University Of Auckland

Solar Monitoring Panel

BTech 451 Final Report

Author:
David Armstrong
5636534
darm230

Supervisor:
Dr. Ulrich Speidel

October 28, 2014

Abstract

This final report contains information about all my project-related work on my BTech 451 project throughout the year until Tuesday 28th October 2014. This BTech451 project is with Vector Limited. The project involves producing a solution to answer a research question. This research question is based on an optimisation challenge that the global solar industry is facing. To help answer this research question, I produce a solution that consists of an intelligent load switching device that can switch a load on in the house to consume power when it is in excess, with the goal of providing additional value to the customer. The hardware component of the solution consists of a Raspberry Pi connected to switches that can turn loads inside a house on or off. The software component of the solution involves a Java program running on the Raspberry Pi that is capable of making intelligent decisions based on the state and recent history of a solar panel system. The completed solution successfully switches loads on when conditions are right for providing additional value, and off in other cases. This completed solution helps answer the research question. I find that this solution can bring additional value to a customer who has a solar panel system installed with storage, but only if the load(s) being switched on use power in an efficient and non-wasteful way.

Contents

Chapter 1

Project Introduction

The BTech (IT) degree is a four year degree offered by the University of Auckland. The courses that are taken as part of this degree are mostly Computer Science and Information Systems papers but include various others. As part of this degree, there is a year-long project in the final year of the degree. This project is known as the BTech 451 project and is worth 45 points, or 3 papers. This project usually involves an industry sponsor that specifies a research question for the student to work on and answer.
My BTech 451 project is formally known as the "Solar Monitoring Panel" and is sponsored by the company Vector Limited. This chapter gives an introduction to Vector, the background behind my project and an overview of my project.

1.1 The Company

Vector Limited [1] is an Auckland based multi-network infrastructure company. Vector was formed in 1999 from Mercury Energy when new government regulations were introduced, splitting up electricity distribution and electricity generation businesses. Electricity distribution is Vector's primary role. They own and operate the electricity distribution network in the greater-Auckland area and they are also the largest power distribution company in the country, however they don't produce the power themselves. Vector also operates a natural gas network, they supply gas to over 150,000 customers across the North Island of New Zealand, using their 9,300 kilometre-long gas network. Vector owns their own fibre-optic telecommunication networks in Auckland and Wellington. They also have a nation-wide fibre network that has access points in other major cities, such as Hamilton, Tauranga and Christchurch.
Recently Vector, like other distribution networks, has begun to investigate an optimisation challenge that is based on over-supply with solar panel systems. The particular solar panel

system that Vector is allowing me to work with for this project consists of a collection of components that are installed in a house or business. The system allows power to be generated from the installed solar panels on a building's roof. This power can be directly used by the building, or stored in an onsite battery for future use. The battery can be used at the same time as power is being produced by solar panels, or later during a time when no solar power is being produced. The home/business is still connected to the power grid like normal buildings, and whatever power the system requires that the solar panel system is not providing on its own will be brought in from the grid like normal. Excess power produced by a system can be exported back into the grid. This occurs when the battery is fully charged and the system is producing more power than the home/business requires. The owner of the home/business is reimbursed for the power exported, however this rate is sometimes less than what it would cost to bring the same amount of power in from the grid. There is also an HTTPS-based API for interacting with each solar panel system.

1.2 The Research Question

A current challenge that the global solar industry faces is optimising the use of solar panel systems to provide customers with the maximum value that the system can offer. The research question for this project is "Can the strategic use of excess power provide additional value to the customer?"
As mentioned before, excess power generated by a system is exported back into the grid, provided the battery is fully charged and the solar panels are producing more power than the house is consuming. The rate that users are reimbursed for excess power export varies across electricity retailers [2]. While in principle this is the desired behaviour of the system, there are situations where there is optimisation potential. An example for such optimisation potential for the customer is when their system exports excess power during the day, but later buys in power from the grid to power a power-hungry appliance. It would be in the customer's interest and more efficient to use the appliance when the excess power is available. Another example of optimisation potential is when excess power is exported to the grid while the export reimbursement rate is low. Depending on how much power has been exported into the grid, the rate that a customer is reimbursed for exporting power can vary. It would be a benefit to the customer to consume excess power with a power-hungry device rather than exporting to the grid when the export rate is low. Each solar panel system needs to make effective use of the excess power that it has generated, however many systems have no control over loads in the house. It is also too much work for the customer to monitor the status of a system to find out when it would be a good time to use the excess power.

1.3 The Solution

To explore and answer this research question, Vector would like me to look for potential added benefits that could arise from having a home/business strategically consume excess power in situations instead of exporting it to the grid. One way of effectively using excess power would be to have an appliance like a hot-water pre-heater run automatically when power is in excess. Pre-heating the water when power is in excess will mean that the normal hot-water heater won't have to work as hard later as the water that enters the hot-water heater will already be hotter than normal. If this device can run at times when the export reimbursement rate is low then this will provide additional value to the customer as this would be more cost effective than exporting power at a low rate and later buying power at a high rate.
As part of this project, my job is to explore and answer the research question. To answer this research question I will produce a solution that will enable an optional power-hungry appliance in the house to be turned on to consume the excess power in a useful and efficient way. Vector wants the solution to be in the form of a device that runs locally in the home/business. The device will contain a program that will look at the current power state of a solar panel system, recent power history of the system, and make a decision to turn an appliance/load on or off accordingly.
This project focuses on a specific case where the user has a power retailer plan which reimburses at a higher rate when little power has been exported, but reimburses at a low rate when lots of power has been exported. This threshold will be known as the export threshold. If the user is paid a constant low rate for any power exported, then this export threshold would be 0.

1.4 People Involved

There are several people involved with this project, from either Vector or The University of Auckland. Here are some of the key people that have been involved with this project.

- Academic supervisor: Ulrich Speidel
  My academic supervisor for this project is Ulrich Speidel. He has been assisting me and supporting me throughout this project. I have kept in regular contact with Ulrich during the project.
- BTech Coordinator: Sathiamoorthy Manoharan (Mano)
  Mano is the BTech (IT) Coordinator and is the one who manages the BTech 451 project course. He was the one who assigned me this project.

- Previous Industry supervisor: Anthony Thornton
  Anthony Thornton was my initial Industry supervisor, however he has recently left Vector.

- Current Industry supervisor: Steve Muscroft-Taylor
  After Anthony Thornton left Vector, Steve Muscroft-Taylor took his place and became my new Industry supervisor. He was able to attend my mid-year seminar.

1.5 Report Overview

The following chapter looks at literature I found that relates to this project. After this, the next chapter looks at research I did on understanding the particular solar panel system that I worked with during this project. Following this, the next chapter looks at the technology that I researched. This technology research includes the type of physical device to be used during this project. After this initial research, the following chapter looks at hardware and software design decisions I made about the solution. The next chapter looks at different decision-making algorithms that I had developed. I then discuss at the actual implementation of both the hardware and the software of the solution in the follow chapter. Next I discuss some of the challenges that I faced during this project and describe how I solved them. The next chapter, the evaluation chapter, is where I evaluate the solution and answer the research question. I then discuss some possible future work and ways that this project can be continued. After this, there is the conclusion chapter which sums up the project. I then finally conclude the report with an acknowledgement section and bibliography.

Chapter 2

Related Work

This chapter looks at literature that relates to my project.

In 2014, Vulic et al.[3] looked at using excess solar power to chill water as an efficient way of dealing with excess power. Instead of water being chilled overnight as usual, they looked at using power during peak solar-generation times to chill large tanks of water. They found this was an efficient way of dealing with excess power as the water was chilled while power was in excess, and there was less power needed to chill water at later stages during the day. This is related to my work as an example of a load that I look at storing excess power with is a hot water pre-heater.

In 2011, Chhabra et al.[4] investigated storing excess solar energy to even out/smooth the use of solar-generated power. This deliberate storing of excess power for later use is known as time shifting. They found that time shifting can bring financial benefits if the battery within a building is used at the right times. This work is similar to mine as some of the loads used with my device could be used for time-shifting power.

In 2011, Matallanas et al.[5] developed a control system for Demand-Side Management (DSM) of a system that is similar to the one that I'm working with as part of this project. This control system maximises use of solar-generated power and enhances local energy performance by scheduling tasks demanded by the user at appropriate times. This work is similar to my own as I am also trying to maximise the use of solar-generated power.

In 2012, Giorgio et al.[6] propose a design for more efficient management of electrical energy in a domestic environment. This paper is focused around smart home controllers. The results from this paper provide a proof of concepts that consumers will get benefits from the more efficient use of energy management. Although this paper isn't based around solar excess use, some of the general efficiency concepts can relate to my work.

In 2014, Giorgio et al.[7] presented an approach for local energy management that automated Demand-Side Management (DSM) programs. They dealt with houses that have a solar panel system similar to the one I'm working with, as well as smart appliances and electric cars. They didn't find any major results during this paper but instead proposed a control framework. This work is similar to mine as it deals with time shifting some loads for more efficient power use.

Out of all the related work I could find, I found the most similar and relevant to be the paper by Vulic et al.[3]. This was due to the fact they were using excess solar-generated power to chill water so less power had to be taken in from the grid to chill it later. This is similar to one of the examples that I use in this report. My example is using a hot water pre-heater to heat water while excess power is being generated by solar panels. This example can save power from being brought in later to heat hot water as it is already heated.

Chapter 3

Solar Panel System

The solar panel system that I'm working with is a complicated system that coexists with existing standard power components of a building. The system contains both hardware and software components. This chapter looks at the system and its components that are installed in and outside the building.

3.1 Hardware

Several hardware components make up the system. They are mostly installed in a fridge-sized cabinet that is located on the outside wall of the building. These components interact with the existing components of the house. All the components are connected to an onsite computer that connects to the internet via the pre-existing home/business internet connection. Figure 3.1 refers to a logical diagram that displays the major components of the system. As this is a logical diagram, the placement of components in this diagram are not accurate or to scale, but it gives an idea of how they work together.

3.1.1 Solar Panel Array

Solar panels are an essential part of the system. An array of solar panels are installed onto the roof of a house. They are placed either just on one side of the roof, or both sides depending on how many the resident wants. If they are installed on just one side of the roof then they are placed in the optimal position to generate the most power from the sun. Power produced by the solar panels is in Direct Current (DC). Power from the panels are fed into the Maximum Power Point Tracker (MPPT).

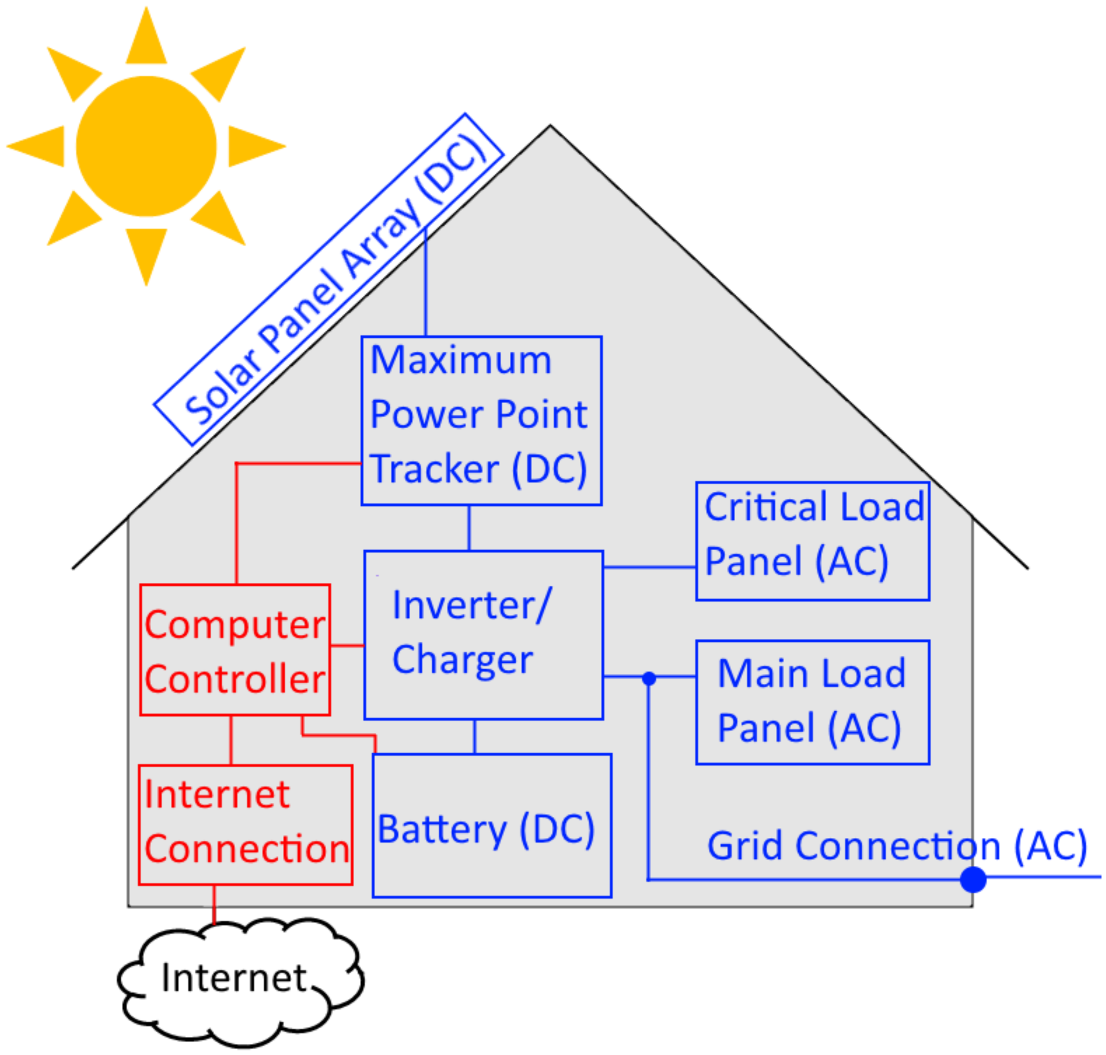


FIGURE 3.1: Solar Panel System - Logical Diagram

3.1.2 Maximum Power Point Tracker (MPPT)

The Maximum Power Point Tracker (MPPT) is a device that aims to optimise the power being produced by the solar panels at any given point in time. The MPPT does this by varying the resistance of a circuit to maximise the Current-Voltage product, and therefore get the maximum amount of power available from the solar panels. The optimally generated power is fed from the MPPT into the Inverter/Charger

3.1.3 Inverter/Charger

The Inverter/Charger is at the heart of the system. Its main job is to convert between DC and AC power when power flows exist between components that use a mix of AC and

DC power. If a power flow between components is in the same form the whole time – e.g. flowing from one AC component to another AC component, it will just pass through the Inverter/Charger without being converted. DC power is used by the MPPT (originating from the solar panels) and the battery. AC power is used by the Critical Load Panel (critical appliances in the house), Main Load Panel (other appliances in the house) and the grid connection.

3.1.4 Battery

The battery stores power for later use. It uses DC power. The power used to charge the battery can come from solar panels (via MPPT), or the grid. Having a chargeable battery means that power can be stored while it is sunny, and used later when no power is produced by solar panels. An example of this would be at night time or while it is very cloudy. Batteries can also be used in the event of a power cut, so essential devices can still run.

3.1.5 Critical Load Panel (CLP)

The Critical Load Panel (CLP) has the "critical" appliances and devices in the house attached to it. It uses AC power. The CLP is connected directly to the Inverter/Charger. In the event of a power cut, the CLP can still be provided with power from the battery (via Inverter/Charger), or if the power cut is during day time it can also be provided with power from the solar panels (via MPPT and Inverter/Charger). Power from the grid can flow to the CLP just like with standard houses (not using the solar panel system) via the Inverter/Charger.

3.1.6 Main Load Panel (MLP)

The Main Load Panel (MLP) has the remaining (lower priority) appliances in the house attached to it. It uses AC power. The MLP either receives its power from Inverter Charger (sourced from the solar panels or the battery) or from the grid (like with standard homes).

3.1.7 Computer Controller

The computer controller doesn't have any large amounts of power flowing through it, but instead has data connections to components in the system. It is also connected to the internet via the home internet connection. The computer looks at several bits of

information from around the system, including the battery charge levels, power being produced by the solar panels, and the flow of power between each component. Based on the information of the system and the current power plan settings, it makes decisions about whether the battery should be charged, left alone, or if power from the battery should be used to power the CLP/MLP/grid.
This computer connects to the internet using the home/business internet connection. This connection allows each system to be managed remotely. The current power plans and rules being followed by the system can be modified. An example of remote management would be before an expected storm, the system could be told to charge the battery and keep it charged in case the storm causes the local power to go out.

3.1.8 Measuring Point

The measuring point isn't a component as such, it is just the point where the Inverter/Charger, MLP and the grid are all connected to each other. If the MLP is requiring more power than the Inverter/Charger is providing (if any), then the remaining required power will be brought in from the grid. Power from the grid is also required if the Inverter/Charger is requiring power from the MP. In this case the grid will power the MLP, and supply some power to the Inverter/Charger. In the case that the Inverter/Charger is sending more power to the MP than the MLP needs, power will be sent back into the grid. In this case the owner of the house will be reimbursed for the power sent back into the grid.

3.2 Software - The API

There exists an HTTPS-based API for communicating with the solar panel systems. It uses a RESTful architecture. By writing custom POST, GET and DELETE HTTP requests, a system can be communicated with remotely. The API doesn't connect to a system directly, but instead connects via the system provider of the solar panel system.
The primary function of the API that I will be using during this project will be to query the state of a solar panel system to get information about battery charge levels and important power flows. These power flows include:

- The amount of power being generated by the solar panels (PV)
- The current power rate that the battery is charging/discharging
- How much power is being taken in from (or exported to) the grid

- The amount of power that the appliances in the house are using (measuring the CLP and MLP)
- And several other flows

The API also allows power plans to be set and viewed. These plans include rules such as how quickly the battery should be charged, and at what charge level power should be exported to the grid.
A new version of the API was introduced last semester. This brought new features and is an overall benefit to my project. However it broke some earlier code and I needed to rewrite it. This caused a minor setback.

Chapter 4

Technology Research

There are several design decisions to be made in this project. Vector wasn't specific about what technologies or devices I should use as long as it does what they want it to, does it efficiently, and is appropriate. This chapter covers some of the choices I made while I was researching and developing.

4.1 Device

The first and main choice was to decide on what hardware I would run my software on. Vector had said it needs to be a device that runs locally in the house/business. The device needs to meet certain requirements. The device has certain requirements, these include:

- Being relatively cheap
- Supporting a programming language that has an HTTPS library
- Has enough processing power to make decisions
- Has an electrical interface to send a control signal to a relay

4.1.1 Arduino

The initial idea as suggested by Vector was to use an Arduino as the device to implement the decision making program on. Arduino is an open-source single board microcontroller [8]. The processor on an Arduino consists of either an 8-bit AVR CPU, or 32-bit ARM CPU. Arduino models vary quite significantly, but are mostly all used for hobby hardware projects with basic computations. Programs on Arduino are written in C or C++.

Although it has the electrical capability of sending a control signal and the computational power to make decisions, it doesn't support any HTTPS libraries. This is the main reason why I couldn't use Arduino as there would be no way to use the API. Without being able to use the API there would be no way to query a solar panel system and make decisions.

4.1.2 Raspberry Pi

After realising Arduino would not be a suitable choice, the next device I looked into was the Raspberry Pi. Raspberry Pi is a credit-card sized single board computer [9]. Raspberry Pi's have a 700MHz ARM CPU and either 256 or 512 MBs of RAM depending on the model. Raspberry Pi's are significantly more powerful than Arduinos. Raspberry Pi's can run a full version of Linux that is designed to run on ARM CPUs. Within a Linux distribution, programs can be written on various programming languages, such as Java, Python or C++. Having an operating system running means there is more CPU overhead when compared to Arduino, but it makes software development much easier. Raspberry Pi's have several General-purpose input/output pins that can be used for sending control signals. Raspberry Pi's can connect to the internet/networks using an inbuilt 100Mb/s Ethernet port, as well potentially using a Wi-Fi dongle in on one of its USB ports.
I chose to use Raspberry Pi as the device as it fulfils all the technical criteria, is easily purchasable, and relatively cheap.

4.2 Raspberry Pi Operating System

Once I had chosen Raspberry Pi, I had to decide on the operating system I would install on it. Unlike Arduino which had a basic operating system built into it, Raspberry Pi needs an operating system to run. There were certain requirements for this Operating system, it needs to be:

- Stable
- Easy to install
- Support a programming language capable of HTTPS requests

4.2.1 Raspbian

Raspbian is a custom version of Debian Linux that is designed to run on a Raspberry Pi [10]. It is currently the most popular and supported operating system. Raspbian comes

pre-installed with many programming languages and development environments.
I chose to use Raspbian as it is the most supported operating system, stable, and supports many programming languages.

4.2.2 Pidora

Pidora is a custom version of Fedora Linux that is designed to run on a Raspberry Pi [11]. Like Raspbian, Pidora comes pre-installed with various programming languages.

4.2.3 Arch Linux

There is also a version of Arch Linux that runs on Raspberry Pi's [12]. Arch Linux requires considerably more setup than Raspbian and Pidora, and comes installed with less programming languages.

4.3 Programming languages

Once I had decided on using a Raspberry Pi as the device to run the program, the next decision was what programming language and platform to choose from. The language needs to :

- Support HTTPS queries for the using the solar panel system API
- Be able to run for a long time without crashing or suffering memory leaks
- Be relatively efficient when performing computations
- Have libraries for using the Raspberry Pi's GPIO ports (sending the control signal)

4.3.1 Python

Python [13] is a multi-paradigm programming language that runs on a Virtual Machine. The use of Python is encouraged on Raspberry Pi's as it is pre-installed on the device and many libraries for raspberry pi specific features (e.g. GPIO) are pre-installed too. Python would have been a good choice for the project, but due to my limited experience with it I chose not to use it.

4.3.2 Java

Java [14] is a multi-paradigm programming language that runs on a Virtual Machine. Java is also encouraged for use on Raspberry Pi's, however not as strongly as Python. Java is pre-installed on Raspberry Pi's, however some libraries (e.g. GPIO libraries) need to be installed from external sources. I chose Java over Python for this project as I feel it is functionally-equivalent with Python for the software that I need to create, however I have far more experience and confidence to code with it. One downside when compared to using Python is that I will need to find and reference external libraries (which are easily available) for using the GPIO interface. I feel this downside is more than out-weighed by the benefits of using a language that I am more confident in.

4.3.3 C++

C++ [15] is a multi-paradigm programming language that runs natively on hardware. Like the Python and Java, C++ is supported on raspberry Pi's and it is pre-installed. Like Java, external libraries would need to be found and referenced to use the GPIO ports, or code could be written in a C++ class to use them without too much effort. Code written in C++ generally runs faster when compared to Python or Java code, but I would need to put more effort into memory management to avoid memory leaks. I chose not to use C++ as I don't have too much experience in it and I would have had to put more effort into writing the program. The speed advantages I would get from writing the program in C++ would be small as my program is not computationally demanding.

4.4 Status Query Result format

Originally with the v1 API, XML was the only format that the retrieved system status data would be in. However with the release of the v2 API, there is now the option of using JSON as the format for the retrieved data to be in, as well XML. I needed to make a decision between using XML or JSON as for retrieving system status data. I needed to pick the format that:

- Uses the least amount of bandwidth
- Is computationally efficient

4.4.1 XML

Extensible Markup Language (XML) is a data encoding language that uses opening and closing tags. An example of data entry returned for how much power being produced by the solar panels would be: <pvWatts>890</pvWatts>. This requires 22 characters. I sampled the returned status in XML format at certain times (both day and night) and I found on average the format used around 1920 characters.

4.4.2 JSON

JavaScript Object Notation (JSON) is also a data encoding language, however it uses name-value pairs instead of tags. JSON is more light-weight than XML and this will reduce data transmitted each query. An example of data entry returned for how much power being produced by the solar panels would be: “pvWatts”:890, this requires 14 characters and represents the same amount of data as in the XML sample for the same value. I sampled the returned status in JSON format at certain times (both day and night) and I found on average the format used around 916 characters. This is less than half the size of XML’s average of XML’s average of 1920 characters.
I chose to use JSON over XML as JSON uses less characters and therefore less internet bandwidth will be consumed. It also has been found to be less computationally intensive to parse JSON when compared to XML [16].

Chapter 5

Solution Design

This chapter looks at the overall design of my solution that will enable me to answer the research question described earlier. It also describes design decisions that have been made during my project. These designs include both hardware and software designs.

5.1 Hardware

Once I had decided on using a Raspberry Pi as the platform for making decisions and sending a control signal, the next step was to design how the Raspberry Pi will be physically installed. It also needed to be decided on how the Raspberry Pi would interact with switching hardware. As this is a software based course, my Academic supervisor, Ulrich Speidel, developed the hardware side of this project. This was due to the fact there was wiring that needed to be done that I didn't know how to do, and wasn't expected to know due to the BTech (IT) degree not covering hardware in enough depth. I also legally couldn't do some of the wiring that was involved.
There were 3 main types of hardware designs that were discussed during this project that specified how the Raspberry Pi and switching hardware would be arranged and connected. We ended up going with the last design, but I have also listed the first two.

5.1.1 Combined Components – Single Switch

The first hardware design was to combine both the Raspberry Pi and the relay together in one box. There would be mains power going into it for both the Raspberry Pi and the relay. There would be a standard mains power socket on the box that an appliance could plug into. There would also be an Ethernet connection for the Raspberry Pi to connect

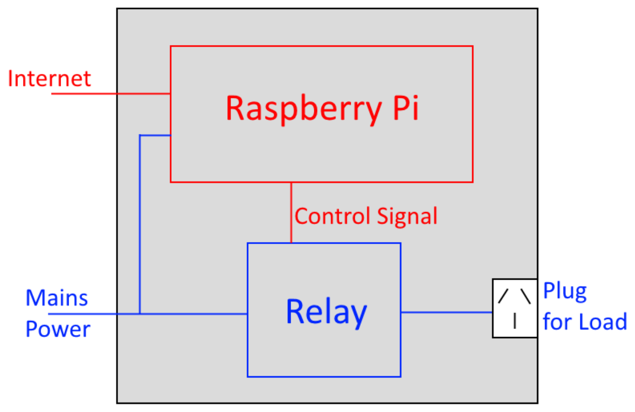


FIGURE 5.1: Combined - Logical Diagram

to the internet.
The advantages of this design include:

- Easier to transport and demonstrate, especially for academic seminars and for presenting to Vector
- Simpler to deploy as there was only 1 box to be placed in the house

The disadvantages of this design include:

- Lack of flexibility as this box (including the Raspberry Pi) had to be placed right next to the desired load
- Potential heat problems due as the relay could have heated up the Raspberry Pi too much
- More dangerous to develop as there was high-voltage circuitry from the solid-start relay right next to the low voltage Raspberry Pi

We chose not to use this design due to the many downsides of this design. Figure 5.1 is a basic logical diagram of this combined design.

5.1.2 Separate Components – Single Switch

The second hardware design was to separate both the Raspberry Pi and relay into separate boxes. The "logic box" would contain the Raspberry Pi. The "relay box" would contain the relay and have a mains power plug embedded into it. The two boxes would be linked

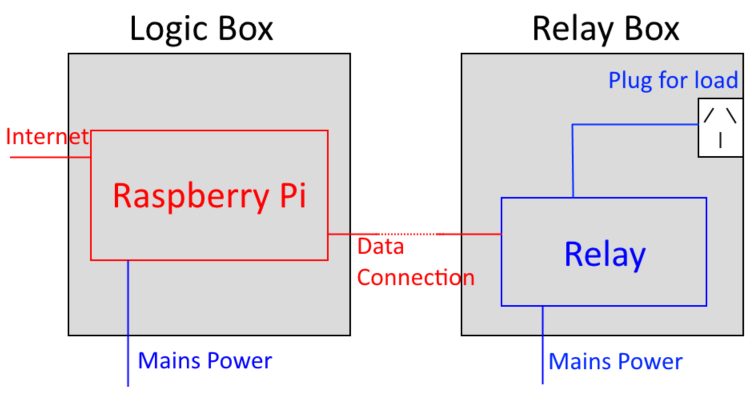


FIGURE 5.2: Separate - Logical Diagram

so the control signal can be sent from the Raspberry Pi to the relay. This link could be either a basic wire, or potentially a type of wireless signal.
The advantages of this design include:

- More flexibility with the arrangement of the two boxes, only the relay box would need to be placed near the load, the logic box could be placed in a different and more easily accessible place.
- No heat or voltage issues as the relay and the PI are physically seperate

The disadvantages of this design include:

- Only one load could be switched at a time
- There could be locations where the relay box is too big to setup

This 2nd design was a better design than the 1st, but there was still room for improvement. We chose not to implement this design as the 3rd design turned out to be much better. Figure 5.2 is a basic logical diagram of a separate design.

5.1.3 Separate Components – Multiple Switches

The 3rd hardware design builds upon the previous “separate components” design, but includes 2 additional forms of switching. This 3rd design has both a logic box and a switch box, also 2 wireless switches. These wireless switches essentially perform the same job as the switch box, but instead they are smaller in size and easier to place around the house due to the wireless connection. To send a wireless signal and communicate with these switches, the logic box would need to have a wireless transmitter build into it. Unlike the relay box, these wireless switches can be bought premade and they are ready to go

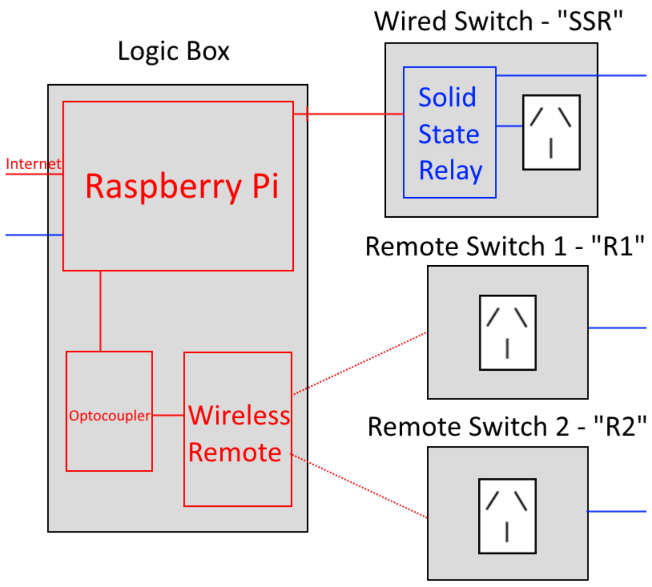


FIGURE 5.3: Separate components, multiple switches - Logical Diagram

out the box, however the wireless transmitter would take some effort to be connected to the Raspberry Pi's GPIO ports. This design requires an optocoupler to be connected between the GPIO pins and the remote control circuity due to voltage differences between the GPIO pins and the remote control.

The advantages of this design included:

- The choice of being able to switch 1-3 different loads at the same time
- The most flexibility with the physical arrangement of the logic box and the switches/relay box

The disadvantages of this design include:

- Harder to demonstrate due to multiple physical components

This 3rd design is the best design out of the 3 for the purposes as it provided the most flexibility for installation and use. Having the option of 3 switching methods means that 1-3 loads can be switched at a time. There were no real downsides with this design, the only slight downside is that it would be harder to demonstrate, but this is very minor considering the advantages of this design.

5.2 Software – Decision Making Program

There is a lot more flexibility with the design of my software compared to flexibility with the design of the hardware. With most programming, there are often several ways of producing the same outcome.

5.2.1 Program Overview

The software that I will be developing as part of the solution will consist of a single program that runs on the device with the sole purpose of making decisions about switching a load on or off. The software will be written in Java and run on Raspbian Linux. This decision-making program will be launched when the Raspberry Pi boots up as this will allow it to initiate and run without human interaction. I am choosing to implement my decision-making program as a command line program. There was no point making this a GUI program as the basic decision-making algorithm needs to run discretely on the device without displaying visual information or requiring human interaction. There is the possibility of producing an optional GUI in the future to allow users to remotely interact with the program, however this is not necessary for now. A command line program will allow many optional input parameters when the program is launched. This is handy for development and testing, but will have limited benefits as the program will be auto-run when the device is deployed.

5.2.2 Logic Overview

The core algorithm of my program involves a sequence of steps that will repeat at certain intervals. Initially I will set the algorithm to repeat every 30 seconds, however this can easily be changed. The core algorithm steps to be repeated at regular time intervals include:

1. Using the API to fetch the currently power status of a solar panel system
2. Store the power status information in memory for use in the near-future
3. Logging the same power status information to a file for future use, this is done in case the program crashes or the device is turned off
4. Using the power state information along with recently power history to make a decision about which loads should be turned on or off
5. Send control signals via GPIO pins to turn a switch on or off

6. Find how long the above steps took, subtract this time from the interval amount (e.g. 30 seconds) and sleep for the resulting amount of time

There will also be certain calculations performed on logged data to enable it to be used by future iterations of the core algorithm. These calculations will be performed at the end of each day, or potentially other times when needed.

5.2.3 Solar Panel System API

The program will need to be able to use the solar panel system API in order to fetch the power status of a system. As this is an HTTPS based API, I will need to manipulate HTTPS queries to match the requirements of the API, this includes modifying HTTPS headers. I will develop methods that perform these HTTPS query manipulations to simplify the development of this program. Being able to use a single method to perform functions like querying the status of a system will simplify the algorithm logic.

5.2.4 Power History Information

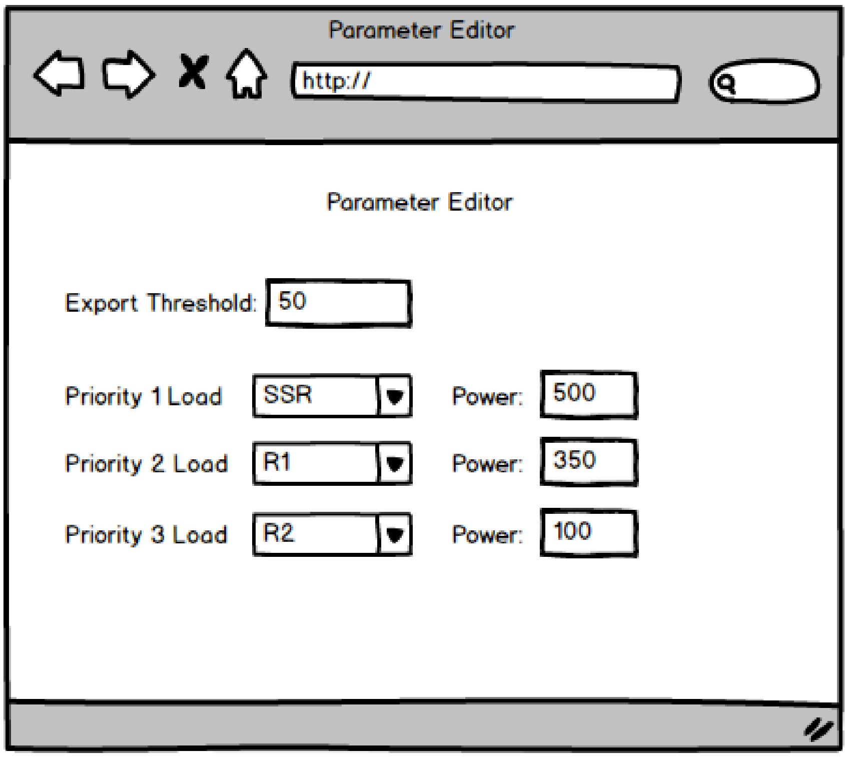
The program will need to have a way of storing recent information for later use. This information will need to be processed to extract the important and relevant information for future use. The program will also need to perform error checking and handling on the data that is entered to account for connection errors and power interrupts to the Raspberry Pi. I will choose to use CSV files rather than a SQL database to store information as I don't feel there is a need to set up a database for this project and it would just add unnecessary complexity and overhead.

5.2.5 GPIO Controlling

To switch loads I will need to implement methods for operating the GPIO pins. These methods would be called to change the state of different GPIO pins.

5.3 Software - Web Parameter Editor

Once the device is set up and running the decision making program, users will need a relatively easy way to change load information. This information includes what switching methods are being used, how much power each load uses, and the priority of each load. For example if the user connects a 500W load onto a wireless switch, they would need to tell



Parameter Editor

Export Threshold: 50

Priority 1 Load	SSR	Power:	500
Priority 2 Load	R1	Power:	350
Priority 3 Load	R2	Power:	100

FIGURE 5.4: A rough design for the web interface

the program that they have done this. It would be too hard at this stage for the program to detect this change automatically and accurately. The user can't be expected to SSH into the Raspberry Pi and edit a configuration file each time this happens. To make this easier for the user, I have designed a simple web interface for modifying parameters on the device. As the Raspberry Pi can easily run an Apache[17] web-server, the web interface can be run on the device. This will allow the user to easily edit load information in a web browser while the device is running, assuming the user is connected to the same local network that the device on.

The actual design of this web interface will be very basic for this project. It will just consist of a form where users can edit the priority and power-consumption of each type of load. The web interface will be constructed using php [18] and will perform server-side validation of entered information. I chose to use php due to being so easy to set up an Apache [19] web server on a Raspberry Pi and run a php script.

Figure 5.4 is a rough mock-up of what the web interface for the parameter editor will look like.

Chapter 6

Decision Making Algorithms

This chapter looks at the algorithms behind making intelligent decisions about whether to turn loads on or off.

6.1 Overview

Making decisions about whether to turn loads on or off is an important part of my project. In the second semester of this project, a large percentage of development time was dedicated towards trying to implement intelligent decision logic. This was an iterative process and several different algorithms were implemented. The goal for this algorithm is to assign 100% of the excess power to discretionary loads if power was being exported at a low reimbursement rate. The fourth and final algorithm that I had implemented turned out to be the most simple, but also the most effective as it achieved the 100% assignment goal. The earlier algorithms were focused on only switching one load on, but the final was focused on switching multiple loads on.

6.2 First algorithm

The first algorithm consisted of 4 main conditions. If all 4 of these conditions were met, the load would be switched on. Multiple loads were not planned to be implemented yet. This first algorithm required logged information to be recorded on the system for future decisions. Information like the amount of power exported each day and the amount of solar energy generated needed to be stored. To store this information I implemented two Java classes, named DataRecord and DayData. DayData stores information about recent days in memory, 1 instance for each day. DataRecord holds multiple DayData objects in

a circular array, with the most recent data at the start of the circular array. DataRecord generates information from several days' worth of data (from DayData objects) for use in decision making.
I chose to not use this algorithm due to the fact it was only assigning about 20% of the available excess power to discretionary loads, despite the fact ample power was being exported at a low rate during the testing of this algorithm. However, I kept the code from my DayData and DataRecord classes and extended them for future algorithms.

6.2.1 Condition 1 – Correct Energy Control State

The first condition checked the current energy control state. If this state enabled energy to be exported to the grid, then this condition passed. Otherwise this condition failed. This condition compared the current state to a set of known hardcoded states

6.2.2 Condition 2 – Exported enough

The second condition checked how much power the system has exported to the grid over the last 10 days. This information was gathered from both csv logs and memory, via the DayData and DataRecord classes. If the system had exported power over the last 10 days that was more than the defined threshold, then this condition passed. Otherwise this condition failed.

6.2.3 Condition 3 – Switching won't result in import

The third condition checked that if a load was switched on it wouldn't result in the system having to import power from the grid. This condition checked if the planned discretionary load's power was less than the current amount of power being exported. If the discretionary load was less than the current amount of power, then the condition passed, otherwise it failed.

6.2.4 Condition 4 – Battery will be full by sunset

The fourth and final condition checked that the battery will be filled by sunset. To make this decision it looked at the estimated amount of power that would be generated that day, and checked that it was greater than the amount of power needed to fill the battery. The estimated power to be generated for the remaining hours of the day was calculated from data that had been logged on the system over the past few days, via the DayData and DataRecord classes. If it was estimated that the battery will be full by sunset with the discretionary load switched on, then the condition passed. Otherwise the condition failed.

6.3 Second Algorithm

The second algorithm for decision making was based on the first algorithm, it was designed to expand on the first 4 set of conditions from the first algorithm. The main addition to this second algorithm was the inclusion of state-machine simulations. These simulations would involve simulating a system from the current point in time until the end of the day, with the goal being to find out the amount of charge left in the battery by sunset. I would have done 2 simulations at each time index during the day. The first simulation was to estimate the amount of charge left in the battery if a discretionary load was to be turned on now. The second simulation was to estimate the amount of charge left in the battery if a discretionary load was to be turned off now. If the simulations estimated that the system would have the same amount of battery charge level by sunset if the load was switched on now, then a condition would have passed. This condition would have either been included with or set to replace one of the previous 4 conditions from the first algorithm.

To simulate a system, I needed to know which rule the system would be in at each point in time. To know the predicted rule at each point in time, I implemented 3 classes. These classes are known as “Rule”, “RuleSet” and “Condition”. These are all basically data classes with a few useful methods for funding the current rule. A RuleSet object holds multiple rules in an internal array. Each rule has at least one exit and entry Condition. I reused the same Condition class for both exit and entry conditions. At each point in time I would check which Rule is currently being used, process numerical power information based on this rule, and finally check if any of the exit conditions of this rule were being met. If exit conditions were being met, then I would check the entry conditions of all the other rules in the rule set to find out which rule the system would be in next. The numerical power information for all points in the future until sunset were based on estimated data that was generated from logged information.

I chose to abandon this second algorithm as I found these state machine simulations were too hard to implement. There were too many variables with the simulations that I just couldn’t estimate or simulate. There was also behaviour going on in the system that was being controlled from the system provider’s side, which I wasn’t aware of. The information that was available to me was not enough to make accurate simulations. Although

I chose not to implement this algorithm, I have kept the 3 classes that I created for this algorithm as part of my solution. These could potentially be used in the future if this project was to be expanded upon.

6.4 Third Algorithm

For the third algorithm, I went back to the first algorithm as the state machine simulation idea was scrapped. Ulrich and I worked on improving the first algorithm to make sure it chose to assign more of the excess power to a discretionary load, as there was still a lot of excess power being exported that was not being assigned to a discretionary load. This third algorithm was based on the 4 conditions from the first algorithm, but with some modifications to conditions 1 and 3. Conditions 2 and 4 for this third algorithm are the same as conditions 2 and 4 from the first condition.
I chose not to implement this third algorithm as it still didn't choose to switch on loads enough. It was an improvement on the first algorithm as it assigned about 50% of the excess power to a discretionary load, however there was still a lot of power being exported that wasn't assigned to a discretionary load.

6.4.1 Condition 1 Modifications – Correct Energy Control State

For the first algorithm, the first condition was relying purely on the name of the energy control state to decide if it passed or not. I found that 1 particular state (that was previously set to fail the condition) did allow export under certain circumstances. This condition was modified to pass for that particular control state, provided the system was also exporting power.

6.4.2 Condition 3 Modifications – Switching won't result in import

For the first algorithm, the third condition was to make sure that the system was exporting power to the grid than the amount of power that the discretionary load consumes. In the first algorithm, the amount of power being exported to the grid was only based on the amount of power being sent from the solar panels to the grid, however it is also possible for power from the battery to be sent to the grid. This condition was modified to include power being exported to the grid from the battery when calculating the export power.

6.5 Fourth and Final Algorithm

For the fourth algorithm, a much simpler algorithm was designed as both Ulrich and I felt that we were previously over-complicating the algorithms. This fourth algorithm was also developed to take multiple discretionary loads into account. I designed each load that is configured with the system to have its own priority. In the case that there is enough excess power to power some discretionary loads but not all the loads, the loads are switched on in order of their priority. This algorithm consists of 2 conditions. The first condition applies to all the loads, and this condition must pass or else no loads will be switched on at all. The second condition applies separately to each load, and each load that passes this second condition will be switched on.

This algorithm turned out to be the most effective as it assigns 100

6.5.1 Condition 1 – Exported Enough

The first condition for this fourth algorithm is the same as the second condition from the first algorithm. This first condition checks how much power the system has exported to the grid over the last 10 days. This information was gathered from both csv logs and memory, via the DayData and DataRecord classes. If the system had exported power over the last 10 days that was more than the defined threshold, then this condition passed. Otherwise this condition failed.

6.5.2 Condition 2 – Switching won't result in import

The second condition for the fourth algorithm is based on the third condition from the first algorithm, but it is designed for multiple loads. This algorithm first calculates how much total excess power is available to be used for discretionary loads. This value is the sum of the amount of power currently being exported and the demand of any discretionary loads turned on in the previous round. Once this value is worked out, each discretionary load is checked (in order of priority) to see if it can be turned on or not. For each load, if the power consumption of the discretionary load is lower than the amount of total excess power available for excess loads then the load will be turned on. The total excess power available then has the power consumption of the discretionary load subtracted from itself to see how much remaining excess power can be used. This repeats till all the discretionary loads have been switched on or off.

Chapter 7

Implementation

This chapter looks at the parts of the load-switching solution that have been implemented throughout the year as part of this project.

7.1 Raspberry Pi configuration

Once I had decided to use a Raspberry Pi as the device for this project, I managed to get a Raspberry Pi Model B, which was the latest model at the time. Since then a model B+ came out which has currently been implemented inside the relay box. Once I had decided to use Raspbian as the operating system on the device, I went ahead and installed Raspbian on the Raspberry Pi. Once it was installed I didn't have to alter the settings much, I left most of the settings as default due to the fact it is already mostly configured to how I want it to be. Java 7 was already pre-installed with Raspbian so there is no need to install it. One thing I noticed was that the default DHCP configuration wasn't working with my network setup so I configured it with a static IP address. Once it was connected on my network, I could use PuTTY [20] to SSH onto the device. From SSH I am able to launch the decision making program and manipulate files. I also installed WinSCP [21] to manipulate files over the network using a GUI as this was easier than using the command line.

7.2 Hardware related

The hardware side of the solution was implemented by my academic supervisor, Ulrich Speidel. This was due to the fact that the BTech (IT) degree is a software-based course and students are not taught the skills or provided with the equipment necessary to implement

the hardware that was required for this project. There was also wiring that I would not have legally been allowed to do. Also apart from a few basic theoretical electrical concepts, no electrical engineering knowledge is taught as part of this degree.
Once we had decided on the separate hardware design with multiple switches, Ulrich implemented the hardware part of the solution. The finished solution involves 4 main components, the logic box, the relay box, and 2 wireless receiver switches.

7.2.1 Logic Box

The Logic box is where the logic and decision making occurs. The control signals for the switches are also sent from this box. This box houses the Raspberry Pi, a USB power supply for the Raspberry Pi, an optocoupler, wireless remote control, reset button and a 3.5mm socket. There is also a power cable on the outside. The Raspberry Pi has a USB Wi-Fi module attached so that it can get an internet connection via a wireless router, hence there is no need for an Ethernet connection. The GPIO pins on the Raspberry Pi are connected to both the optocoupler and to the 3.5mm socket. The pins connected to the optocoupler are to send control signals to the wireless remote as the optocoupler is connected to the wireless remote. The optocoupler is needed due to the voltage difference between the wireless remote as the GPIO pins couldn't be directly connected to the wireless remote. The pins connected to the 3.5mm socket are used to send control signals to the relay box, these pins are connected directly to the socket and there is no need for an optocoupler. The Reset button is used to reset the Raspberry Pi in the event of it crashing as the box is hard to access.
Figure 7.1 shows the logic box when it is closed. The mains power cable for the USB power supply is on the left, and the 3.5mm socket is on the right. Figure 7.2 shows the logic box when it is open. The bottom section of the box is in the lower half of the picture. This bottom section contain the USB power supply on the left, Raspberry Pi on the right, and the reset button at the top. The top section of the box contains the optocoupler top right, and the remote sender bottom right.

7.2.2 Relay Box (SSR)

For programming purposes, I refer to this switch/box as "SSR", which stands for Solid State Relay. The relay box contains a sold-state relay and a 3.5mm socket. On the outside of the box there is a mains power point, and a power cable. The 3.5mm socket is connected to the control signal socket of the relay and is used to control the state of the relay. The power cable is connected to the solid-state relay to provide it with power to

FIGURE 7.1: The Logic Box (Closed)

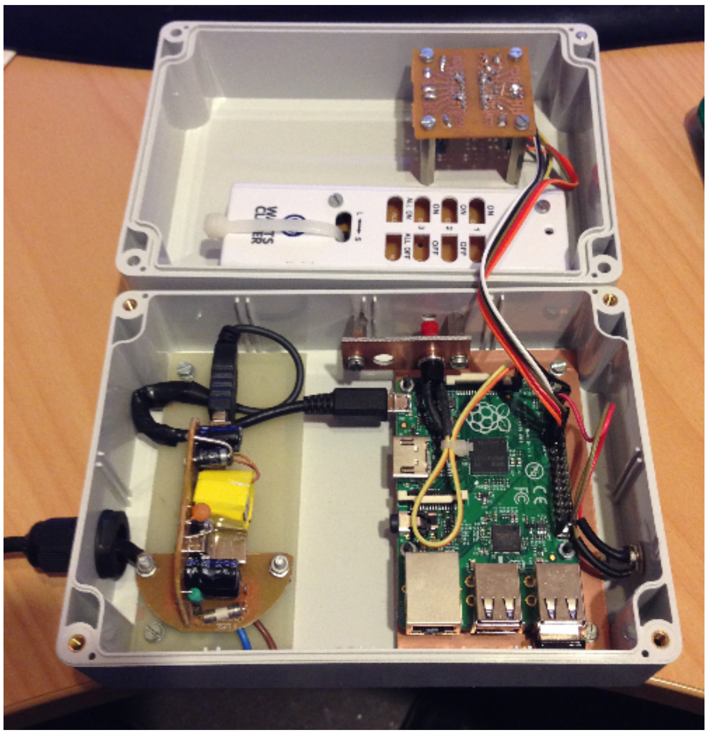


FIGURE 7.2: The Logic Box (Open)

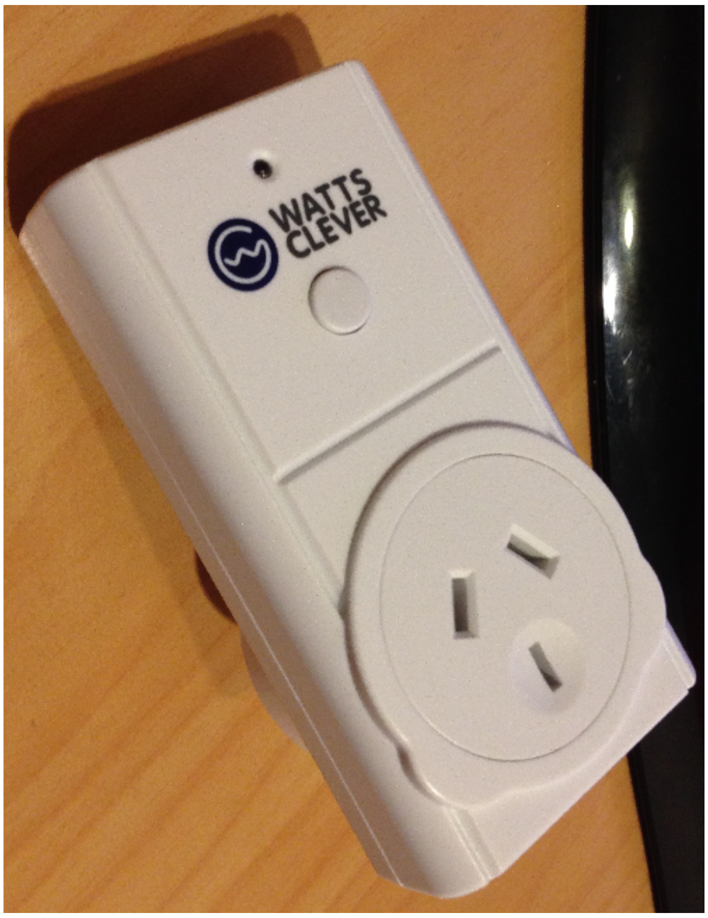


Figure 7.3: A Remote Switch

drive a device. The relay is also connected to the mains socket. When a control signal is sent to the relay, it will allow power to pass through the relay and it will flow from the mains power cable to the mains socket.

7.2.3 Remote Switch (R1/R2)

For programming purposes, I refer to these switches as R1 and R2, which stand for Remote 1 and Remote 2 respectively. Unlike the logic and relay boxes, the remote switches were bought premade. These switches simply plug into a mains socket, and the mains load to be switched is plugged into the remote switch. These switches function similar to the relay box, except instead of a 3.5mm cable delivering the control signal from the logic box it is a wireless signal. These remote switches can be placed in various locations in the house, away from the logic box. When these remote switches receive an appropriate wireless signal, they allow power to pass through the switch and to the load. Figure 7.3 is a picture of one of the remote switches (R1/R2).

7.3 Software – Decision Making Program

As planned in the design chapter, I have implemented the decision making program as a command line program in Java that runs on a Raspberry Pi (running Raspbian as the operating system). This Raspberry Pi runs in the logic box as mentioned in the previous section. The decision making program consists of a core logic loop that repeats every 30 seconds. This loop will repeat forever unless specified otherwise with a command line parameter. This loop contains the main decision making logic and several objects of different classes are interacted with.

7.3.1 Before the Loop

Before the loop begins, some one-off code runs that sets up the program to loop successfully. This includes setting up variables such as the current time index, loading data from recent CSV files into memory (via a DataRecord object) and initiating an API connection (via an APIConnection object). The time index is an integer that specifies which 30-second section of the day is the current section, it ranges from 0 (0:00:00 – 0:00:29) to 2879 (23:59:30 – 23:59:59). The connection to the API is initially created during this pre-loop code, however this connection will be reset/re-initialised during the main loop if there are problems, or at midnight (time index 2879). A load controller instance is also created unless the program has been run specifically just to log information.

7.3.2 During the Loop

Once these initial parameters are set up, the main logic loop begins. At the start of every loop iteration, parameters are loaded from the config.txt file. This is the configuration file that the web interface produces. This contains information about load priorities, load power consumption, load switching types, and the export threshold. The next step is to request the status of the system using the API. I have implemented a method inside API connection class that uses the API to request the status and if the request is successful then it returns this information as a SystemStatus object. If there was a network or parsing error then this status request method returns null. This request status method uses either XML or JSON depending on the launch parameters of the program, but both types will return an equal SystemStatus object.

If the system status request is successful then the program begins to look at decision making. In this case the program first sends the relevant numerical data from the SystemStatus object to a DataRecord object to be stored in memory. The DataRecord object selects the current DayData object that is used for the current day and stores the information in this object. This stored information will be used for future decisions. The amount of power currently being exported is calculated from information within the SystemStatus object. The program then begins the main decision-making algorithm. This algorithm begins with getting the amount of power exported over the last 10 days via a DataRecord object. This 10-day export value is checked against the export threshold which was defined in the config.txt file using the web interface. If the 10-day export is higher than the export threshold, then the next step of the decision making begins. Each configured discretionary load that was read from the config.txt file is checked to see if the power consumption of the discretionary load is lower than the current amount of excess power being generated. If this is the case then that load will be set to be switched on. All the configured discretionary loads are checked in order of their priority.

If the system status request is not successful (the object was null), the program adds blank data to the DataRecord object. This lets the DataRecord object know that the status request failed and that the DataRecord object should attempt to fill the recent gap of data if possible. All the discretionary loads are also set to be turned off, regardless of how they are configured.

Once all the switching state of all the discretionary loads have been decided, the program begins to physically switch them via a LoadController object. This LoadController object contains methods for controlling GPIO pins and sending the appropriate control signals to the switching devices. Once these control signals are sent, the corresponding loads are turned on or off. After the physical switching the next step of the program is to log the current information to a CSV file. This logging is done via a static method that belongs to the Logger class. The information that is logged to the CSV file includes the time index, the current time, numerical information about the recent status request, the 10 day export amount, the amount currently being exported and the switching state of each load. The program then checks if it is the end of the day or not by looking at the current time index. If the current time index is 2789, then the program calls a method on the DataRecord object that generates some statistical information based on logged data. This statistical information can be used in the future for decisions. Finally at the end of the loop iteration, the program decides how long to sleep for. The programs times how long these previously mentioned steps took to execute (including network delay when querying the server) and then sleeps for the corresponding amount of time. For example with the default setting of looping every 30 seconds, if these steps took 2.5 seconds it will sleep for 27.5 seconds to ensure it loops every 30 seconds. If it is the case that the calculated sleep time is less or equal to 0 then it will just begin the next loop iteration straight away. The program finally checks if there are still remaining loop iterations, if there are remaining iterations then the program will go back to the start of the loop, if there are no more iterations then the post-loop code will be executed.

7.3.3 After the Loop

Once the loop has finished iterating, some final code is executed. First the API is disconnected from using an APIConnection method called disconnect. Finally all the discretionary loads are switched to turn off, regardless of their previous states or configuration. Figure 7.4 shows a rough high-level view of the logic that is executed .

7.3.4 Solar Panel System API use

I have currently created 5 methods for using the solar panel system API. These include:

- Initiating a session with the API, this needs to be done first or else no other methods will work
- Querying the status of a system in JSON format, I plan to use JSON as the return format when querying the server due to reasons given earlier. This method parses the JSON information that is returned, constructs a SystemStatus object, and returns this object as part of the method call.
- Querying the status of a system in XML format, although I plan to use JSON as the return format I will leave the XML code in the class for now as it may have a future use. This method parses the XML information that is returned, constructs a SystemStatus object, and returns this object as part of the method call.
- Obtaining rule set information about a particular rule set in XML format. I perform a query based on a specific rule set ID and I get XML information about a rule set. I then construct a RuleSet object from this parsed XML information and return the RulseSet object.
- Disconnecting from the API session, this is optional but good practice

To reset a connection, I call the disconnect method, directly followed by the initiate method.

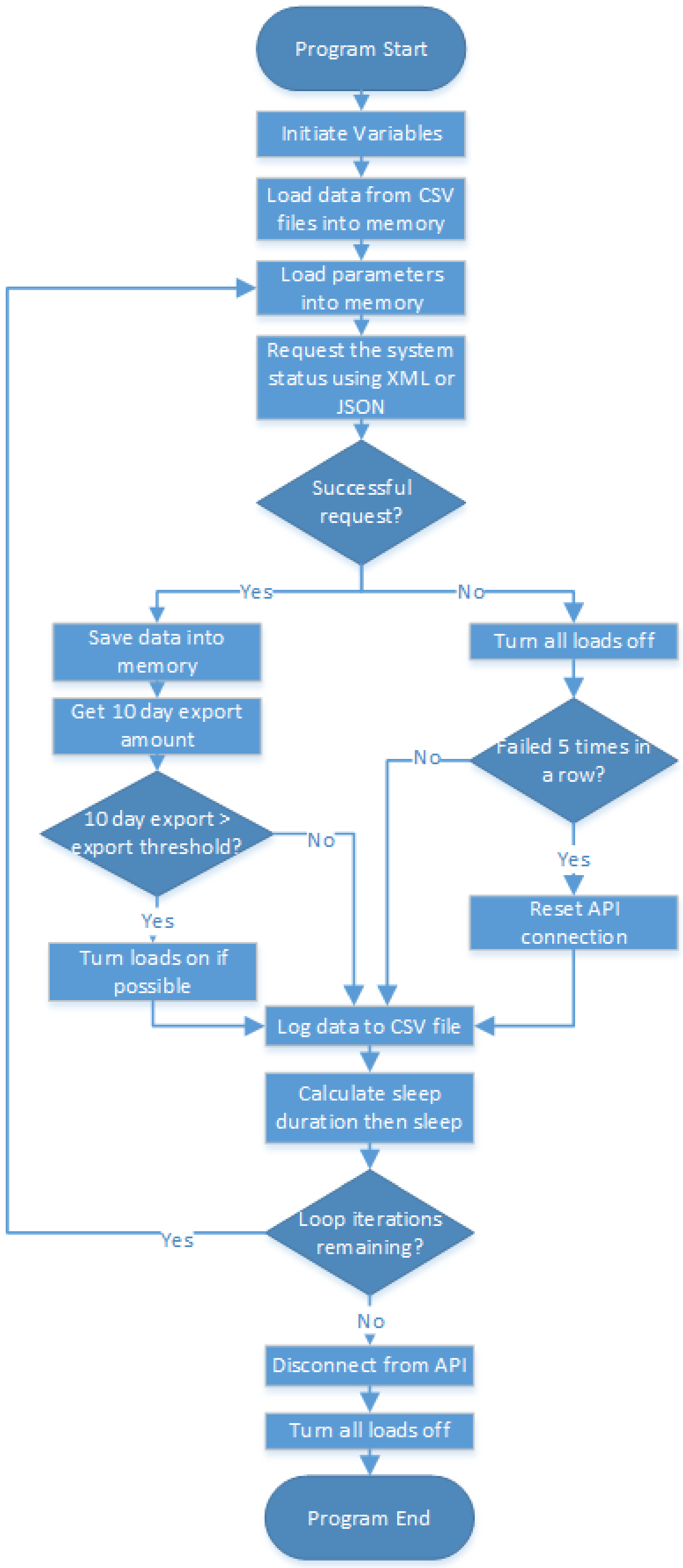


Figure 7.4: Program Structure

7.3.5 GPIO output

I am currently able to successfully send a control signal using the Raspberry Pi's GPIO ports. I installed the Pi4j library to do this. The Raspberry Pi isn't hooked up to a relay yet, however I have tested the GPIO output with an LED. The LED successfully lights up when my program wants it to. Later when the Raspberry Pi is connected to a relay, I will just send the same signal as I am doing currently.

7.4 Software – Web Parameter Editor

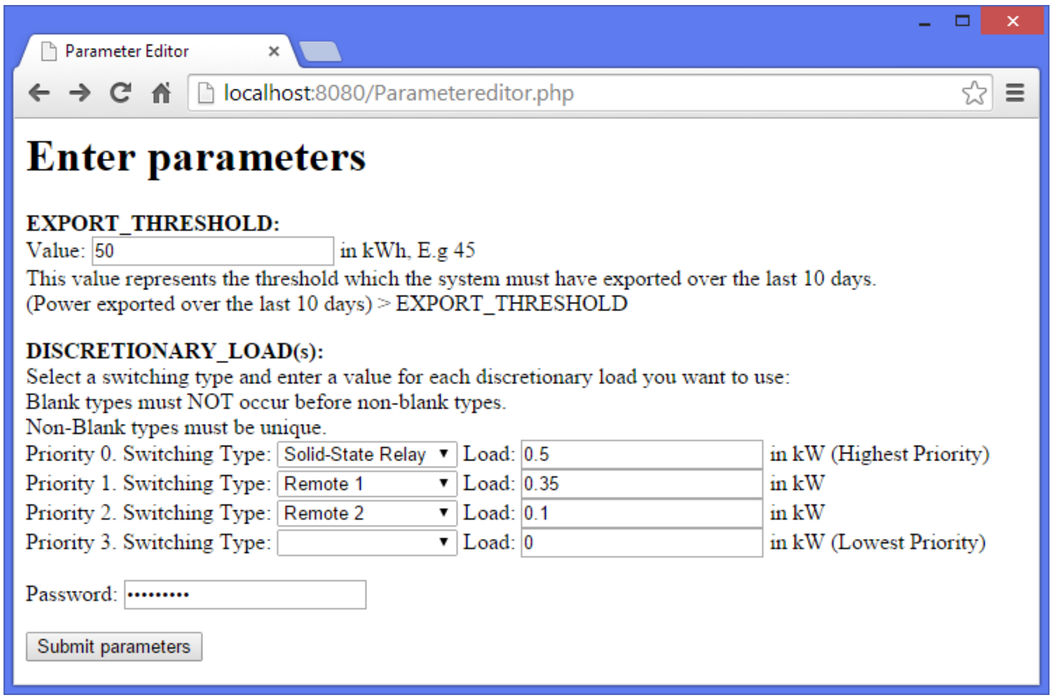
I constructed the web interface for the parameter editor using php in Notepad++ [22]. There was no point getting a php IDE installed as the size of the code was small and the complexity was low. I have implemented 2 php files as part pf this web interface, known as "ParameterEditor.php" and "Result.php".

ParameterEditor.php contains the main form that the user is prompted with. This form has fields where users can enter the export threshold, enter values for attached loads and select the priority for the loads. There is also a password check at the bottom to stop unwanted people editing values. Upon clicking the submit button at the bottom of the page, the values of these input fields are validated.

Result.php contains server side validation for the inputted information. Due to time constraints I wasn't able to implement client side validation. This server side validation checks all of the inputs to check they are in the correct format and that the password is correct. If all the information was validated successfully, the corresponding information is written to a file called config.txt. The decision making program will read this configuration file and load the parameters into memory for use when switching.
Figure 7.5 is a screenshot of the web interface.

7.5 Used Classes

I currently have 7 Java classes that I have implemented and used as part of my final submission of this project.



Parameter Editor

localhost:8080/Parametereditor.php

Enter parameters

EXPORT_THRESHOLD:

Value: 50 in kWh, E.g 45

This value represents the threshold which the system must have exported over the last 10 days.

(Power exported over the last 10 days) > EXPORT_THRESHOLD

DISCRETIONARY_LOAD(s):

Select a switching type and enter a value for each discretionary load you want to use:

Blank types must NOT occur before non-blank types.

Non-Blank types must be unique.

Priority 0. Switching Type:	Solid-State Relay	Load:	0.5	in kW (Highest Priority)
Priority 1. Switching Type:	Remote 1	Load:	0.35	in kW
Priority 2. Switching Type:	Remote 2	Load:	0.1	in kW
Priority 3. Switching Type:		Load:	0	in kW (Lowest Priority)

Password: •••••••••

Submit parameters

FIGURE 7.5: The Web Interface

7.5.1 DecisionMaker.java

The current DecisionMaker class consists of code from both the old DecisionMaker.java and LoopTask.java as mentioned in the mid-year report. I ended up merging the code from both these classes into 1 class as I realised the old DecisionMaker class only really consisted of a main method. This main method could easily be merged with the LoopTask code so I merged them. This class now implements Runnable.
This combined class now contains the main method of my program that is called when the program is launched, as well as the main high level logic of the program. When the main method is called, this method sets up some logon information about the particular solar panel system that the program is designed to connect to. This main method also initiates some objects like an APIConnection instance, then finally starts running the main logic loop that loops every 30 seconds. The main logic loop implements runnable and launches on a different thread. This provides more flexibility and control over the main logic computation as it could be modified to finish early. At times I have noticed a loop iteration can take more than 30 seconds to execute. In the event of this happening the thread will not sleep at all, it will immediately being the next loop iteration. The program keeps a track of how many loop iterations have occurred, and at which time each iteration should sleep. By doing this the program can correct its timing and return to a regular pattern in the event that 1 or more iterations take more than 30 seconds. This class consists of 445 lines of code.

7.5.2 APIConnection.java

This class contains the 4 wrapper methods mentioned earlier for interacting with the HTTPS based API. This class enables the LoopTask thread (where the main high-level logic is) to interact with the API using simple methods, rather than having several lines of manipulating and parsing HTTPS queries each time the class wants to use the API. This class consists of 609 lines of code.

7.5.3 Logger.java

This class contains static wrapper methods for writing information to a .csv file. It was made to simply file writing for logging. Currently the class separates its logs into 1 .csv file per day for simplicity, in such a way that they are nicely ordered. At the moment this class is currently being used by a LoopTask instance to log information about the system at regular intervals. This information will be used to work on and improve my decision-making algorithm. At the earlier stages of development I logged every possible bit of information about power flows in case they will be required later by a decision making algorithm. For the final version of my program I log less information as I have finalized the current decision making algorithm. Each log entry stores information from the XML/JSON status request, as well as some basic load information about what loads have been turned on or off.
During the first semester, logging methods were non-static and an instance of this class had to be created before logging methods could be used. However upon examining code I found there was no reason to use an object as there was no state information that needed to be kept. I then changed all the methods to static methods as this simplified code. This class consists of 108 lines of code.

7.5.4 LoadController.java

This is a simple class that simplifies GPIO output on the Raspberry Pi. It contains methods for switching the GPIO output on or off. These methods will be used for controlling the loads that will consume excess power. An instance of the LoopTask class will use these methods after it has finished making a decision. This class uses one of the external libraries that I am currently using, Pi4j, as Raspbian doesn't include a Java library for GPIO interfacing. Initially this class was designed around switching one load, but in the second semester this class was modified to switch multiple loads.
This class consists of 88 lines of code.

7.5.5 DayData.java

This class holds data for a particular day. Every time a system status is requested, the relevant numerical data is stored in an array, with an array for each time index of the day. At the end of each day, relevant statistical information is generated from these arrays for future use. The main decision logic doesn't interact with DayData objects directly, but instead interacts with them via the DataRecord class.
This class consists of 275 lines of code.

7.5.6 DataRecord.java

This class contains information from recent days. It contains a circular array of DayData objects, where the most recent objects are at the start of the circular array. At the end of each day, after the DayData calculations have been made, the DataRecord class calculates its own data from the previously calculated data that the DayData objects had produced. This includes information such as how much power has been exported over the last 10 or so days, and what the average total solar power generation is per day. This data is used for intelligent decision making, and the decision making algorithm can accesses this data directly from a DataRecord object. This class also contains methods for loading data from CSV files into memory and methods for adjusting CSV information for daylight savings.
This class consists of 441 lines of code.

7.5.7 SystemStatus.java

This is a simple data class that holds recently retrieved information about a system. It is simple as the main goal of this class is to hold 2 String arrays and 1 integer array. If Java had better support for tuples I could have done without this class.
This class consists of 71 lines of code.

7.6 Unused Classes

There were also 3 classes that I developed during this project that aren't currently being used. I have chosen to talk about them as I have submitted these 3 classes with my final code submission as these classes could be used or expanded upon in the future. These classes all work as they should, there is just no use for their functionality at this point in time.

7.6.1 RuleSet.java

This class is a data class but includes some methods. A RuleSet object contains multiple Rule objects in an internal array. I created this class when trying to implement state machine simulations as the state machine would have obeyed one of the rules in a RuleSet at each point in time. A RuleSet object can be created from an APIConnection object when it fetches relevant Rule information from the API.
This class consists of 89 lines of code.

7.6.2 Rule.java

This class is also a data class that contains some methods. A rule object represents the current rule that the system is in. I created this class when trying to implement state machine simulations as the state machine would have obeyed a rule at each point in time. A rule object contains at least 1 entry condition and at least 1 exit condition.
This class consists of 117 lines of code.

7.6.3 Condition.java

This class is essentially a data class but also contains a few useful methods for comparing conditions. A Condition object can be used as either an exit or entry condition of a rule as both these types of conditions are very similar in terms of implementation. Entry conditions of a rule are checked to see if the system can enter a particular rule or not. Exit conditions of a rule are checked in a similar way to see if the system can exit a particular rule or not.
This class consists of 36 lines of code.

7.7 External Libraries

I tried to limit the amount of external Java libraries used during this project. I only used an external library when it enabled my program to do something it couldn't do with standard Java libraries, or I felt the library brought significant benefits.
The .jar files for these libraries are stored in a lib folder that is added to the classpath when the program is run. Having these files in a separate folder simplifies code as it is easier to tell which classes I have implemented myself, and which ones are external.

7.7.1 GSON

GSON is a Java library for JSON parsing [23]. It is created by Google. I needed to use a JSON parsing library as Java surprisingly doesn't include one with the standard libraries. I chose GSON as it is easy to use and fairly efficient when compared to other external Java JSON libraries. This library enabled me to return data in JSON format, which uses up less bandwidth than when compared to XML.

7.7.2 Pi4j

Pi4j is a Java library for using the GPIO ports on a Raspberry Pi [24]. I needed a library for interacting with the GPIO ports as Raspbian doesn't include one for Java by default. This is the only downside I have found when I chose to use Java over Python, as Raspbian includes a simple Python GPIO library. I chose Pi4j as it is the only GPIO library I could find for Java. Pi4j provides 3 particular methods for interacting with the GPIOs that I have used in my code. The first two methods, high() and low(), are used to supply a GPIO with voltage or turn it off respectively. These two methods are used on the same pin for turning the solid-state relay on or off. The other method, pulse(), is used to supply a pin with voltage (high) for a particular amount of time, and then turn it off (low). This method is used for controlling the wireless switches as to turn them on I need to set a pin to high for 1000 milliseconds, then to turn it off I need to set a different pin to high for 1000 milliseconds.

7.7.3 Joda-Time

Joda-Time is a Java date and time API [25]. It offers many useful time/date methods that the standard Java API doesn't offer yet. These methods can be used for manipulating and getting information from time/dates. There were two cases when I found this API particularly useful. The first was when dealing with the time index of the day. I defined the time index as an integer between 0 and 2879 which represents which 30-second time slot that the current day was in. The Joda-Time API provided a method called millisOfDay() which showed how many seconds have passed since the start of the day. Using the result of this method I was easily able to calculate the time index of the day by performing a simple calculation on it. The second case of finding the Joda-Time API useful was when dealing with log files. This API provided a method called daysBetween() which could easily get the amount of days between two dates. This was useful when dealing with log files as I could easily see how old log files are by getting the amount of days between the date of the log and the current date. Using the Joda-Time API simplified my code as it allowed me to easily perform certain time/date operations that would have otherwise been much more difficult with the standard Java API.

7.8 Implementation Results

I implemented the software to drive the hardware successfully and the device runs as it should do. The program successfully connects to the API to retrieve system statuses, logs information to CSV files, loads information from CSV files when it needs to, stores information in memory for later use, makes intelligent decisions about switching loads on or off and sends appropriate control signals to switches which successfully turn devices on and off. The information and parameters about loads can be modified using the web-interface regardless of whether the program is running or not. This information is taken into account upon the next iteration of the decision loop.
Loads are only turned on if 2 strict conditions are met. The first is that the amount of power that the system has exported power over the last 10 days is greater than the export threshold. The second condition is that there is enough power being exported such that turning on a load won't cause the system to bring in power.

Chapter 8

Challenges

There were many challenges that occurred during the design and implementation of this project. This chapter is not talking about the optimisation challenge that is mentioned in the introduction.

8.1 Daylight Savings

One challenge that I faced during this project was the fact that clocks get adjusted by 1 hour during daylight savings. This didn't affect or occur to me in April when daylight savings ended as I hadn't started constructing the decision making program, however this was a problem in September when daylight savings started. The problem was that my logged information in CSV files became offset by an hour when daylight savings started. At this point I was developing the 3rd decision making algorithm which relied heavily on time-based information. When I calculated values such as the estimated amount of solar power to be generated until sunset, the data to work with was an hour off and these calculations produced erroneous results.
To solve this problem, I implemented an algorithm to offset previously recorded data in CSV files to neutralise the offset of daylight savings. Initially this algorithm was just designed to adjust data going into daylight savings (September) but I later parameterised it to work for both the start of daylight savings and the end of daylight savings.
It turned out that once I implemented the 4th decision making algorithm, a 1 hour offset either way made no different to the data I needed to collect from the CSV files. However I have left this code in as it could be used in the future.

8.2 Decision Making Algorithms

As mentioned in the Decision Making Algorithm chapter, the iterative task of developing an accurate decision making algorithm was a challenge that I faced. It was a challenge as it was hard to get an accurate result from this algorithm that decided to switch on loads enough. To solve this problem, Ulrich and I approached this challenge in an agile way. With each algorithm, it was planned, implemented and then tested. After letting the new algorithm run and be tested for a week or two, we would decide whether it was good enough or not based on how much power it decided to assign to excess loads. Finally by the 4th version of the algorithm it was choosing to assign an appropriate amount to discretionary loads so I kept and finalised that 4th algorithm. As testing each version of this algorithm took at least a week, we had to assign plenty of time to developing this algorithm, which is why we started from the start of the second semester.

8.3 API not displaying the Current Rule

When I was attempting to implement state machine simulations as part of the 2nd decision making algorithm, I found there was a challenge where the API didn't tell me the current active rule. I did however know the ID of the current rule set. To solve this problem I implemented the APIConnection method that creates a RuleSet object based on rule set XML information. Once I had this RuleSet object, the next task was to go through all of the rules and find out which one was the current rule. To do this I had to iterate through every rule in the RuleSet, comparing current information with each entry condition of each rule. Upon doing this I would find out which rule should be the current rule being applied in the system.

8.4 Java Classpath Configuration

Once I started using external Java libraries, a challenge occurred about configuring Java classpaths. Classpath configuration was not a problem when I was coding the solution as the Eclipse IDE [26] took care of that configuration for me. However the problem occurred once I started to deploy the program on a Raspberry Pi where I had to launch the program from a command line. Each time I wanted to launch the program I had to configure classpaths and parameters. It would be even more of a problem when it came to deploying the program later on. To solve this problem I implement Unix shell scripts that had all the command line parameter information including classpath details already configured. This made running the program very simple as only the script needed to be run without parameters.

Chapter 9

Evaluation

This chapter evaluates the solution that has been implemented. It also looks at using this solution to help answer the research question.

9.1 Testing

Due to the nature of my program, there were limited ways I could perform testing. Performance testing wasn't appropriate for my program as my program is not computationally intensive. The majority of the time the program is sleeping until the next loop iteration, or while it is awake time is mostly spent waiting for API responses or waiting during/after sending control signals. As this program is a command line program that discreetly runs on the Raspberry Pi, usability testing wasn't appropriate either. Usability testing could have been performed on the web interface, but I feel this would be unnecessary due to the web interface being such a small part of my project. The only real testing that could be done was checking the program ran correctly, and that it switched on loads appropriately. The physical switching has been tested with both the remote switches (R1 and R2), although I haven't had a chance yet to use the sold state relay switch (SSR).
Throughout development I have had current versions of my program running on my own Raspberry Pi at home. From the start of development I have been able to test the overall stability of my program. Using my own Raspberry Pi at home has allowed me to constantly test for crashes and code that broke my program. Later in development I have also used my own device at home to log information about decision outcomes. This was particularly useful during the development of the decision making algorithm.
I had limited opportunity to test the program on the actual hardware that Ulrich had implemented. The logic box that housed the Raspberry Pi was only presented to me during the last 2 weeks of the project. Upon testing my program on the Logic Box I found

the core of the program worked fine, however there were some problems when it came to switching loads. Some of the problems turned out to be related to my GPIO-operating code, and some were related to the wireless switch setup. I was able to fix up the software problems on my side, and Ulrich fixed up the hardware problems. After these fixes the switching worked fine, and I was even able to give a successful demonstration during my final seminar. Apart from this brief load-switching testing, I haven't been able to test the load switching side of the solution much. The load-switching tests I did do though worked correctly and as they should. These were just basic tests where given a situation I made an assumption about which loads should be turned on. These basic tests were all correct.
In terms of testing the accuracy of the actual decision making program, I was able to perform these tests on the Raspberry Pi running at my home. The metric used for these tests was how much of the excess power was assignable to discretionary loads. Provided the system has exported more power over the last 10 days than the export threshold, all of the excess power should be assignable to discretionary loads. Otherwise if the power exported over the last 10 days is less than the export power, then no power should be assignable to discretionary loads. The last iteration of the decision making algorithm (the one I have implemented), successfully assigned all excess power to additional loads.

9.2 Answering the Research Question

The research question as defined in the first chapter is "Can the strategic use of excess power provide additional value to the customer?" My answer is yes, but this is dependent on the circumstances. As mentioned in the first 2 chapters, users are reimbursed by their electricity retailer for power that their system exports back into the grid. Depending on the retailer, this value can vary a lot [2]. Some retailers pay a fixed amount no matter how much power is exported, and some pay a variable rate depending on how much has been exported recently. If the current reimbursement rate is relatively low, the user will get more value out of the power their system has generated if they can use this power with a discretionary load inside the house. A basic but clear example of additional value is as follows:
Say there are 2 almost identical houses, A and B, both with a 1kW electrical hot water heating system and a solar panel system just like the one mentioned in this report. For this example they both generate the same amount of power from solar panels, have the same power demands in the house, and have the exact same power reimbursement plan. Each house gets charged 20c for bringing in 1kWh from the grid, and gets paid 10c for exporting 1kWh. However house B has a 1kW hot water pre-heater and a load switching device as discussed in this report. For this example house B's hot water-pre heater is just

	Midday Export	Evening Import	Net Change
House A	1kWh @ 10c/kWh = +10c	1kWh @ 20c/kWh = -20c	-10c
House B	0kWh @ 10c/kWh = +0c	0kWh @ 20c/kWh = -0c	0c

Table 9.1: Simple Example of Additional Value

as efficient as the standard hot water heater and the heat energy lost from this pre-heater is negligible. At midday on a particular day there is 1kWh of excess energy generated by each house's system. House A exports this power to the grid and gets paid 10c. Instead of exporting, house B's load making device decides to turn on 1kWh pre heater for an hour. Later during the evening, both houses get low on hot water and 1kWh is needed to heat the water back up. House A powers up the in-build hot water heater and is charged 20c for the 1kWh of power brought back into the grid. However for house B, the already hot water from the pre-heater flows straight into the hot water cylinder and requires no heating. In this case House A has spent a total of 10c (+10c -20c) for this 1kWh of power, whereas House B has spent 0c for this 1kWh of power. Under these circumstances, house B gets an additional 10c of value per kWh. 10c doesn't sound a lot for this small example, but over time this would add up and provide plenty of additional value for the customer. Table 9.1 has a summary of the calculations performed in this example.

In theory, additional value can be added if efficient loads are switched on while export rates are low. To perform this switching I have implemented a decision making device to intelligently switch loads when excess power is available. This device works as it should and in theory this device should bring additional value to a customer when paired with efficient loads, but I haven't been able to thoroughly test it.

Chapter 10

Future work

This chapter looks at the possible future work and ways that this project could be expanded upon in the future.

10.1 Thorough Testing

Due to time constraints, I wasn't able to test this program as much as I would have liked to. This was due to both how the program needs to run for a long time to see get more detailed information, and that implementation was only completed during the last week of the semester. This implementation includes both my software, and the hardware that Ulrich was working on. If I had more time I would like to let the device run for weeks and/or months to examine the long term effects of the device. It would have been interesting to look at long term trends about at what times loads are switched and how frequently switching occurs.

10.2 Displaying Information

The decision making program that I have implemented is a command line application which just displays basic current and recent information. This information is displayed to the console and written to CSV files. A possible way this project could be extended would be to work on a GUI that would display more relevant information that was more meaningful to the general public. This GUI could be in the form of a web app, or a smartphone app. A web app would be able to be viewed on either a PC, or a smartphone, across many operating systems. A smartphone app would only be able to be viewed on a specific smartphone operating system. For the purpose of displaying information to the

user I feel a web app would be better as it would run on many more devices and operating systems. I also don't feel there would be any benefit to using a native smartphone app in this case.
A (web) app could be used to display current and recent information to the user. It could also display longer term information (as mentioned in the previous chapter) about power/time trends. Information could also be displayed about power saving due to more efficient excess power use. The functionality of the implemented php-based parameter editor could be integrated into this (web) app to create a single app companion to the decision making program.

10.3 Alerting and Manual Load Switching

The current solution for load switching automatically switches loads that have been pre-configured and already plugged in. These loads are in a state where as soon as they receive mains power they will start operating. Another method of load switching could be to alert a user when there is excess power available, and let the user turn on a load themselves. An example load that could be manually switched could be a clothes dryer. A clothes dryer would not work very well being automatically switched as wet clothes would have to wait for potentially several hours before the machine turned on, and even then it may not be switched on for that long constantly.
To know when to manually turn on a load or not, a user could be alerted that there is excess power available, and that it is a good time to manually turn on a load. This alert could be done via an email, txt, or app notification. An app notification could easily be pushed on the local network where a user would receive a notification on their smartphone app. I feel an email would be the best choice as it targets people using either a computer or a phone, and many people these days have a constant internet connection to their email-capable phone.

Chapter 11

Conclusion

This small chapter is just to conclude and sum up the report.

This report has been focused around the development and evaluation of an intelligent load switching solution that operates inside a house that has a solar panel system with storage installed. The purpose of this device is to turn on loads at appropriate times to make better use of excess power generated by a system. The reason for developing this device was to answer the research question assigned to of "Can the strategic use of excess power provide additional value to the customer?" This research question was assigned to me by Vector to explore and answer.
I started by researching the particular system and API that Vector allowed me to work with. I then looked into various technologies that could be used and then designed an appropriate load-switching solution. This solution involved both hardware and software components. I implemented the software component of the solution using Java to run on the Raspberry Pi-based hardware that my academic supervisor, Ulrich Speidel had implemented. This finished load switching solution was shown to assign all the available excess power to discretionary loads only if 2 strict conditions were met. These 2 conditions make sure that the system would currently export at a low rate, and that switching on a load won't result in unnecessary power import from the grid. The solution was also shown to successfully be able to switch loads inside a house using a variety of switching methods. This solution enabled me to answer the research question as it enabled loads to be switched in an intelligent way that can provide users of the solar panel system with additional value. My answer to the research question is yes, but only if the loads being switched are efficient with the power that they consume.

Acknowledgements

I would like to thank Dr. Sathiamoorthy Manoharan (Mano) for assigning me this project at the start of the year. I also want to thank him for organising and giving me feedback on my seminars.

I would like to thank everyone at Vector who has helped or given feedback during this project. I also want to thank Vector for giving me the opportunity to work on this project in the first place and for allowing me to work with a solar panel particular system. Specifically I would like to thank my academic supervisor, Steve Muscroft-Taylor for replying promptly to my questions about the project and for attending my seminars.

Finally I would like to give a massive thanks to my academic supervisor, Dr. Ulrich Speidel. Ulrich has been a great help during this project as he has always kept in contact and given me appropriate advice and assistance when needed. He has also put in a lot of work with the hardware side of this project due to this being a software-based project.

Bibliography

[1] Vector. Vector - about us. http://vector.co.nz/about-us.

[2] PowerSmart Solar. What will you be paid for your excess electricity? http://powersmartsolar.co.nz/Compare_electricity_retailers.

[3] Natasa Vulic, Malvika Patil, Yongjie Zou, Sri Harsha Amilineni, Christiana B. Honsberg, and Stephen M. Goodnick. Matching ac loads to solar peak production using thermal energy storage in building cooling systems - a case study at arizona state university. In *Photovoltaic Specialist Conference (PVSC), 2014 IEEE 40th*, pages 1504–1509, June 2014. doi: 10.1109/PVSC.2014.6925200.

[4] M. Chhabra, M. Lim, and F. Barnes. Frequency stabilization using solar smoothing, leveling and time shifting in a hybrid renewablenetwork. In *Innovative Smart Grid Technologies - India (ISGT India), 2011 IEEE PES*, pages 275–281, Dec 2011. doi: 10.1109/ISET-India.2011.6145396.

[5] E. Matallanas, M. Castillo-Cagigal, A. Gutiérrez, F. Monasterio-Huelin, E. Caamaño-Martín, D. Masa, and J. Jiménez-Leube. Neural network controller for active demand-side management with {PV} energy in the residential sector. *Applied Energy*, 91(1):90 – 97, 2012. ISSN 0306-2619. doi: http://dx.doi.org/10.1016/j.apenergy.2011.09.004.

[6] Alessandro Di Giorgio and Laura Pimpinella. An event driven smart home controller enabling consumer economic saving and automated demand side management. *Applied Energy*, 96(0):92 – 103, 2012. ISSN 0306-2619. doi: http://dx.doi.org/10.1016/j.apenergy.2012.02.024. Smart Grids.

[7] Alessandro Di Giorgio and Francesco Liberati. Near real time load shifting control for residential electricity prosumers under designed and market indexed pricing models. *Applied Energy*, 128(0):119 – 132, 2014. ISSN 0306-2619. doi: http://dx.doi.org/10.1016/j.apenergy.2014.04.032.

[8] Arduino. Arduino - home. http://www.arduino.cc/.

[9] RASPBERRY PI FOUNDATION. Raspberry pi. http://www.raspberrypi.org/, .

[10] Raspbian. Raspbian: Frontpage. http://http://www.raspbian.org/.

[11] Pidora. Pidora - raspberry pi fedora remix. http://pidora.ca/.

[12] Arch Linix ARM. Raspberry pi - archwiki - arch linux. https://wiki.archlinux.org/index.php/Raspberry_Pi.

[13] Python. About python — python.org. https://www.python.org/about/.

[14] Oracle. Java software — oracle. https://www.oracle.com/java/.

[15] cplusplus.com. A brief description. http://www.cplusplus.com/info/description/.

[16] Nurzhan Nurseitov, Michael Paulson, Randall Reynolds, and Clemente Izurieta. Comparison of json and xml data interchange formats: A case study. 2009.

[17] RASPBERRY PI FOUNDATION. Setting up an apache web server on a raspberry pi. http://www.raspberrypi.org/documentation/remote-access/web-server/apache.md, .

[18] The PHP Group. Php: Hypertext preprocessor. http://php.net/.

[19] The Apache Software Foundation. Welcome to the apache software foundation! http://www.apache.org/, .

[20] Simon Tatham. Putty: A free telnet/ssh client. http://www.chiark.greenend.org.uk/~sgtatham/putty/.

[21] WinSCP. Free sftp, scp and ftp client for windows. http://winscp.net/.

[22] Notepad++. About. http://notepad-plus-plus.org/.

[23] Google. Google-gson. https://code.google.com/p/google-gson/.

[24] Pi4J. The pi4j project. http://pi4j.com/.

[25] Joda. Joda-time - java date and time api. http://joda.org/joda-time/.

[26] The Eclipse Foundation. Eclipse luna. https://www.eclipse.org/, .