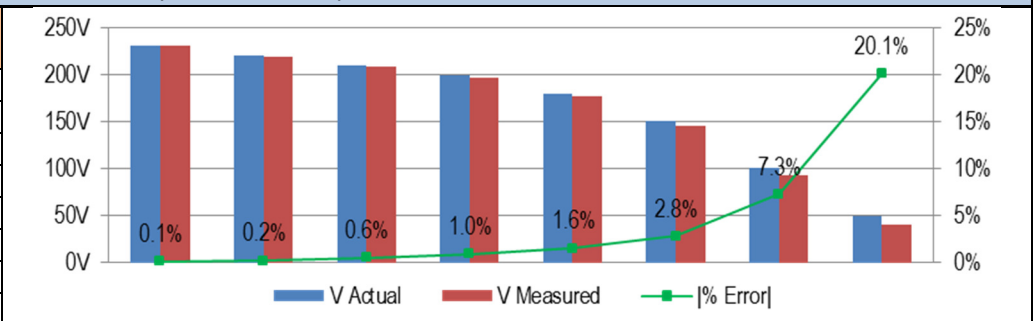
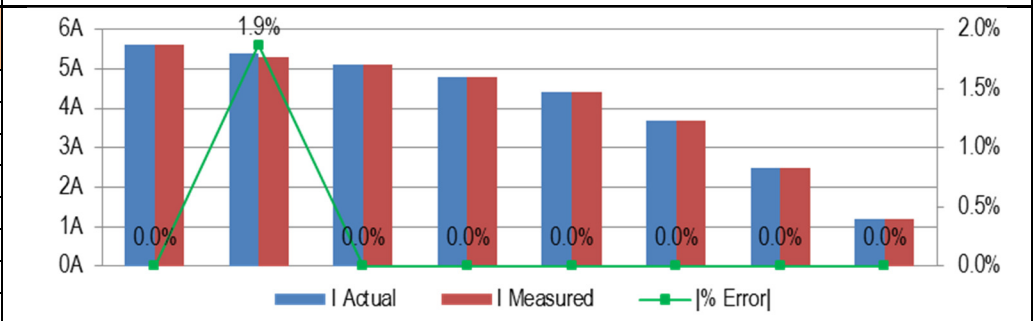


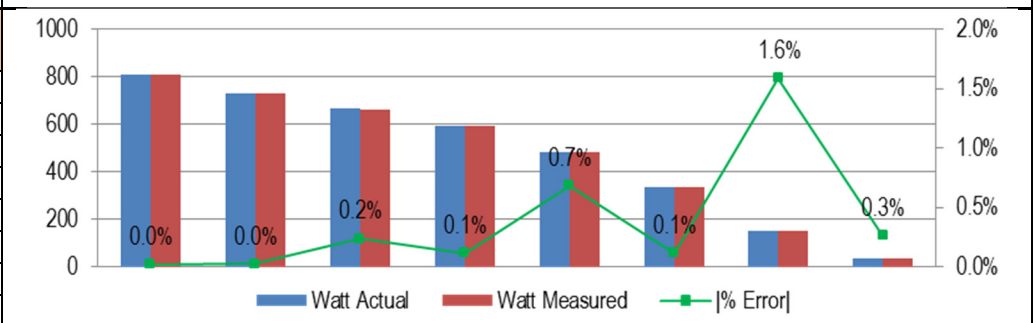
Table D.3 – Calibration Verification (PF=0.6 Inductive)	
V Actual	**V Meas.**
231.1	231.4
220.5	220.0
210.5	209.3
199.2	197.3
180.3	177.5
150.5	146.3
100.8	93.7
50.3	41.1



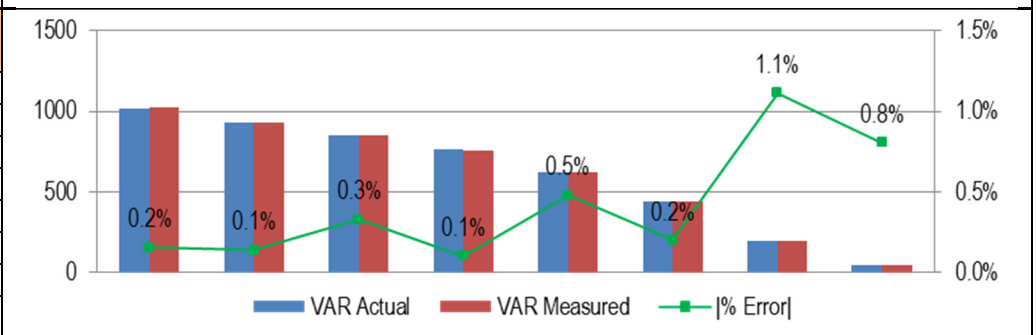
I Actual	**I Meas.**
5.6	5.597
5.4	5.331
5.1	5.089
4.8	4.815
4.4	4.370
3.7	3.656
2.5	2.463
1.2	1.232



Watt Actual	**Watt Meas.**
808.0	808.2
731.0	730.8
665.0	663.4
593.0	592.3
485.0	481.7
337.6	337.2
152.0	149.6
37.7	37.6



VAR Actual	**VAR Meas.**
1020.0	1021.6
930.0	928.7
850.0	847.2
761.0	760.2
626.0	623.0
440.0	439.1
199.0	196.8
49.8	49.4



VA Actual	**VA Meas.**
1301.0	1300.9
1183.0	1178.5
1078.0	1070.4
964.0	954.7
793.0	774.1
554.0	537.3
250.0	229.6
62.4	50.9

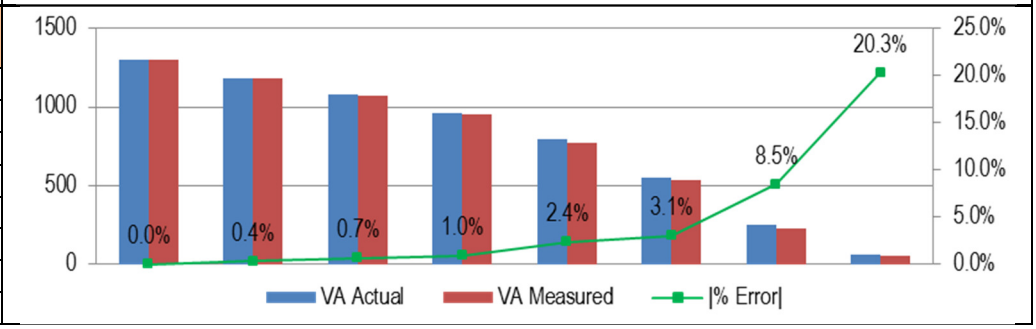
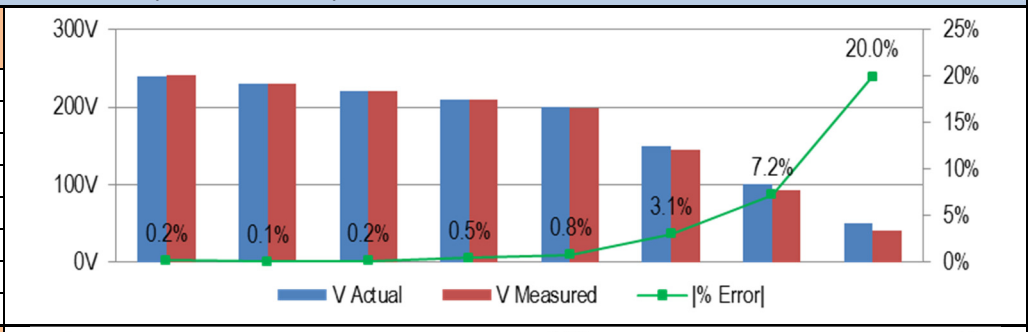
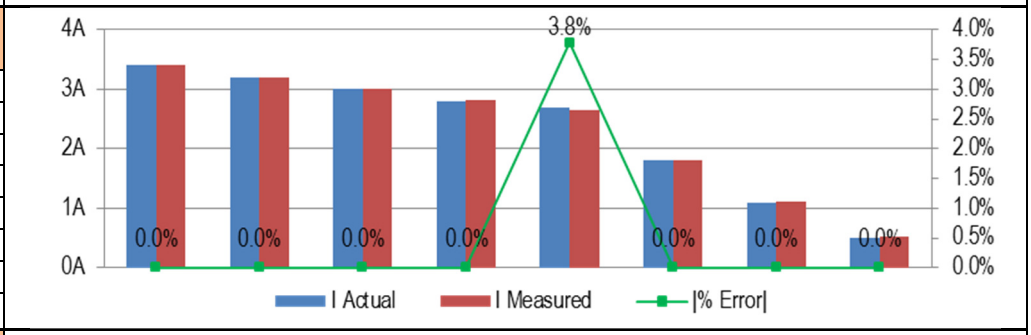


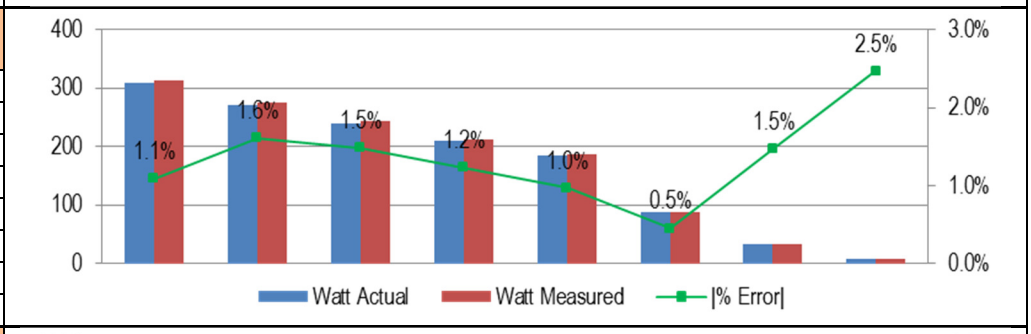
Table D.4 – Calibration Verification (PF=0.4 Inductive)	
V Actual	**V Meas.**
240.1	240.6
230.1	230.3
220.3	219.9
210.0	209.0
200.1	198.5
149.7	145.2
100.4	93.4
50.7	41.5



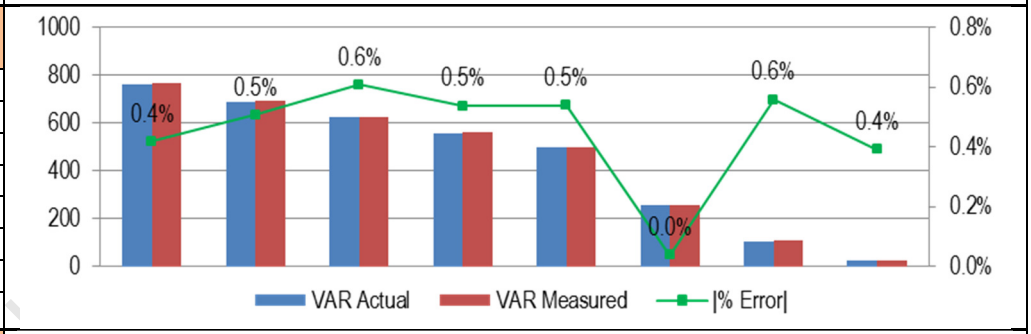
I Actual	**I Meas.**
3.4	3.408
3.2	3.201
3.0	3.014
2.8	2.819
2.7	2.641
1.8	1.808
1.1	1.111
0.5	0.521



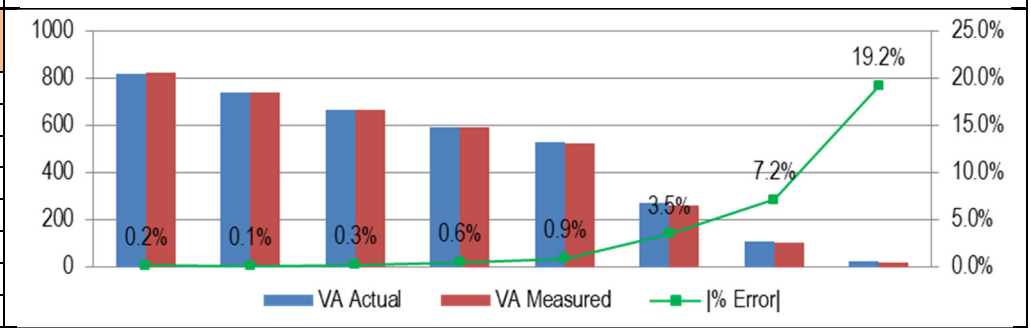
Watt Actual	**Watt Meas.**
309.0	312.4
271.0	275.4
240.0	243.6
210.0	212.6
184.5	186.3
87.5	87.9
33.8	34.3
8.2	8.0



VAR Actual	**VAR Meas.**
762.0	765.2
689.0	692.5
623.0	626.8
557.0	560.0
498.7	501.4
258.7	258.8
107.2	107.8
25.5	25.6



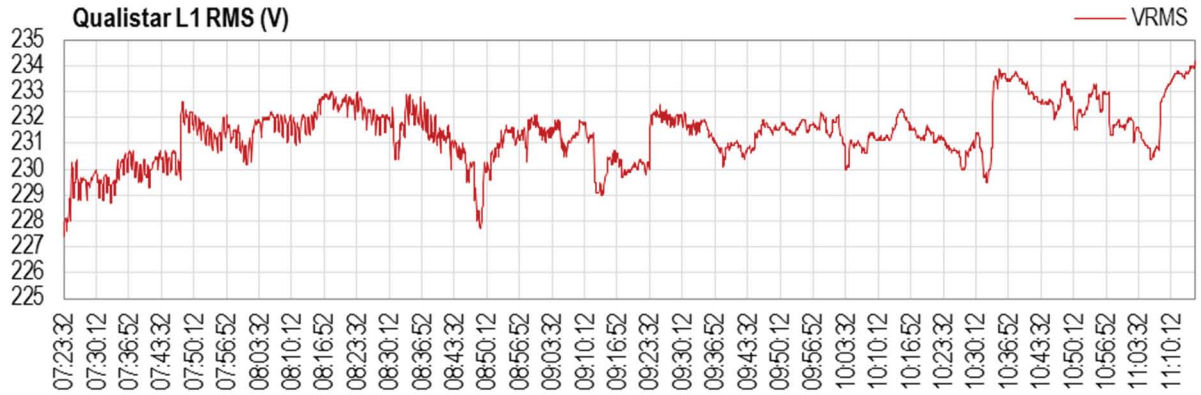
VA Actual	**VA Meas.**
822.0	823.9
741.0	740.0
668.0	666.2
595.0	591.6
532.0	527.1
273.0	263.5
112.4	104.6
26.8	22.1

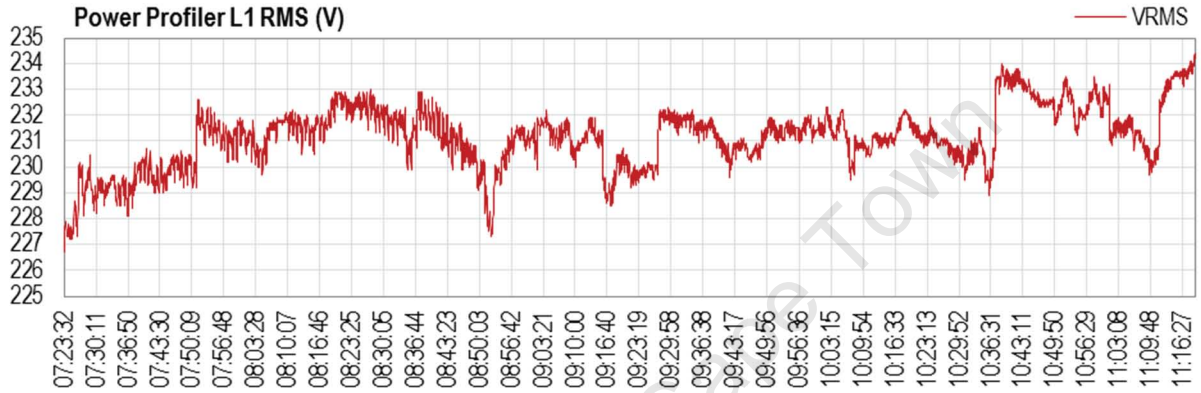


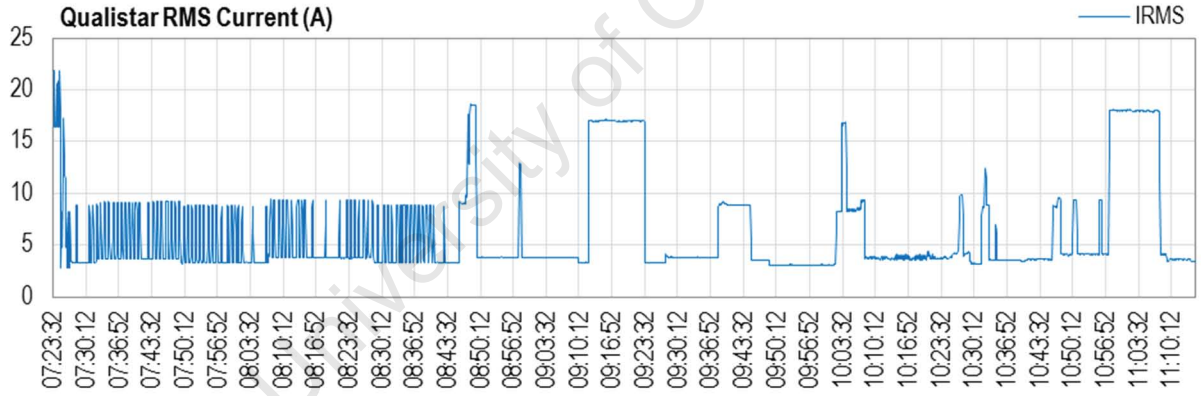
Appendix E

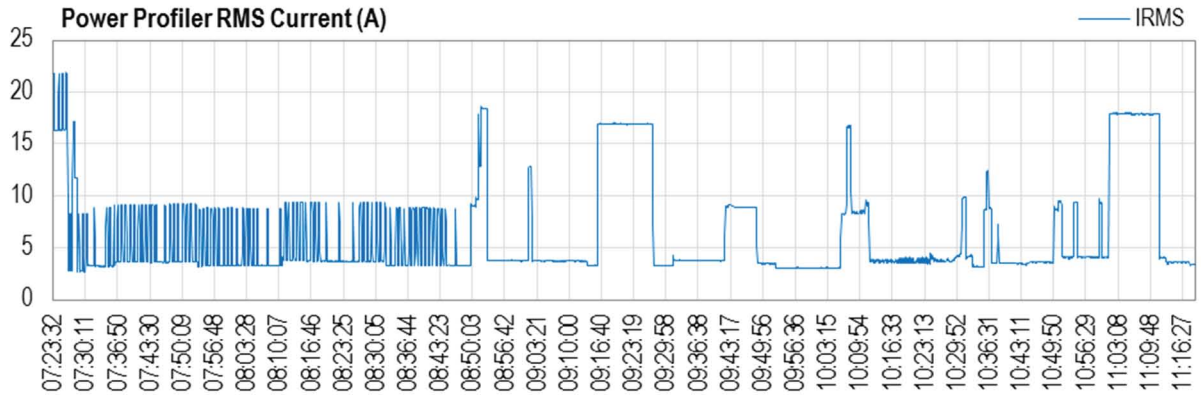
Field-Testing Profile Data

Comparison RMS Voltage and Current Measurements

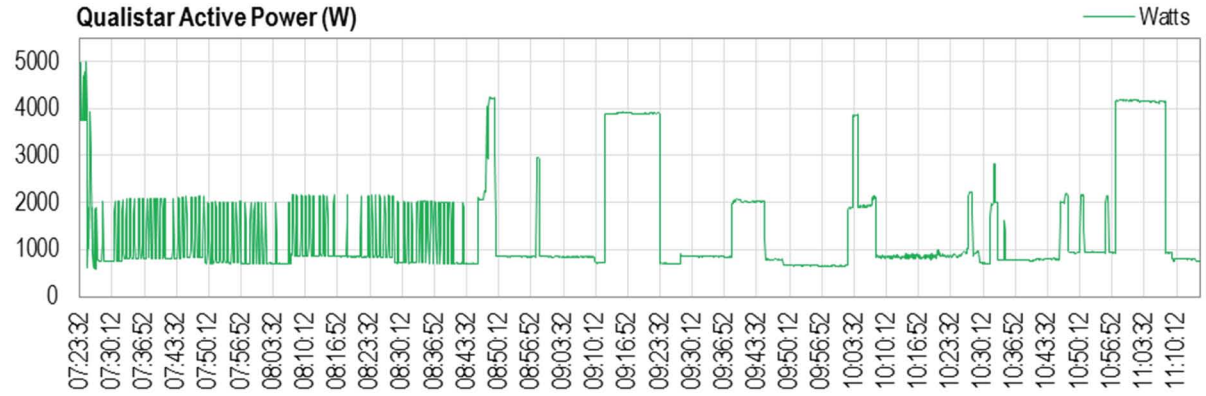


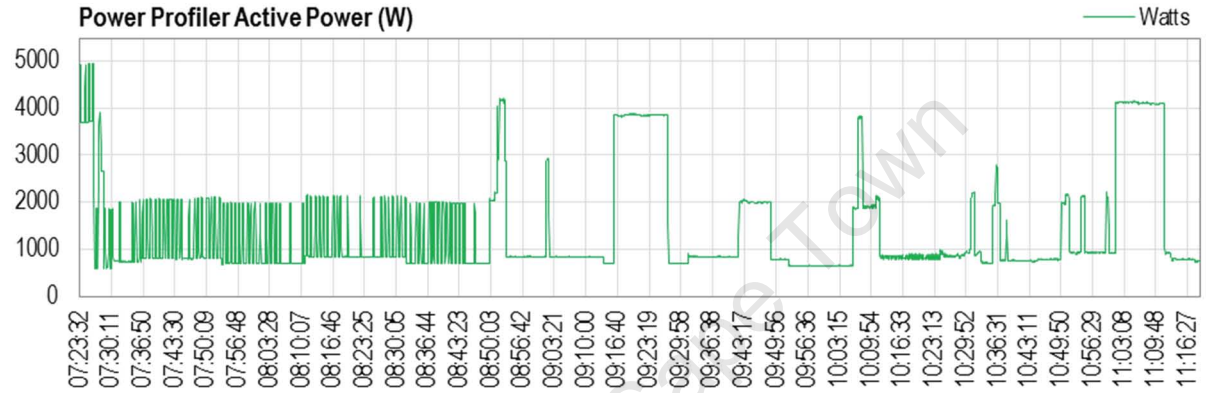


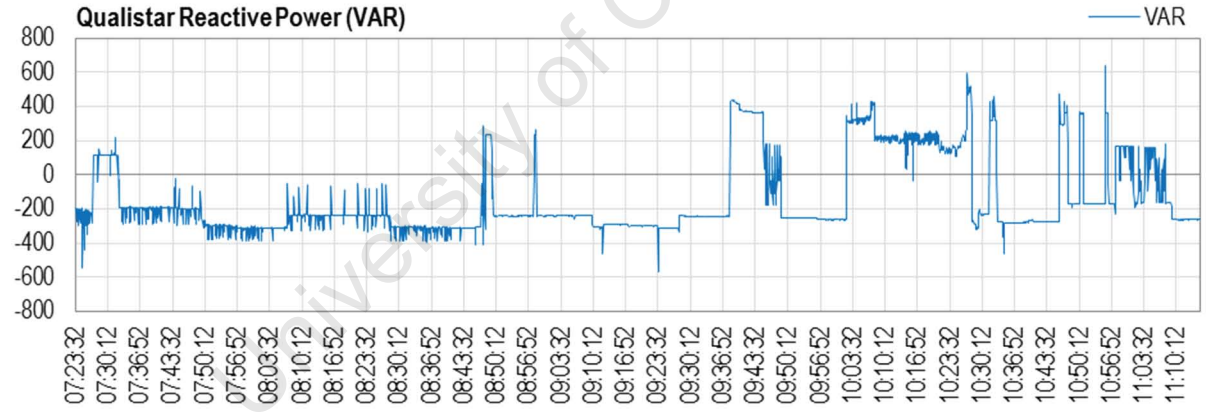


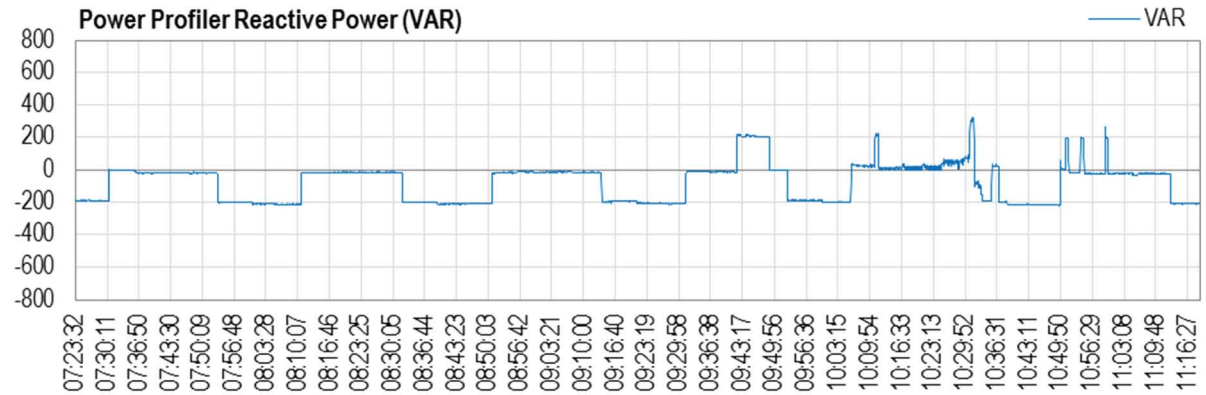


Comparison Active and Reactive Power Measurements

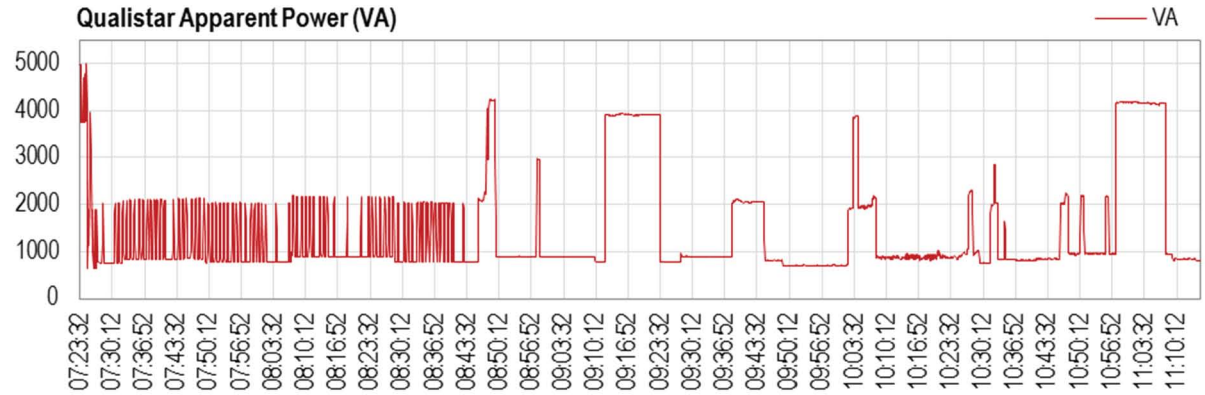


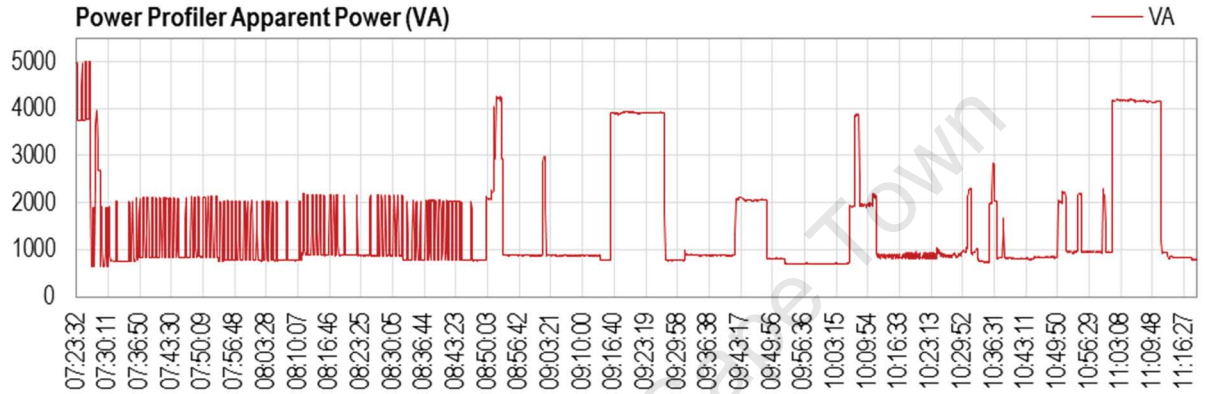


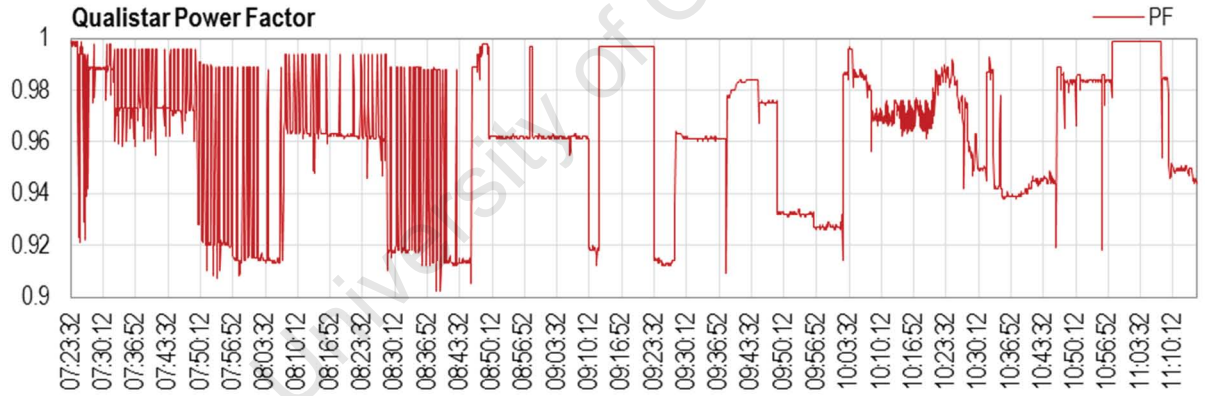


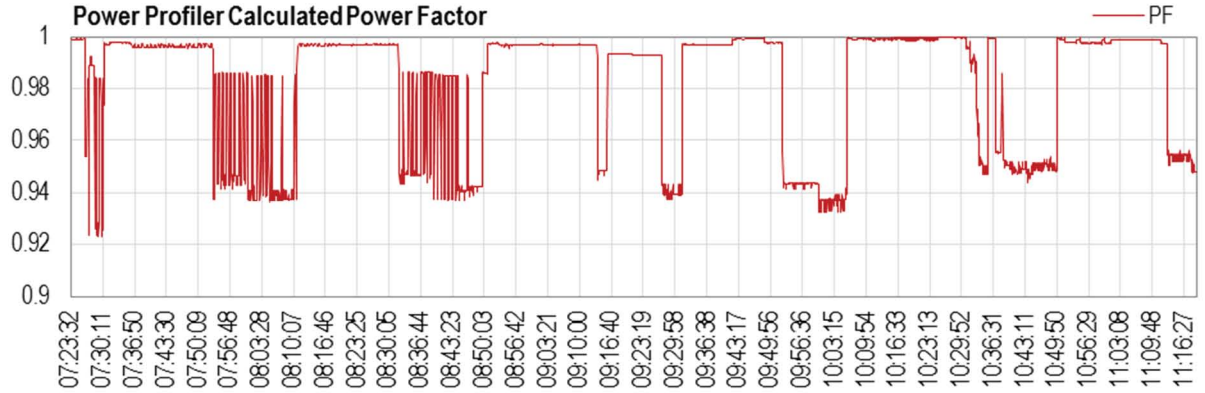


Comparison Apparent Power and Power Factor Measurements

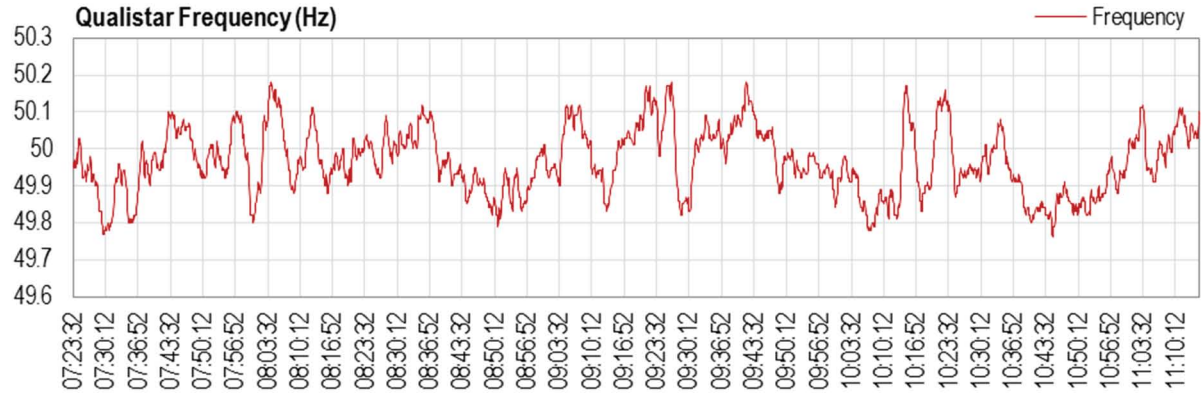


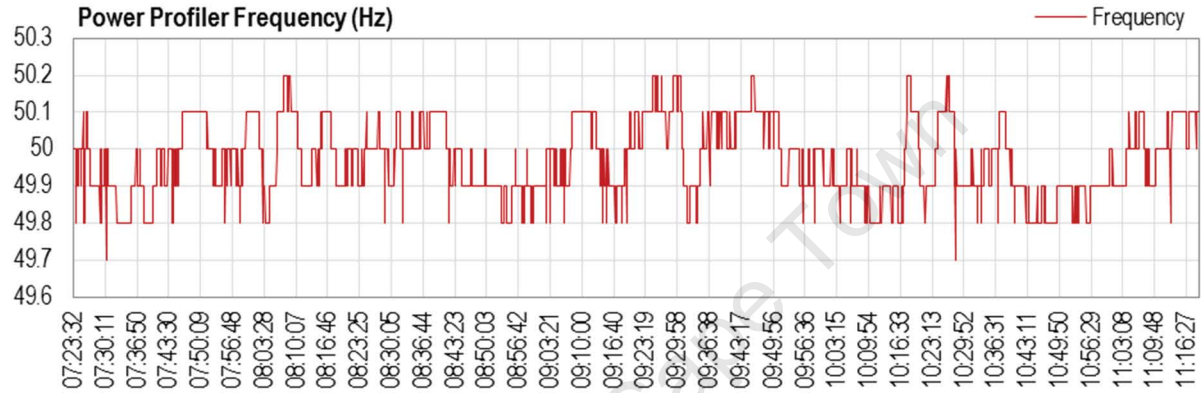






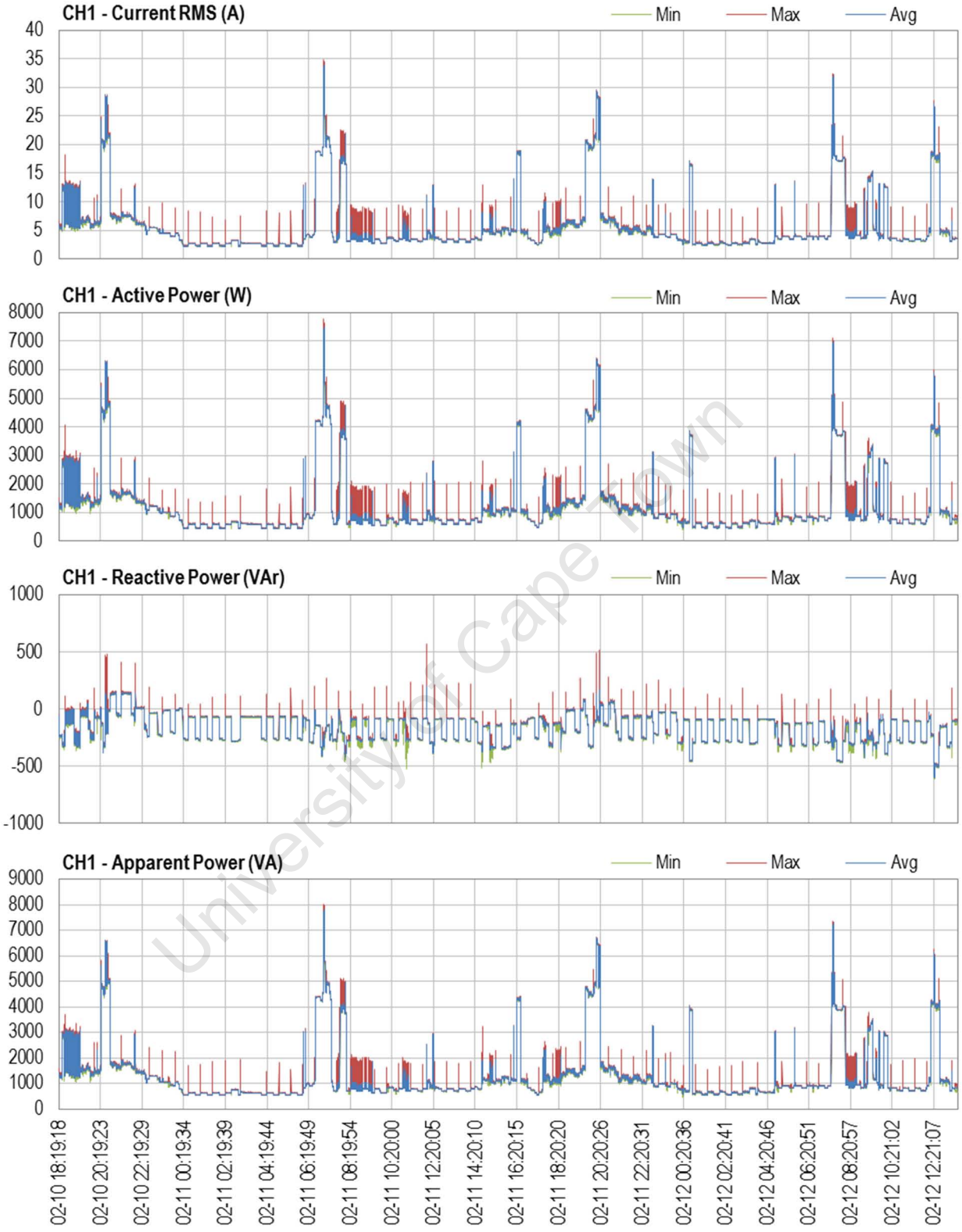
Comparison Frequency Measurements



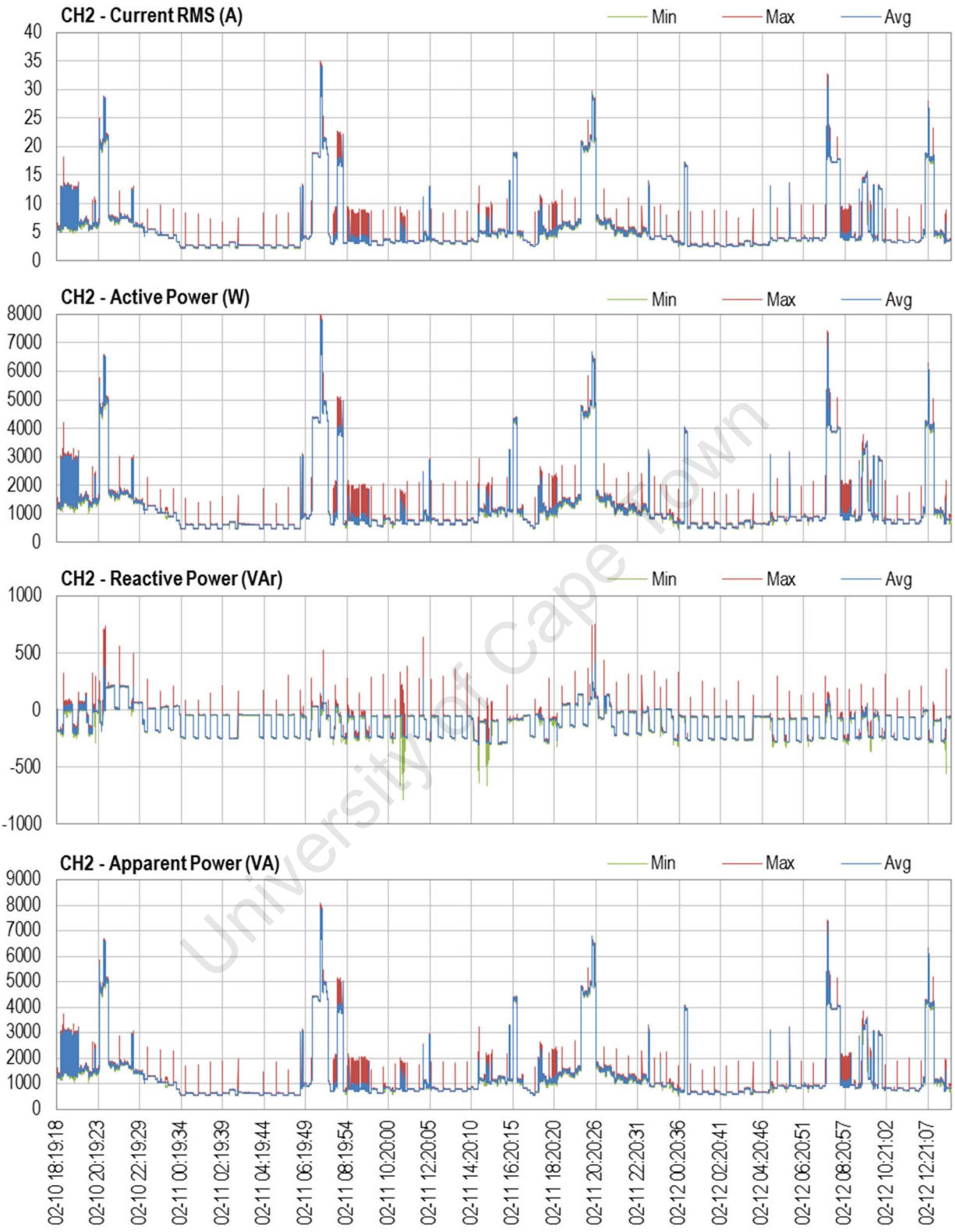


5 sec - Voltage Channels (L1, L2, L3) and Frequency

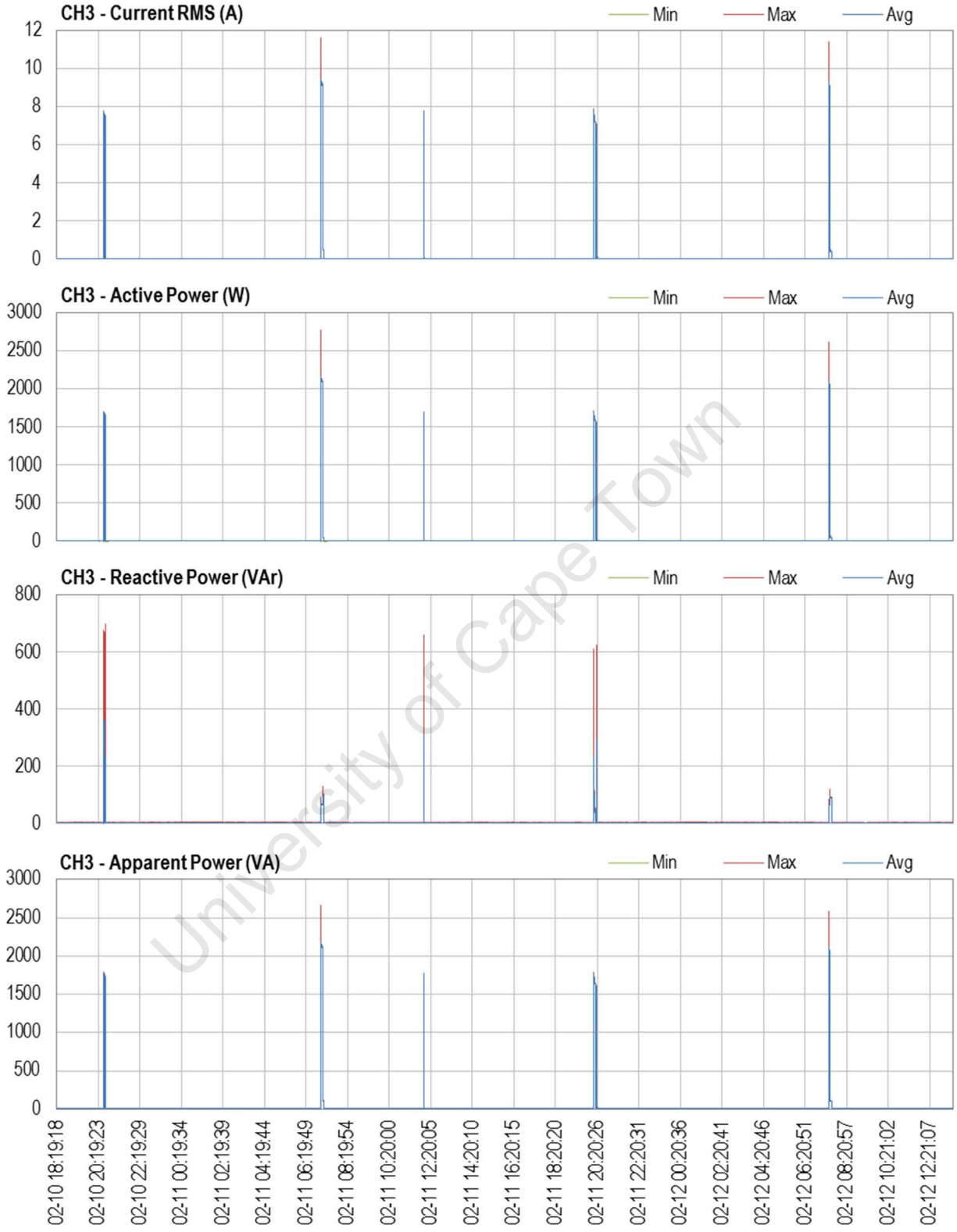
5 sec - Channel 1 – Main Supply (106A_{RMS} CT range)



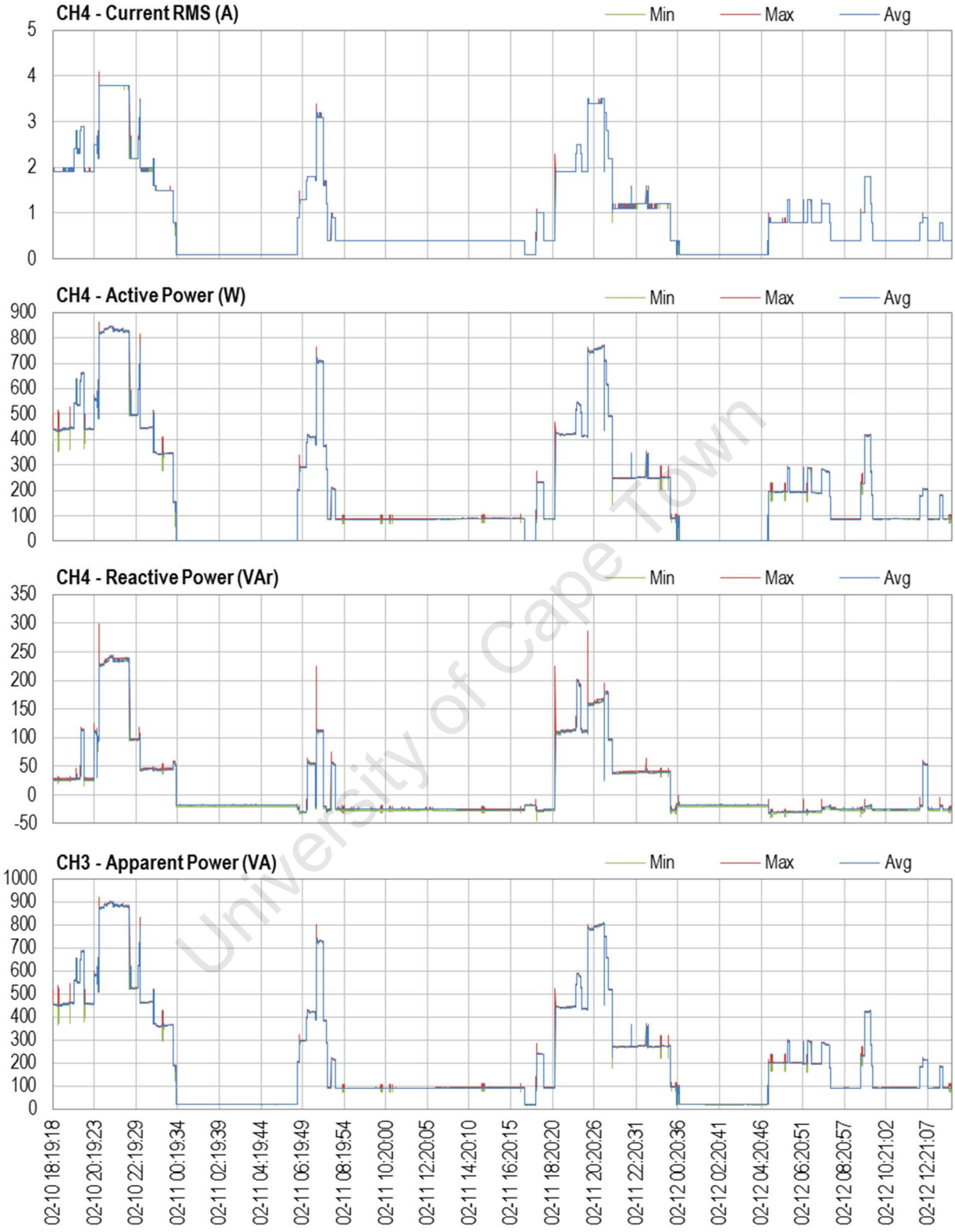
5 sec - Channel 2 – Main Supply (53A$_{RMS}$ CT range)



5 sec - Channel 3 – Oven Circuit (53A$_{RMS}$ CT range)



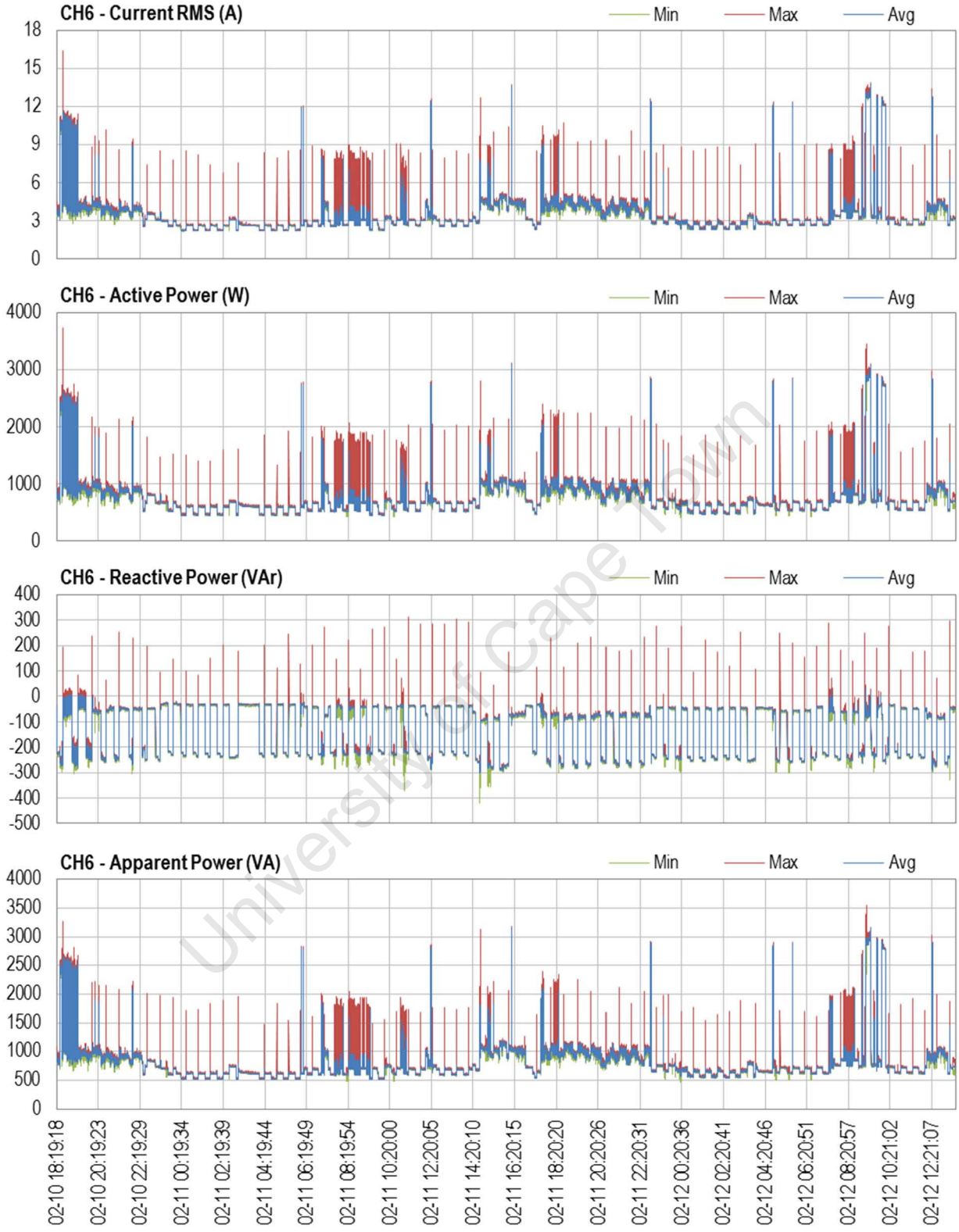
5 sec - Channel 4 – Lights Circuit (53A_{RMS} CT range)



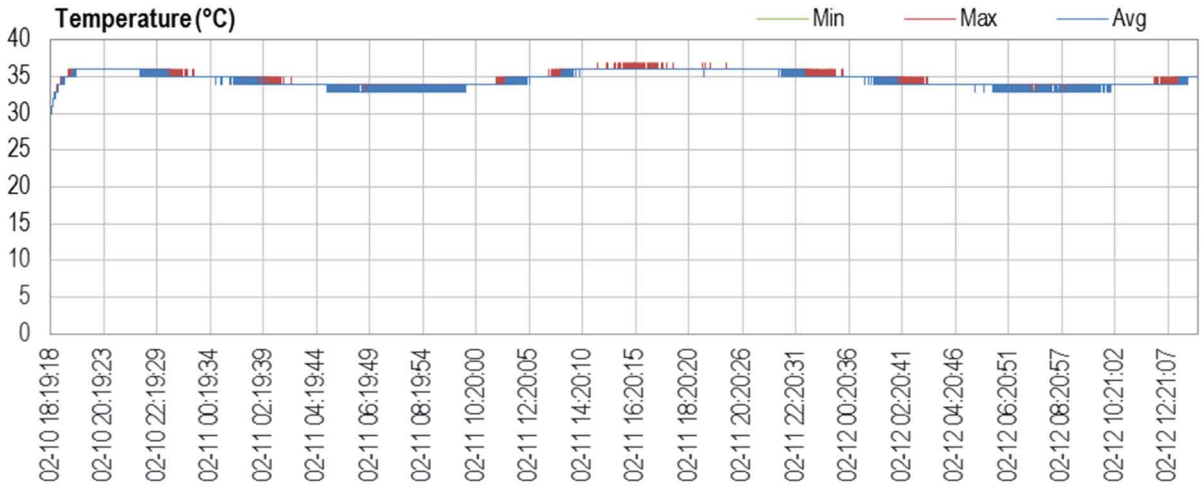
5 sec - Channel 5 – Geyser Circuit (53A_{RMS} CT range)

Please note: During the first period of the field test the split-core CT on the Geyser circuit was not correctly coupled, resulting in the sizeable reactive power measurement before 11 February 16:20. This did not impact the overall measurement data as this was taken from channel 2.

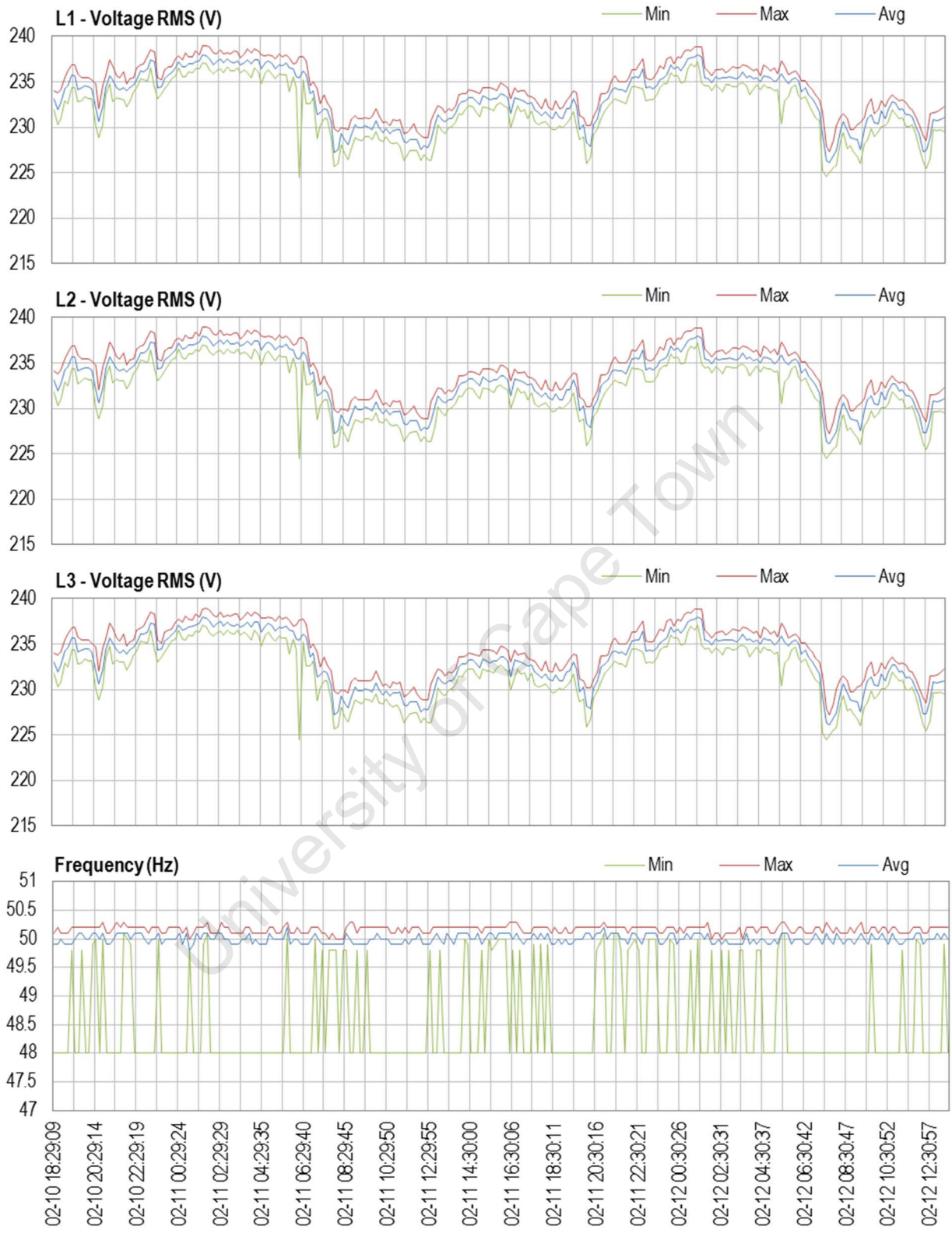
5 sec - Channel 6 – Plugs Circuit (53A_{RMS} CT range)



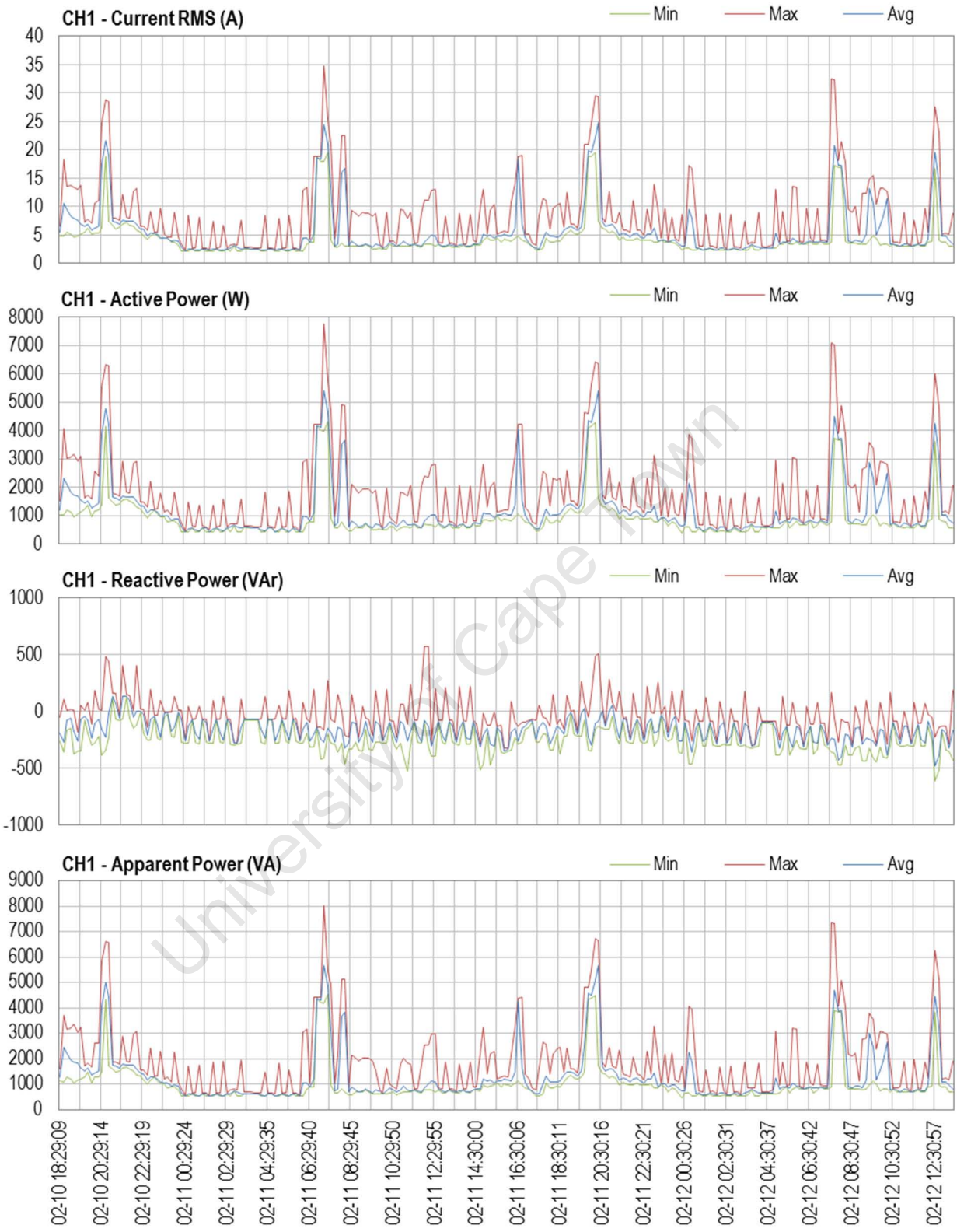
5 sec - Temperature



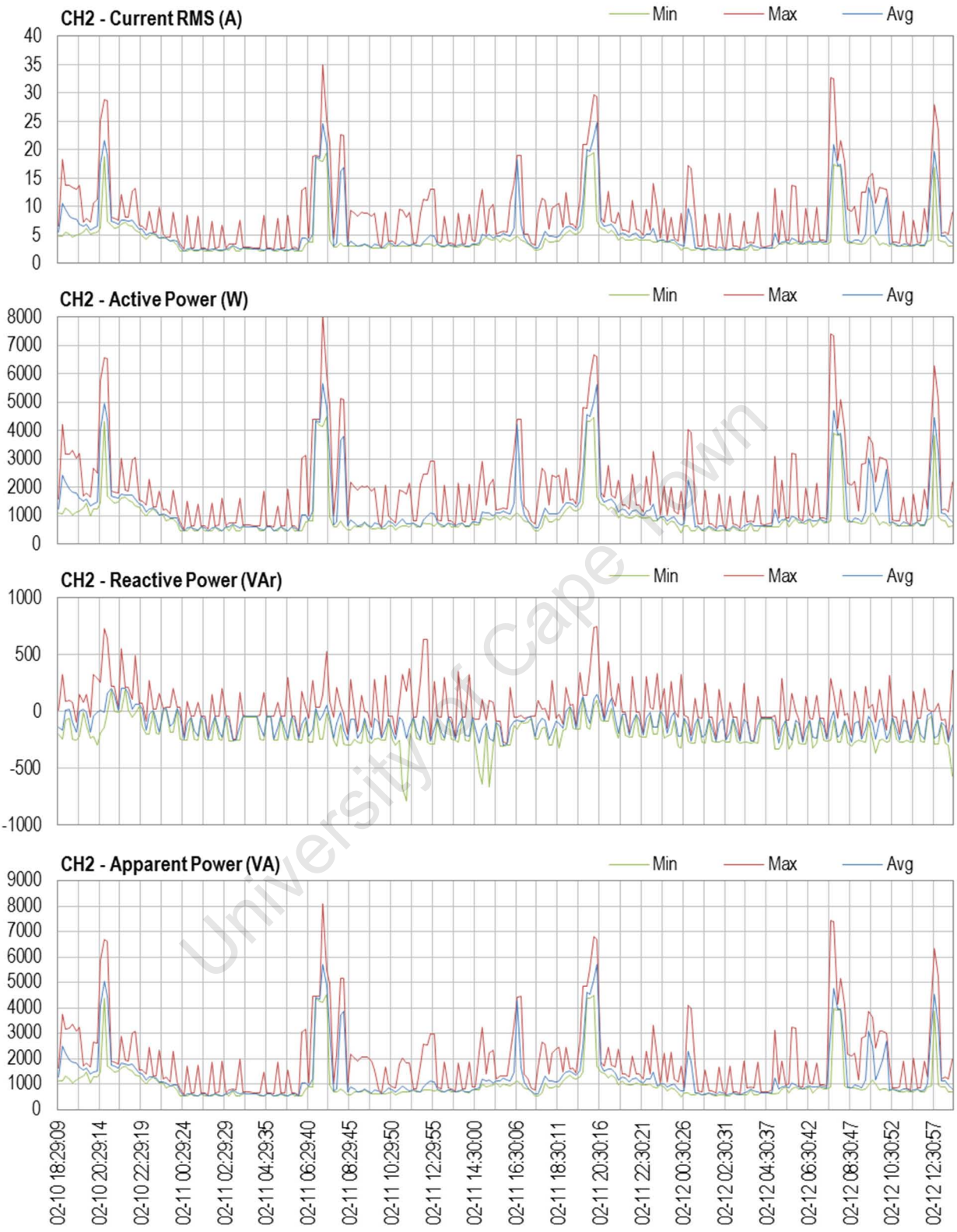
10 min – Voltage Channels (L1, L2, L3) and Frequency



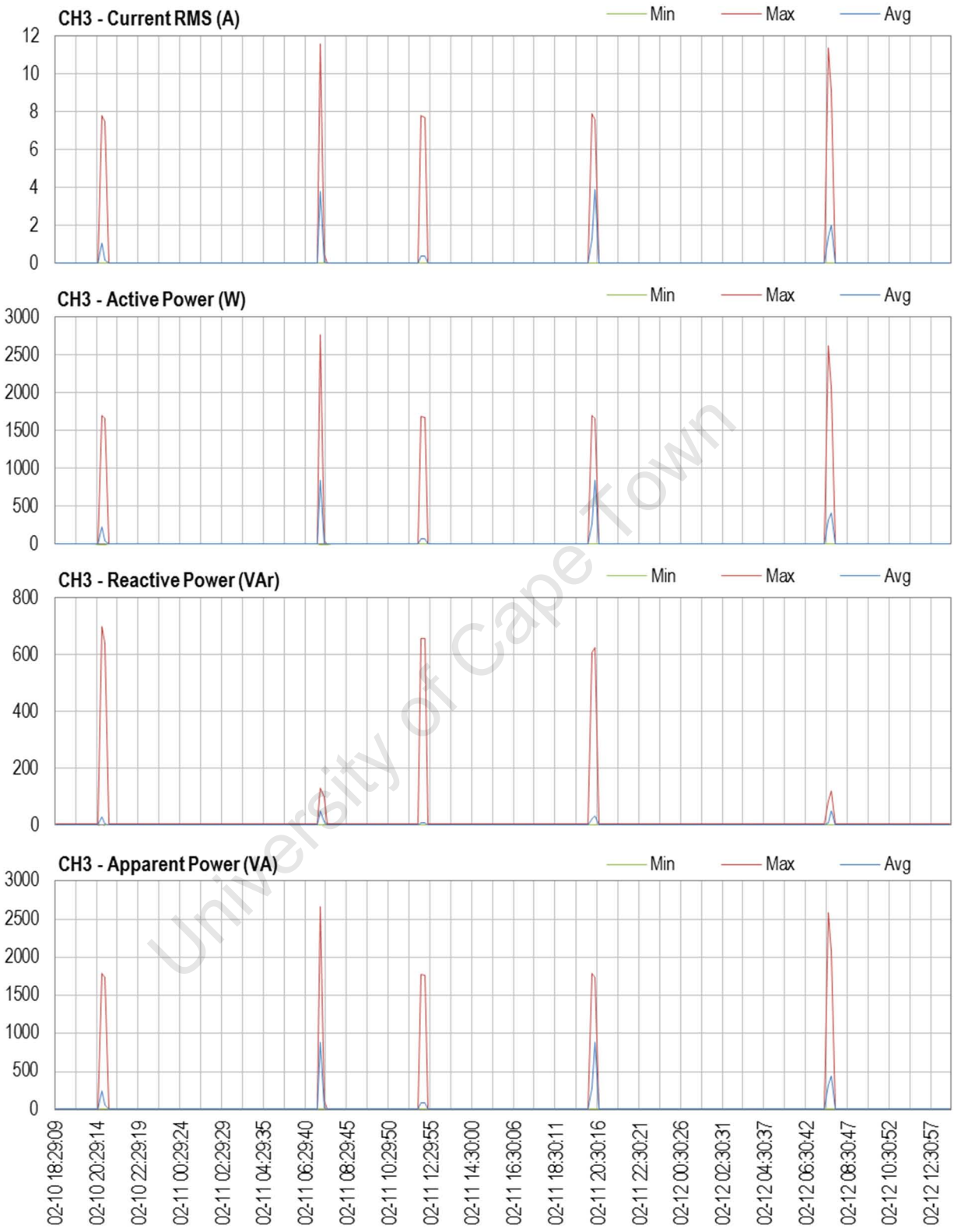
10 min - Channel 1 – Main Supply (106A_{RMS} CT range)



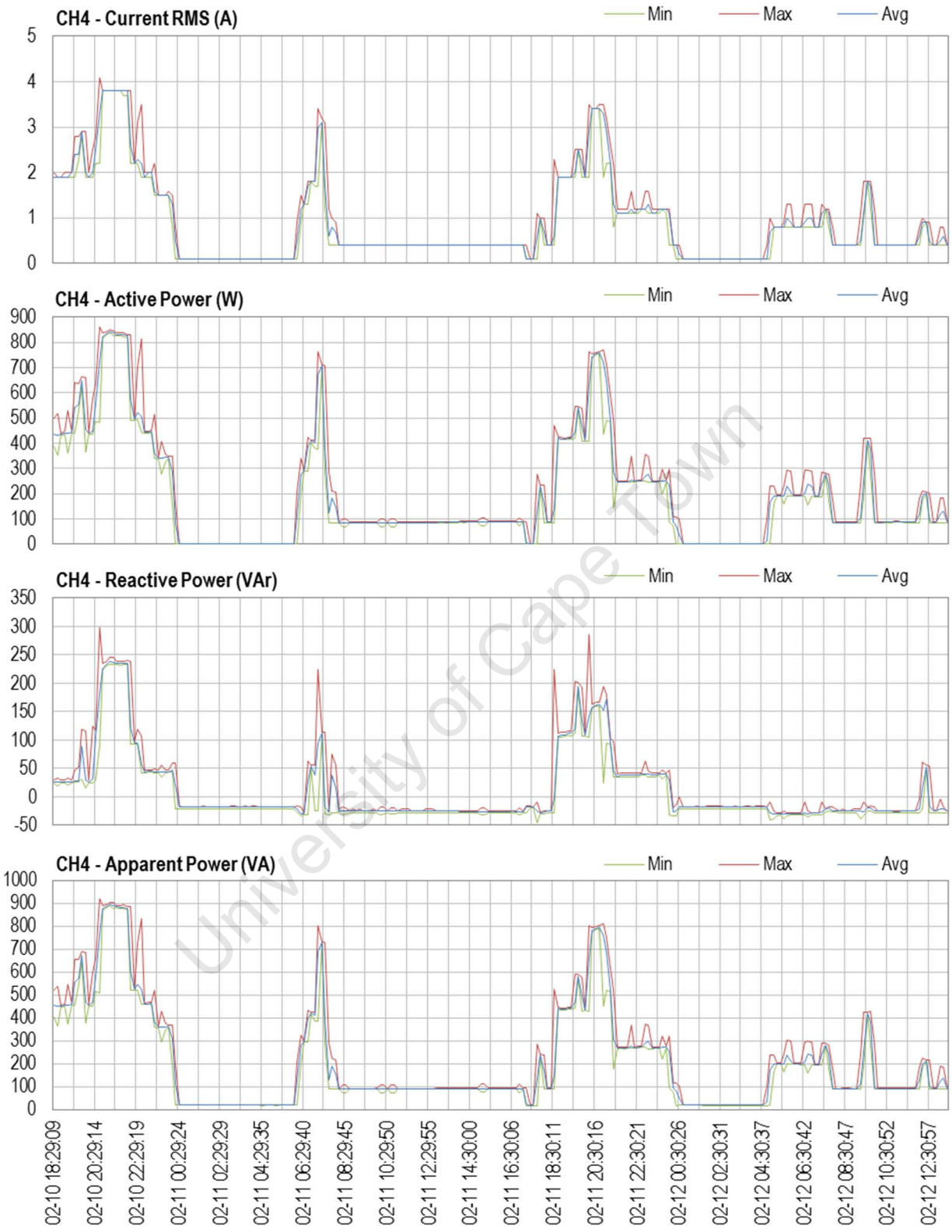
10 min - Channel 2 – Main Supply (53A_{RMS} CT range)



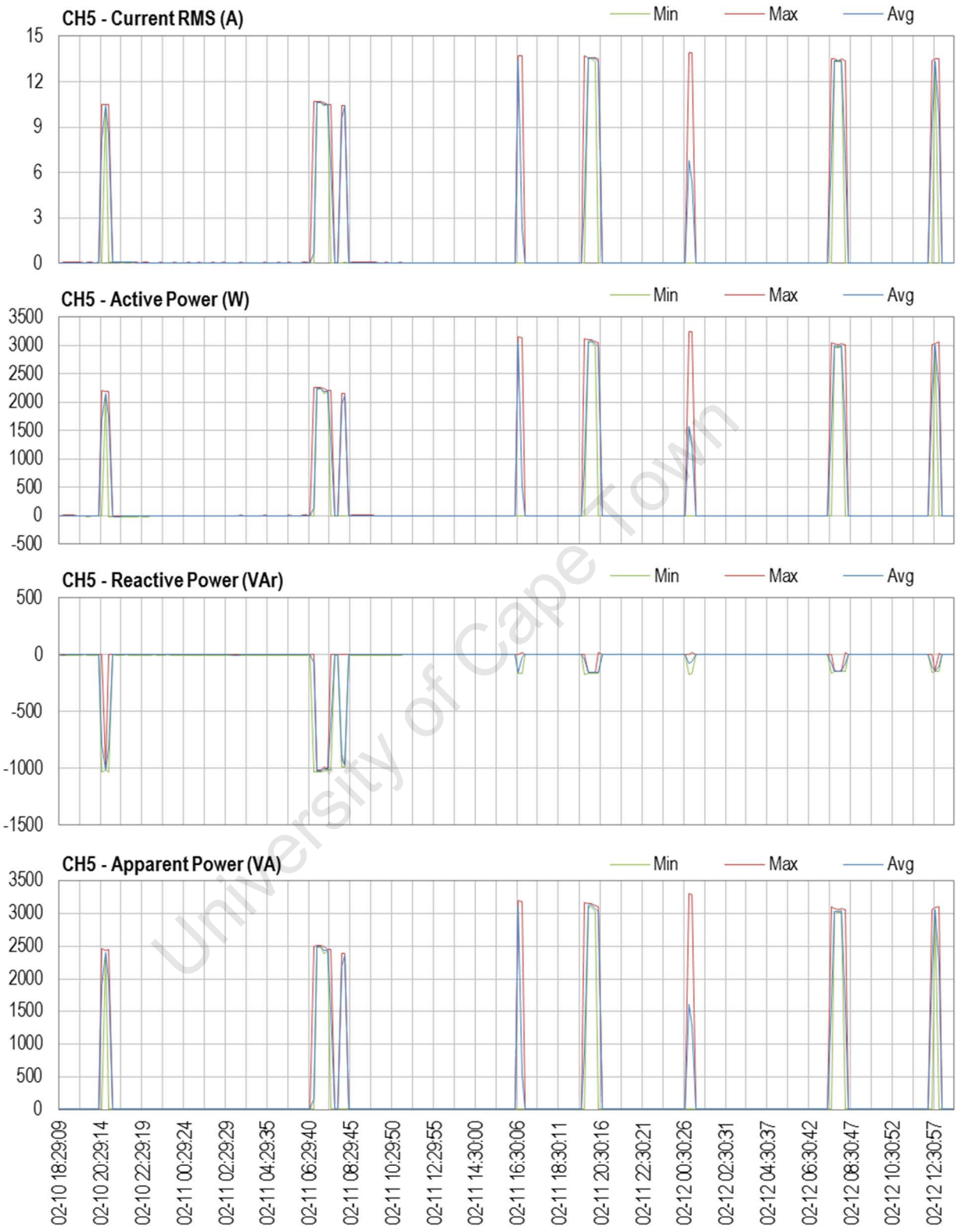
10 min - Channel 3 – Oven Circuit (53A_{RMS} CT range)



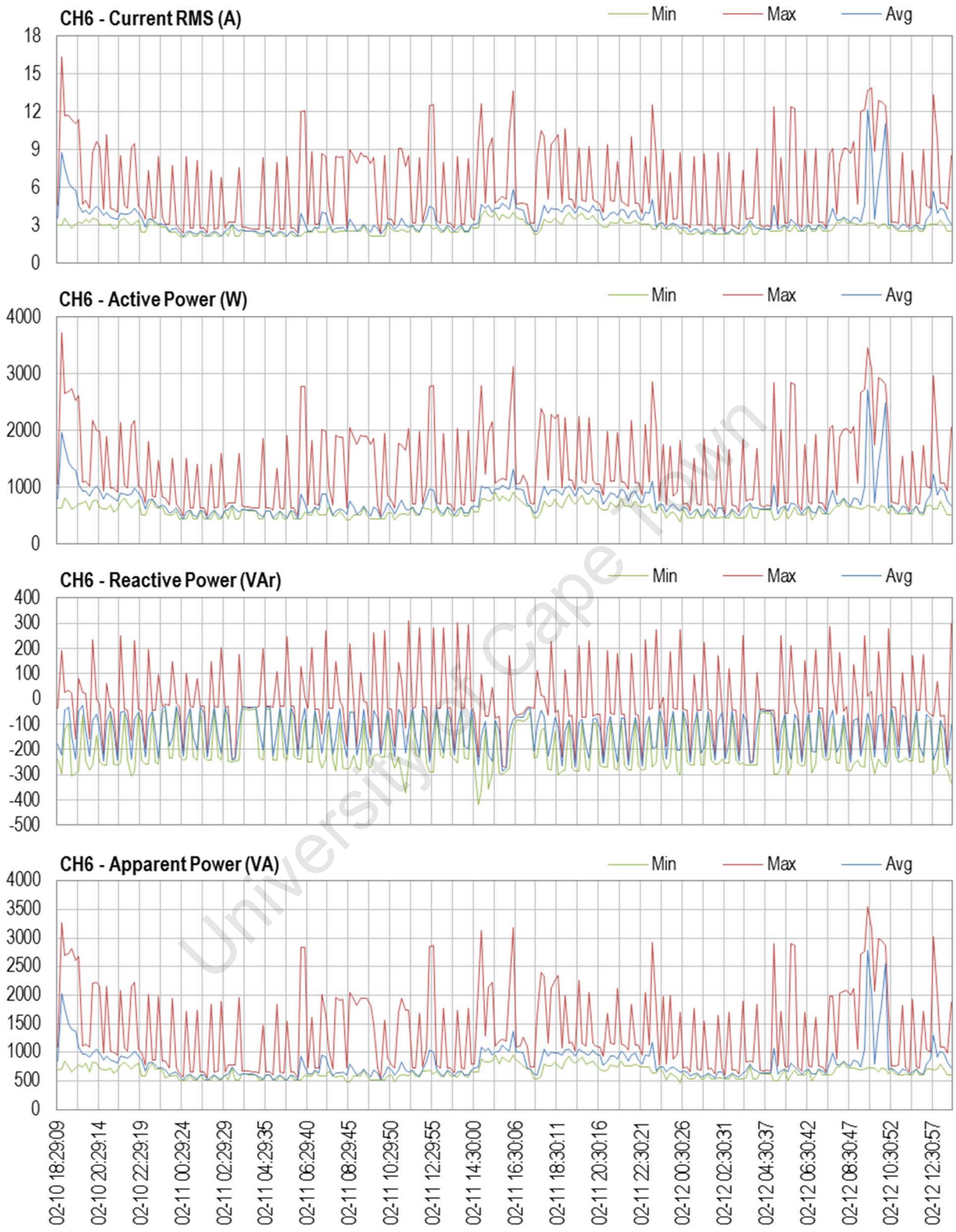
10 min - Channel 4 – Lights Circuit (53A_{RMS} CT range)



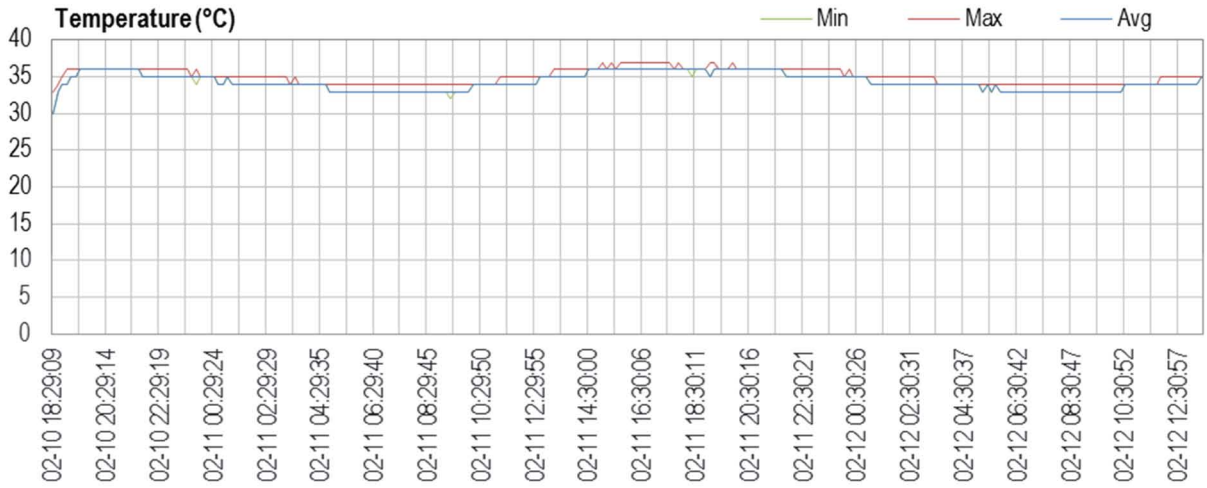
10 min - Channel 5 – Geyser Circuit (53A_{RMS} CT range)



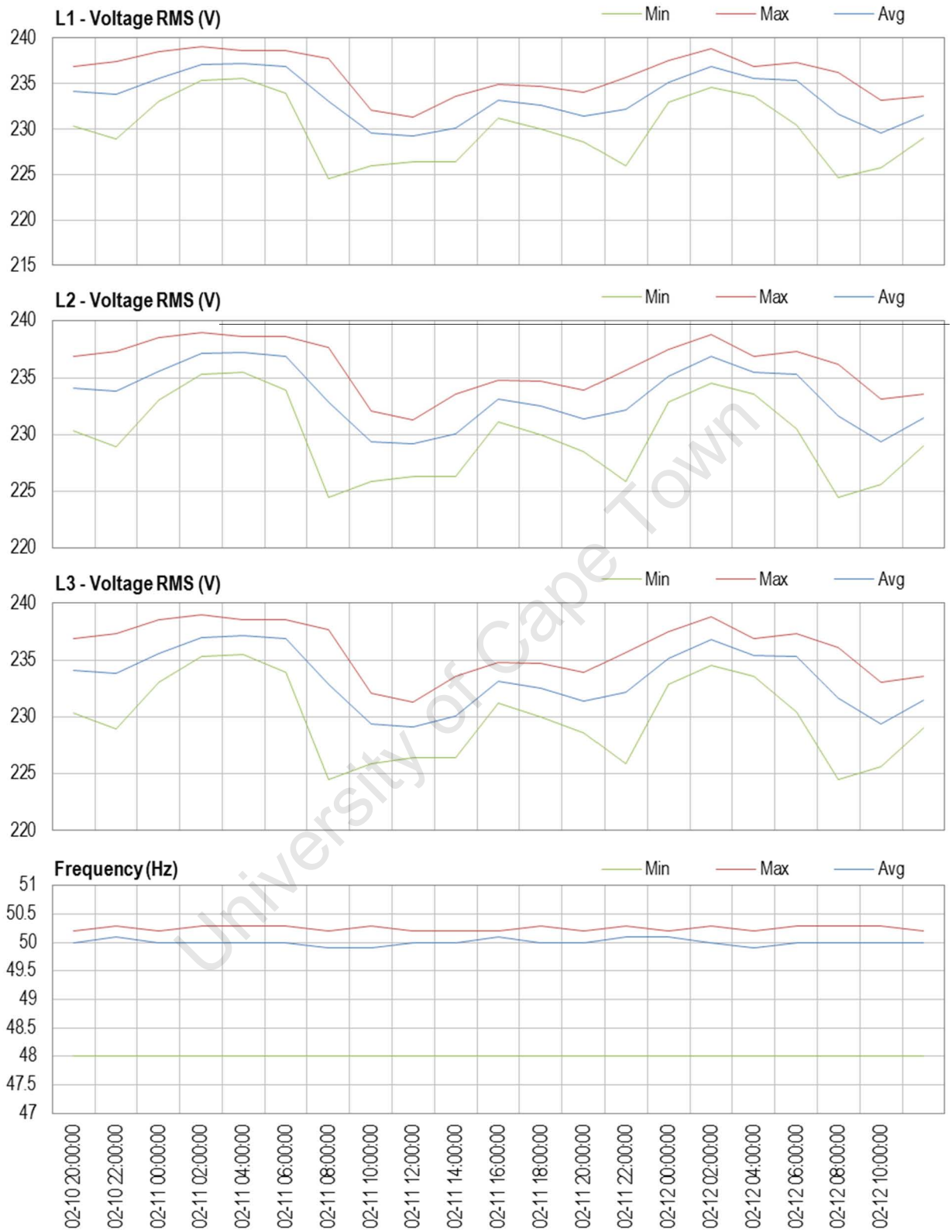
10 min - Channel 6 – Plugs Circuit (53A_{RMS} CT range)



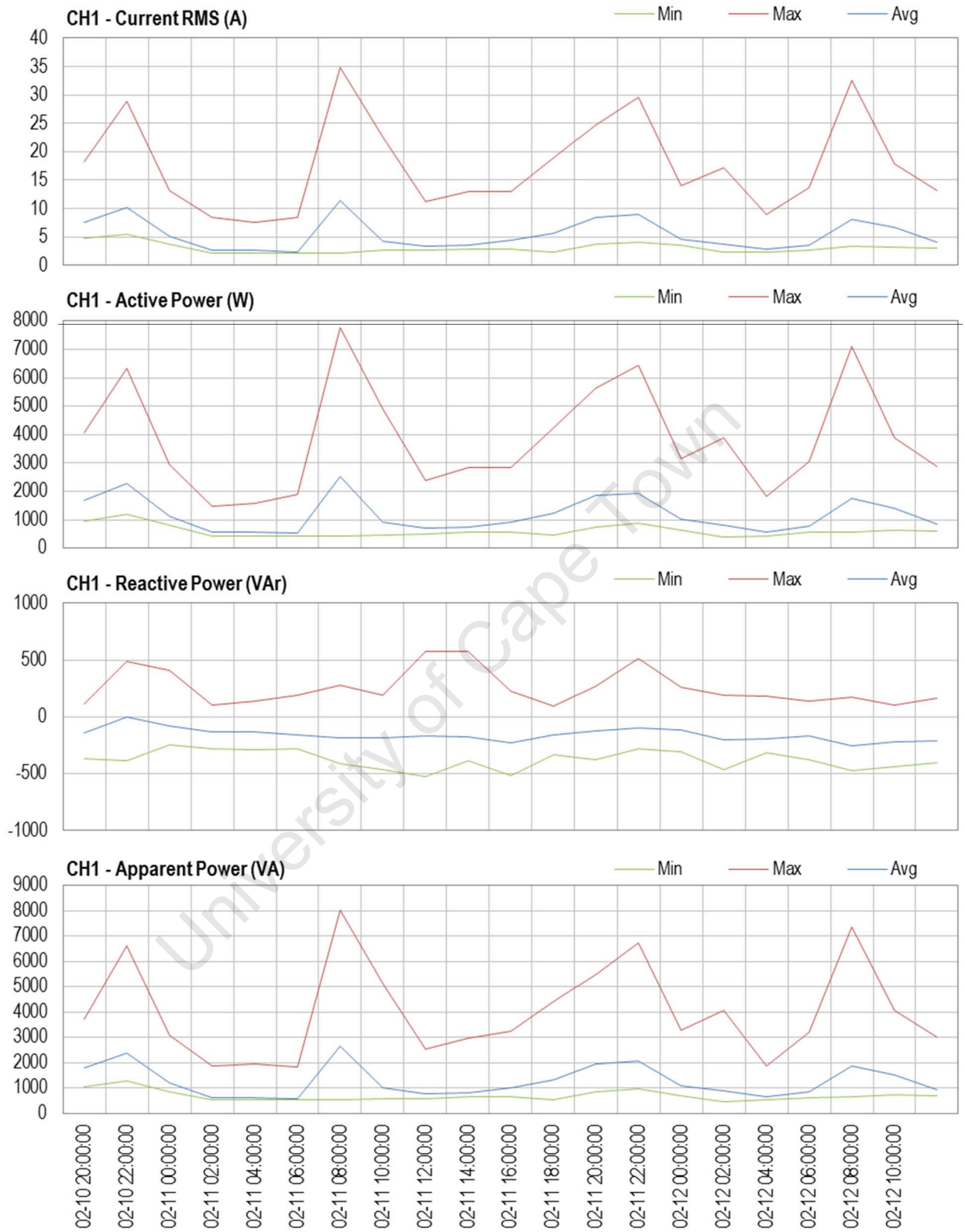
10 min - Temperature



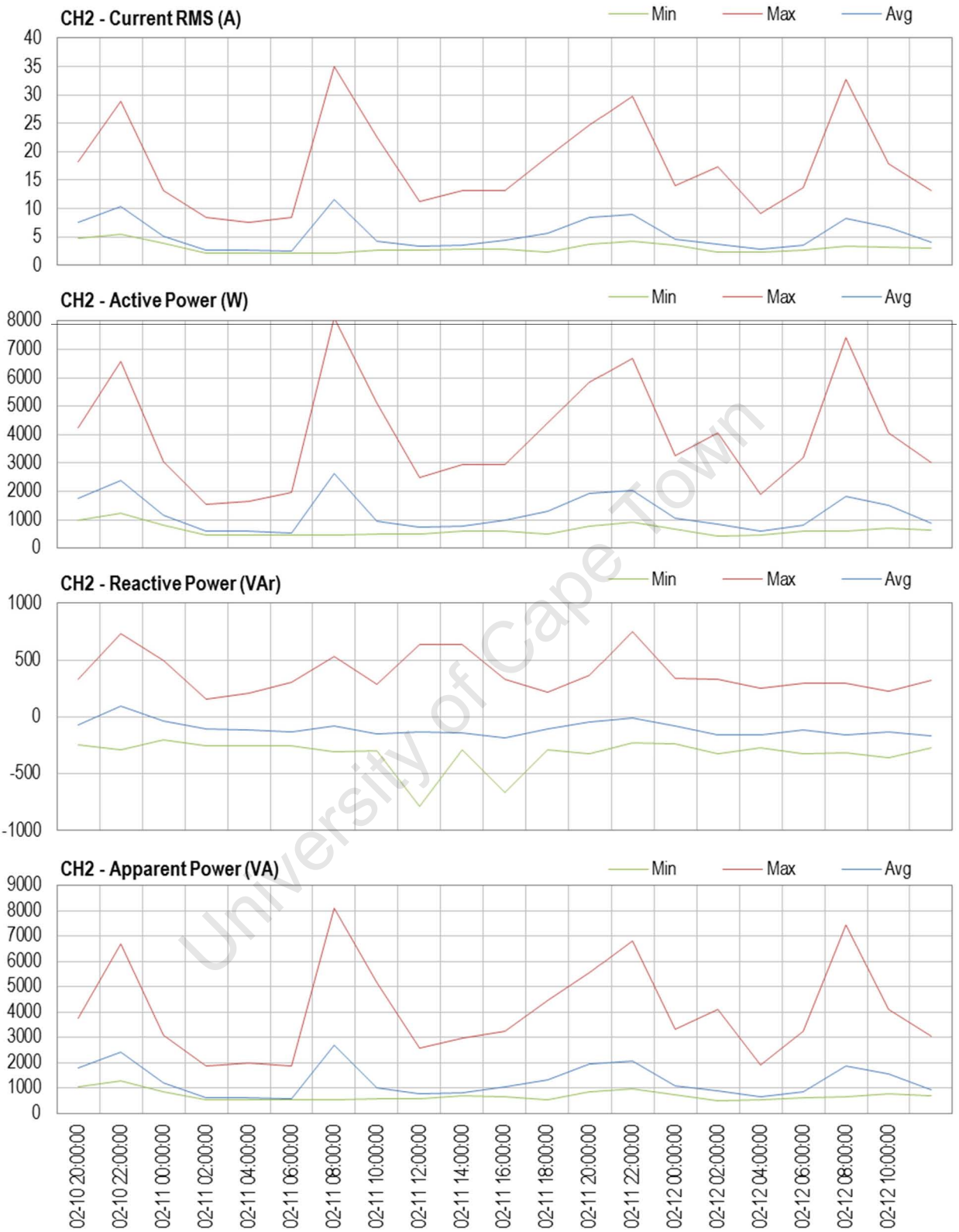
2 hrs – Voltage Channels (L1, L2, L3) and Frequency



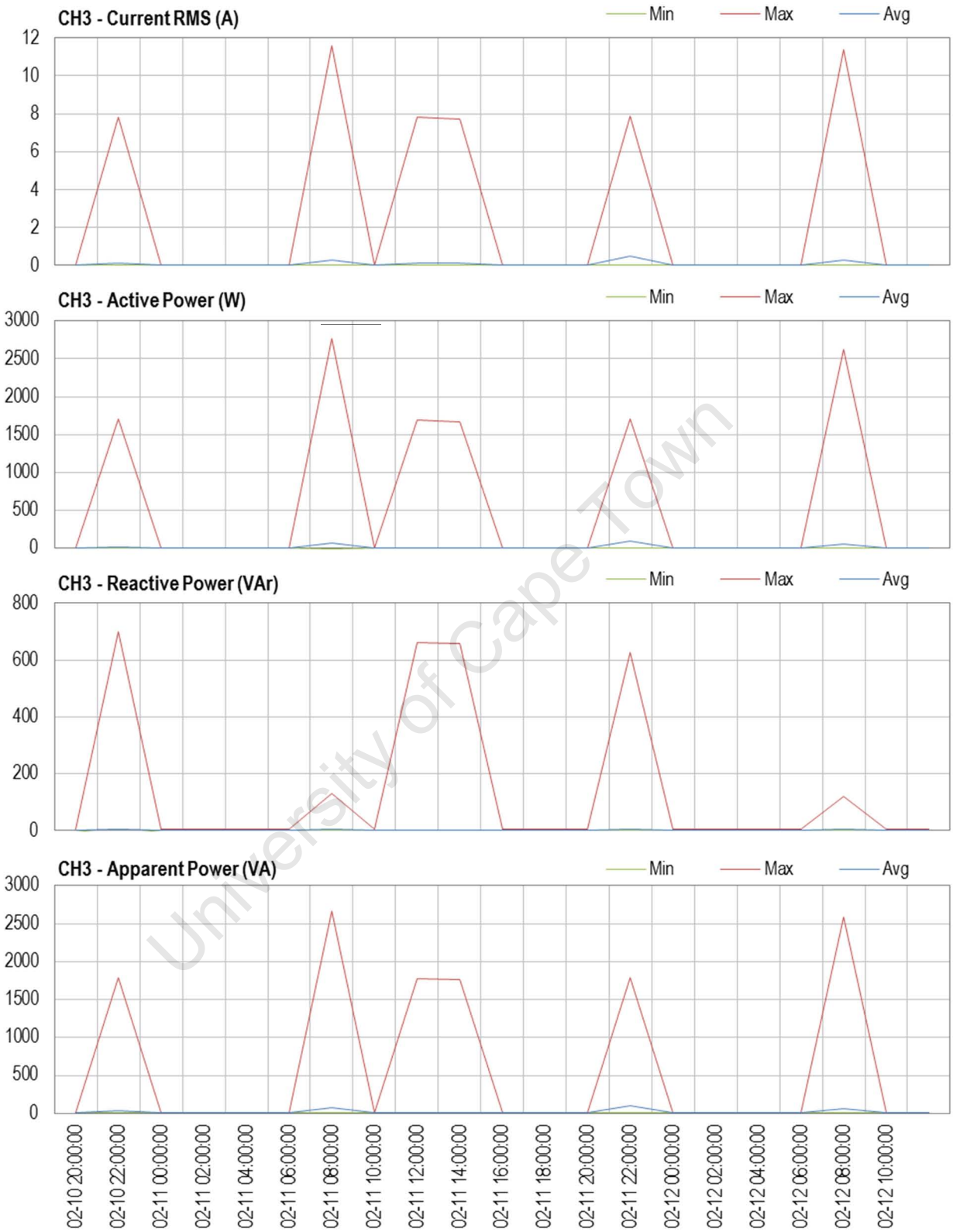
2 hrs - Channel 1 – Main Supply (106A_{RMS} CT range)



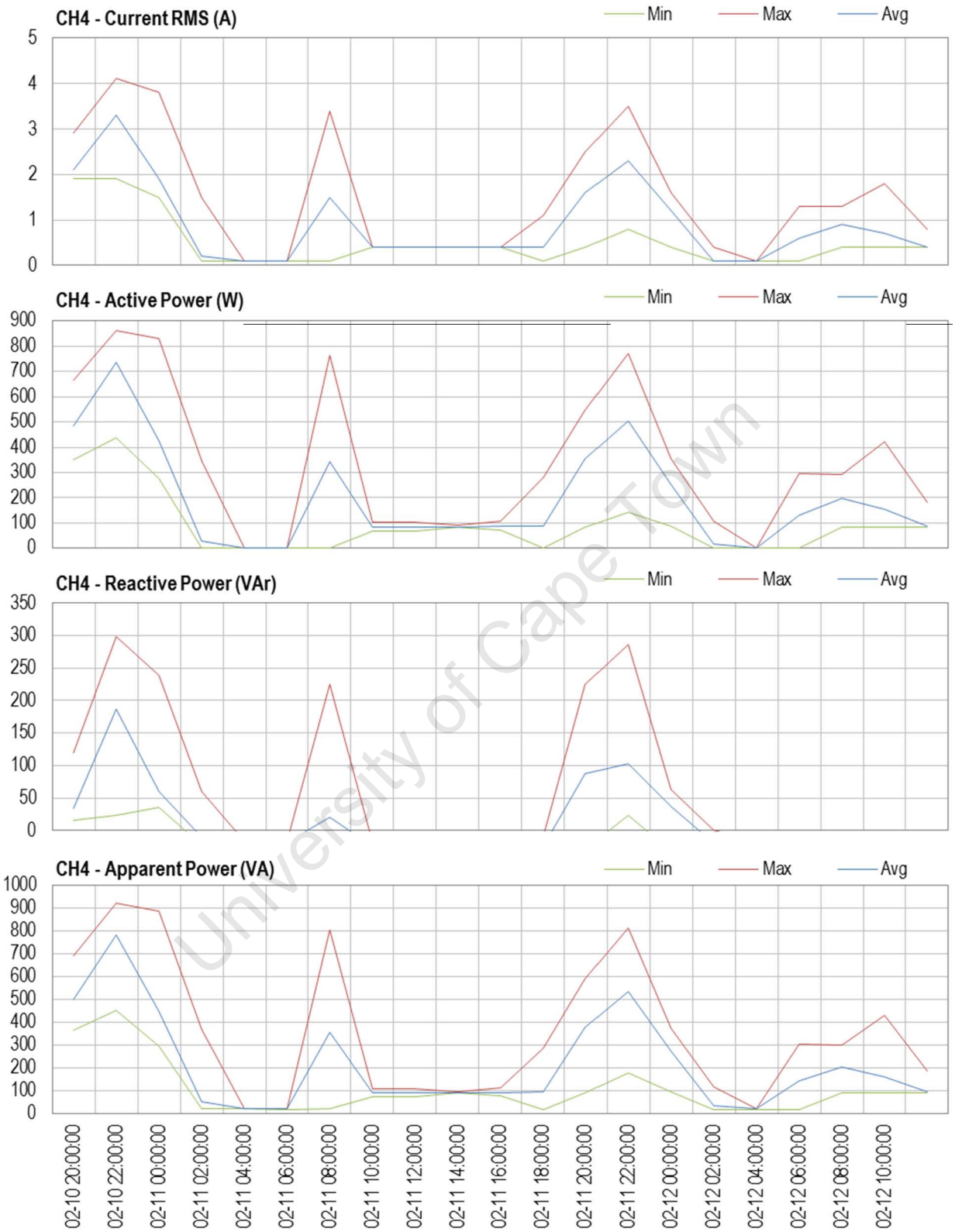
2 hrs - Channel 2 – Main Supply (53A_{RMS} CT range)



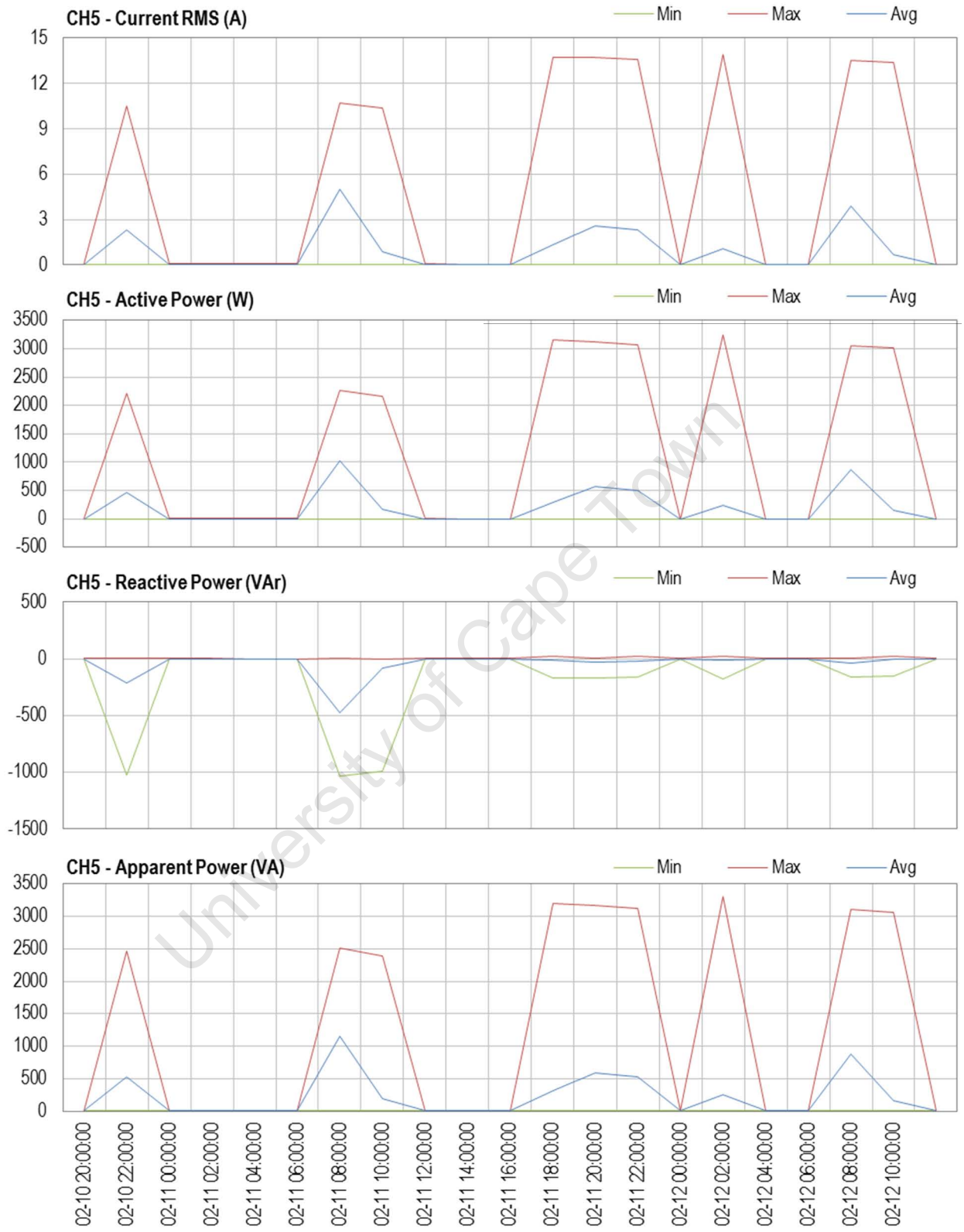
2 hrs - Channel 3 – Oven Circuit ($53A_{RMS}$ CT range)



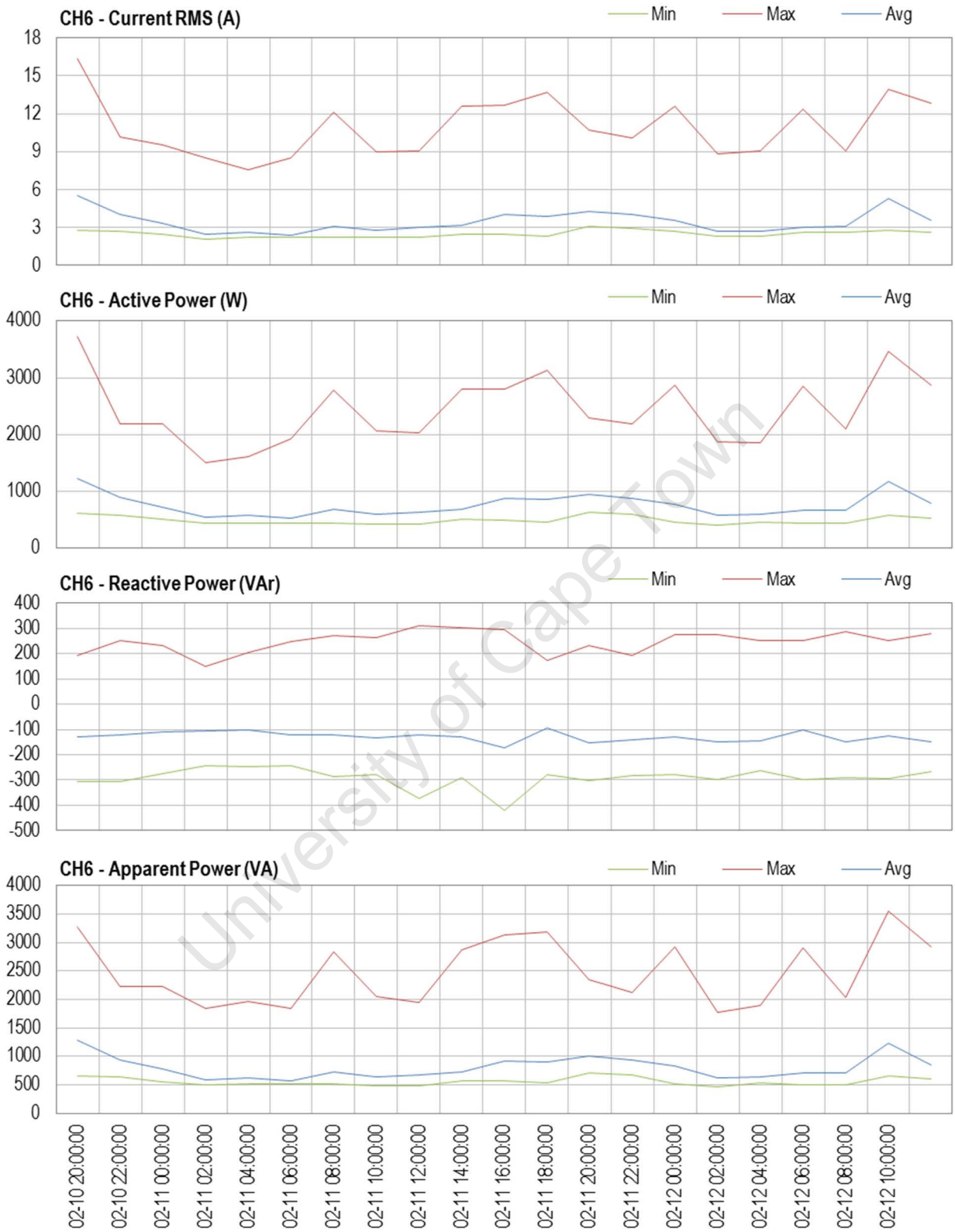
2 hrs - Channel 4 – Lights Circuit (53A_{RMS} CT range)



2 hrs - Channel 5 – Geyser Circuit (53A_{RMS} CT range)



2 hrs - Channel 6 – Plugs Circuit (53A_{RMS} CT range)



2 hrs - Temperature