Integrated Sensor and Controller Framework

A thesis presented in partial fulfilment of the requirements for the degree of

Master of Engineering

In

Information and Telecommunications Engineering

At Massey University, Palmerston North, New Zealand

Ryan David Weir

2007

Abstract

This thesis presents a software platform to integrate sensors, controllers, actuators and instrumentation within a common framework. This provides a flexible, reusable, reconfigurable and scalable system for designers to use as a base for any sensing and control platform.

The purpose of the framework is to decrease system development time, and allow more time to be spent on designing the control algorithms, rather than implementing the system.

The architecture is generic, and finds application in many areas such as home, office and factory automation, process and environmental monitoring, surveillance and robotics.

The framework uses a data driven design, which separates the data storage areas (dataslots) from the components of the framework that process the data (processors). By separating all the components of the framework in this way, it allows a flexible configuration. When a processor places data into a dataslot, the dataslot queues all the processors that use that data to run.

A system that is based on this framework is configured by a text file. All the components are defined in the file, with the interactions between them. The system can be thought of as multiple boxes, with the text file defining how these boxes are connected together. This allows rapid configuration of the system, as separate text files can be maintained for different configurations. A text file is used for the configuration instead of a graphical environment to simplify the development process, and to reduce development time.

One potential limitation of the approach of separating the computational components is an increased overhead or latency. It is acknowledged that this is an important consideration in many control applications, so the framework is designed to minimise

the latency through implementation of prioritized queues and multitasking. This prevents one slow component from degrading the performance of the rest of the system.

The operation of the framework is demonstrated through a range of different applications. These show some of the key features including: acquiring data, handling multiple dataslots that a processor reads from or writes to, controlling actuators, how the virtual instrumentation works, network communications, where controllers fit into the framework, data logging, image and video dataslots, timers and dynamically linked libraries. A number of experiments show the framework under real conditions. The framework's data passing mechanisms are demonstrated, a simple control and data logging application is shown and an image processing application is shown to demonstrate the system under load. The latency of the framework is also determined. These illustrate how the framework would operate under different hardware and software applications. Work can still be done on the framework, as extra features can be added to improve the usability.

Overall, this thesis presents a flexible system to integrate sensors, actuators, instrumentation and controllers that can be utilised in a wide range of applications.

Acknowledgements

I would like to thank my supervisors – Gourab Sen Gupta and Donald Bailey. They have put a lot of effort in directing my thesis to where it is now.

I would also like to thank friends and family for supporting me or providing assistance – especially the other postgraduate students. Your help and support is appreciated.

List of Figures

Key for Figures

Below in Figure 1 is a key for most of the figures in this thesis.

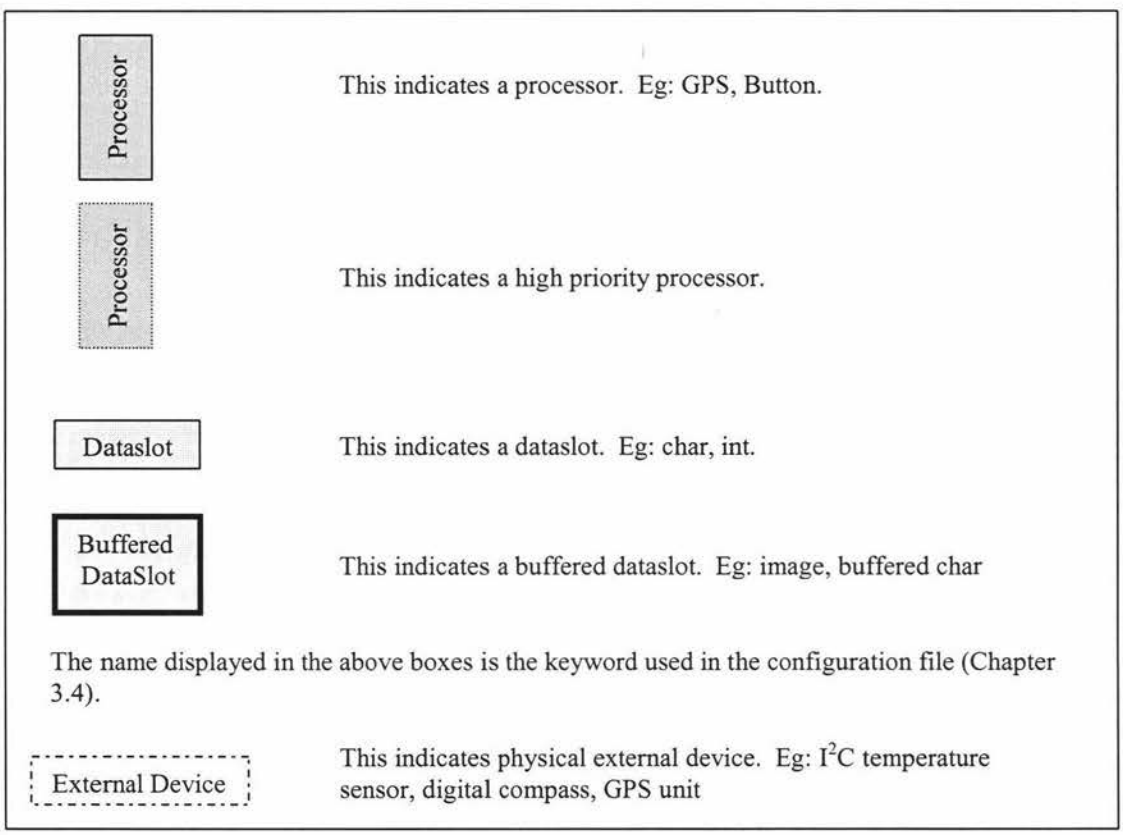


Figure 1: Key for most figures in this thesis.

List of Tables

Table of Contents

1 Introduction

There is a wide range of applications where a control system is needed with sensors and actuators. Most of the control systems currently implemented are specific for the task for which they were designed. A generic sensing and control system would save development time. This thesis explores the possibility of such a system.

The first part of this chapter discusses various sensing and control applications. This reviews the type of applications that a generic sensing and control system could be used in. The requirements of a general purpose system are analysed, and previous research in this and similar fields is then reviewed. Finally, an outline of the remainder of the thesis is presented.

1.1 Sensing and Control Applications

A wide range of applications require sensing and control for the system to function. To be truly generic, a generic sensing and control system would need to be able to be readily adapted to each of them. Five examples of these application areas are:

- Home automation
- Environmental monitoring
- Industrial control
- Remote vehicle control
- Security systems

Home automation is the automation of a private home to provide increased comfort and security for its occupants (Ryan 1989; Wacks 2002). This usually requires a number of sensors and control devices distributed throughout the house. There needs to be a way to allow human interaction with the control systems, for example for an occupant to set their preferences or desires. This interaction could be via one or more remote controls, a PC program or a touch screen. Home automation covers a wide variety of tasks. For example, sensors could monitor the temperature inside the house. When the temperature drops below the occupier's preferences, heaters can be turned on. Appliances could be turned on from the office, so when the occupant comes home the

oven is already cooking dinner. Lights could be switched on and off depending on where people are inside the house. Security systems could monitor the house while the occupants are sleeping or are at work. Input to a security system would consist of cameras, motion sensors and door and window sensors. When an intruder is detected, an alert could be sent to the occupants and to a monitoring agency.

(Choi et al. 2005) presents a home automation control system, that provides a automatic home service based on the users' preference. The system comprises of an appliance controller (provides communication between appliances), a context aware agent (the descision engine) and a browser (provides user input and displays information). Every person's preferences are stored, as it is a context system – every person has their own preferences for a comfortable environment. It uses six data sets derived from sensors (pulse, body temperature, facial expression, room temperature, time and location) to feed into its momentum back propagation neutral network (Zhifei et al. 2001) to learn and predict the user's preferences. Conflicting preferences between different users are not discussed. A PDA (Personal Digital Assistant) carried by the user acquires the pulse, body temperature and facial expression. A person is located in a room by four cameras. This system operates with no human intervention. Choi's paper presented a system that integrates sensors and actuators with a neutral network controller, but does not discuss virtual instrumentation. Each user needs to carry a PDA, which limits its usefulness. Considering that each room needs four cameras to locate a person, the number of hardware devices and processing needed just to locate everyone in a house is significant.

(Liang et al. 2002) discusses a home automation control architecture. It is a multi-agent based architecture that has five components. The space agents component deals with connecting devices to the system. Function agents interpret low level device data (e.g. temperature) into high level meaningful data describing the home environment (e.g. the temperature in the room is above the minimum and below the maximum limits). They then interpret a high level command, such as saving power. The result would be to turn some devices off. Personal preference agents store the preferences of each person. For example each person would have their own set levels of turning heaters on and off. The

environment variable server stores the data in the system. It allows other components of the system to access its data. By adding permissions, Internet access to this data can be granted. The environment variable server can update some variables automatically from the Internet. Each of these components runs on a separate computer. By separating the components of the system out, the architecture is made more flexible. There is a questionable need for a multi-agent system in a home automation environment, given that ideally the system should use as little power, and take as little room as possible.

Environmental monitoring is another area where a flexible sensing and controller system could be used. Weather monitoring is very important for farmers. Real time monitoring of water levels (Carpenter et al. 1991), allows farmers to move stock to higher ground before they become isolated by rising river levels. Water for a farm is taken from a dam or a river, which can be monitored. This water needs to be pumped to all of the troughs. By having water level sensors in each trough, the pump can be turned on when the water level drops too low. To stop some troughs overflowing while other toughs are still filling up, a simple ballcock value can be used at each trough. When the last trough is full, the pump can turn off again. Data logging is needed as the water level of the water supply to a farm needs to be monitored over a long period. This would indicate whether the farmer would need to buy or acquire more water over the summer months. In potential fish farms (Froysa 2004), oxygen levels, sediment condition, water quality and current conditions are monitored to give a good indication whether it would be beneficial to establish a fish farm at a particular location. Temperature monitoring would allow grape growers to know if their crops were likely to be affected by a frost, thus they then can take action to reduce the effects of it. As most farmers are not going to be in front of a screen all day, alerts can be sent to their mobile phone or a buzzer to wake them during the night. Having a network of video cameras would allow farmers to check on stock, crops and the general condition of the land, without travelling to remote areas of the farm. This would be especially useful during critical weather conditions. The farmer could prioritise areas of the farm according to severity.

(Anumalla et al. 2005) describes a system that monitors ground water with smart sensors. Each sensor is connected to an FPGA (Field Programmable Gate Array) which transmits its real time data via an 802.11 wireless link. The sensor, FPGA and antenna together are called a DAU (Data Acquisition Unit). Multiple DAUs are connected together in a local wireless network. At least one DAU is connected to the base station. This station collects all the data and allows access to this data via the Internet. The end user can access and process this data. Anumalla's paper focussed solely on gathering information from sensors, but no actuators or controllers are defined.

Industrial control is another application area for a generic sensing and control system. Industrial control includes shop floor automation, inventory tracking and control, fleet management, process control and monitoring, autonomous robotics and machine vision. Shop floor automation (e.g. Smith et al. 1994; Junior and Pereira 2003) involves the automation of machinery within a factory. This machinery needs to be monitored and an alert sent when maintenance is needed or when a variable has exceeded it preset limits. Inventory tracking (e.g. Cope et al. 1997) and control involves knowing the location of every item in stock and having control over what happens to that item. In a warehouse this means storing the location of each package in a database. If the package needs to be retrieved, its exact location and destination can be determined quickly. These packages could even be placed into the warehouse in a manner optimised for efficient removal at a later date. Fleet management (e.g. Silva and Mateus 2003; Giralda et al. 2005) involves tracking the vehicles belonging to a company. This is useful as knowing where each vehicle is, the company can better plan which goods can be taken from one location to another, provide updates to customers (e.g. estimated delivery times or passenger bus arrival times), monitor the speed of the vehicles (to keep costs down from tickets) and monitoring when the driver is having breaks (or conversely, making sure the driver is not too tired to drive). Process control (e.g. Sullivan et al. 1994; Venkatesan and Abachi 2002) and monitoring involves checking to make sure that each process associated with a task is operating correctly. This could be ensuring that the proportions of chemicals going into the same mix are right, to displaying the current temperature of a wave soldering machine and the speed of the PCBs (Printed Circuit Board) travelling through it. Autonomous robots (e.g. Oriolo et

al. 1998; Sen Gupta et al. 2002) running a generic system could take the data from various sensors to find its way around in the world. These sensors could be cameras, ultrasonic sensors, lasers and bump switches. A decision would be made from the onboard controller, and it would signal to the actuators that it should move in a certain direction to carry out a particular task.

Machine vision is the application of cameras, to allow viewing and image processing on the frames received. Machine vision can be used for grading (e.g. Bailey et al. 2004), quality control, inspection (e.g. Zhenzhong et al. 2001; Bulanon et al. 2004) and automation (e.g. Pomerleau and Jochem 1996). It works well for inspection of manufactured goods, as it can run at high speed, 24 hours a day with accurate results. Machine vision can be also be used in hazardous environments, where it is too dangerous to for humans to operate. As machine vision is very different to human vision, the lightning conditions need to be strictly controlled.

Remote vehicle control is the direct control of a vehicle from another location. This vehicle could have sensors to locate itself in the environment, such as ultrasound, GPS devices, cameras and other sensors used during the operation of the task that it was designed for. Remote vehicles can be used for entering hazardous environments (Savall et al. 1999), search and rescue (Murphy 2000), surveillance (Saptharishi et al. 2002), remote monitoring and bomb disposal. For hazardous environments there could also be radiation detectors. Search and rescue after an earthquake, for example, may require manoeuvring through tight spaces to locate survivors so rescue efforts can be directed to those locations. For surveillance operations, cameras and long distance microphones can be used. Telerobotics is the control of remote robots (Graves and Czarnecki 1999). Considerations for telerobotics include latency (a long delay may result in the robot moving into a undesirable location due to not getting directions in time) and fault tolerance in the communications link. This may require error detection and correction, retransmission of data, to reduction of transmitted data in video: reduction of frames, dimensions, or reduction of redundant data (Lu et al. 1996).

Using the framework for security systems would require positioning sensors to monitor the secured area. This could include cameras, movement detectors, and door, window and heat sensors. The data gathered would need to be displayed and recorded, with the possible result of any detected intrusion being a call to the police, activating sirens and sending SMS messages to a cell phone to inform the owner of the affected property.

1.2 Framework Requirements

As no specific system would work in all the above applications, a generic system is appropriate. A generic system uses the same core for all applications, but extra code is placed around the core to adapt it to the particular sensing and control application. The core therefore forms the framework around which the specific system is built.

A sensing and control framework would need five components. Sensors are required to gather real world information. Controllers are used to make a decision based on the data coming in from the sensors. Then there are actuators which implement the decisions from the controllers. Some instrumentation may be required to indicate to users what is happening with the system. Most systems also require controls for user input. These can be either physical controls, or virtual (buttons on a screen).

The framework must be flexible in the type of applications that it can handle. It should be able to control a game of robot soccer, run on an aerial platform, catalogue and track packages in a warehouse, or implement a home automation system.

This framework should be reusable. As the framework is the core of the system, it should only be designed once, and then each application can use it to implement the desired system. Adding the necessary code to implement each specific application should take a minimum of effort. If all the code needed by the components for a particular application is already in the framework, then a simple configuration is all that is required. This configuration should not require detailed programming knowledge. However, if a new sensor or actuator is added, extra code to handle this device will need to be programmed. The framework should be structured to make this process as easy as possible and allow the new modules to be used in other applications where appropriate.

The framework should be reconfigurable. It should take minimum effort to set up the configuration for a particular application. A consequence of using a generic system is that the processing time may be longer than that for program designed specifically for each application. However, this will be offset by having a reconfigurable system. It is not intended that the system be self-reconfigurable. Self reconfigurable systems make assumptions about what needs to be done with the data received, so the system needs to be designed in this way. To limit the scope of this thesis, self reconfigurable systems will not be investigated.

The framework should be scalable – both in the design and computational requirements. Having 20 sensors or actuators should work just as well as having only a few. The computational requirements of the system should scale in proportion to the number of sensors and actuators there are. By building networking into the framework, a distributed system may be created where computation is spread over many physical processors. This has the advantage of reducing the load on the system collecting the data, allowing for more sensors and actuators to be connected if needed. The disadvantage with a program hard coded for each application is that expandability is often not considered at design time. As such, adding new devices can have an exponential increase in the amount of time taken to process the new data. This is due to an inefficient design, which can be manifested as a significant number of links between the system's components. The resulting dependencies between all of the modules of the system also make it more difficult to maintain. The proposed framework avoids this by planning the interactions between the components.

1.3 Previous Research

(Sohrabi et al. 2004) presents a method for a wireless network to organise itself, as a self-reconfigurable system. In a dynamic environment, sensors could be coming online and going offline at various times. There needs to be a way for a system to recognise new devices upon start up, and continuously organise the network upon addition or removal of sensors. This could be used in unattended ground-based systems, remote planetary exportation and condition-based maintenance. This is not a sensing and

controller system; however this proposal would be a valuable addition to the framework if wireless sensors were used as a part of the whole system.

A scaleable sensing and alert management system is presented in (Cohen et al. 2005). It tracks specific business metrics, and generates notifications or alerts based on predefined rules. It consists of an event stream processor which handles all tasks. These tasks include receiving data from various sources, conversion of the data to a common format, and calculating the metrics based on incoming events. One metric could be 'running average of vehicle speed'. This task covers checking if rules where followed during the operation of the system. Such rules as 'the twenty minute running average of vehicle speed should not exceed 75 kilometres per hour' are stored in XML format. Once the result of this task is known, the appropriate notification or alert can be given. Scalability is addressed by separating the parts of the system. A common data storage area allows access by multiple events, saving on storage area. Actuators are not a part of this system, limiting its usefulness in a sensing and controlling framework.

Lately much work has been done to create intelligent networked and wireless sensors. A system architecture for a wireless habitat monitoring has been proposed in (Mainwaring et al. 2002). It uses a hierarchical network with wired and wireless sensor nodes linked to a base station via a gateway. Data is logged at the base station. The gateway is connected to the Internet, allowing remote user access to the data produced by the sensors. This network is located on an island. Each node has multiple sensors on it: photoresistor, temperature, barometric pressure, humidity, passive infrared (thermopile) sensors. The nodes are difficult to access over many months, due to poor off season weather conditions. During this time, power for each node will be supplied by a pair of AA batteries. As such, these nodes are only turned on for 1.4 hours per day. This is enough to record measurements throughout the day. Transmission of each sensor node's data does not go directly to the gateway, instead it may have several jumps via other nodes first. This routing tree for each node is determined when the node is first switched on. Organisation of the transmission of network messages involves remotely configuring each node; however it takes a significant amount of power to achieve this. It takes one days worth of power to reprogram one node, which

has an estimated battery life of nine months. Considerable effort has gone into reducing the power requirements of the nodes. This architecture focuses on connecting all of the components of the network together. No actuators and user controls are included, thus it does not meet the requirements of the framework for this thesis.

(Estrin et al. 1999) presents a data centric wireless network. The object is to allow communication between a large number of wireless sensors. Traditionally there is a central node which gathers the data and provides computation required for the application. Due to the large number of sensors, this would not be practicable. Instead localized algorithms are used for communication. They reduce traffic by only interacting with sensors in the vicinity, but still collectively achieves the desired global objective. Each sensor adjusts its transmission power to a minimum value so that it maintains full network connectivity. This helps with power savings. A clustering algorithm is used to choose each sensors parent. This network achieves the frameworks objective of collecting data, but no option exists to control actuators.

A great deal has been done to improve sensors, making them more intelligent (Staroswiecki 2005, Tian 2001, Hossain 2002). Such intelligent sensors can sample analogue data, perform data calibration, retransmit data and can incorporate some decision making. Increasingly sophisticated controllers have been developed (e.g. Lasserre et al. 1998, Pal et al. 1983, Jump 1988, Body et al. 1997). These controllers use complex methods to provide a greater accuracy of the actuators they communicate with.

The IEEE-1451 standard defines a smart transducer interface to allow communication between transducers, microprocessors and networks (Lee and Schneeman 2000, IEEE 2000, IEEE 1998). The goal of the standard is to allow access to transducer data through a common set of interfaces, whether the transducers are connected by wired or wireless means to a system or network. The standard defines an area on a transducer that stores an electronic data sheet called TEDS (Transducer Electronic Data Sheet). The TEDS stores identity, calibration and correction data, measurement range and other device specific information. A NCAP (Network Capable Application Processor) is

located between the transducer module (which can include a TEDS) and the network. The job of a NCAP is to be an object oriented embodiment of a transducer. The NCAP defines a set of classes, methods, attributes and behaviours which describe the transducer that it connects to. This allows system developers to quickly integrate multiple transducers. This standard is still evolving, so the specifications are not finalized. Parts of an IEEE-1451 compliant network are covered in the standard, plus connecting these devices together. The result is a general purpose way of communicating with a transducer. Applications may be reused and migrated to other networks without major engineering effort. This IEEE-1451 standard does not describe any controllers, or create any framework for application design.

The research stated so far does not describe a complete sensing and control system, as only limited work has been done to combine sensors, controllers and actuators within a single flexible framework.

One of the first attempts was the CODGER system (Shafer et al. 1986). This was designed for a large remote vehicle. It consists of several modules running independently of each other. They are linked together by a database and a manager. The manager schedules when each module can run. First each module must meet the conditions to start running. These conditions are set by the currently executing module. Due to outdated software and hardware compared to their modern equivalents (object oriented programming and modern PCs), significant developments can be made with respect to this system.

A SPB (Scalable Processing Box) is presented in Hohmann et al. 2003. It uses a modular design to connect sensors, actuators and controllers together on a robotic platform. Embedded Linux is used to run the SPBs. Each SPB acts as a controller, with data storage areas called mailboxes. Multiple SPBs can be connected together to achieve a task. The first SPB would acquire data from a sensor, and do pre-processing. The location of the robot would be determined by the next SPB. The last SPB would work out where to move next, and send this to the actuators. All of this data would be stored in the mailboxes. This architecture enables a scalable sensing and control

platform to be constructed. The separation of the tasks allows easier reconfiguration. Each reconfiguration needs a recompile of the source code, which is not desirable if only a few variables need to be changed. Each mailbox may be accessed independently. By separating the mailboxes from the SPBs, the mailboxes are not tied to any operation. This would allow all data contention code to be placed in the mailboxes, instead of being spread out amongst the SPBs. Benefits would be gained by placing each processor in its own SPB. This way each processor can be connected to other processors without going through an intermediary step if it is not needed.

1.4 Outline of this Thesis

The aim of this thesis is to develop a software framework designed to integrate sensors, instrumentation, controllers and actuators. The framework needs to be flexible, scalable, reusable and reconfigurable. The core of the framework will be designed so that any sensing and control system can use this as a base on which to build the system for the specific application. The framework specifies the components, and how these components should interact. Information is gathered from sensors and stored for processing. Controllers can access the sensed data and use it for decision making. Instrumentation can be used for user interaction and monitoring. Actuators are used to implement the controllers' decision in the real world. All of these functions are integrated and coordinated by the framework.

The purpose of using a framework is to decrease development time, and allow more time to be spent on designing the control algorithms, rather than implementing the system.

Chapter two reviews the software architectures on which the framework was based. The conceptual design of the framework is developed. Chapter three presents the major components of the framework. It describes the mechanisms through which the components interact with each other. Chapter four builds on these basic components, and shows how typical operations for sensing, control, actuation, and instrumentation are implemented. Also covered is the use of networking to create distributed systems, data logging, image processing, and the use of dynamically link libraries to simplify the

extension of the system on a PC platform. Chapter five describes several experiments that demonstrate the operation of the framework. Finally, chapter six evaluates the framework and the design approach taken. Use of the framework on other hardware platforms, and for other applications is considered. Suggestions for future improvements to the framework are recommended. Chapter 7 lists the publications that resulted from this research.

2 Framework for Sensors and Controllers

This chapter outlines the basic design of the framework. The design objectives of the framework are: flexibility, reusability, reconfigurability and scalability. The first section considers the various architectural styles to determine which would be most appropriate for the framework. The second section shows how the framework was developed. This leads to the third section, which discusses the hardware and software platforms that the framework could use.

2.1 Architectural styles

To design the framework, various architectural styles must first be reviewed, to find the advantages and disadvantages of each. There are five groups of architectural styles (Table 1 - Bosch 2000): dataflow systems, virtual machines, independent components, data-centred systems and call-and-return systems.

Dataflow Systems
Batch sequential
Pipes and filters
Virtual Machines
Interpreters
Rule-based systems
Independent Components
Communicating processes
Event Systems/Implicit invocation
Data Centred Systems
Databases
Hypertext systems
Blackboards
Call And Return Systems
Main program and subroutine
Object orientated systems
Hierarchical layers

Table 1: A table of common software architectural styles.

Dataflow systems place prime importance on the data in a system. They comprise of data processing areas and the movement of data between these areas (Srini 1986). In a batch sequential system the data are moved in groups or batches. In the pipes and

filters style, a continuous stream of data flows through the pipes and is processed by the filters (Meunier 1995). Dataflow systems are generally very flexible, allowing for changes with minimal re-writing of the code. This is due to the separation of the data and processors. A limitation is that any requirement changes will often involve recoding multiple filters or processes. Dataflow systems have explicitly defined inputs and outputs to the processes. This facilitates the use of verification, authorization and encryption/decryption (Meunier 1995). The fewer links there are between various modules of a program, the easier it is to identify where security is important, and to implement it in the program.

Data centred systems have a central data repository surrounded by processes that access that data (Bosch 2000). The processes scan the system repository for data items that they are able to use. They then take and process the data, and place the results back into the data storage area. If the central data structure acts like the manager of a system – selecting processes to execute – then it is using the blackboard style. In contrast, databases are purely a data storage area, and are often used within data centred systems to organise the data more efficiently. Data centred architectural styles have traditionally been used for applications requiring complex interactions between the various components of the system, where it is not known which processes will be needed and in what order (Bosch 2000). Data centred systems can have poor performance due to time consuming and computational tasks that do not involve the actual processing of data. These overhead tasks include roaming the database or blackboard to obtain (or even find) the correct data, or arbitrating between multiple components trying to access the same information (Bosch 2000). These styles do have easy dynamic addition and removal of data. Because any process can write to the blackboard, any errors that are introduced may propagate through many processes. Hypertext systems refer to systems where clicking on a link will gather information for you from a central data storage area. An example of this is web browsing.

The virtual machine group consists of two styles which effectively create a virtual machine (e.g. Loyot and Grimshaw 1993; Smith and Nair 2005) in software. A virtual machine translates data from one form to another. The interpreter style is based around

a pseudo program which is interpreted (or executed) by the virtual machine. The interpretation engine encompasses the definition of the interpreter and the current state of its execution (Garlan and Shaw 1993). An interpreter has four components, an interpretation engine to do the work, memory to contain the code to be interpreted, the state of the program, and the interpreter state. The data component (which records the program state) gathers input from outside of the interpreter. The memory feeds the program to be interpreted to the interpreter state component. The interpreter state component sends the current state to the interpretation engine. The engine reads the program state and produces an output. Interpreters are used to fill the gap between an engine as defined by the program (what the program wants to do), and the engine available in hardware (the actual hardware available to achieve the programs objective). Rule based systems makes use of pattern matching and the context that the information is in to determine what actions to perform on data (Bosch 2000). As rule based systems are suited to pattern matching and rule interpretation, they are more suited to interpreting the data in a specific application, rather than for a generic sensing and controlling framework.

The independent components group consists of styles that have components that process data, which are linked by message passing. This message passing could be point to point, synchronous, or asynchronous (Bosch 2000). This is similar to the pipes and filters, but the message passing is more flexible in this style. Implicit invocation (e.g. Garlan and Scott 1993) modifies this by a component announcing an event. Other components register an interest by associating a procedure with it. Then when the event is called the system invokes all the procedures that have been registered with that event. The event handling mechanism requires a certain amount of processing to call the required components. Where a reply is needed, two events need to be sent. This may lead to the fragmentation of the order of operations. These styles do allow for easy addition and removal of components to the system.

Call and return systems refer to architectural styles that call subsystems which then return control to the main program when the subsystem code has finished processing (Bosch 2000). The main program and subroutine style often mirrors the programming

language in which the system was written. This means the programming language used influences the type of architecture used. This is due to some architectures being very difficult to implement in certain programming languages. The programmer may be very familiar with certain languages, so architecture styles that this language supports are preferred.

The object orientated style (e.g. Nascimento and Dollimore 1993) organises the system in terms of objects. Only one copy of the code is put into memory, which can be called or executed multiple times. Object orientated design allows inheritance, encapsulation, abstraction and polymorphism. Inheritance creates new classes that are based on existing classes. This new class can override the existing classes' methods, and/or create new ones of its own (Jacobson 2004). Encapsulation is providing a well defined interface to a set of functions in a way that hides their internal workings. This can be useful for security and simplicity reasons. An example of this is DLLs. A well defined interface allows multiple software products to access its features – but it hides how it does this. Abstraction is reducing the visible details by hiding some of the fine detail, allowing the user to focus on the big picture. If a program was going to be released to the general public, giving fewer options to the user can give him or her a higher satisfaction rating of the software product as they are not overwhelmed with too many options. Menus can help with this by giving only a few options at one time. Polymorphism is allowing a single definition to be used with different types of data (Jacobson 2004). If many functions are used to perform the same basic operation on various types of data, this can be made into one method that is redefined for different types.

The last style in this group is hierarchical layers (e.g. Dittenbach et al. 2000). This is a style where the program is split into hierarchical sections. Each layer provides a level of abstraction from the layer below, and presents an interface to the layer above it. The layer style organises tasks based on their abstraction, rather than their computational relationship. There can be a decreased performance due to switching between the layers. This can happen when the top layer sends data to the lowest layer, which has to pass through all the intermediate layers, which causes a delay from the overhead of

multiple calls. A relaxed hierarchical system allows direct access, reducing this delay (Bosch 2000). This causes extra complexity when layers need to be changed or removed. Changes to a component or layer can be done with minimal effort provided that there are only few dependencies between the components on a layer. However, if multiple layers need to be changed, the maintainability of the system is compromised. A higher level layer can contain fault tolerant systems for the layer below. In this way the higher level layer may be able to deal with faults that the lower level layer cannot (Bosch 2000).

These five groups of architectural styles present various methods that could be used for designing an integrated sensor and control framework. To design the best framework, a combination of styles is likely to achieve the best result, as the disadvantages of each style can be minimised by incorporating other styles.

2.2 Development of the Framework

The traditional approach is to make the system controller centric (Figure 2). This results in a system that is difficult to reconfigure, as when new sensors and actuators are added, the controller must be explicitly modified. Complex controller systems (consisting of many sensors and actuators) can be difficult to maintain unless the controller program is well structured.

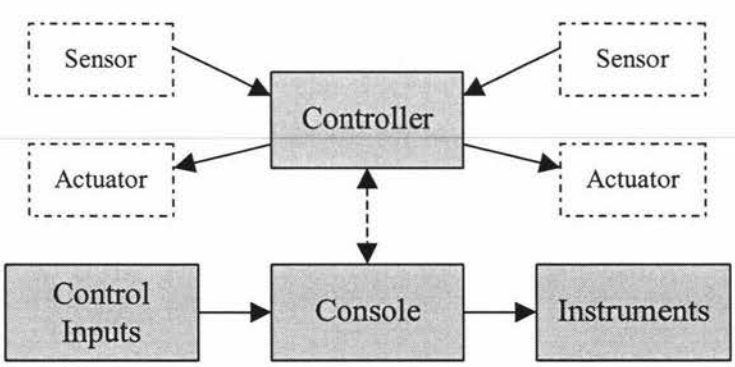


Figure 2: Controller centric system.

A different model is the distributed control system (Figure 3), where the controller is not longer centralised, but split into multiple smaller controllers. Each small controller

deals specifically with it own local sensors and actuators, but shares data with other controllers to implement the complete system. This approach is more flexible than the traditional method, as a change in a sensor will only result in the change of one controller. An increase in the number of sensors results in an increase of controllers. However, this means there is an increasing complexity in communication between the controllers.

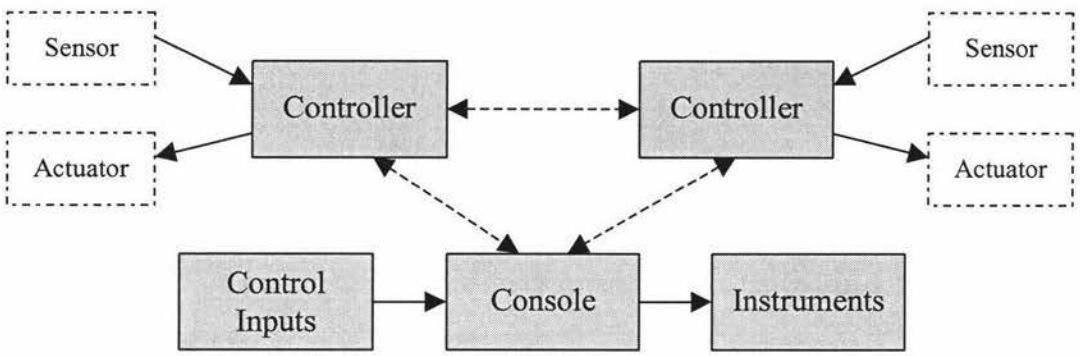


Figure 3: Distributed control system.

A different approach is a framework where the core of the system is separated from the controllers (Figure 4). The core of the system handles the passing of data, and executing the processors (which operate on the data). The advantage of this system is the core part of the framework stays the same. Additional controllers can be added with no change to how the system operates.

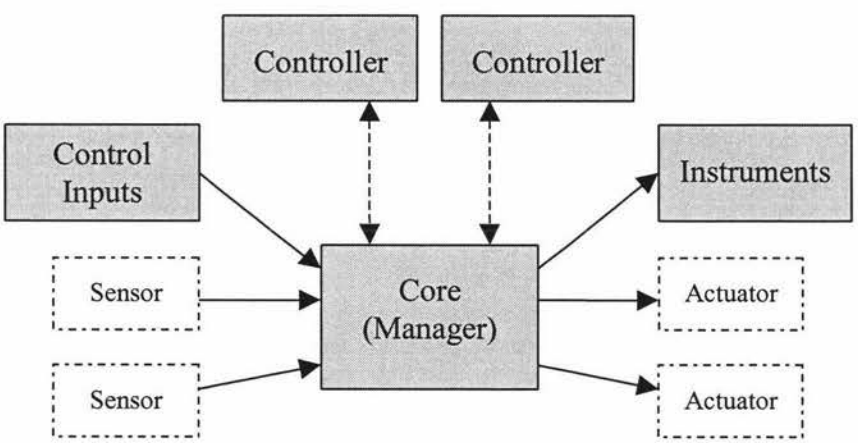


Figure 4: Model of a framework that controls sensors and actuators showing the controllers outside the core of the system.

Parts of the framework that process the data have been separated from the core, leaving the data storage areas within the core. By also separating the data storage area from the core, these data storage areas become accessible to other parts of the framework (Figure 5). This allows maximum flexibility, as it keeps sensing and actuation processors separate from operations preformed on the data. A separate data storage area is required for each data source, although a single data source may have multiple output storage areas. These storage areas provide the interface between the data sources and any associated data sinks.

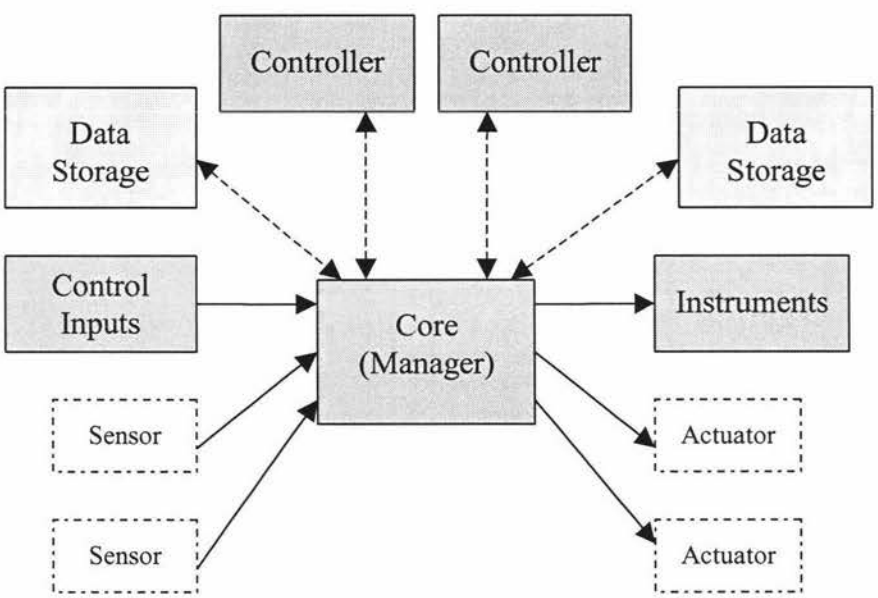


Figure 5: Model of a framework that controls sensors and actuators showing both data storage and controllers outside of the core of the system.

The data storage area for a single item of data is called a dataslot. It controls adding, updating and other operations related to the data. The parts of the program that process the data are called processors. These processors take information from one or more dataslots. After this data is processed, the results are written to one or more other dataslots. Using this design dataslots and processors can be joined together to achieve the desired system operation.

The core controls when operations are scheduled in the framework. During operation of a typical control system, data is acquired from a sensor and placed in a dataslot. Then a processor will use the sensor data to produce new information, based on the type of controller implemented, with this new information placed in another dataslot. This data will then be used to drive an actuator.

Separating the system in this way allows concurrent processing. For example, for one task a processor could be acquiring data from the sensor at the same time a sink processor could be notified that data is available to be transmitted to an actuator. This allows reduced latency for any one part of the system. This is a data driven design as processors do not start until new data has arrived (Figure 6).

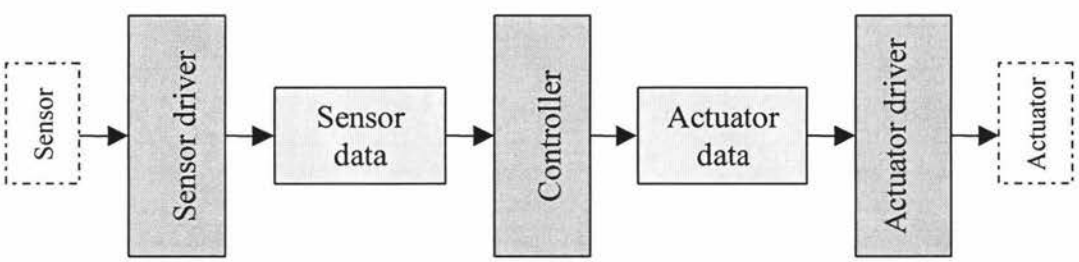


Figure 6: Data driven design for an integrated sensing and controlling framework. This shows one complete task. One or more processors can be processing at the same time.

This architecture incorporates the useful features of the software architecture styles defined above in section 2.1. By separating the data from the processes, like in the dataflow systems, the inputs and outputs can be explicitly defined. This helps with the removal or addition of dataslots or processes. Implicit invocation can be used by processes registering with each dataslot that provides input data to the process. This is used with the setup of the program. All the links are created during start up, so during the running of the program, the links can be used to go directly to a location, instead of searching for the correct location. Data centred systems (the system described in this thesis has data centred characteristics) have a central data repository, so data is available to the whole system. An object oriented architecture can reduce the amount of code needed to implement the framework, provide structure for the design of the framework's code, and restrict access to ensure all data and processes can only be

accessed at the correct place. Hierarchical layers are useful as one process could be called to move a robot to a certain location, but other sub-processes could be called from that main movement process to deal with the speed profile and direction of how to get to that location.

2.3 Hardware and Software platforms

A number of different hardware platforms could be used to implement such a sensing and controlling framework (Table 2). A PC, microcontroller, PLC (Programmable Logic Controller) or FPGA could be used, depending on the control application. FPGAs are hard to develop for. PLCs are generally used in industrial applications. Its hardware does not allow the flexibility of a PC. Microcontrollers would perform well on smaller control systems, but have problems with video processing and displaying. The PC is a flexible development platform, which has advanced and mature development environments. However in many simple control applications, using a PC would be overkill. The key attributes for different potential framework implementations are compared in Table 2. A PC will be used as the hardware platform to implement this framework.

	Framework using C++ on a PC	**Traditional controller on a PC or microcontroller**	**LabView on a PC**	**PLC**
Reconfigurability				
• Programming	Configuration specified in a text file; no programme change required	Recoding the application for each project when a change is made	Graphical icon based; flow diagrams need to be changed	Ladder logic or function blocks need to be changed
Scalability				
• Connections	Limited by processor speed and bandwidth	Limited by processor speed and bandwidth	Limited by processor speed and bandwidth	Limited by I/O and bus bandwidth
• Overload performance	Prioritise certain processors, the data rate of other processors will decrease	Implementation dependant; generally processor lag will increase	Implementation dependant; processor lag if not using threads	Possible data lost or inaccurate results
Flexibility				
• Effort to add new sensor, actuator or controller	Only change is to configuration file, or add new code via a	Implementation dependant; add extra code to project,	Use a DLL or recompile code into main application	Modify ladder logic, then recompile

	DLL	ensuring code fits into current resources		
• Range of applications	Wide range due to ease of coding and range of devices designed to be connected to a PC	Wide range due to relatively low cost of microcontroller implementation	Wide range due to ease of coding and range of devices designed to be connected to a PC	Generally limited to industrial use
• Control architecture	Flexible control architecture, easily altered	Inflexible	Partially flexible	Inflexible
Performance				
• Limits to performance	Port and CPU speed	Port and CPU speed; memory and IO availability for microcontroller	Port and CPU speed	Controller speed and scan time
• Latency	Minimal due to prioritised queues and multithreading	Implementation and problem size dependant	Implementation dependant	Controller speed and scan time
Advantages / primary strengths	Framework can be used over multiple projects; easily reconfigured	Slightly faster execution due to less overhead	Visual language is intuitive to programme; large base of pre-programmed modules	Embedded design; small form factor
Disadvantages / primary weaknesses	Overheads from reduced coupling between components	Application needs to be designed each time; adding new devices may require major system changes	Application needs to be designed each time; expensive	No threading functionality

Table 2: Feature comparison between the proposed framework, and other platforms for controller implementation.

A flexible programming language will be required for this framework. This language will need to be able to implement the data driven design of the framework. An object orientated language would fit this need. Inheritance is useful as there is a need for many different types of dataslots and processors. By deriving these from a dataslot and processor base classes respectively, a common interface between the different types simplifies the design. An object oriented design easily allows many different types to be created – all using the same basic set of code. C++ was chosen to implement this framework, as it is object oriented and is able to implement the data driven design.

2.4 Summary

This chapter discussed several possible software architecture styles. Then the framework is constructed using a data driven design. With this style, all of the components are separated from the core to allow flexibility in configuration. The handling and passing of data in the system is split amongst the frameworks components. This reduces a bottleneck in the system and allows multiple processors to run at the same time. This framework is implemented on a PC using C++ as the development platform, as they meet the software and hardware requirements of this framework.

3 Component Mechanisms

This system can be thought as having four main components. These are the mechanisms that interact with each other to create the system. It is a data driven design, so there are dataslots to store the data. Processors are used to process the information in the dataslots. Queues are used to queue processors, and schedule their execution (as processors can have various priority levels). The last component is the configuration mechanism. This sets the system up to operate how the system designer wishes.

The way the framework has been designed, as described in the previous chapter, the data and processors have been separated. This leads to two ways in which the system can be thought of: data focused, or processor focused. The data focused interpretation focuses on the data, with the links between the dataslots made up of processors. In contrast, the processor focused interpretation focuses on the processors, with the links between the processors made up of data slots. Both interpretations are valid, as neither dataslots or processors are more important than the other.

3.1 Dataslots

Dataslots are used as a storage area for information, both as data sources and to processors. Methods of the dataslot class are used to enter information, for a processor to inform the dataslot that it has updated and modify attributes of the dataslot. The class hierarchy is shown below in Figure 7. Three types of dataslots are shown.

3.1.1 Types of Dataslots

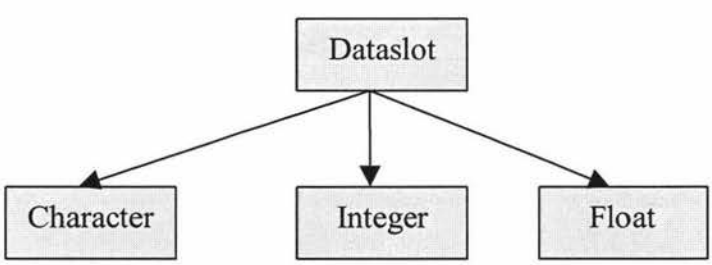


Figure 7: Class hierarchy of various Dataslot types.

The purpose of these basic classes is to store their respective information; character, integer and floating point data types. Other custom dataslots can be implemented by deriving them from the base class and modifying the storage type. In this way all of the data control mechanisms of the framework are retained.

3.1.2 Operation of a Dataslot

Data is added to a particular data slot through a virtual method called `add_data`. This method is called by a processor. The benefit of this is to go directly to the particular `add_data` related to the type of data, eg: character `add_data`, or integer `add_data`.

Any process which makes use of data within a dataslot, is registered with that dataslot to receive notification when the data is updated. This is called implicit invocation (Bosch 2000). This ensures that the processor is operating with the correct data. The method to do this is another virtual method called `data_updated`. The same processor that added the data calls the `data_updated` method. The reason why `data_updated` is not automatically called by the `add_data` method is to allow the process to store partial results, and give the `data_updated` signal on completion (for example when processing an image or other large block of data). It also allows a dataslot to activate one or more other processes, without necessarily providing input data (for example using one process to trigger another process when a set of conditions have been met).

3.2 Processes

The object of a `Processor` is to process the data in some way. This is done by having an array of dataslots that the processor reads from, and having another array that it writes to. These arrays are called the `datain` and `dataout` arrays respectively.

3.2.1 Types of Processors

The framework provides a number of different Processor types. These Processors can be grouped by the type of operation that they perform, as shown in Figure 8.

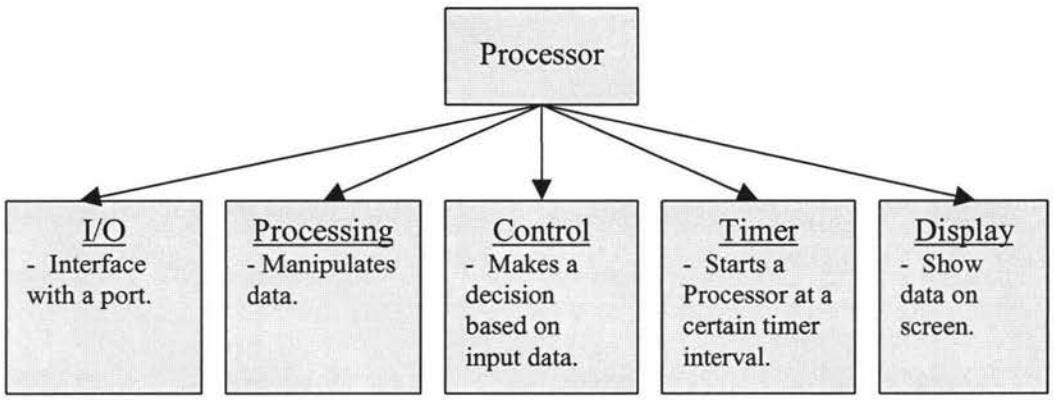


Figure 8: Processor types grouped by type of operation that they process.

The I/O processors deal with interfacing between the real world and the PC. This would include the serial and parallel port processors. The processing processors deal with manipulating the data, or extracting the useful information. A GPS processor that extracts the useful information from a GPS string, converting it from text to numbers representing latitude, longitude, altitude, etc. would be an example. The control processors deal with making a decision based on the data in its datain array. A control processor would use the position of the ball to instruct the robots to move to certain locations in the playing field. A timer processor is used to start another processor at regular time intervals. Display processors are used to display the data in certain ways on the screen. Display processors include both inputs and outputs, so for example a button or a gauge would be a display processor.

3.2.2 Operation of a Processor

A processor is activated by calling its `execute` method. This method acquires the data from the datain array, does it processing (dependant of type of processor), then writes any new data that is produced to the particular dataslots in the dataout array. On completion, the `data_updated` method of each dataslot written to is then called. This tells the dataslot that new information has been entered, and queues any downstream processes.

3.3 Queues

The purpose of queues in this program is to facilitate quick responsiveness amongst the different areas of a program, and to ensure that achieving one task does not impede on other tasks.

When new data is placed in a dataslot, all of the processors listed as listeners in that dataslot are queued for execution. This breaks the potentially long task of executing all the downstream processes into several smaller independent tasks. An example of this is taking information from the serial port, and passing it to multiple processors at once. These processors could display the data on screen, transmit the data over the network, and break the data up into its components. Once the current task is finished, the next item in the queue is then executed. By splitting the downstream processors up, it allows higher priority tasks to execute by being on a higher priority queue.

3.3.1 Operation of a Queue

When a processor calls the `data_updated` method of a dataslot, the dataslot queues all the listeners that it has. It does this by calling the corresponding queue's add method for each listener. This queues each processor for execution.

Each instance of the queue class has an execution thread. If this thread is in the idle state (not currently executing) then when a processor is queued, it is immediately executed in this thread. If the thread is busy, the processor is queued for later execution. When the current task is completed, the next processor in the queue is popped off and executed. If there are no processors queued, the thread returns to the idle state.

Figure 9 shows how the queue is used. For example the GPS string dataslot passes the GPS processor to the queue. The queue adds this processor to the bottom of its queue. Because no other tasks are in operation the queue thread then calls the execute method of the GPS processor and removes it from the queue.

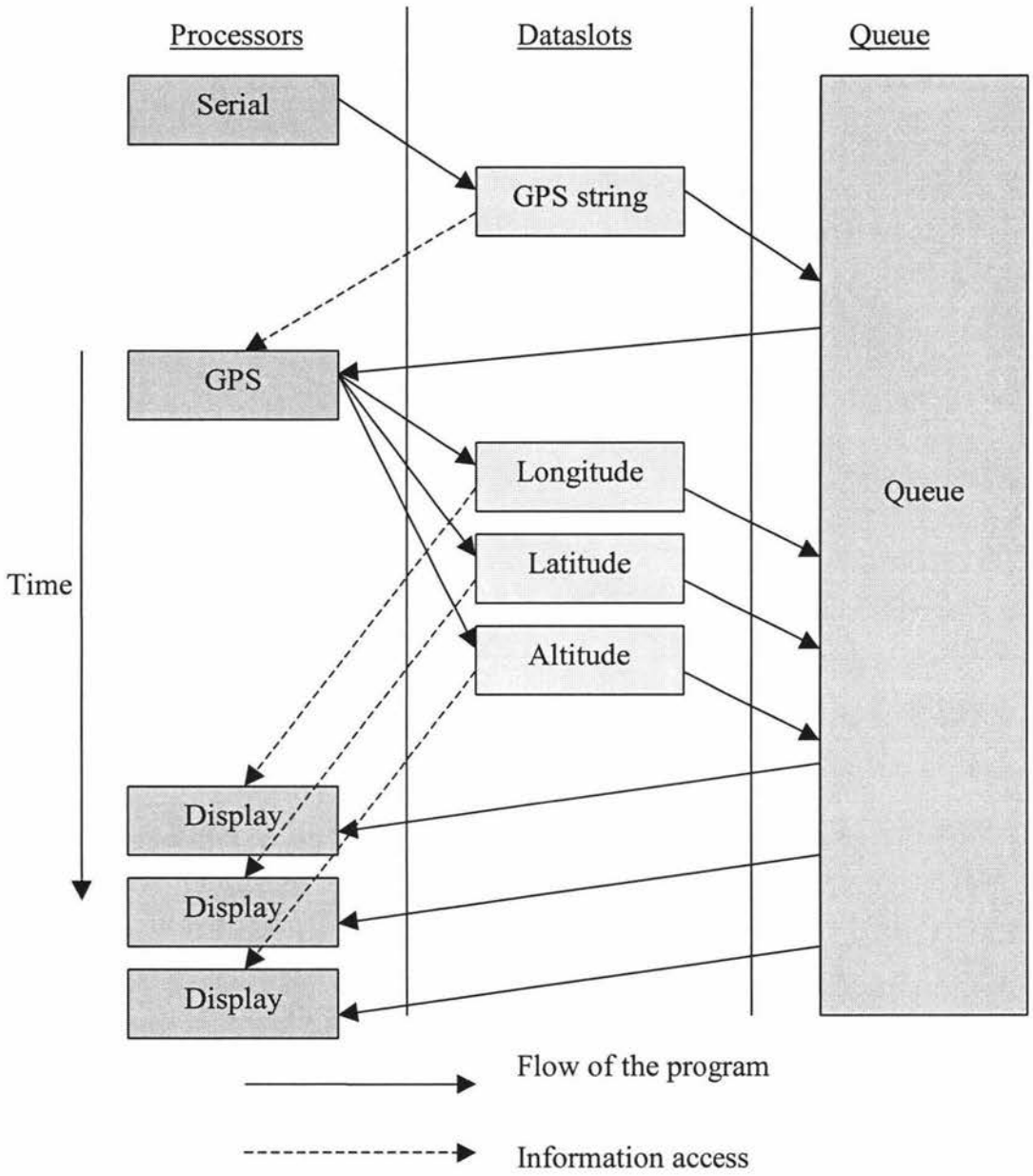


Figure 9: Showing the operation of the program with multiple queues at different priority levels.

3.4 Configuration

The problem with most sensing and control systems is that they are hard coded. These systems are only designed for a certain set of inputs (sensors) and outputs (display and actuators). A more flexible way is to allow for various types of inputs and outputs to the system. A flexible configuration is part of the framework described in this thesis. Configuration also covers linking the various components of the framework together.

Configuration is done using a text file. Having a graphical user interface for configuration would be a more user friendly design, but this is beyond the scope of this thesis. By having multiple configuration files, the same framework can be used for vastly different applications.

3.4.1 Configuration File

The configuration file defines the dataslots and processors that are going to be used – and the relationship between them.

To demonstrate the use of a configuration file, a simple application which acquires information from a GPS device is used as an example. Figure 10 shows the dataslots and processors of the system. A GPS device is connected to the computer via the serial port. Information will be requested from it every five seconds. The information will then be displayed on a window. This figure is used to create the text based configuration file, listed in Figure 11.

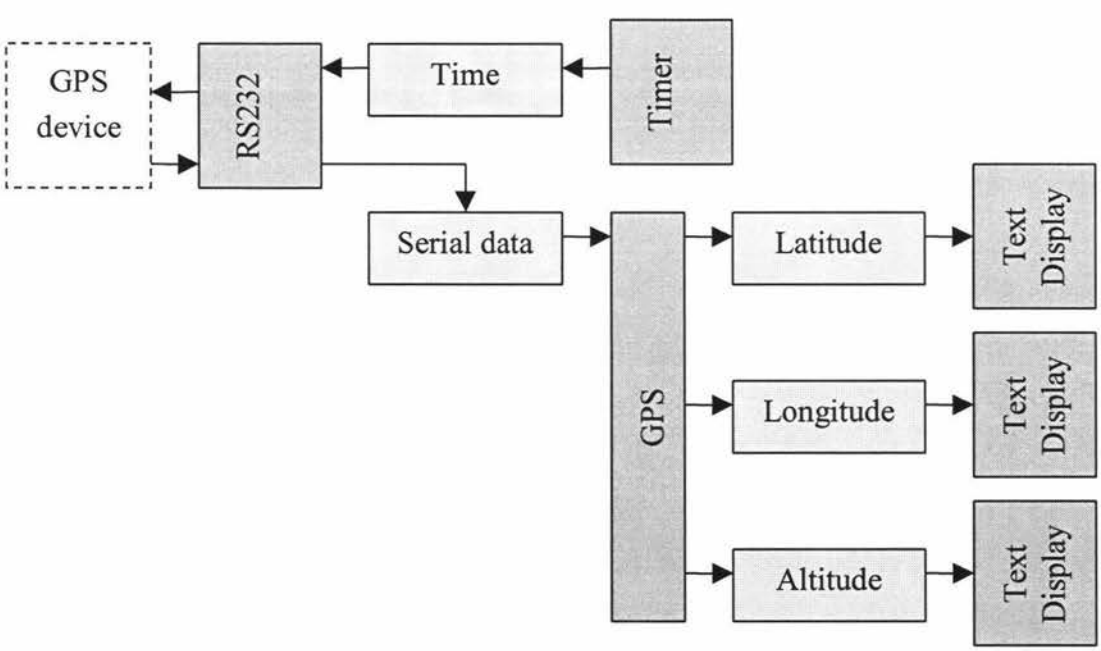


Figure 10: GPS processor and dataslot configuration.

```
// Configuration file for GPS device attached to the serial port

// Creating a integer dataslot that is activated every 5 seconds
DATA=\int[1]
      NAME="time"
      TIME=5

// Dataslot to store information from the serial port
DATA=\char[100]
      NAME="Serial data"

// Store the longitude data
DATA=\float[1]
      NAME="longitude"

// Store the latitude data
DATA=\float[1]
      NAME="latitude"

// Store the altitude data
DATA=\float[1]
      NAME="altitude"

// Creates a window.  Any display item (listed below - text box, etc)
// is automatically put on the window last defined
PROCESS=\Display
      NAME="GPS Window"
      LABEL="GPS"
      START_POS = (300,250)
      SIZE=(400,400)

// Create a timer process
PROCESS=\timer
      NAME="Timer"
      DATAOUT="time"

// Create a process to communicate to and from the serial port
PROCESS=\RS232
      NAME="Comm Port"
      COM="COM1"
      BAUDRATE=9600
```

```
      ByteSize=8
      Parity=0
      StopBits=1
      EvtChar=\n
      DATAIN="time"
      DATAOUT="Serial data"

// GPS processor that splits the GPS string up into its component
// parts
PROCESS=\GPS
      NAME="GPS"
      DATAIN="Comm_Data"
      DATAOUT#Longitude="longitude"
      DATAOUT#Latitude="latitude"
      DATAOUT#Altitude="altitude"

//Display the longitude data on the window defined above
PROCESS=\Text
      NAME="Long"
      LABEL="Longitude: %f"
      DATAIN="longitude"
      START_POS = (30,100)
      SIZE=(150,20)

//Display the latitude data on the window
PROCESS=\Text
      NAME="Lat"
      LABEL="Latitude: %f"
      DATAIN="latitude"
      START_POS = (30,150)
      SIZE=(150,20)

//Display the altitude data on the window
PROCESS=\Text
      NAME="Alt"
      LABEL="Altitude: %f"
      DATAIN="altitude"
      START_POS = (30,200)
      SIZE=(150,20)
```

Figure 11: Configuration file for a GPS device attached to the serial port.

Figure 11 defines the dataslots, processors and queues of the system. In this example defining the queue is left out. If that is the case, a default queue is created which is used for all processors. Five dataslots are defined, and seven processors. This is one extra than what is shown in Figure 10. This is because an extra processor is needed to create the window for displaying the text output.

3.4.2 System Configuration

On start up the framework loads the configuration file. As it reads each line, it processes the keyword and value pair, creating the necessary data structures. There are two sections to the data file: the first defines the data slots and the second defines the processes. The 'DATA' keyword specifies the type and size of data contained in a data slot. The dataslot name must be unique as it is used to identify the data buffer to any processes that may use data from or produce data for that slot. Custom data slots allow more complex data structures to be implemented. The 'PROCESS' keyword defines a data source or sink and specifies the procedure that will implement the process. This may either be a built-in process, or contained within a DLL. Data sources use the 'DATAOUT' keyword to specify which data slot the output data is to be written to. If a process is a data sink, the 'DATAIN' keyword indicates where the data is to come from. This registers the process as a sink on the corresponding data slot. Many of the remaining attributes are process dependent and may specify additional setup information required by the process, such as what port or protocol to use to gather the data from a sensor or the location of a virtual instrument on the screen. Where necessary, a process may use additional keywords to distinguish between multiple inputs and outputs. Figure 12 shows the class structure which is used in the parsing of the configuration file.

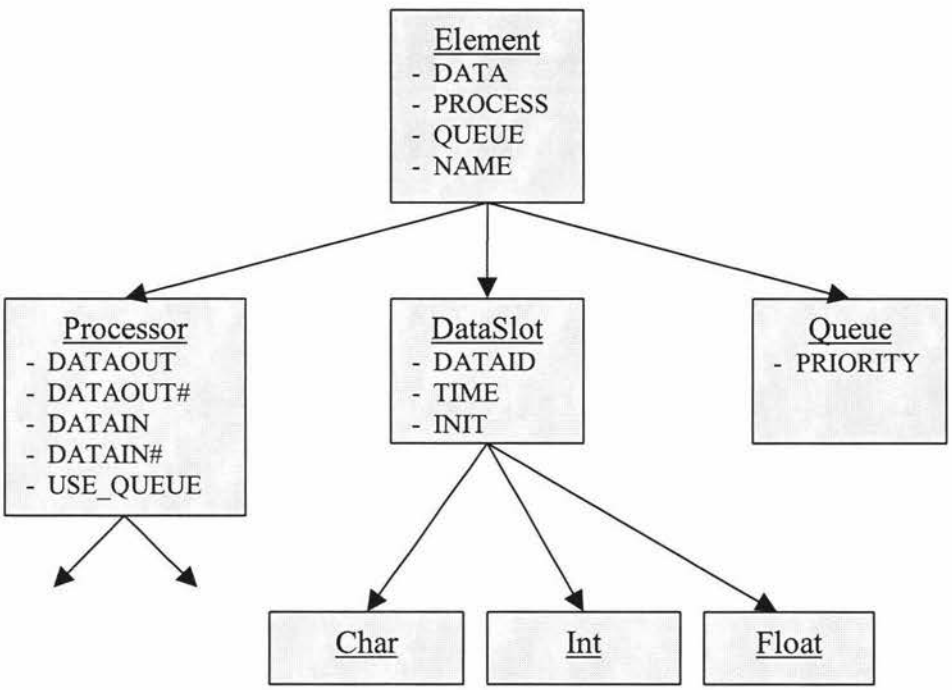


Figure 12: Class structure of the framework. Used in the parsing of the configuration file. The keywords that each class recognises are stated.

To demonstrate how the parsing works, Figure 13 has an example dataslot definition from a configuration file.

```
DATA=\char[1]
        NAME="Lamp"
        DATAID=1
        INIT="0"

PROCESS=\Display
```

Figure 13: Example dataslot to demonstrate parsing.

The first line would be read. The keyword DATA would be recognised at the element class, and a new char dataslot containing one storage element would be created. A pointer is kept to the newly created dataslot so that successive attributes can be applied to the correct object. When the NAME attribute is read, it is passed to the char dataslot that was just created. The search for the NAME attribute is passed from the top of the

tree to the base class – element. The reason for this is that common keywords are most likely to be relvevant, and therefore parsed, higher up in the class structure. For example Figure 13 has five keywords, two of them are directly recognised by the element class. The element class recognises the NAME keyword, so it is passed there, but the name is stored in the char dataslot. The next keyword is DATAID. Again this is passed to the char dataslot for parsing. The search begins with the base class. This is not recognised at this level, so the search works down the tree to its sub class – dataslot. The keyword is recognised and the value parsed at that level. This process continues for all lines in the configuration file. Parsing of each line in the file involves moving through the class structure to where the keyword is recognised. This reduces the amount of code needed in each class.

The last line in this example is "PROCESS=\Display". The purpose of this line is to create a new window. When the element class encounters this keyword, it recognises that it is creating a new object. So before that line is processed, the finish method of the char dataslot class is called. This method is called at the end of every object specified in the configuration file. This purpose of this method is to check all of the attributes of the dataslot, processor or queue that just has been created. If some required attributes are missing errors are thrown, otherwise appropriate default values are set.

Once the whole configuration file has been read, the initialisation method for each processor is called. This method performs any processing that involves more than one object in the configuration file. For example the network processor on a client would send information about its dataslots to the server so that the server knows what information to send to that client.

3.5 Summary

This chapter discusses the major components of the framework – dataslots, processors, queues and the configuration. Dataslots are the locations where individual items of data are stored. Processors use the data in dataslots to execute an operation. Queues are used to queue processors for execution when data is changed. The configuration file defines the dataslots, processors and queues that are used in a particular application.

The chapter covered how these components interact and fit into the overall class structure.

4 Application Details

Now that the major components of the framework have been presented, the implementation of typical operations required in an integrated sensor and controller framework can be described. These operations include acquiring data from sensors, handling multiple dataslots, actuators, controllers, virtual instrumentation, networking, data logging, image processing, timers and dynamically linked libraries. Acquiring data from sensors covers the hardware interfaces, and how the data is acquired. Handling multiple dataslots discusses how dataslots are handled for a processor that has multiple dataslots on the input and output. Using actuators is discussed, along with using various controllers to use the data from the sensors to control the actuators. How virtual instrumentation (such as a button on a window) fits into the framework is shown. Networking is an important part of the framework as it allows communication with remote vehicles. Data logging is used to review data at a later date. Image processing is used as sensing and control systems often have cameras. How this information is stored is discussed. Timers are used to activate events at certain time periods. Lastly, dynamically linked libraries are used to extend the framework – to add additional dataslots and processors.

4.1 Sensors

A sensor and controller framework must provide access to real world devices. On a PC platform, this is achieved through the many ports that are available: RS232, RS485, parallel, USB, Ethernet, IEEE-1394, etc.

The best way to access data via a port is to use an interrupt driven system device driver. External activity is buffered in the port hardware, and when data is available, the hardware interrupts the main processor. Within the framework, this is best handled by having a thread waiting for the device driver interrupt. This way, as soon as data is available, the thread acquires the information and places it in a dataslot.

If no such interrupt mechanism is available, the sensor thread can regularly poll the device to check if data is available. There is a trade off with the use of polling. The

more often the thread checks for data, the faster the information is available to the rest of the program. However, continual polling requires more CPU time, both in terms of actually checking for the presence of data, and also in context switching between the polling thread and other processes. A high polling rate leaves less time for more useful data processing.

The third option is to have a sensor produce new information on request. A processor would trigger the acquisition of data. The sensor would then acquire and store the new data in its memory. After a predetermined time (allowing time for the sensor to acquire and store the data) the processor would read the data from the sensor and write the information to a dataslot. The disadvantage of this is the latency between when the sensor produces new data, and when a processor in the framework reads the data from the sensor. The data in the sensor could be read more frequently to reduce the time between producing the data and reading the data, but this creates extra load on the framework.

4.1.1 Parallel port

In demonstrating this framework, the parallel port has been used to attach an I^2C bus (Philips Semiconductors 2000) to the PC and a heater controller circuit. The parallel port is an attractive port to use with non commercial systems for two reasons. First, the parallel port is not often used and is therefore is more likely to be available. The second reason is communication. It is relatively easy to develop software that communicates through the parallel port using a generic system device driver.

The I^2C bus is a two wire bus; with one line for data the other for the clock. Four pins of the parallel port are used (Appendix A): clock out, data out, data in and ground. The master sets the clock speed on an I^2C bus; in this case the PC always acts as the bus master.

Currently parallel port communications do not involve a separate thread. The reason for this is there is no facility for the port to indicate when data is available. However, the use of a thread, along with a device driver that indicates when data is available, would

be recommended in non demonstration systems. In such a system the I^2C clock would be generated directly by the device driver. Currently it is generated at the application level. A disadvantage of the current method is between changing the output of the I^2C clock between high and low, CPU time is taken up waiting till the next clock transition.

4.2 Data Converters

One of the types of processor is a data converter. Such a processor takes information from the dataslots on its input, processes the data, then puts the new data to its output dataslots.

When a processor has multiple input or output dataslots, it is necessary to distinguish between the different inputs or outputs. Initially, in the configuration file the dataslots were only identified by the DATAIN and DATAOUT keywords (Figure 14). They could be distinguished by their order in the configuration file. However as there is nothing to enforce a particular order, it would be easy to make mistakes that would be difficult to debug. For robustness, a change of order of parameters in a configuration file should not change the operation of the system. The names of the dataslots could be used to differentiate between the dataslots (Figure 14). However, dataslot names are global, and there may be multiple sensors generating similar data. They cannot all be given the same name. Also, multiple instances of the same sensor cannot use the same name.

```
PROCESS=\GPS
        NAME="GPS"
        DATAIN="Comm_Data"
        DATAOUT ="Longitude_deg"
        DATAOUT ="Latitude_deg"
```

Figure 14: Initial processor with multiple dataouts.

Another option is to use an extra tag in the configuration file to differentiate between the dataslots. A parameter tag could be added to the DATAOUT keyword, with a separator symbol between them. This tag would be processor specific. The framework

would recognise the DATAOUT keyword and process it appropriately during configuration. Later, during the finalisation step when loading the configuration file, the separator would be recognised, enabling the processor to associate the dataslot with a particular input or output. This is shown in Figure 15.

```
PROCESS=\GPS
    NAME="GPS"
    DATAIN="Comm_Data"
    DATAOUT#Long_deg="Longitude_deg"
    DATAOUT#Lat_deg="Latitude_deg"
    DATAOUT#Long_min="Longitude_min"
    DATAOUT#Lat_min="Latitude_min"
    DATAOUT#N/S="N/S"
    DATAOUT#E/W="E/W"
    DATAOUT#FIX="Fix"
    DATAOUT#Sat="Sat"
```

Figure 15: An example of DATAOUT# being used in a configuration file.

Consider a GPS data converter. It would take the text string received from the GPS unit (Figure 16), and split it into its component data parts (the interpretation of this string is given in Appendix C). Each wanted component part is placed in a separate dataslot for subsequent processing. Figure 15 shows that eight components of that GPS are taken and placed in dataslots.

```
$GPGGA,092204.999,4250.5589,S,14718.5084,E,1,04,24.4,19.7,M,,,,0000*1F
```

Figure 16: A string from a GPS device in NMEA format.

To improve execution efficiency, the references to the input and output dataslots are stored as an array. Rather than decode the parameter tags each time the data converter is called, these are decoded once when the configuration file is loaded, and the corresponding indices in the array recorded. When the data converter is called, these indices are used to indirectly access the correct data slots.

4.3 Actuators

Actuators are used to implement the control decision of the controller. Just like sensors, they are interfaced through the ports of a PC.

4.3.1 Switching a device

Any binary (on/off) device may be switched via the parallel port. This includes lamps, signal lines to microcontrollers and other devices, and LED's. For a demonstration (Chapter 5.3), switching of a lamp is done in this manner. The heater control circuitry switches the heater on and off (Appendix B).

4.4 Controllers

A controller takes information from various sources (sensor, video camera), and makes a control decision based on this input information. This decision is then sent to real world devices (actuators, transducers) to implement. A controller can be implemented using one or more processors in the framework. Since a controller converts the values on the control inputs into values on the control outputs, it is effectively a specialised data converter, and operates using exactly the same mechanisms described in the previous section. Any control algorithm could be implemented within the control process: conventional linear PID controllers, fuzzy based controllers, neural network based controllers, etc.

Usually most processors will be executed by the queue thread, by running the execute method in each processor. In situations where a single processor will do processing that takes a significant amount of time, the execute method can create a separate thread to do its processing. This will allow the queue's thread to start executing other processors in a timely manner, resulting in a small latency for any one processor.

4.5 Virtual instrumentation

Virtual instrumentation consists of displaying information from sensors and actuators via simulated instruments on a computer monitor. It is important as it displays the state

of the system (including key inputs and outputs) to the user. A virtual instrument panel can also provide user input components as well as output components.

A virtual instrument panel is implemented on the PC by creating a window that contains the required inputs and outputs. In terms of the framework, this has a separate process that manages the display and user input, and provides an application level driver between the user and the underlying operating system. The instrumentation window is therefore just another process, and is created by specifying the appropriate keywords and values in the configuration file (Figure 17). This gives the window a name, a title to display, the position and size of the window on the display.

```
PROCESS=\Display
        NAME="WINDOW_1"
        LABEL="Integrated Sensor and Controller Framework"
        START_POS = (300,250)
        SIZE=(400,400)
```

Figure 17: Configuration file code to create a window.

4.5.1 Input

Buttons, slider controls and text boxes can be used as inputs to the system. They are contained in an instrumentation window. When an input virtual instrument is activated, the framework must link this to the corresponding handling process. When a window item is created, a unique identification number is generated for that item. A handle to each processor is placed in an array with the position associated with the item's identification number. When the item is activated (such as a button being pressed), a window message is generated automatically. This message contains the window item's identification number. This number is then used in the framework to get the correct index into the array stated above. From here the item executes method is called.

4.5.2 Output

Text and static boxes, gauges, dials and graphs are outputs from the system, contained in a window. They operate like any other processor in the framework. They are data

sinks, with the processor modifying the contents of the corresponding virtual instrument in the window.

Take a gauge as an example. A dataslot gets updated with a new value. The gauge processor is registered as a listener to that dataslot, so the gauge processor gets queued. When executed, the processor uses the new information from the dataslot to determine a new position for the needle in the gauge. Then the processor requests the window to refresh the corresponding region on the display, updating the gauge in the window to reflect the new needle position.

4.6 Timers

Timers are used to activate processors at certain time intervals. One use can be to periodically query data from a sensor. The timer process triggers a timer dataslot (by indicating that new data has arrived) when its time has elapsed, causing any listening processors to be queued.

Only a single timer process is needed. By using multiple timer dataslots, each with their own time interval, multiple timing events can be activated. Upon initialisation of the program, the timer process obtains the intervals specified for each timer dataslot. It then creates an index of when to activate the next dataslot. For example, consider a one second and a two second timer dataslot connected to the output of the timer process. Upon start up, the timer process looks at these two the dataslots and determines that the one second dataslot is to be activated next. The timer process then sleeps until one second has passed. It then wakes up and triggers this dataslot, queuing any listening processors. It then determines that the next event will be in 1 second, so the timer processor then sleeps for another second. After waking up, it triggers both the one and two second dataslots, and so on.

For the example in Figure 18 shows data being gathered from an ultrasonic sensor. The *Timer* process will activate every four seconds. The *Four second* dataslot will then be updated by the timer. This dataslot will then queue the *RS232* processor to run, which then acquires the data and places it in the distance dataslot.

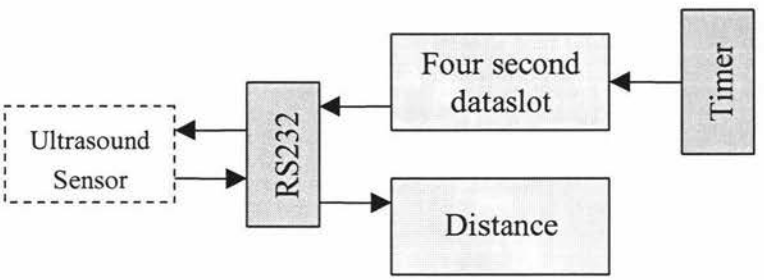


Figure 18: Demonstration of a timer with an ultrasound sensor attached to the serial port.

4.7 Networking

If the sensing and controlling framework is being run on a distributed or remote platform, there needs to be a way to transmit the data between systems at different locations. The purpose of the network processor is to mirror the data in the dataslots at each end of the network connection. This requires that the framework is run on both the remote platform and on the local system. In the current implementation, the network operates over TCP/IP, although in principle, any network channel could be used.

If a network processor is defined in the configuration file, upon parsing of the file, the initialisation method of the network processor is called. The setup process depends on whether the program is operating as a server or client. In the client one set of send and receive network threads are created upon start up, whereas in the server, a set of send and receive threads is created for each client that connects. Figure 19 shows what happens on the client and server ends when a client establishes a connection to a server. Using separate threats for network communication allows the server and client to send and receive data without significantly slowing the rest of the framework. Associated with the send and receive threads, a network queue is created. The purpose of this is to queue the list of dataslots that contain data to be sent over the network. The send thread reads the queue and transmits the data to the other end.

On initialisation of the client, it requests access to a subset of the dataslots on the server during operation, by sending to the server the list of dataslots that it has available. This list contains information about the dataslots - like type, size, name and ID number. The server checks this list of dataslots against its own. If they match up, the server keeps a

list of these dataslots. When new information comes into that dataslot, the network process sends this data across the network to the client. If there are any errors (e.g. the client is requesting a dataslot that does not exist on the server), both the server and client close the network connection and terminate the corresponding threads. A warning is logged to the console. Errors can be generated during setup when the size of the dataslot on the client, is smaller than the size of the dataslot (with the same identification) on the server. In the same situation, but with the data travelling from the client to the server, only a warning would be generated upon the client connecting to the server. If there are no errors, the server and client exchange the data currently in their respective source dataslots to the other side of the network connection (Figure 20). Once the connection is established, which end the client and server is not important. The both behave identically. The distinction between client and server only relates to establishing and closing the link.

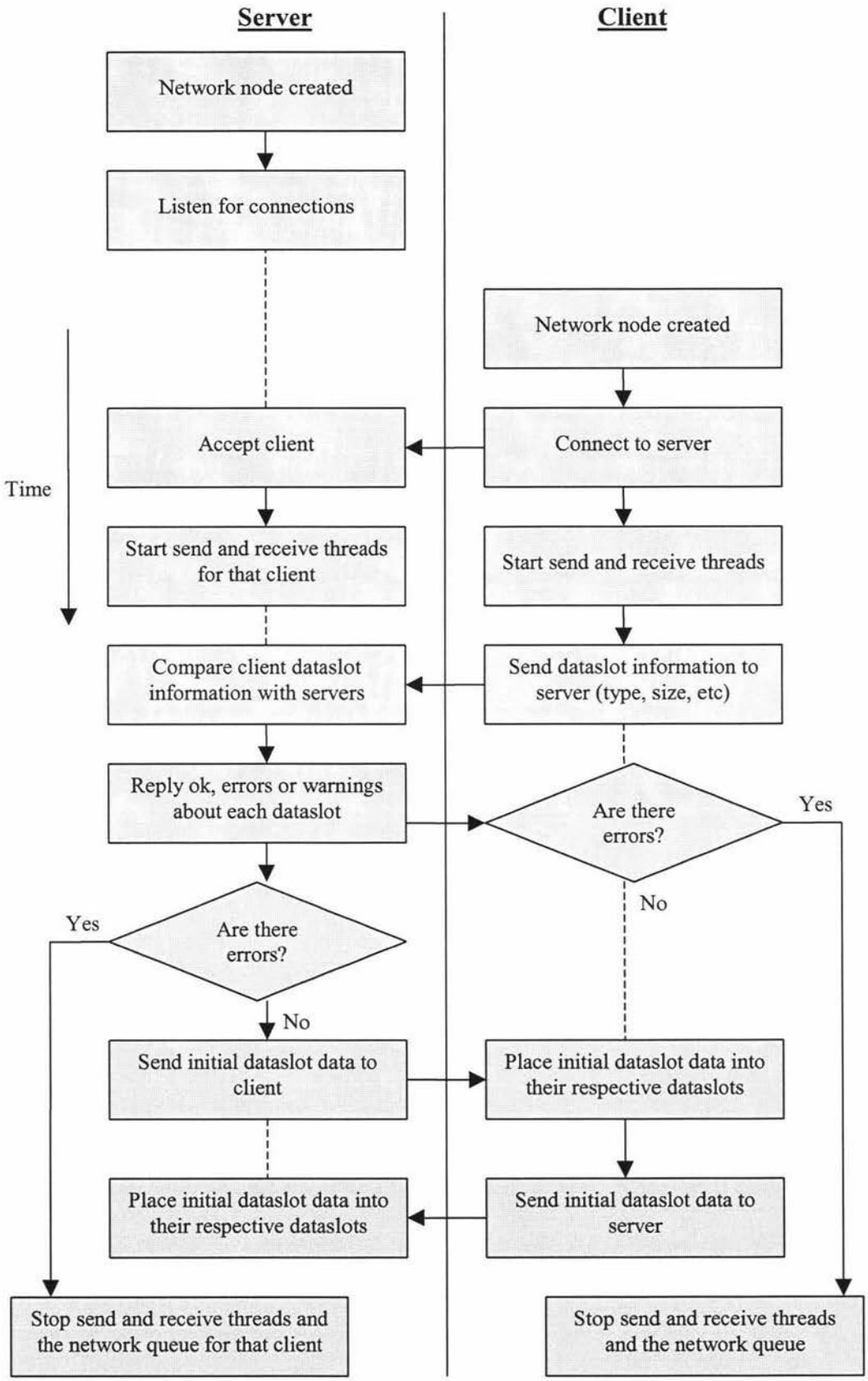


Figure 19: Operations on the server and client end when a client connects to a server.

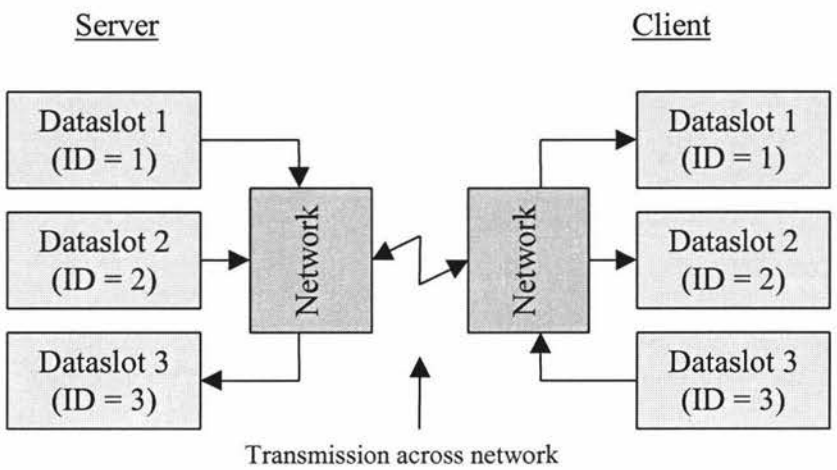


Figure 20: Data transference across a TCP/IP network happens through the network processors. The sending network processor acquires the data from a dataslot, while the receiving network processor places the data in a dataslot which has the same identification.

When the client disconnects from a server, the server automatically detects this. First the network queue for the client on the server is shut down. Then the send and receive threads for the client are closed. When the server disconnects from the network, the client closes its network queue for the server, then closes its send and receive threads.

Information being transmitted over the network is first assembled into packets. This helps the receiving network identify the data being transmitted across the network. The processor can then wait till all the data is received, or place it in a dataslot if all the data has been transmitted across. Network packets can be looked at to reduce the amount of traffic. This includes transmitting numbers in binary, rather than ASCII characters. The different packet structures for data and control messages are shown in Appendix D.

Having a network processor is important for scalability, as it allows complex distributed systems to be built. For example, a remote vehicle can have a simple implementation of the framework (to keep the size and power requirements to a minimum), while it sends data to a powerful PC to do calculations. Or it could allow multiple people to view and control a remote event. Having network communications would allow each person to talk to each other about this event.

4.8 Data Logging

Data logging is capturing and maintaining a record of a dataset over a period of time. This usually involves recording this data to a file, so that the data can be reviewed at a later date. If data is coming into a standard dataslot more quickly than it is processed, some of the data samples will be dropped. This must not happen in a logging application, so buffering is used to gather a group of data to ensure that all samples will be recorded. A buffered dataslot is similar to a normal dataslot but has extra memory. Usually a buffered dataslot consists of two arrays, for input and output. This is to allow data to be written in to one array, while at the same time read from the second array. This ensures data is not being changed while reading it is being processed. Once the system has filled the input array, the arrays are swapped. The system starts writing to the second array (the new input), while the first array (now the output) is then used as the source for writing to a file. These arrays need to be long enough so that the data can be written to a file before the arrays are swapped again. This length is related to how quickly the data is generated by the sensor, and how much load the framework is under.

Figure 21 below shows data being generated from a digital compass attached to the USB port. The direction from the digital compass is placed in the integer direction dataslot. The data logger process takes this direction and places it in the buffered integer dataslot. To ensure that the data in the "Direction" dataslot does not get overwritten by subsequent inputs, it is important that the data logger process runs with a high priority. The reason the USB driver process does not place the direction directly in the logged direction dataslot is this direction is used in further processing elsewhere in the system (shown by the controller processor in Figure 21).

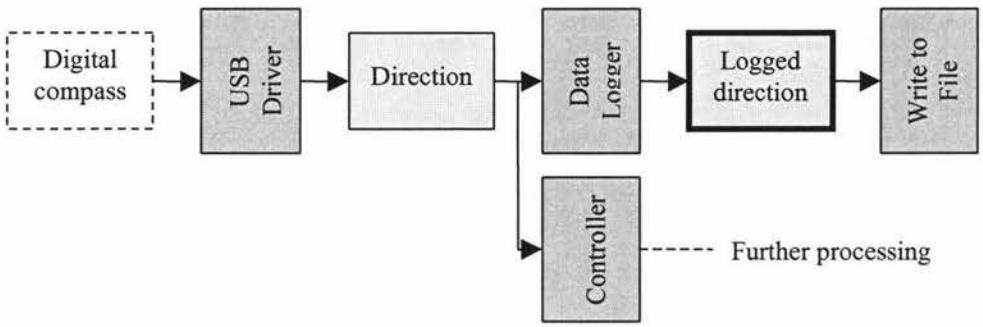


Figure 21: Example data logging application showing a buffered dataslot.

Figure 22 shows how the buffered integer dataslot would be specified in the configuration file. Both the input and output arrays are one integer wide and 30 integers in length.

```
DATA=\buff_int[1]
        NAME="Logged direction"
        BUFF_SIZE=30
```

Figure 22: How to create a buffered char dataslot in the configuration file.

4.9 Image Processing

In image processing, the storage of pictures or frames of a video becomes important. Image dataslots have been created for this purpose.

An image dataslot is a particular buffered dataslot with three buffers. The first buffer is an input buffer use by the process generating the image data. Another is an output buffer which is read by the downstream process. The third buffer is a storage buffer and is required so that when changing from one frame to the next, the input and output do not need to be synchronised. The third buffer allows the output process to work with a whole frame at a time. Without this intermediate buffer, at the end of the frame on the input, the input and output buffers would need to be swapped to allow for subsequent input. However, if the output process has not completed its processing on the frame, part of the image will be processed from one frame, and the remainder from the other.

On static images this does not matter, but on dynamic images (the camera is moving relative to the scene) this could result in erroneous measurements.

With the three buffer system, upon frame completion of the input, if the output is being used, the storage and input buffers are swapped (Table 3). Otherwise the input and output buffers swapped and any downstream processes are queued. On frame completion on the output, and if the storage buffer contains valid data, the output and storage buffers are swapped, again with any listening processes queued.

Condition	Upon finishing the condition IN points to buffer:	1st buffer	2nd buffer	3rd buffer	Upon finishing the condition OUT points to buffer:
Upon start up	1	Empty	Empty	Empty	3
First data set comes in	2	Fill	Empty	Empty	1
Second data set comes in	3	Fill	Fill	Empty	2
Third data set comes in	1	Fill	Fill	Fill	3
Dataslot is read	2	Fill	Fill	Empty	1
Dataslot is read	3	Empty	Fill	Empty	2
Dataslot is read	1	Empty	Empty	Empty	3

Table 3: Table showing what buffers are read from and written to under certain conditions.

Figure 23 shows a FireWire camera connected to the framework. The FireWire driver takes each frame of the camera and places it in the image dataslot. This frame is then displayed on a window using the video display processor.

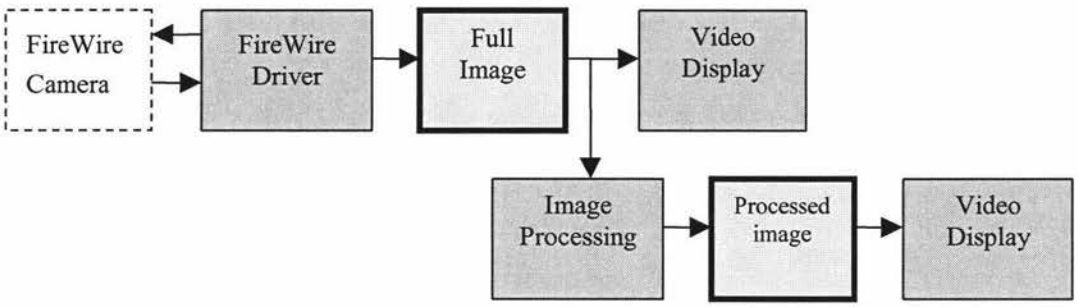


Figure 23: Security system showing frames being captured from a FireWire camera then placed into an image dataslot. The video display processor then displays this frame. The image is also taken and a image processing operation is done on the frame. From there the new image is displayed.

4.10 Dynamically Linked Libraries

Dynamically linked libraries are used to extend the framework for a particular application. There are two reasons why DLLs are useful.

The first use of DLLs is to connect to an external device from within the framework. On PCs, there is no direct access to hardware. However, some vendors provide programming interfaces to their device drivers, allowing an application to access the hardware. These interfaces are usually supplied in the form of DLLs. One example is the Logix4u driver (Logix4u 2006) we have used to access the parallel port.

The second use of DLLs is to add extra processors and dataslots to the framework. Currently new processors and dataslots must be complied with the existing code. This requires recompiling the whole system whenever devices are modified or added. By having new processors and dataslots in a DLL, extra code can be added quickly to the existing system, without affecting the framework. If each process is implemented in a separate DLL, this facilitates their reuse in other applications. The use of DLLs has not been implemented to add extra processors and dataslots at this point, but would be one of the major targets for any future development on the framework.

4.11 Summary

This chapter described some of the typical operations (acquiring the data from sensors, handling multiple dataslots, actuators, controllers, virtual instrumentation, networking, data logging, image processing, timers and dynamically linked libraries) that are used in an integrated sensor and controller framework. It described how these operations fit in with the major components of the framework – dataslots, processors, queues and the configuration of the system. In a sensing and control system these operations are very important. They are commonly used, so they must be well documented and easy to implement.

5 Demonstration Systems

This chapter covers a number of experiments to test the framework. The first two are simple applications to verify that the data passing mechanism between processors is working. The framework is then used in a simple control and data logging application to demonstrate these features of the system. The next application demonstrates how a more data intensive application such as image processing may be implemented. The effect of system load on processing throughput is investigated. Finally the latency of the framework is considered, as this will limit the range of applications for which the system may be applied.

5.1 Simple Button/Indicator

The first application investigates the core of the framework. Processors use the data in the dataslots to achieve their function, while each dataslot queues all the processors that use its data. This application would confirm that of the data passing and queuing parts of the manner in which they were intended.

A button and indicator are created on a window (Figure 24). By pressing the button, the indicator turns on. Pressing the button again turns the button off.

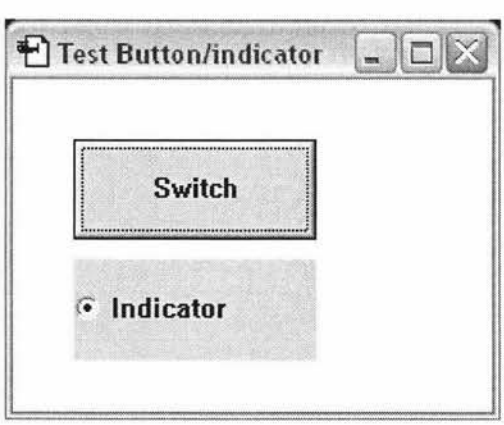


Figure 24: Screen shot of the display while the indicator is on.

The system configuration for this application is shown in Figure 25 with the corresponding configuration file listed in Figure 26. When the button is pressed, the

Activate toggle dataslot is activated by the ***Button*** process. This dataslot queues the ***Toggle*** process to run. The ***Toggle*** process acquires the state of the indicator from the ***Indicator state*** dataslot. This state is then toggled, and written back to the dataslot. The ***Indicator state*** dataslot queues the ***State indicator*** processor. This processor then turns the indicator on or off, depending on the data in its input dataslot. A second press of the button will invert the ***Indicator state*** again, and the currently indicated state on the window will change.

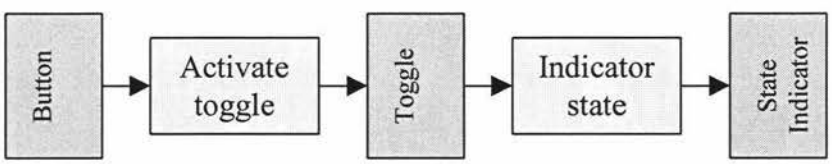


Figure 25: Interaction of dataslots and processors for a simple button/indicator.

The configuration file (Figure 26) shows the four processors (the fourth one is to create the window that holds the button and indicator), and two dataslots. Notice the `INIT` keyword is used to enter the initial data in the ***Indicator state*** dataslot, and other keywords that are used to create all the dataslots and processors.

```
//Configuration file for simple button/indicator

DATA=\char[1]
      NAME="Activate toggle"

DATA=\char[1]
      NAME="Indicator state"
      INIT="0"

PROCESS=\Display
      NAME="WINDOW_1"
      LABEL="Test Button/indicator"
      START_POS = (300,250)
      SIZE=(250,200)

PROCESS=\Button
```

```
      NAME="Button"
      LABEL="Switch"
      START_POS = (30,30)
      DATAOUT ="Activate toggle"
      SIZE=(120,50)

PROCESS=\Radio_Button
      NAME="State indicator"
      LABEL="Indicator"
      START_POS=(30,90)
      DATAIN="Indicator state"
      SIZE=(120,50)

PROCESS=\Toggle
      NAME="Toggle"
      DATAOUT = "Indicator state"
      DATAIN = "Activate toggle"
```

Figure 26: Configuration file for simple button/indicator.

This simple system shows that the basic framework operates in the intended manner. Processors write to, and activate dataslots. The dataslots queue processors, which when run, carry out their operation. In this application the pressing of the button resulted in the indicator switching on and off.

5.2 Networked Button/Indicator

The next application has a second indicator on a separate computer. The purpose of this is to verify that the framework correctly sets up the network connection between two independent computers and correctly synchronises the data between the two systems. This is of importance for scalability as it allows complex distributed systems to be built.

By pressing the button on the server computer (Figure 27), the indicator state should toggle – both on the server and client computers.

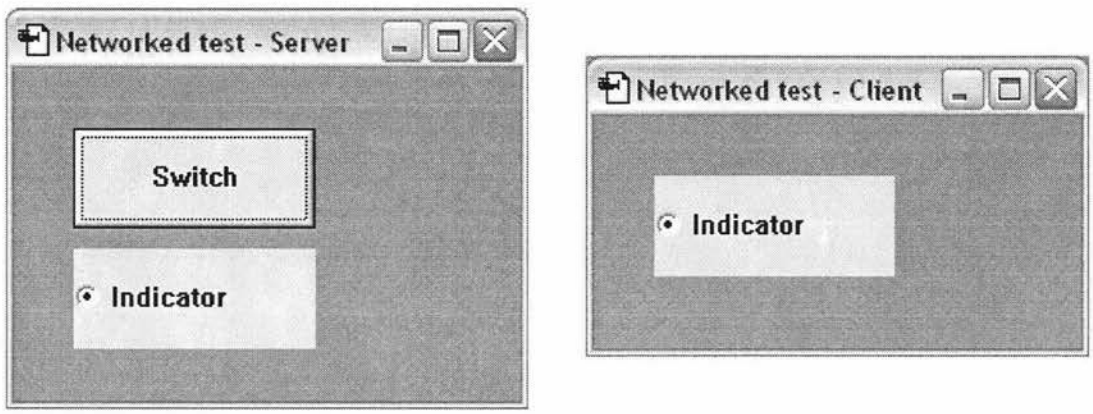


Figure 27: Screen shots of the server and client windows showing the button and indicators. The indicator is on the server to check that the indicators on both computers are either on or off at the same time.

As before, the configuration for this application is illustrated in Figure 28, with the corresponding server and client configuration files listed in Figure 29 and Figure 30. When the button is pressed, the *Activate toggle* dataslot is activated, which in turn queues the *Toggle* process to run. The Toggle process again toggles the state in the *Indicator state* dataslot. This time, however, the *Indicator state* has two dependent processors – the *State indicator* and *Network*. The *State indicator*, when executed behaves as in the previous example. When the *Network* processor runs it takes the data in the *Indicator state* dataslot and transmits it over the network to the client computer. The clients' network processor puts the data in its *Indicator state* dataslot which triggers the clients' *State indicator* processor to run, displaying the current state of the indicator on its window. A second press of the button will invert the indicator state again, resulting in the change of the indicator on both the client and server ends.

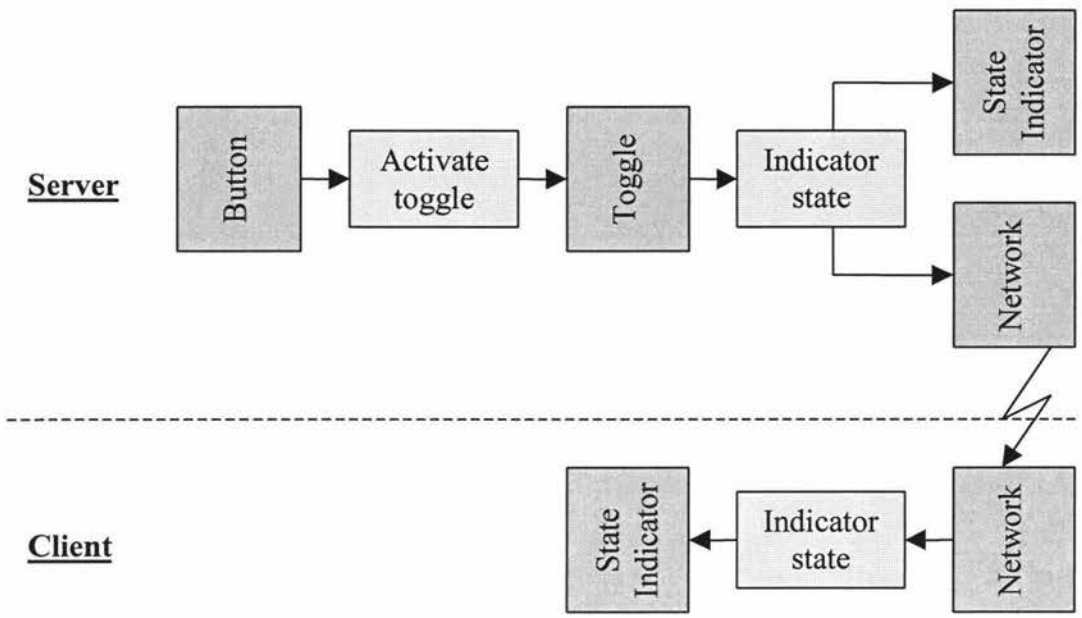


Figure 28: Interaction of dataslots and processors for the networked button and indicator. Both the server and client systems are shown.

In the configuration files in Figure 29 and Figure 30, note the network processors. On the server, the ***Indicator state*** dataslot is listed as an input, whereas on the client, it is an output to the network processor. While the port to be used and the socket type are stated at the server end, the client also specifies the IP address of the server.

```
// Configuration file for the server end of the networked
// button/indicator

DATA=\char[1]
      NAME="Activate toggle"

DATA=\char[1]
      NAME="Indicator state"
      DATAID=1
      INIT="0"

PROCESS=\Display
      NAME="WINDOW_1"
      LABEL="Network test - Server"
      START_POS = (300,250)
```

```
      SIZE=(260,200)

PROCESS=\Button
      NAME="Switch"
      START_POS = (30,30)
      DATAOUT ="Activate toggle"
      SIZE=(120,50)

PROCESS=\Radio_Button
      NAME="State Indicator"
      LABEL="Indicator"
      START_POS=(30,90)
      DATAIN="Indicator state"
      SIZE=(120,50)

PROCESS=\Network
      NAME = "Network"
      DATAIN = "Indicator state"
      SOCKET_TYPE = "server"
      PORT = 4000
      PROTOCOL = "TCP"

PROCESS=\Toggle
      NAME="Toggle"
      DATAOUT = "Indicator state"
      DATAIN = "Activate toggle"
```

Figure 29: The server end of the networked button/indicator application is setup using this configuration file.

```
// Configuration file for the client end of the networked
// button/indicator

DATA=\char[1]
      NAME="Indicator state"
      DATAID=1

PROCESS=\Display
      NAME="WINDOW_1"
```

```
      LABEL="Network test - Client"
      START_POS = (270,150)
      SIZE=(250,150)

PROCESS=\Radio_Button
      NAME="State Indicator"
      LABEL="Indicator"
      START_POS = (30,30)
      DATAIN = "Indicator state"
      SIZE=(120,50)

PROCESS=\Network
      NAME = "Network"
      DATAOUT="Indicator state"
      SOCKET_TYPE = "client"
      PORT = 4000
      PROTOCOL = "TCP"
      SERVER="130.123.82.35"
```

Figure 30: Configuration file used to setup the client side of the networked button/indicator application.

This application shows that framework correctly sets up a network connection and synchronises the data between the two nodes of a distributed system. This application only shows data flow going from the server to the client computer. Once the network connection is established, data can be transferred in both directions.

5.3 Brewery Temperature Controller

A complete sensing and control application is now considered. For this a brewery temperature controller is implemented. During brewing, the brewing vat must be kept within a certain temperature range to ensure optimal conditions for fermentation. Departure from the optimal temperature may result in reduced quality or process yield.

Two temperature sensors are attached to the parallel port via an I^2C bus (Appendix A). One is inside the temperature controlled area and one is outside (Figure 31). The internal temperature sensor is used by the controller to maintain the vat within the

optimum range. The external sensor is not required by the simple controller, but provides additional information for interpreting the results. The controller compares the inside temperature with the allowed range. If the temperature is too hot, the heating element is turned off, and if too cool, the element is turned on (Appendix B). If the temperature is in between the maximum and minimum values, the heater stays on or off depending on which limit was last reached.

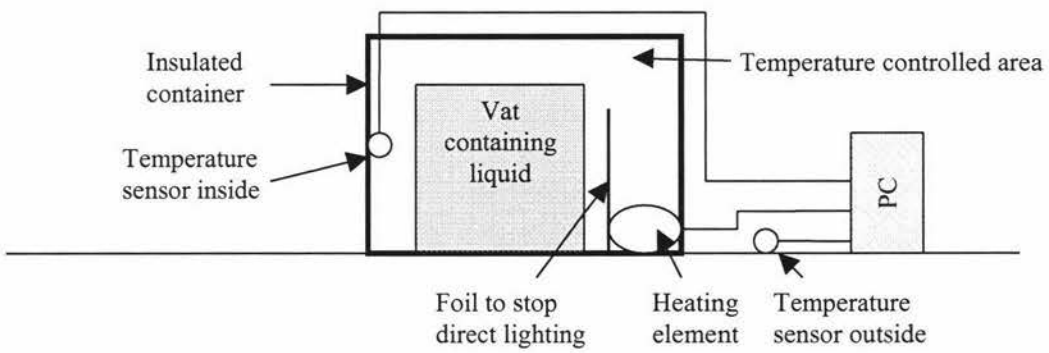


Figure 31: How the brewery temperature controlled system was setup.

The configuration for this application is shown in Figure 32. The I^2C temperature sensors are polled every five seconds. This rate was chosen because it is significantly faster than the thermal time constant of the system (Appendix E), and therefore enables effective control using a simple on/off controller. Temperature sensing is therefore controlled by the ***Timer Process*** which is configured to update the ***Timer*** dataslot every five seconds. This triggers the ***I^2C Driver*** to access the sensor hardware and read the current temperatures, which are placed in the ***Temperature in*** and ***Temperature out*** dataslots. In this application, a simple on/off controller with hysteresis is used. The hysteresis values for the controller are set to 19°C and 20°C, in the ***Desired min temperature*** and ***Desired max temperature*** dataslots respectively. For brewing, the desired temperature range of the liquid is 18°C to 25°C. The hysteresis values are set to 19°C and 20°C because in this application the temperature sensor is located outside the vat and may not represent the actual liquid temperature. Setting the range in this way ensures the temperature of the liquid does not exceed these limits. Each time the ***Temperature in*** is updated, the controller compares it with the desired temperatures limits. If ***Temperature in*** is greater than ***Desired max temperature***, the heater is

turned off, otherwise if ***Temperature in*** is less than ***Desired min temperature*** the heater is turned on. If the temperature is within range, the ***Heater state*** is unaffected. Whenever the ***Heater state*** is updated, the parallel port driver then switches the heater element on or off.

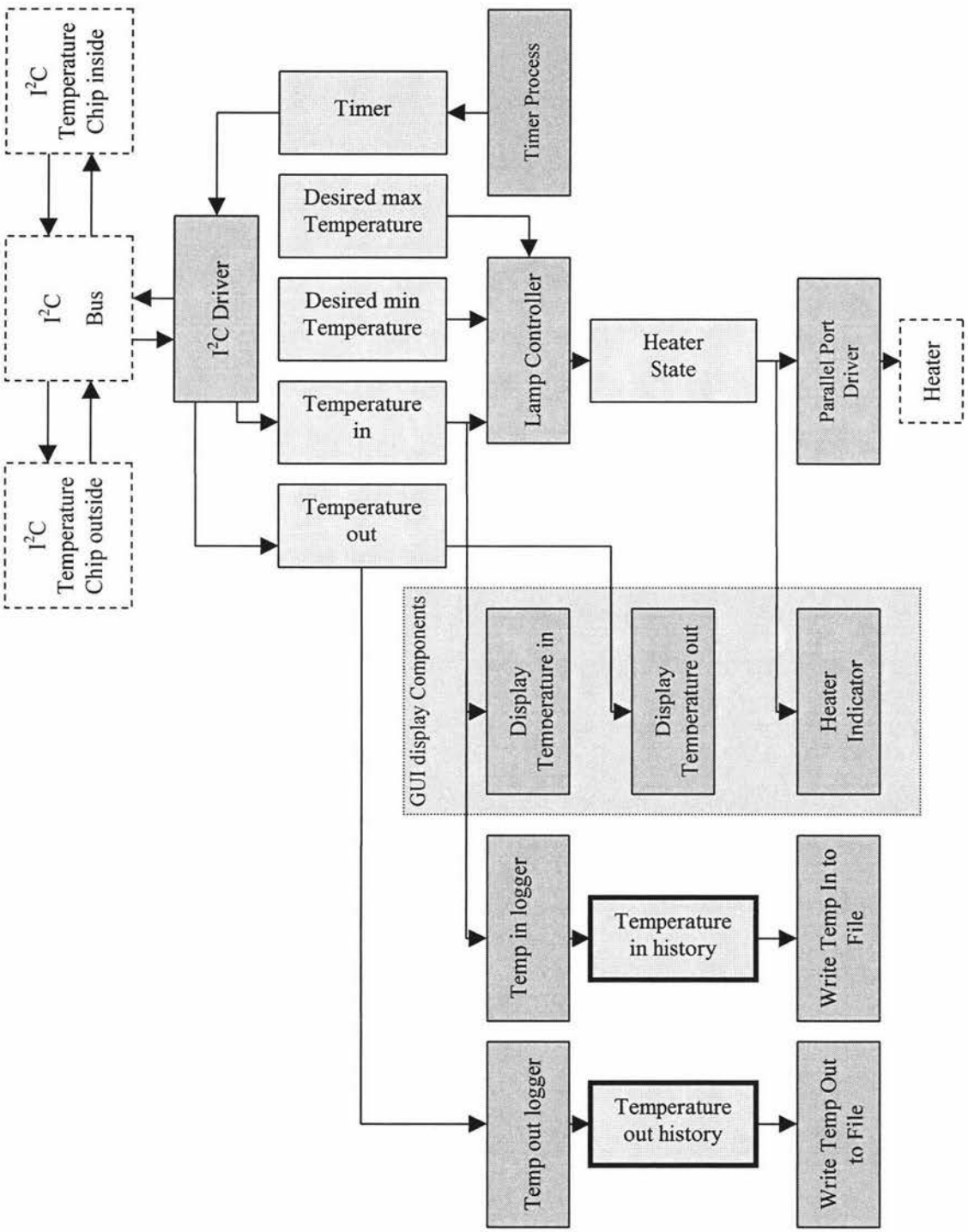


Figure 32: System design for the brewery temperature control application.

The current temperatures are displayed and also logged using the data logging features of the framework. The temperatures are logged to a file so the temperature over time can be checked. This can be useful to find out if the heater is powerful enough for the system, by checking the rate of change of temperature. It also serves as validation that the correct temperature was maintained. No cooling is used in this application as the brew temperature never gets too warm. The configuration file for this application is listed in Appendix F.

Figure 33 shows the screen shot of the window that is displayed. The temperature inside and outside of the controlled area is shown, along with an indicator to show the current state of the heater.

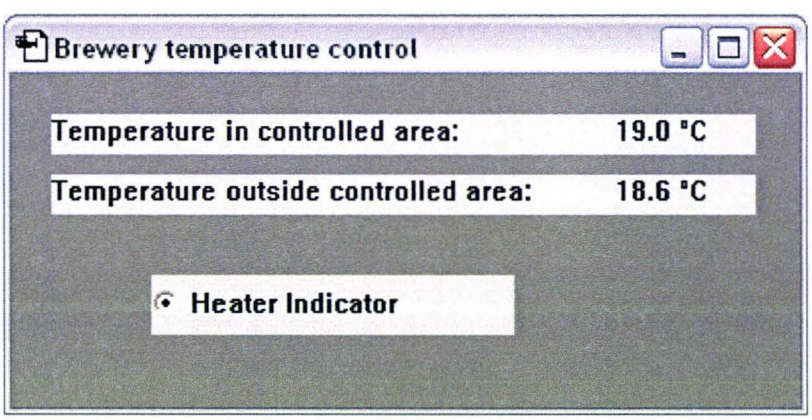


Figure 33: Screen shot of the window used in the brewery temperature control system.

Figure 34 shows the results of this experiment run over two days. The graph shows that the temperature stays within the desired temperature range. The inside temperature line shows the temperature varying, but only just going above 20°C and below 19°C. At the end of the graph the inside temperature stays between the two limits without bouncing between them. This is due to a high external temperature, shown by the temperature outside line.

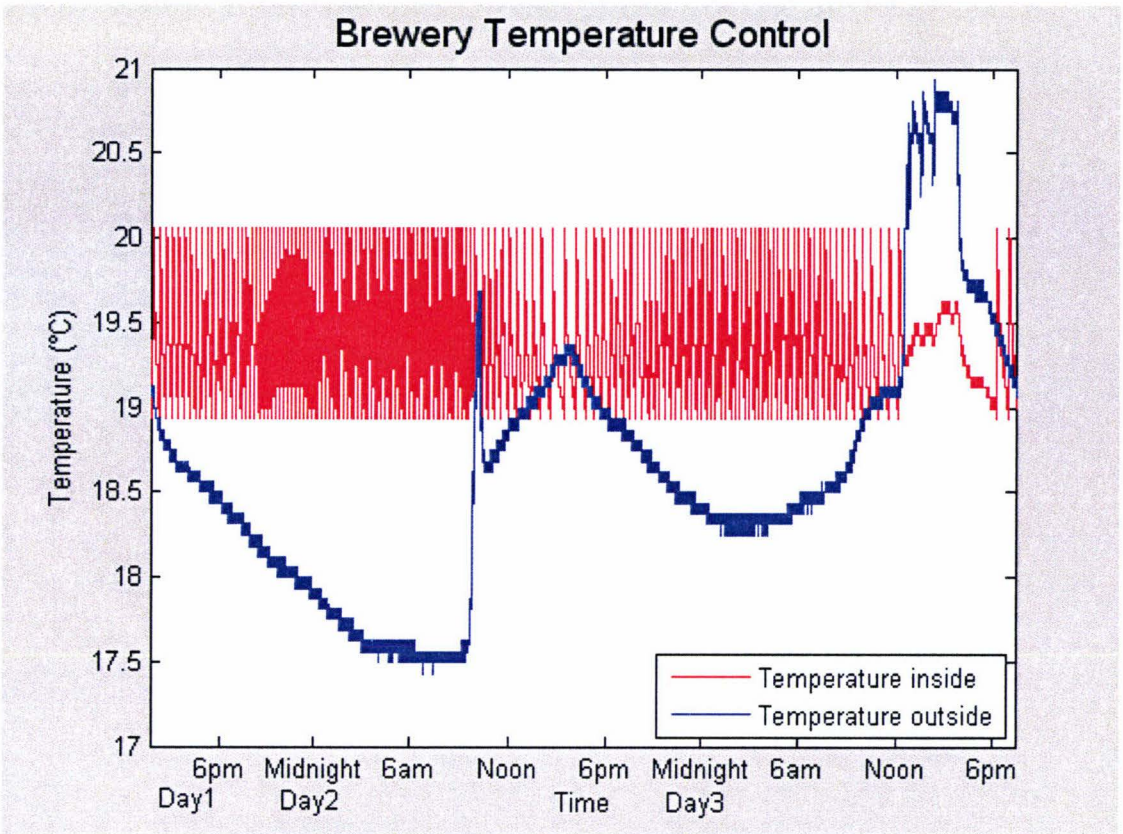


Figure 34: Graph of vat temperature over 2 days.

Figure 35 shows the first hour and a half in more detail. Notice the slope on the inside temperature line when the temperature is increasing. As this line does not start to plateau near 20°C, this indicates that for these range of temperatures, the current heating solution works well. As 20°C is the higher limit, the temperature needs to go above this for the controller to switch the heating unit off. When the temperature falls below 19°C, the controller switches the heating unit back on. There is some jitter in the measurements – especially in the outside temperature values. This is due to quantisation error in the temperature sensors.

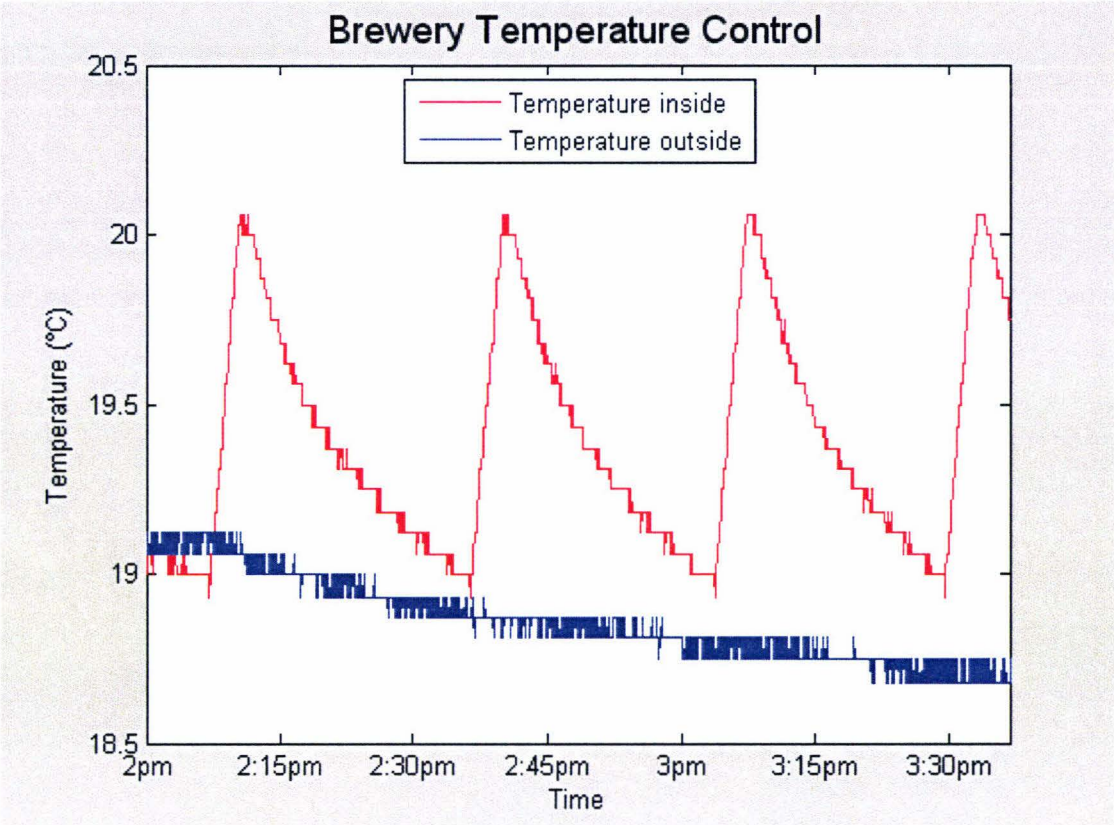


Figure 35: Zoomed in image of the first hour and a half.

This application shows the framework can provide the basis for a complete sensing and controlling platform. It shows data can be acquired, decisions taken, then actions implemented through actuators. This task is only run every five seconds, which is a very light load. Further applications will be used to show that the framework can operate at a much faster speed.

5.4 Image Processing

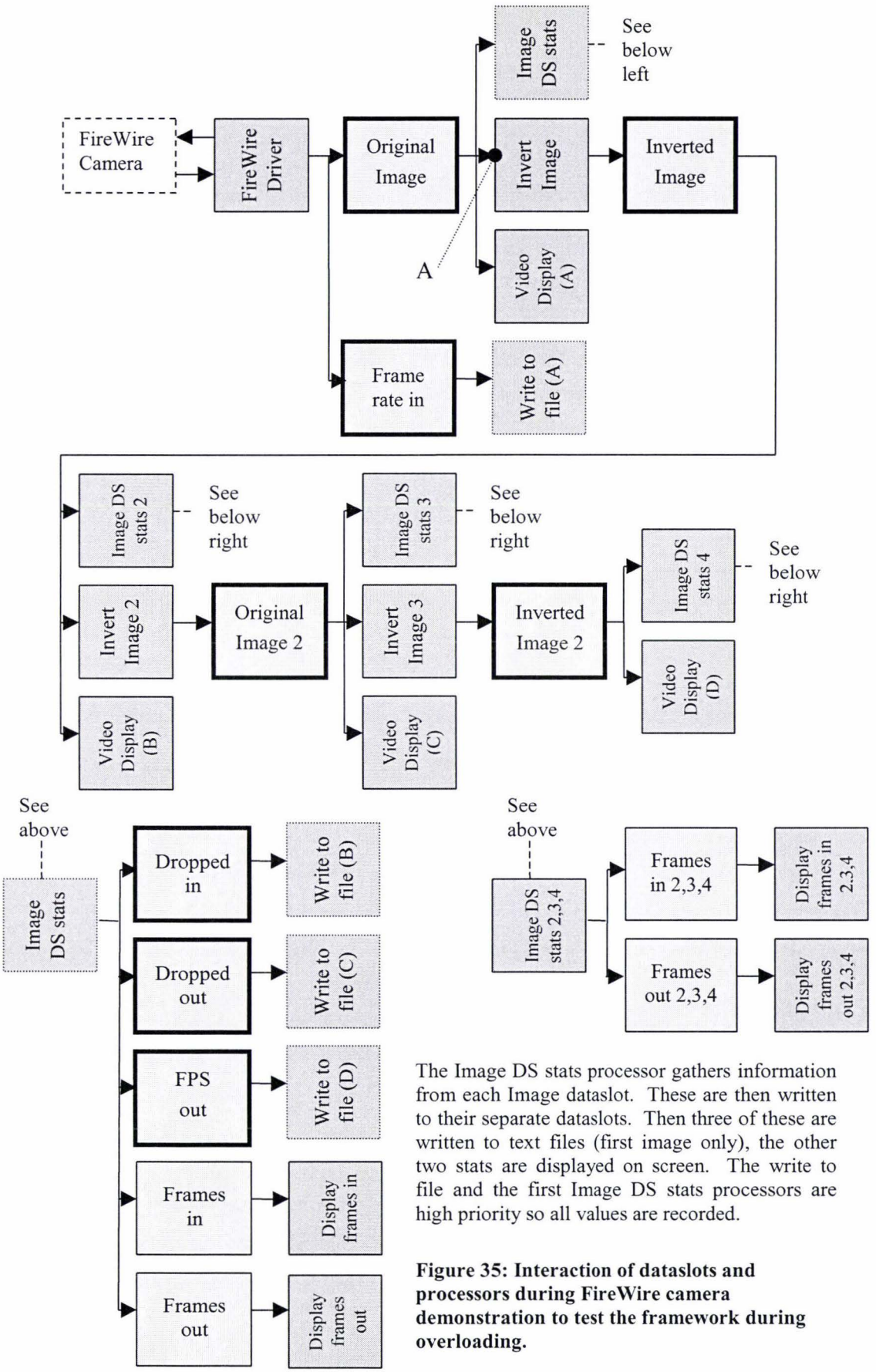
The previous application was not particularly demanding in terms of computational requirements. In this application the system captures video from a FireWire camera. In order to stress the system, several successive steps are used. The results of each stage are also displayed in an image window. The repetitive display of images consumes sufficient system resources that all of the processing steps cannot be completed within a single frame time on the host system. This will confirm that the behaviour of the system degrades gracefully with as many operations as possible being completed. The system should keep overriding the dataslots while the system is under load. Processors

should be queued ready to execute when the load reduces, to run with the latest data available in the dataslots on its inputs.

The demonstration was set up as shown in Figure 36, using the configuration file in Appendix G. The camera used produces 640 by 480 pixel images at 7.5 frames per second. At that resolution, 7.5 frames per second is the highest frame rate for the camera and computer used. There are two modes the camera can be run in. The first is continuous acquisition mode, where the camera continuously provides images to the computer at a rate of 7.5 frames per second. The second mode is a single frame capture mode where the FireWire driver requests a frame, it is read from the camera. In this application the first mode, continuous acquisition, is chosen. This mode will help to load the PC, as frames are constantly being captured from the camera and placed in an image dataslot. However, this is not enough to fully load the system. To achieve that the image is inverted and displayed three more times. In each inversion process, the image is inverted multiple times to create extra load on the system. Frame statistics obtained from each image dataslot are displayed in a window and logged to file for later analysis. These statistics will be used to judge whether the framework meets the requirements of a system that degrades gracefully. With the first 200 samples, processing is stopped each time at point A (Figure 36). This is done by a counter inside the *Invert Image* processor. Between 200 and 350 samples all processors are active, once they are started from the queue. After 350 samples processing is stopped at point A again. This will show what happens to the framework when it is under, and not under, load. Each image dataslot queues three processors to run. One displays the data in a window. Another processor is run to invert the image so the result can be placed into another image dataslot. The last gathers statistics from the dataslot to display and record to a file.

Six performance properties are investigated for the image dataslots: frame rate into and out of the dataslot, the number of frames in, the number of frames out and the number of dropped frames in and out of each image dataslot. The frame rate into an image dataslot is measured by the FireWire driver. The other five statistics are updated when data is written to, or read from an image dataslot. A frame is dropped at the input

whenever the source has to write a frame into a buffer that still has valid data in it – i.e. data that has not been processed yet. This is due to high load on the CPU, so the processors that take the data have not run yet. A frame is dropped at the output whenever a new frame is available but the out processor that was queued hasn't started running yet. The frame rate in is the rate at which the frames are written to the dataslot. The frame rate out of a dataslot is recorded to give an idea of the amount of load on the CPU at any time. The number of frames into and out of each dataslot gives an idea of where the framework is degrading to keep the system working as well as possible.



The Image DS stats processor gathers information from each Image dataslot. These are then written to their separate dataslots. Then three of these are written to text files (first image only), the other two stats are displayed on screen. The write to file and the first Image DS stats processors are high priority so all values are recorded.

Figure 35: Interaction of dataslots and processors during FireWire camera demonstration to test the framework during overloading.

Figure 36 shows a screen capture of the window showing the video from the camera and the inverted videos. Only the first image dataslot has a difference between the number of frames being stored in the dataslot, and the number of frames coming out (shown in the text between the videos in Figure 36). This means frames are being dropped only in the first image dataslot. As such, only the first image dataslot statistics will be examined.

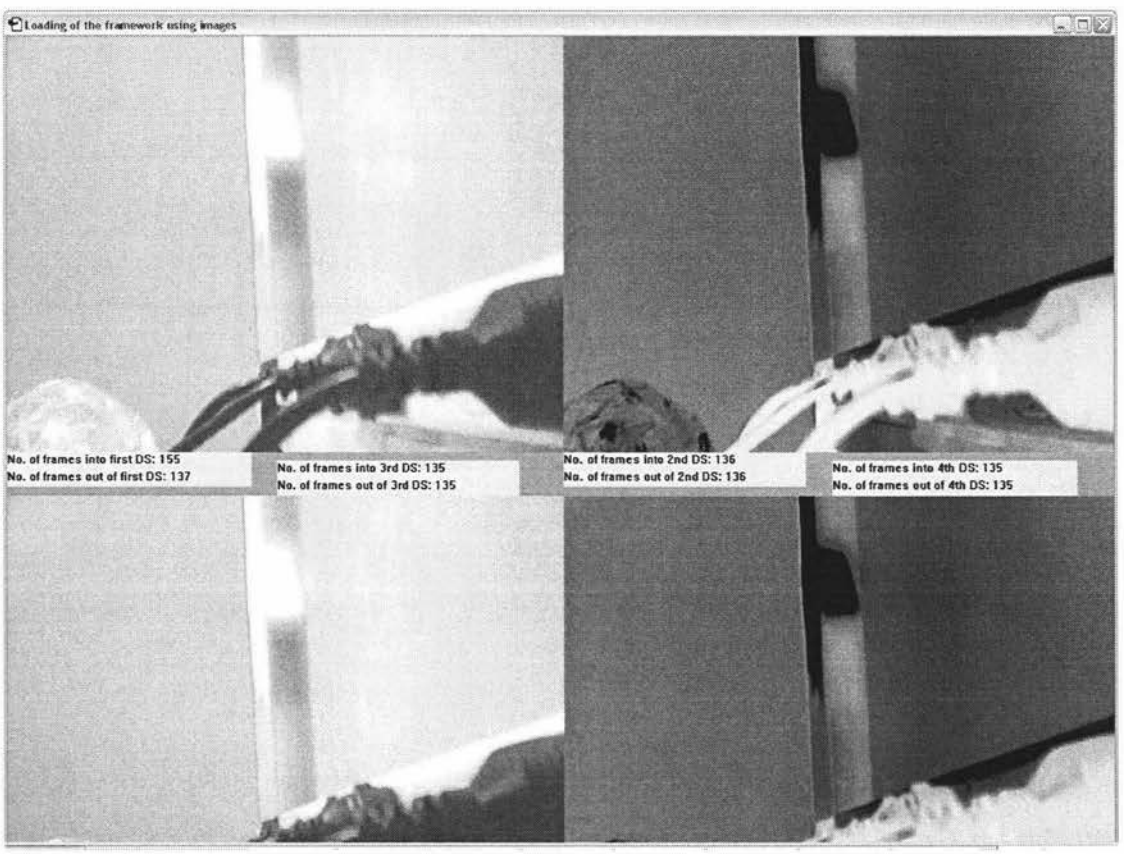


Figure 36: Displaying the FireWire video and colour inverted videos. The video in the top left corner is the video from the FireWire camera. The top right video is the top left video inverted. The bottom left video is the video in the top right inverted. The last video in the bottom right is the video in the bottom left inverted.

Figure 38 shows three graphs. The first shows the frame rate of frames being written to the first image dataslot. The second graph shows the cumulative number of dropped frames in the first image dataslot, both writing and reading from the dataslot. The third graph shows the number of frames dropped inside the camera.

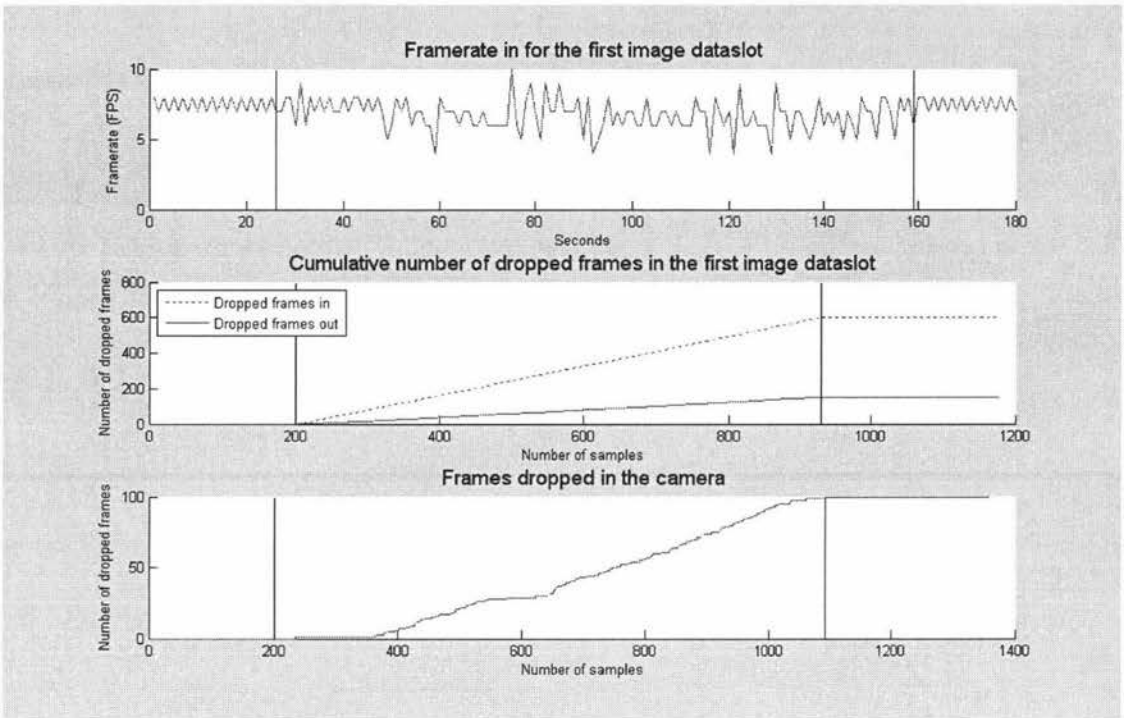


Figure 37: Comparison of the frame rate being stored in the first image dataslot and the frame rate coming out, with the cumulative number of dropped frames in the same dataslot.

The system starts with processing only one image dataslot. The computer is not under significant load, as the frame rate is stays near 7fps. After 200 samples processing is done on all four image dataslots. The frame rate fluctuates, indicating the PC has a problem getting all the tasks completed. After the load is reduced to one image dataslot again, the frame per second stabilises again. While under load the cumulative number of dropped frames for both the input and output increase. The number of frames dropped in the input increases at a much faster rate. This is because the thread (FireWire Camera) putting the data in is running at a high priority, and the thread taking the data out is running at a normal priority. As such, data is not taken out as fast as it is put in, resulting in dropped frames in the input to the image dataslot. When the load reduces, there are no further increases in the number of dropped frames. The number of dropped frames in the camera refers to when the camera has a frame ready to be transmitted across to the PC, but the PC does not access the camera to grab the frame. So then the camera overrides the data with a new frame. The FireWire driver only slows down just before 400 samples, as that is when the camera starts dropping frames.

This shows all the processors get slowed down by the increased processing that is occurring, as even the high priority processor starts to slow down.

This demonstration shows that under load the framework operates as expected. Data gets overridden in their respective dataslots, as shown by the number of dropped frames in the input and output of the first image dataslot in Figure 38. Processors get queued but do not have enough time to run. This is shown by the dataslots being overridden. Once the load drops at 350 samples (passing point A), the processors have time to run, so the dataslots are not being overridden. The frame rate returns to normal.

5.5 Latency

The latency of the framework is important because it is a measure of how fast operations can be done, thus what applications this framework is useful for. Two applications are considered. For both a signal generator is connected to a pin on the parallel port. The application will see a change in the signal, and invert the waveform to another parallel port pin. The first application will invert the result in the parallel port driver. The second will use the framework with the input and output data being stored in dataslots, with a processor used to invert the signal.

5.5.1 Driver only

The purpose of this application is to see how fast operation of the system could happen. It provides a baseline from which to compare the framework against in Chapter 5.5.2. There is a thread running in the parallel port driver (Figure 38). This thread constantly checks the input signal. If it is high, a low signal is written to the output pin. If it is low, a high signal is written.

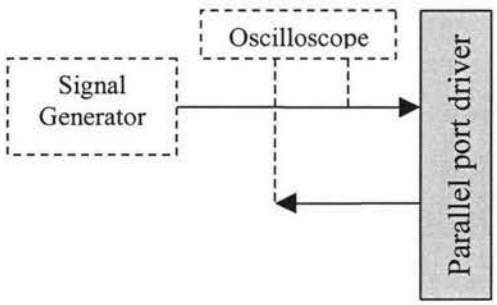


Figure 38: System configuration during testing to see how fast a hard coded example can run.

This system was run for 10 changes of the input signal. The time difference between the input and output values were measured. The average time is 18.8μs.

5.5.2 Driver and framework

This application is to see how long it takes for the system to respond to an input when the framework is used to store and process the data. The driver checks the input. The input signal is stored in the input dataslot (Figure 39). The parallel port thread then suspends. This is because if the thread continuously checked the pin status the total time used by the first task would increase, reducing the time available for the framework to complete the task. The invert dataslot processor takes the parallel port pin status and inverts it. This is stored in the output dataslot. The parallel port thread then wakes up and writes the new data to the particular output pin in the parallel port.

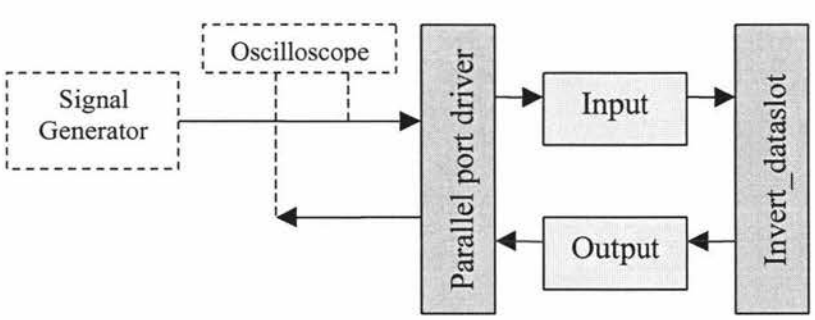


Figure 39: System configuration during testing of latency using the framework.

The system was run 10 times, giving an average time of 68.9µs. This means it takes 3.7 times longer to use the framework for this task than to have it all hard coded. This is to be expected because of the overhead in writing to dataslots and pushing and popping from queues. This is the trade off between speed and flexibility. A slight increase in processing will be made up with the flexibility to work with many devices and a system that can be easily configured. The maximum speed of operation with the framework is 14.5kHz, decreasing with the amount of data that needs to be sent.

To compare this rate with a real sensor, a FireWire camera produces 640 by 480 RGB images. Each frame would be about 1MB in size (including headers and synchronisation communication). Over a FireWire connection (400Mb/s = 50MB/s), up to 50 frames could be sent every second, corresponding to a new data frame arriving every 20ms.

This framework allows higher data rate sensors and actuators. The minimum time the framework can check for an item or block of new data is every 69µs.

5.6 Summary

This chapter shows various aspects of the framework in operation. The reason for these applications is to show that the framework works in the correct manner. The first two applications verify that the data passing mechanisms between processors works. A simple control and data logging application is then used to demonstrate this. The next application was image processing. This showed the effect of high system load on the framework. Finally the latency of the framework is measured to see what applications this framework could be used for.

6 Discussion

This chapter discusses how well the framework achieves its objectives – to create a flexible, reusable, reconfigurable and scaleable integrated sensing and controlling framework. First other applications are discussed. This includes implementation on various hardware platforms, and other software applications are discussed. Future work to be done and to improve the framework is also covered in this chapter. Finally, the discussion centres on how well the objectives of the framework were achieved.

6.1 Other Applications

Running this framework on other hardware platforms is discussed. Also two further examples of applications that can use the framework are shown. These show the complexity and breadth of sensing and controlling applications that could use the framework.

6.1.1 Hardware

A natural migration of the framework would involve implementing the design on a microcontroller. In applications that involve only a few sensors and actuators, especially remote platforms, having a small, low power implementation of the framework would be beneficial.

Due to the limited resources memory available on a microcontroller, the implementation framework would need to be adapted, however operating principles of the framework would be the same. For example the queue could consist of one or two bytes, associating each bit in the queue word with a single processor. When a bit in queue is set, it would indicate that the corresponding processor is queued. Dataslots are more likely to have preset memory locations. It is unlikely that separate threads are used for different queues, although this could be developed if necessary. Rather than the framework being reconfigurable from a file, the configuration is more likely to be determined at compile time. However, by maintaining the principle of keeping data and processing separate, many of the advantages of this framework could be maintained even on a small microcontroller.

6.1.2 Robot Soccer

One application of this framework is controlling a game of robot soccer (Bailey and Sen Gupta 2004). For the 3-aside game, the playing field is 2m by 1.5m, with a single camera above the field looking down. Each team has three soccer 'men', identified by the various colours and patterns on top of each robot. Each team has a computer as the controller, using the camera as input to sense the robot positions, and an RF transmitter so send movement instructions to each robot.

Figure 40 shows the proposed control structure. A FireWire camera continuously captures an image of the soccer field. During initialization, the image is processed to extract calibration data that enables the robot position data to be corrected for lens distortion, perspective distortion, and parallax (Bailey and Sen Gupta 2004). During game play, the *Robot Location* process locates each robot from the captured image. These raw positions are then corrected for the various distortions to give the true location in field coordinates. The *Role Assignment* process determines the role of each of the controlled robots based on the selected game strategy. A separate *Path Planner* process is instantiated for each robot, which uses its role information and the locations of the other robots to determine where it should be positioned. The derived control data is transmitted to the robots via an RF transmitter attached to the serial port.

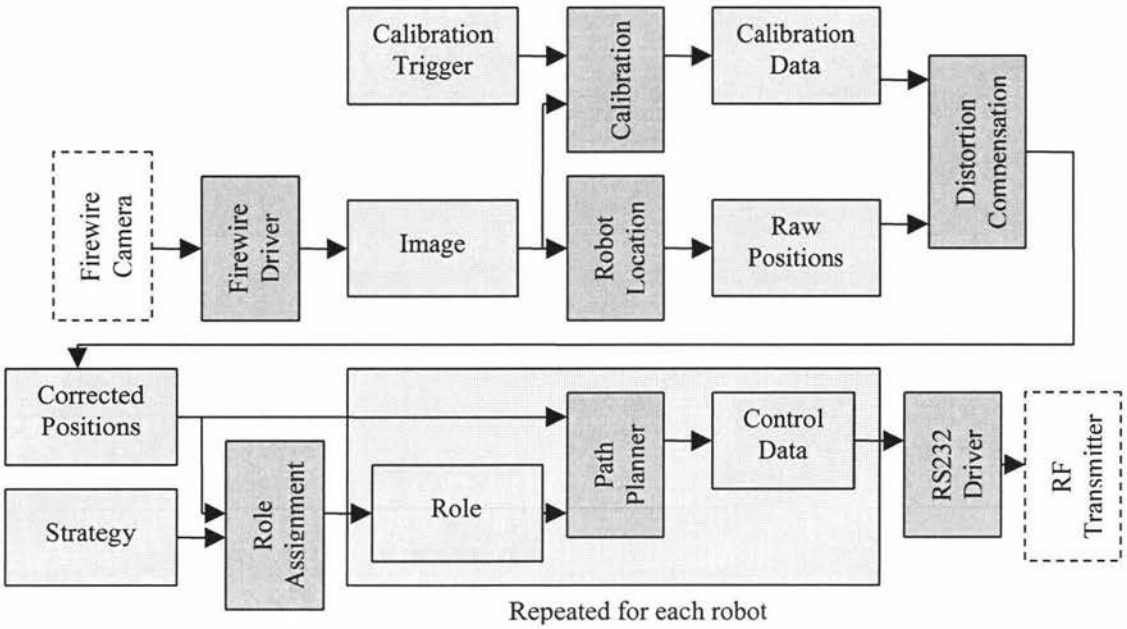


Figure 40: Control flow for the robot soccer application (for clarity, the user interface components have been omitted).

This structure demonstrates several features of the framework. The controller is split into a two level hierarchy (Sen Gupta et al. 2004) by separating the role assignment and path planning. Path planning could also be further divided hierarchically. The higher level determines the target location for a particular robot while the lower level determines the speed profile the robot will use. The robot soccer application requires that the complete control loop be executed for each image that is captured, typically 60 times per second.

This application shows another example of where the framework could be used. The modular design would be very beneficial by enabling experimentation with different strategies or techniques for each of the steps of the complete process. For example, when a better path placement algorithm is devised, this could be swapped in place of the current algorithm very quickly without impacting on the rest of the programme.

6.1.3 Home Automation

Home automation is another example of how the framework can be used in a different application. Home automation is about automating daily tasks around the home. Light

control, control of doors and blinds, climate control, security and surveillance are some of the tasks that can be automated.

Figure 41 shows how a PC running the framework would connect to various hardware devices that it controls. Connection could be through an standard home automation interface (e.g. X10), a standard bus (e.g. I^2C), or use a control signal from the PC to switch one or more relays, which in turn, switches higher voltage devices. Processors and dataslots are not shown as these would be implementation dependent.

Using the framework would make integrating home automation devices easier as all of these devices can be controlled from the one location, without buying new devices. Some modern home automation equipment requires all the devices to talk the same protocol. This limits the number of devices that can be used, without spending money to ensure communication compatibility. Having all the devices' communication functions available in one location improves the user interaction and satisfaction.

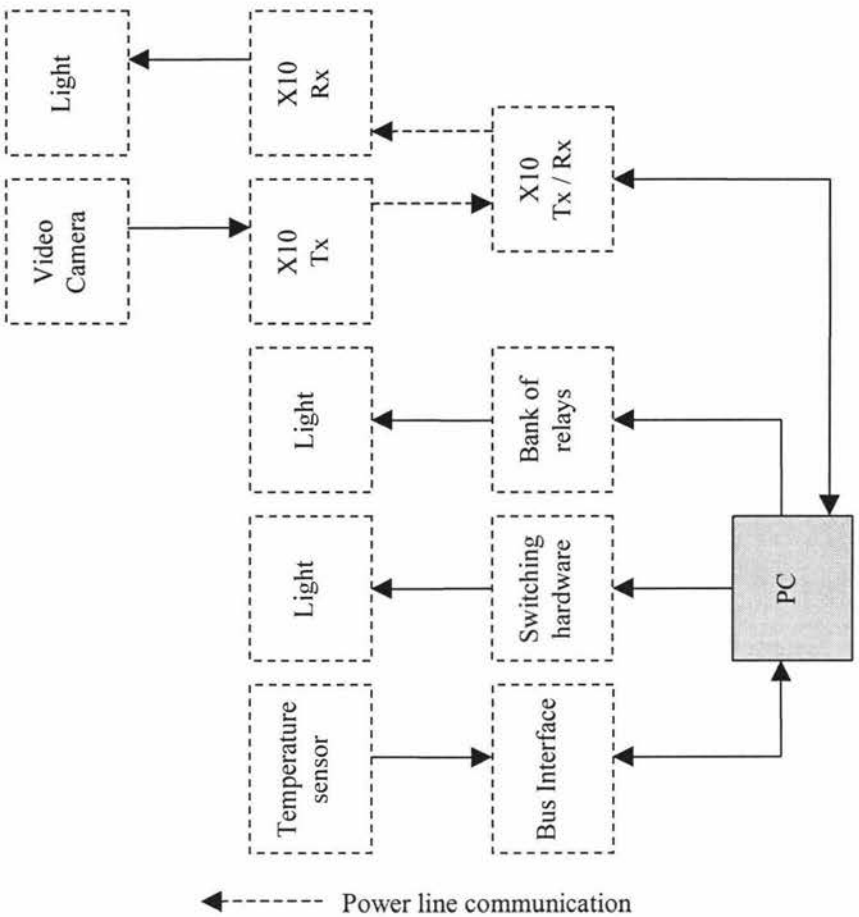


Figure 41: How a PC running the framework would connect to devices it controls.

6.2 Future Work

A number of steps can be taken to further improve this framework. Currently all dataslot and processor code needs to be compiled in the main executable. To make the framework more flexible, the framework could be changed to allow dataslots and processors defined in DLLs. To link the DLL to the main program, the DLL would be defined in the configuration file enabling it to be loaded during the configuration process. This would remove the need to recompile the base code after adding new code for a new sensor or actuator. In contrast, a traditional controller would require significant rework and recompilation even for a relatively minor change.

Currently the configuration of each system that uses the framework is done by listing all the components in a text file. The usability of the configuration would improve by

making the configuration a graphical one. Connecting boxes would be a friendlier environment than listing out a text file. Warnings about different data types could be indicated by a red line – indicating an error. It is noted this is similar to the software program LabView, which provides a flexible framework for integrating sensing and control. However, there is a need for a low cost generic platform. This is where this framework would be beneficial.

With the current implementation of the framework, only a few window items are available (text boxes, buttons, sliders, image displays and indicators). For a fully operational system, other items would likely also be required. This could include a range of various graphs and dials, thermometer type indicators, and so on.

The network processor can be extended to transmit images and video more efficiently by including compression and rate control. This would allow remote viewing, which can be very important for remote vehicles. A chat facility between computers running the framework would also prove very useful. Consider flood monitoring. An integrated chat facility could allow people from Civil Defence to suggest places to the remote operator of a flying remote platform to view.

The examples stated here are improvements that can be made to the framework. However, as the current implementation of the framework is just a proof of concept, these features did not need to be added at this stage.

6.3 Conclusion

This thesis has covered a data centric framework that integrates sensors, actuators, instrumentation and controllers. A key point to the framework is separating the control of the framework amongst the processors, which communicate via the dataslots. This makes the framework more flexible, as there is no central point of contention.

Another design feature is making the framework modular. This separates the interactions between components, which allows faster configuration changes to be made

as there are few dependant links between the modules. It also allows for easier debugging and code reuse.

Components are created and linked together, by defining them in the configuration file. Changes can be made to the system merely by modifying the configuration file. Thus components can be added or removed without recompiling the code. A text file is used for the configuration instead of a graphical environment to simplify the development process, and to reduce development time. For larger, more complex systems, a graphical configuration environment would be preferred.

A hierarchical class structure ensures code reusability through inheritance. New components may inherit properties from established classes and with the required functionality added to the derived class. This reduces the amount of code in the framework, and makes it simpler to add new classes as they only need a few methods as they can inherit the rest.

One potential limitation of the approach of separating the computational components is an increased overhead or latency. It is acknowledged that this is an important consideration in many control applications, so the framework is designed to minimise the latency through implementation of prioritized queues and multitasking. This prevents one slow component from degrading the performance of the rest of the system.

A multi-threaded program architecture ensures task exclusivity, high data throughput, prevents system delays and allows a graceful degradation in performance if the system is overloaded. This prevents the problem in single threaded programs where one processor can take all the CPU time, resulting in other processors not running.

The framework is flexible. It can handle applications as diverse as controlling a game of robot soccer, to remote farm, or habitat monitoring and surveillance. It is reusable; the framework is only designed once, and then each application can use it to implement the desired system. It is reconfigurable by using a different configuration file. By doing this the whole operation of the sensing and controlling system can change. The

framework is scalable – both in the design and computational requirements. Having 20 items on a window works just as well as having only a few. The computational requirements of the system scales with the number of sensors and actuators there are.

This framework is implemented on a PC due to the flexibility of the PC platform, but it could be implemented on a range of hardware devices.

The demonstrations show the framework works as it is described. The data passing mechanism between the processors ensures that the data reaches its desired location. The framework is shown under high system load. The dataslots get overridden under this load, ensuring the latest data is presented to the processors. The processors get queued, making sure no operation gets lost. Thus when the load drops off, the processors start running with the latest data.

Overall, this thesis presents a flexible, useable, low cost system to integrate sensors, actuators, instrumentation and controllers.

7 Authors Publications

Ryan D. Weir, G. Sen Gupta and Donald G. Bailey, "Software Architecture to Integrate Sensors and Controllers", Proceedings of the 1st International Conference on Sensing Technology, ICST 2005, Palmerston North, New Zealand, Nov 21-23, 2005, ISBN 0-473-10504-7, pp. 505-510.

The paper below has been accepted and will be published in 2007:
Weir, R., G. S. Gupta and D. Bailey (2007). "A Scalable Architecture to Integrate Sensors and Controllers." International Journal of Intelligent Systems Technologies and Applications (IJISTA).

8 Appendices

Appendix A

Figure 42 shows the I^2C interface schematic. A hex inverter chip, the SN74LS05N, is used to interface the I^2C bus to the parallel port. Parallel port communication happens through an open source driver (Logix4u 2006). Two gates of the inverter chip are used to switch an LED used for testing purposes. The SCL and SDA lines are normally high. They are pulled high through two pull up resistors: R6 and R7.

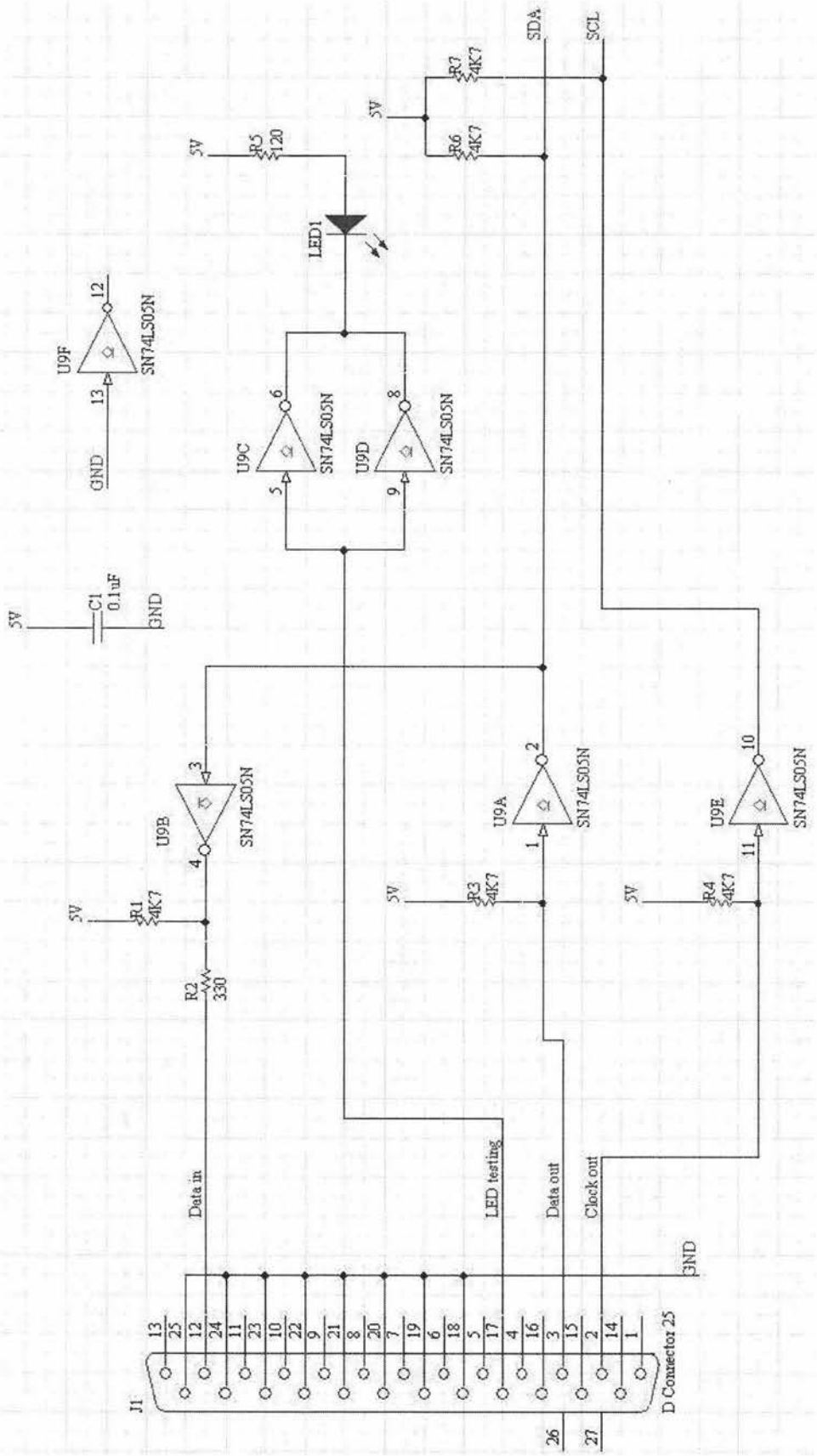


Figure 42: Schematic to connect the I²C bus to the parallel port.

Appendix B

Figure 43 shows the schematic of the circuit used to connect the heating unit to the parallel port. Signals from the parallel port are either 5V or 0V. When the pin on the parallel port is high, Q1 turns on. This will pull the gate of Q2 low, turning it off. This will result in the heating unit being off as well. When the parallel port pin is low, Q1 is off. This will result Q2 turning on, turning the heating unit on.

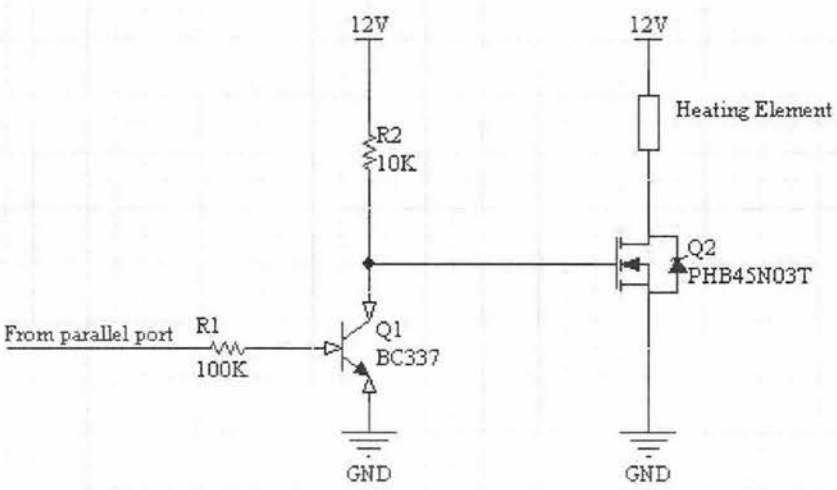


Figure 43: Schematic of the circuit used to connect the heating element to the parallel port.

The resistor values and device limits have been calculated as follows:

Heating unit specifications: 60W at 12V. First calculate the current that the MOSFET must switch.

$$I = \frac{P}{V} = \frac{60W}{12V} = 5A$$

The MOSFET can handle 45A (Philips Semiconductors 1997), so 5A is well within limits. Now calculate the resistor values. First is R2. The capacitance in the input capacitance of the MOSFET: 740pF. The cut off frequency is chosen to be 22KHz. This frequency was chosen to give a good size resistor value as the frequency that the MOSFET switches on or off is not too important as in normal operation this only happens once every 30 minutes to 2 hours or so.

$$\tau = RC = \frac{1}{2\pi F_c}$$

$$R = \frac{1}{2\pi C F_c}$$

$$R = \frac{1}{2\pi * 740e^{-12} * 22000}$$

$$R = 9776.1\Omega$$

This is the calculated value for R2. However, using a real 10KΩ resistor, the cut frequency is now

$$F_c = \frac{1}{2\pi RC}$$

$$F_c = \frac{1}{2\pi * 9776.1 * 740e^{-12}}$$

$$F_c = 21.507KHz$$

The power dissipation of the MOSFET by switching is negligible, due to how infrequent switching occurs, and the current being used. The power dissipation when the MOSFET is on is:

$$P = V * I = RI * I = 5^2 * 0.024 = 0.6W$$

This is within limits stated in the datasheet (Philips Semiconductors 1997). The purpose of R1 is to limit the current into the transistor. First the load current on is worked out.

$$I_c = \frac{Vs}{R_L}$$

$$I_c = \frac{12}{10000}$$

$$I_c = 0.0012A$$

From here the base resistor can be calculated. The h_{FE} value used is 100, and V_c refers to the maximum voltage on the input to the transistor.

$$R_b = \frac{Vc - Vbe}{Ib} = \frac{(Vc - Vbe) * hfe}{Ic} = \frac{(5 - 0.7) * 100}{0.0012} = 358K\Omega$$

The real world value used is 100KΩ. This ensures that the transistor goes into saturation.

Figure 44: Calculating resistor values and checking that parameters stay within devices limits.

Appendix C

This appendix divides a GPS string (Figure 45) into it components (Table 4). The NMEA standard defines many formats, some include: GGA, GLL, GSA, GSV, MSS, RMC, VTG and ZDA (Navman 2004). These formats produce different outputs, depending on what data is required.

$GPGGA,092204.999,4250.5589,S,14718.5084,E,1,04,24.4,19.7,M,,,,0000*1F

Figure 45: A string from a GPS device in NMEA GGA format. This string is from (TELETYPE GPS 2007).

FIELD	EXAMPLE	COMMENTS
Sentence ID	$GPGGA	GPS receiver, GPS fixed data
UTC Time	092204.999	hhmmss.sss
Latitude	4250.5589	ddmm.mmmm
N/S Indicator	S	N = North, S = South
Longitude	14718.5084	dddmm.mmmm
E/W Indicator	E	E = East, W = West
Position Fix	1	0 = Invalid, 1 = Valid SPS, 2 = Valid DGPS, 3 = Valid PPS
Satellites Used	04	Satellites being used (0-12)
HDOP	24.4	Horizontal dilution of precision
Altitude	19.7	Altitude in meters according to WGS-84 ellipsoid
Altitude Units	M	M = Meters
Geoid Separation		Geoid separation in meters according to WGS-84 ellipsoid
Separation Units		M = Meters
DGPS Age		Age of DGPS data in seconds
DGPS Station ID	0000	
Checksum	*1F	
Terminator	CR/LF	

Table 4: Example GPS string broken down into its components. This table shows the components of a GGA string. This example was taken from (TELETYPE GPS 2007).

Appendix D

This appendix describes the network packet structures. There are two types of packets: those are used to transfer data between dataslots (standard packets) (Figure 46), and those used to initially set up the network connection between the client and server (network setup packets).

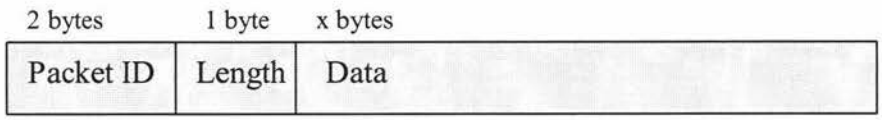
The first item in any packet is the packet ID. This takes up two bytes, allowing up to 65,536 dataslots. The ID of 0 is reserved for setting up of the network. The next field is the length. This specifies the length from the end of this field to the end of the packet. It is one byte long, so the data can be up to 255 characters long. The rest of the packet if it is a standard packet is just data. The position field in an image dataslot relates the data to where in the frame the data is positioned.

The link registration packet is sent from the client to the server during configuration. Its purpose is to send information about its dataslots to the server, so the server can compare the dataslots parameters with its own. The fields of the link registration packet list information about the dataslot to be transmitted.

The link acknowledgement packet is used to send warnings, errors or an ok signal for each link registration packet. It is sent from the server to the client. An ok is sent of the dataslot parameters match at both ends. A warning is sent if the data does not quite match, but it matches enough for the dataslots at each end to be used. A error is sent if the parameters of the dataslots do not match.

The network finish packet is sent by the client when it has finished sending all the link registration packets. It tells the server that setup is now finished; all packets received from this point contain actual data. The server can now send data to the client. At this point the connection can be thought of as a sender and a receiver, rather than a server and a client.

Standard Packets:



Standard packet layout

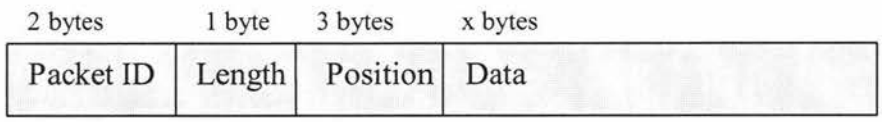
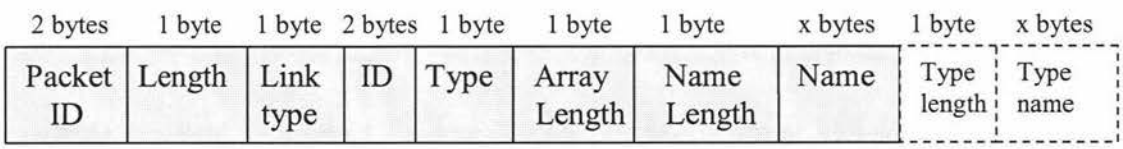


Image packet layout

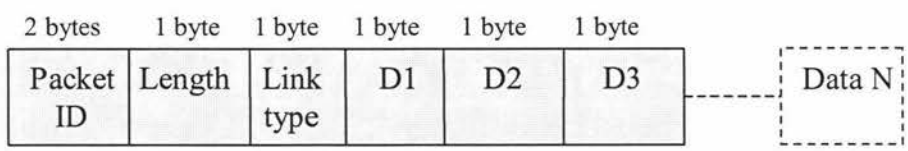
Network Setup Packets:



Link Registration

Link Type:
0 - Link registration
1 - Link acknowledgement
2 - Network finish packet

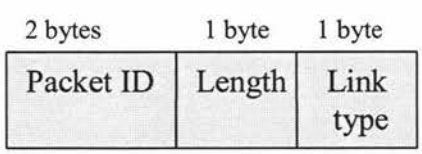
If type is U, user defined, so packet has extra fields.
As shown in the diagram as dotted line fields.



Link Acknowledgement

	ID	Type	Size
DATAID 1	D1	D2	D3
DATAID 2	D4	D5	D6
DATAID 3	D7	D8	D9
DATAID 4	D10	D11	D12

Table showing where information from the link acknowledgment packet is obtained from.



Network finish packet layout

Figure 46: Packet structures being transmitted over the network.

Appendix E

This application was initially run to determine the time constant of brewery temperature control system. The results from Figure 47 are from the temperature sensor located next to the vat. The time constant is the time it takes to get to 63% of the total value. From 18°C to 25°C, the 63% mark would be close to 22.4°C (allowing for the temperature to go slightly over and under the temperature limits). This would represent about a 25 minute time constant.

The cooling time constant is calculated using 63% of the decrease from 25°C to 15°C (ambient temperature). This would be a temperature close to 18.7°C. This would be a time constant of about 40 minutes.

A general rule to simplify the controller and avoid excessive overshoot with an on/off controller is to have least 10 times the number of samples within the time constant. This corresponds to a sample rate of at least once every 2.5 minutes (using the smallest time constant). The temperature is sampled every five seconds. This is to ensure there are enough points to draw the graph accurately.

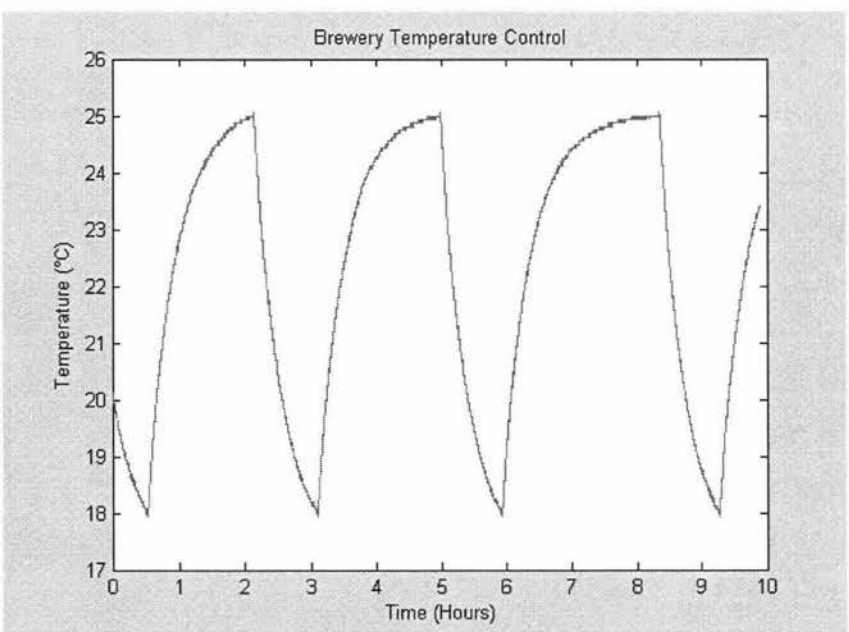


Figure 47: Graph of temperature next to vat over 10 hours.

Appendix F

Configuration file for the brewery temperature controller (Figure 48). Here the I2C driver has been split into its individual processor components: temp_sensor, I^2C and parallel processors.

```
//Configuration file for the brewery temperature controller

DATA=\int[1]
      NAME="Timer"
      TIME=5

DATA=\int[1]
      NAME="Desired max Temperature"
      INIT=20

DATA=\int[1]
      NAME="Desired min Temperature"
      INIT=19

DATA=\float[1]
      NAME="Temperature in"

DATA=\float[1]
      NAME="Temperature out"

DATA=\char[1]
      NAME="Heater State"
      INIT="0"

DATA=\buff_float[1]
      NAME="Temperature in history"
      BUFF_SIZE_FLOAT=20

DATA=\buff_float[1]
      NAME="Temperature out history"
      BUFF_SIZE_FLOAT=20

PROCESS=\Display
```

```
      NAME="WINDOW_1"
      LABEL="Brewery temperature control"
      START_POS = (300,200)
      SIZE=(400,200)

PROCESS=\Parallel
      NAME="Parallel"
      LPT=1

PROCESS=\I2C
      NAME="I2C"
      Bus = 400

PROCESS=\Temp_Sensor
      NAME="Temp_Sensor"
      I2C_ADDRESS=150   //with the read/write bit set to write (0)
      Reg_ADDRESS=170 // Register of the temp
      Command=81
      Delay=1000  // in ms
      PRECISION=6
      DATAIN="Timer"
      DATAOUT="Temperature in"

PROCESS=\Temp_Sensor
      NAME="Temp_Sensor"
      I2C_ADDRESS=146   //with the read/write bit set to write (0)
      Reg_ADDRESS=170 // Register of the temp
      Command=81
      Delay=1200  // in ms
      PRECISION=6
      DATAIN="Timer"
      DATAOUT="Temperature out"

PROCESS=\Hardware_Lamp
      NAME="Lamp"
      DATAIN="Heater State"

PROCESS=\Lamp_Controller
      NAME="Lamp Controller"
```

```
      DATAIN#temp="Temperature in"
      DATAIN#max="Desired max Temperature"
      DATAIN#min="Desired min Temperature"
      DATAOUT="Heater State"

PROCESS=\Text
      NAME="Display Temperature in"
      LABEL="Temperature in controlled area:          %.1f °C"
      DATAIN="Temperature in"
      START_POS = (20,20)
      SIZE=(350,20)

PROCESS=\Text
      NAME="Display Temperature out"
      LABEL="Temperature outside controlled area:     %.1f °C"
      DATAIN="Temperature out"
      START_POS = (20,50)
      SIZE=(350,20)

PROCESS=\Radio_Button
      NAME="Heater Indicator"
      LABEL="Heater Indicator"
      START_POS=(70,100)
      DATAIN="Heater State"
      SIZE=(180,30)

PROCESS=\Log
      NAME="Temp in logger"
      DATAIN="Temperature in"
      DATAOUT="temperature in history"

PROCESS=\Write_to_File
      NAME="Write Temp In to File"
      FILE="D:\\Temperature_inside.log"
      DATAIN="Temperature in history"

PROCESS=\Log
      NAME="Temp out logger"
      DATAIN="Temperature out"
```

```
      DATAOUT="Temperature out history"

PROCESS=\Write_to_File
      NAME="Write Temp Out to File"
      FILE="D:\\Temperature_outside.log"
      DATAIN="Temperature out history"

PROCESS=\timer
      NAME="Timer Process"
      DATAOUT="Timer"
```

Figure 48: Configuration file for the brewery temperature controller.

Appendix G

```
//Configuration file for image processing application
DEBUG=false       // Turn off debug messages.  Off by default.

DATA=\Image[640,480,3]
      NAME="Original Image"

DATA=\buff_float[1]
      NAME="Frame rate in"
      BUFF_SIZE_FLOAT=20

DATA=\Image[640,480,3]
      NAME="Inverted Image"

DATA=\buff_int[1]
      NAME="Dropped in"
      BUFF_SIZE_INT=20

DATA=\buff_int[1]
      NAME="Dropped out"
      BUFF_SIZE_INT=20

DATA=\int[1]
      NAME="Frames in"
      INIT=0

DATA=\int[1]
      NAME="Frames out"
      INIT=0

DATA=\buff_float[1]
      NAME="FPS out"
      BUFF_SIZE_FLOAT=20

DATA=\int[1]
      NAME="Frames in 2"
      INIT=0

DATA=\int[1]
```

```
      NAME="Frames out 2"
      INIT=0

DATA=\Image[640,480,3]
      NAME="Original Image 2"

DATA=\Image[640,480,3]
      NAME="Inverted Image 2"

DATA=\int[1]
      NAME="Frames in 3"
      INIT=0

DATA=\int[1]
      NAME="Frames out 3"
      INIT=0

DATA=\int[1]
      NAME="Frames in 4"
      INIT=0

DATA=\int[1]
      NAME="Frames out 4"
      INIT=0

PROCESS=\FireWireCamera
      NAME="Camera"
      CAM_NUM=0
      FORMAT=0
      MODE=4
      FRAMERATE=3
      DATAOUT#image="Original Image"
      DATAOUT#frame_rate="Frame rate in"

PROCESS=\Display
      NAME="WINDOW_1"
      LABEL="Loading of the framework using images"
      START_POS = (0,0)
      SIZE=(1280,1100)
```

```
PROCESS=\Video_Display
      NAME="Video Display (A)"
      START_POS = (0,0)
      SIZE=(640,480)
      DATAIN="Original Image"

PROCESS=\Text
      NAME="Display frames in"
      LABEL="No. of frames into first DS: %s"
      DATAIN="Frames in"
      START_POS = (0,480)
      SIZE=(280,20)

PROCESS=\Text
      NAME="Display frames out"
      LABEL="No. of frames out of first DS: %s"
      DATAIN="Frames out"
      START_POS = (0,500)
      SIZE=(280,20)

PROCESS=\Invert_Image
      NAME="Invert Image"
      DATAIN="Original Image"
      DATAOUT="Inverted Image"

PROCESS=\Video_Display
      NAME="Video Display (B)"
      START_POS = (640,0)
      SIZE=(640,480)
      DATAIN="Inverted Image"

PROCESS=\Text
      NAME="Display frames in 2"
      LABEL="No. of frames into 2nd DS: %s"
      DATAIN="Frames in 2"
      START_POS = (640,480)
      SIZE=(280,20)
```

```
PROCESS=\Text
      NAME="Display frames out 2"
      LABEL="No. of frames out of 2nd DS: %s"
      DATAIN="Frames out 2"
      START_POS = (640,500)
      SIZE=(280,20)

PROCESS=\ImageDSstats
      NAME="Image DS stats 2"
      DATAIN="Inverted Image"
      DATAOUT#frames in="Frames in 2"
      DATAOUT#frames out="Frames out 2"

PROCESS=\Invert_Image
      NAME="Invert Image 2"
      DATAIN="Inverted Image"
      DATAOUT="Original Image 2"

PROCESS=\Video_Display
      NAME="Video Display (C)"
      START_POS = (0,520)
      SIZE=(640,490)
      DATAIN="Original Image 2"

PROCESS=\ImageDSstats
      NAME="Image DS stats 3"
      DATAIN="Original Image 2"
      DATAOUT#frames in="Frames in 3"
      DATAOUT#frames out="Frames out 3"

PROCESS=\Text
      NAME="Display frames in 3"
      LABEL="No. of frames into 3rd DS: %s"
      DATAIN="Frames in 3"
      START_POS = (310,490)
      SIZE=(280,20)

PROCESS=\Text
      NAME="Display frames out 3"
```

```
      LABEL="No. of frames out of 3rd DS: %s"
      DATAIN="Frames out 3"
      START_POS = (310,510)
      SIZE=(280,20)

PROCESS=\Invert_Image
      NAME="Invert Image 3"
      DATAIN="Original Image 2"
      DATAOUT="Inverted Image 2"

PROCESS=\Video_Display
      NAME="Video Display (D)"
      START_POS = (640,530)
      SIZE=(640,480)
      DATAIN="Inverted Image 2"

PROCESS=\ImageDSstats
      NAME="Image DS stats 4"
      DATAIN="Inverted Image 2"
      DATAOUT#frames in="Frames in 4"
      DATAOUT#frames out="Frames out 4"

PROCESS=\Text
      NAME="Frames in 4"
      LABEL="No. of frames into 4th DS: %s"
      DATAIN="Frames in 4"
      START_POS = (950,490)
      SIZE=(280,20)

PROCESS=\Text
      NAME="Frames out 4"
      LABEL="No. of frames out of 4th DS: %s"
      DATAIN="Frames out 4"
      START_POS = (950,510)
      SIZE=(280,20)

QUEUE=\
      NAME="High Queue"
```

```
      PRIORITY=2  // -15=Idle, -2=Lowest, -1=Below normal, 0=normal,
      //1=above normal, 2=Highest, 15=Critical

PROCESS=\ImageDSstats
      NAME="Image DS stats"
      DATAIN="Original Image"
      DATAOUT#dropped in="Dropped in"
      DATAOUT#dropped out="Dropped out"
      DATAOUT#frames in="Frames in"
      DATAOUT#frames out="Frames out"
      DATAOUT#FPS_out="FPS out"
      USE_QUEUE="High Queue"

PROCESS=\Write_to_File
      NAME="Write to File (A)"
      FILE="D:\\LOG-Frame rate in.log"
      DATAIN="Frame rate in"
      USE_QUEUE="High Queue"

PROCESS=\Write_to_File
      NAME="Write to File (B)"
      FILE="D:\\LOG-dropped_in.log"
      DATAIN="Dropped in"
      USE_QUEUE="High Queue"

PROCESS=\Write_to_File
      NAME="Write to File (C)"
      FILE="D:\\LOG-dropped_out.log"
      DATAIN="Dropped out"
      USE_QUEUE="High Queue"

PROCESS=\Write_to_File
      NAME="Write to File (D)"
      FILE="D:\\LOG-FPS out.log"
      DATAIN="FPS out"
      USE_QUEUE="High Queue"
```

Figure 49: Configuration file for the image processing demonstration showing what happens when the system is overloaded.

Appendix H

Configuration file (Figure 50) to test the latency of the framework. This configuration file is just testing the driver so only the parallel process is used.

```
//Configuration file to test the latency of the parallel port driver

PROCESS=\Parallel
        NAME="Parallel"
        LPT=1
        LOOPBACK="true"
```

Figure 50: Configuration file testing the latency of the parallel port driver.

Figure 51 shows the configuration file to measure the latency of a complete sensing and controlling program.

```
//Configuration file to test the latency of the framework

DATA=\int[1]
        NAME="Input"

DATA=\int[1]
        NAME="Output"

PROCESS=\Parallel
        NAME="Parallel"
        LPT=1
        DATAIN="Output"
        DATAOUT="Input"

PROCESS=\Invert DS
        NAME="Invert dataslot"
        DATAIN="Input"
        DATAOUT="Output"
```

Figure 51: Configuration file testing the latency of a complete sensing and controlling program.

9 References

Anumalla, S., B. Ramamurthy, D. C. Gosselin and M. Burbach (2005). Ground water monitoring using smart sensors. IEEE International Conference on Electro Information Technology, 2005, 6 pp.

Bailey, D. and G. Sen Gupta (2004). Error assessment of robot soccer imaging system, Akaroa, New Zealand. Proceedings of Image and Vision Computing New Zealand 2004, 119-124

Bailey, D. G., K. A. Mercer, C. Plaw, R. Ball and H. Barraclough (2004). High Speed Weight Estimation by Image Analysis, Palmerston North, New Zealand. National Conference on Non Destructive Testing, 2004, 89-96

Body, N. B., W. H. Page, J. Y. Khan and R. M. Hodgson (1997). Efficient Mapping of Image Compression Algorithms on a Modern Digital Signal Processor, University of Waikato, New Zealand. 4th Annual New Zealand Engineering and Technology Postgraduate Students Conference, 59-64

Bosch, J. (2000). Design and use of software architectures: Adopting and evolving a product-line approach. New York, NY, USA, ACM Press/Addison-Wesley Publishing Co.

Bulanon, D. M., T. Kataoka, H. Okamoto and S. Hata (2004). Development of a real-time machine vision system for the apple harvesting robot. SICE 2004 Annual Conference, 595-598 vol. 1

Carpenter, G. F., M. Salim and J. Pugh (1991). A pragmatic approach to low-cost, real-time, environment monitoring and prediction. Third International Conference on Software Engineering for Real Time Systems, 1991, 73-78

Choi, J., D. Shin and D. Shin (2005). Research and implementation of the context-aware middleware for controlling home appliances. International Conference on Consumer Electronics, 2005. ICCE. 2005 Digest of Technical Papers., 161-162

Cohen, M. A., J. Sairamesh and M. Chen (2005). Reducing business surprises through proactive, real-time sensing and alert management. Proceedings of the 2005 workshop on End-to-end, sense-and-respond systems, applications and services. Seattle, Washington, USENIX Association: 43-48.

Cope, J., E. Kamel and K. Kamel (1997). A multidrop PC network/database server for material flow and inventory control. IEEE International Symposium on Intelligent Control, 1997, 215-220

Dittenbach, M., D. Merkl and A. Rauber (2000). The growing hierarchical self-organizing map. Neural Networks, 2000. IJCNN 2000, Proceedings of the IEEE-INNS-ENNS International Joint Conference on, 15-19 vol.6

Estrin, D., R. Govindan, J. Heidemann and S. Kumar (1999). Next century challenges: scalable coordination in sensor networks, Seattle, Washington, United States, ACM Press. International Conference on Mobile Computing and Networking, 263 - 270

Froysa, K. G. (2004). Environmental fish farm monitoring, with examples from Scottish and Norwegian rules. OCEANS '04. MTS/IEEE TECHNO-OCEAN '04, 453-460 Vol.1

Garlan, D. and C. Scott (1993). Adding implicit invocation to traditional programming languages. Software Engineering, 1993. Proceedings., 15th International Conference on, 447-455

Garlan, D. and M. Shaw (1993). "An Introduction to Software Architecture." Advances in Software Engineering and Knowledge Engineering **1**.

Giralda, D. B., M. A. Rodriguez, F. J. D. Pernas, J. F. D. Higuera, D. G. Ortega and M. M. Zarzuela (2005). Intelligent system for dynamic transport fleet management. 10th IEEE Conference on Emerging Technologies and Factory Automation, 2005. ETFA 2005., 4 pp.

Graves, A. R. and C. Czarnecki (1999). Distributed generic control for multiple types of telerobot. IEEE International Conference on Robotics and Automation, 1999, 2209-2214 vol.3

Hohmann, P., U. Gerecke and B. Wagner (2003). A Scalable Processing Box for Systems Engineering Teaching with Robotics, Coventry, UK. International Conference on Systems Engineering, 267-271

Hossain, A. (2002). An intelligent sensor network system coupled with statistical process model for predicting machinery health and failure. 2nd ISA/IEEE Sensors for Industry Conference, 2002., 52-56

IEEE (1998). IEEE standard for a smart transducer interface for sensors and actuators - transducer to microprocessor communication protocols and Transducer Electronic Data Sheet (TEDS) formats. IEEE Std 1451.2-1997.

IEEE (2000). IEEE Standard for a Smart Transducer Interface for Sensors and Actuators-Network Capable Application Processor (NCAP) Information Model. IEEE Std 1451.1-1999.

Jacobson, I. (2004). Object-Oriented Software Engineering: A Use Case Driven Approach, Addison Wesley Longman Publishing Co., Inc.

Jump, L. B. (1988). A novel architecture for modular implementation of neural networks. IEEE Workshop on Languages for Automation: Symbiotic and Intelligent Robots, 1988, 25-29

Junior, W. P. and C. E. Pereira (2003). A supervisory tool for real-time industrial automation systems. Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2003., 230-237

Lasserre, V., G. Mauris and L. Foulloy (1998). Fuzzy modelling of measurements acquired by an intelligent ultrasonic telemeter. Instrumentation and Measurement Technology Conference, 1998. IMTC/98. Conference Proceedings. IEEE, 837-842 vol.2

Lee, K. B. and R. D. Schneeman (2000). "Distributed measurement and control based on the IEEE 1451 smart transducer interface standards." Instrumentation and Measurement, IEEE Transactions on **49**(3): 621-627.

Liang, N.-S., L.-C. Fu and C.-L. Wu (2002). An integrated, flexible, and Internet-based control architecture for home automation system in the Internet era. IEEE International Conference on Robotics and Automation, 2002. Proceedings. ICRA '02., 1101-1106

Logix4u, Inpout32.dll for WIN NT/2000/XP, Logix4u, http://www.logix4u.net/index.htm, 2006.

Loyot, E. C., Jr. and A. S. Grimshaw (1993). VMPP: a virtual machine for parallel processing. Parallel Processing Symposium, 1993., Proceedings of Seventh International, 735-740

Lu, Y., T. Q. Chen, C. F. R. Weiman and B. Novak (1996). Video compression for remotely controlled vehicles. International Conference on Image Processing, 1996., 251-254 vol.3

Mainwaring, A., J. Polastre, R. Szewczyk, D. Culler and J. Anderson (2002). Wireless sensor networks for habitat monitoring, Atlanta, Georgia, USA. International Workshop on Wireless Sensor Networks and Applications, 88-97

Meunier, R. (1995). The pipes and filters architecture. Pattern languages of program design. New York, ACM Press/Addison-Wesley Publishing: 427-440.

Murphy, R. R. (2000). "Marsupial and shape-shifting robots for urban search and rescue." IEEE Intelligent Systems and Their Applications **15**(2): 14-19.

Nascimento, C. and J. Dollimore (1993). "A model for co-operative object-orientated programming." Software Engineering Journal **8**(1): 41-48.

Navman, NMEA Reference Manual, http://www.ekf.de/c/cgps/cg2/inf/nmea_reference_manual.pdf, 2004.

Oriolo, G., G. Ulivi and M. Vendittelli (1998). "Real-time map building and navigation for autonomous robots in unknown environments." IEEE Transactions on Systems, Man and Cybernetics, Part B **28**(3): 316-333.

Pal, S. K., R. A. King and A. Hashim (1983). "Image description and primitive extraction using fuzzy sets." IEEE Transactions on Systems, Man and Cybernetics **13**: 94-100.

Philips Semiconductors (1997). TrenchMOS transistor Standard level FET PHB45N03T, Philips Semiconductors.

Philips Semiconductors, The I2C-Bus Specification Version 2.1, Philips Semiconductors, http://www.nxp.com/acrobat_download/literature/9398/39340011.pdf, 2000.

Pomerleau, D. and T. Jochem (1996). "Rapidly adapting machine vision for automated vehicle steering." Expert, IEEE **11**(2): 19-27.

Ryan, J. L. (1989). "Home automation." Electronics & Communication Engineering Journal **1**(4): 185-192.

Saptharishi, M., C. Spence Oliver, C. P. Diehl, K. S. Bhat, J. M. Dolan, A. Trebi-Ollennu and P. K. Khosla (2002). "Distributed surveillance and reconnaissance using multiple autonomous ATVs: CyberScout." IEEE Transactions on Robotics and Automation **18**(5): 826-836.

Savall, J., A. Avello and L. Briones (1999). Two compact robots for remote inspection of hazardous areas in nuclear power plants. IEEE International Conference on Robotics and Automation, 1999., 1993-1998 vol.3

Sen Gupta, G., C. H. Messom and S. Demidenko (2004). State Transition Based (STB) Role Assignment and Behaviour Programming in Collaborative Robotics, Palmerston North, New Zealand. Proceedings of the 2nd International Conference on Autonomous Robots and Agents, ICARA 2004, 385-390

Sen Gupta, G., C. H. Messom and H. L. Sng (2002). State Transition Based Supervisory Control for Robot Soccer System. International Workshop on Electronic Design, Test and Applications (DELTA), 338-342

Shafer, S., A. Stentz and C. Thorpe (1986). An architecture for sensor fusion in a mobile robot. Robotics and Automation. Proceedings. 1986 IEEE International Conference on, 2002-2011

Silva, A. P. and G. R. Mateus (2003). Location-based taxi service in wireless communication environment. 36th Annual Simulation Symposium, 2003., 47-54

Smith, J. E. and R. Nair (2005). "The architecture of virtual machines." Computer **38**(5): 32-38.

Smith, J. S., R. A. Wysk, D. T. Sturrock, S. E. Ramaswamy, G. D. Smith and S. B. Joshi (1994). Discrete event simulation for shop floor control, Orlando, Florida, United States, Society for Computer Simulation International. Proceedings of the 26th conference on Winter simulation, 962 - 969

Sohrabi, K., W. Merrill, J. Elson, L. Girod, F. Newberg and W. Kaiser (2004). "Methods for scalable self-assembly of ad hoc wireless sensor networks." Mobile Computing, IEEE Transactions on **3**(4): 317 - 331.

Srini, V. P. (1986). "An architectural comparison of dataflow systems." Computer **19**(3): 68 - 88.

Staroswiecki, M. (2005). "Intelligent sensors: a functional view." Industrial Informatics, IEEE Transactions on **1**(4): 238-249.

Sullivan, M., S. W. Butler, J. Hirsch and C. J. Wang (1994). "A control-to-target architecture for process control." IEEE Transactions on Semiconductor Manufacturing **7**(2): 134-148.

TELETYPE GPS, NMEA String Information, TELETYPE GPS, http://www.teletype.com/pages/support/Documentation/RMC_log_info.htm, 2007.

Tian, G. Y. (2001). "Design and implementation of distributed measurement systems using fieldbus-based intelligent sensors." Instrumentation and Measurement, IEEE Transactions on **50**(5): 1197-1202.

Venkatesan, G. and H. Abachi (2002). Parallel processing computer architectures for process control. Proceedings of the Thirty-Fourth Southeastern Symposium on System Theory, 2002., 234-238

Wacks, K. (2002). "Home systems standards: achievements and challenges." Communications Magazine, IEEE **40**(4): 152-159.

Zhenzhong, W., Z. Guangjun and L. Xin (2001). The application of machine vision in inspecting position-control accuracy of motor control systems. Proceedings of the Fifth International Conference on Electrical Machines and Systems, 2001. ICEMS 2001., 787-790 vol.2

Zhifei, C., A. Yuejun, J. Kebing and S. Changzhi (2001). Intelligent control of alternative current permanent magnet servomotor using neural network. Proceedings of the Fifth International Conference on Electrical Machines and Systems, 2001. ICEMS 2001., 743-746 vol.2