Distributed Control Applications Using Local Area Networks

A LAN Based Power Control System at Rhodes University

Submitted in fulfilment of the requirements for the degree of

MASTER OF SCIENCE

Of

Rhodes University

By

Anthony John Sullivan

December 2001

Abstract

This thesis describes the design and development of both the hardware and software of an embedded, distributed control system using a LAN infrastructure for communication between nodes. The primary application of this system is for power monitoring and control at Rhodes University.

Both the hardware and software have been developed to provide a modular and scalable system capable of growing and adapting to meet the changing demands placed on it.

The software includes a custom written Internet Protocol stack for use in the embedded environment, with a small code footprint and low processing overheads. There is also Linux-based control software, which includes a web-based device management interface and graphical output.

Problems specific to the application are discussed as well as their solutions, with particular attention to the constraints of an embedded system.

Acknowlegements

I would like to acknowledge the help and support from the following people, including their infinite patience.

- Prof. J Jonas - For his advice and guidance in this project as my supervisor.
- FF. Jacot-Guillarmod - For his help in configuring the DHCP server and other network related issues.
- J. Mackay - For his help in organising many of the parts needed for this project.
- Viv James & Frank Ellis (Trax Interconnect) - For their help with the CAM files and PCB manufacture.
- Richard F. Man (Imagecraft) - For his help in tracking down some of the bugs and quirks of the ICCAVR compiler.
- The Power Control Committee - For enabling me to undertake this project.

Finally I would like to thank my friends, who sometimes did not see me for days at a time, for making me take the occasional break.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the past, commercial data acquisition and control systems have typically been based on costly microcomputers, and more recently on personal computers. In recent years the electronics industry has seen major advancements in the state of the art, particularly the area of inexpensive, fast and feature-rich microcontrollers. As an example the newly available Atmel ATmega64, a RISC-based microcontroller in the same family as those used in this project, runs at 16MHz with a throughput of up to 16 MIPS. These inexpensive devices have seen the integration of what used to be external peripherals such as ADC's, timers and RAM, into a single package. This has resulted in reduced component counts on boards, with an increase in versatility, which has enabled the introduction of microcontrollers into applications previously not explored.

This project deals with one such control application: the control of non-essential electric heating devices on the Rhodes University campus, or the so-called Power Control project. While the original design for the system was for a generic control system, this was refined and redesigned to accommodate the more specific needs of a power control system.

1.1 History & Motivation.

The prime motivation for this system is cost savings. Rhodes' main electricity supply is via two sub-stations situated on campus, which are supplied directly by the muncipality. Rhodes is charged for consumption as well as peak demands above the specified maximum expected. It is the charges from these peak demands that contribute most to the costs incurred by Rhodes.

If the demand for electricity could be cut during these peak periods then it is possible to reduce electricity costs dramatically. Figures supplied by the Estates Division indicate that savings of between R 100,000 and R 500,000 annually could be achieved. With the current economic climate

and the cut-backs in tertiary education funding, the possible savings generated could be used in other areas of the university where improvement is needed.

The system would also provide the electricians with a real-time view of the power consumption on campus and allow manual control of boilers and other systems, thereby increasing their productivity.

The system's basic function will be to monitor the overall consumption at the sub-station and then shedding the load should the consumption approach the threshold. This will be done by switching off power circuits in buildings that have non-essential heating equipment attached to them. Most of the buildings on campus already have seperate circuits for heaters and the new residences have taken this into account during the design phase.

1.1.1 The previous Power Control System at Rhodes

Power Control has proven to be a contentious issue at Rhodes in the past, with the various parties involved not seeing eye to eye. The origins of the previous system are only available by word of mouth and there is no documentation available on the functionality or architecture of the system. Initially I considered reverse-engineering the previous system's hardware and software to get a clearer picture of its functionality, but after some consideration it was decided to design a new system from the ground up. After initial investigations of the hardware and interviews with people who had dealt with the previous system several issues became apparent:

- Reliability: The communications channels were prone to malfunction, since they used twisted pair cabling as the medium between buildings. This is the primary example of the advance of the state of the art. When the previous system was designed, fibre-optics were still new and not widely available.
- Technology: The previous power control system had used solid-state relays to control individual heaters. These solid-state relays had the unfortunate trait that one of their common modes of failure were to go into half-wave rectification of the mains supply. This proved to be hard to detect since the heaters would still turn (half) on.
- Human Interaction: There was no easy means of human interaction with the system. While there were status indicators on the FEPs (see Glossary for all abbreviations and definitions), the only other means of interaction was the PC that was running the control software, which was not connected to a network.
- Failure Notification: Almost all of the system failures were only detected at night, even if they occurred during the day, which resulted in an electrician being called out.

- Responsibility: As far as I could figure out, no one in the maintenance division was willing to take responsibility for the system, often indicating that it was someone else's, or some other division's responsibility.

These problems would have to be addressed in the new system or it would suffer a similar fate of falling into disrepair. In short these problems were dealt with by the carefull design of the system. The only problem that remains is the responsibility issue, but with clear boundaries between network infrastructure, power control electronic hardware and high voltage electrical hardware this will hopefully not cause much trouble. Later sections deal with the issues in more depth.

1.1.2 More than just Power Control

While the main objective of this project was to develop a replacement Power Control System, a secondary need became apparent. Many of the scientific research departments need to monitor and control various experiments, and while commercial equipment is available, it is prohibitively expensive. While it is possible to tailor individual solutions for the various tasks, this results in discarded equipment gathering dust in the corner of some laboratory. If a standardised control environment (consisting of hardware and protocols) could be instituted then it would be possible to re-use the same hardware for different applications, at a fraction of the cost of commercial systems.

Gradually during this dissertation it will become apparent that care has been taken in the design so as to ensure that the system could just as easily be used for monitoring and controlling some process variables of an experiment. Obviously the control decision algorithms would have to be tailored for the particular experiment, but it would use the same (modular) hardware, which is the only expense, in an academic environment where labour is 'free'.

During the early stages of the project, I developed and implemented sensor and control instrumentation for the Biotechnology department, who needed some means of measuring Redox-potentials and PH-levels of an experiment and controlling input rates of solutions via a pump. While the hardware and software initially developed for them was a far cry from the hardware and software used in the final version of this Power Control project, that same project could be undertaken today, using the power control hardware (with only modifications made to the FEP (Front End Processor) to interface to the necessary sensors) in a fraction of the time.

1.1.3 Making Technology Useful

The other prime motivation for a distributed control system on Rhodes campus is the unification and proper utilisation of IT related resources on campus. The extension of the campus-wide LAN has opened up non-academic possibilities.

- Access Control: With the network present in residences it would be possible to enable network based security over the LAN. This is already happening in the new Computer Science / Information Systems Building[1] . The hardware from the Field Controller is planned to be used in this application. (Since it is cheaper than the commercial alternatives.)
- Dining Hall Meal Delivery: This has been implemented in the Rhodes Dining Halls and while it does not use any of the hardware covered by this project it does use similar technologies.
- Class Attendance Registers: For the larger classes it would be simpler to log attendance as students enter the door using the students' Dallas iButtons, rather than sending an attendance register around the class. It would then be possible to make attendance statistics available over the web and send automated reminders to students via e-mail. A rudimentary system was built for the IS department, and while not network enabled, it will hopefully prove useful and encourage the development of the system.
- Monitoring of security guards: It is also proposed that the a system based on Dallas iButtons could be used to monitor the routes of the campus security guards and log their movements.

1.2 Is this a Real-Time System?

It should also be noted that while many would regard the system described in this thesis as a *real-time* system, it is in fact not. Even though there is much debate in various public forums about the definition of a *real-time* system, and this system does meet several of the commonly accepted criteria, I would hesitate to describe it as such.

A Real-Time system has to fulfill in a (timely) predictable way to unpredictable external stimuli arrivals. In short, a Real-Time system has to fulfill the following under extreme load conditions:

- Timeliness: It has to meet deadlines. It is required that the application finish certain tasks within pre-defined time boundaries.

[1] Hamilton Building

- Simultaneity: More than one event may happen simultaneously and all deadlines should be met.
- Predictability: The real-time system has to react to all possible events in a predictable way.
- Dependability: It is necessary that the real-time system environment can rely on the electronics of the system.[DSE]

While the tasks executed are finished within certain time constraints, these are relatively flexible and the system is designed in such a way that if a task is not able to execute because another task is blocking a certain resource, it is rescheduled.

The system would be better described as a 'Distributed, embedded, fault-tolerant system' as described by the terms given in the glossary. It should be noted that the term 'dedicated' is omitted, since 'embedded' implies that the system is dedicated to one task and this system in fact is not dedicated since it can also be used to control sprinkler systems among others. It is also my personal opinion that 'fault-tolerant' should be omitted, since any well designed system should be able to detect a malfunction and/or fault and accommodate it and not cease operation. To illustrate my point consider the operating systems Windows 95^{TM} and any Unix based OS. Whereas Windows^{TM} would produce the dreaded '*blue screen*' on a critical error, the Unix-based OS would perform a core dump and continue operation. (The Unix-based OS can be considered the fault-tolerant system.)

1.3 Distributed Systems

With any truly distributed system, communications form an integral component of that system. The physical method of communication will depend on the amount of data, distance and environment through which the data must be transmitted. A distributed system may make use of a logically transparent external system for its communication. An example of this would be a security system using a LAN as a transport for data. The LAN is an entirely independent system that effectively has no knowledge of the security system and the security system, in turn, has no knowledge of the LAN except that it transports its data between logical points in the security system. Further examples of distributed systems include most plant control systems, where data is gathered from a wide range of sensors that are geographically distributed. One of the larger examples being Eskom's distribution network, where data is gathered nationwide and relayed to the control system that controls the power generation plants. In my initial stages of research I tried to contact Eskom to get information on their data communications systems used to relay consumption data, but they were not forthcoming with any information.

Most distributed industrial control systems use a system called SCADA (Supervisory Control and Data Acquisition) which can operate over a variety of communications media, but these systems are typically expensive. This has emerged as one of the leaders in distributed industrial control over recent years, but obtaining the specifications proved futile, as with most proprietary protocols in the electronics and computer industries.

Chapter 2

System Overview

This project originally began as a generic distributed control system, that had to fulfill the following criteria:

- Low unit cost: The use of expensive hardware was ruled out due to the fact that in an acedemic environment money is frequently a major issue, not so in the industrial sector.
- Ease of use: The system should not be overly complicated, but intuitive and simple to operate.
- Modular: The hardware should not be monolithic, but rather modular in nature to ensure easy repair and upgrading.
- Scalable: The system should be able to grow, without changes to the architecture or operation.

While is easy enough to design a system that is modular and scalable, it becomes harder to design when the ‘ease of use’ and ‘cost’ criteria are factored in. The ‘ease of use’ criteria presented the most difficulty, since this covered installing and configuring hardware, which should really be done by a skilled technician. The various problems arising from these often conflicting design criteria are covered in the sections of this dissertation were they were dealt with.

2.1 Hardware Choices

With the vast array of microcontrollers currently available in the electronics sector, the choice of processor could quite easily have become a major stumbling block. The first decision was whether or not to focus on the higher-end 16 and 32 bit processors or the lower-end 8-bit processors. This choice was relatively easy to make because the 8-bit processors were significantly cheaper, while still having enough computing power for the application at hand. My choice of the Atmel AVR range of 8-bit Flash-based microprocessors was made for the following reasons:

- Low unit cost: The processors chosen were in the price range of R50 - R80.
- The future: Atmel has in the past shown that while it may stop producing a particular IC, there is always a pin-compatible replacement with more features and greater power. (For instance, the replacement for the ATMega103 is the ATMega64, which is capable of running at 16 MHz).
- Availability: They are readily available from local distributors.
- Past experience: I had previously used this range of products and was familiar with their capabilities.
- Power: While I would have like to use one of the ARM-based processors, they would have been overpowered for this application, but if the project were to be expanded these processors may become more suited.

The choice of Ethernet controller had already been made in my honours project, after exhausing all other possibilities.

2.2 System Outline

The distributed control system logically consists of three types of nodes as seen in figure 2.1:

Front End Processor: This is the data acquisition section of the system, as well as the actual control interface. While it takes no part in the control decisions of the system it does play an extremely crucial role, besides data acquisition. The FEP processes the raw data, presenting only processed information to the higher level devices. This means that a standard interface and data format is used by the controller even if the method of data acquisition changes, since the FEP will process this data in a different manner to present the information in the standardised format. This also means that any information can be passed on to the higher levels and used in control decisions. This enables easy reconfiguration of the FEP without having to make any changes to the rest of the system. The logical layout of the FEP can be seen in figure 2.2.

Field Controller: This node continuously updates a data structure representing the physical state of the FEP nodes below it in the heirachy. This data structure is then used as an input for the control algorithm. If the communication channel to the master controller is non-functional, a local control algorithm is instituted for all FEPs below this node. Under normal operation this device will pass the data structure to the master controller and relay instructions from the Master Controller to individual FEPs. The logical layout can be seen in figure 2.3.

Master Controller: This node handles the control of the entire system under normal operating conditions as well as providing a human interface. While no routine human interaction is needed, there may be times when direct human control of the physical system is desired. This node also generates human readable status reports and sends system error notifications should part of the system fail. The logical layout can be seen in figure 2.4.

These three types of nodes represent the logical layout of the system, but fail to provide an adequate physical representation. While these three types of nodes are in fact physically separate they are accompanied by other physical elements that do not form part of the logical model. The other physical components of the system include the data distribution infrastructure. This consists of the ethernet cabling, serial data cabling and signal distribution hardware. The signal distribution hardware provides an interface between the different physical media used for serial communication (fibre-optics and UTP copper cabling) and also supplies and distributes power for the various system nodes.

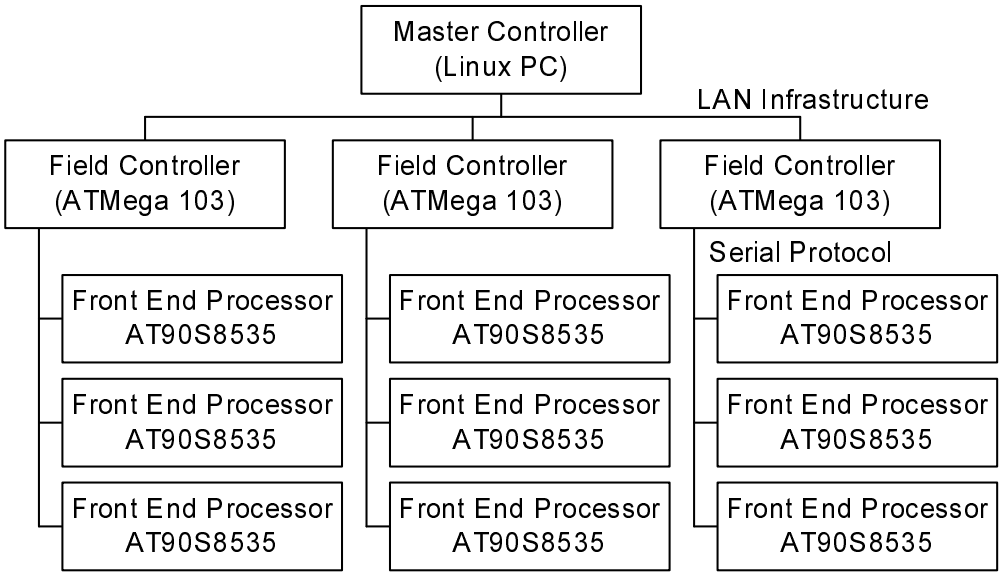


Figure 2.1: Logical Layout of the overall Distributed Control System

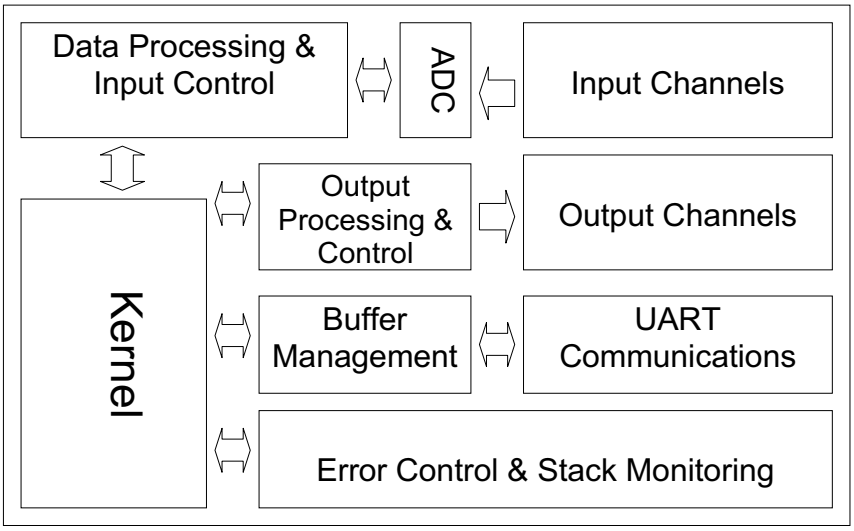


Figure 2.2: Front End Processor Logical Layout

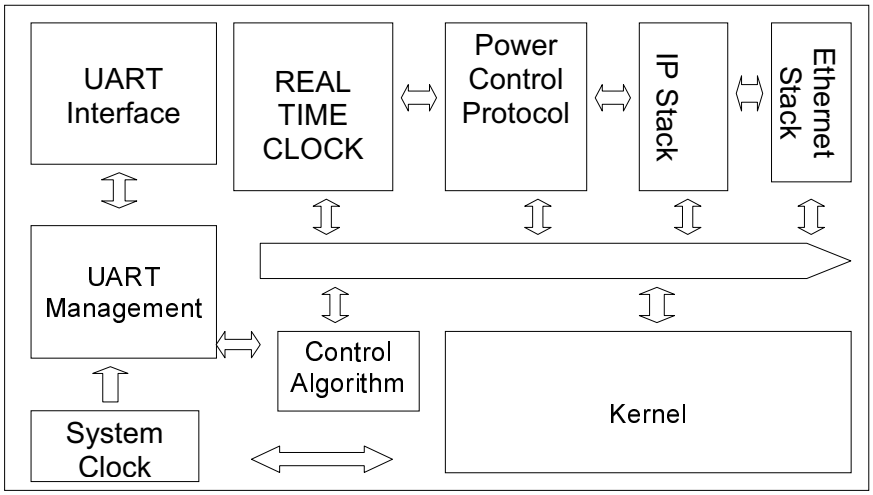


Figure 2.3: Field Controller Logical Layout

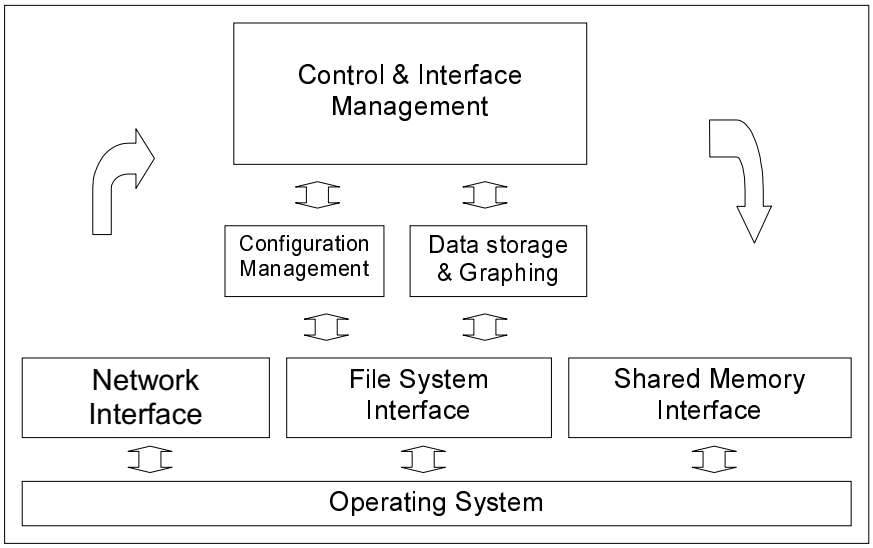


Figure 2.4: Master Controller Logical Layout

2.2.1 Physical Overview

A simplified overview of the physical layout is shown in figure 2.5. The power distribution ring is shown between the sub-station and several residences, as well as a communications channel.

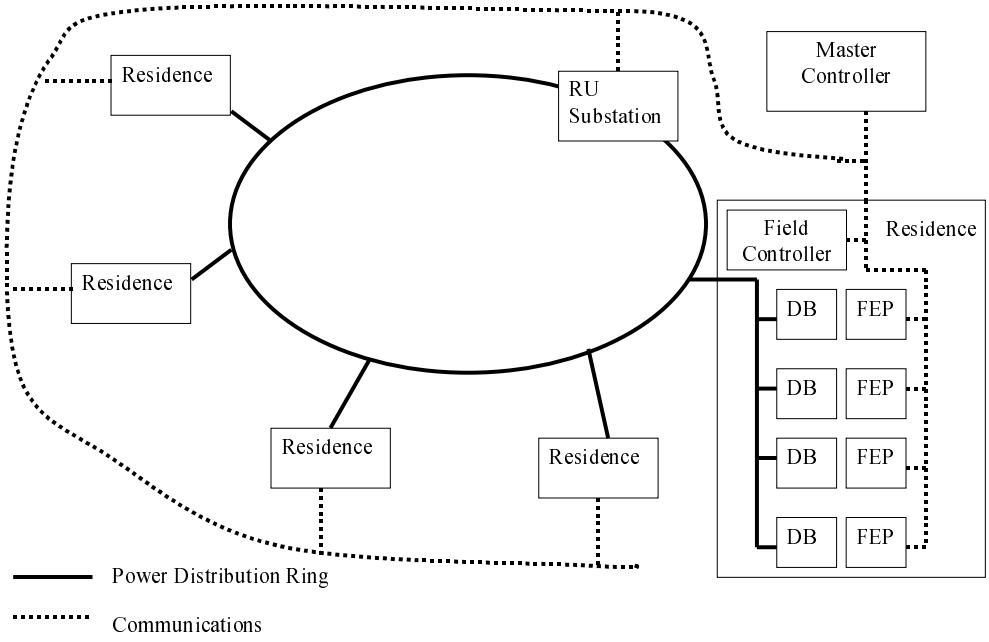


Figure 2.5: The abstracted physical layout for the system

Firstly, the communications channel is shown as a continuous, unbroken channel. This is done to illustrate a point, the *RU Substation* can in fact be connected to a field controller in one of the residences via serial communications over fibre. It does not logically make a difference (from the Master Controller's point of view) whether each residence has its own Field Controller or not.

2.2.2 Serial Communications

The selection of the physical channel used for serial communication in a system is usually dependent on the data bandwidth, distance transmitted and type of environment. This means that it is not always possible to use the same media throughout the system. This is the case with the power control system. In the logical model there is a direct link between the Field Controller and the FEP, but in the physical model there is a distribution board that divides the link into two different media types.

Fibre: Fibre was needed to eliminate any direct electrical connection between buildings to avoid the dangers associated with ground loops and . One of the faults of the old power control system was the unreliability of the communications channels due to damage from differences in ground potential. With the use of fibre between buildings there is no danger of this problem. Using fibre also reduces the probability of errors being induced onto the transmission line by the electrical interference.

Twisted Pair Cabling: While in theory bell-wire would suffice for distributing the power and communications channels to the FEPs, twisted pair cabling was used with differential mode signaling to reduce the possibility of errors being induced by the environment. Standard Ethernet cat 5 cabling was used to distributed the signal so that the cabling could be used in future upgrades to the system should the need arise to have ethernet connectivity directly to the FEPs as well as keeping a consistent cable plant.

2.2.3 Network Communications

The entire Network communications hardware is built around the CS8900A[1] Ethernet controller. A scaled down IP (Internet Protocol) stack was implemented to enable basic communication over an ethernet network. While there are existing protocols for information appliances, there is not an emerging standard at this point in time. Hence instead of implementing a protocol that may cease to be supported, a decision was made to use a simple messaging protocol implemented using UDP (User Datagram Protocol). Issues concerning the hardware and software development of the network communications will be discussed in the appropriate sections.

2.2.4 Data Acquisition & Processing

The actual data acquisition occurs in the lowest level of the structure (in the FEPs). The physical data acquisition is done via the on-board Analogue to Digital Converter (ADC) on the Atmel 8535. It is also possible to acquire digital data via the Serial Peripheral Interface (SPI) port from an external transducer, thus allowing further applications for the system. As far as the acquisition of the power consumption is concerned, at this initial stage of the implementation, a simple RMS measurement of the current supplied by the mains distribution board is made using current transformers.

[1] Cirrus Logic (See appendices)

2.2.5 Control Implementation

The control can occur at two different levels, either in the field controller or the master controller. The control implementations are discussed in the relevant software sections, but work on similar principles.

The control algorithms take a list of possible output devices for any given input device, this is set in configuration files. These output devices' state should affect the state of the given input device. Along with these parameters, an upper and lower threshold are given as to the desired state of the input. When below the lower threshold the system does not attempt to alter the outputs, when the upper threshold is reached more outputs are switched off. While between the two thresholds, the outputs are changed in a round-robin style.

Considering figure 2.5. Under normal operation the Master Controller will receive data from the current sensors in the sub-station. It will then control loads in each of the residences based on the control rules given. So if the Master Controller detects that the current being drawn at the sub-station exceedes the threshold set, it would then start to switch individual controllable loads off in the various residences until the load had been reduced.

If for some reason the Field Controllers lost communications with the Master Controller, they would institute local control. The Field controllers would then use the data from the FEPs connected to the current transformers in each residence as inputs for their own local control. Each Field Controller would then be responsible for keeping the local power consumption below some pre-defined threshold.

It should be noted that normal theoretical control theory methods are not easily applicable in a power control application since the demand is not something that can be affected by turning off a contactor, since loads might be switched on or off independently, making the demands random, which in essence makes the system unpredictable. While the control algorithms use only two possible output states, with very little effort it would be possible to make the system control variable speed motors, since the data is always treated abstractly, until it reaches the control algorithm.

2.3 Commercial Products

Over the preceding 18 months the number of commercially available embedded network products has risen dramatically. As with most commercial systems the major drawback to these is their proprietary nature and their cost. While the products mentioned below were rejected, they are mentioned to provide an overveiw of what is commercially available.

- **AVR Embedded Internet Toolkit:** This product features the same chips as used in this project (CS8900A & ATMega103), but comes with a hefty price tag of R3000. There is a TCP/IP stack included, but only the object files are distributed, making it impossible to make adjustments to the IP stack.

- **Ceibo TCP/IP for the 8051:** This offering from Ceibo is based on the 8051 microcontroller with the TCP/IP stack costing between $1500 to $5500, depending on the options. While the 8051 architecture has seen advances from when it was first released, it is still essentially a CISC based microcontroller and in my opinion is outclassed by the Atmel AVR range of microcontrollers.

- **CMX Systems CMX-MicroNet TCP/IP stack:** Unfortunately I could not find much information on this implementation except that it does not require an RTOS.

- **Netsock 100:** This is a small embedded PC with a built-in TCP/IP stack and RTOS. Again the code sources are not distributed, only the API for the networking functions.

- **Embedded Linux:** While this is only the OS, it does provide more power and versatility than any of the other options. The bonus is that there are versions of Embedded Redhat Linux available for many ARM based processors. Many are even available on development boards such as the Cirrus Logic EP9312, a 200MHz RISC processor. Most of the ARM development boards cost in the range of R2000-4000, while Linux is free.

- **PICDEM.net**[2] : This is a small development board offered by Microchip. This development board features a PIC18C452 / PIC16F877 with a RTL8019[3] Ethernet controller; and is supplied with a TCP/IP stack as well as a web-server. Unfortunately the PIC family of microprocessors is not as powerful as the Atmel range (some people may disagree) and there is a limited amount of RAM available.

[2] Ref No DS51240A, Microchip Technology inc.

[3] Realtek, The RTL8019 is a popular chipset in older ISA/PCI 10BaseT network cards. (Also used on some Genius^{TM} Network adapters.)

All of these products have the disadvantage that they are expensive and hence oppose the first design criterion mention earlier, that of cost. Therefore it was decided to develope a solution from the ground up, as described in section 2.1.

If a high power, simple solution were called for, I would have to recommend Embedded Red Hat Linux running on an ARM-based platform, which could provide more than enough processing power.

Chapter 3

Hardware

Issues regarding the hardware design of each section of the system are discussed in this chapter. It should be noted that hardware development occurred concurrently with the software development in the initial stages. At this stage all hardware prototyping was done on protoboard to enable design changes should problems have occurred. Hardware development was also done on the various components of the system concurrently to ensure their correct interaction.

3.1 Front End Processor Hardware

The FEP hardware has been through many design iterations, from the initial concept of using SPI as a means of connecting the FEPs to the control processor to the final RS-422 serial interface. It would be possible to re-engineer the entire system to use SPI in the layers below the Field Controller without changing any of the master control processor's software, since the lower layers are totally abstracted as far as it is concerned. As long as the required data format in the UDP datagram is observed the modification would be transparent.

The FEP is designed around the Atmel AT90S8535. Before the design phase could begin, I had to fully aquaint myself which its architecture by studying the product data sheet [Atmel 2]. The basic hardware sections of the FEP can be seen in figure 3.1, with the full schematics and PCB layouts available in the appendices and the complete design files on the accompanying CD-ROM.

3.1.1 Initial SPI Hardware Design

The first attempt at designing the FEP was based on the assumption the there would be at least one control processor per geographical location, but budgetary constraints and changes in the policy of extending the Rhodes campus Ethernet backbone meant that this was not feasible. This was due to the fact that the control processor had to have control of the selection of the SPI slave, which meant

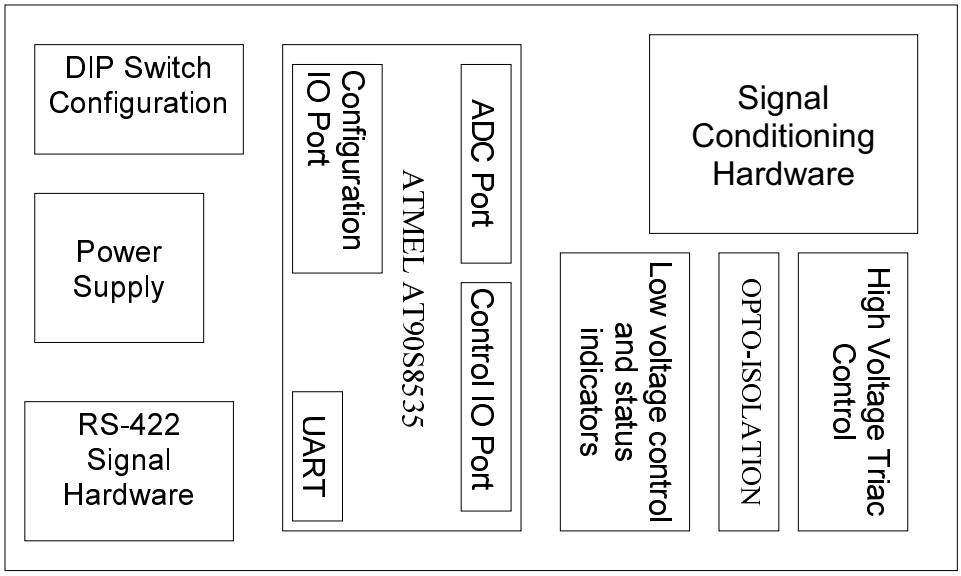


Figure 3.1: FEP Hardware components.

that it would have to be part of the signal distribution hardware. This design was implemented on protoboard and tested before it became apparent that it would not be feasible. Even though this design was scrapped, it allowed a greater understanding of the SPI protocol and how important it is to modularise a system, to allow changes to a section or module without affecting the rest of the system. The actual data acquisition side of the system remained virtually unchanged due to the modular approach that was taken, the only legacy of the SPI version is in the name of some of the variables in the FEP source-code. The other drawback to the SPI system was that it used an unbalanced 5 volt signaling system with a clock signal, and while this design was scrapped before proper implementation, it was doubtfull as to whether the physical component of the SPI protocol would be able to stand up to the harsh interference environment associated with the heavy current electrical systems that the hardware would be exposed to. The original SPI design would have been more suited to a small laboratory control system. This was the first of many cases in development where the physical nature of the control application forced design changes from the original conceptual model.

3.1.2 Analogue Front End

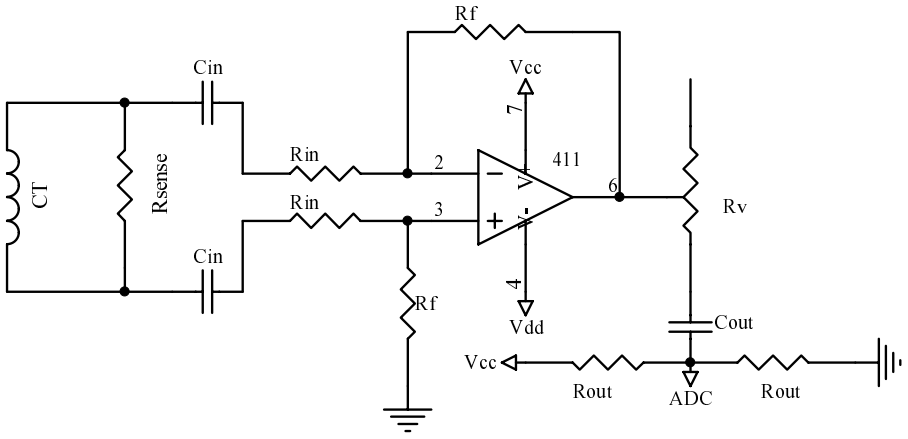


Figure 3.2: FEP Frontend

This circuitry was adapted from an earlier project undertaken for the Biotechnology Department, where I used a microprocessor to collect data from various sensors attached to the system and sent the data obtained to a PC connected via an RS232 serial link.

In figure 3.2 it can be seen how the current transformer is loaded with a resistor and the voltage across this resistor is then amplified by the operational amplifier (acting as a differential amplifier) after it has had any DC offset removed by the capacitors C_{in}, while this is not strictly necessary, it was included to err on the side of caution. This design is an adaptation of the classic differential amplifier [HI&H, Figure 4.18 page 185]. The output of the amplifier is given by the equation:

$$V_{out} = \frac{R_f}{R_{in}} * V_{in} \tag{3.1}$$

Where V_{in} is the voltage developed across the 100 ohm (R_{sense}) resistor by the induced current in the current transformer. Thus the relation between the output voltage and the current being measured is:

$$V_{out} = \frac{R_f}{R_{in}} * R_{sense} * CT_{ratio} * I_{meas} \tag{3.2}$$

The input capacitors can be removed from the circuit and replaced with wire links on the PCB. The output of the operational amplifier is then sent through a potentiometer so that the signal can

be attenuated, which allows for hardware calibration. The signal is then yet again passed through a decoupling capacitor which allows the two remaining resistors to bias the signal to half of the ADC range. This signal is then fed to an ADC port on the Atmel 8535. (A full schematic of the FEP can be found in the appendices.) Care was taken to minimise noise from the processor during the schematic design through the use of smoothing capacitors at the voltage regulators as well as at all the integrated circuits.

3.1.3 Triac Drive Circuitry

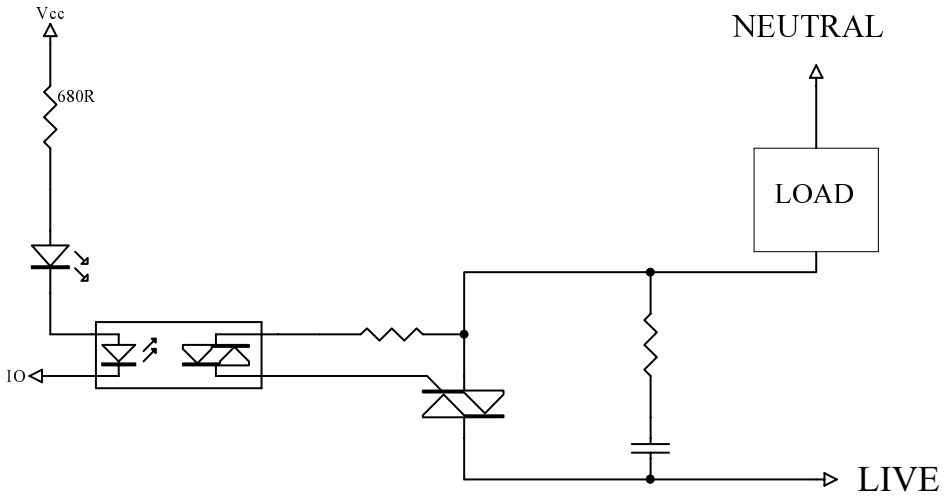


Figure 3.3: Triac Drive Schematic

An arrangement of opto-isolated triacs and high current triacs are used to drive the contactors necessary to switch the high currents present in the mains voltage distribution board. An extract from the FEP schematic is shown in figure 3.3.

On the low voltage side a LED is used as an indication as to the status of the output, to enable a visual verification of whether the FEP is working correctly. Using the opto-isolated triac allows a high degree of isolation between the high voltage power electronics and the low voltage control electronics. (According to the the data sheets available the opto-isolated triac should provide isolation for a peak AC voltage of about 7000 volts for one second.) [Fairchild]

Unfortunately the design of the RC snubber circuit is not precise and most documents that I could locate indicated that the values for R & C in the snubber circuit are chosen using a trial and error method. However as a guideline the valuesare typically in the order of R=33 ohms and C=0.01uF. In the final prototype a snubberless triac[1] from ST Microelectronics was used, thus minimising the space required on the PCB.

3.1.4 SPI Sensor Interface

Provision was made for four SPI sensors to be added to the FEP. This was primarily to make the design future proof. Since the number of types of sensors with an SPI interface has increased dramatically over the past two years it is possible to find an SPI sensor for virtually any application. This should also allow the FEP circuit boards to be used for different functions, other than current sensing, such as measuring temperature and humidity in boiler rooms or water pressure. As can be seen from the FEP schematic in the appendices, the data lines are set up in a wired-OR configuration for both active high and active low data lines. This allows the SPI interface to be populated based on the specifications for the SPI sensor that is attached. This was done due to the varied nature of the interfaces that I have encountered in the past.

3.1.5 Overall FEP Design

The design criteria for the FEP are that it has to read in voltages from the ADC ports, present this data to the control processor when queried, and control the output channels connected to high voltage contactors via triacs. Apart from the software and PCB design issues, the only other points that were taken into consideration were the support electronics. A dual (positive and negative) 5 volt supply voltage was needed for the op-amps, these supply voltages were also used for the analogue section of the 8535. A separate 5 volt supply was used for the digital electronics to minimise noise present on the ADC channels. The original FEP was also equipped with place for both RS232 and RS422/485 transceivers. This was a generalisation made to the FEP so that it could also be used for data capture directly to a PC, but this was removed for the final design since an RS232 interface was included on the Distribution Board. The power electronics side of the design presented the hardest challenge, since the actual design could not be breadboarded and tested due to the dangers presented by working with mains voltages and the high currents involved in the triac drive circuit for the contactor. This part of the design was based mostly on application notes and reference designs found on the internet and advice from Dr. Jonas. The design for the FEP may seem simplistic, but this is because all the electronics present are merely for signal conditioning and microcontroller support. The code running on the microcontroller provides the functionality.

[1] BTA10-600BW

The DIP-switch present in the design is used to manually set the identification number of the FEP. This allows totally generic hardware to be used and allows non-specialist personnel to to effect field replacement of the FEP (in case of malfunction) from a pool of identical replacement parts by only setting the DIP-switch to correspond to the device ID of the FEP being replaced. While it would be possible to include the device ID in EEPROM, which could be reprogrammed by a common Keyfob[2] programmer, this would mean that the personnel replacing the device would require some sort of technical training regarding the setup of the EEPROM files.

The final schematic for the FEP can be found in the appendices, while the Protel design files can be found on the accompanying CD-ROM.

	Current [mA]
Processor	$\approx$ 10 mA
1 x Power LEDs	$\approx$ 10 mA
3 x output channels	$\approx$ 30 mA
411 supply current (x3)	$\approx$ 15 mA
MAX 488 (average)	$\approx$ 20mA
Other	$\approx$ 15 mA

Table 3.1: FEP Power Consumption

The FEP draws about 100mA when in operation, this can be broken down as shown in table 3.1.

3.2 Distribution Hardware

The distribution hardware exists only to provide conversions between the different data transmission media and to enable multiple FEPs to be connected to one Field Controller. It functions similarly to an ethernet hub, with one small difference; the devices connected to it can only communicate with the master (the Field Controller). This means that each FEP is totally unaware of the existence of any other FEPs (on the hardware level). The basic components of the distribution hardware can be seen in figure 3.4.

3.2.1 Fibre Optics

Agilent fibre transmitters (HFBR-1414T) and receivers (HFBR-2412TC) were used as the optical transmitters and receivers. These both feature the popular ST series bayonet connection used by

[2] See AVR Keyfob.pdf on CDROM

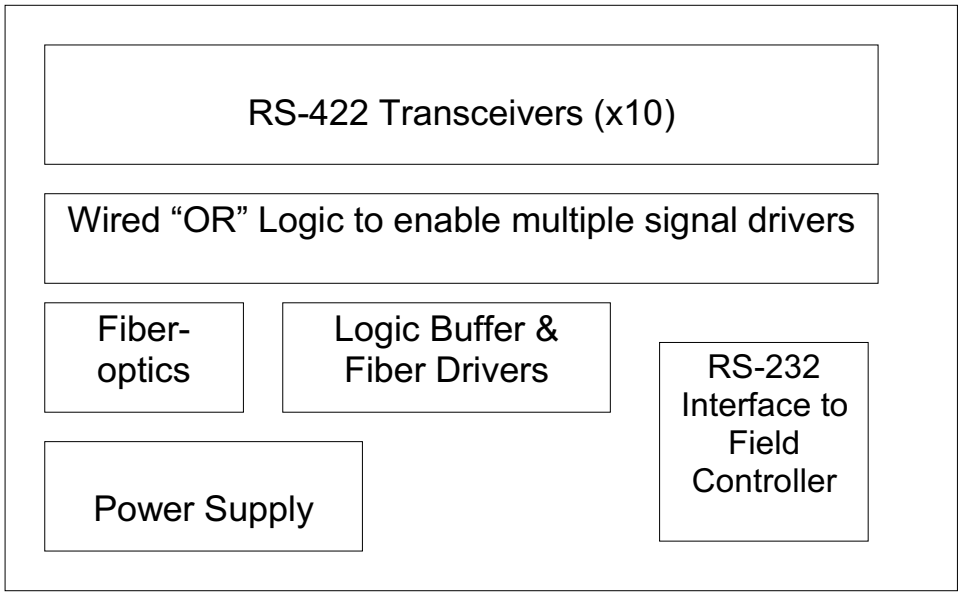


Figure 3.4: Distribution Board Hardware

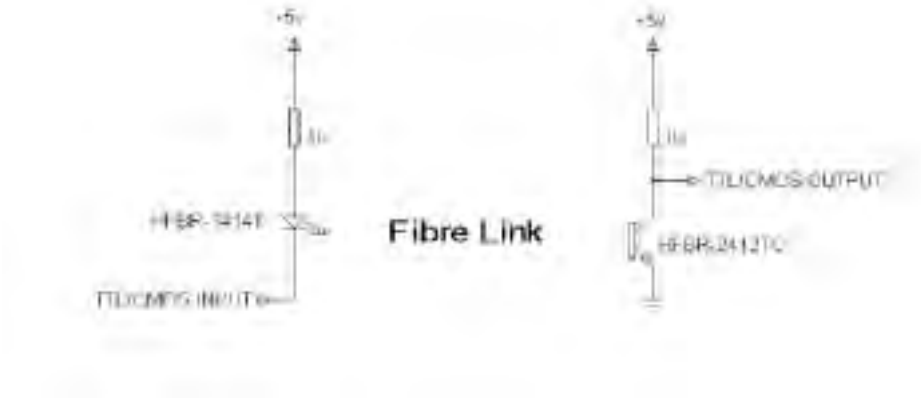


Figure 3.5: Fibre Drive Schematic

Rhodes network devices. Since the signals sent are neither modulated analogue or high frequency digital signals no expensive driver chips or complex circuits were used. The fibre optic transmitters are driven with the TTL/CMOS output from the RS-232/422/485 transceivers and the fibre receivers drive the TTL/CMOS inputs of the RS-232/422/485 drivers directly with minimal support circuitry. The fibre transceivers are actually both driven by and drive loads through a logic buffer. This is to avoid any problems that may arise due to overloading the transceivers. The simple drive circuitry can be seen in figure 3.5.

While this design may seem simplistic for a fibre optic interface it is capable of handling what is expected of it, and if more complicated/sophisticated drivers had been used, it would have been a waste of money. The design is based on the reference design provided by Agilent [Agilent 1, Agilent Technologies' Datasheets].

3.2.2 RS-422

RS-422 was chosen as the serial interface to be used due to its balanced line signaling. This improves the noise immunity over other serial interfaces such as RS232 which don't use balanced signaling. Even though the voltages used are lower than the ± 12 volt range used by RS-232, any common mode interference is subtracted out. RS-485 was not used due to the fact that a single bus is used for both transmission and reception, which could present problems if contentions for the line arise. It was decided to totally avoid this problem by using point-to-point RS422. This has also been widely used at Rhodes in the past in the line drivers used for leased lines for internet access, with problems only arising when the length of the transmission lines were extended, but since all the RS422 devices in the power control system should not have line lengths of more than about 200m there are no problems forseen because they are in individual buildings.

3.2.3 General Considerations

When designing the hardware, care had to be taken with the 'reversed' logic used on the inputs to the RS422 driver chips. Due to the fact that many inputs from the FEPs would drive the output to the Field Controller I had to make sure that their were no conflicts when two or more MAX488 chips tried to drive the line at the same time. This was solved by using a wired 'OR' configuration for driving the input to the Field Controller interface. A simplification of the circuit can be seen in figure 3.6.

As is evident from the schematic, the the RS422 driver will only drive the 'TTL_UART_DIM' node low if a '1' (ie. a 0 volt signal on the TTLinterface) is received over the UART. This also

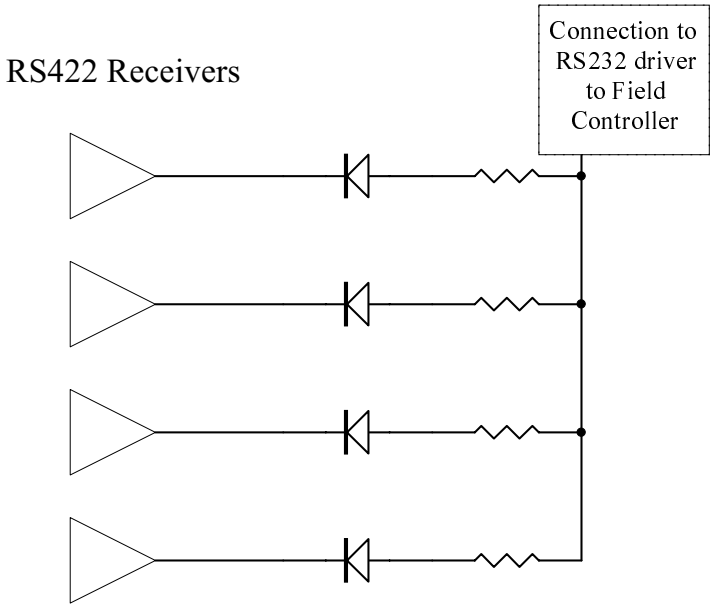


Figure 3.6: Wired ‘OR’ Configuration

ensures that if an FEP is unplugged the RS422 driver will not affect the transmissions from other FEPs since the default state for the 5-volt interface is 5 volts.

3.3 Field Controller Hardware

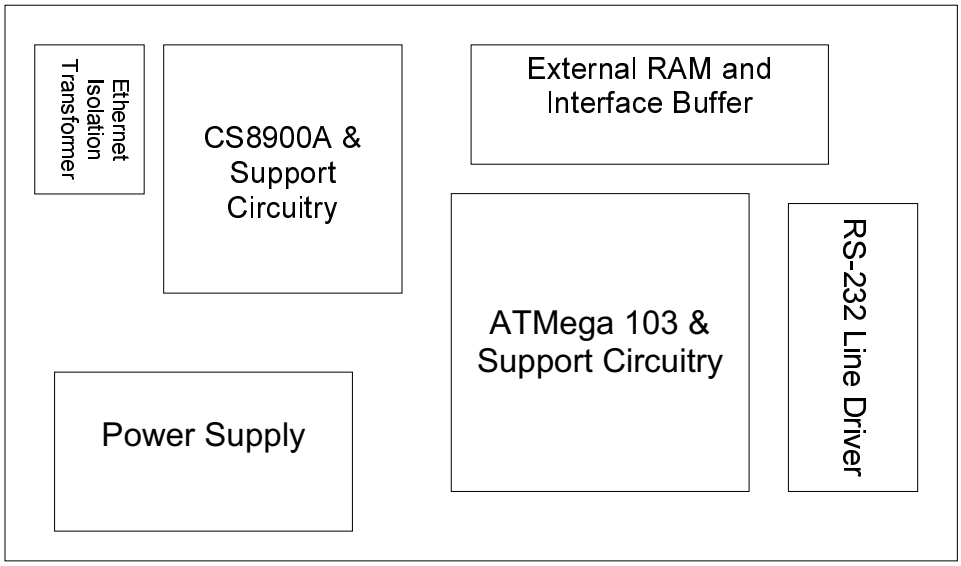


Figure 3.7: Field Controller Hardware

Designing this was the most time consuming stage of development due to the intricacies involved with the CS8900A integrated circuit. The technical documentation had to be read for both the CS8900A[Cirrus 1] and the ATMega103[Atmel 1]. The basic components of the Field Controller can be seen in figure 3.7

3.3.1 History

While I used the previous version of the CS8900A (CS8900) in my honours project[3], there were notable differences in the operation of the chip and issues that prevented me from using it in the same manner. I had previously used the CS8900 in its 8-bit mode (in my honours project) which had several issues, which continued to affect the CS8900A.

1. Interrupts were not supported in 8-bit mode, which meant that the chip had to be used in polled mode, which is both slow and not desirable in real-time applications.

2. The chip would occasionally ’hang’ when in 8-bit mode, which was both hard to detect and resulted in lost frames.

[3]Embedded Communications - Using Ethernet as a transport for embedded communications.

3. 8-bit mode could not handle high traffic situations adequately when the chip was configured to receive broadcast frames, which was necessary for a proper IP implementation.

The only way to overcome these obstacles was to use the chip in one of its three 16-bit modes of operation. These are listed below:

1. The first of these modes uses DMA (Direct Memory Access) transfers. This is a mode of operation where the chip itself initiates communication with a DMA controller and transfers the received ethernet frame to host memory. While this is acceptable in a sophisticated bus system, it is not suitable for the embedded environment in the project.

2. Secondly, there is Memory Mapped mode: This mode allows the host to randomly access the internal memory of the CS8900A to both send data and access incoming data. The only drawback to this is that 16 data lines are needed as well as at least 16 address lines and 4 controll lines. This was not practical due to the limited number of IO lines available on the processor and due to the fact that the CS8900A's default mode of operation is IO-mode and it is used to set the CS8900A up to operate in memory mapped mode.

3. The default IO mode is described in the next section.

3.3.2 CS8900A in IO Mode

When the CS8900A operates in IO mode it makes use of a proprietary system of access and data storage, the so-called Packet Page Architecture. This system allows access to almost the entire 4 of internal memory and registers by mapping them into 8 IO ports accessed via a 16-byte memory window from address location 0x0300 - 0x030E. This means that with only 5 address lines, the full range of Packet Page Registers can be accessed through these 8 IO ports. This is because address lines SA8 and SA9 (see page 12 of CS8900A Product Data sheet) are tied together and SA0-SA3 are used to address the individual IO ports. SD0-SD15 are used for word data transfers. The only other control lines needed are RESET, IOR, IOW, SBHE and the interrupt lines INTRQ0-INTRQ4. Although only one interrupt line is needed, all four are connected to the microcontroller so that interrupt priority can be set in software.

Table 3.2 shows a list of the IO ports available on the CS8900A, with descriptions below:

- **Transmit Data (Port0):** This port is used to read and write data to be sent over the network after setting up the chip as appropriate and bidding for space to transmit or after being notified of an incoming frame by the CS8900A

- **Receive/Transmit Data (Port1):** This port is not used, but may be used when 32-bit work transfers are needed. (Page 76 of CS8900A Product Data Sheet)

Offset	Type	Description
0x00	Read/Write	Transmit/Receive Data Port 0
0x02	Read/Write	Transmit/Receive Data Port 1
0x04	Write only	Transmit Command (TxCMD)
0x06	Write only	Transmit Length (TxLength)
0x08	Read Only	Interrupt Status Queue
0x0A	Read/Write	Packet Page Pointer
0x0C	Read/Write	Packet Page Data Port 0
0x0E	Read/Write	Packet Page Data Port 1

Table 3.2: CS8900A IO ports

- **Transmit Command Port:** This port is used to notify the CS8900A that the host wishes to write a frame to the buffer for transmission. (This is discussed in greater detail in the software section)
- **Transmit Length Port:** This port is used to indicate the length of the frame that the host wishes to transmit.
- **Interrupt Status Queue:** This port is used to inform the host about which event caused the hardware interrupt. It queues the interrupts that have occurred and not been read by the host, the hardware interrupt line remains low until all pending interrupts have been read from the ISQ.
- **Packet Page Pointer:** This port is used to set the location in internal memory (Packet Page memory) that the host wishes to read from or write to. The Packet Page registers used will be discussed later.
- **Packet Page Data (Port 0):** This port is used to write to or read from the Packet Page register that the Packet Page Pointer is currently set to.
- **Packet Page Data (Port 1):** This port is not used, but may be used when 32-bit word transfers are needed.

3.3.3 CS8900A design considerations.

Since a new revision of the IC was being used, (not the CS8900 that was previously used in my honours project), I decided to prototype a simple board to make sure that everything would work properly. This prototype board was designed to accommodate all the analog components needed to support the CS8900A; such as a TTL oscillator, isolation transformer and other assorted passive components, while all the IO pins were taken to header sockets for easy prototyping on breadboard.

This was most fortunate since the functions of the AEN, REFRESH and SBHE pins had changed, which resulted in many hours of debugging and scouring the technical manuals until I resorted to the errata for the manual, where the corrected descriptions of the pins were given.

Memory Location	Usage
0x0000 - 0x0FFF	Internal RAM
0x1000 - 0x7FFF	External RAM
0x8000 - 0x800F	CS8900A
0x8010 - 0xFFFF	Unused

Table 3.3: Field Controller Memory Map

Since the system was going to use external RAM, I had to design the CS8900A interface around the built-in SRAM interface on the Mega103. The other obstacle was that the CS8900A would have to operate in 16-bit mode and the Mega103 is an 8-bit microcontroller. Solving these problems could not be done in hardware alone, so the driver functions for the CS8900A had to be designed carefully to avoid access clashes with the SRAM. To get around this I had to use the standard external memory interface to the SRAM, but with the address bytes swapped, since this would not really affect the accesses to the SRAM. Next the 8 bits of the multiplexed address/data port used for SRAM access was used as one of the data ports to the CS8900A and another free bi-directional port was used to access the other 8-bit data port on the CS8900A. The highest address bit on the Mega103 was used to distinguish between whether the CS8900A or the SRAM was being accessed. This was because the SA8 & SA9 pins on the CS8900A had to be high for the chip to respond, while the CS on the SRAM was low enabled, thus allowing easy switching between SRAM and the CS8900A. This resulted in a ‘memory map’ that can be seen in table 3.3.

This memory map is not strictly true, since the CS8900A is not mapped into external memory space. It just occupies the address space. This ensures that both the CS8900A and the external RAM connot accidentally be active at the same time, which would lead to problems. To access the CS8900A, the Mega103 has to disable external memory access and then access the CS8900A through special driver functions and then reactivate the external memory access so that the rest of the software does not know the difference. (This will be discussed further in the software section.) It should be noted that the unused portion of the address space as shown in table 3.3 is actually occupied by the CS8900A, but is not valid for IO mode operation. This is due to the fact that the CS8900A is enabled by the highest bit on the address bus being in the ‘1’ state, but the remainder of the address bits would result in an invalid address. While this appears to waste valuable SRAM locations it in fact does not, since a 32kB SRAM was used and not the maximum 64kB. The extra

logic required to make better use of this available address space is not worth the cost and board space since 32kB is more than enough memory for this application. It would be possible to add address decoding logic to enable the addition of more RAM or other memory mapped extentions.

The entire CS8900A interface was designed using information from the manufacturer's datasheets and application notes [Cirrus 1], [Cirrus 2] and [Cirrus 3]. It is impossible to quote individual sections since all of the factors presented in these data sheets had to be taken into consideration.

3.3.4 Support Circuitry

The Mega103 itself requires very little in the way of support components. A 32kHz watch crystal was added so that one of the internal timer/counters could be used to implement a real-time clock. Besides SRAM there is provision for both RS-232 and RS-485/422 serial drivers, depending on the application, as well as a serial programming port for uploading code and EEPROM data. As far as power supply circuitry is concerned, there is a bridge rectifier as well as smoothing capacitors and a LM7805 voltage regulator so that the 5 volts necessary can be obtained from either AC or DC sources of up to 12 volts. Two debugging LED's were also added on spare IO lines to simplify the debugging process. These are used as status indicators in the final product. Two ADC ports were also taken out to headers in case the need for an analogue to digital conversion arises. It should be noted that the PCB shown in the appendices was originally meant to be used only for development as is evident from the large number of headers used to enable easier debugging. The PCB can actually be reduced to approximately half of its current size if internal power planes and more surface mount components were used, however this would not have facilitated easy debugging.

3.3.5 Field Controller Power Consumption

The Field Controller draws about 120mA in normal operation, which can be broken down as in table 3.4. The value given for the CS8900A is the maximum consumption based on the datasheets, since it was not possible to measure it by itself.

	Current / [mA]
Processor	$\approx$ 10 mA
CS8900A (maximum)	$\approx$ 120 mA
4 x Indicator LEDs	$\approx$ 40 mA

Table 3.4: Field Controller Power Consumption

It can be seen that the CS8900A is the most power intensive IC in the Field Controller, the somewhat high current needed can be justified by the fact that the intended application does not call for extended battery life, since it is powered from the mains supply. Total power consumption could be lowered by using 3.3 volt logic, since the two main ICs are also made for 3.3 volt operation.

3.4 PCB Design

All PCB designs were done using Protel Design Explorer 99 trial version. This software package enabled PCB design directly from the device schematics after appropriate footprints had been defined for the various components. Since all signals are of a relatively low frequency (ie less than 20MHz), no special track placements were needed.

Protel Design Explorer, in my opinion, is one of the more intuitive PCB design packages available. Having worked on both Seetrax Ranger and Tango (two of the other popular PCB design packages), I decided to use Protel's package due to its ease of use and large array of vendor libraries, that are easily available over the Internet. While initially development was slow, there was a steep learning curve which enabled me to design schematics and route PCBs in less than a day nearing the end of my project. As with most auto-routing PCB design programs, the auto-router in the Protel package proved to be next to useless. This output from the auto-router made excessive use of vias and round about routes, this output was also extremely non-intuitive, which meant that debugging the hardware / software would have been more time consuming and frustrating than it needed to be. While my routing may not be optimal, it is easy to follow and has not yielded any problems.

3.5 Board Assembly

The population of the PCBs was not too challenging since almost all of the components on the FEPs and distribution boards were conventional through-hole components. The only major obstacle faced was the population of the two prototypes of the Field Controller. This was due to the surface mount components that were used. Special mention needs to be made of the mounting of the CS8900A, with a pin density of about 2 pins/mm. Figure 3.12 shows the size of the CS8900A in comparison to a 2c coin.

The CS8900A was mounted by securing it to the PCB with a small amount of *Prestick*^{TM} and then carefully aligning the pins with the pads on the PCB. A small amount of surface-mount solder paste[4] was then applied to the corner pins and then soldered in place. After the chip was secured,

[4] Available from Maplin

the rest of the pins were then soldered. The surface mount resistors were considerably easier to mount, they were just held in place with a screwdriver and then the solder paste was applied and heated to secure the comnponent.

The original test board that was manufactured to test the CS8900A in the 16-bit configuration has since been stripped for parts. Figures 3.8 & 3.9, show the board used for most of the software development[5] and the final Field Controller[6] respectively.

Figures 3.10 & 3.11 show the final test prototypes of the FEP and Distribution Board. The uppermost output channel (the lower group of components on the right-hand side of figure 3.10) is populated with the snubberless triac. Two of the input channels have also had the input decoupling capacitors removed and replaced with wire links.

The RS422 drivers are also surface mount packages[7], but due to the manufacturing process used to make the prototypes for the FEP and distribution board, I was not able to use the correct footprint for the MAX-488 line drivers. I instead used DIP-8 footprints on the PCB and soldered the MAX-488s to DIP-8 sockets, which could be inserted into the sockets on the PCB. (Figure 3.12 also shows the size of the MAX488 IC with respect to a 2c coin.)) This was done to make sure that the design worked, before paying to get a double-sided, high presision board made. The prototype boards could only be one-sided, hence a large number of wire links were used to overcome the routing problems on these single-sided boards.

[5]Totally manually populated.
[6]Both surface mount ICs were professionally mounted.
[7]SOP-8 (Small Outline Package)

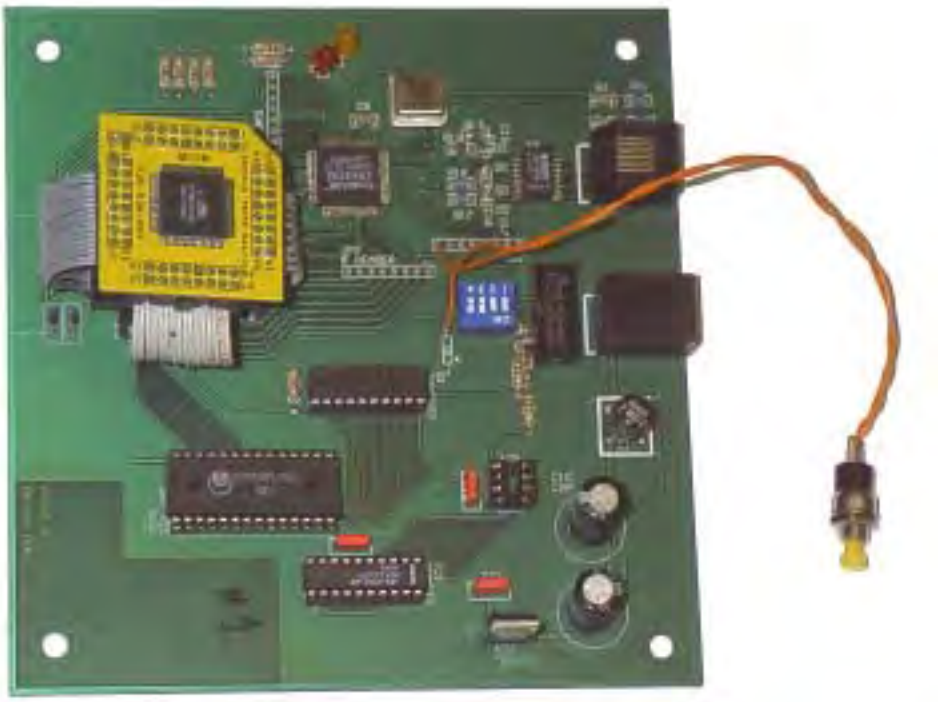


Figure 3.8: Field Controller protoype used for most of the development

Figure 3.9: Final Production model of the Field Controller

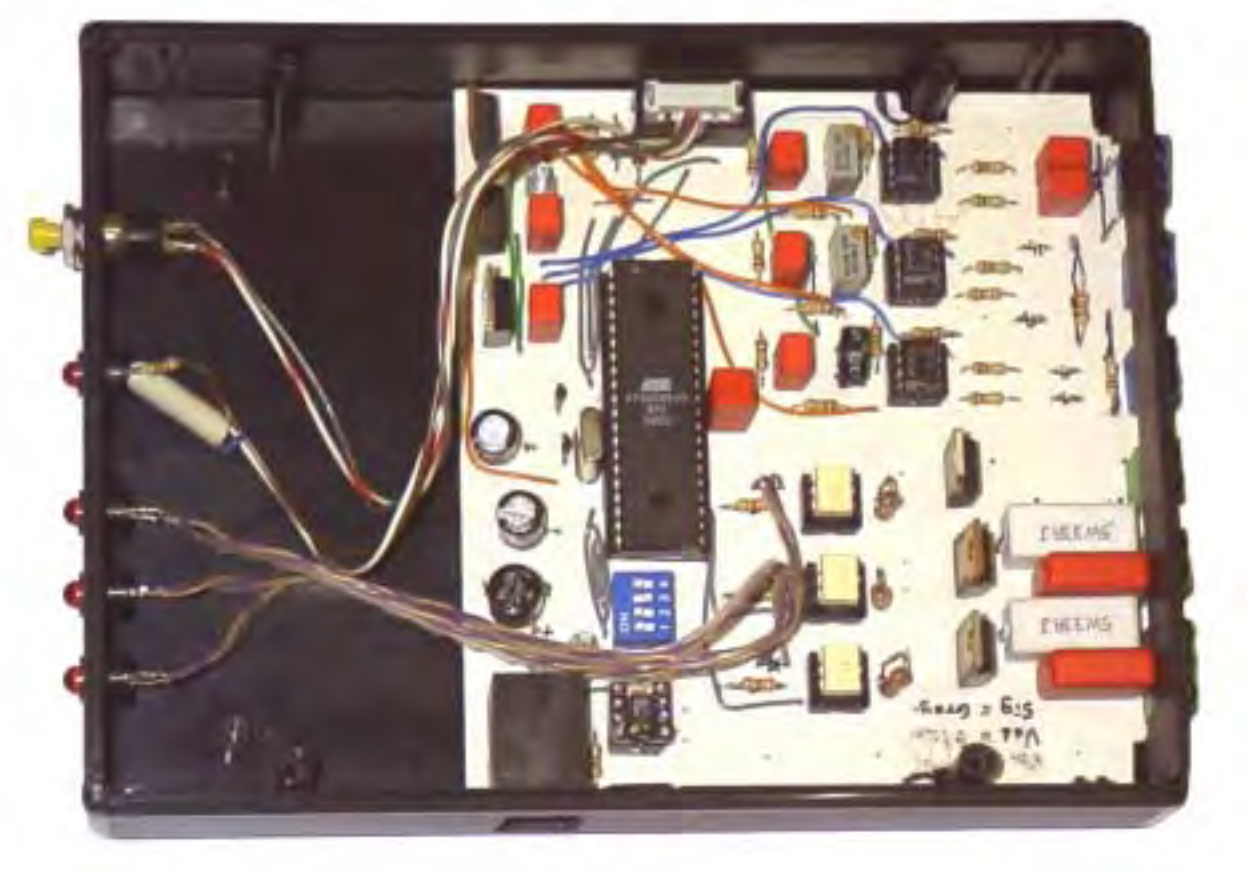


Figure 3.10: Development FEP

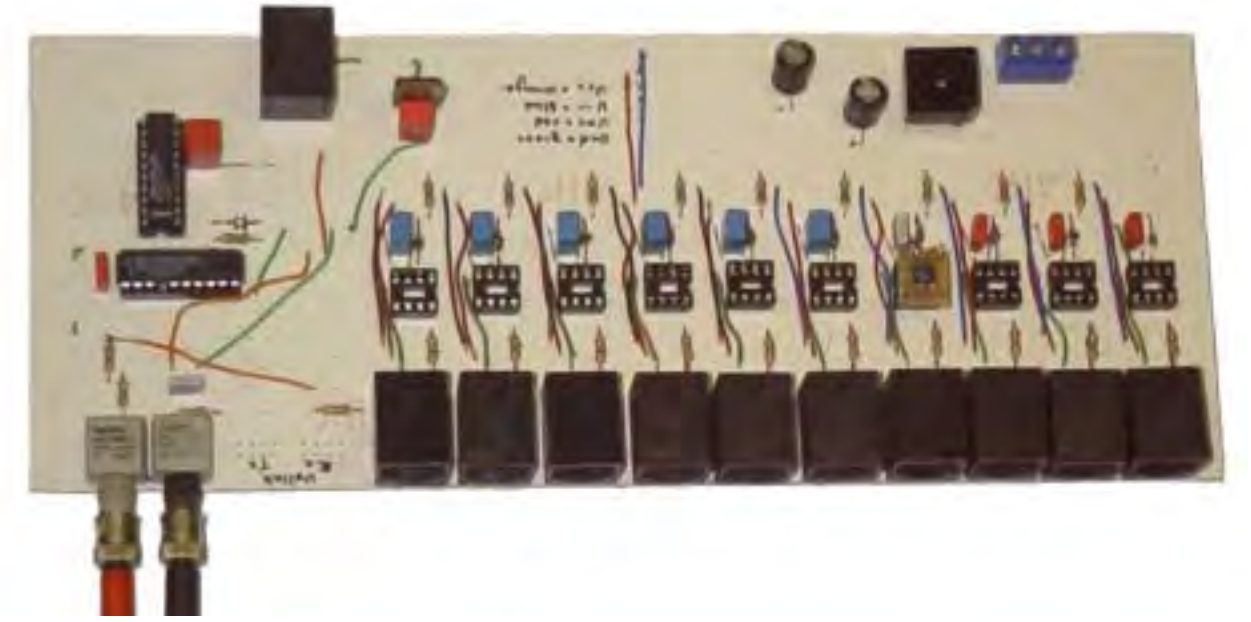


Figure 3.11: Development Distribution Board

Figure 3.12: The CS8900A & MAX488 ICs in perspective

Chapter 4

FEP Software

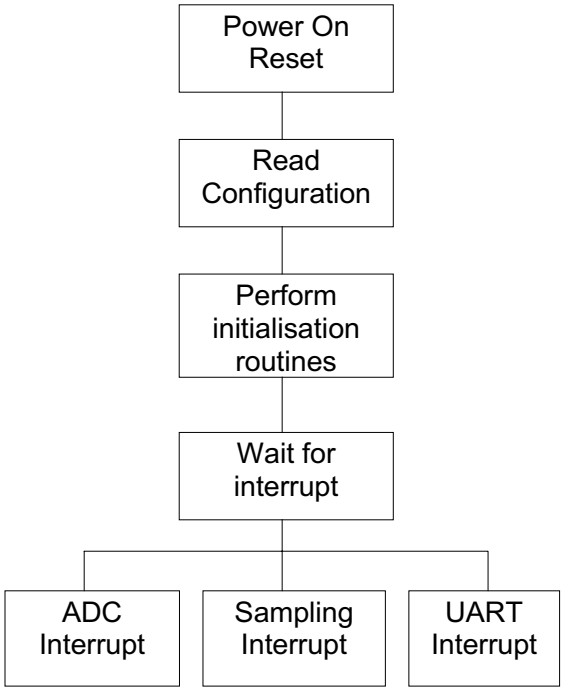


Figure 4.1: FEP Software Flow Diagram.

The FEP software is written almost entirely in C code. The Imagecraft ICCAVR compiler [ICCAVR] and Integrated Development Environment was used for this project. Initially a modular approach was taken in the source code, with each module of code contained in individual C files. This was later abandoned in favour of a monolithic C program to consolidate the code into a more manage-

able source file. The basic operation of the FEP source code is given in figure 4.1, with the various sections explained below.

4.1 Overall Operation.

All references in this section apply to figure 4.1 unless otherwise stated.

4.1.1 Power On Reset

If for any reason the power should drop below the threshhold voltage, then a reset will occur [Atmel 2, page 20]. This is a feature of the AT90S8535 which ensures correct and reliable operation. On any reset (including stack overflow, described later) the AT90S8535 starts execution from code address 0x0000 and continues operation as per usual.

4.1.2 Configuration

The FEP first reads in the settings on the DIP-switch, which sets the UID (UART ID) for the particular FEP. The EEPROM is then read to determine the conversion constants for the analogue input channels.

4.1.3 Initialisation

The output channels are initialised to their initial 'ON' state after a short delay. This is to ensure that if there is a hardware malfunction the contactors will not be switched on and off too rapidly, should the device get stuck in a re-boot cycle. The internal buffers are then manually cleared, should any previous data still be present (there is no guarantee that the memory locations will be clear if there was not a clean reset). The peripherals that are used are then initialised. After the setup has completed, the global interrupt flag is set.

4.1.4 Stack Overflows

While a stack overflow is not expected, it is possible and should a check not be in place, data corruption and incorrect operation could occur. This is most probable in the RMS measurement (detailed below), since it is possible that the data sampled could cause problems in the mathematical manipulation stage. Hence checks are performed on the hardware and software *sentinel bytes* shown in figure 4.2. These bytes are assigned known values by the compiler and then checked at various stages of operation by the call to *_Stackcheck()*. This function performs a software reset should one or both of these values differ from the expected value.

It is prudent at this stage to mention the differences between the hardware and software stacks. The hardware stack is used to store the return addresses used in subroutine calls and interrupt handlers, whereas the software stack is used for passing parameters and storing local and temporary variables.

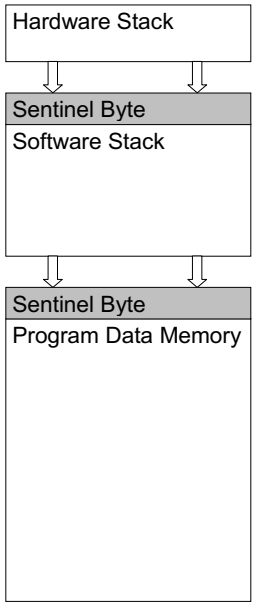


Figure 4.2: Stack Structure

The size of the hardware stack has to be specified at compile time and care has to be taken to ensure that if the code makes use of embedded subroutine calls that the value specified is large enough. All the code in this project (besides the Linux code) has been designed such that interrupts can only occur in the main program loop, hence the maximum value needed for the hardware stack can safely be calculated.

4.1.5 Interupts

The three interupts mentioned are:

- ADC interrupt: This interrupt occurs after an ADC conversion has been completed. The code stores the recorded value in the variables set aside, namely $\sum$x, $\sum$x^2 and *number of samples*. These values are then used by the conversion functions called from the sampling interrupt.

- Sampling Interrupt: This is a periodic interupt that is used to trigger ADC conversions. When the *'number of samples'* variable reaches the desired value, the other accumulated values are then used to calculate the RMS value, as detailed later.

- UART Interrupt: This is the interrupt generated by the reception or transmission of a character over the UART. A full flow diagram is included in the Appendix A2, which more clearly shows the UART operation. Section 4.4 on page 42 covers the UART functionality in more detail.

4.2 Data Acquisition

Data acquisition is not driven by requests from the Field Controller in order to minimise latency between the request and response. Since the FEP is likely to be idle most of the time, I decided to use this idle time to do the data acquisition. The actual measurement specifics are detailed in the section below - 'RMS Measurements'. The ADC cycles through its various inputs, moving on to the next input after a complete measurement has been taken on each input. After the data is processed it is stored in the relevant memory location corresponding to the logical data channel that the current ADC input is associated with. This allows almost instant access to data upon request, even though it may be up to 0.1 seconds out of date this is negligible with respect to the time taken for the information to be passed all the way to the master controller. While this does dramatically increase processor utilisation, since the processor is always busy reading data and performing mathematical calculations, it does reduce the latency between the request for data and the response. The interrupt latency is increased slightly, but not enough to seriously affect the performance.

4.3 RMS measurements

Since the sensed current waveform could not be guaranteed to be a sine wave, simple peak measurements of the waveform would yield incorrect information. To account for any serious distortions, the 50Hz signal is sampled 83 times over two consecutive cycles, giving a sampling rate of approximately 2kHz (shown in figure 4.3). The assumption of a 50 Hz wave is perfectly valid since this is the only real guarantee that Eskom places on the supply. The number of samples was chosen to be 83 since the period associated with this sampling rate was easily obtained from the timer used. The diagram shown in figure 4.3 on page 42 shows the sampling of the 50Hz signal. While a larger number of samples would have improved the quality of the data, it would have lead to numbers that could not be manipulated fast enough. The compiler used makes use of 2-byte integers and 4-byte long integers, so to accommodate the maximum value from the analogue to digital converter (1023), and the square of this value as well as the square of the sum, the long integer had to be used. The actual equation used to calculate the RMS value from the samples is given below in equation 4.1:

$$Value_{RMS} = \sqrt{(\frac{\sum < x^2 >}{n} - \frac{\sum < x >^2}{n^2})} \tag{4.1}$$

The square-root function used is implemented using successive approximations and is less accurate than the function used by the C-compiler. This was done because the compiler square root function was not fast enough and the extra precision was not needed. Since the square-root function

only returns a value to the closest integer value, a simple mathematical trick is used to maintain precision. The value sent to the function is multiplied by a known perfect square and the return value is then divided by the square root of the perfect square.

$$Value = \frac{\sqrt{Number * 256}}{16} \tag{4.2}$$

This can be best seen in equation 4.2, where *Value* is the calculated square root of *Number*. 256 was chosen as a perfect square since it is also a power of 2 and hence the multiplication and division can be accomplished by simple *shift left* and *shift right* instructions. Again it should be noted that while the pair of 10 and 100 would produce identical results in theory, due to the fact that the multiplication and division would have to be done in software using binary arithmetic, this would take more clock cycles as opposed to the 2 clock cycles required for a *shift left* or *shift right*.

The data is only scaled after the RMS value has been calculated since this is more efficient than scaling each individual sample. With a more powerfull processor in the FEP, such as the Mega163, the data could be processed more accurately and faster. The deficiency in the AT90S8535 is the fact that there is no hardware multiply instruction and all multiplication must be performed in software using shifts, addition and subtraction. (The Mega163 is not yet locally available.) The code space on the AT90S8535 is also limited and the compiler's built-in functions consumed a fair amount of space.

After the RMS measurement is calculated, the value is scaled using relationship 4.3 given below, which can then be combined with equation 3.2 to yield equation 4.4, which is used to convert the value from the ADC to a measured current.

$$\frac{ADCValue}{1023} = \frac{V_{out}}{V_{ref}} \tag{4.3}$$

$$I_{measured} = \frac{R_{in}}{R_f} * \frac{V_{ref}}{1023 * R_{sense} * CT_{Ratio}} * ADCVal \tag{4.4}$$

$$I_{measured} = Constant * ADCVal \tag{4.5}$$

While the above calculations are all performed using *float* data types, the function to print a *float* into a string requires more code space than is available on the AT90S8535 and therefore the values are written into strings by writing the integer part the *float* to the string, subtracting this number from the *float*, multiplying this answer by 10 and writing the integer part of this new result as the decimal place in the string. Since we are measuring relatively high currents, this precision

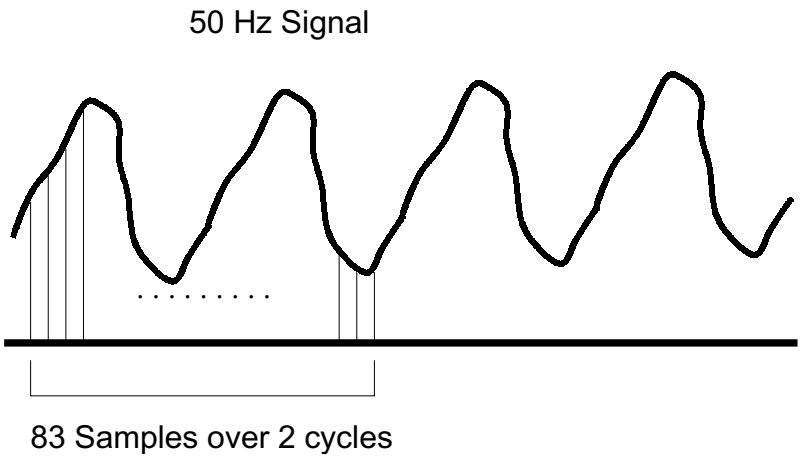


Figure 4.3: Sampling of the Waveform

is unnecessary since the error induced by dropping the fractional part of an Ampere would be negligible compared to the amount of current drawn by even one 1000 Watt heater. Equation 4.4 is not actually calculated each time a conversion is done. The *Constant* from equation 4.5 is calculated and stored in EEPROM, from where it is read when the device is powered up. During the conversion a simple IIR filter is applied to the measurements. If the value calculated before the IIR filter is invalid for some reason, the previous value is used. This was done to prevent known invalid data from being sent to the Field Controller.

4.4 UART Driver

The UART driver for the FEP works similarly, but not identically to the Field Controller UART Driver. This difference arises because parsing the request for data is much simpler than parsing the returned data. The UART driver utilises two buffers, one for transmission and one for reception. Upon character reception the UART driver compares the received token with its device ID. If they do not correspond then the UART ignores all characters until the reception of the EOT (0x04h) character. If the first character received matches its ID then it stores the received characters and parses the received data after the reception of the EOT character. The second character is always a channel ID number, ‘0’ being reserved for the FEP itself as opposed to one of its data channels, this allows extended functionality to be built in without redesigning the protocol. The structure of these messages can be seen below:

Master -> Slave: [Device ID] [Channel ID] [Operation] : [Data] [EOT]

Slave -> Master: 00:[Device ID] [Channel ID]:[Data] [EOT]

The reason that there is a ‘00’ header in the slave to master response is so that if the physical layout were redesigned with the FEPs daisy-chained together, then there would not have to much code alteration to allow the FEPs to pass responses through to the server. The first ’0’ indicates the ’FEP ID’ of the Field Controller, while the second indicates the ’channel ID’. A valid data channel number is not used because it is possible for the Field Controller to have data acquisition and control channels. If an error is detected or a character is missed, the FEP disregards all further input until the EOT character has been received.

It can clearly be seen that, even in designing the communications protocols, the versatility and scalability of the system were taken into account, presenting a transparent logical model that is, in my opinion, simple to follow.

4.5 Other Considerations

I must stress that since there was no detailed information provided about the interfaces to the electrical system, (i.e. specific information about the contactors and current transformers), I had to design around these areas. The Rhodes electricians were unable supply details about the consistency of the current transformers or contactors, so when implementing individual FEPs care will have to be taken, with possible tweaking of component values. This is discussed further in section 7 on page 83.

Space is allocated in EEPROM for information pertaining to the particular FEP such as *Code Date*, *EEPROM Date* and version information so that it is possible to determine whether or not a code upgrade is needed. This does pre-suppose that a database of code-versions is kept so that should a FEP fail and it has to be replaced, the correct code and conversion factors are used. This stems from the fact that the FEP monitoring current in the main substation will have different conversion factors than those present in the power distribution boards in a residence.

Chapter 5

Field Controller Software

The field controller is located at the second level in the heirachy of the design (the Master Controller is at level 1) as shown in figure 2.1. This chapter will cover the the issues that arose during the coding of the software running on the Field Controller. The structure of the software, data structures as well as the interaction and functionality of the various components will be discussed. I have made references to *the kernel* in several sections. This refers to the functionality of several of the core modules that would ususally be present in an operating system's kernel. Unfortunately the lines of division separating the modules are sometimes blurred. This is due to the close interaction that is necessary between the modules and also to avoid processing overheads.

5.1 Coding Issues

It is probably worthwhile to mention that coding practices that may seem unorthodox or dangerous to the 'Computer Science' C-programmer are needed to overcome some limitations of the processor and quirks of the compiler used. As an example there are several buffers that are declared as global variables, i.e. they can be seen and modified from almost anywhere in the code. This is seriously frowned upon by almost any computer programmer, but since there is no operating system to control shared memory or handle semaphores, most conventional approaches by the traditional 'C' systems or applications programmer are not possible. There are however variables in place that serve to fulfill some of the functions of semaphores. These 'semaphore' variables indicate whether a resource is currently being used or not. The other method to prevent data corruption is rather simple - 'do not touch memory or variables that don't belong to you'. This may seem dangerous to a computer scientist, but since the code is in effect an operating system, with only one process and no user applications running, it poses no danger, except to the careless programmer. Many data variables are declared and allocated space independently of the local variables in procedures, which are stored in the stack. Several of the large data variables are actually declared in assembler files and forced to reside in external RAM, where enough space is set aside so that there is no

danger of data corruption. These are then declared as *extern* variables in the C code so that the compiler does not bother with them and just accepts that there is space for them. This is not as dangerous as it seems due to the fact that the compiler is made to think that there is no external RAM, but external RAM is actually manually activated in the driver routines for the CS8900A. This may seem a little strange, but it will be further explained in the relevant ‘drivers‘ section (section 5.5) that follows later in this chapter.

5.2 Data Structures

Figures 5.1 & 5.2 show the main data structures used in the Field Controller. A brief overview of the different structures follows.

- **UART Data Structures**: These data structures are used to hold the data being sent and received, as well as to provide information as to the availability of the UART to the driver functions and the kernel.
- **Network Data Structures:** From figure 5.1 it can be seen that space is maintained for both inbound and outbound ethernet frames. This is to ensure that data from an inbound frame can still be read after a transmission has been made in response to the incoming frame. The miscellaneous variables include state variables to aid in the correct boot sequence and recovery after a network outage.
- **Real-Time Clock:** This is discussed further in section 5.11 on page 61.
- **System Data Structure:** This data structure is used to store the states of the various channels of the connected FEPs. The inter-operation is detailed in section 5.3.
- **Control Rules Data Structure:** This data structure contains the definitions of the control rules, both timed and dynamic. The control operation is covered in section 5.4.

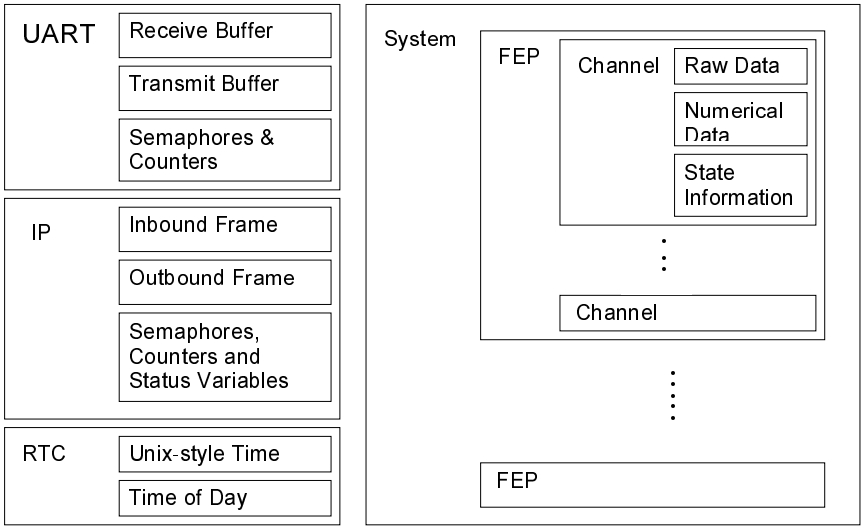


Figure 5.1: Data Structures A

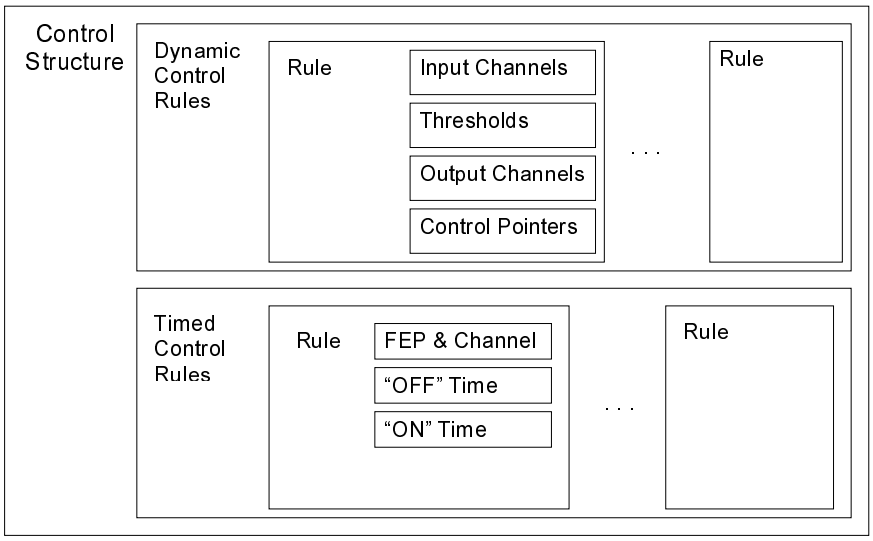


Figure 5.2: Data Structures B

5.3 Data Flow

Figure 5.3 shows the communications between the various parts of the system, as far as the flow of information and control data is concerned.

- The arrow labelled A shows the flow of control signals to the UART driver from the Power Control Protocol and the acknowledgements that the UART receives from the FEPs back to the Power Control Protocol.
- B shows the reading of data from the System data structure by the Power Control Protocol upon reception of a request for data from the Master Controller. The operation of the Power Control Protocol is explained in greater detail in section 5.10.4.
- C shows the reading of data by the local control algorithm when local control is needed.
- D shows the transmission of commands to the various FEPs by the local control algorithm.
- E shows the storing of data (not acknowledgements) from the FEPs in the system data structure.
- F shows the periodic update signal sent to the FEPs via the UART driver. This signal causes the responses indicated by E.

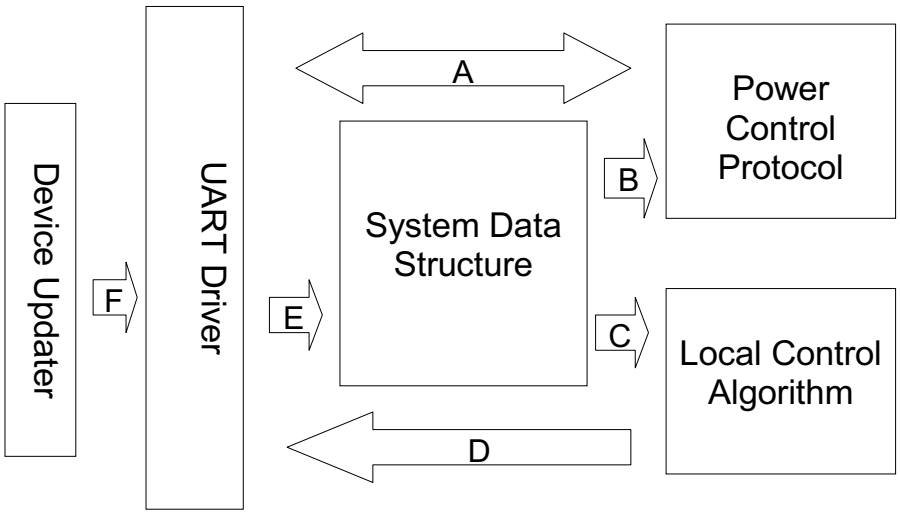


Figure 5.3: Data Communications paths on the Field Controller

5.4 Control Algorithms

The control algorithms in the Field controller are based on a simple rule structure giving possible controllable devices for each input device. If the value of the input sensor rises above an upper threshold then more devices for the given input are switched off, thereby lowering the current drawn. If this causes the current drawn to fall between the upper and lower thresholds then the devices are switched in a round robin fashion. When the current falls below the lower threshold then the devices are switched on. Care is taken not to switch devices too rapidly as this could cause mechanical failure in the contactors used to control the heating circuits. The details of the control parameters will be discussed further in the section entitled *'Power Control Specifics'*.

There is also an option for timed control of outputs based on the state of the Real Time Clock. It should be noted that these *timed rules* will only be followed if the Field Controller has established a connection to the Power Control server and obtained the system time since its last re-boot. This limitation is made since after booting, the volatile RTC will not represent the true time and hence cannot be used to make sensible control decisions. If the server cannot be found after power-up then only the local load shedding rules obtained from the EEPROM will be executed. All local control is suspended if there is a current connection to the server. It should be noted that the local control rules should be set up as a backup for when the network communications channel is disrupted.

These local control rules should be set up in accordance with some predetermined *Power Control Policy*[1], the only parameter currently hardwired into the system is that no device will be switched more than once every 5 minutes. This was done to limit stresses to the contactors.

The current code limits the number of local control rules to 10 (5 timed control rules and 5 input based control rules), which is sufficient for the intended application.

5.4.1 Timed Control Rules

The timed control rules take two arguments, besides the channel to be switched. The first is the 'ON' time, or the time of day that the device is to be switch on. This argument is actually the number of seconds since midnight (00h00:00). This was based loosely on the Unix method of storing time (the number of seconds since the epoch, January 1, 00h00:00), except the epoch is the beginning of the current day. The same argument applies to the 'OFF' time. It should be noted that all channels on all FEPs initially start in the 'ON' state.

[1] As of writing this document no 'Power Control Policy' has been established.

The time is stored locally as a ‘time of day’ (and the UNIX-style time), which is initially calculated from the Unix style time received from the Master Controller and then incremented every second along with the Unix style time by the RTC. This approach to time-keeping simplifies roll-around and accuracy problems associated with storing the time as hours:minutes:seconds. Future versions of the hardware could incorporate a non-volatile RTC[2] which would only require setting the time once and thereafter keep the correct time, even during power failures.

5.4.2 Input Based Control Rules

Each input-based control rule can have up to 3 associated inputs. If more than one input is specified then the sum of the inputs is compared to the threshhold values given. Up to 10 output channels can be associated with each rule. This results in a maximum total of 50 controlled channels, which is more than the maximum of 27[3] that can currently be connected to one Field Controller. An upper and lower threshhold are also specified for each rule. These control rules operate as described in the Introduction.

[2] Dallas Semiconductor DS1243(8K + RTC), DS1244(32K + RTC), DS12877(RTC)

[3] Maximum of 9 FEPs, with 3 output channels on each FEP.

5.5 Drivers

5.5.1 CS8900A Driver

This code is completely contained in the *'CS8900_driver.c'* source file on the CD. On viewing the code the first things to note are the include files. The first two are <*iom103.h*> and <*stdio.h*>, which let the compiler know what device we are building for and where to find the object files for the standard functions.

The next include file is the *'hardware.h'* file. This file redefines the IO ports in terms of meaningfull names, eg. *DATAL_OUT* instead of *PORTA*, which enables easy modification of the code if the target device is changed and the port names change. There are also several definitions (processed by the C pre-compiler) that make the code easier to read, such as *IOW_LOW* instead of *CONTROL&=~0x40*, so that the reader is able to easier understand the meaning of a line of code. This also introduces a layer of abstraction, so that the programmer does not have to remember which ports do what and the code can be separated slightly from the hardware level. It should be noted however that the programmer does still need to have some idea of the hardware layout, because to ignore this would be a serious mistake.

The next include file is *'CS8900_equates.h'* which defines IO addresses for the CS8900 and CS8900 internal register settings in terms of easier understood 'English' words, for example, *ppRxCfg*, for the internal address of the Packet Page Receiver Configuration register, instead of 0x0102h. Again it would be possible for the programmer to refer to the CS8900 documentation, but then the code would become unreadable.

The *'configuration.h'* file contains configuration information about the system setup.

The *'Ethernet_layer.h'* file contains all the relevant Ethernet numbers[4], this may seem to be out of place in the CS8900A driver file, but this is due to moving some of the decision making ability from the Data Link layer to the driver. This is to make the code slightly more efficient. If a broadcast frame is being read in, the interrupt handler in the driver checks to see if the data contained within is an ARP packet, if it is then this is passed on to the relevant function, if not the frame is discarded. This is to improve the performance of the device, since the only broadcast frames that we are interested in are ARP frames and all others are of no interest. If all broadcast frames were passed to the Ethernet layer by the driver, there would be a serious performance loss.

Next is the pragma to tell the compiler about our interrupt handler for the CS8900A, in this case it uses interrupt vector 9, which is used for the external interrupt number 7, which is associated with INTRQ3 on the CS8900.

Next come the variable definitions, which are all externally declared since they must be visible to procedures in other code files.

[4]RFC 1700

The *InitCSLines* procedure is next. This is used to set up the IO ports on the Mega103 and is also used to make sure that the CS8900 is fully reset. This is the first function used after the *Initports()* function, which is the first function called to make sure that all the ports are set as input ports. This is to ensure that everything starts up in a known state. It also enables the external RAM.

portRead is one of the low level drivers. This function reads in from the specified port and returns the value that was read. To do this the driver first disables external RAM, then asserts the address of the requested port on the address bus, enables the *IOR* line to the CS8900, reads in both bytes of data, disables the *IOR* line and then re-enables the external RAM. If the external RAM was not disabled then the IO ports could not be accessed as IO ports. (page 22 of the Mega103 Data sheet.)

portWrite is the complementary function to *portRead*. It takes two arguments, the data to be written and the desired port on the CS8900A that is to be written to. The external RAM is first disabled, the address is then asserted, the datalines are then set to be outputs, the data is written to the Mega103's IO pins, the *IOW* signal is asserted, after a short delay it is disabled, whereupon the data lines are reconfigured as inputs and the external RAM is re-enabled.

SoftReset is a software reset of the CS8900A that is achieved by setting the reset bit in the Self Control Packet Page register. It is not actually used since a hardware reset is just as effective, but it was written just for the sake of completeness.

Configure is used to set up the CS8900A after a reset or power-up. Firstly the the desired interrupt is activated and the receiver configuration register is set up to only cause an interrupt if an entire frame is received without errors. [Cirrus 1, page 53]. The receiver control register is then configured to only accept frames of correct length, that pass their CRC check, and are either broadcast frames or addressed to the unique MAC address given in the configuration file. The Bus control register is then used to enable interrupts on the CS8900A. The line control register is then set up so that the chip can transmit and receive on the 10baseT interface. The desired MAC address is then set in the CS8900A.

The last and most important piece of code here is the interrupt handler. The interrupt handler first reads in the ethernet frame checking to make sure that the only broadcast frames accepted are ARP requests. If a broadcast frame is detected, but is not an ARP frame, then the whole frame is rejected. The handler then passes the ARP request on to the ethernet layer, or if the frame was addressed to the device's unique MAC, then it is passed on to the IP stack.

It is important to note that although these routines may take a long time, any interrupts that may have occurred during their execution are stacked and executed according to hardware priority when the interrupt that is currently being processed has ended. This does introduce a slight interrupt latency, but a few microseconds are not too critical in this application. A flow diagram of the CS8900A driver operation and IP interaction can be seen in the appendix A1.

An important fact regarding the CS8900A driver is that it configures the hardware to oper-

ate even if a link-beat is not detected. This was done to eliminate any possibility of the device 'hanging' should the link beat fail after a frame has started to be transferred to the CS8900A. The overheads of paranoid checking are not worthwhile when the device can be configured to transmit regardless of the link status. This has no serious repercussions since all network failure detection and recovery is handled by the netstat() command described later in section 5.10. In doing this there may be some complications when connecting the device directly to a 10/100 switch that defaults to 100Mbit transmission, but since I did not have access to any such equipment I unfortunately could not check this. This limitation could be rectified with some subtle manipulation in the driver files, if trouble is detected, but since I had no way of verifying the changes to the configuration of the CS8900A while connected to such equipment, I decided to remain with the original configuration.

5.5.2 UART Driver

The driver maintains two buffers, one for transmission and one for reception and it is the only code that is allowed to write to these buffers. Any procedure that wishes to send data out through the UART must call the 'Write2UART' function, passing arguments identifying which channel on which device is the destination as well as a function identifier and the data which is to be sent. If the UART is currently busy then the message is passed to the scheduler and scheduled for transmission as soon as possible. The scheduler would then pass the message back to the UART driver when it is scheduled to do so.

The transmit interrupt routine sends the next queued character in the buffer on completed transmission of the previous character, until the 'null' character is detected, whereupon the routine sets the UART_Semaphore variable to indicate that the buffer is free to be written to.

The receive routine writes received characters to the receive buffer until the 'end of transmit' character is received. This 'EOT' character is appended to all UART messages when they are written to the UART transmit buffer. The routine then calls the kernel by asserting a software interrupt, the kernel uses the highest priority interrupt, so that is will always be the first interrupt to be serviced under all conditions. This software interrupt is discussed later.

The UART driver code also contains the initialisation code to set up the UART, which must be executed before any UART activity can occur.

5.6 Software interrupts

Many of the procedures make use of software interrupts to call various sections of the kernel. The term software interrupt is a bit misleading, since although to a programmer it appears to be a software interrupt, it is actually an IO port manipulation of one of the IO pins configured as an interrupt pin. This then generates a hardware interrupt condition that is handled by the Mega103 upon completion of the current interrupt. This may seem dangerous at first, since a reckless programmer may call one of these interrupts from a section of code in the main() procedure by accident while still running some of the setup code. If this interrupt were to be executed before all of the initialisation had completed there could be some unpredictable behaviour. Fortunately the whole kernel and operation of the device is designed to be interrupt driven and there should be no code whatsoever in the main() procedure after interrupts are enabled and under no circumstances should the global interrupts be enabled before all setup functions are completed. That is to say, that once all the setup functions have been called and the global interrupts enabled, the processor will just remain in a loop if no interrupts are generated. This is similar to an *event driven* application in normal computer programming. All interrupt code is 'interrupt proof', that is, it cannot be interrupted. An interrupt that is generated during another interrupt will not be serviced until the current interrupt is complete. In the unlikely event of a severe lockup, where no interrupts are being serviced, the watchdog timer [Atmel 1, Page 51] will cause a system-wide reset.

A closer look at this idle loop may provide an insight into why embedded programming is so different from the programming that is taught in a Computer Science class. Below is an extract from the *void main()* function:

```
while(1)
    {
     asm("nop\n");
    . . .
    . . .
     asm("nop\n");
     asm("nop\n");
    }
```

Which would normally be coded as:

```
while(1)
   {
   }
```

The reason for all the in-line assembler *asm("nop\n");* lines is that the first extract would produce assembler code similar to:

```
LOOP1: nop
       nop
       ...
       nop
       rjmp LOOP1
```

While the second example would yield assembler code similar to:

```
LOOP1: rjmp LOOP1
```

While the second example of assembler code looks more efficient and takes up less code space, it has one flaw when considering interrupt latency. Since the *rjmp* op-code takes 2 clock instruction cycles to execute, an interrupt has that much more time to wait than in the first assembler example. This may seem like a insignificant difference, but it is this sort example that sets embedded real-time programming apart from *Computer Science* or *Information Systems* programming. In actual fact this particular example makes little difference in this project, but it does serve to illustrate a point ... an understanding of the hardware and architecture of an embedded system is essential.

Without the use of these software interrupts it would be possible for a careless programmer to structure their code so that interrupts would be lost. While I have coded the entire project to date, it is possible that the code will be maintained in the future by someone who may not have the necessary skills in embedded programming to spot these dangerous areas.

5.7 The kernel

The kernel is a loose collection various functions like the ‘updater’ (described below), the ‘scheduler’ (also described below), the control algorithms and the ‘system_tick’ which is similar in functionality to the unmaskable interrupt used in personal computers to ensure that the operating system has a guaranteed access to processor time. The kernel is not just one section of code, but rather the interaction of several sections of code all interacting to control the various aspects of the system.

5.8 The Updater

The updater module is run off of the system clock (not the real-time clock) to continuously update the local data-structure representing the state of the devices lower in the power control heirarchy. The updating is not in any way driven by requests from the Master Controller via the network since this would increase the latency of the response to the network. The updater requests updates at a rate of approximately 40 channels per second. This means that the data from each channel is refreshed about once every second. The updater has no intelligence, that is, it requests data from all possible channels. This is done so that no reconfiguration of the device is needed if a new FEP is 'hot-swapped' into the system. While this is not an optimal method for retrieving data, it does mean that maintenance is cut down and FEPs can be added and removed without re-configuring the control processor. It also aids in RAM management, since the device structure is fully populated and will not grow. If a variable number of devices were used, then an implementation of linked lists would be needed. While this is good coding practice and what any computer scientist would do, it does unnecessarily increase the code size and slow the implementation down significantly. There is also the possibility of stack or heap overflows. The static nature of the structures used allows effective control over the external RAM, since there is no operating system to deny allocation of RAM it is better to allocate it statically to prevent accidental corruption of other data.

5.9 UART Scheduler

Since there is more than one source of data for the UART, (namely the device updater and instructions from the Master Controller), there has to be some sort of control mechanism to control access to the UART resources. Since the update messages sent by the device updater are not time critical to the same extent as those sent by the master controller, when a message from the master controller must be relayed to an FEP the current message being transmitted is completed but the updater is suspended until the FEP has responded or timed out. The UART scheduler allows messages from the Master Controller to be queued. Again it is possible to see the legacy nature of the code since it is still called a *scheduler* as opposed to a *manager*. This is because there was initially a single scheduling algorithm that handled both UART transmission and pending tasks from the control algorithm. This was later split up to aid in debugging and memory management.

5.10 Network Protocols

5.10.1 Ethernet & IP Development

Originally I had planned to base the protocol stack on the Linux TCP/IP implementation, but after inspecting the Linux source-tree and examining how its IP stack is implemented it was decided that this would not be productive. Most of the Internet resources on embedded IP stacks proved to be vague and misleading, so it was decided to code the stack from the ground up based on the protocol definitions in the RFCs[5] and non-technical literature[6] that was available, as well as the 'World of Protocols' Homepage.[7] The documents listed in the footnotes are not explicitly referenced only due to the fact that I would have to include the entire text. These documents explain the operation of the Internet Protocols which had to to be understood before implementation. Below I will describe the stages of development as they occurred.

Test Transmission: This stage involved manually constructing an Ethernet frame in the Field Controller's RAM and transmitting it over the network, a *'Packet sniffer'*[8] was used to see if the frame was actually transmitted. (At this stage the Field controller was only connected to a small LAN consisting of a Windows 95 and Linux machine.) This was the phase when the configuration of the CS8900A was debugged.

Test Reception: After successfull transmission was accomplished, the reception of frames by the CS8900A was tested. At this stage the Address Resolution Protocol was not yet implemented so the ARP cache of the PC had to be manually manipulated[9] to enable the Windows PC to transmit the frame. A known frame was transmitted to the Field Controller, after which the Field Controller sent the data contained back to the PC via a serial link, so that the validity of the data could be established.

Address Resolution Protocol: For the Field Controller to operate properly it would have to be able to respond to ARP requests so that other network devices could determine its MAC address given its IP address from an IP datagram. The code dealing with this can be found in the *'Ethernet_Layer.c'* and *'Ethernet _Layer.h'* files. Owing to the fact that this Protocol is in the Data Link layer and is used during the boot process (discussed later) it will respond to an ARP from

[5]RFC 768 - User Datagram Protocol, RFC 951 - Bootstrap Protocol, RFC 2132 - DHCP Options and BOOTP Extensions, RFC 2131 - Dynamic Host Configuration Protocol, RFC 1180 - A TCP/IP Tutorial, RFC 1700 - Assigned Numbers, RFC 826 - An Ethernet Address Resolution Protocol.

[6]Computer Networks, Andrew S. Tanenbaum.

[7]World of Protocols - www.protocols.com

[8]Ethereal Network Analyser - A network analyser for Linux-based systems.

[9]eg. The command arp -s 146.231.84.153 00-00-ab-cd-ef is used to associate the IP address (given in dotted decimal notation) with the hardware address (given in hyphenated canonical form).

any machine. While this is part of the standard, it will become apparent during the discussion of the IP implementation that I have deviated from the standards in several ways to improve security and performance. This deviation has not caused any problems or compatibility issues, and will not provided the rest of the LAN infrastructure implements the standards properly.

Internet Control Message Protocol: The full range of ICMP functions were not implemented due to their relatively large overhead and minimal value in this application. Restricting the ICMP implementation to *'ping'* alone also makes the system more robust, since many of the ICMP functions can be used by hackers to gain information about a device and manipulate it. The *'ping'* was implemented since it is a handy diagnostic tool to see if a device is connected to the network. While a function with similar functionality was implemented in the Power Control Protocol, the *'ping'* function allows the user to diagnose problems such as the case when the incorrect BOOTP information is supplied and the Master Controller is unable to establish a connection with the Field Controller using the Power Control Protocol. All incorrect *'ping'* packets or any other ICMP packets are silently discarded. The source code for the ICMP ping can be found in *'IP_layer.c'* and *'IP_layer.h'*.

Internet Protocol: The IP implementation is rather rudimentary, since this is all that is required to send UDP datagrams. Some of the IP limitations include:

- No packet fragmentation. No fragmented packets will be accepted, since the maximum size of the Ethernet frame of about 1.5 kB is sufficient for all data transfers. The re-assembly of fragmented packets would also require larger amounts of RAM, which is not feasible.
- No source-routed packets. The IP stack will silently discard any source-routed packets in order to make it harder for hackers to gain control of the system.
- Restricted IP access: The stack will only accept packets from the Power Control Server as set in the BOOTP response. There is no reason for the device to respond to any machine other than the Power Control Server.
- Options Ignored. If any of the extended options are used the packet is ignored. This is determined by the *'Internet Header Length'* field.

The IP checksum is calculated as the 16bit one's complement of the one's complement sum of all 16 bit words in the header [RFC 791, page 14]. On reception of an IP datagram the checksum is not checked since an error in the IP header (causing a checksum error) would have been picked up by the Ethernet CRC-check performed by the CS8900A, which is configured to ignore packets which do not pass the CRC check, thus saving valuable processor time. The *'Type of Service'*[10] field is

[10] See IP Header Structure in Appendices or consult RFC 791.

ignored, since this is not important. When transmitting the *'ToS'* field is set to *'routine'*. The *'time to live'* field is also ignored on reception of an IP datagram, since if it had expired it would not have been passed on by the gateway and since the Field Controller does not relay packets it is irrelevant. On transmission the *'TTL'* field is set to the maximum of 255, even though a value of as low as 10 should see the datagram reach the Power Control Server. The source files dealing with the IP layer are *'IP_layer.c'* & *'IP_layer.h'*.

User Datagram Protocol: The UDP protocol is implemented as an extention to the IP layer, this was done to improve performance and reduce overheads.

UDP Reception: The reception of UDP packets is limited to port 61440, which was done to limit the amount of processing done on packets that managed to pass other filters. Only Power Control packets need to be passed since the system is dedicated to this application. Again the checksum is ignored since packets are not fragmented so any errors introduced would be picked up by the Ethernet CRC check.

UDP Transmission: This is handled trivially as well. All UDP packets are constructed so that they will be directed to the Power Control Server on one of the pre-determined ports.

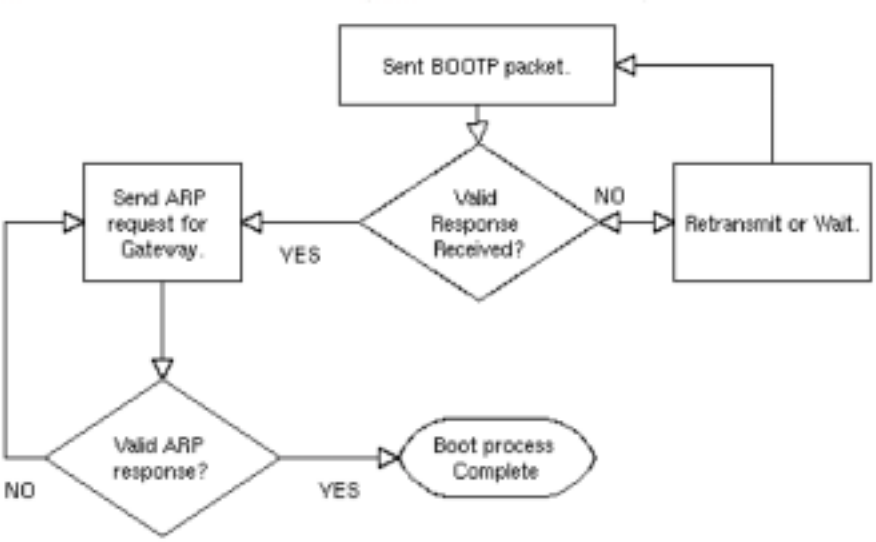


Figure 5.4: Boot Process

BOOTP: The BOOTP protocol is handled separately from the User Datagram Protocol even though it is a broadcast UDP packet directed to port 67. This was done to simplify the IP stack.

The entire Ethernet frame is constructed statically, since it is a broadcast Ethernet frame containing a broadcast (255.255.255.255) IP packet. The BOOTP packet is a small part of the entire boot process, which can be clearly seen in the flow diagram below.

All timeouts are handled by the Real Time Clock which calls a network status sub-routine (netstat)[11] that determines the appropriate course of action based on a state variable. This routine also handles recovery after a network outage and maintaining the status / debugging LEDs.

5.10.2 Deviations from the networking standards

As stated before, several deviations from the standard have been committed, but with justifiable reason. The most obvious are re-iterated below along with motivations.

- **Not validating checksums:** The validation of checksums is time consuming and since (as stated before) any errors in the packet would have resulted in a CRC error in the Ethernet frame, which would have been discarded by the CS8900A. The CRC check performed by the CS8900A is also far more secure than the IP checksums. As an example, if for some reason 2 bytes were corrupted so that they appeared reversed, it is possible that the IP checksum would not detect this error, but because the Ethernet CRC also depends on the order of the data it would not pass the CRC check. If datagrams were allowed to be fragmented and then re-assembled, then it would be necessary to perform IP and UDP checksums, since a fragment may have been lost, which would then affect the checksum.

- **Sending datagrams:** When sending datagrams all Ethernet frames are always sent to the local gateway's MAC address. In the standard, an ARP request is supposed to be sent to determine whether or not the required IP address is on the same physical segment as the sender. If this ARP request was implemented, then a local ARP-cache would have to be maintained as well. By sending all traffic to the local gateway, anyone using IP-spoofing to sabotage the system (on the same subnet) would be picked up by the gateway.

- **ICMP:** In the proper ICMP implementation, if a UDP datagram was sent to a non-used port, or if it was not accepted, an ICMP message conveying this would be sent back to the remote host. This serves no usefull purpose as far as the Power Control application is concerned and would merely induce higher overheads to the system. This also deals with DoS[12] attacks in the most effective manner available.

[11] Named for the 'netstat' command in Linux.
[12] Denial Of Service

5.10.3 Network Status

The netstat() function checks the state of several status variables indicating the current stage in the boot process and the status of the link with the Master Controller. To ensure proper operation it is run from the RTC interrupt so as not to overload the system. The state variables are altered by the code handling the various layers of the protocol stack. Running this command from the RTC also allows persistant re-transmission of BOOTP and ARP requests which are needed to ensure a complete boot sequence.

5.10.4 Power Control Protocol

The messaging protocol developed is based on several 2-byte identifiers to identify the type of data or query present. The first two bytes of data in the UDP datagram indicate the operation, these were chosen arbitrarily.

Refresh: This primitive is used by the master controller to obtain updated values for a specified FEP connected to the Field Controller.

> **Master-> Field Controller:** [0x41] [0x41] : [UID]
> **Field Controller -> Master:** [0x41] [0x41] [UID] [0x7C] [Channel number] [0x3A] [Channel Data] [0x3B]

The field controller response is not limited in the number of channels that it returns and since the channels are separated by semi-colons and the master parses the packet until the end of the packet is reached or until an invalid channel is detected. The FEP ID byte is included in the response so that if a request for data is missed the Master controller does not associated the data with an incorrect FEP in its internal data structure. The IP address is not included since the Master controller can determine this from the IP header.

Write: The 'write' primitive is used to write data to an output channel.

> **Master-> Field Controller:** [0x22] [0x22] [IP address in dotted decimal format] : [UID] : [Channel number] ; [Message] [0x04]
> **Field Controller -> Master:** [UID] [Channel number] : [ACK]

The acknowledgment is sent to the acknowledgment port (61441). This is done so that the control software can keep track of what devices have responded to commands. The acknowledgment is generated by the relevant FEP and is relayed back to the Field Controller for relaying to the Master Controller. This is done to ensure that if an acknowledgment is sent then the desired action has

actually taken place at the FEP level. The ACK is only generated in the FEP when the software has successfully decoded a messaged and performed the action required.

Time: The *'time'* primitive is used to query the remote time as held be the Field Controller's internal RTC and set it to correspond to the Master Controller's clock should it deviate. There are three possible scenarios:

- Field Controller requests time after booting:

  FC-> Master: [0x33][0x30][0x3a] *{FC requests time from Master}*
  Master-> FC: [0x33][0x31][0x3a][Unix style time in seconds since epoch][0x3b] *{Master Responds}*

- Master requests time from Field Controller:

  Master-> FC: [0x33][0x30][0x3a] *{Master requests time from Field Controller}*
  FC-> Master: [0x33][0x33][0x3a][Unix style time in seconds][0x3b] *{Slave responds}*

- Master sets time on Field Controller:

  Master-> FC: [0x33][0x31][0x3a][Unix style time in seconds][0x3b] *{Master sends time}*
  FC -> Master: [0x33][0x33][0x3a][Unix time][0x3b] *{Slave Responds}*

There is a response for setting the time but this is not specifically needed and it was decided to not make use of this in the Master Controller code, but just to silently discard the Field Controllers response. This enables the Master Controller to verify the time on the Field Controller that it has just set without changing the code on the Field Controller.

5.11 Issues Surrounding the Time.

The time is stored as an *unsigned long integer* on the Field controller which is only 4 bytes long. This means that the highest value that it can hold is 4294967295. This style of time is not used throughout the software, a similar *'seconds since the beginning of the day'* variable is calculated from the Unix-style time and used for the timed control loops. This does mean that the Power Control System will fail (or at least act erratically) sometime near the beginning of 2106. If the system is implemented and is still in use in 2106, I think it would be a fair testament to its robust nature.

5.12 EEPROM Issues

The control rules are stored in a .eep file after compiling the code on the development platform. Unfortunately there is no simple, elegant or easy way of enumerating the structures containing the control rules and it is easy to make mistakes in the formatting of the structure. Below is an extract of the control rule setup:

```
#pragma data:eeprom

char EEMAC[10]={0x00,0x00,0x00,0xAB,0xCD,0xEF,0x00,0x00,0x00,0x00};

LC_RULE myErules[5]={
/*start of R1*/     {
/*in1.1*/    {{'5',1},
/*in1.2*/     {'5',2},
/*in1.3*/     {'5',3}},
/*#inputs*/   3,
/* out 1.1*/   {{'5',4},
/* out 1.2*/    {'5',5},
/* out 1.3*/    {'5',6},
/* out 1.4*/    {0x00,0x00},
. . .
/* out 1.10*/  {0x00,0x00}},
/* #outputs*/     3,
/* upper thresh*/ 100,
/* lower thresh*/  50
/*end of R1*/     },
 . . .
/* end of array of rules*/ };
#pragma data:data
```

As can be seen the structure has to be enumerated explicitly at one time, with arrays and sub-structures being delimited by curly braces. Even though there are only 3 outputs, the entire array of 10 outputs must be initialised. The EEPROM also contains information such as code version, code date, EEPROM version, EEPROM date, a device ID and the Ethernet MAC address. The space allocated for some of these variables is more than is needed, but since there is 4K of EEPROM available on the Mega103, this wasted space is not critical.

5.13 MAC Address Issues

An ethernet MAC address has the following format: AB:CD:EF:GH:IJ:KL, where AB:CD:EF is the OUI supplied by the IEEE to network adapter manufacturers and the GH:IJ:KL represents the serial number assigned to the card by the manufacturer. (All MAC addresses are given in this canonical form.) Since it is costly to buy an OUI from the IEEE, all MAC addresses for the power control system should be generated according to the following rules to avoid clashes with existing network equipment.

- The addresses should be '*locally assigned*'. This is defined in the IEEE standard, which I could not get access to, but can also be found in [RFC 2153, Page 4]. This means that the second least significant bit in the AB byte must be a one.
- The rest of the OUI should be generated so that it is specific to power control devices. A suggestion for the entire OUI would be 02:00:FC.
- The rest of the MAC address should be generated sequentially so that no two devices have the same address. It would be adviseable to maintain a database of these addresses, as well as the name assigned to them by the IT division.

If these simple guidelines are followed then there should be no conflicts on the Rhodes network, unless some other locally assigned MAC address has the same OUI and serial number. This specification is also in line with the IEEE specifications.

5.14 Local or Remote Control?

The Field Controller keeps track of when it was last contacted by the Master Controller, and if there is ever a delay of more than a pre-determined number of seconds, it switches to local control. The time since the last contact from the Master Controller is maintained by both the RTC and the code dealing with the Power Control protocol. In the event of a loss of link-beat normal operation will continue until the previously mentioned time-out has occurred.

Chapter 6

PC-Based Master Controller Software

The software for the controlling PC was developed for Redhat Linux 7.1 (2.4.3-12 kernel upgrade) using Kdevelop, a graphical integrated development environment. Originally Kdevelop 1.3 was used under KDE 1 running on a Redhat 6.2 installation, but this was dropped in favour of the 7.1 upgrade due to some inconsistencies in some of the support libraries used and since KDE 2 under Redhat 7.1 came with all needed libraries as part of the installation CD. The code should be able to be compiled under any Unix-style environment since only standard system calls were used, with no special libraries. For ease of future development and code readability the functionality had initially been divided between two programs, namely the network interface and the actual control software, but due to difficulties that arose when the web-interface was been coded, the network module was integrated into the control program. This model was then altered again: the code dealing with the CGI web-page generation was moved to a separate program, the reasons for which will be discussed in the relevant secions.

The program initially reads in the configuration files *'system.cfg'*, *'control.cfg'* and *'preferences.cfg'* to know what devices are logically attached to the system and what their control parameters are, as well as some of the variable parameters. The program then launches several threads to handle the various tasks. Initially threads were used as opposed to processes due to the fact that sharing memory between processes is tedious to code, whereas threads share global variables automatically. This was later changed when shared memory was needed for the CGI interface, but it was decided to leave the code as thread-based, rather than process-based due to the sharing of not just global variables, but filehandles as well.

6.1 Overview

The basic flow of operation, with regard to thread and process creation is shown in figure 6.1.

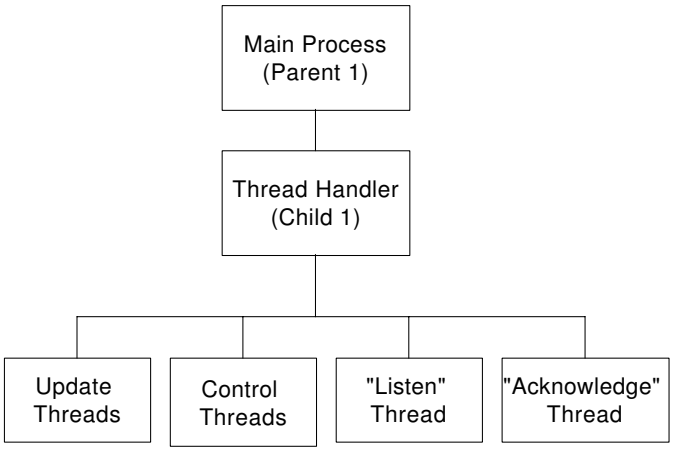


Figure 6.1: Process Structure

As can be seen (from the code) the program initially sets up the shared memory segment, a signal handler, reads in the configuration files and then *'forks'* a child process (child1).

The child process *(child1)* then spawns threads to deal with the various aspects of the system, as described in the sections below.

The signal handler is set up so that when the child process catches the signal (SIGINT or ˆc from the keyboard) it kills all the threads that it has spawned, and the parent sets the object variable *sys.Online='N'* , indicating that the system is going offline.

The parent process *(parent1)* sits in an idle loop checking the *sys.Online* variable, when this indicates that a signal has been received, (i.e. the system is shutting down), it sleeps[1] for a short time, enabling the child process *(child1)* to kill all its threads. The parent process *(parent1)* then cleans up all of the IPC[2] methods used and then exits. This is done so that the program does not leave allocated system resources when it exits and ensures that there is no *core dump*[3].

[1] *sleep(time)*, A Unix system call that suspends a process for *time* seconds.
[2] Inter Process Communication
[3] If a process performs some violation or gets corrupted, Linux will store an image of the process on the hard disk.

Earlier versions of the program just deleted the shared memory segment, but then the child processes would try to access the shared memory after it had been de-allocated, which resulted in a core dump. Similarly killing all the processes resulted in resources that were not freed.

All Interprocess Communication (both that which was used and those methods which were discarded) were based on the examples laid out by Brian Hall [BG1]. This guide provided much insight into the inner workings of the SysV IPC that is used by Linux (as well as other flavours of Unix). The use of IPC allowed the control program and the user interface to communicate without the need for code hacking.

Below is an extract of the signal-handler code showing how the threads are killed as described above:

```
/*************************************************************
  The Signal Handler that will cause the pro-
gram to exit gracefully
  by killing all the threads, detaching and delet-
ing the shared memory.
 *************************************************************/
void sigint_handler(int sig)
{int ret;
 if (pid==0)
   {
    printf("Now to exit gracefully . . . \n");
    ret=pthread_kill(udlthread, SIGKILL);
    ret=pthread_kill(cgi2ctlthread, SIGKILL);
    ret=pthread_kill(ackthread, SIGKILL);
    ret=pthread_kill(datathread, SIGKILL);
    for (int j=0;j<((*sys2).rules.length()-1);j++)
        {
         ret=pthread_kill(ctlthreads[j],SIGKILL);
        }
    for (int i=0;i<(*sys2).numRCs; i++)
        {
         ret=pthread_kill(threads[i],SIGKILL);
        }
   }
```

```
    else
      {
       (*sys2).Online='N';
      }
   }
```

While I describe the methods used and their motivations, I have not gone into great depth with regard to the programming intricacies of the structures, classes, algorithms and their implementation since this project is oriented toward the embedded environment rather than applications or systems programming. The only exceptions are the core functions which I felt needed to be included to highlight some of the obstacles I had to overcome.

6.2 Update Thread

The 'parent' thread *(child1)* creates an individual thread *(threads 1..n)* to handle the updating of each Field Controller. (Currently the limit to the number of Field Controllers is 100. This is an arbitrary limit that had to be imposed so that a static array could be used to keep track of the thread IDs.) Every few seconds the thread requests an update of each individual FEP's data. The thread then 'sleeps' for a predetermined amount of time and starts the request cycle again. It should be noted that only FEPs that are listed in the configuration file will be checked. As each thread starts, it outputs a line indicating which Field Controller it will be handling. This is a usefull debugging aid which can be checked by a human operator each time the program is started. The code used to actually transmit the requests to the Field Controllers is described in *'Network Interface Code'* .

6.3 Listen Thread

This thread is created by the 'parent' *(child1)* to listen on the predefined UDP port (61440) for data from the Field Controllers. The thread opens a file handle and binds to port 61440 and waits for an incoming UDP datagram. If an update response is detected, the data is parsed and entered into the system data structure. If it is unable to bind to the port for some reason then it will end the application and print an error message to <*stdout*>. It should be noted here that I don't know what will happen if this program is run as a daemon, since I have relatively little experience in dealing with the more complex issues surrounding Unix-based operating systems. Again the actual code used to listen to the port is described in 'Network Interface Code'.

6.4 Acknowledgment Thread

This thread is used to listen for acknowledgments from the network (UDP port 61441) for when a particular channel on an FEP is switched on or off, this is then used to stop the persistent transmission that is used by the control program to ensure that a device responds. The acknowledgment thread passes a signal to the to the Control Thread to indicate that a device has responded.

6.5 CGI-Interface Thread

6.5.1 The initial attempt

Initially this thread was used to provide a user interface via dynamically generated web-pages. The thread listened for data on a FIFO in the */cgi-bin/* directory of the webserver. Data was written to this FIFO by a perlscript executed by the webserver. The arguments sent to the perlscript were communicated via the FIFO to the CGI-Interface thread where they were decoded and a dynamic html page was generated. This HTML code was then sent back to the perlscript which returned the webpage to the browser concerned. As far as I can figure, with my limited script experience, there was little security risk involved, since the perlscript did not execute any code and hence the probability of someone feeding corrupt data to the script to get it to crash and hence create a security hole was minimal. The data received from the web-browser was not parsed at all by the perlscript, merely passed to the FIFO. If the data received by the CGI thread did not yield what was expected, then an error page was returned. Since all the .html pages were generated by the thread in any case, there is no possibility of users entering incorrect data accidentally. It would have required a conscious effort on the users part to alter the data sent to the perlscript, the only possibility for errors was in the configuration files, but these should not have altered in the day to day operation of the system.

6.5.2 The second attempt

While the first attempt at a CGI interface worked, it had the drawback that only one user could could connect at a time, unless more FIFOs were used. It was decided to implement a shared memory block by which a CGI script could retrieve data from the system and generate its own web-pages. The first attempt at this new approach involved using Perl, but this was scrapped for two main reasons:

- Firstly I had no way of navigating a structure generated by C-code from within Perl.
- Secondly the perl SysV IPC functions did not return IPC keys consistent with those returned by the SysV IPC functions in C.

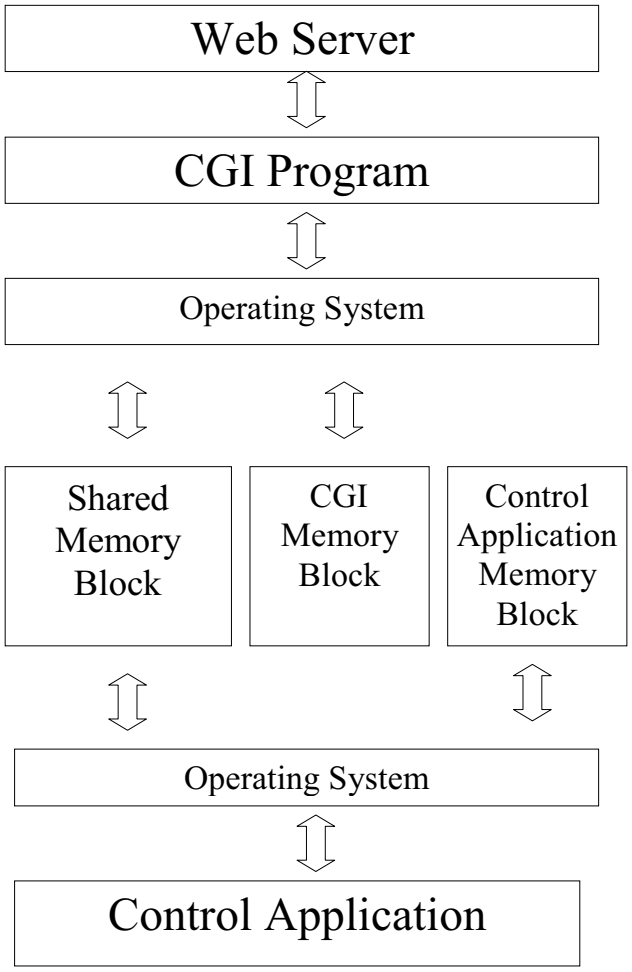


Figure 6.2: Accessing the Shared Memory Block by the Control Program and CGI interface

It was then decided to write a second C-program that would act as a CGI-script. Functions[4] were written so that the generalised data passed to the program from a web-server could be parsed as variable/data pairs. (Since the CGI specification stipulates both the variable name and its value are passed, in no particular order, from the webserver to the CGI application. That is *http://www.server.com/cgi-bin/cgiscript?Var1=value1&Var2=value2* results in a variable with a name of *Var1* with a value of *value1* and another variable with a name of *Var2* and an associated value of *value2* being passed by the webserver running on *www.server.com* to a CGI script or executable called *cgiscript*.) This code returns dynamically generated web-pages based on the variables passed to it. These are simple web-pages that allow the states of the FEPs to be monitored and changed manually. The state displayed on these pages is the last known state of the system and changes made may take up to 5 seconds to be reflected. This is due to the finite amount of time

[4]These functions can be found in the cgivars.h file in the CGI_interface code.

that the Field Controller takes to updated its representation of the FEPs that are connected to it. This information is retrieved from the shared memory segment used by the control program. If for any reason the control program is not operational the CGI executable will detect that the memory segment with the expected key is not present and will then output a webpage informing the user of the problem. The interactions between the memory segments and the various programs is shown in figure 6.2.

6.5.3 CGI Output

Figures 6.3, 6.4 & 6.5 show the web pages generated by the CGI program for the desktop test system.

- In figure 6.3 it is indicated that the second Field controller listed was last updated at the beginning of the epoch. This is because that Field Controller has never responded to the Master Controller, hence the initial value of the variable, zero, has never been updated.
- In figure 6.4 it appears as though corrupt data has been returned for channel 0 of both FEPs. This is not so, this channel is used by the Field Controller to record the last time that the FEP responded to it. The value shown in the table is the number of seconds since the epoch.
- Figure 6.4 shows a brief status of each FEP connected to the Field Controller.
- Figure 6.5 shows an in depth view of a specific FEP, giving details as to the control status of each of the FEP outputs. It also allows the user to change the control status between manual and automatic.

In my opinion these pages provide an adequate representation of the system because they are both easy to understand and navigate. It would be possible to create an image map of the campus with links that have the correct CGI variables set to enable a direct link from the graphical layout to the pages shown in figures 6.4 & 6.5.

Figure 6.3: System Outline Page

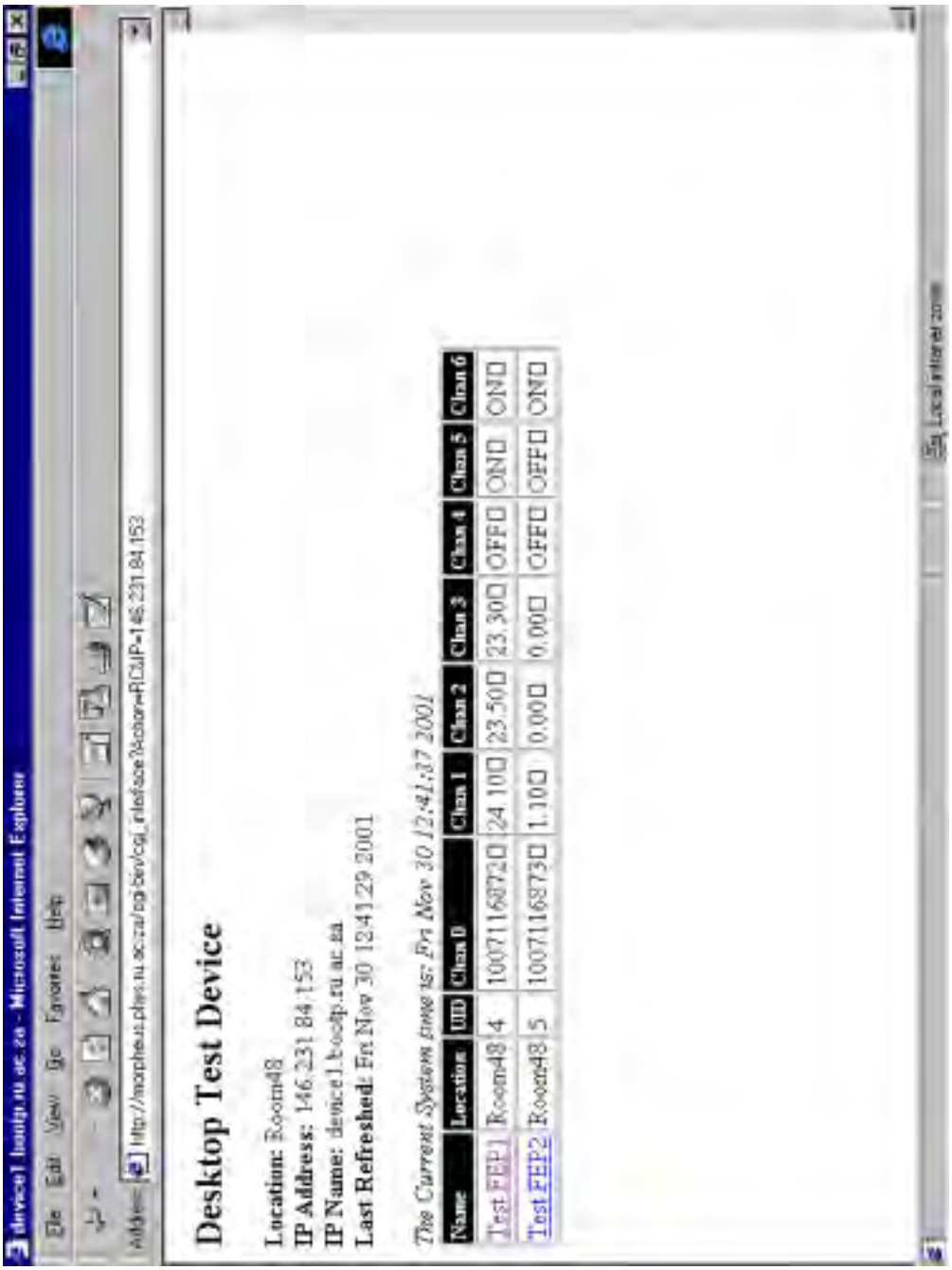


Figure 6.4: Field Controller Details

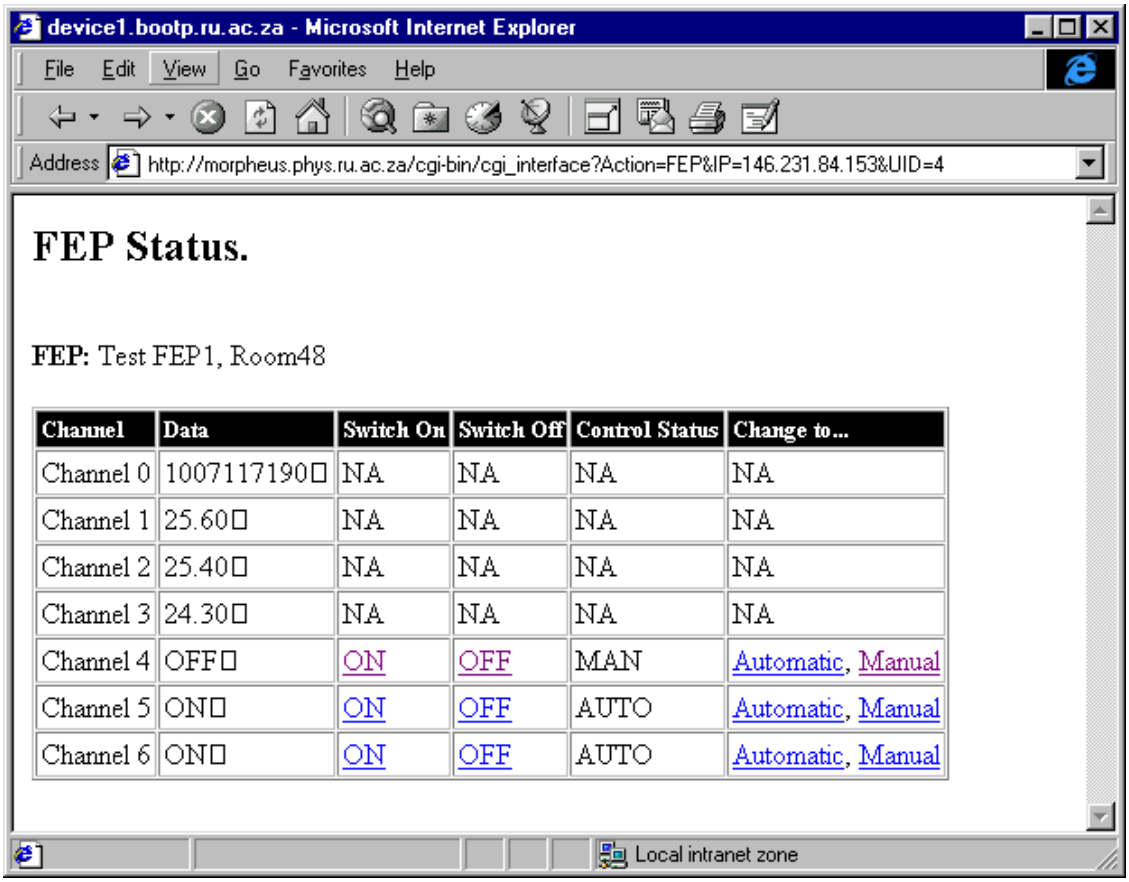


Figure 6.5: Front End Processor Control & Status Page

6.6 Class Structure

While a strong object-oriented approach was taken while designing this software package, there were several occasions when this approach was rejected in favor of a less structured approach, mostly due to my limited experience in achieving results with conventional Object-Oriented techniques in complicated examples. The 'standard' approach by most computer scientists / information systems programmers would have been to design the whole system using some sort of planning standard such as UML or Rational Rose. While these techniques may help in an environment with many programmers coding the software, they presented too much of an overhead in this case. The objects were modelled using their physical and logical attributes, but other properties were added to aid in the simplification of coding.

It should also be noted that while a restriction of nine FEPs per Field Controller is imposed by the software running on the Field Controller, no such limit is imposed in the class structure. This makes the system scalable, without having to do major changes to this software.

On examination it can be seen that the classes that are used are embedded in each other logically, but are declared publically. This was done to make the code more legible, since all the class definitions are implemented in header (.h) files, rather than splitting them between declaration in header files and implementation in code (.cpp) files. This made the code more manageable, since changes in the code do not have to be mirrored in two files. This was invaluable during development when constant changes were being made to class definitions.

6.6.1 Channel Structure

The 'channel' class contains variables to hold the channel number, FEP ID, the channel's function, whether or not it is on automatic control, its data as a string and its numerical data (if any). There are no functions in this class, which effectively means that it could have been implemented as a *struct*, but for the purposes of completeness a class was used, making it easier to alter if data manipulation functions were to be added in the future.

6.6.2 FEP Structure

The FEP class contains some house keeping information such as a *name* and *location* variable along with its ID and Field Controller variables. These were included so that some meaning could be added to the otherwise abstract container. These name and location variables are used for easy identification of specific FEPs on the web-pages and graphs. It also allows easier location of a malfunctioning device, but this does pre-suppose that the configuration files were correctly set up

though. The FEP class also contains an array of 7 channels, as defined by the *channel class*. A *last_refreshed* variable was also included to aid in the fault finding functions of the system, this is discussed in detail in section 7. Again there are no functions in this class, even though there could be primitive communication functions, such as *switch_device_on()*. This was not done since it would mean that the communications functions would have to be embedded in the data structure class, something that I wished to avoid, rather keeping the data structures and communications separated.

6.6.3 Field Controller Structure

As with the channel and FEP classes the Field Controller is purely a data structure containing the lower level classes and other information. It should be noted that arrays were used as containers for the lower level classes as opposed to linked lists. This does mean that the maximum number of FEPs per Field Controller is set at compile time, but can easily be changed and then recompiled. This was to overcome the problems and limitations imposed by shared memory. When creating a linked list, it was not possible to make sure that the space allocated for new nodes was within the shared memory boundary, which meant that it would be impossible to use shared memory as an interface to the CGI-executable without it causing a memory violation. These structures underwent many changes before the final (stable) versions were decided upon.

6.6.4 Power Control System Structure

As with the data structures it contains (i.e. Field Controllers) the *PC_sys* class makes use of arrays to overcome the problems mentioned above, but it does however use linked lists for the control rules. This was because the control rules are not to required be visible from the CGI-executable. While it may mean that the control rules cannot be changed from the web, it also means that they can not be corrupted in any way either. This was a major decision that was made without consulting the Power Control Committee, since many of the people who would use the system have little or no computer skills and much like with a Unix server, it is not advisable to give all users the equivalent of *'root'* access.

6.7 Handling External Scripts

The use of external scripts to handle some of the functionality of the system allows changes to be made without a binary re-compile. All of the interfacing to the data archives via RRD (see section 6.14 on page 81) was done through the use of scripts. Executing the scripts with given parameters was accomplished using the *exec()* family of functions as provided by *stdlib.h*. The only problem with these functions is that on successfull completion they do not return because the

calling process is replaced with the new process image. This means that if the original process is not to be lost, it must spawn a child process that will then run the script. It is also necessary for the parent to wait for the child to complete execution so that it can receive the signal that the child process sends when it completes execution. If the parent does not do this then the child process becomes a *zombie* process. This is because when a process completes its execution it calls the *exit()* routine provided by the system,which releases all the system resources that were being used, but saves the process ID and exit status. This is then kept until the parent process checks this status and tells the kernel to completely remove the process.

Initially, due to my limited systems programming experience, I did not check the child's status, which resulted in the process table filling up with zombie processes and degrading system performance.

6.8 POSIX Threads Library

The POSIX (Portable Operating System Interface) threads library was used as opposed to other less portable libraries due to the fact that I knew that it would be supported under FreeBSD, the operating system that will most probably be used for the Power Control Server. It should be noted that the options *'-D_REENTRANT -lpthread'* should be used as arguments to the C-compiler to ensure the correct operation of the software. This ensures that the program is compiled as 'thread-safe'. All standard functions used are safe for use in threads, according to the their manual pages. The only exceptions to this are in the *.Readrules* and *.ReadConfig* functions of the *PC_sys* class. This acceptable is because they are used before the program is split into threads and it was simpler to use the non thread-safe functions.

The POSIX threads library, libpthreads, is released with glibc2.2.2.

6.9 Network Interface Code

The initial testing of the system's network capabilities were conducted using Visual Basic running on Windows 95^{TM}. This was done to speed up the development of the network interface on the embedded devices. While network transmissions in Linux (and other variants) can *'easily'* be accomplished by using the predefined primitives defined in the API, understanding these functions was slightly less trivial.

Due to these somewhat confusing 'man' pages and obscure programming references with regard to Unix-based sockets programming I was forced to turn to the Internet to find more helpfull

references. This is how I discovered *'Beej's Guides'*[BG2], a series of programming tutorials for Unix programmers who have a basic knowledge of what they want to do, but need help to understand the 'man' pages.

After the subtle nuances had been sorted out it was possible to begin coding the networking interface to the control program.

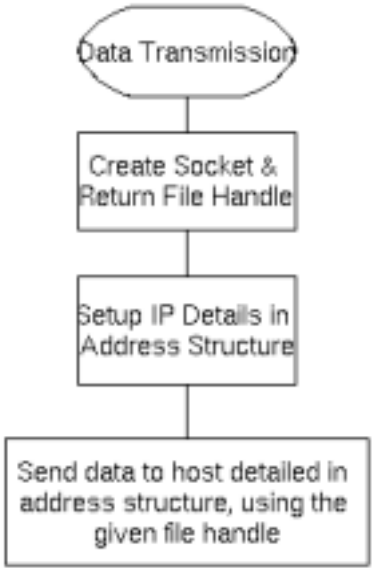


Figure 6.6: Unix Packet Transmission Process

UDP Transmission: The important fact to remember here is that the UDP port number that we transmit from cannot be the same as the port that is used to receive datagrams from the Field Controllers. The port number is actually irrelevant in this case. The steps can be seen in the flow diagram shown in figure 6.6.

The system call to open a socket is *int socket(int domain, int type, int protocol);* where *domain* is the protocol (in this case *AF_INET* was used which resolves to IP version 4), *type* is the type of data transfer used (stream or datagram transfer) and *protocol* sets the layer 4 protocol used ('0' lets the operating system decide based on the previous arguments). This call associates a file descriptor with the given arguments, at this stage no port is associated with the file handle.

Details such as destination IP address, destination port and protocol type are then set up in the *sockaddr_in* structure. (For a more in depth understanding please consult the source code available on the CD-ROM).

The data is actually sent using the *int sendto(....)* system call. This call is used to send data using the given file descriptor.

UDP Reception: The reception of UDP datagrams is very similar to their transmission, except that the file descriptor is bound to a specific port using the *bind()* function, and instead of using the *sendto()* function the *recvfrom()* function is used to store data from the received datagram in a given buffer. The source code for the network related functions can be found in the 'network.h' file.

6.10 BOOTP Server Configuration

While a large amount of time was spent ensuring that the Field Controller was independent enough to not need human intervention it does still rely on obtaining relevant configuration information from a BOOTP server. To enable a correct BOOTP response for a specific Field Controller, I had to register a MAC address with the Rhodes IT division. Not wanting to cause any problems with the Rhodes IT infrastructure I decided not to set up a local BOOTP server, but rather gave the specifics of what was needed to the IT division and let them handle the configuration of the Rhodes DHCP server to handle requests from Field Controllers. The only options that need to be set for the Field Controllers are:

1. MAC address: The MAC address for each Field Controller has to be registered. While strictly speaking a block of MAC addresses should be purchased from the IEEE[5] , I opted to use unlikely MAC addresses that were not present on the Rhodes network. (The test prototype Field Controller was registered as device1.bootp.ru.ac.za with a MAC address of 00:00:00:AB:CD:EF).

2. IP Address: The Field Controller needs an IP address that is valid for the subnet to which it is logically connected. Since the Field Controllers will be spread throughout the LAN, IP addresses will vary according to the subnet that they are connected to.

3. Host and Domain name: Each Field Controller needs to have a registered host name, so that the names given in the configuration files of the Master Controller can be associated with the relevant IP addresses. IP addresses are not used in the configuration files, since they are not necessarily static and may change if the LAN is restructured.

4. Next Server: This field in the BOOTP configuration usually indicates the next server that a device is supposed to use in its boot process. For the Field Controllers this is set to the IP address of the Master Controller.

5. Filename: This field is usually used to indicate the name of the boot image file that the device should use in its boot sequence. In the case of the Field Controllers this is parsed but not used and could be easily adapted to provide configuration information about default control states and rules. (It was decided to not allow remote configuration of Field Controllers due to the security risk.)

An extract of the DHCP configuration files from the Rhodes servers is available in the appendices, giving the configuration information for the prototype Field Controller.

6.11 Control Software

The control algorithm is relatively trivial, utilising an upper and lower threshold to determine what sort of action should be taken. A thread is spawned for each control rule, with the effective control rate, as well as other variables being read from the *'preferences.cfg'* file. This was done so that the effective policy regarding power control could fine tuned when properly implemented. At the time of writing this document the Power Control committee has still not come forward with a policy document. The Master Controller control algorithm is in effect the mirror of the control software running on the Field Controller described earlier.

6.12 Configuration File format

The configuration file format is relatively simple and self-explanatory and can best be explained by looking at the sample configuration files *preferences.cfg*[6], *system.cfg*[7] and *control.cfg*[8]. These are available in the appendices as well as on the CD-ROM.

6.13 IP Naming Conventions

In order to provide a hierarchical naming convention that has some meaning, it was decided to name the devices according to the buildings that they will be situated in. According to the official Rhodes University maps each building had a unique identifier consisting of an alphabetical character and a number. For example:

A4 Physics Building

[5]Institute of Electrical and Electronics Engineers.
[6]This file specifies some of the Master Controller's variable parameters
[7]This file specifies the devices connected to the system.
[8]This file specifies the control parameters

C8	Hazardous Materials Store next to the Chemistry building
D13	Atherstone House
D14	Smuts House
D13a	RU1 Substation

By using these naming conventions for a Field Controller (based on where it is situated) it provides a common reference scheme. By just looking at a devices name the exact location can be pin-pointed. This will result in names like: FC1.D14.pcon.ru.ac.za, for the first field controller in Smuts house that is part of the power control system. This naming system allows for more than one Field Controller per location, as well as differentiating between the power control system and any other system that may later be used, such as network based security, which would have a device name such as door1.D14.security.ru.ac.za, for a specific door in Smuts house that uses a networked security system. The actual IP addresses are immaterial, since they are allocated by the IT division based on where the device is connected.

The Rhodes System Administration was consulted about the naming convention, since they would have to handle the changes to the DNS records, but otherwise this was a unilateral decision based only on ease of use and understanding. The naming of the devices in no way affects the operation of the system, as long as the DNS records, BOOTP server and power control setup are all consistent.

6.14 RRDTool

RRDTool[9] (Round Robin Database Tool) is a popular open source tool used mostly for generating graphs of network traffic. It is rapidly superceding the success of its predecessor MRTG[10] (Multi Router Traffic Grapher).

In the early stages of development, a prototype FEP was connected to a PC via a RS232 serial link to monitor the power consumption in the Physics Department[11] and MRTG was used to graph this data. Once the Field Controller testing had begun, this FEP was removed and the monitoring was stopped. The somewhat complicated interface (in my opinion) to MRTG prompted me to either write my own graphing functions or find a better open source tool. This led to the discovery of RRDTool. The main reason that this tool was chosen was its ease of use and popularity among the Internet community. It made sense to use a tool that was not obscure nor badly supported.

6.14.1 What does RRDTool do?

The simple answer to this question would be 'It draws pretty graphs', but this would be a gross understatement of its capabilities. While I have barely scratched the surface of RRDTool's functionality in its application to this project, it provides many data manipulation functions and other features far to numerous to mention here. One of the biggest bonuses is that it can be run from the command line, or its Perl-module can be used to access its features from within Perl.

6.14.1.1 Data

RRDTool allows the capture of data from a variety of sources, which it stores in a .rrd file (Round Robin Database). Each of these files contains one or more RRA's (Round Robin Archives) which store the data according to some predetermined archiving strategy. The possible archive functions are MIN, MAX, AVERAGE and LAST, which store either the minimum, maximum, average or last value of some time-series. It is possible to set the *resolution* of the stored data by setting how many aggregated points are stored, and how many primary data points are used to create one aggregate point. It is also possible to make RRDTool interpolate data if a datapoint is not be entered. When entering data a time is given as well as the values of all data sources. So if there were 3 data sources having respective values of 34.6, 78.1 and 21 at a time of 1005039296[12] it would be entered using the commands line string *rrdtool update rrdfilename 1005039296:34.6:78.1:21*. As can be seen,

[9] By Tobias Oetiker
[10] By Tobias Oetiker & Dave Rand
[11] Dept. Physics & Electronics, Rhodes University, Grahamstown.
[12] Unix style time, seconds since 1st January 1970 00:00:00.

the data is entered as raw data and RRDTool stores it according to the archiving functions specified in the .rrd file.

6.14.1.2 Graphs

Graphing is just as simple as storing data, all that is required is to set up the period to be graphed, add any titles, and specify the data sources, data archiving functions and line styles for the graph.

6.14.2 Implementation

6.14.2.1 RRD creation

The RRD files have to be created manually for each FEP defined in the *system.cfg* file. Since this is a tedious process that is prone to error, a perlscript *(create_rrd.pl)* was written to create the files based on two input parameters, Field Controller and FEP ID. The script can be found in Appendix C4.

It was decided to store averages as well as minimum and maximum values for given periods. This was done since the minimum and maximum values provide greater detail as to the patterns of power consumption.

Archives were created to store the minimum, maximum and average values for

- every minute for a day,
- hourly values for a week
- daily values for a year

The size of these files is about 150 kilobytes.

6.14.2.2 Graph generation

The graphs are automatically updated every 5 minutes, with separate graphs being generated for each FEP over the preceding day, week and year. Three sets of graphs are generated, displaying the minimum, maximum and average values. The details of the graphs can be changed by editing the perlscript *(graphrrd.pl)* called by the control program. Some graphs generated by RRDTool can be seen in section 7.5.

Chapter 7

Power Control Specifics and Related Obstacles.

Throughout development certain restrictions and problems arose that stemmed directly from the nature of the system being controlled. Some of these have been discussed briefly in the preceding sections covering the specific area that they pertained to. Here I will attempt to clarify some of my choices and tie up some of the loose ends.

7.1 Practical Control Theory

As with many areas in physics, electronics or computing, the ideal solution to a problem is not always practical or even possible. The very nature of the electrical supply system and the demands made on it made it impossible to model it using linear control theory, which would have been the ideal theoretical approach. Several physical constraints made this impossible:

- No Theoretical Model: There was no theoretical model available to base the calculations on. Experimental determination also posed a problem since there is no effective record of all the appliances that are connected.
- Non-Deterministic nature: While power consumption follows definite general trends, the mere fact that consumption is based on fickle and sometimes irrational users (i.e. using heaters in mid-summer) means that it is impossible to model the system because of the users' behaviour.
- Even if such a model did exist, due to its complicated nature, the mathematical processing involved would probably require more computing power than would reasonably be expected in a power control system.

Hence conventional control theory would have to be side-stepped and a system resembling 'fuzzy control' would have to be implemented. The formal 'fuzzy theory', being a whole branch of mathematics in itself, was well beyond my understanding and beyond the scope of this project. Hence the *'fuzzy theory 101'* approach was taken, not the most scientific approach, but successful none the less.

7.2 True Power Readings

If the system were to evolve to the point where more accurate, true power readings were needed, a power metering chip[1] could be added to the FEP. This was not considered in the project due to budgetary constraints, but if the current implementation is successful then future versions of the FEP could be expanded to include the above mentioned chip. Since all of the costly semiconductors on the FEP use sockets, the upgrading of the FEPs could be implemented with minimal parts costs except for the power metering chip. Due to the modular design of the system, this change would be transparent to the rest of the system and could be implemented only where deemed necessary.

Owing to problems like this, using some of the more specialised components would have undoubtedly caused problems when constructing prototypes, and while they could have been bought in bulk, this would have had serious implications for the project budget.

The main obstacles to making true power readings with the current components are:

- The sampling of the voltage and current waveforms could not occur at the same instant, which would produce inaccurate results. This could be solved by using two separate external ADCs triggered to start their conversions at the same time, but then a total of 6 such ADCs would be needed to cover all 3 phases, which would increase both the size and cost of the FEP.
- Interfacing directly to the mains voltage would mean that a high degree of risk could be involved. If components failed and caused a domino effect throughout the system, more than just one FEP could be damaged.
- The current microcontroller (AT90S8535) does not have a hardware multiply instruction, and the added instruction cycles could adversely affect performance. If the RMS power were to be calculated, both current and voltage conversions would have to be done on each ADC sample before the product could be calculated, which would mean a total of 3 floating point multiplications per ADC conversion. This could seriously affect performance if there is no hardware multiplication instruction available.

[1] Cirrus Logic 5460.

7.3 Component Sourcing

While one or two rather specialised Integrated circuits were used, the use of ASICs was avoided due to the erratic supply of small quantities of such components in South Africa. Even extremely common capacitors have proved to be difficult to obtain in certain footprints. The prime example of this being that no supplier was willing to supply any surface mount components in quantities of less than 1000, and while they are cheap individually, a stock of approximately 900 unused surface mount capacitors does represent a sizable expense. Unfortunately there are some areas where conventional through-hole technology would have been totally inappropriate, such as the supply capacitors surrounding the CS8900A on the Field Controller PCB.

As can be seen, none of these problems are insurmountable with money, which is precluded by the fact that the system was designed to be a cost-effective (i.e. cheap) solution to the problem of power control on the Rhodes campus. Hopefully in the long term the necessary upgrades to the hardware can be made from the money that is saved by the system. For the time being the simpler, less elegant hardware, will have to suffice.

7.4 Loading the Current Transformers

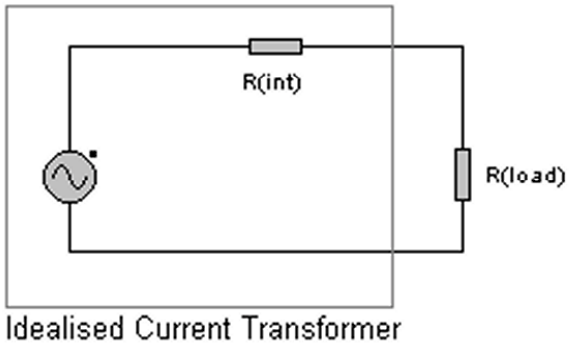


Figure 7.1: Theoretical Model Of a Current Transformer

The current transformers are neither a perfect current source nor a perfect voltage source, but can be viewed as in figure 7.1. The figure shows some internal resistance (reasonably high) in series with an ideal voltage source to model a current source. If the value of R_{load} should become high enough, in comparison to R_{int}, then the voltage source that forms part of the idealised current source cannot provide a high enough voltage to force the correct current to flow through the load. Conversely if a low R_{load} is used then a measurable voltage across the load is not obtained. Table 7.1 shows the experimental values obtained from the current transformers[2] located in the Department of Physics & Electronics. The current transformers A, B & C were attached to the three separate phases of the mains supply entering the building. While the values are mostly consistent, there is a definite drop in the measured voltage (or calculated current) when a load of 330 ohms is used.

Unfortunately this information does not indicate the maximum expected voltages across any load. It would be a fair assumption that the current measured could in fact double under very high loads. Bearing in mind that the maximum RMS voltage that can be measured by the FEP is about 1.7 v (RMS), the values of R_{load} (and the resistors controlling the feedback of the differential amplifier on the FEP) would have to be chosen so that the voltages presented to the ADC on the FEP were in the range 0v - 1.7v RMS. Since extended field trials would be needed to determine the peak values of current, the only recommendations that I can make would be to have R_{load} in the range of 10 - 68 ohms[3], with the gain of the differential amplifiers set at either 0.1, 1 or 10, depending on the

[2] Part of the previous Power Control System.
[3] Reccommended values would be from the E12 series for resistors (10R, 33R, 47R, 68R).

	R_{load}	$V_{measured}$ (RMS)	$I_{Calculated}$ (RMS)
Current Transformer A	10 ohm	0.148 v	0.0148 A
	100 ohm	1.50 v	0.0150 A
	330 ohm	4.75 v	0.0144 A
Current Transformer B	10 ohm	0.048 v	0.0048 A
	100 ohm	0.455 v	0.00455 A
	330 ohm	1.420 v	0.00431 A
Current Transformer C	10 ohm	0.143 v	0.0143 A
	100 ohm	1.420 v	0.0142 A
	330 ohm	4.49 v	0.0136 A

Table 7.1: Variation of Current Transformer output with various loads.

highest expected value for the current being measured.

7.5 Viewable Results

In the figures (7.2, 7.3 & 7.4) sample graphical outputs are shown, as generated by RRDTool's graphing functions. Unfortunately they are of relatively low resolution, but some pertinent information about the system is conveyed.

- Firstly a signal generator (set to output a 50Hz sine wave) was connected directly to one of the current measuring inputs of an FEP. This was used in place of a current transformer due to the fact that the signal generator could not provide enough current through the primary winding of the current transformer[4] to produce a significant output from the current transformer.
- Figure 7.2 shows the recorded values over a 24 hour period, with the zero-level included.
- Figure 7.3 shows those same values, but only over the y-axis range for which there is data.
- Figure 7.4 shows the difference between the minimum and maximum values recorded over each 10 minute interval for the same 24 hour period.
- The time constant for the averaged data in the FEP is of the order of milliseconds, so any periodic averaging effect that the FEP would induce would be impossible to see in the graphs. If the raw data from the FEP or Field Controller is studied, then it is apparent that the RMS readings from the FEP are relatively constant over consecutive readings, hence indicating that the FEP data capture and manipulation are sound.
- Throughout the period shown there were no software failures, (i.e. stack overflows) or device resets. (The software was adjusted so that operation would cease should an error condition occur, instead of restarting as would normally be the case.)

- In the area marked as 'A' on the graphs it can be seen that the values were more constant than elsewhere, this is not caused by the same data being repeatedly entered into the RRD file, because on the larger scale graph (7.3) small variations can be seen.

- Small errors are expected since the sampling period of the 50Hz wave is not 100% precise due to small interrupt latencies and the signal generator used would not remain stable at such a low frequency.

- Now if a closer inspection is made, from equation 4.5 (page 41) and using the constant of $1.52737\frac{mA}{ADCValue}$, which was being used in the FEP at the time of capturing the data, and the maximum error obtained (graph 7.4) of about 5mA, a maximum error of 3.27 ADC units is obtained. (The maximum obtainable ADC value for an RMS measurement would be 362, since the maximum ADC value obtainable with a 10bit ADC is 1023). So from this we can see that the maximum error obtained is less than 1%.

- To see the data and its variation in the proper context graph 7.5 shows the captured data on a set of axes that cover the full possible range of current values. It can now be seen that the variations that appeared in graphs 7.2 & 7.3 are almost insignificant when viewed over the entire intended range .

7.6 Failure Notifications

The system is currently capable of detecting failures and sending alerts to appropriate service personnel via email. The failure detection differs between FEPs and Field Controllers. There is also support included for interfacing to the *Big Brother* system that Rhodes uses on its servers. This was done by writing the system status to a text file that can be read in by a script to send the relevant data to the Big Brother server. This was not implemented due to the fact that I could not get the *Big Brother* server running reliably on my test platform. The only reason that the *Big Brother* system was considered is because it is capable of sending SMS's to different people based on criteria setup in the *Big Brother* configuration files. This will be done when the system is implemented properly to further help the integration of IT related resources on campus.

7.6.1 FEP Failure Detection

This failure detection is spread between the Master Controller software and the Field Controller software. The Field Controller writes the local time into channel 0 in the locally stored data structure for a particular FEP on reception of valid data from the FEP. This is then read by the

[4]The only current transformer available had a winding ration of 1:1000.

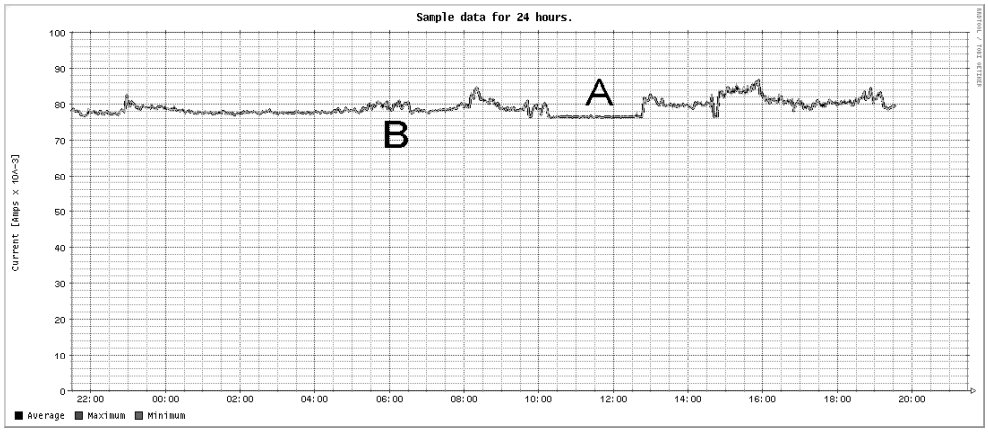


Figure 7.2: Sample Data

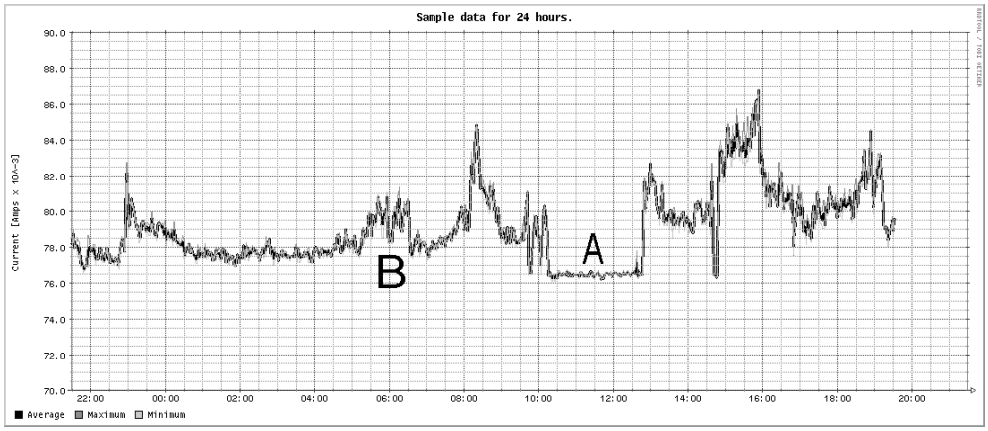


Figure 7.3: Sample Data (close-up)

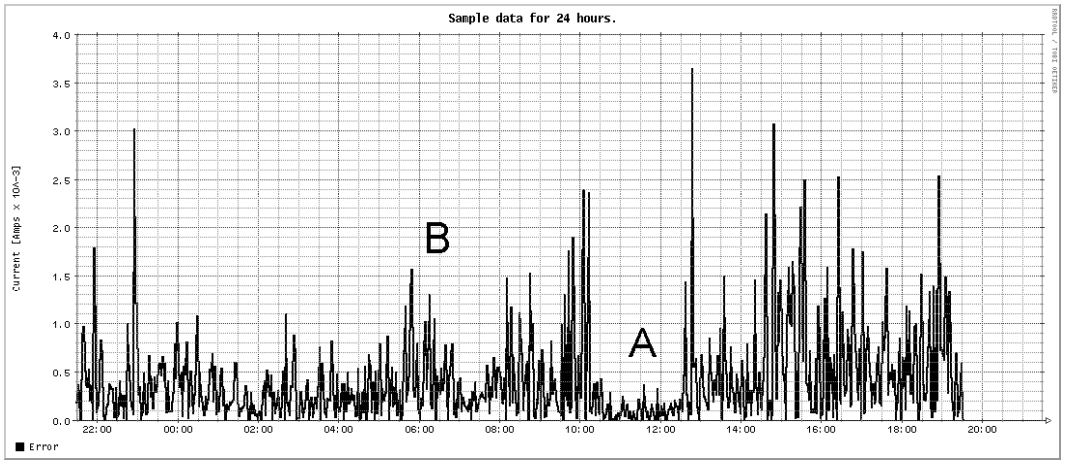


Figure 7.4: Sample Output (Error)

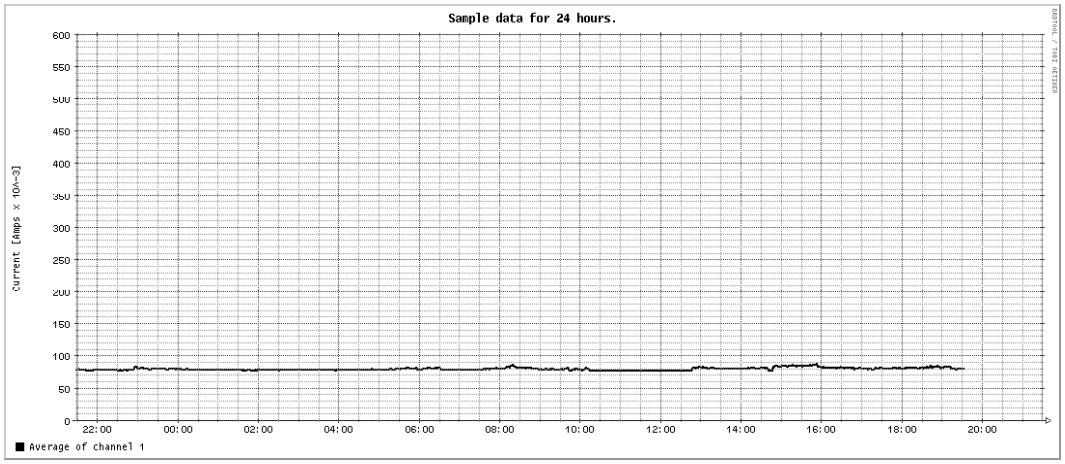


Figure 7.5: Sample Output (appropriately scaled)

Master Controller when it requests data updates from the Field Controller. The 'alarm thread' then compares this time with the Master Controllers time and runs the mail script should the times differ by more than the time specified in the preferences.cfg file. An e-mail notification is only sent when the system first detects the failure, but the data written to the Big Brother data file reflects that the FEP is offline until such time as the problem is rectified.

7.6.2 Field Controller Failure Detection

This failure detection is based on whether or not the Master Controller has received a valid response from the Field Controller in question within the stipulated time. If a Field Controller is detected as malfunctioning then the FEPs attached to it are ignored, since it is not possible to determine if they have failed or not.

7.7 Running the control software

Due to the fact that is was unlikely that the control software would be run at a terminal, it was decided to write a bash[5] script to handle the correct start-up and shutdown.

Firstly when starting the program from the script, the standard output is redirected to /dev/NULL, so that no debugging information is displayed. (The program can still be run directly if the debugging output is desired).

To correctly shut the program down the script should be called again with the *stop* argument, the script then reads the .pid file written by the control software indicating the parent process' PID, to which the script then sends the SIGINT signal causing the signal handler in the control program to shut the program down correctly.

This could be set to execute in one of the run-levels by making the appropriate entries in the */etc/rc.d* directory or started manually.

[5] Bourne Again SHell

Chapter 8

Conclusion

In conclusion, the system developed is capable of monitoring and controlling multiple devices over an IP-based Ethernet network in accordance with the specifications of these protocols. As of writing this document the trial system has not been installed, but the desktop prototype functions correctly, even during real and simulated network failures and power outages. It has operated within design specifications for over two weeks of continual operation. The test was terminated in order to change parameters, not because of a failure.

Laboratory trials have shown that the FEPs are robust enough to withstand the demands of switching the inductive loads of the contactors, as well as resistive high current loads (up to 10 amps), while maintaining accurate data measurements.

The prototype has demonstrated that the system matches the design criteria, specifically:

- Cost - The cost of the of the components for the system is suitably low. The Field Controller costs about R800, with the FEPs (including enclosures) costing about R400 and the Distribution Boards (excluding fibre transceivers) costing about R400.
- Modularity & Scalability - The only limits on scalability are the generous constraints of 9 FEPs per Field controller and 100 Field Controllers (as limited by array size) connected to the master controller. As far as modularity is concerned, it is self-evident from the hardware structure detailed that all the components are modular in nature, allowing replacement of faulty modules or addition of new modules without having to modify the rest of the system.
- Ease of use - This still has to be proven, but the web-based interface and graphs present both a clear picture of the system and an easy to understand user interface. The only questionalble aspect of the ease of use is the installation and configuration phase, but anyone with Unix/Linux experience should be able to get the system up and running with a minimum of effort.

This project needs further development before it can be released as a commercial system. Most of this development would be at the hardware layout level to provide a more professional production model, with minor code revisions needed to present a standardisd API.

The microcontroller code could have been written entirely in assembler, thus reducing some of the overheads, but this would have meant that the code would have been extremely difficult to maintain, modify or interface to without an in-depth knowledge of the Atmel assembly language.

A small-scale implementation is planned for 2002 in one of the larger residences on campus. The necessary infrastructure is already in place and all that is needed is to finalise an enclosure for the Field Controller. This implementation will initially only monitor the power consumption of the residence. It will also monitor the consumption at the substation to gauge the power needs of the university. At a later stage load shedding will be added as more residences are connected to the system.

Overall, the system meets its requirements, while maintaining a minimum cost, through the use of simple, yet reasonably powerful, microcontrollers.

Bibliography

[BG1] Beej's Guide to Unix Interprocess Communication. Brian Hall. (Version 0.9.3 14 May 1997), http://www.ecst.csuchico.edu/~beej/guide/ipc

[BG2] Beej's Guide to Network Programming. Brian Hall. (Version 2.2.1 25 June 2001) http://www.ecst.csuchico.edu/~beej/guide/net

[Hons] Embedded Communications- Using Ethernet as a transport for Embedded Communications. Anthony Sullivan, 1999

[RFC 768] RFC 768 - User Datagram Protocol. J Postel, 1980

[RFC 1180] RFC 1180 - A TCP/IP Tutorial. T Socolofsky, C Kale, 1991

[RFC 951] RFC 951 - Bootstrap Protocol. Bill Croft, John Gilmore, 1985

[RFC 2131] RFC 2131 - Dynamic Host Configuration Protocol. R Droms, 1997

[RFC 2132] RFC 2132 - DHCP Options and BOOTP extentions. S Alexander, R Droms, 1997

[RFC 1700] RFC 1700 - Assigned Numbers. J Reynolds, J Postel, 1994

[RFC 826] RFC 826 - An Ethernet Address Resolution Protocol. D Plummer, 1982

[RFC 2153] RFC 2153 - PPP Vendor Extensions. W Simpson, 1997

[Fairchild] MOC3162 Application Note, Fairchild Semiconductor

[Cirrus 1] CS8900A Product Data Sheet, Cirrus Logic, 1999

[Cirrus 2] CS8900A Revision B Errata, Cirrus Logic, 1999

[Cirrus 3] AN181 - Using the CS8900A in 8-bit Mode, Cirrus Logic, 2000

[Atmel 1] ATMega103 Product Data Sheet, Atmel Corporation, 1999

[Atmel 2] AT90S8535 Product Data Sheet, Atmel Corporation, 1999

[Pearman] Power Electronics - Solid State Motor Control, Richard Pearman, 1980.

[HI&H] The Art Of Electronics 2nd Edition, P Horowitz, W Hill, 1989

[DSE] Dedicated Systems Encyclopaedia.
http://realtime-info.be/encyc/publications/faq/rtfaq.htm

[ICCAVR] ICCAVR C Compiler Manual. Imagecraft, 2001

[Agilent 1] Low Cost, Minature Fibre Optic Components HFBR-0400 Series Datasheets, Agilent Technologies, 1999

[RFC 791] Internet Protocol - DARPA Internet Program Protocol specification, 1981

Chapter 9

Glossary

CISC: - Complex Instruction Set Computer. A computer architecture that ustilises a large number of complex instructions, each taking several clock cycles to execute.

Data Link Layer: - The layer in the ISO model responsible for error-free transmission and establishing logical connections between stations.

Dedicated System: - A system where the functionality is once and for all tied up into the hardware and software. (Real-Time systems encyclopedia - http://realtime-info.be)

DoS: - Denial of Service. This is a form of attack on a network-based system whereby an attacker attempts to overload the system resources by various means, including, but not limited to the use of ping floods.

Embedded System: - A system that is enclosed in another system and makes an essential part of it. OR Hardware & Software, which forms a component of some larger system and which is expected to function without human intervention. (Real-Time systems encyclopedia - http://realtime-info.be)

Fault-Tolerant system: - Systems that continue working in all circumstances (except for physical destruction). OR The ability of a system or component to continue operation despite the presence of hardware or software faults. (Real-Time systems encyclopedia - http://realtime-info.be)

Frame: - An Ethernet string of bits that includes the Destination Address, Source Address, optional length field, data and pad bits.

IEEE: - Institute of Electrical and Electronics Engineers.

Interrupt Latency: - The time taken from when an interrupt occurs to when it is serviced.

ISO: - International Standards Organisation. A worldwide federation of national standards bodies promoting standardisation with a view to facilitating the international exchange of goods and services.

MAC: - Medium Access Control.

Network Layer: - The layer in the ISO model responsible for addressing and control functions (eg routing)neccesary to move data through the network.

OUI: - Organisationaly Unique Identifier. This is the identifier assigned by the IEEE to network adapter manufacturers.

Physical Layer: - The layer in the ISO model responsible for the transmission of unstructured bit streams over a physical medium.

Ping Flood: - An often misused application of the *ping* command whereby ICMP ping packets are sent to a host repeatedly, without waiting for a reply between transmissions.

RISC: - Reduced Instruction Set Computer. A computer architecture that uses a reduced instruction set of simple instructions that usually only require only one clock cycle to complete.

SPI: - Serial Peripheral Interface. A three wire serial protocol using two data lines and a clock signal for interfacing peripheral chips to microprocessors.

Transport Layer: - The layer in the ISO model responsible for reliable, transparent transfer of data between end points on a network.

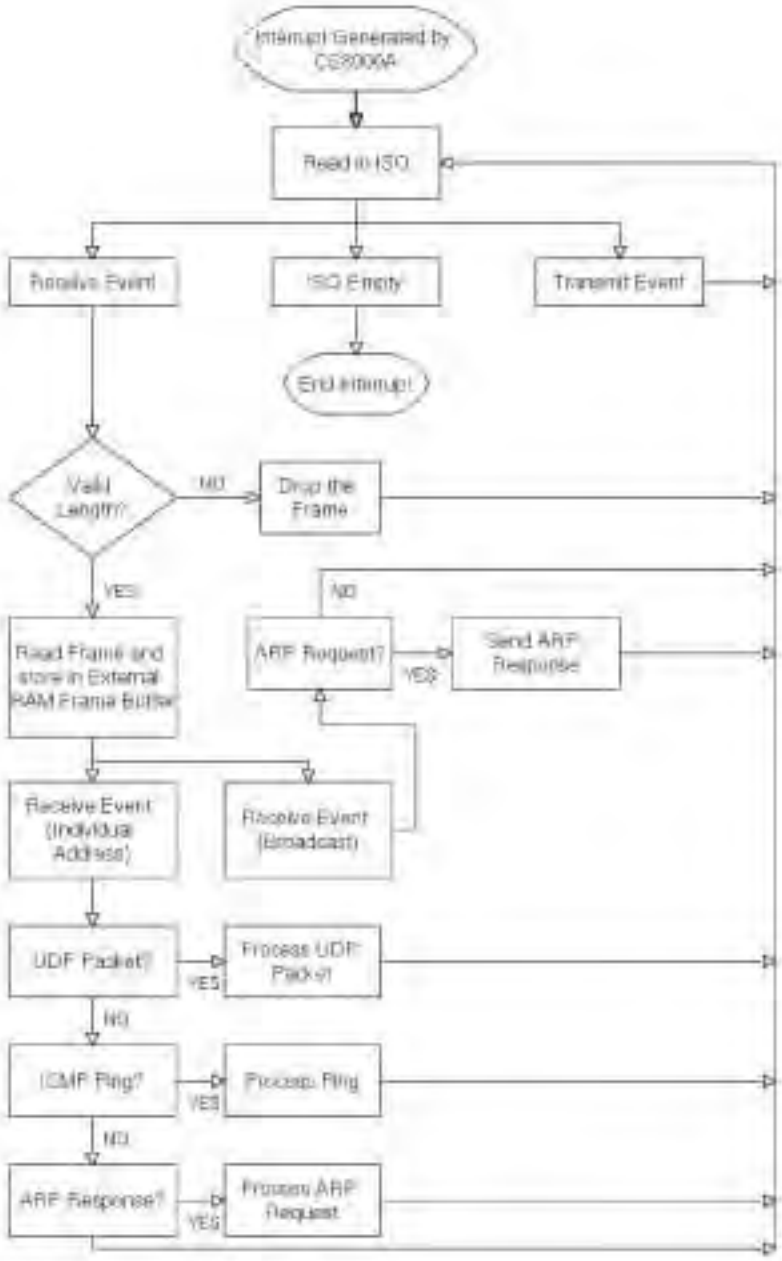
Zombie Process: - A zombie process is a process that has finished executing and will be deleted at a later time. (Also known as *defunct process*)

Chapter 10

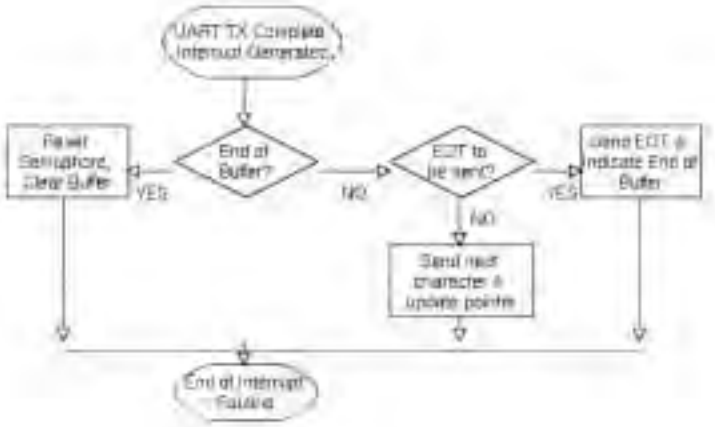
Appendices

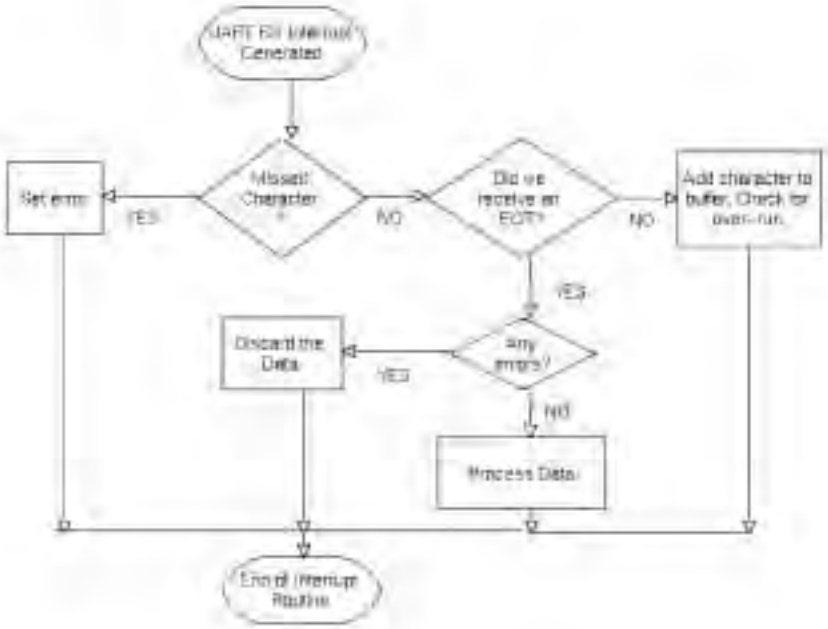
10.1 Appendix A - Field Controller

10.1.1 Appendix A1 - CS8900A Interrupt Routine Flow Diagram

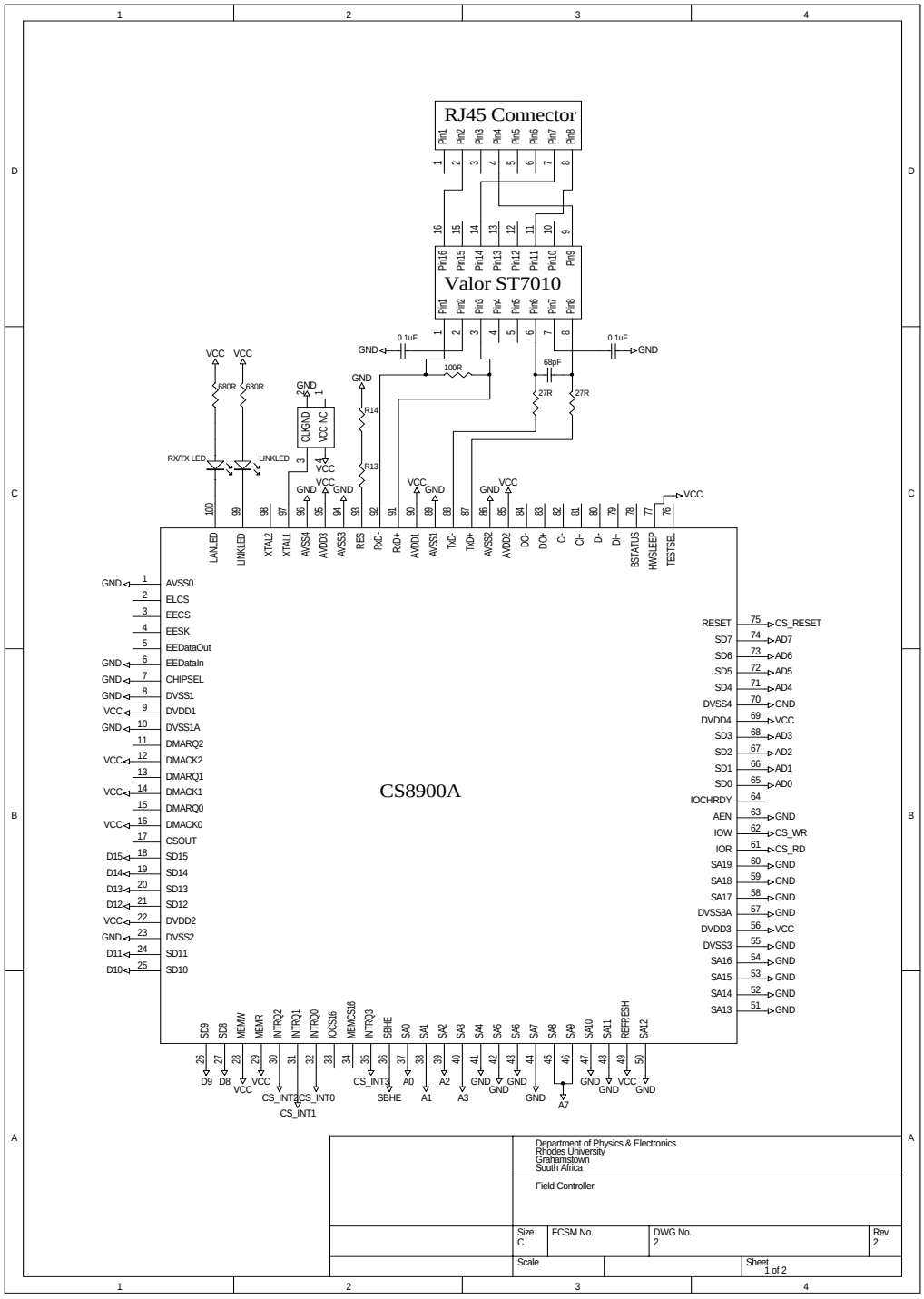


10.1.2 Appendix A2 - UART Interrupt Routine Flow Diagram

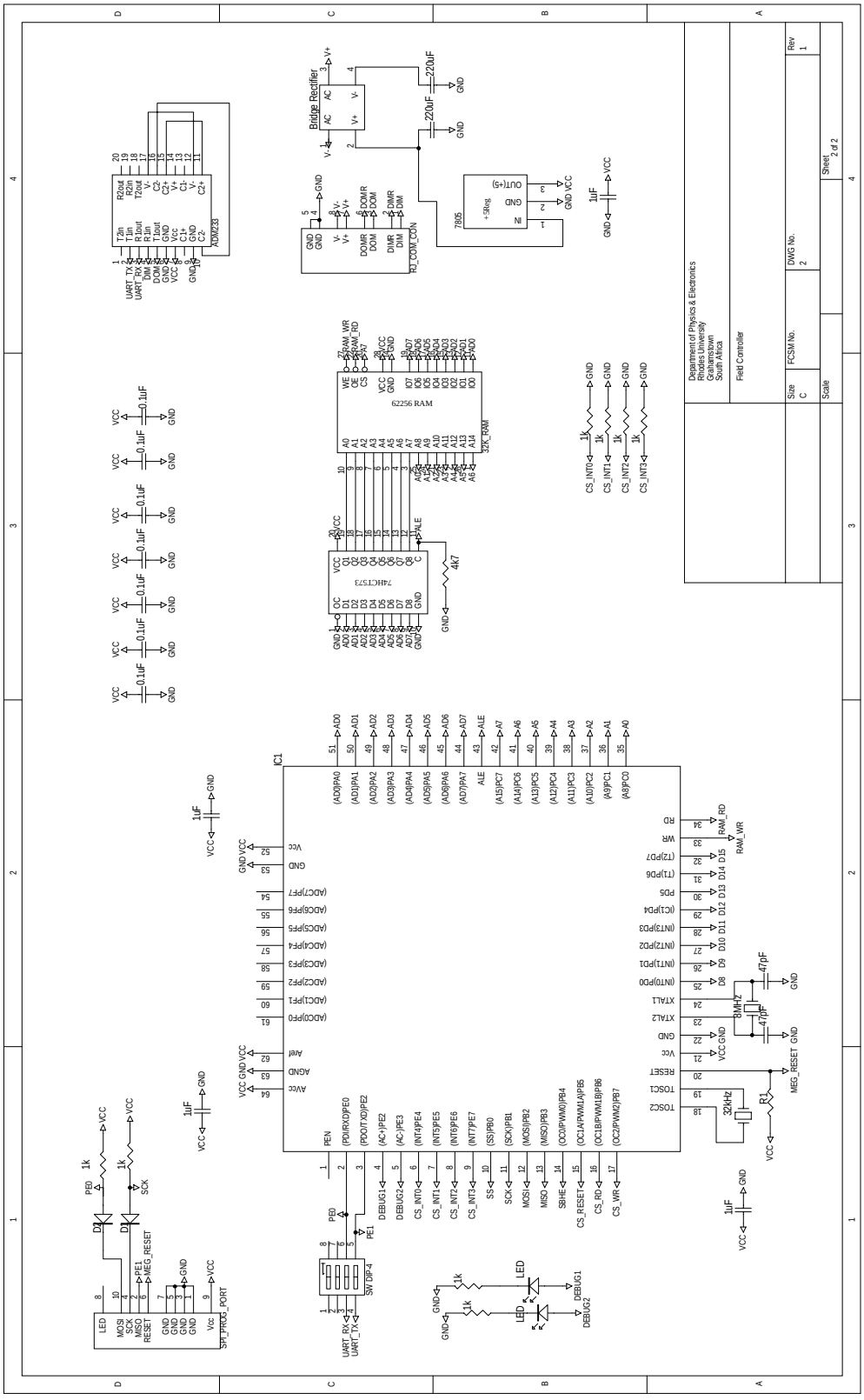




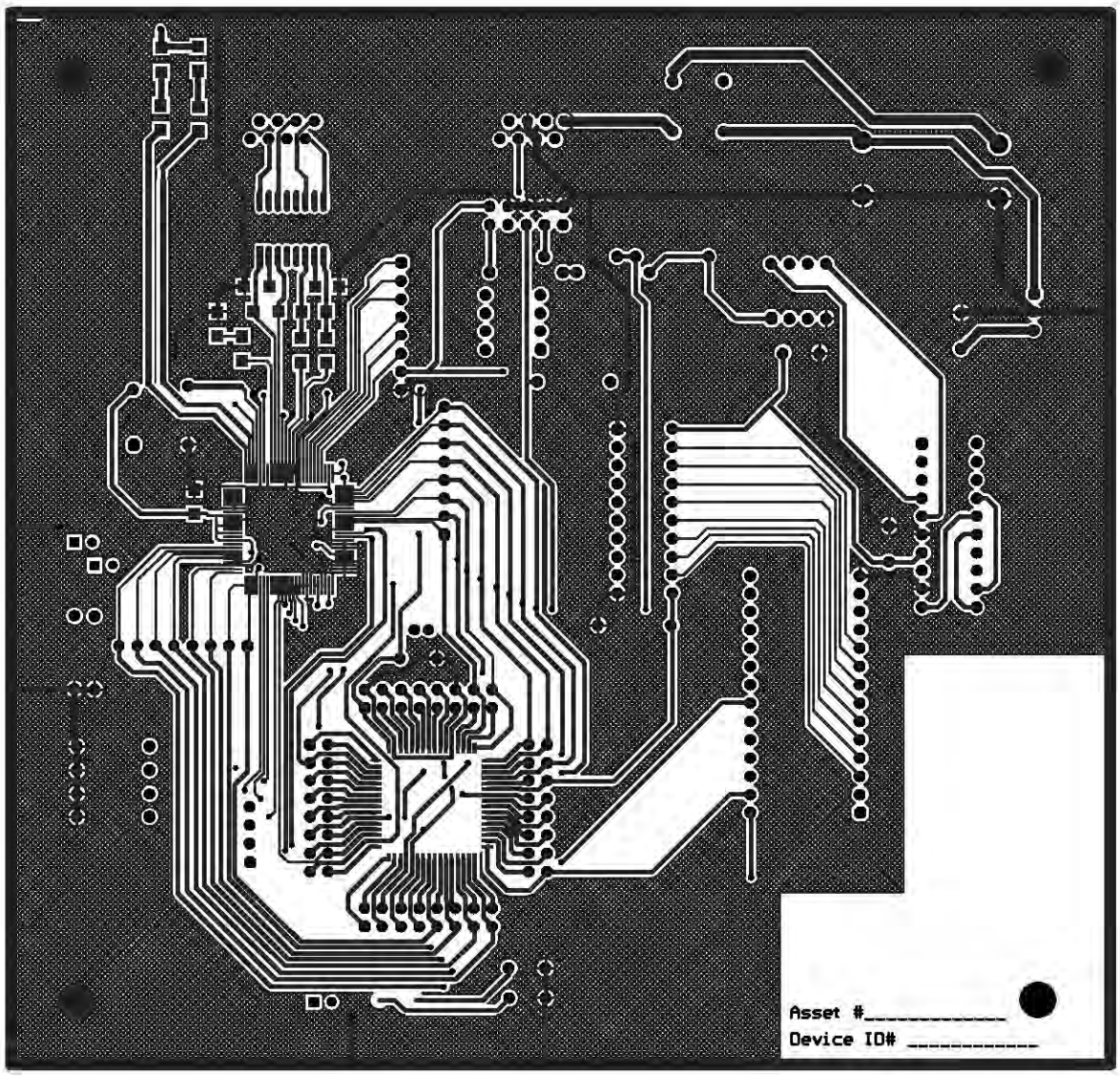
10.1.3 Appendix A3 - Field Controller Schematic (1/2)



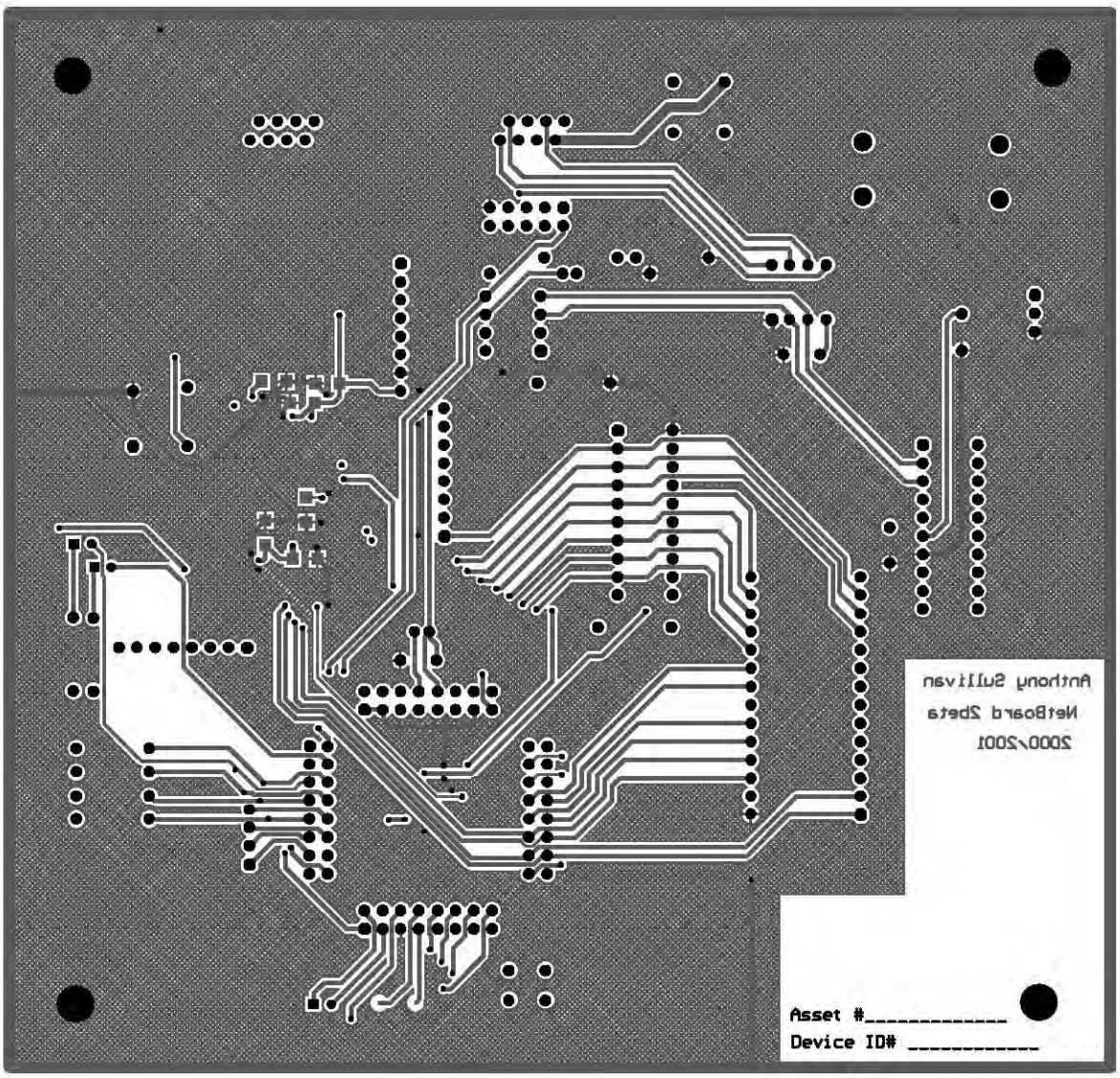
10.1.4 Appendix A4 - Field Controller Schematic (2/2)



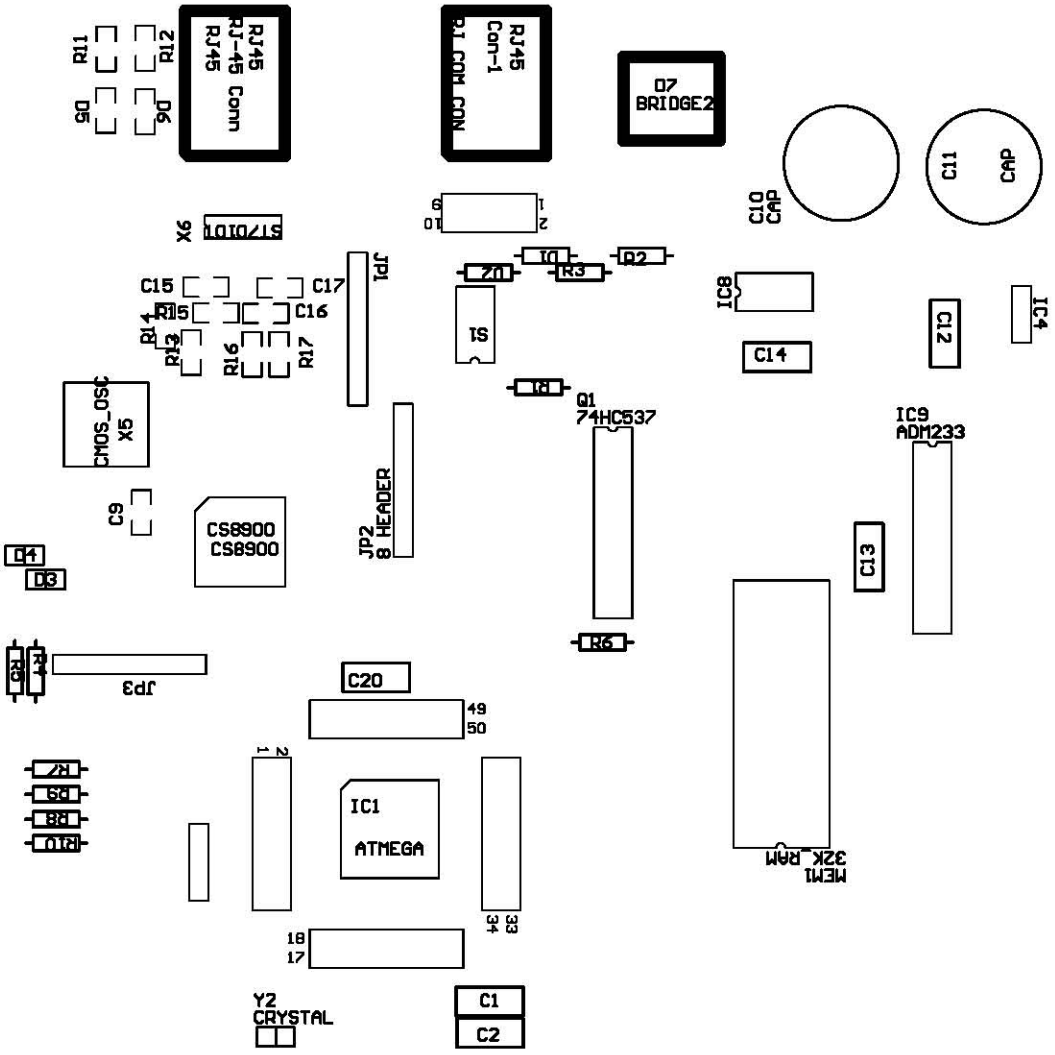
10.1.5 Appendix A5 - Field Controller PCB Foils (Top Layer)



10.1.6 Appendix A6 - Field Controller PCB Foils (Bottom Layer)

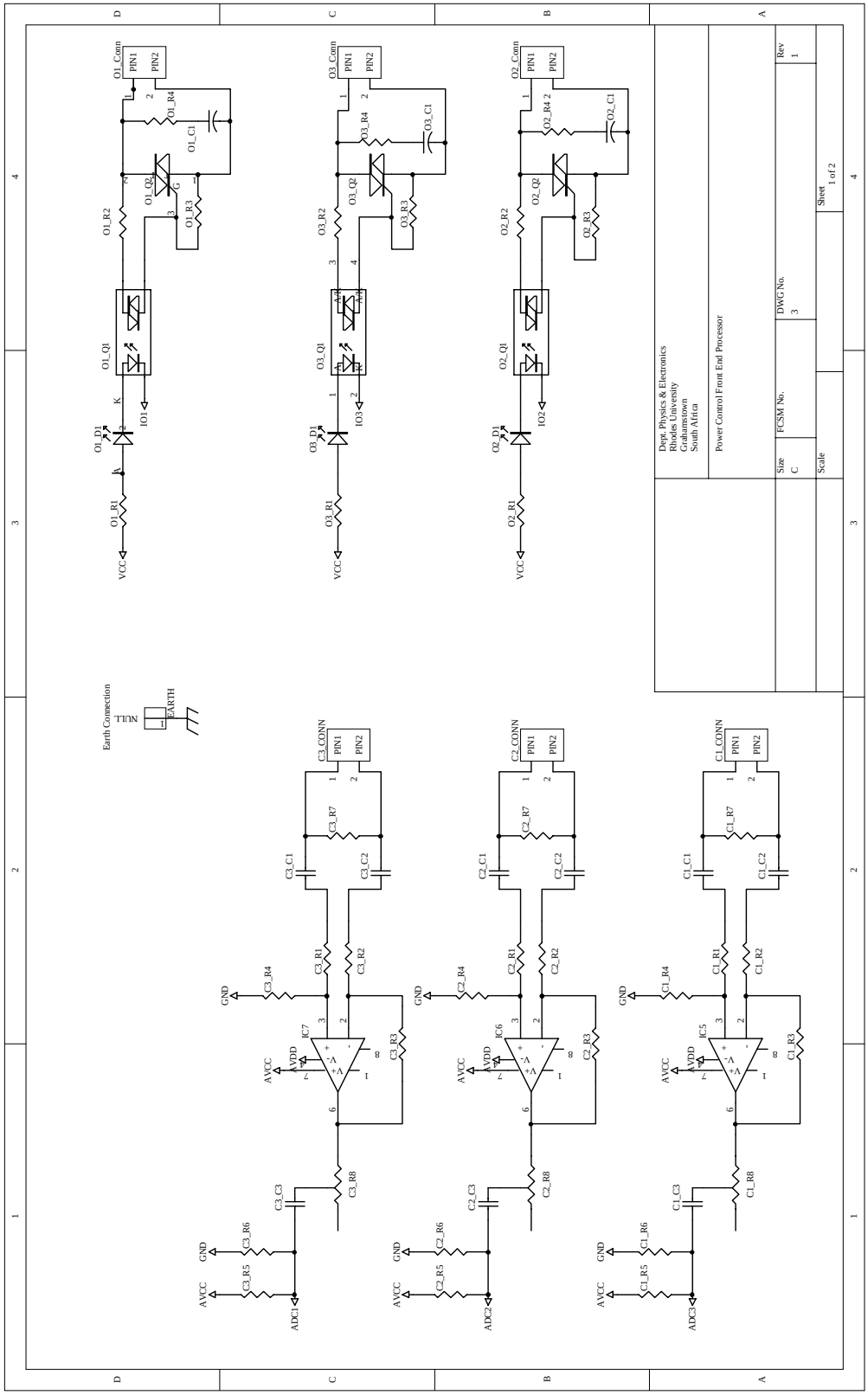


10.1.7 Appendix A7 - Field Controller PCB Foils (Top Overlay)

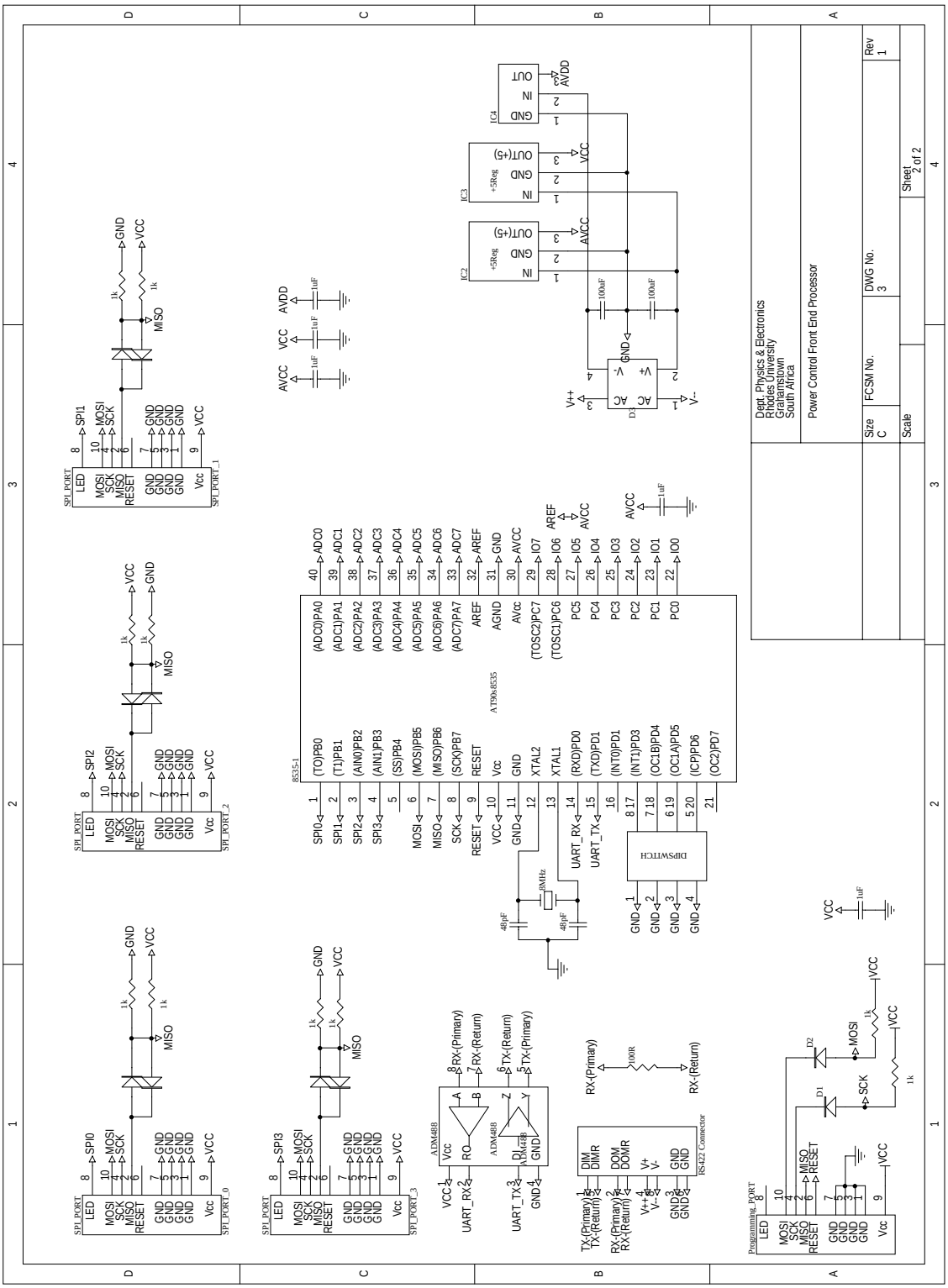


10.2 Appendix B - Front End Processor

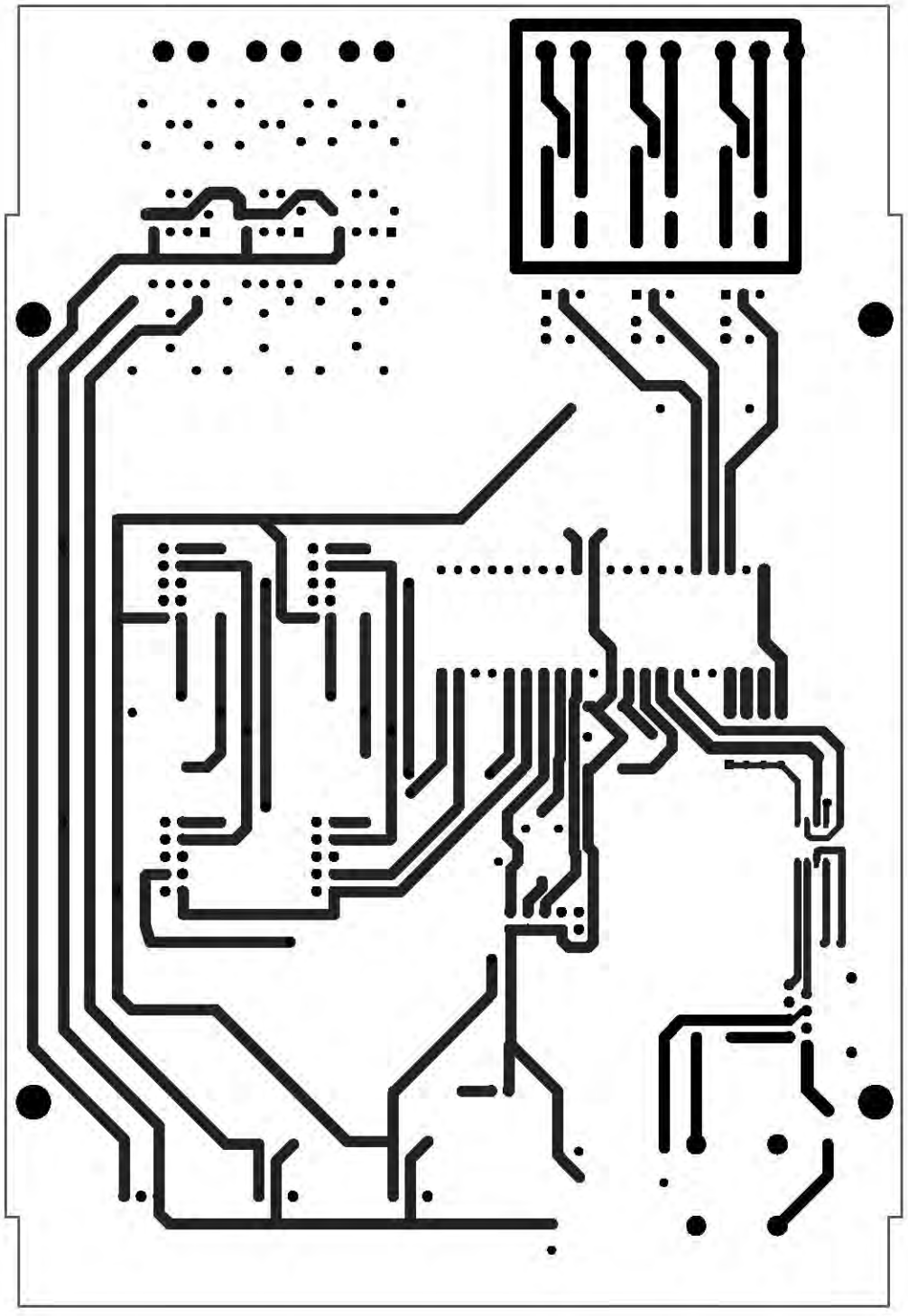
10.2.1 Appendix B1 - FEP Schematic (1/2)



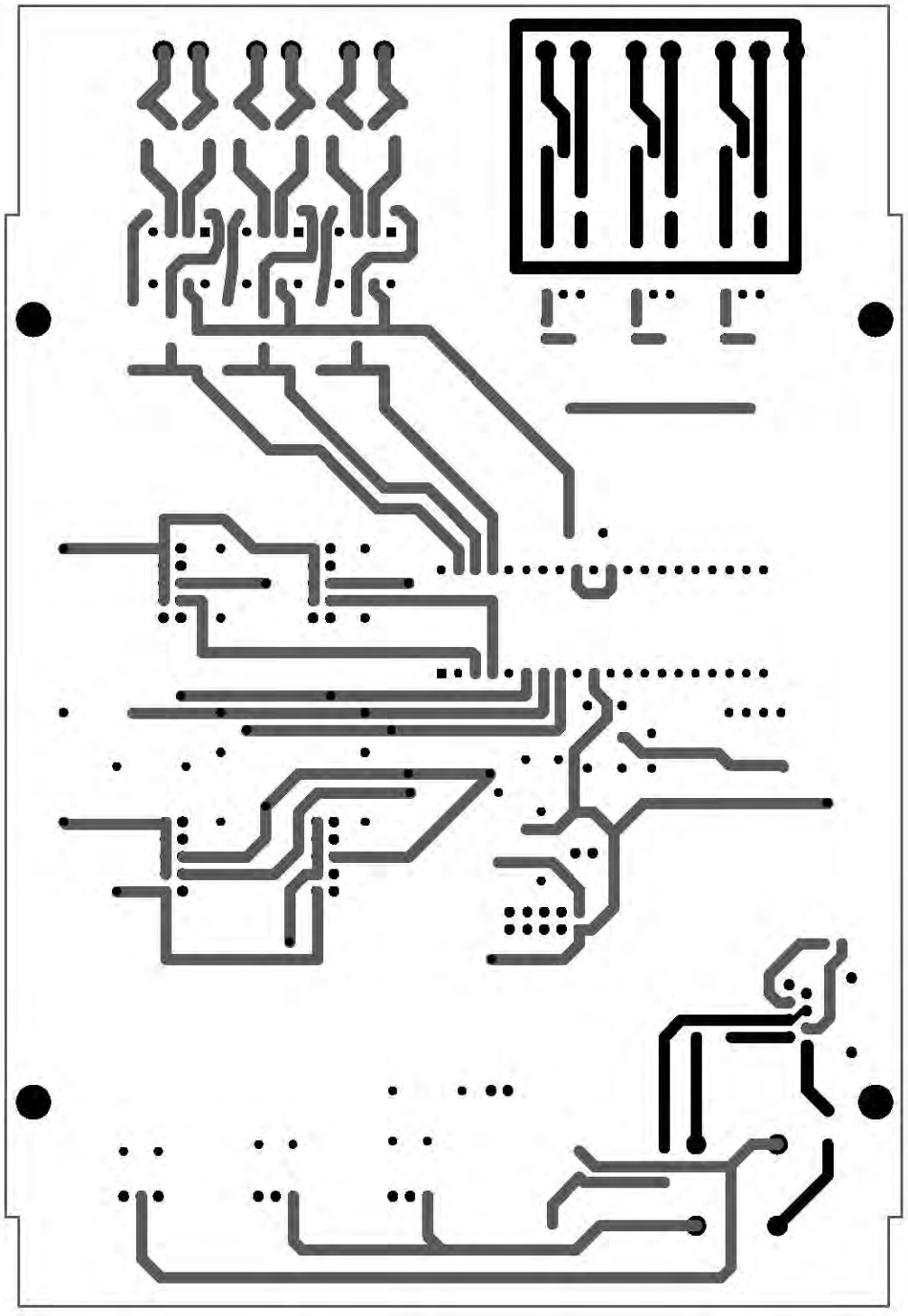
10.2.2 Appendix B2 - FEP Schematic (2/2)



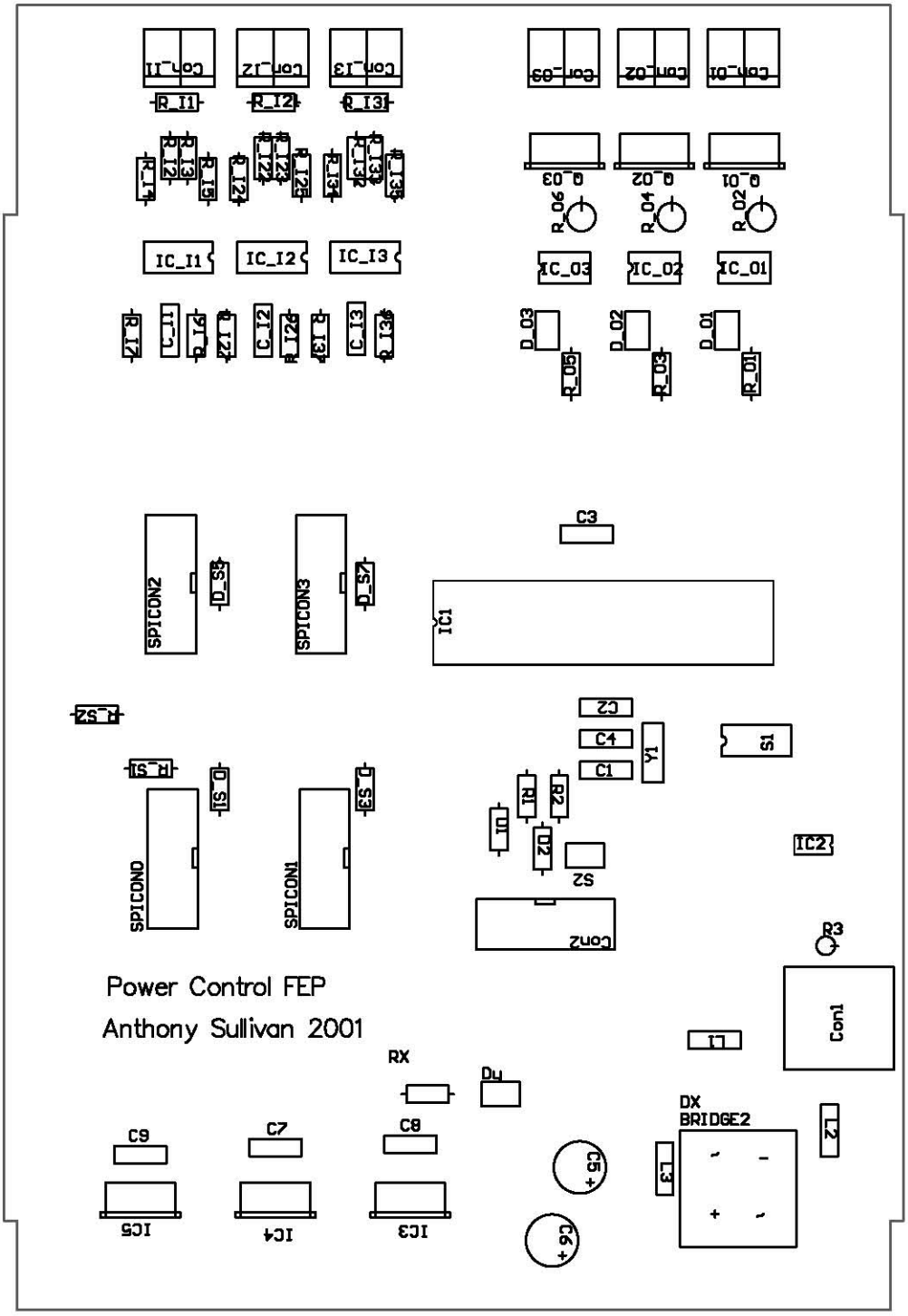
10.2.3 Appendix B3 - FEP PCB Foils (Top Layer)



10.2.4 Appendix B4 - FEP PCB Foils (Bottom Layer)



10.2.5 Appendix B5 - FEP FCB Foils (Top Overlay)



10.3 Appendix C - Master Controller

10.3.1 Appendix C1 - Sample Configuration Files

Sample *control.cfg*

```
#Power Control Configuration file -
see documentation for details
[RULE1]
SENSOR=cancer.phys.ru.ac.za,5,2,volts;
UPPERTHRESH=600;
LOWERTHRESH=300;
OUTPUTS=*cancer.phys.ru.ac.za,5,4;
        *cancer.phys.ru.ac.za,5,5;
        *cancer.phys.ru.ac.za,5,6;
%.
```

Sample *system.cfg*

```
#Power Control Configuration File -
see documentation for details
#
[device1.bootp.ru.ac.za]
Name:Desktop Test Device
Location:Room48
*5:Test FEP1; Room48
%
#
#
[cancer.phys.ru.ac.za]
Name:Test device 2
Location:Physics Department
*1:Monitor1 ; Physics Stairwell
*5:Desktop1 ; Room48
*6:Desktop2 ; Room48
%
```

10.3.2 Appendix C2 - Initial CGI Interface Perlscript

```
#!/usr/bin/perl  -wT
# $request_value=$ENV{QUERY_STRING};
$FIFO_OUT=CGI2CTL;
$FIFO_IN=CTL2CGI;
unless (-p $FIFO_OUT) {
      unlink $FIFO_OUT;
      system('mknod',$FIFO_OUT, 'p')
                &&die "can't mknod $FIFO_OUT: $!";
 }
open (FIFO_OUT, "> $FIFO_OUT") or die "can't write $FIFO_OUT: $!";
print FIFO_OUT $ENV{QUERY_STRING}; #$request_value;
close FIFO_OUT;
 unless (-p $FIFO_IN) {
      unlink $FIFO_IN;
      system('mknod',$FIFO_IN, 'p')
                &&die "can't mknod $FIFO_IN: $!";
 }
open (FIFO_IN, $FIFO_IN) or die "can't write $FIFO_IN: $!";
read FIFO_IN, $buf, 500000;
close FIFO_IN;
print <<ENDOFHTML;
Content-type: text/html
Expires: 0
<HTML>
<HEAD>
<TITLE>Power Control System - Test site</TITLE>
</HEAD>
$buf
</HTML> ENDOFHTML
```

10.3.3 Appendix C3 - Extract from Bootp Server Configuration

Relevant extracts from /etc/dhcpd.conf on the various dhcp servers:

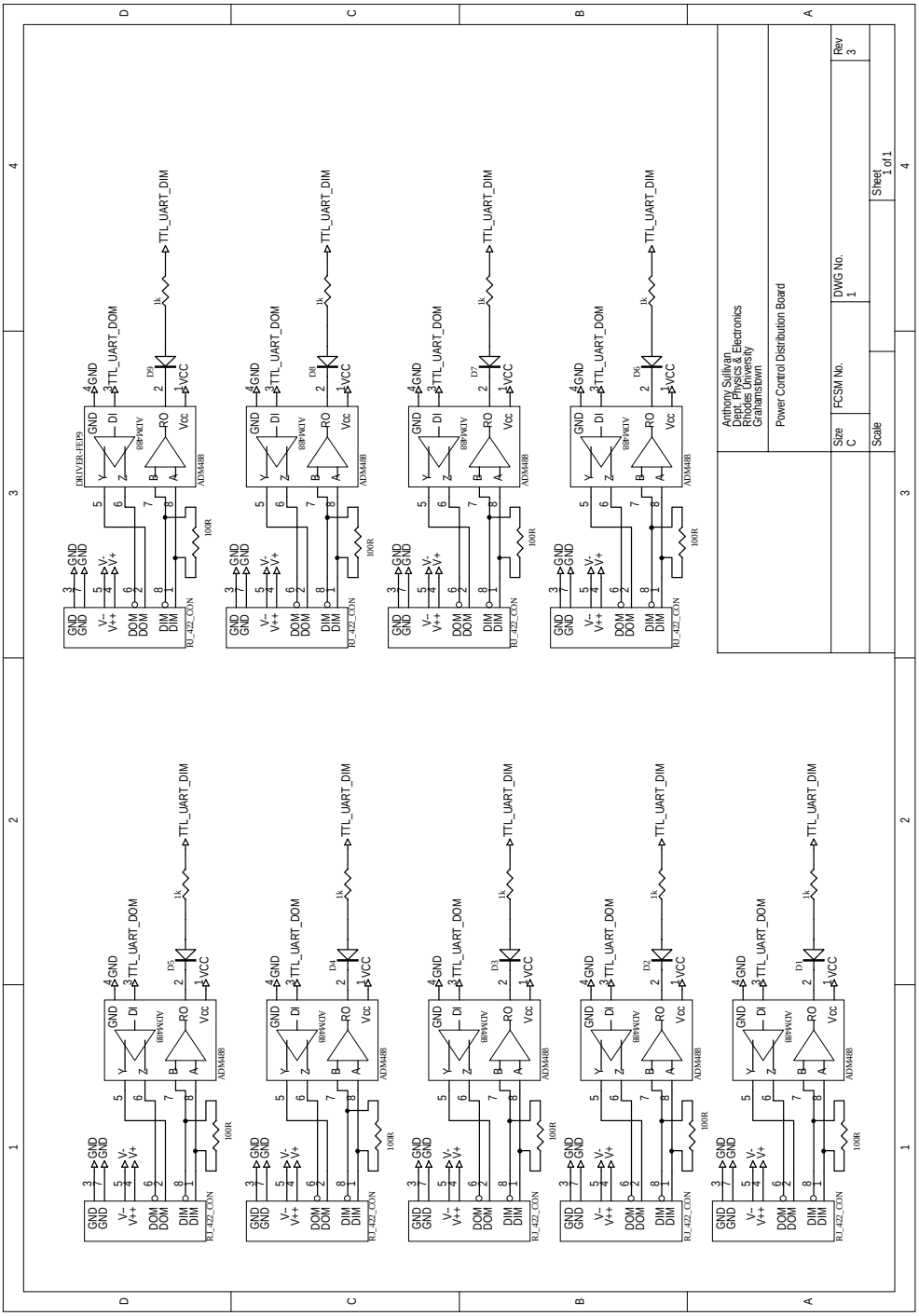
```
option  domain-name-
servers     146.231.128.6, 146.231.128.1;
option  ntp-
servers             146.231.128.6, 146.231.128.8;
option  ip-forwarding           off;
default-lease-time              28800;
max-lease-time                  172800;
option  time-offset             7200;
option netbios-node-type        8;
option  netbios-name-servers    146.231.128.6;
option netbios-dd-server        146.231.128.6;
option  ieee802-3-encapsulation off;
authoritative;
...
shared-network RHODES-geog {
  subnet  146.231.80.0  netmask      255.255.248.0 {
         option  routers             146.231.80.1;
         option  broadcast-address   146.231.87.255;
        }
  }
 ...
group {
...
    host    device1 {
    hardware        ethernet        00:00:00:ab:cd:ef;
    fixed-address                   146.231.84.153;
    option  domain-name             "bootp.ru.ac.za";
    filename        "/pcondir/ver1/device1.img";
    next-server     146.231.84.151;
     }
...
}
```

10.3.4 Appendix C4 - RRD creation script

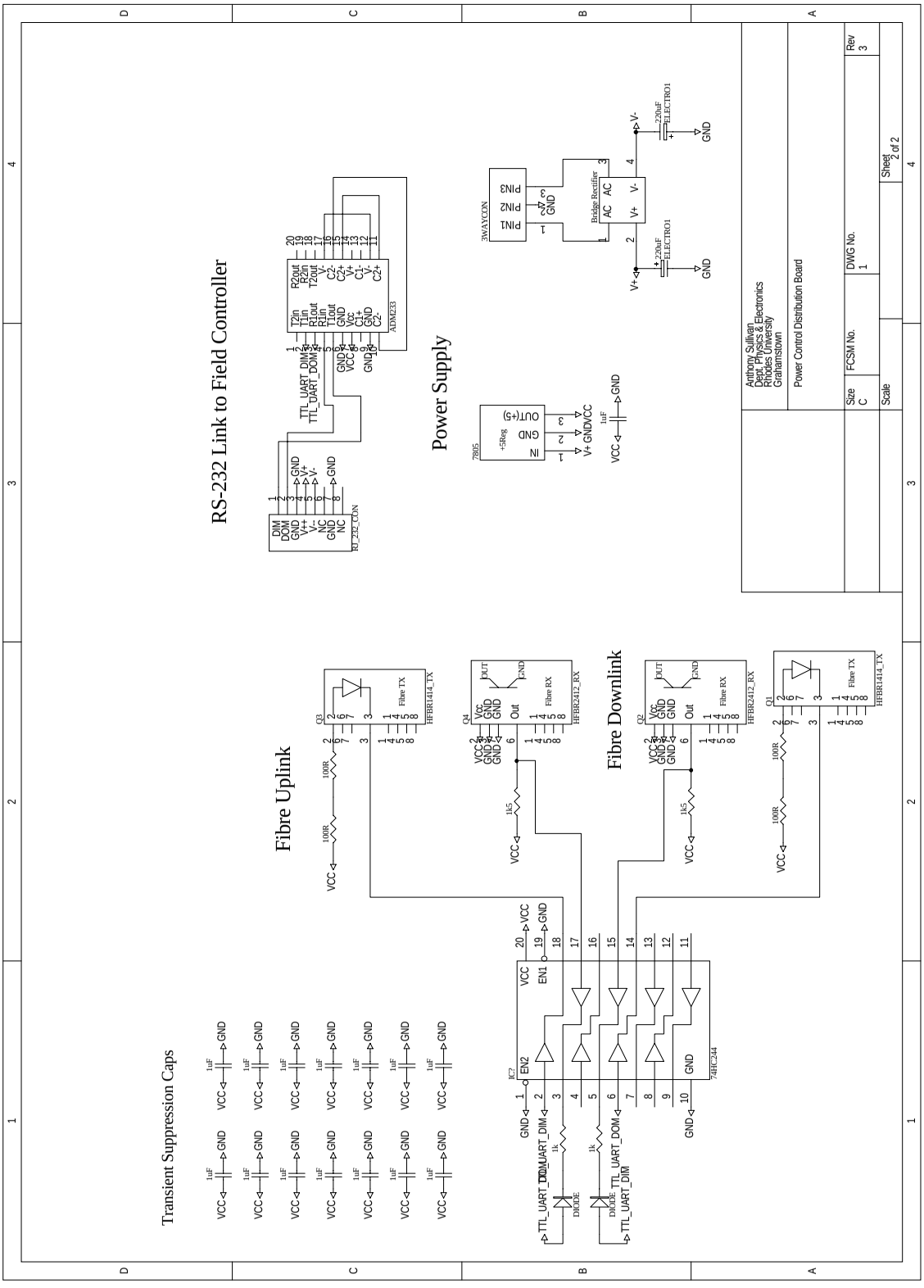
```
#!/usr/bin/perl -w
    #This script is used to create the rrd databases with the required properties
    #the first argument is the Field Controller IP name, the second is the FEP id
    ($FCName, $FEPid)=@ARGV;
    $systime=time();
    $rrdname= sprintf("%s_%s.rrd",$FCName,$FEPid);
    use RRDs;
    RRDs::create "$rrdname", "–start", "$systime", "–step", "60",
            "DS:chan1:GAUGE:90:0:U", "DS:chan2:GAUGE:90:0:U", "DS:chan3:GAUGE:90:0:U",
            "RRA:AVERAGE:0.5:1:1440", "RRA:AVERAGE:0.5:60:168", "RRA:AVERAGE:0.5:1440:370",
              "RRA:MIN:0.5:1:1440", "RRA:MIN:0.5:60:168", "RRA:MIN:0.5:1440:370",
              "RRA:MAX:0.5:1:1440", "RRA:MAX:0.5:60:168", "RRA:MAX:0.5:1440:370";
     my $ERR=RRDs::error;
     die "ERROR while creating .rrd: $ERR\n" if $ERR;
    printf ("$rrdname\n");
```

10.4 Appendix D - Distribution Board

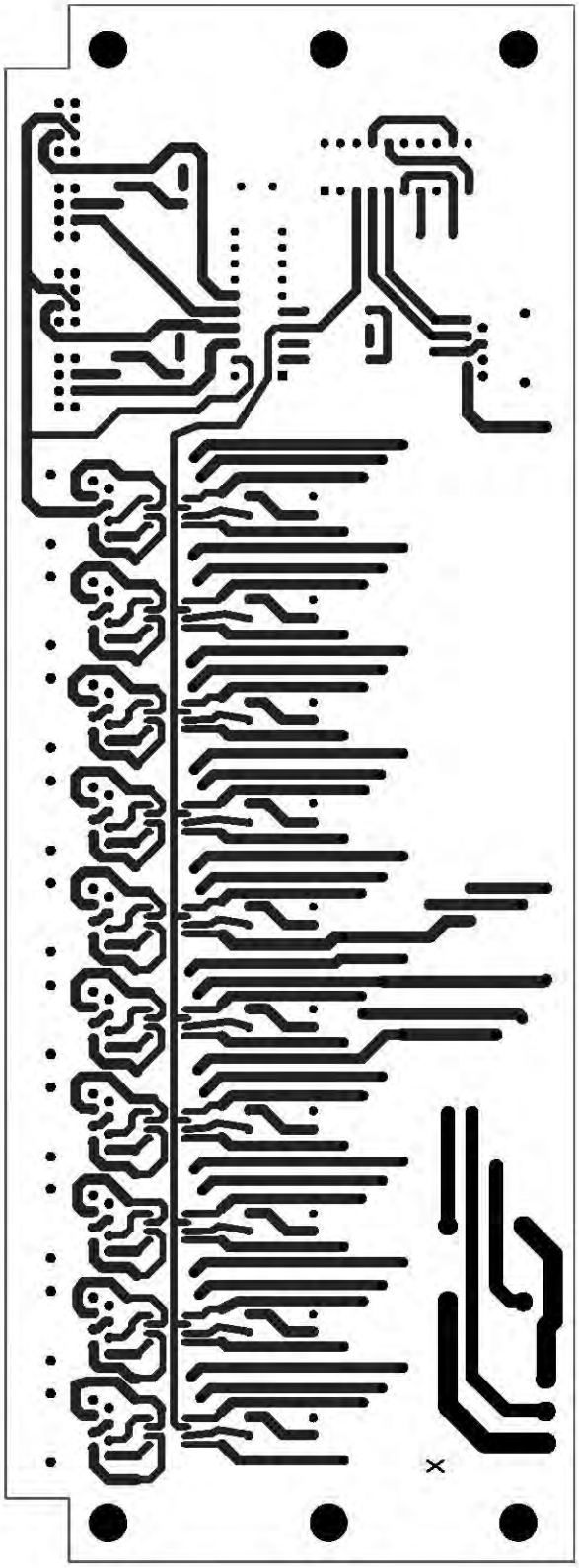
10.4.1 Appendix D1 - Distribution Board Schematic (1/2)



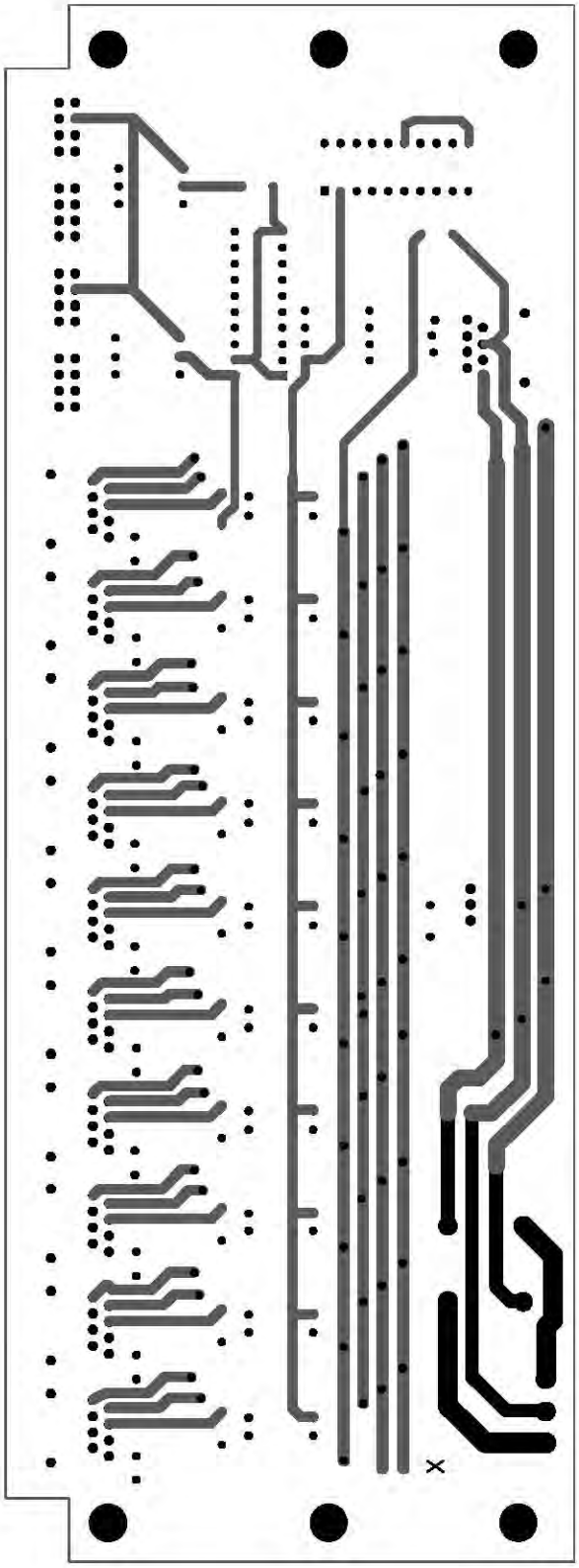
10.4.2 Appendix D2 - Distribution Board Schematic (2/2)



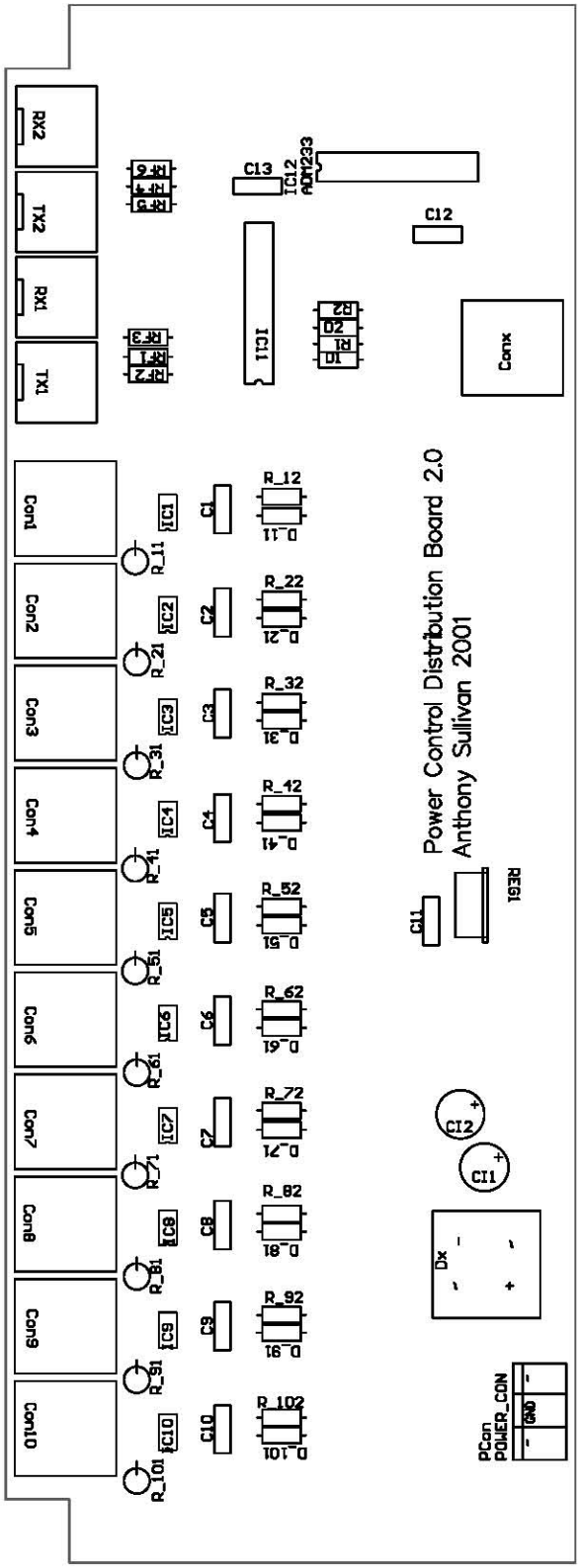
10.4.3 Appendix D3 - Distribution Board PCB Foils (Top Layer)



10.4.4 Appendix D4 - Distribution Board PCB Foils (Bottom Layer)

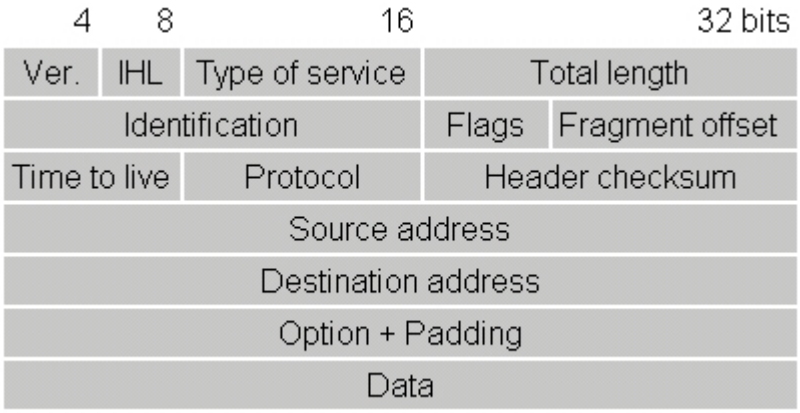


10.4.5 Appendix D5 - Distribution Board PCB Foils (Top Overlay)



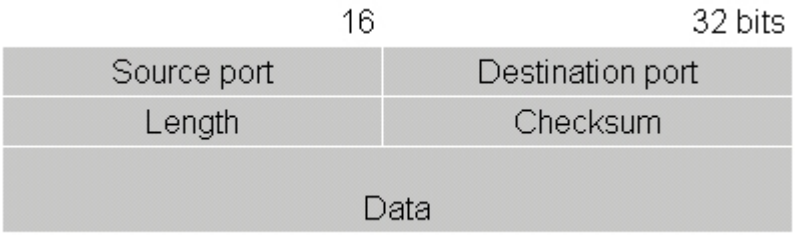
10.5 Appendix E - Network Packet Structures

10.5.1 Appendix E1 - IP Packet Structure



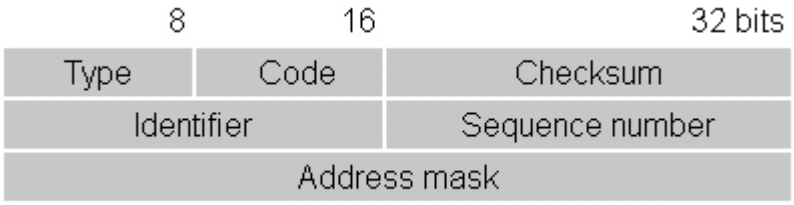
- Ver: The version field indicates the format of the Internet header
- IHL: Internet header length is the length of the IP header in 32-bit words. It also serves as a pointer to the beginning of the data.
- Type of Service: Indicates the type of service regarding delay, reliability and throughput.
- Total Length: This is the total length of the datagram (header and data) in bytes.
- Identification: Identifying value assigned by the sender to aid in assembling datagram fragments.
- Flags: Fragmentation settings.
- Fragment Offset: This is the offset given in 8-bytes units of the current fragment.
- Time to live: The number of hops that a datagram has left to live.
- Protocol: Indicates the next level protocol contained.
- Header Checksum: The checksum performed on the header only, this is calculated at each hop.
- Source / Destination Address: The 32-bit IP address.
- Options and Padding: This specifies the extended options that are available such as source-routing.
- Data: The data payload of the IP datagram.

10.5.2 Appendix E2 - UDP Packet Structure



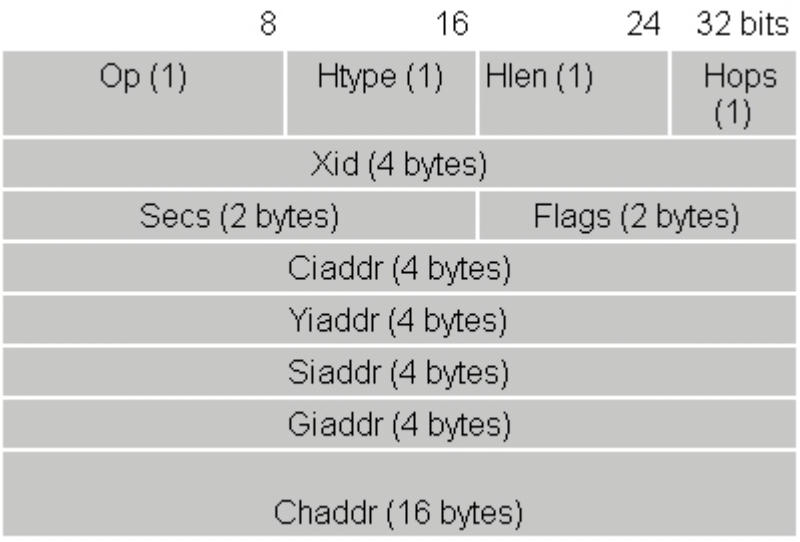
- Source Port: This is an optional field and indicates the port of the process sending the UDP datagram and is generally assumed to be the port to which to reply to.
- Destination Port: This is the port to which the datagram will be delivered.
- Length: The length of the data payload in bytes (minimum of 8).
- Checksum: This is the 16-bit one's complement of a pseudo-header made up from sections of the IP header as well as the UDP header and data.

10.5.3 Appendix E3 - ICMP Packet Structure



- Type & Code: These two fields determine the type of ICMP message that is contained in the datagram.
- Checksum: The 16-bit one's complement of the one's complement of the message.
- Identifier: This is an identifier that is used to aid in matching ICMP requests and replies.
- Sequence Number: This sequencing number is also used to aid matching requests and replies.

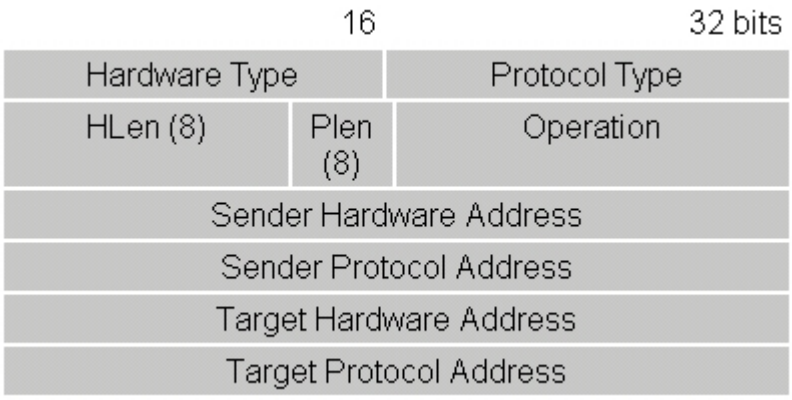
10.5.4 Appendix E4 - BOOTP Packet Structure



- Op: The operation type, either BOOTREQUEST or BOOTREPLY.
- Htype: The hardware address type.
- HLen: The hardware address length.
- Xid: The transaction ID.
- Secs: The number of seconds that have elapsed since the client began the address acquisition or renewal process.
- Flags: The flags indicating optional data.
- Ciaddr: The client’s IP address.
- Yiaddr: The client’s IP address.
- Siaddr: The IP address of the next server to use in the boot process.
- Giaddr: The IP address of the client’s gateway.
- Chaddr: The client’s hardware address.

Specifics can be found in RFCs 951, 1531, 2131, 2132.

10.5.5 Appendix E5 - ARP Packet Structure



- Hardware Type: Specifies a hardware interface type for which the sender requires a response.
- Protocol Type: Specifies the type of higher level protocol address that the sender has specified.
- HLen: The Hardware address length.
- PLen: The protocol address length.
- Operation: Specifies the operation (ARP request/response, RARP request/response)

The specifics can be found in RFC 826 (with updates in RFC 1293 & RFC 1390)

10.6 Appendix F - General

10.6.1 RJ-45 Connection Information

The RJ45 sockets used all had the tab on the PCB side of the socket. Sockets with differing footprints and pinouts are not recommended without carefull consideration.

The following tables refer to the diagram below:

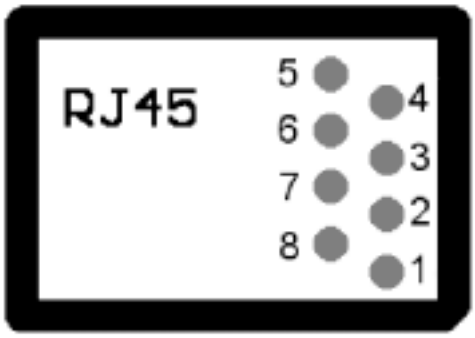


Figure 10.1: RJ-45 Pinout

Pin #	DB Pinout	FEP Pinout	Function
1	RX	TX	Data into Master
2	TX	RX	Data from Master
3	Gnd	Gnd	
4	V++	V++	
5	V–	V–	
6	TX (ret)	RX (ret)	Data from Master
7	Gnd	Gnd	
8	RX (ret)	TX (ret)	Data into Master

Pin #	DB Pinout	FC Pinout	Function
1	TX	RX	Data into Master
2	RX	TX	Data from Master
3	Gnd	Gnd	
4	V++	V++	
5	V–	V–	
6	NC	NC	
7	Gnd	Gnd	
8	NC	NC	

10.6.2 CD-ROM Guide

Path	Filename	Description
/Datasheets	AT90S8535.pdf	Datasheets for the Atmel AT90S8535 used for the FEP
	8900a.pdf	Datasheets for the Cirrus Logic Crystal 8900A Ethernet controller
	AVR Keyfob.pdf	Datasheets for the Keyfob programmer for programming AVR microcontrollers in the field.
	Cs5451-4.pdf	ADC from Cirrus Logic suited for Current and Voltage measurements
	Cs5460-6.pdf	Power Metering IC from Cirrus Logic
	AN181.pdf	Application note for using the CS8900A in 8-bit mode
	er271a1.pdf	Errata for the CS8900A Datasheets
	Mega.pdf	Datasheets for the ATMega103 that was used in the Field Controller
	MOC3162M.pdf	Fairchild semiconductor datasheet for opto-isolated triac
	Mega161.pdf	The proposed replacement MCU for the AT90S8535 in the FEP
/Fibre	*	Various Data sheets and application notes for the Agilent Fibre Transceivers used.
/RFCs	*	Text versions of the RFCs referred to in the thesis
/designs	net board.ddb	Protel 99 Design Database for the Field Controller
	PCON_SE.ddb	Protel SE Design Database for the FEP and Distribution board
/code	Ether_dev.zip	The zip archive of the Field Controller Code
	FEP.zip	The zip archive of the FEP code
	CGI.tar	The tar file containing the source and executable of the WWW CGI interface
	control_prog.tar	The tar file containing the source and executable of the control program
	scripts.tar	The tar file containing all the scripts used.
	config.tar	The tar file containing all three sample configuration files.
	Readme.txt	Installation instructions
/Thesis	*	The electronic version of this thesis.