2004-09-03

An Open Architecture for Versatile Machine and Actuator Control

Michael Scott Baxter
Brigham Young University - Provo

AN OPEN ARCHITECTURE FOR VERSATILE MACHINE AND ACTUATOR CONTROL

by

Michael Baxter

A thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science Mechanical Engineering

Department of Mechanical Engineering

Brigham Young University

April 2005

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Michael S. Baxter

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date:	W. Edward Red, Chair
Date:	C. Greg Jensen
Date:	Timothy W. McLain

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Michael S. Baxter in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date:

W. Edward Red
Chairman, Graduate Committee

Accepted for the Department:

Matthew R. Jones
Graduate Coordinator

Accepted for the College:

Douglas M. Chabries
Dean, Ira A. Fulton College of
Engineering and Technology

ABSTRACT

AN OPEN ARCHITECTURE FOR VERSATILE

MACHINE AND ACTUATOR CONTROL

Michael Baxter

Department of Mechanical Engineering

Master of Science

Automatic control technology increases usability, reliability and productivity in manufacturing, transportation, and climate control. There are many additional areas of modern life that could benefit through automatic control; however, current automation components are too expensive or aren't sufficiently flexible. For example, the cost of current commercial motion control components precludes their use in an average home.

This thesis describes an automatic control methodology that is low cost and is flexible enough for a wide variety of control applications. Typical applications could include:

- Home lighting, security and appliances
- Commercial building heating, ventilation and air conditioning
- Industrial machine tool and process control

This automation methodology eliminates several expensive and inflexible aspects of present-day industrial automation. This is accomplished by implementing application-specific control algorithms in software run on a generic computer rather than on purpose-built hardware. This computer calculates control values for each control application connected to it via real-time communication network.

This technique is similar to that of a desktop PC. When using a peripheral device, such as a printer or scanner, the PC executes device driver software to calculate control values for the devices. These values are communicated to the device over a shared bus. The automation methodology described here seeks to emulate this software-based control paradigm.

This methodology reduces cost and increases flexibility in two ways. First, it eliminates application-specific control hardware and replaces it with software. This reduces the cost by eliminating the need for unique, proprietary control hardware for each product or system. Second, the software approach increases flexibility. For example, one could download a new clothes washing machine cycle via the Internet. Software control provides considerable freedom in designing and implementing control systems by allowing the designer to change system functionality without having to replace or modify hardware or even be present at the location where the control system is used.

This thesis describes the development of this new control methodology. To validate its performance a home automation system is implemented. This implementation included control of laundry appliances, lighting, TV and other common household devices.

ACKNOWLEDGMENTS

This project benefited from help and encouragement from many people. Brent Baxter, my father, provided many useful reviews of this this research. His encouragement and technical understanding provided significant guidance to see this project move successfully to completion.

Daniel Thompson provided helpful review at the outset of the project. He is also credited with the creation of the 2^n scheduling algorithm described as part of this thesis. I would like to thank Hans Fugal for his methodical review of this architecture and instrumental work in its implementation. Brian Fretz, Anshul Malvi, and Greg Woodworth were likewise instrumental in designing, fabricating and testing the hardware used in the implementation of this control system. Kurt Didenhover and Dave Grant were key to the success of the graphical user interfaces. Numerous family members provided valuable help editing the final document. Dr. Red likewise provided guidance and encouragement through many phases of the work.

Funding came from the Utah State Center of Excellence Program.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

GLOSSARY

Actuator: Network device that performs an intended action

Asynchronous Data: Data exchanged whenever specified conditions have been met

Communication Pattern: Sequence of data items comprising a transaction

Device: Equipment or system controlled over a network

Digital Signal Processor (DSP): Computing device used to process isochronous data

Inter Process Communication: Information exchanged between software processes

Isochronous Data Transfer: Data exchanged on a uniform time intervals

Linear Time-Invariant (LTI): System exhibiting time-invariant, linear behavior

Local Area Network (LAN): Shared bus communications interconnect

MIMO: Multiple Input, Multiple Output

NCS: Network Control System, or Network-based Control System

Network Control System: Shared network used to communicate devices control values

Network Interface Card: Circuit board containing network interface circuitry

Proportional-Integral-Derivative (PID): Control law in the form $K_p + K_i * 1/s + K_d * s$

Real-time Personal Computer (PC): Computer provided with isochronous capability.

Real-time Scheduling: Providing shared resource access within isochronous deadlines

Shared Bus: Interconnection based on a shared set of resources

SISO: Single Input, Single Output systems

Synchronous Data: Data exchanges occurring under control of a periodic clock signal

Unshared Connection: Point-to-point interconnection with between pairs of devices

Wide-Area Automation: Networked control of large, distributed systems

CHAPTER 1 INTRODUCTION

Automatic control systems appeared at the dawn of the industrial revolution and provided great improvements in efficiency, allowing machines to perform repetitive tasks without continuous attention. Digital computers further revolutionized automatic control methodology by greatly expanding the complexity that could be managed in a control system. Lewis[28] expands this historical background.

Three challenges emerged from these advances in control technology as explained by Franklin[12][13], and Lewis[28]. The first is to provide stable, reliable control, often through the use of feedback from the device being controlled. The second is to allow an operator to provide user input to the system. The third is to coordinate independently controlled systems that have unique control requirements. Unfortunately, these challenges often force today's system engineers to use a control methodology employing proprietary control hardware and software specific to each requirement. For example, it is common for each item of today's manufacturing equipment to be controlled locally by its own digital signal processor with a PC dedicated to its user interface while a remotely located PC coordinates the machine with others.

This arrangement has some significant limitations. First, such a system has limited flexibility for addressing requirements outside the original application. Second, this methodology provides no standard way to coordinate independent systems, making system integration and modification difficult. Finally, such a system is inherently expensive to install and maintain because of the custom nature of each component.

Overcoming these drawbacks requires good control system performance, configuration flexibility and inexpensive hardware. A key concept for doing this is called *direct control* which means that a computer can directly control several machines, actuators and sensors over a shared network, as shown in Figure 1.

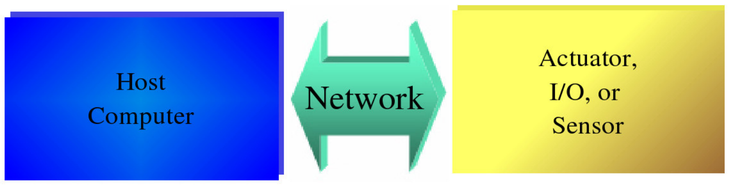


Figure 1: Direct computer control of actuators and sensors

An embodiment of this concept would include a computer that directly controls one or more actuators on a machine, causing it to follow a specified motion trajectory. The direct control concept underlies much of the work described in this thesis and is referred to as the Versatile Machine Actuator Control (VMAC) methodology.

An example of this concept implemented to control an automatic box closing system is shown in Figure 2. In this implementation sensors indicate to the computer when a box has arrived on the conveyor belt. In turn the computer directly operates various actuators to prepare the box for shipment. The computer communicates with the actuators and sensors via a Local Area Network (LAN).

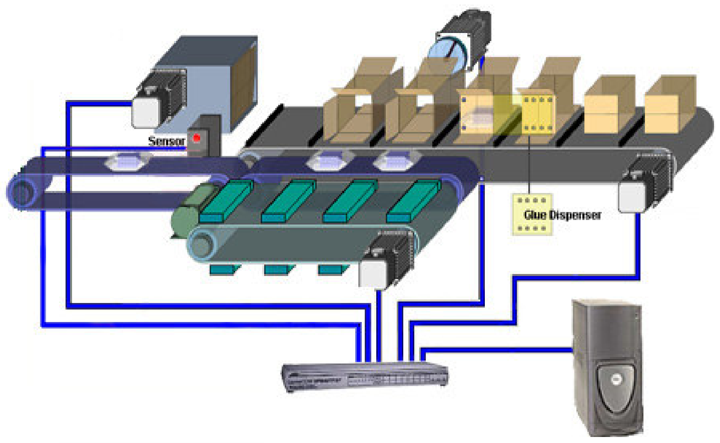


Figure 2: Box closing system based on VMAC

A prime example of the difference between VMAC control and traditional control comes from the history of desktop document printers. Consider how printers were attached to the original Apple II series personal computers, shown in Figure 3. Customers could purchase a limited selection of printers made by Apple and it was necessary to purchase an interface card that plugged into a specific connector on the computer's main circuit board. After installing the hardware, the system software had to be configured to use the particular printer. The software setup required an in-depth understanding of the system and a certain amount of low-level programming (see http://web.pdx.edu/~heiss/technotes/atlk/tn.atlk.4.html for more detail). The computer communicated with the intermediate control card which in turn drove the actuators inside the printer.

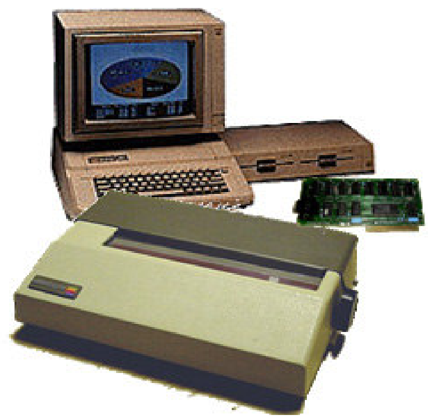


Figure 3: Apple IIe® personal computer with interface card and printer

Over time, new standardized computer interfaces allowed control algorithms to be implemented as driver software, eliminating the need for a custom interface card with its associated setup difficulties. Modern personal computers directly control printers, and there is no need for intermediate hardware. Today one may purchase a wide variety of printers, such as is shown in Figure 4, from many different manufacturers and install them with little or no setup effort. The printer can be attached using a standard communication interface, such as USB, and the system will automatically load the correct driver software for the particular printer.

Figure 4: Modern PC printer with no intermediate control hardware

It is convenient to think of the early document printer control technology as a three staged system. The stages are, user interface computer, real-time control loop hardware and actuators and sensors, as shown in Figure 5.

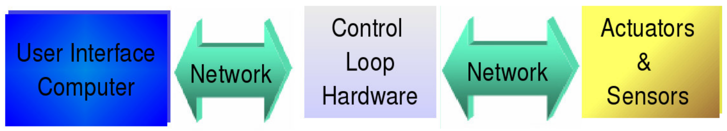


Figure 5: Traditional three stage control system

The current state of automation technology reflects the early printer technology described above. Many of today's systems contain a dedicated computer to operate the the user-interface, usually derived from a personal computer, with proprietary intermediate hardware, perhaps based on a digital signal processor, for the feedback control loop. The BYU heating plant works this way, where myriad actuators, pumps and motors each have their own controller hardware that executes their real-time control loops while communicating status information to an external user interface computer. The user interface computer shown in Figure 6 (foreground) communicates with the intermediate hardware (background, right) that controls individual motors and actuators (far background). This system could be simplified by standard interfaces, just as was the case with document printers for personal computers.

Figure 6: BYU heating plant

Three stage control systems have proved expensive to purchase, install and maintain, while providing only limited flexibility. The limited flexibility results from having the control and coordination implemented in purpose-built, distributed hardware. This methodology has been problematic to the BYU heating plant when trying to upgrade or replace equipment which, almost invariably, requires replacing control hardware at great expense.

Conventional computer controlled milling machines provide another example of the ubiquitous nature of three stage control systems. The different components of an Anlam controller mounted on a milling machine at BYU are shown in Figure 7. A personal computer runs user interface software and communicates through proprietary interconnections (the gray bundled cables) with the control loop hardware shown on the right. This real-time control hardware drives the milling machine's three actuators (the gold vertical cards) as well as the spindle motor (the cream colored box). If the controller were to use a more flexible control methodology it would be possible to use the same machine for multiple processes such as coordinate measuring, welding, part assembly, and friction stir welding.

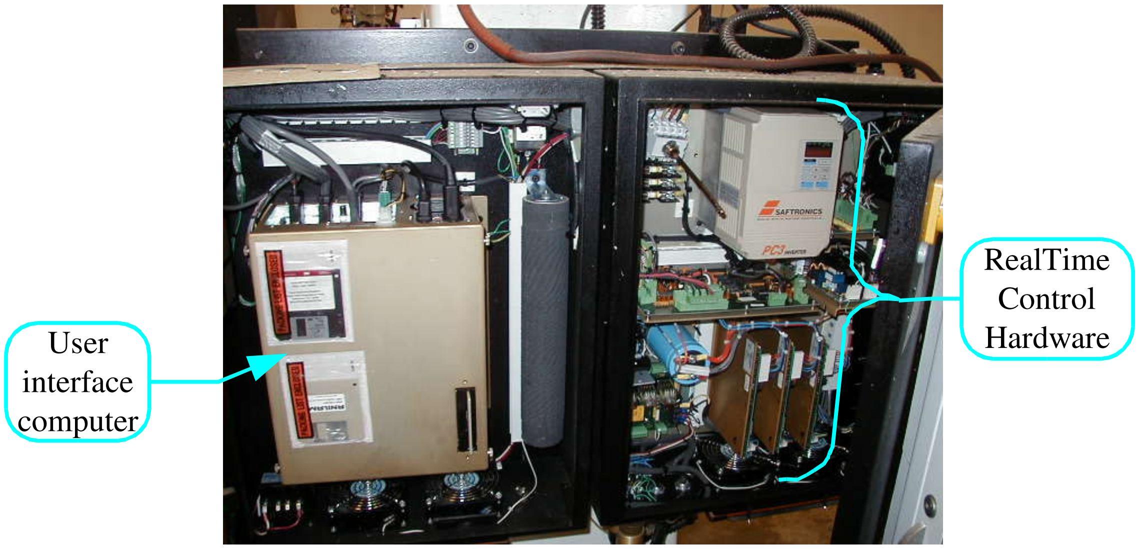


Figure 7: Current computer controlled milling machine

The VMAC approach is to move control loops from proprietary hardware to software running on a generic, low cost computer. Stated differently, the system or machine to be controlled, no longer has a native controller. Taking a hint from the personal computer's USB communications and device driver example, the VMAC control methodology treats each actuator and sensor, or system of actuators and sensors as separate software controlled device attached to a common network.

This methodology provides wide latitude in usage because the function of the machine or system being controlled can be determined by software using a standardized network communication protocol. The new system can be comparatively low cost because it runs on readily available personal computer components instead of purpose built hardware specific to each actuator or sensor. A significant benefit of this is that there is no need to replace supporting control loop infrastructure as equipment is replaced. The common interconnection network and standard configuration allow system designers the flexibility to add devices as requirements dictate. This flexibility is shown in Figure 8. The master control function resides in a computer that runs the software drivers for each device connected to the control network. When devices are attached or removed from the control network the proper software is loaded or unloaded in the host controller.

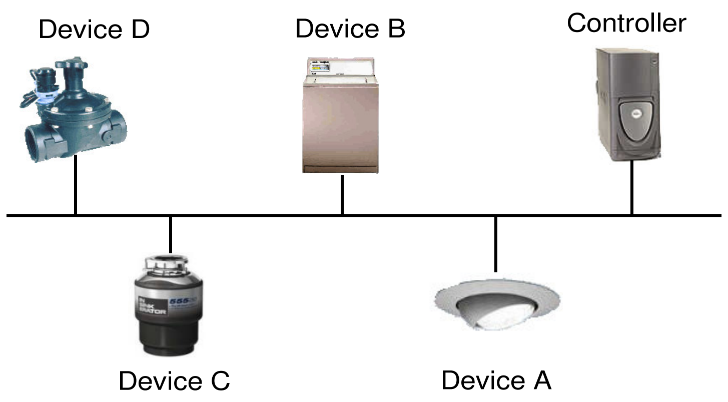


Figure 8: Software controlled VMAC actuators and sensors

The research described in this thesis contributes to automatic control technology by demonstrating:

- Digital control loop operation over standard PC network components
- Automatic device discovery (e.g. “Plug 'n Play”)
- Versatile user interface system
- Use of “off the shelf”, low-cost, computing and network hardware

These contributions are utilized to demonstrate a low-cost control system that is easily installed and maintained. It's automatic configuration allows such a system to be setup by comparatively untrained individuals. Its direct control allows great flexibility in system reconfiguration and control application. The user interface system provides flexible venues for providing user input. Each of these contributions are described in detail in Chapter 4.

CHAPTER 2 Review of Pertinent Literature

Much of the current automation research is focused on incrementally extending the traditional, three-stage control methodology. This expansion is accomplished primarily in two ways.

- Incorporating a network as a communication path for control loops
- Incorporating more programmability into the real-time control hardware.

The VMAC effort draws on direct control research as well as real-time network communication and real-time scheduling.

2.1 Network-Based Control Systems

Branicky[4] introduced the concept of a network-based control system (NCS) as a control system where one or more data paths is via a LAN. When using a common computer network, communication latency is inherently non-deterministic. A considerable body of research literature is focused on making the network latencies more deterministic or designing control algorithms that are more robust to communication delay or loss.

Most recent research is focused on two system configurations. The first passes only feedback information over the network. This configuration is shown in Figure 9.

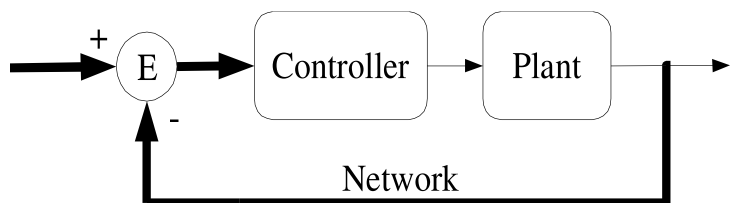


Figure 9: NCS with feedback via LAN

A second configuration, show in Figure 10, uses the network for communicating both control and feedback information.

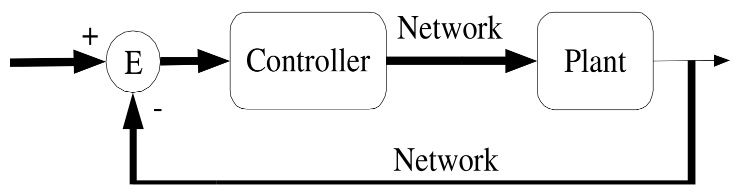


Figure 10: NCS with all communication by LAN

Kelling[22] and Walsh[47] enumerate some of the advantages of a network-based control system, which include:

- Easy integration with Internet technologies
- Greater flexibility in control application
- Mobility of user interfaces and system access
- Potentially reduced cost

2.1.1 **Methods for Dealing with Network-Induced Delay**

Walsh[47][48] modeled the network-induced delay as a process following a Poisson distribution. Lee[27] and Montestruque[33] considered approaches based on learning algorithms to develop a time-varying model for network latency. These models are used by most researchers to develop techniques for mitigating the effect of this delay.

Network latencies can be made more deterministic by limiting network utilization or by explicitly scheduling network traffic to avoid collisions.

Brocket[5] proposed the idea of minimum attention control as a technique for reducing overall network traffic. This problem is posed as an optimization of an attention cost function that minimizes $\|\partial u / \partial t\|$ and $\|\partial u / \partial x\|$. The resulting control law *u* has the least dependence on time *t* and plant state *x*. He showed that this control law requires less frequent updating than for one designed using traditional methods.

Zhang and Branicky[52] analyzed an another approach for minimizing network traffic where only the feedback sensor is connected to the network. This relieves the network of the controller/plant communication. They presented a theoretical analysis to determine the upper bound on the return path latency. A modest reduction in network traffic is demonstrated with this technique.

Hristu-Varsakelis[20] presented a technique for reducing communication bandwidth of control loops where both the forward and back bath is over a network. He established a communication pattern that closes the control loop at intermittent times while incorporating feedforward control. He proposed criteria for deciding when to close a particular control loop as well as the use of a zero-order-hold for the plant and sensors when communication is lost. He analytically presented stability criteria for this technique.

Montestruque[33] presented an elegant analysis and technique for minimizing the required communication bandwidth of an NCS where only feedback takes place via network. He showed that by utilizing a plant model in the controller that incorporates network delay it is possible to stabilize a system at reduced communication rates. He presented a technique for calculating the maximum feedback delay.

Hristu-Varsakelis[19] presented an analysis of scheduled communication for a NCS. He examined a collection of LTI systems being controlled via network with forward and back paths over a network. He showed that the condition for stability under periodic communication is also sufficient under an interrupt-based communication policy. This technique allowed a more efficient utilization of the network by controlling access to it through a formalized communication policy and exhibited improved performance.

Walsh[48] demonstrated improved performance of a full closed loop NCS when using a Try-Once-Discard (TOD) scheduling methodology. The TOD scheduler transmits data with the largest control error first up to a certain time bound when the data is discarded and a new value used. He proposed criteria for deciding where this bound is that will guarantee system stability. He applied this technique to SISO systems and verifies asymptotic stability through simulation and proof. Walsh[49] improved on this technique and applied it to MIMO systems and verified its performance through simulation and proof.

2.1.2 **Improved Controller and Plant**

Several approaches have been suggested to improved performance of the NCS through more sophisticated control laws.

Zhang, et al.[3][53] presented a well constructed analysis of network-induced delay for a control loop where both the forward and back path communication happens over a network. They proposed a computational technique for defining a stability region in terms of loop sample rate and communication latency. Xie[51] analyzed the same system as Zhang[53] but provided a more computationally efficient algorithm for calculating the stability region. To compensate for network-induced delay, Zhang et al.[53] proposed a more sophisticated state estimator that incorporates different models of the delay. These models depend on control system configuration. They verified their analysis by implementing their design in hardware.

Walsh[47] shows that a NCS can be used to control linear or non-linear plants. He showed the stability criteria for a linearized, inverted pendulum system given feedback over a network.

Lee, et al.[27] analyzed fuzzy-logic based controllers for NCSs. Their analysis showed that a fuzzy-logic controller can be implemented in a NCS with improved performance over other types of controllers such as PID. They experimentally verified their results over Profibus with two computers.

The research focused on network based control systems provides a foundation from which the research described in this thesis is drawn. The research described focuses on methods for handling the non-deterministic behavior of common local area networks. The work of this thesis presents a method for obtaining deterministic network communication.

2.1.3 Greater Real-Time Hardware Programmability

An additional body of pertinent research seeks to incrementally move toward a direct control paradigm. Several industrial automation companies are working on incrementally expanding the functionality of the traditional automation system. Their work is primarily focused on increased hardware programmability. This programmability allows greater flexibility, and interoperability in their systems. Unfortunately this approach provides only incremental improvement in system utility while increasing overall system cost.

Ormec Company(http://www.ormec.com/) is currently selling a line of motion control systems under the ServoWire name where the servo controller and amplifiers communicate via IEEE 1394. They advertise increased performance and utility through programmability of their servo controller and amplifiers.

Agile Systems, Inc.,(http://www.agile-systems.com/) is likewise pursuing digital interfaces for many of its motion control components. VanKampen[45], explained the benefits of this digitization.

One of Rockwell Automation's (http://www.rockwellautomation.com/) new control systems illustrates the benefits of greater programmability[25]. This programmability allows greater flexibility in usage application of their systems as latter-logic and other control parameters can more easily be re-programmed.

Grimble[16], Haines[17], Liu, et al.[29], and Newman[34] described the virtues of this trend and suggested that it is the direction of future automated systems.

2.2 VMAC Control Research

The direct control concept embodied in the VMAC system treats actuators and sensors as peripheral devices in a similar way that a PC treats a printer, scanner or external disk drive as a peripheral device. Each device, or set of devices being controlled has an associated software device driver that manages its real-time control loops and coordination.

Currently there is limited research literature exploring direct control. Consequently the VMAC effort draws on research focused on real-time network communication and real-time scheduling algorithms in addition to research focused on direct control.

2.2.1 **Direct Control**

Several teams at Brigham Young University are studying methods for direct control. McBride[32] proposed an architecture for directly controlling a milling machine. Ghirmire[15] proposed a device interface architecture that connects McBride's controller to hardware actuators. McBride and Ghirmire demonstrated the functionality of their work by implementing a motion controller for a bench-top milling machine. They successfully machined complex surfaces with their machine. Bosley[2] extended their work by developing an interface that could be used by a CAD software package. Bosley's interface facilitated transfer of surface description data to the systems developed by McBride and Ghimire. The resulting architecture, shown in Figure 11, allowed a surface to be directly machined from the CAD surface description.

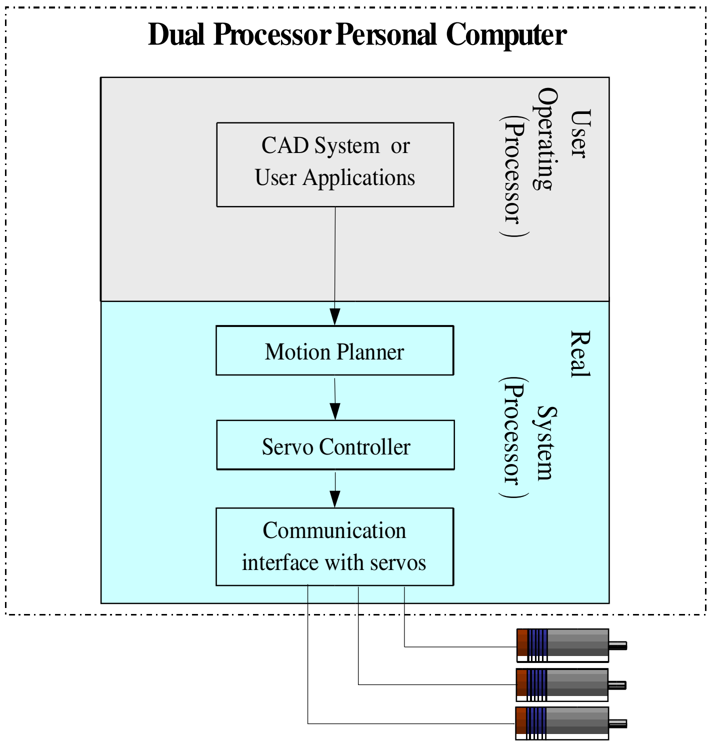

Dual Processor Personal Computer
CAD System or User Applications
User Operating (Processor)
Motion Planner
Servo Controller
Communication interface with servos
Real System (Processor)

Figure 11: User interface interacts with real-time software

Bosley[2] developed a motion planer for use with a milling machine. This planner calculated trajectories and paths given different motion constraints. An important implication of his work is that the whole controller needn't be re written to drive a unique machine. Bosley's work formed the foundation of a device driver for a particular machine. By loading different motion planning software virtually any machine tool configuration could be controlled.

Ghimire[15] proposed the interface that connects the servos to McBride's controller. His work included a communication protocol and real-time communication environment. Initially he proposed using an IEEE 1394 physical layer to take advantage of its native isochronous data transport, but this proved problematic. He circumvented these problems using an optical, unshared communication path to each actuator in a physical and logical star configuration.

2.2.2 **Real-Time Communication**

The VMAC methodology requires a host computer to communicate with various peripheral devices on a periodic schedule. Some of the most commonly researched communication standards for industrial control include Controller Area Network (CAN), Profibus, Foundation Fieldbus, DeviceNet, LonWorks, X10, CeBus, Mil-Std 1553B, and Ethernet. There are myriad other existing standards, however this research will only consider these.

Kuberi and Shin[55] presented scheduling solutions for distributed systems that communicate via CAN. They presented their Mixed Traffic Scheduler (MTS) which manages the traffic on the network to ensure timely communication. This system took advantage of CAN's media access by priority, which requires each device connected to the network have the same scheduler running. They demonstrated that this technique globally ensures that tasks of similar priority will have similar communication latencies.

Törngren[42] noted several qualities and problems in using CAN as a basis for distributed real-time control systems. He noted the convenient media access of CAN by priority, as well as drawbacks including a limited error handling policy, and a lack of clock synchronization.

Tovar[43] explored the use of Profibus as a basis for real-time control of distributed systems and points out that, in the worst case, Profibus doesn't allow for complete transmission of all available messages. Tovar's solution, called Constrained Low Priority Traffic, provided for complete R/T communication subject to certain constraints.

Rehg[37] explored Foundation Fieldbus as a viable real-time communication system. He noted that Foundation Fieldbus, though limited in current communication bandwidth, looks promising with the completion of new higher speed variants of the original standard.

Schiffer[39] explored the Control and Information Protocol family (CIP). The most well known members of this protocol are DeviceNet^{tm} and ControlNet^{tm}. He proposed that the CIP family of protocols is more versatile than many common standards, such as CAN, but is likewise more expensive and complex to work with. Additionally, he proposed a variant of Ethernet, called Ethernet/Industrial Protocal (Ethernet/IP), and showed its suitability for many communication requirements on a factory floor.

Gerhart[14], Humpleman[21], and Lucas[31] described low cost communication standards used in many structures, often utilizing the structures power system as the communication media. Gerhart pointed out the limited 60 bit/s bandwidth of X10 as unacceptable. He showed that CeBus is more versatile than X10 but concluded that LonWorks is the most functional protocol with a more usable bandwidth of 1.25 Mbit/s. LonWorks can also utilize a power system as its communication media.

To demonstrate the functionality of LonWoks, Echelon Corporation[11] implemented a rail car control system based on the LonWork standard. They demonstrated that it is possible to implement real-time control systems where the communication takes place over the power bus of a rail car.

Mil-Std-1553B is one of the most venerable and trusted real-time communication standards available and is described in a Defense Department standards document [10]. This standard has been used successfully over many different media with primary application in military avionics. The introduction to the standard points out that it is well suited for real-time control systems. However, it is very expensive and not widely used outside military aviation.

Novacek[35] described a novel approach to real-time communication that can be implemented using many standard network protocols and media. Coordinated nodes on a network are allocated regular times to communicate with the host controller whether they need to or not, instead of communicating after an external asynchronous event. He showed that this technique provides an upper bound on network latencies, making it well suited for many control applications.

The Ethernet standard is probably the most widely researched standard today. There are many attractive benefits to using Ethernet, as Samaranayake[38] and Loundsburry [30]pointed out.

- The hardware is inexpensive
- Relatively high bandwidth of 100 Mbits/s and greater
- Well established base of understanding in its structure and use
- Easy integration with Internet technologies.

Samaranayake[38] and Loundsburry[30] both pointed out that there is one significant drawback to using Ethernet in a real-time control environment. In its standard implementation communication latencies are inherently non-deterministic. Kooperman[26], Samaranayake[38] and the IEEE 802.3 standard[41] noted that the reason for Ethernet's non-determinism is the lax media access rules. In standard Ethernet there are no rules about when a device can access the network media. When two devices do transmit simultaneously, the resulting collision causes unusable data. To resolve this problem, Ethernet has a collision detection scheme where each transmitter detects the collision and waits a random period of time before transmitting again. If this transmission collides again the previous waiting period is increased exponentially. As Cisco[7] pointed out, the advantage of this communication technique is its low hardware cost.

Kerkes[23] explored one technique for modifying the Ethernet standard slightly to obtain a deterministic communication environment. Kerkes designed a don't-speak-unless-spoken-to policy where the controlling node on the network instigates all communication. By this technique collisions are completely eliminated and the communication latency is tightly bounded to that of the soft and firmware implementations of the network interfaces. Kerkes and his team successfully implemented this modified Ethernet to control a full-motion flight simulator.

Loundsbury[30] pointed out some considerations when using Ethernet as an industrial control network. He showed that connectors, media and the electrical isolation of Ethernet are not as well suited for a control system as are other standards such as Profibus. He concluded that there needs to be an amount of future development in Ethernet hardware to enable it to move onto the factory floor.

As part of his research, Samaranayake[38] pointed out that network switching technology influences system performance. He notes that a network switch adds about 10 μs latency to each communication.

2.2.3 **Real-Time Scheduling**

Branicky, et al.[4] explored co-design of the control law and the communication scheduling algorithm used with it. They formulated the optimal scheduling problem under both rate-monotonic (RM) scheduling constraints and NCS-stability constraints. They also developed an analytical bound on the communication drop-out percentage and latency that will guarantee stability. They concluded that the RM scheduler is best for fixed priority systems where the priority can be encoded in the message identifier.

Stankovic, et al.[40] and Walsh, et al.[46] explored dynamic scheduling techniques for NCSs. They also showed that conventional queuing theory is not helpful in designing a communication scheduler. Walsh, et al.[46] proposed that queues be eliminated completely and a modified try-one-discard scheduler called maximum-error-first with try-once-discard (MEF-TOD) be used.

Bonuccelli, et al.[1], Cuco, et al.[9], Kim, et al.[24], and Park, et al.[36] all presented communication scheduling techniques for NCSs requiring three types of data transport: sporadic, periodic and message. Sporadic is asynchronous data that is part of a devices control scheme. Periodic is isochronous data that is part of a device's control loops. Message data pertains to global system events. Park's work allowed the network bandwidth to be allocated to the three types of data, and guarantees real-time communication of periodic and sporadic data. Park presented several implementations to demonstrate the validity of their scheduler called maximum allowable delay bound (MADB). Bonuccelli studied the same system and proposes a simplified scheduling algorithm of Earliest Due Date and showed system stability for network utilization less than 50%.

The automation research taking place today is primarily focused on improving NCS performance through increased communication determinism and more sophisticated control algorithms. The VMAC research effort draws on research focused on real-time network communication and communication scheduling. The background presented in this chapter forms the foundation from which the VMAC architecture is developed.

CHAPTER 3 RESEARCH METHODOLOGY

This chapter contains a description of the research done for this thesis. This work included development of each component needed for the VMAC control methodology as well as filing for legal protection.

The developments described here provided a control architecture that satisfied the research objectives of this thesis, which are:

- Digital control loop operation over standard PC network components
- Automatic device discovery (e.g. "Plug 'n Play")
- Versatile user interface system
- Use of "off the shelf", low-cost, computing and network hardware

3.1 VMAC System Components

The five major components that comprise the VMAC control methodology are shown in Figure 12. With the exception of the network interface card each of the components shown required specific consideration and design.

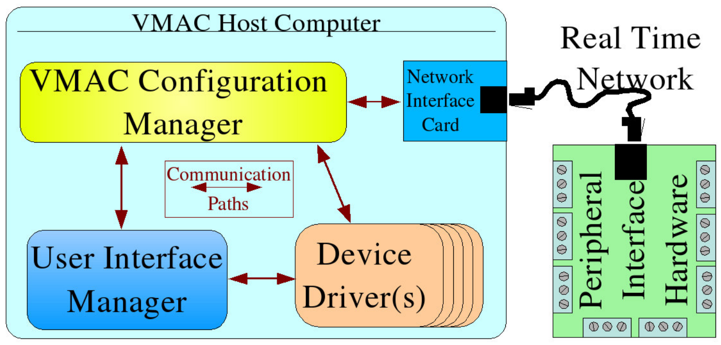


Figure 12: VMAC control architecture

The real-time network facilitates data transport in real-time. The VMAC configuration manager facilitates device discovery as well as coordinates each of the system components. The user interface manager facilitates the display of process status information generated by the device drivers. The peripheral interface hardware forms the interface with the control application hardware.

3.1.1 **Real-Time Communication Environment**

Timely, periodic communication with devices connected to the VMAC system is fundamental for successful operation. In particular, digital control loops require information to be transferred to and from the device at a periodic rate high enough to ensure stability. This performance requirement primarily placed a constraint on the selection of a communication standard.

The communication environment developed for this control methodology has the following characteristics:

- Real-time communication at loop rates in excess of 100 samples per second
- Communication over distances greater than 100 feet
- Bandwidth greater than 10 Mbit/s

The loop communication sample rate is significant because many common computer networking standards are optimized for bulk data transport rather than the typically small data payloads common to feedback control. A maximum rate of 100 samples per second provided satisfactory performance of all control loops used in the implementation of this control methodology. Practicality dictated the minimum required distance over which control loops can be communicated. The effective bandwidth requirement provides an adequate trade off between the number of devices being controlled and their respective control loop sample rate.

After selecting the Ethernet implementation of IEEE 802.3 a communication scheduler was developed to ensure deterministic behavior of the communication environment. By explicitly scheduling all communication data collisions on the network are eliminated. This ensures that the network is always available whenever a communication is needed.

With every communication standard there is a certain amount of computational overhead needed to format data to be sent. This overhead must be as small as possible. A reduced overhead implementation of Ethernet was developed for this control methodology.

3.1.2 **VMAC Configuration Management**

The VMAC Configuration Manager is responsible for coordinating each component of the VMAC system. This required a set of communication and device management services to automatically configure the VMAC system. These services:

- Identified new devices and start the appropriate driver software running
- Unloaded device driver software when a device is disconnected

- Enforced error policies

- Facilitated communication between software components and the network

This management provided the ability to connect and use control application hardware without input from the user. For example, when a machine is connected to the VMAC control system the system automatically configures its self to use the new machine. When the new machine is in operation, the VMAC manager facilitates communication from the machine's device driver to the network and vise versa. Likewise, the new machine can be disconnected from the control network without input from the user. This plug and play interoperability serves to make this system usable and inexpensive.

The communication between the device drivers and the VMAC manager facilitated the encapsulation of all application-specific control software into a device driver. This encapsulation provided a control system that was completely generic. The device driver software for a control application entirely defined the functionality of that application.

3.1.3 **Device Drivers**

The VMAC control methodology is designed to be completely generic. This means that it natively contains no application-specific control abilities. This project developed a standard set of programming interfaces that allow a control application's device driver to access the services of the VMAC system.

Programming interfaces facilitate:

- Real-time network communication

- Communication between the driver and system software

The VMAC manager contains these two interfaces. When a driver has an update for its respective peripheral hardware it passes the update to the VMAC manager. At the appropriate time according to the communication schedule the VMAC manager will send the update to the peripheral hardware over the network. Likewise, when there is a status update for the user interface system, the driver passes this first to the user interface manager which formats the data and passes it to the VMAC manager to be sent on the network.

This separation of application control software into a device driver provided the flexibility, as described in Chapter 1, to determine application functionality purely through software modification. This flexibility implies the ability for a third-party company to write and remotely maintain or update driver software run on this control system.

3.1.4 **User Interface System**

The user interface system allowed device drivers to display process feedback and accept user input. The system presented information viewable by any modern web browser. This allows any device that is able to run a web browser to function as a UI portal for this system.

This development included:

- System interface with the Internet
- Interface management software
- Communication protocol for UI devices
- Web-compatible representation of user interface information

The interface with the Internet was a computer running a web server. The interface management software was the User Interface Manager depicted in Figure 12. This software was responsible for managing the web server computers and also formating UI information passed to it.

For the implementation described in Chapter 5 Macromedia's Flash provided the graphical environment to interactively display user interface information. Many different methods, such as Java, or simple XML could also be used for the same purpose.

The UI web server computer was connected to the real-time network as well as the Internet. When the host computer had a status update from a driver, it sent it to the web server computer over the network. The UI web server then updated the respective device's web page. These device pages were served via the Internet, making it available anywhere in the world.

3.1.5 **Peripheral Interface Hardware**

The peripheral interface hardware formed the interface between the control application hardware and its software driving running on the host computer. This interface hardware electrically connected to the relays, sensors and motor amplifier of the control application as well as the VMAC network.

This interface hardware contained no application-specific control ability. By making the peripheral interface hardware completely generic the same hardware could be used for all applications demonstrated. Chapter 4 describes the role and function of this device, while Chapter 5 describes the hardware built for the implementation.

3.2 Scope

To keep this research focused on the stated goals the following were outside the scope of this research:

- Wireless communication
- Optimization of control loop dynamics
- Interface hardware not specifically required to meet the research objectives
- Custom machine-mounted user interface hardware

At the outset of this research wireless communication technology had not advanced sufficiently to be a viable option. With the advent of many new standards it is presently more attractive.

The control loops developed to run the demonstration hardware were simple PID control laws with no attention to system modeling or other optimization. The interface hardware designed for the implementation contained the functionality needed for the implementation. The user interface system was focused on web-serving the device web pages. Custom display systems for individual control-applications is left for future work.

3.3 Intellectual Property Protection

The research team filed a patent application to protect the intellectual property developed as part of this research. The patent sought protection for:

- Overall VMAC architecture
- User Interface system

- real-time communication scheduling algorithm

This protection is sought in anticipation of future developments of this technology.

3.4 **Summary**

The developments described in this chapter formulated a system capable of the following:

- Digital control loop operation over standard PC network components
- Automatic device discovery (e.g. "Plug 'n Play")
- Versatile user interface system
- Use of "off the shelf", low-cost, computing and network hardware

Control loops ran over a real-time implementation of Ethernet according to an explicit schedule. Configuration software managed the plug and play interoperability of devices. The user interface system provided for access anywhere in the world by any web-enabled device. This design can easily be implemented using low-cost hardware.

The details of this development is described in Chapter 4.

CHAPTER 4 VMAC CONTROL SYSTEM

This chapter describes, in detail, the development of the five components explained previously and listed below:

- real-time communication environment
- System configuration and management tools
- Device driver architecture
- User interface system
- Peripheral interface hardware

The integration of each of these areas is shown again for the reader's convenience in Figure 13. The VMAC management software manages the configuration of the system as devices are added and removed. The User Interface Manager coordinates all user input and formats data shown on display systems. The peripheral interface hardware provides physical interconnects to the control application's relays, sensors and motor amplifiers.

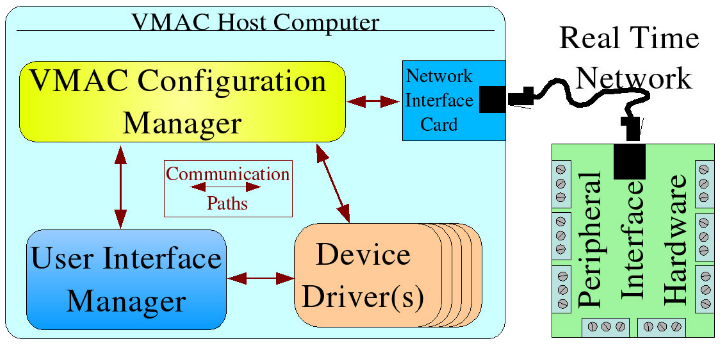


Figure 13: Components of the VMAC system

4.1 Real-time Communication Environment

This section describes the development of this real-time communication environment. The reasons for selecting Ethernet are described, followed by several operational rules that will eliminate collisions on this network to enable isochronous data transport. A simplification to the Ethernet standard is then described that reduces the computational overhead of Ethernet. Finally, a communication scheduler is presented that will manage all communication on the network. This scheduler provides deterministic communication while evenly loading the network.

4.1.1 **Selecting Ethernet**

A communication system for real-time control must meet the following requirements.

- Ability to communicate in real-time
- Communicate over at least 100 ft
- Functional bandwidth sufficient to support 10,000 samples per second

Several common communication standards are compared in Table 1.

Table 1: Most applicable communication standards

Standard	***Media***	***Bandwidth (Mbits/S)***	***Topology***	***Cost***	***Native R/T***	***Distance (ft)***
IEEE 802.3b Ethernet	UTP[1]/Coax /Fiber	100	Serial bus	Low	No	1000
Profibus	RS-485/ Fiber	12	Serial bus/Tree	Medium	Yes	600
Foundation FieldBus	UTP/ Fibre	100	Serial bus	Medium	No	2000
DeviceNet	UTP/ Fibre	0.5	Serial bus	Medium	Yes	1500
LonWorks	Power Line / UTP	1.25	Serial bus	High	Yes	6000

The Ethernet implementation of IEEE 802.3 has several compelling advantages.

- Least expensive
- Readily available communications hardware
- High bandwidth
- Well developed software tools

These advantages make Ethernet a very attractive choice.

4.1.2 Real-Time Implementation of Ethernet

The non-determinism of Ethernet is primarily due to the lack of control over media access. Any device using Ethernet is allowed to transmit at any time. When two devices do transmit simultaneously, the resulting collision causes unusable data necessitating re-transmission. For token-ring or token-bus topologies, which could be

1 Unshielded Twisted Pair

implemented using Ethernet, non-determinism is due primarily to variable response latency in individual nodes.

In [23] Kerkes proposes several rules to ensure determinism when using Ethernet as a control network:

- One device on the network is designated as the host and all others are considered remote nodes
- No remote node may access the network bus until the host sends a request to it
- Remote nodes will have a predetermined amount of time to respond to a controller request

These rules eliminate data collisions by allowing only one device to communicate on the network at anytime under the direction of the host. These rules as applied to the VMAC system can be simplified and restated as follows:

- Don't speak unless spoken to
- Respond within the time allotted

These rules give the host complete control over who transmits on the network. By keeping a schedule that specifies when each device gets use of the network data can be sent reliably to any device on a regular schedule.

4.1.3 **Communication Scheduler**

The communications scheduler developed satisfied four important requirements:

- Schedules communication on the network

- Update schedule to accommodate new devices without disturbing existing devices
- Makes efficient use of available time on the network
- Provide for a trade off in the number of devices being controlled and the device sample rate

Updating the schedule requires the system to build a new communication schedule while the system is in operation. Control processes that are running will not be disturbed by the new schedule.

It is also important for the network to be evenly loaded to make efficient use of the available network bandwidth. This means that there must not be substantial idle periods.

Finally, the scheduler must provided the ability to schedule more devices at a slower rate, or fewer devices at a higher rate.

The scheduling algorithms reviewed in Chapter 2 are not a good fit for the centralized control system described here because they cannot meet one or more of these requirements.

A new scheduler described here meets all four of these requirements. The scheduler will allocate time on the network for each device according to its sample rate.

To implement the rules discussed above, time can be sliced into intervals called time slices. During each time slice, the network is dedicated to communication with a particular device. This is shown in Figure 14.

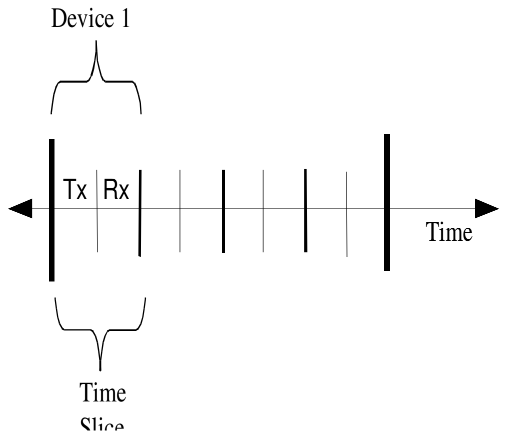


Figure 14: Scheduler allocates each device a periodic time slice

Next a series of time slices comprises a time slot. One time slot is shown in Figure 15. Within each time slot the system may communicate with many devices.

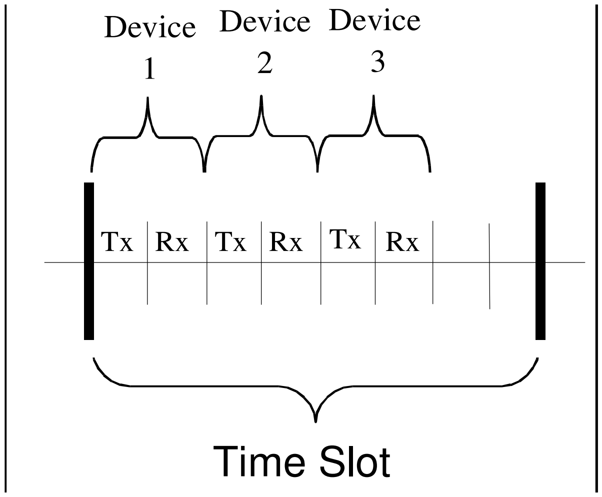


Figure 15: Time slot is composed of several time slices

The duration of the time slot determines the maximum scheduling frequency. As devices request service the schedule is built in terms of 2^n time slots. Where n specifies the sampling period for a device's control loop. For example, a device may request service at $n=0$ and will be scheduled to run every slot period. If a device requests service at $n=1$ it will be serviced every other slot period. Stated more generally, if a device requests service at $n=k$ it will be scheduled to run every 2^k time slots. The largest n defines the length of the scheduling block. This block is repeated indefinitely, or until a new schedule is built. An example is shown in Figure 16.

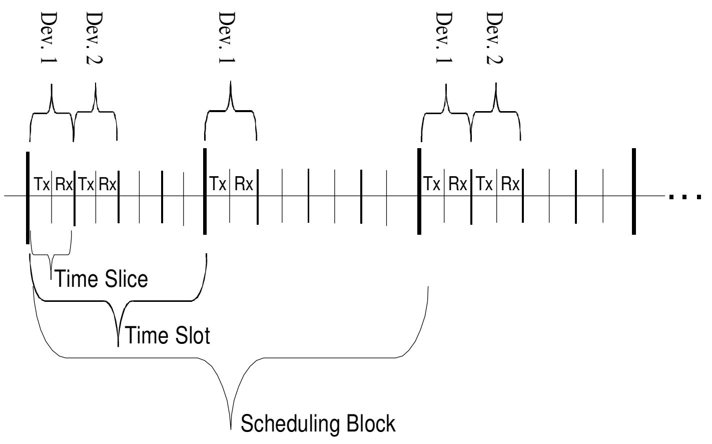


Figure 16: Exponential scheduling algorithm

Figure 16 illustrates the entire scheduling algorithm. Device 1 is scheduled for service at $n=0$. Device 2 is scheduled for service at $n=1$. For this example, the scheduling block contains two periods because the largest *n* for this schedule is 1.

This scheduling method is referred to as a 2^n, or exponential, scheduling algorithm. This scheduler satisfies the functional requirements for the VMAC system. See Appendix I for more detailed analysis of this scheduling algorithm.

A controller of this type having ten slices per slot (1 ms time slot) is illustrated in Table 2. A time slice of 100 µs is a realistic time requirement for the host computer to calculate control loop values. The network bit rate value corresponds to 100BaseT Ethernet. The number of time slices per time slot was chosen as a to allow ten devices to run at 1,000 samples per second. The number of bytes per communication corresponds to the minimum number of bytes that can be sent using a standard TCP/IP protocol.

***Table 2**: Full schedule network utilization*

Time Slice Period (μs):	100
Bytes per communication:	106
% utilization:	16.96%

Table 3 illustrates that the control system described in Table 2, provides a wide control bandwidth. Given the ten slices per slot of this hypothetical controller ten devices could be controlled at the highest frequency of 1khz. At slower control loop sample rates more control loops can be scheduled. Table 3 illustrates this trade off between control loop bandwidth and the number of control loops that can be scheduled using this algorithm.

***Table 3**: Trade off with number of devices and loop frequency*

1 Khz	10
100 Hz	100
10 Hz	1000

The analysis of Table 2 and Table 3 illustrates that there is sufficient communication bandwidth for the controller at hand when using 100BaseT Etherner. The network need only be as fast as the host can calculate and communicate these values.

The scheduler described here eliminates data collisions on the control network. It allows each device to request service and be scheduled with out interrupting the service of other devices. The performance of standard 100 Mbit Ethernet provides sufficient bandwidth for the expected controller performance.

4.1.4 **Reduction of Communication Overhead**

The computation times required to prepare each communication packet must be small compared to a time slice. In preliminary experiments the standard TCP/IP stack implementation, illustrated in Figure 17, the round trip communication time was greater than a 100μs time slice.

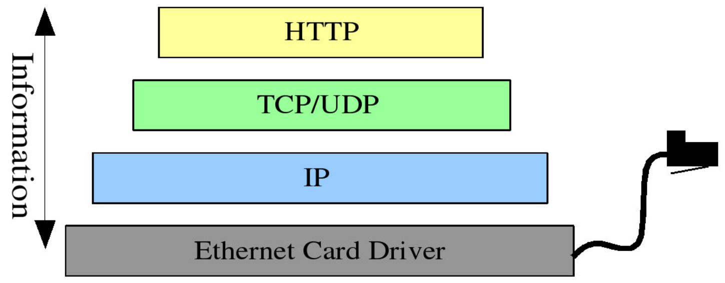


Figure 17: Simplified view of OSI stack model for network communication

The generality and functionality of the application layer (HTTP), transport layer (TCP/UDP) and, network layer (IP) are not needed for this control system.   By eliminating these layers a significant reduction in computational overhead can be realized. This revised communication model is shown in Figure 18.

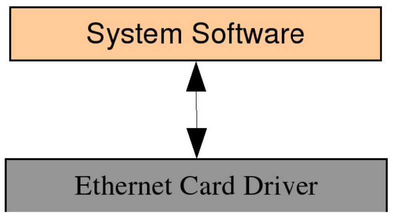


Figure 18: Reduced overhead Ethernet communication for the VMAC system

This modification reduces the minimum number of bytes transmitted on the Ethernet media from 106 to 62, the minimum allowable Ethernet frame size. The resulting VMAC communication packet is shown in Figure 19, where the body contains the application data.

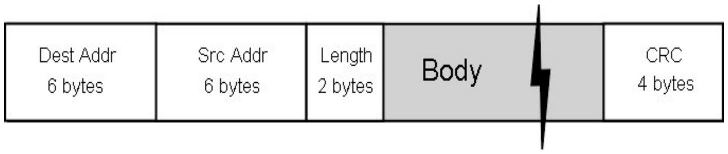


Figure 19: VMAC communication frame with reduced communication overhead

The reduction in software overhead by this modification is noticeable. To verify this, a series of tests were run to measure the time to send a 20 byte payload from one computer to another and back over 100baseT Ethernet. The details of these tests are explained in Appendix III, and the results are show in Table 4. It is significant to note that for this test the total network utilization was less than 20 percent. This implies that the limiting factor in the performance of this control system is the computational speed of the host computer rather than the available network bandwidth.

***Table 4**: Communication savings with Raw Ethernet*

Loop Time With TCP/IP Stack (μs):	114
Loop Time With Modified Stack (μs`	84

This modification to the Ethernet standard allows communication within on time slice.

4.1.5 **Physical Topology**

The VMAC system utilizes a physical star topology shown in Figure 20. The host and all the machines are connected to a network hub, which is the common point of the star. A hub is preferable to the more common switch because it introduces no latency which would significantly degrade the schedule.

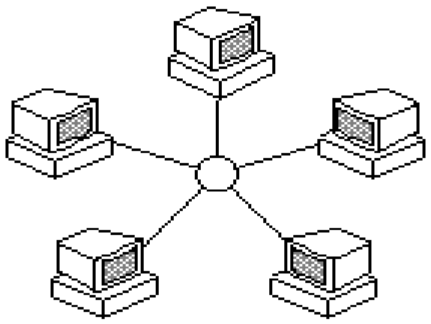


Figure 20: Star network topology

4.1.6 **Logical Topology**

The VMAC control network is logically a bus, as shown in Figure 21. This topology is critical because the VMAC protocol is only sufficient to communicate directly with devices on one bus. This is unique from both the Internet and many commercial networks that are a composite of buses.

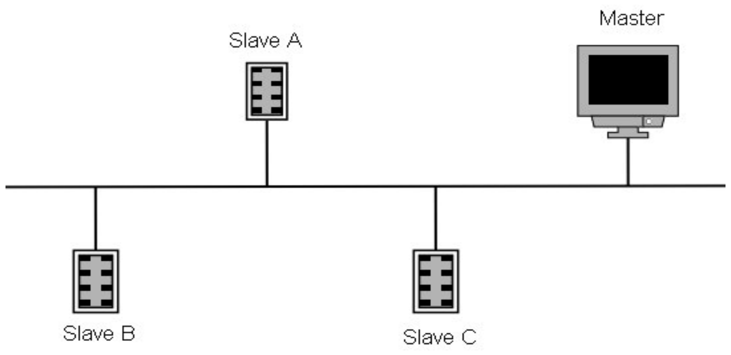


Figure 21: Logical topology of the VMAC network

4.1.7 **Error Handling**

In the event there are problems communicating with peripheral devices the device will be flagged as problematic and its driver unloaded from the schedule. Stated differently, this is a zero-tolerance policy for not meeting a deadline. Likewise, the peripheral interface hardware will shutdown everything it is connected to in the event of a communications failure.

4.1.8 **Communication Environment Summary**

This communication environment provides:

- Real-time communication environment
- Scheduler that ensures determinism
- Low-overhead protocol

4.2 **System Configuration**

The plug and play interoperability of devices required a software set that manages the configuration of the VMAC host, including:

- Device discovery
- Communication between device drivers and the network
- Enforce error policies

4.2.1 **Device Discovery**

The device discovery software reconfigures the VMAC controller when peripheral devices are connected or disconnected from the control network.

4.2.1.1 **Connecting a New Device**

When a new device is connected to the control network four things happen.

- Recognize the device type in a timely fashion
- Load the appropriate driver for device
- Schedule the device for services
- Configure the user interface system to incorporate the device

During system startup the VMAC manager schedules a periodic broadcast query on the network for any new devices. When a new device is connected to the network it responds to this query with the information needed to identify it. The configuration manager uses this to load the appropriate driver software. When the driver is loaded it is scheduled for periodic service at the rate it requests. After the driver is scheduled for service it passes all needed user interface information to the user interface manager, which incorporates the new device into the hierarchy of available UI pages.

There are a few exceptions to this process. In the event that there is no available driver for the new device the device is put into a sleep mode and and error message is sent to the UIM describing the device, and action needed. In the event that the communication schedule has no space left for new devices, then the driver is not loaded and a different error message is sent to the UIM describing the situation.

4.2.1.2 **Disconnecting a Device**

When a device is disconnected from the control network the system four things take take place.

- Recognize absence of the device

- Unload the device's driver
- Rebuild the network schedule
- Display message on the UI system indicating the removal of the device

The host recognizes when a device is removed from the control network when the device does not respond to a request in a timely fashion to communication from the host. When this happens, the driver is flagged as errant and is unload from the schedule. Once unloaded from the schedule all associated processes are ended and a message is sent to the user interface manager indicating the absence of the device. This message is used by the user interface manager to unload all user interface pages associated with the device. As with connecting a device to the control network, disconnecting likewise requires no user input and will not disturb neighboring control processes. The device can be disconnected at any time and the system will reconfigure its self automatically.

4.2.2 **Data Communication**

The VMAC manager processes all communication. The VMAC manager inspects network packet headers to determine where they need to go. When an actuator responds with its current position, the manager determines which driver is associated with that particular device and routes the response accordingly.

When there is user input to the system, the VMAC manager hands the traffic directly to the user interface manager which routes the user input to the proper driver(s). Communication from a device driver marked for a peripheral device is handed directly to the VMAC manager which in turn sends it on the control network. Communication marked for a UI is sent first to the user interface manager which formats it for the UI system.

This communication routing software implements the network schedule and provides a standard device driver interface. When there is a problem the manager can react properly to it by enforcing the error policy.

4.2.3 Error Policy Enforcement

The types of errors that will be handled by the VMAC Manager only involve devices or drivers missing deadlines. In the case of process or application-specific errors, the error handling is left to the driver writer.

4.3 Device Drivers

The device drivers contains the application-specific control and user-interface software. This makes the host controller completely generic and independent of any control application. For example, a metal working punch press could be connected to this control system and its user interface loaded while a clothes washing machine also is connected and running. The two applications have unique user interfaces and control requirements, yet the system is able to control them both. The same system can control different applications because application-specific control and UI software are contained within the device drivers.

4.3.1 Structure of a Driver

Each device driver is an executable file with two main purposes. On startup, the driver gathers, formats and delivers the user interface information to the user interface manager and configures the peripheral interface hardware. Once running, the driver executes the control software for the device, accepts user input and provides feedback to the user about the process being run.

4.3.2 Driver Interfaces

During setup and operation the driver used a series of interfaces with the VMAC system. The interfaces with the driver were formed by passing structured messages between the driver process and desired recipient. When a driver needs to communicate with either the UIM or the VMAC Manager it formulates a message containing a header and data structure. This data structure contains sender-identifying information as well as information germane to the request identified by the header. These interfaces are divided into needed functionality, which include:

- Scheduling
- Network communication
- User interface communication
- Application error handling

4.3.3 **Driver Operation**

During operation the device driver can request a particular service frequency by sending a message to the VMAC manager. This interface allows the driver to dynamically change its service frequency. For example, a device that is not in active use could request to be scheduled at a very low frequency. When a user provides input to this particular driver it could then request to be rescheduled at a higher rate more appropriate for the control requirements of the application.

The interface with the user interface manager allows the driver to provide process feedback to the user as well as handle errors in the control application. For example, if the paper jams on a large paper cutter being controled on this system, the driver can notice the jam and post the appropriate information on its user interface. This allows the user to rectify the problem and resume operation of the device.

4.3.3.1 **Communication Protocol**

The general form of all communication from the driver to peripheral hardware follows a specific ordering. This communication structure is shown in Figure 22 and is encapsulated within the Ethernet data field.

[Op Code][Data Length][Data]

Figure 22: VMAC device communication protocol

The different sections of this protocol are used to navigate a tree structure of commands shown in Figure 23.

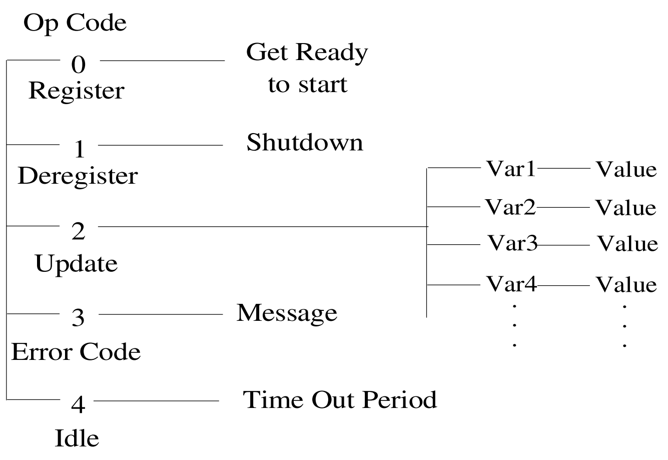


Figure 23: Command tree structure for peripheral devices

Communication from the peripheral interface hardware back to the driver follows the same structure.

4.3.4 **Driver Failure**

Each of the processes of the VMAC system runs within its own process address space. This protects the system in the event of a device driver failure. The failed process will be flagged as errant and an appropriate message will be sent to the user interface manager.

The device driver architecture described above provides a way to package all control and user-interface information that is pertinent to a particular device. This allows the plug and play interoperability of devices as well as control several applications.

4.4 **User Interface System**

The user interface system allows device drivers to display process feedback and accept user input. The system presents information viewable by any modern web browser. This allows any device that is able to run a web browser to function as a UI portal for this system.

The user interface system consists of two major components. The first is the VMAC system, more specifically its user interface manager. The second is a web server connected to both the VMAC control network and the Internet. This system is depicted in Figure 24.

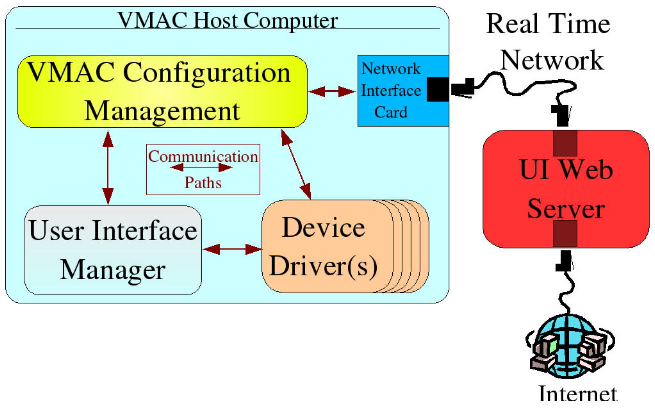


Figure 24: User interface system

An example of this arrangement is depicted in Figure 25.

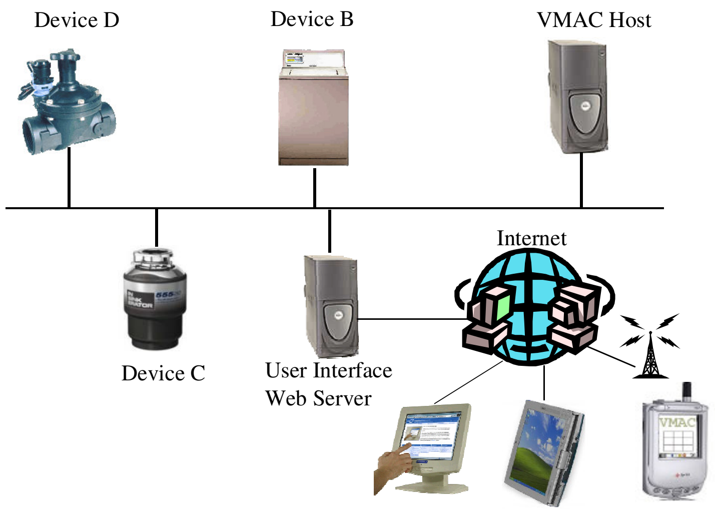


Figure 25: Control network with web server and devices

This architecture provided several key benefits.

- Computational overhead of web serving UI pages is removed from host
- Scalable number of user input/display units
- User interfaces can be remote from the control application
- Inexpensive web interface to the control system

Under this system the host controller was able to preserve the majority of its processor clock cycles for running real-time control loops of connected devices. The host only need format the UI data and hand it to the web server. Additionally, there may be an arbitrary number of web servers, which allows for scalability in the UI system.

An example of what this system could look like on a factory floor would include a wired control network to each of the machines. Internet access could be broadcast wirelessly which would allow access to the control system by hand-held device or other portable or fixed Internet enabled device.

4.4.1 User Interface Manager

The user interface manager routes all UI communication between device drivers and the user interface web server. The functionality needed to do this is divided into three areas:

- Configuration of user interface servers
- Route user input to the proper device driver
- Route device status updates it to the UI web server(s)

The configuration of each new UI server involves scheduling it for service with the VMAC Manager and then getting a copy of each device's UI page to the new server. During the initial transmission of the device drivers UI pages the new server is set to communicate at the maximum rate possible, after which it is set to a low rate.

The VMAC user interface manager sends status updates from a device driver to each UI web server. This ensures that each server has the current UI state for all devices connected to the control network.

When a UI web server is removed from the control network the user interface manager deletes the entry in its update list. The removal of a UI server will not disturb system operation.

4.4.1.1 **Driver/User Interface Manager Communication**

The general form of all communication between the user interface manager and UI servers is the same as was shown in Figure 22, accept for the addition of a process ID field.

The communication protocol allows the user interface manager to configure and maintain the UI servers. This includes loading and unloading device UI pages, updating device UI page states, device errors and rescheduling the server's service frequency.

4.4.2 **User Interface Web Server**

The UI web server is a computer connected to both the Internet and to the VMAC control network. This link allows a user to access the VMAC control system from anywhere in the world via any type of web-enabled device. The major software components of the UI web server include:

- VMAC network manager
- VMAC device UI state manager
- Web server

The VMAC network manager ensures that the UI server communicates on the control network according to the established rules. The VMAC device UI state manager manages the current state of all device's UIs. The web server makes available via the Internet the information from the VMAC device UI state manager .

When ever the UI server receives an update from the host the status updates are stored in a database maintained by the device UI state manager. The web server regularly polls these values to display to the user continuously updated process feedback.

When the user provides input the web server writes the input into the database and marks it to be set back to the host. This information will be incorporated in the next packet the network manager prepares to send to the host.

The user interface system described in this section provides a low cost, scalable web interface with the VAMC control system.

4.5 **Peripheral Interface Hardware**

The peripheral interface hardware, as shown in Figure 26, connected both the control network and the application hardware. This peripheral interface hardware had no application-specific control software loaded on it. This made it generic and enabled its use on any control application.

4.5.1 **Hardware**

The peripheral hardware interface provides the following functionality:

- Set and read digital outputs and inputs
- Digitally control a motor amplifier

These requirements give rise the the structures depicted in Figure 26 and Figure 27.

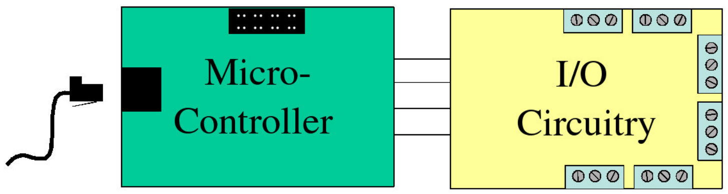


Figure 26: Peripheral interface hardware

Figure 26 depicts the configuration of the hardware when used only for I/O. The micro-controller manages the network communication. This micro-controller sets and reads the state of state of the I/O circuitry through a parallel communication bus (Black lines between micro-controller and I/O circuitry). The I/O circuitry provides the interface between the micro-controller and application hardware and sensors.

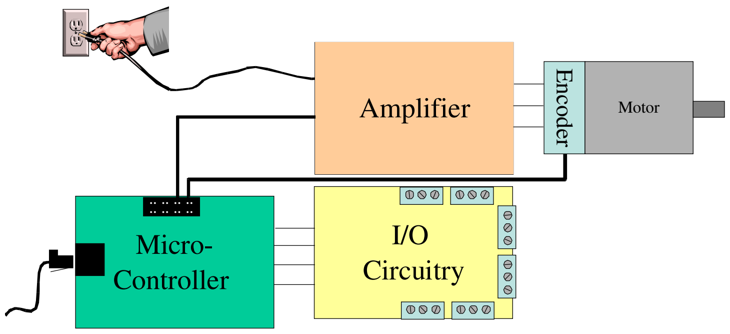


Figure 27: Peripheral interface with amplifier and motor

Figure 27 shows the configuration of the peripheral interface hardware when used to provide feedback control of a servo motor. The mirco-controller digitally communicates the power output values sent by the host to the amplifier (Bold black line between amplifier and micro-controller). The amplifier sets the voltage on the motor windings (Three lines between Amplifier and Motor for three phase motor). Each time the micro-controller formulates a feedback packet it captures the state of the I/O, as well as the motor shaft encoder (Bold black line between the encoder and motor) and sends this back to the host.

The benefits of this modular device architecture are many.

- No application-specific programming needed to use this hardware
- Low cost servo control interfaces
- The same hardware design can be used on a wide variety of control applications

- Many types of motors can easily be used for servo control

The circuitry needed for the peripheral hardware can be packaged in many ways. Figure 28 depicts an example servo motor with associated VMAC interface hardware. This type of servo system can be much less expensive than traditional servos for two reasons. First, the interface hardware for this servo can be generic for all servos. This allows potential economies of scale in producing this control circuitry. Second, the interface circuitry need only have sufficient capability to communicate with the network and amplifier rather than full motion control capabilities. This type of precision servo actuator is projected to cost about $600, a quarter that of the common servo motors commercially sold today. See Appendix V for more details.

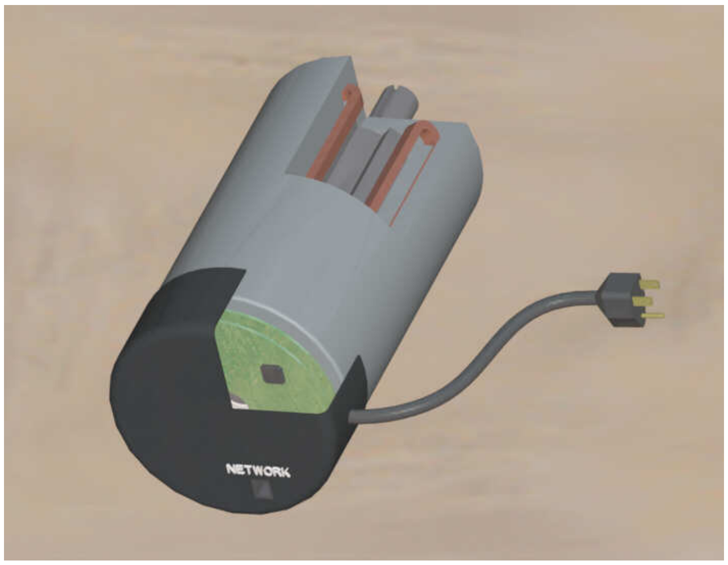


Figure 28: VMAC servo motor

4.5.2 **Operating Procedures**

When a peripheral hardware interface is plugged into the VMAC system and turned on it follows the following procedure

- Wait until the host issues a query for new devices
- Respond with system identification information
- Receive any setup information from the host

These steps facilitate the plug and play interoperability described earlier. When the device is configured and in normal operation it follows the VMAC rules for communicating on the control network.

4.5.3 **Peripheral Device Communication**

Communication with the actuators follows the protocol outlined in section 4.3 under Device Drivers. Principal design issues with this hardware are as follows:

- The device will respond to the host with feedback within 100 µs
- If the hardware losses communication with the host it will shutdown all connected hardware

The response of the hardware within 100 µs allows the use of a slice time of the same duration. This allows many devices to be controlled as illustrated in Table 3 of section 4.1.

4.5.4 **Failure Analysis**

The only error that the peripheral hardware responds to is communication loss. It responds by shutting down all connected hardware and waiting for another discovery packet to be sent by the host.

4.6 **Summary**

This chapter describes low-cost, network-based feedback control system. Principal components include, a real-time communication environment, automatic device discovery, a web-based user interface system and generic peripheral interface hardware. All application-specific software is contained within a device driver, making the system completely generic.

This system meets the objectives of this thesis project as outlined in Chapter 1.

CHAPTER 5 VMAC DEMONSTRATION

A residential home control system was constructed, which included the following:

- Feedback control for AC motors
- Devices requiring state control

5.1 Demonstration Layout

The mock-up kitchen was setup in Provo, Utah on the BYU campus. The conceptual layout of the implementation is shown in Figure 29.

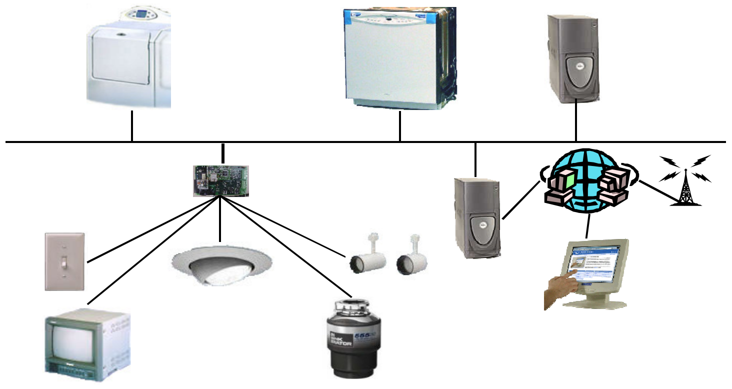


Figure 29: Home automation system including appliances, lights and switches

A peripheral interface controls each device shown in Figure 29. From left to right and top to bottom, this hardware includes:

- Maytag Neptune® clothes washing machines
- Maytag Jetclean II® dishwasher
- Dell OptiPlex running the VMAC host software
- Light switches to provide user input
- Several lights to be turned on and off
- Dell OptiPlex PC running the UI web server software
- User interface computer connected to the Internet
- Wireless Internet access point also connected to the Internet
- Television to be switched on and off
- Garbage disposal

A photo of the kitchen shown in Figure 30.

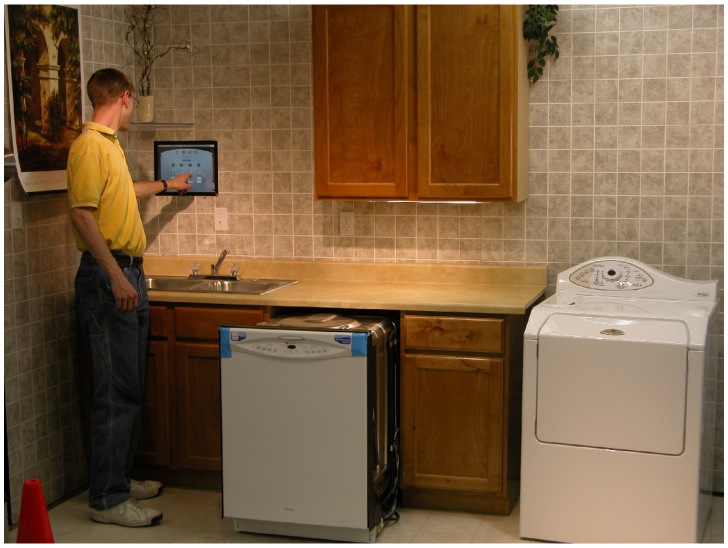


Figure 30: Demonstration home control system

5.2 Communication Environment

Communication between each of the devices depicted in Figure 29 is done by category-5 wire, commercially available connectors and network hubs. This interconnection is depicted in Figure 31 using NetGear brand 10/100 baseT Ethernet hubs.

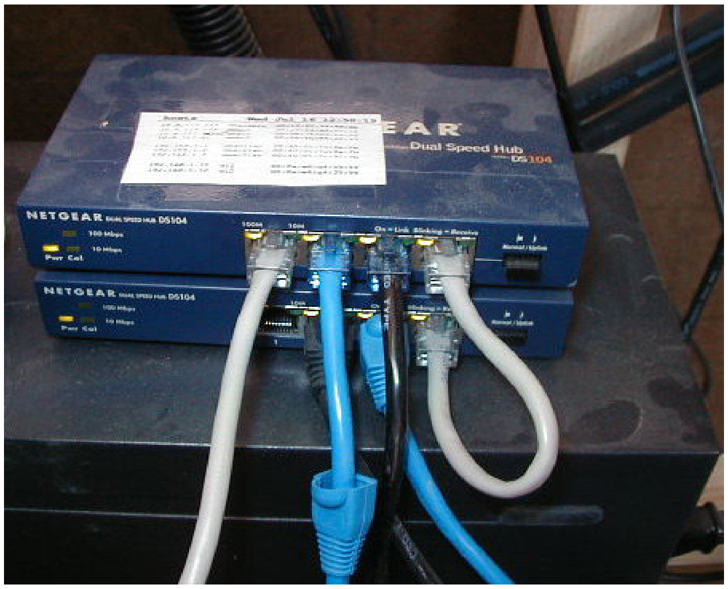


Figure 31: Network communication via standard Ethernet components

The scheduler was setup to run a 10 ms time slice with three slices per time slot. This provided a maximum service frequency of 33 Hz. This frequency was the highest possible given the UI web server communication latency.

5.3 System Configuration Software

The host computer, shown in Figure 29, runs the system configuration software described in section 4.2. The operating system selected to run these processes is QNX. QNX is a miro-kernel real-time operating system that supports the Intel 32 bit processor.

The design of the configuration processes allows for a straight forward prioritization because the execution sequence is explicitly determined. This prioritization is shown in Table 5. This prioritization allows the configuration process to look in the communication schedule to determine which driver to pass device feedback to. The driver begins executing upon receipt of feedback from its peripheral device. If there is an error, such as a missed deadline by the driver, then the configuration process can unload the driver and send the appropriate message to the UI system.

The scheduler used to determine when each of these process can run on the CPU of the host computer must also be chosen. The default QNX scheduling algorithm is First In First Out (FIFO). FIFO scheduling allows the first process that is ready to execute to do so until completed or interrupted by a process of higher priority. FIFO scheduling works well for all tasks except the device drivers. The device drivers require a consistent starting time and allocated run duration. In the event that a device driver takes longer than it should, then it needs to be interrupted so the next driver can start on time. To accomplish this, the device drivers are scheduled using a round robin scheduling algorithm. When using this scheduler a device driver will be interrupted and placed at the end of the list of tasks ready to execute if the device driver exceeds its time allotment. If the driver is late enough that it is not prepared for its next sample period, it will be unloaded and a message indicating so will be displayed on the UIs. This scheduling is also shown in Table 5.

Table 5*: Delineated prioritization ensures proper performance*

Process	***Priority***	***Scheduler***
Configuration	High	FIFO
User Interface Manager	Medium	FIFO
Device Drivers	Medium	Round Robin

The hardware chosen for the VMAC host is a Dell OptiPlex PC with the following features:

- Intel Pentium 4 processor, 1.8 GHz clock rate
- 512 Mb DDR RAM
- 2 standard Ethernet interface cards

5.4 Device Drivers

There were three drivers for this implementation, one for each of the peripheral interfaces. These interfaces drove the following systems:

- Clothes washing machines
- Dish washing machines
- State devices

The driver for the clothes washing machine incorporated feedback velocity control of the basket drive motor, and state control of its other systems. The driver for the dishwasher employed coordinated state control of its system. The driver for the state devices employed uncoordinated state control. Uncoordinated state control means that each of the state devices can be controlled independently from the others.

5.5 **User Interface System**

The significant aspects of the UI system for this implementation includes the following:

- Software run on the host
- Software run on the web server
- Three computers connected to the Internet

5.5.1 **UI System Software**

This implementation used Macromedia, Inc.'s Flash to interactively communicate the graphical UI information for each device driver. Each device driver had one or more web pages containing Flash movies that define the UI for the particular device. The individual Flash movies of each device's web page references the data base maintained by the UI state manager for current values. For example, if a driver were maintaining the temperature of something it would send the current temperature to the UI sever via UIM where it would be stored in the database. The UI for this driver would reference that entry in the database for the current temperature to display.

These web pages are served using a copy of Apache 1.3.21 run in Debian Linux. The database access of the Flash is done by PHP scripting, and is refreshed three times per second.

5.5.2 **UI Hardware**

For the testing there were three clients connected to VMAC UI web server to provide user input to the system. The first two communicated via wired connection with the Internet. One was the wall-mounted computer visible in the mock kitchen. The other was a computer located in Salt Lake City, 40 miles away. The third computer was a hand held PC that communicates via wireless (IEEE 802.11b) connection.

The wall-mounted touch panel shown in Figure 30 has the following features:

- 15 inch Custom Video Display touch sensitive LCD
- AMD processor, 1.1 GHz clock rate
- 100baseT Ethernet interface
- 512Mb RAM

This flat panel PC ran the Windows 2000 operating system and Mozilla Firebird version 0.8 web browser.

The hand-held compact tablet PC ran Windows XP and uses Microsoft Internet Explorer version 6.0 for its web browser.

The operating system and web browser of the computer in Salt Lake City was purposely unspecified.

5.6 **Peripheral Interface Hardware**

The peripheral interface hardware designed and fabricated for this implementation is shown in Figure 32.

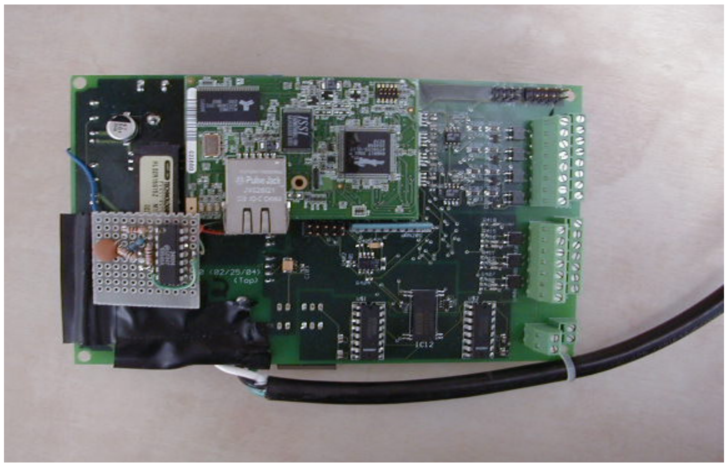


Figure 32: Peripheral interface hardware

The interface hardware consists of a Rabbit Semiconductor RCM3200 core module (middle, upper) and interface circuitry (background). The RCM3200 includes a processor, static RAM, flash RAM as well as a 100baseT Ethernet interface. The Rabbit core also provides RS-485 and pulse-width modulation digital interfaces. The interface circuitry provides 7 digital outputs, 8 digital inputs, 2 analog inputs.

The Rabbit core module was slightly modified to facilitate rapid response to network packets. When a packet is received, the Ethernet transceiver sets one of its status pins to logic high. This pin was connected to a processor interrupt pin. The interrupt service routine (ISR) written for this interrupt causes what ever information is in the send buffer to be sent back to the host. This modification allowed the Rabbit hardware to reliably respond to network communication from the host in less than 100μs.

Both the dishwasher and the clothes washing machine were modified for direct access to their individual actuators, valves, sensors and motors, each of which was connected to the respective inputs or outputs of a peripheral hardware interface. The modification of the clothes washing machine is shown in Figure 33. The wires on the right connect to the sensors, relays and existing motor amplifier needed to operate the washing machine.

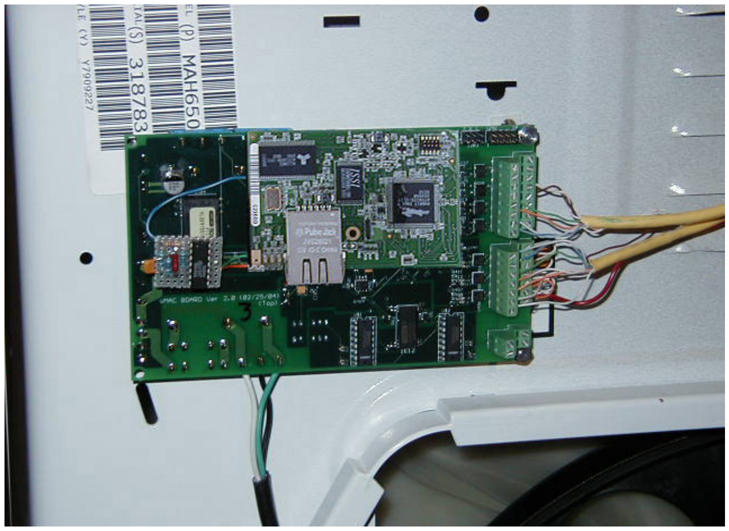


Figure 33: Peripheral hardware interface for the washing machine

The installation for the dishwasher is similar.

The peripheral hardware interface installation for the lights and other devices is shown in Figure 34. Peripheral interface outputs are connected to relays used to switch the devices (small wires on right connector). Two wall-mounted toggle switches are connected to the inputs (black wires on left connector). These switches enable the user to provide input at either the light switch or via the soft interfaces of the UI system.

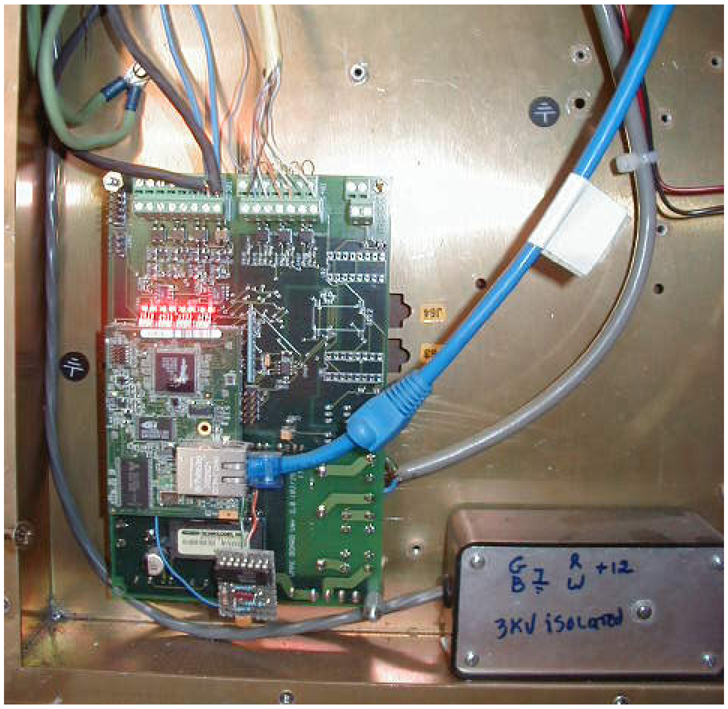


Figure 34: Interface board connected to lights

5.7 **Summary**

The implementation described in this chapter provided a divers set of control requirements. The clothes washer required comparatively high frequency feedback control of 33 Hz while the other devices required state control at lower frequencies of 5 to 20 Hz. This implementation was used to validate the functionality of the VMAC control architecture.

CHAPTER 6 TESTING AND RESULTS

The testing and associated results described in this chapter validate the research objectives of this thesis by demonstrating:

- Digital control loop operation over standard PC network components
- Automatic device discovery (e.g. "Plug 'n Play")
- Versatile user interface system
- Use of "off the shelf", low-cost, computing and network hardware

6.1 Test Procedure

The tests of this system went as follows:

- Start system
- Connect devices to the control network
- Provide user input to control each of the devices connected to the system and observe feedback
- Disconnect devices from the control network one at a time

This test procedure took place in one session and was repeated five times with device connection and disconnection occurring in random orders each time.

The system performed well throughout the test. It discovered the devices and controlled them based on user input from various locations. The control processes were undisturbed by removing or adding devices to the control network.

6.2 System Start

The start up sequence was to first, power up all computers, next start the VMAC process running on the host PC and start the user-interface processes running on the UI server. After loading the URL of the VMAC control web page in a browser on the user-interface computers they showed a blank page as is shown in Figure 35. This indicated there were no devices connected. A check of the error page revealed that there were no errors and the system was prepared for devices to be connected.

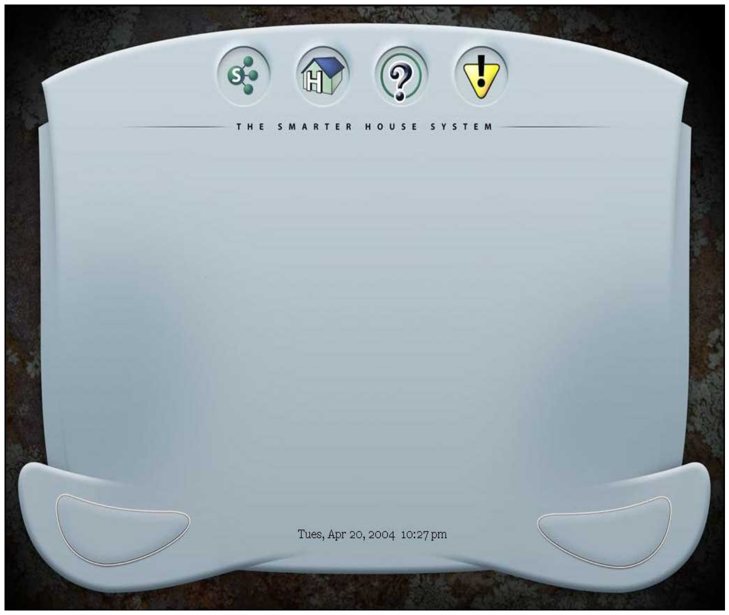


Figure 35: Blank UI indicates no devices connected

6.3 Connecting Devices

As each device was connected to the control network the discovery process took place and the appropriate driver was loaded. This is indicated to the user by the device icon appearing on the user-interface web page as shown in Figures 36, 37, and 38. The devices were connected in a random order. This process required no interaction with the user.

A check of all output from the VMAC processes running on the host and UI server indicated that the each device's driver was properly loaded and the hardware was properly configured for use. This system configuration output is shown in Figure 39.

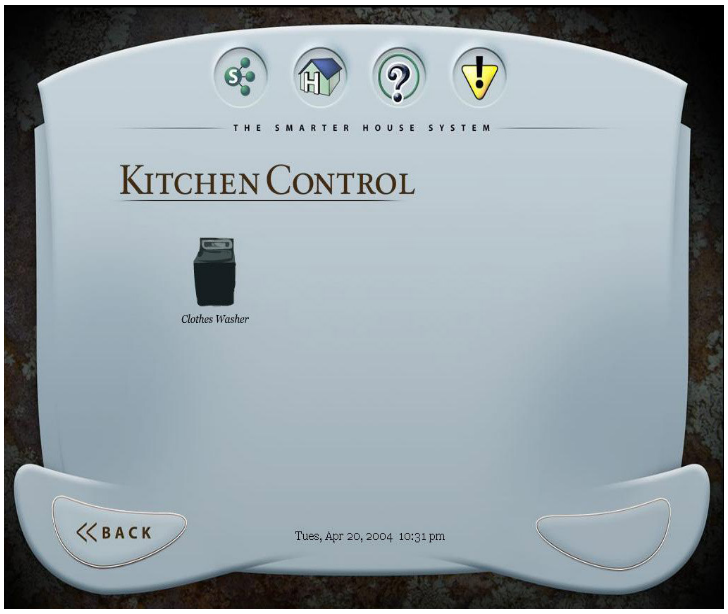


Figure 36: One icon indicates one device connected

Figure 37 and Figure 38 show the remaining devices connected to the control network.

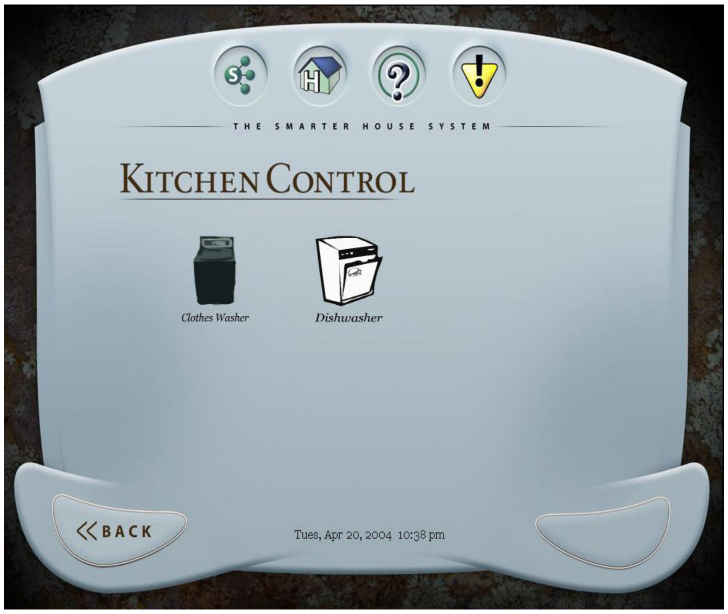


Figure 37: Dishwasher and clothes washer are ready for use

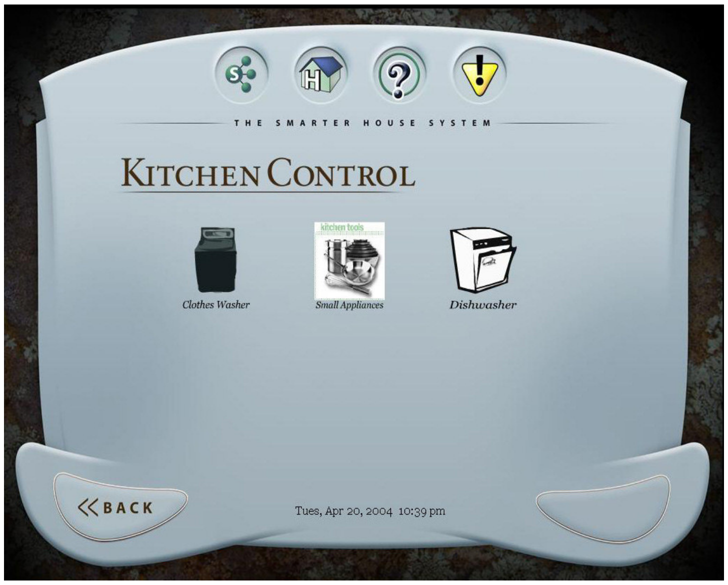
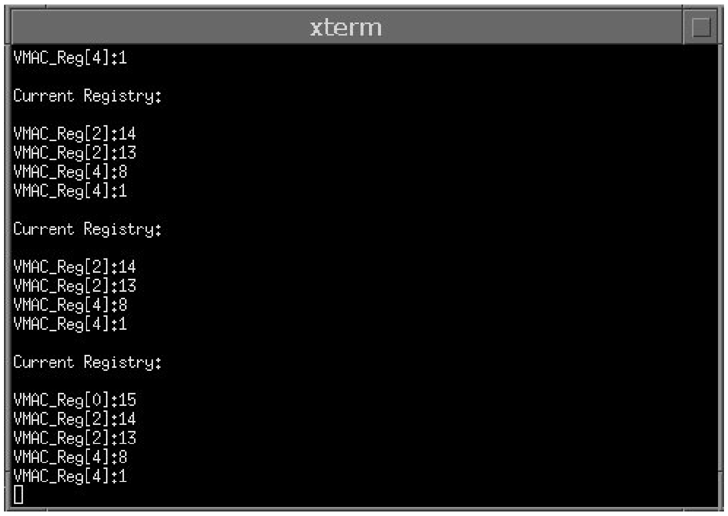


Figure 38: Three peripheral interfaces connected



```
VMAC_Reg[4]:1

Current Registry:

VMAC_Reg[2]:14
VMAC_Reg[2]:13
VMAC_Reg[4]:8
VMAC_Reg[4]:1

Current Registry:

VMAC_Reg[2]:14
VMAC_Reg[2]:13
VMAC_Reg[4]:8
VMAC_Reg[4]:1

Current Registry:

VMAC_Reg[0]:15
VMAC_Reg[2]:14
VMAC_Reg[2]:13
VMAC_Reg[4]:8
VMAC_Reg[4]:1
```

Figure 39: Registry with 5 entries

The lower left of Figure 39 indicates there are 5 entries in the VMAC registry. This corresponds to three peripheral interfaces, the UI web server, and the new device query, which indicates that the discovery procedure works without error.

6.4 User Interaction

Once all devices were connected to the system the functionality of each was tested. The system performed properly in that each device worked as expected either simultaneously or independently. For example, the dishwasher could be run alone, or while the clothes-washer was running.

One interesting limitation came when many clients tried to attach to the UI web server. The UI system worked great with up to four clients connected to the web server. Beyond four the web server was over allocated and the system response was very slow. This is attributed to the computational overhead involved with Macromedia's Flash and also the refresh rate of three times per second for each client.

Another interesting limitation arose from the comparatively slow communication of the web server. The web server was not able to respond reliably to the host in less than 10 ms, which is 100 times slower than the peripheral device's 100 μs response time. This slow response required a time slice period of at least 10 ms which served to limit the host's control bandwidth. This limitation arises from the priority-based scheduling of the standard Linux 2.4 kernel running the web server. Though not as bad as Microsoft Windows, Linux still has no hard upper bound on process response latency.

6.4.1 User interaction Via Touch Screen

Input provided directly via the touch screen illustrated the need for a web design that is easily manipulated by touch input. The individual options of each of the devices shown in Figure 40 have some small radio buttons that are difficult to locate with one’s finger. This caused users the need to touch the screen several times before they could select the option.

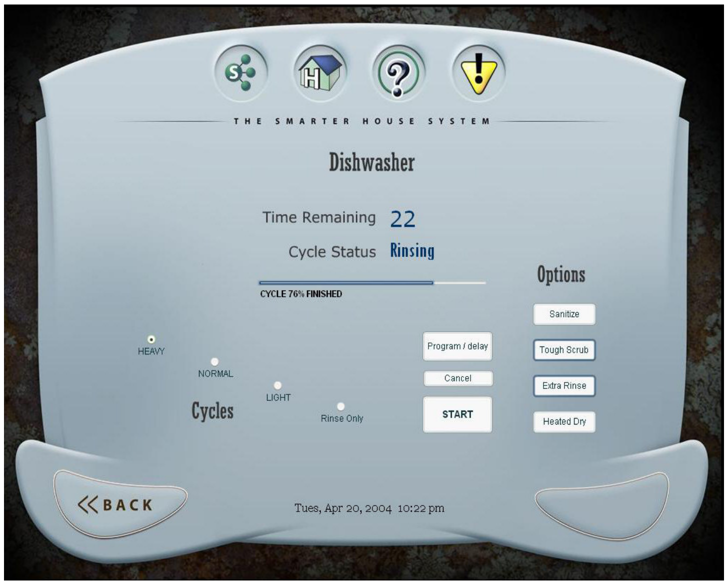
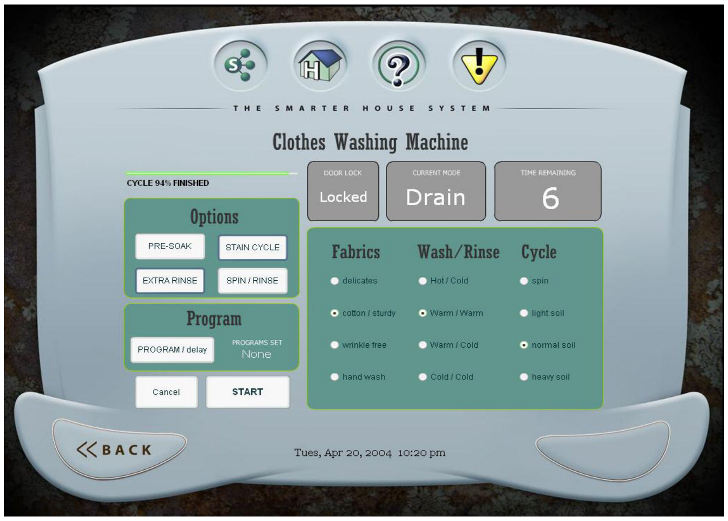


Figure 40: User touches buttons of interest to provide input

Once the options were selected and the process started, the system provided the user process status information in real-time. Figure 40 shows the dishwasher nearing completion of its cycle.

Figure 41 shows the clothes washer running after selecting the highlighted options. Figure 42 shows the status of various lights being controlled.



CYCLE 94% FINISHED

DOOR LOCK Locked

CURRENT MODE Drain

TIME REMAINING 6

PROGRAMS SET None

Fabrics: cotton / sturdy; Wash/Rinse: Warm / Warm; Cycle: normal soil

Tues, Apr 20, 2004 10:20 pm

Figure 41: Process feedback of clothes washer

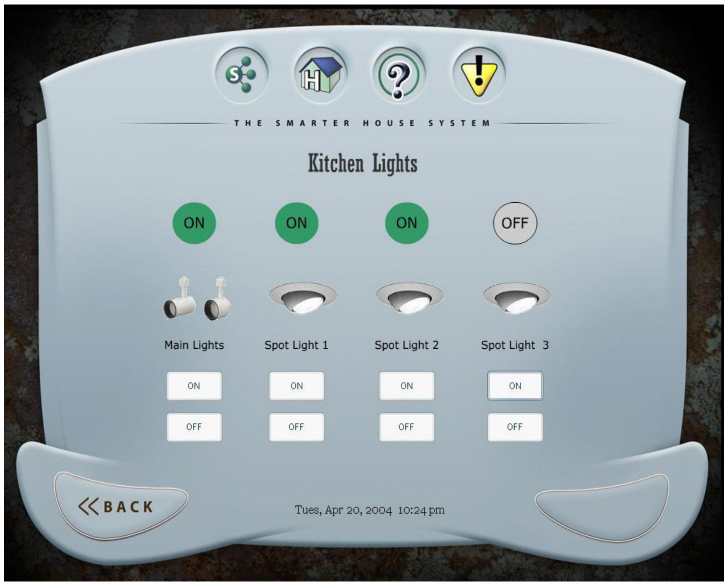


Figure 42: Process feedback of state devices

6.4.2 **User interaction Via Wireless Hand-held Device**

The system worked well with input provided via wireless connection to the tablet PC, however its stylus proved somewhat unintuitive to many users. Again, the web layout is critical to helping facilitate ready and quick interaction with the system.

In the same fashion as the wired touch screen the wireless tablet PC provided real-time information about the status of each device or process. Figure 43 shows the dishwasher running after selecting the highlighted options as viewed from the tablet PC.

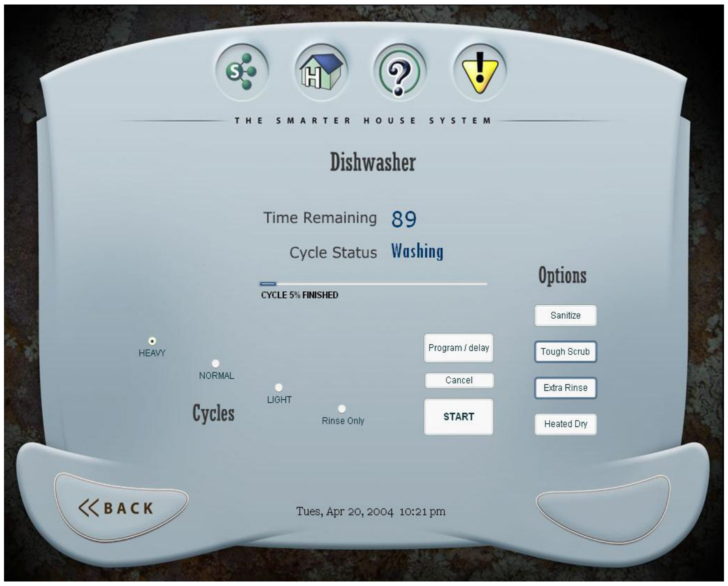


Figure 43: User interface run on hand-held PC

6.4.3 **User Interaction Via Remote Computer**

As part of this test, an individual located in Salt Lake City, 40 miles away, connected to the VMAC control system from his desktop PC via the Internet. From his remote location he was able to control each device on the VMAC system. This included switching the lights, selecting various options for the dishwasher and starting it.

Feedback to the user in Salt Lake City was verified verbally. The user described the same process feedback as was viewed on the UI computer locally. He described the system as responsive with no noticeable latency.

6.5 **Disconnecting Devices From the Control Network**

After establishing the functionality of each device connected to the control system they were disconnected one at a time in a random order. As each device was disconnected from the system the user-interface web page was automatically reconfigured and each of the remaining devices continued to function properly. Disappearance of the device icon as well as a text message on error page indicate to the user the device's removal. The functionality of the remaining devices was established by providing input to each. Figure 44 shows the state device interface is still connected after the dishwasher and clothes washer have been disconnected from the control network.

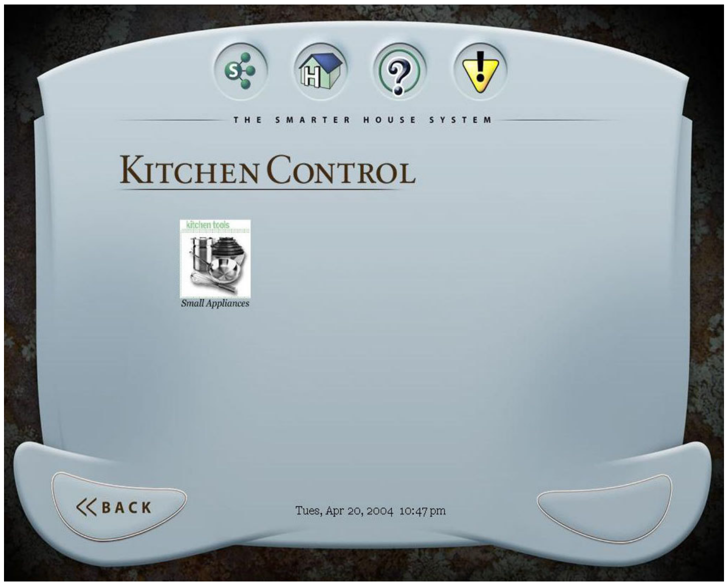


Figure 44: Only the state device interface is still connected

6.6 **Feedback Control**

The various feedback control requirements of this system functioned well.

The coordinated state control and velocity control loop used to run the main motor of the clothes washing machine functioned as expected. The various state devices of the washing machine operated at their appointed time in the cycle. For example, initially the water valves opened according the user selected criteria of hot or cold, next the main motor ran followed by the drain pump motor, etc. The driver was able to control the main basket motor velocity through two speeds. The first was a slow speed of about 45 RPM for agitation. After a set length of time the driver brought the basket speed up to a water extract velocity of about 400 RPM. It held this velocity for the balance of the spin cycle. The safety systems coded into the driver functioned throughout the operation of the cycle. At any point in the cycle if the user manually unlocked and opened the front door of the clothes washer the driver shut everything off and waited till the door was closed before locking the door and resuming. Likewise, if the user canceled the cycle the cycle was stopped and the water drained from the drum of the washing machine.

The coordinated state control used for the dishwasher likewise performed as expected. After the user started a cycle, each of the systems ran in proper succession. The water fill valve opened until the wash chamber was sufficiently filled with water before turning on the main wash motor. The wash motor ran for a set duration and switched off when the drain motor turned on. This process repeated through the wash and rinse portions of the main cycle. The safety systems continued to function properly as well. At any point in the cycle if the user opened the front door of the dishwasher the driver shut everything off and waited till the door was closed to start again.

The uncoordinated state control of the lights, television, and garbage disposal also performed as expected. A user could provide input to the system either from the user interface system or from the traditional light switch and the proper on/off state of the device was displayed on the user interface. When using the wall switch to turn on a state device a slight delay was noticed from user input to display of state. This is attributed to the 3 hz update rate of the UI system.

Another simple test was performed without the UI Web server connected to the control network to determine the maximum control bandwidth. All of the devices of the system behave well down to a time slice period of 100 μs with ten slices per slot. This demonstrates that device service frequencies for real-time nodes on the control network are quite feasible at frequencies of 1 kHz and potentially faster.

6.7 **Summary**

The VMAC control methodology provides a low-cost, easily configured feedback control system that is independent of application. The implementation tested and described here illustrates several key abilities of this technology.

This architecture supports plug and play interoperability. In the same way that a desk top PC discovers and configures itsself for use of a new document printer, the clothes washing machine, dishwasher or state device interface were discovered when connected to the control network and the system automatically configured itsself.

The ability to perform feedback control over a network is also evident in this control methodology. The feedback requirements ranged from user feedback of state devices to a velocity control loop used to to run the clothes washing machine. In each of these cases there was prompt display of feedback to the user as well as proper operation of the velocity control loop.

The VMAC control methodology supports a flexible UI system. A user can provide process input while directly at the machine, from a wireless hand-held tablet PC or from a remote location via the Internet.

The low cost nature of this control methodology was demonstrated by the hardware required for this implementation. The implementation required two low-cost PCs, cat-5 wire, two Ethernet hubs and three peripheral interfaces. The peripheral interfaces were the most expensive part of this implementation. In volume production, the cost of these interfaces is expected to be minimal.

CHAPTER 7 RECOMMENDATIONS

Throughout the development of the VMAC control methodology, several areas for continued research emerged. This chapter briefly describes recommendations for continued research.

7.1 Real-Time Communication Environment

Improvements to the communication environment include the following:

- Higher utilization of the network media
- Use of wireless media

The current limiting constraint in the network utilization is the processing speed of the host computer. When using 100baseT Ethernet there is sufficient bandwidth that potentially more than one host computer could be connected to the same network. This is especially true with the new 1000baseT Ethernet. This potentially might involve a technique for interlacing control communications between the two or more host computers in a effort to prevent communication collisions. If successfully implemented, this technique would provide a massive increase in available control bandwidth with comparatively small marginal cost.

Research into the use of wireless media would provide much greater flexibility in implementing this control technology in existing structures. This research would likely involve an detailed study of wireless communication technologies that have a reasonable loop communication bandwidth. The reliability and latency of communication from host to device and back to host is the critical element to understand for a wireless design to be successful.

7.2 System Configuration

The main enhancements to the system configuration processes running on the VMAC host involve techniques for improving reliability. One such is the High Availability Manager (HAM) available in the QNX operating system. The HAM of QNX provides the ability to manage processes from a high level as if a user were present monitoring the computer. For example, if a process fails, a preprogrammed HAM can respond by restarting processes in proper sequence. A HAM could also be set to use different, available hardware in the event of a hardware failure.

Another technique for improving the reliability could be to utilize the process shadowing capabilities of QNX. A shadow process is dormant until the failure of the primary process. At this point the shadow seamlessly becomes the primary process to avoid a denial of service.

7.3 Device Drivers

The significant improvement in the device drivers include:

- Methodology for controlling an application with multiple peripheral interfaces
- Greater abstraction of the interface overhead
- Developing a script interpreter

The ability to control multiple peripheral interfaces from one driver would provide greater flexibility in precision control applications such as metal working milling machines. By incorporating a greater abstraction in the device driver interface more of the overhead associated with its setup could be obscured from the driver writer. A script interpreter would allow a high-level text file to be used to describe the needed control dynamics rather than a fully executable file written in C or C++.

7.4 User Interface System

Several improvements to the UI system would primarily facilitate lower hardware cost and lower computational overhead in its use. Some research areas include:

- Graphical UI system not based on Macromedia's Flash
- Develop UI web server with response latency of about 100 μs
- Develop UI system that runs on the same hardware as the real-time control
- Develop web security system

The computational overhead associated with Macromedia's Flash lends it to not being a good UI methodology. Research could be done to find a web language or browser plug-in to accomplish the same graphical display of device information.

The comparatively slow network response performance performance of the Linux-based UI web server would be impractical for a commercial implementation. Developing a web server that could respond on the control network quickly could involve writing a kernel module to specifically manage the control network traffic, or potentially developing a new network interface card.

Designing the UI system to run on the VMAC host along side the control hardware would further reduce the cost. This development could involve exploration of symmetric multi-processing (SMP) the control and UI processes across two system processors. If high control bandwidth isn't a significant need the UI system and control processes could be run on one CPU given some attention to timing and prioritization.

Further work in web security policy for the UI system is also needed. This control policy development would include exploration into current web security techniques.

7.5 **Peripheral Hardware**

The peripheral hardware could be improved by developing a more modular design. Such a design might include replacing the Rabbit module with its component pieces on a printed circuit board. This would more easily facilitate experimentation with different networking standards.

7.6 **Summary**

There are many improvements that could be made to the VMAC control methodology. These improvements focus on reducing cost and improving functionality.

CHAPTER 8 CONCLUSIONS

The work of this thesis proposes solutions the three challenges of automation. These challenges are:

- Reliable feedback control
- System for user input
- Method for coordinating systems

The VMAC provides convenient solutions to these challenges by using a direct control paradigm. Digital feedback control over a LAN is demonstrated. The host computes and communicates control values for each device over a real-time network. The user interface system provides flexible venues for providing user input. The user is able to provide input directly at the control application hardware, via touch panel, or via the Internet from anywhere in the world. This system provides an explicit venue for coordinating independent systems being controlled. The device driver for each of the systems being controlled can have one or more interfaces for communicating with other drivers. These solutions are done in a way that allows the use of low-cost hardware and is easy to setup and use.

The progression in control methodology illustrated in this thesis mimics the progression made in early document printers for personal computers. Early document printers employed an indirect control methodology that required intermediate control hardware, was difficult to setup and expensive to purchase. Modern PC device control methodologies allow a user to purchase a printer from a variety of manufacturers and connect it to a communication bus such as USB whereupon the system automatically configures its self. This modern document printer methodology provides a low-cost, flexible control system that is convenient to use.

Tradition automated control systems differs from the VMAC methodology by employ hardware for each stage of a control system. It is common for each item of today's manufacturing equipment to be controlled locally by its own digital signal processor with a PC dedicated to its user interface while a remotely located PC coordinates the machine with others. This arrangement has limited flexibility, provides no standard way to coordinate independent systems and is inherently expensive to install and maintain. The VMAC system overcomes these drawbacks.

The principle contributions of this thesis are as follows:

- Digital control loop operation over standard PC network components
- Automatic device discovery (e.g. “Plug 'n Play”)
- Versatile user interface system
- Use of “off the shelf”, low-cost, computing and network hardware

The home control implementation of this technology demonstrated each of these contributions.

The use of automatic control systems will continue to expand in the future. The VMAC control methodology facilitates this expansion into new and unexplored uses for automatic control.

REFERENCES

[1] Bonuccelli, M., and Cló, M. (1999). EDD Algorithm Performance Guarantee for Periodic Hard-Real-Time Scheduling in Distributed Systems. 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing. 668-677.

[2] Bosley, J. (2000). A Modular Open-Architecture Controller for Direct Machining. Brigham Young University. Dept. of Mechanical Engineering. Provo: Utah.

[3] Branicky, M., Phillips, S., and Zhang, W. (2000, June). Stability of Networked Control Systems: Explicit analysis of Delay. Proceedings of the American Control Conference. 2352-2357.

[4] Branicky, M., Phillips, S., and Zhang, W. (2002). Scheduling and Feedback Co-Design for Networked Control Systems. Proceeding of IEEE Conference on Decision and Control, December 10-13, 2. 1211-1217.

[5] Brockett, R. (1997). Minimum Attention Control. Proceedings of the 36th Conference on Decision and Control. 2628-2632.

[6] Brooks, P. (2001, October) Ethernet/IP: Industrial Protocol. IEEE Emerging Technologies and Factory Automation. 505-514.

[7] Cisco Networking Academy Program: First-Year Companion Guide. (2002). Indianapolis: Cisco Systems.

[8] Cisco Networking Academy Program: Second-Year Companion Guide. (2002). Indianapolis: Cisco Systems.

[9] Cuco, L., Kocik, R., and Sorel, Y. (2002). Real-time scheduling for systems with precedence, periodicty and latency constraints. <http://www-rocq.inria.fr/~cucu/english.htm>. (2004, February 2).

[10] Department of Defense. (1978, September 21). Digital time Division Command/Response Multiplex Data Bus: MIL-STD-1553B. Washington, DE: U.S. Government Printing Offcie.

[11] Echelon Corporation. Interoperable Networked Control for Rail Transit Systems with LonWorks ® Networks. <http://www.lonmark.org/solution/trans/railtrn.pdf> (2003, July 18).

[12] Franklin, G., Powell, J., and Emami-Naeini, A. (1995). Feedback Control of Dynamic Systems. (3rd Edition). New York: Addison-Wesley.

[13] Franklin, G., Powell, J., and Workman, M. (1998). Digital Control of Dynamic Systems. (3rd Edition). New York: Addison-Wesley.

[14] Gerhart, J. (1999). Home Automation & Wiring. New York: McGraw-Hill

[15] Ghimire, G. (2000). Specification and Application of a Digital Control Interface. Brigham Young University. Dept. of Mechanical Engineering. Provo: Utah.

[16] Grimble, M. (2001). Industrial Control Systems Design. West Sussex, UK: John Wiley & Sons.

[17] Haines, R., and Hittle, D. (2003). Control Systems for Heating, Ventilating and Air Conditioning. (6th Edition). Dordrecht, Netherlands: Kluwer Academic

[18] Honeywell. (2004). FutureSmart Product Catalog 2004. <http://www.futuresmart.com/benefits/downloads/catalog.jsp> (2004, June 19).

[19] Hristu-Varsakelis, D. (2000). Interrupt-based feedback control over a shared communication medium. Proceedings of the 41st IEEE conference on Decision and Control. 3223-3228.

[20] Hristu-Varsakelis, D. (2001). Feedback Control Systems as Users of a Shared Network: Communication Sequences that Guarantee Stability. Proceedings of the IEEE conference on Decision and Control, 4. 3631-3636.

[21] Humpleman, et al. (2002). U.S. Patent No. 6,466,971. Washington, DC: U.S. Patent and Trademark Office.

[22] Kelling, N., and Heck, W. (2002, March). The BRAKE Project - Centralized Versus Distributed Redundancy for Brake-By-Wire systems. SAE 2002 World Congress. Presentation 2002-01-0266.

[23] Kerkes, J. (2001, February). Real-Time Ethernet. Embedded Systems Programming. <http://www.embedded.com/story/OEG20010221S0086> (2003, July 18).

[24] Kim, T., Shin, H., and Chang, N. (1998). Scheduling Algorithm for Hard Real-Time Communication in Demand Priority Network. Proceedings of the 10th Euromicro Workshop on Real-Time Systems, June, Berlin, Germany. 45-52.

[25] Kinetix. The New Science of Integrated Motion. (2003, June). <http://www.ab.com/motion/kinetix/MOTION-BR001C-EN-P.pdf>. (2004, February 2).

[26] Kooperman, P., and Upender, B. (1994, November). Communication Protocols for Embedded Systems. Embedded Systems Programming, (11) , 46-59.

[27] Lee, S., Lee, S., and Lee, K. (2001). Remote Fuzzy Logic Control for Networked Control System. Proceedings of the 27th Annual Conference of the IEEE Industrial Electronic Society. 1822-1827.

[28] Lewis, F. (1992). Applied Optimal Control and Estimation. Englewood Cliffs, N. J.: Prentice Hall.

[29] Liu, W., Liu, X., Zeng, Y., and Han, H. (2001). The Network Based IMS. Computer-aided Production Engineering CAPE 2001. 139-142.

[30] Loundsbury, B. (2001, May 21). Ethernet: Surviving the Manufacturing and Industrial Environment. <http://www.ab.com/networks/enetpaper.html> (2004, February 2).

[31] Lucas, et al. (2003). U.S. Patent No. 6,505,087. Washington, DC: U.S. Patent and Trademark Office.

[32] McBride, C. (2002). An Open Architecture for Direct Machining and Control. Brigham Young University. Dept. of Mechanical Engineering. Provo: Utah.

[33] Montestruque, L., and Antsaklis, P. (2001). Model-Based Networked Control Systems –Stability. ISIS Technical Report, University of Notre Dame, ISIS-2002-001.

[34] Newman, M. (1994). Direct Digital Control of Building Systems. New York: John Wiley and Sons.

[35] Novacek, G. (June, 2003). Time-Triggered Technology. Circuit Cellar. 50-59.

[36] Park, H., Kim, Y., Kim, D., and Kwon, W. (2002, May). A scheduling Method for Network-Based Control Systems. IEEE Transactions on Control Systems Technology. 318-330.

[37] Rehg, J., Swain, W., Yangula, B., and Wheatman, S. (1999, November). Fieldbus in the Process Control Labortory- Its time has come. 29th ASEE/IEEE Frontiers in Education Conference, San Juan, Puerto Rico, Session 13b4. (12-17) New York: IEEE Press.

[38] Samaranayake, L., and Alahakoon, S. (2002, September). Closed loop Speed Control of a Brushless DC Motor via Ethernet. <http://www.ekc.kth.se/eme/publ/pdf/lilantha_sanath_colombo2002.pdf> (2004, February 20).

[39] Schiffer, V. (2001, October). The CIP Family of Fieldbus Protocols and its Newest Member – Ethernet/IP. IEEE Emerging Technologies and Factory Automation. 377-384.

[40] Stankovic, J., Lu, C., Son, S., and Tao, G. (1999, June). The Case for Feedback Control Real-Time Scheduling. Proceedings of the 11th Euromicro Conference on Real-Time Systems. 11-21.

[41] The Institute of Electrical and Electronics Engineers, Inc. (2002, March 8). 802.3: IEEE Standard for information technology- telecommunications and

information exchange between systems- local and metropolitan area networks-specific requirements. New York: IEEE Press.

[42] Törngren, M. (1995) A perspective to the Design of Distributed Real-Time Control Applications based on CAN. In Proceedings of 2nd International CiA Conference, 3-4 October. London-Heathrow.

[43] Tovar, E., and Vasques, F. (1999, December). Real-Time Fieldbus Communications Using Profibus Networks. IEEE Transactions on Industrial Electronics. 1241-1251.

[44] Underwood, C. (1999). HVAC Control Systems: Modeling, analysis and design. New York: E & FN Spon.

[45] VanKampen, L. (2001, October). Integrated Motor Control – The Next Generation of Brushless Motor Technology. <http://www.agile-systems.com/images/Integrated%20Motor%20Control.pdf>. (2004, February 2).

[46] Walsh, G., and Ye, H. (2001, February). Scheduling of Networked Control Systems. IEEE Control Systems Magazine. 57-65.

[47] Walsh, G., Beldiman, O., and Bushnell, L. (2001, July). Asymptotic Behavior of
Networked Control Systems. IEEE Transactions on Automatic Control. 1093-1097.

[48] Walsh, G., Ye, H., and Bushnell, L. (1999, June). Stability Analysis of Networked Control Systems. Proceedings of the American Control Conference. 2876-2880. Retrieved July 18, 2003 from IEEE Xplore database.

[49] Walsh, G., Ye, H., and Bushnell, L. (2002, May). Stability Analysis of Networked Control Systems. IEEE Transactions on Control Systems Technology. 438-446.

[50] Wright, P., Pavlakos, E., and Hansen, F. (1991). Controlling the Physics of Machining on a New Open-Architecture Manufacturing System. Design, Analysis, and Control of Manufacturing Cells. 129-144.

[51] Xie, L., Zhang, J., and Wang, S. (2002). Stability Analysis of Networked Control System. Proceedings of the First International Conference on Machine Learning and Cybernetics. 757-759.

[52] Zhang, W., and Branicky, M. (2001). Stability of Networked Control Systems With Time-Varying Transmission Period. Allerton Conference on Communication, Control, and Computing. Retrieved June 6, 2003 from <http://dora.cwru.edu/msb/pubs.html>.

[53] Zhang, W., Branicky, M., and Phillips, S. (2001, February). Stability of Networked Control Systems. IEEE Control Systems Magazine. 84-99.

[54] Zhou, M., and Venkatesh, K. (1998). Modeling, Simulation, and Control of Flexible Manufacturing Systems. Singapore: World Scientific.

[55] Zuberi, K., and Shin, K. (1996, November). Real-time decentralized control with CAN. Proc. IEEE Conference on Emerging Technologies and Factory Automation. (93-99).

APPENDIX

APPENDIX I:
A SCHEDULING ALGORITHM FOR OPTIMAL EVEN DISTRIBUTION OF EXPONENTIALLY RELATED, PERIODIC REAL-TIME TASKS

Adapted From a Course Paper By

Daniel Thompson

24th April 2003

By

Michael Baxter

LIST OF FIGURES

INTRODUCTION

This document describes an algorithm for scheduling real-time, periodic tasks that have an exponential relationship between between sample periods. Presented here is a proof that the optimal layout is possible, and the nature of the proof will indicate the algorithm that can perform such scheduling.

DEFINITIONS

- **Optimal Even Distribution**: The average loading on the network is constant
- **Exponentially Related:** The relationships of task service frequencies is a power of 2
- **Task**: Required computation
- **Time Slice**: Shortest division of time used in this scheduling algorithm
- **Slot**: Period of time defined by a set number of time slices
- **Scheduling Block**: Period defined by the slowest task service period

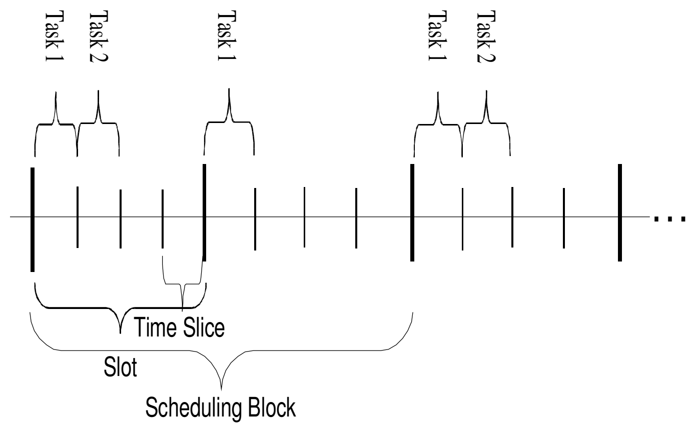


Figure 1: Definitions of scheduling algorithm terminology

ALGORITHM

Definitions of scheduling algorithm terminology For a given set of tasks, who's service frequencies are a power of two of the base frequency, there exists a scheduling algorithm that allows for optimal even scheduling. The optical even schedule provides that the fullest slot have no more than one more device than the emptiest slot.

Task scheduling is done in terms of the exponent *n,* where *n* corresponds to the frequency at which it is scheduled by the relationship of equation 1.

$$f = F * 2^n \tag{1}$$

where *f* is the actual service frequency of the task, *F* is the slot frequency, and *n* is the scheduling level of the task.

An example of the task set A, B, C, D is as follows. Task A, scheduled at $n=0$, has frequency of once every slot. Task B, scheduled at $n=1$, has a task frequency of once every two slots. Task C, scheduled at $n=2$, has a task frequency of once every four slots. Task D, scheduled at $n=k$, has a task frequency of once every 2^k slots.

Tasks are scheduled in ascending order of *n.* As tasks are scheduled they are registered for a particular slot.

PROOF

Hypothesis P(n): For a set of tasks with a maximum task level *k*, it is possible to schedule all tasks such that the fullest slot has no more than one more task scheduled than the emptiest slot.

- Base case: $n = 0$

A set of tasks with maximum task level 0 is a set of one or more tasks that need to be run once every slot, as shown in Figure 2. In order to schedule the set of tasks for execution, we must consider a scheduling block of one slot. Every task in the set is registered for the slot, and since it is the only slot, the emptiest and fullest slot are the same slot. Thus P(0) clearly holds.

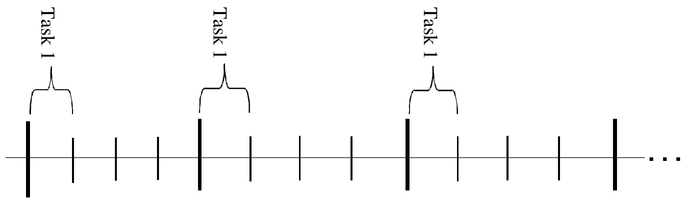


Figure 2: $n = 0$ devices scheduled

- Base case: $n = 1$

A set of tasks with maximum task level 1 consists of zero or more tasks that need to be run in every slot and at least one or more tasks that needs to run in every other slot. In order to schedule this set of tasks for execution, we consider a scheduling block of two slots. First, we register the $n = 0$ tasks that need to be run every slot for both slots in the scheduling block. When we have registered all level 0 tasks, the number of tasks registered for each slot is the same.

Next, we consider the remaining level 1 tasks. If there is only one level 1 task, we can register it for whichever slot, as shown in Figure 3. The number of tasks registered for that slot will be one more than for the other slot, and the hypothesis holds. If there are two level 1 tasks, then we can register the first for slot 1 and the second for slot 2, and we are done. The slots will each be registered for the same amount of tasks, and the hypothesis holds. Additionally, we note that we are now in the same position as when we finished scheduling all the level 0 tasks--each slot is registered for the same number of tasks. If we add more level 1 tasks, we can consider them in groups of two and simply register them in the same fashion as the first two level 1 tasks. Thus, there will never be a difference greater than one between the number of tasks registered for the two slots; P(1) holds.

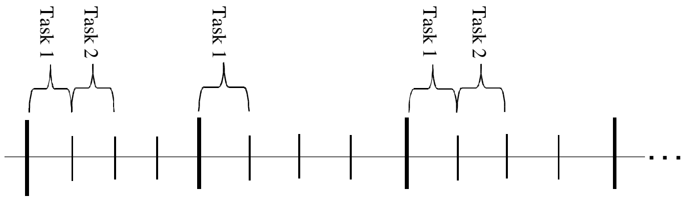


Figure 3: $n = 1$ devices scheduled

INDUCTIVE HYPOTHESIS: ASSUME THAT P(K) IS TRUE FOR SOME K; K>0.

IF P(K) IS TRUE, THEN P(K+1) IS TRUE.

Consider a set of tasks with maximum task level $k+1$. To schedule this set of tasks, we need to consider a scheduling block of 2^{k+1} slots.

First, all the tasks are scheduled up through level *k*. This produces a filled scheduling block of 2^k slots, and we know by the inductive hypothesis that the maximum difference between the number of tasks registered for the fullest and emptiest slot is 1. We now work with a scheduling block of $2^{k+1}=2^k+2^k$ slots, so the first 2^k slots will consist of this smaller scheduling block, and the second 2^k slots will be exactly the same as the first 2^k slots. Since the two halves of the larger, 2^{k+1} slots scheduling block are identical, the emptiest slot and fullest slot for the whole scheduling block is the same as for the two 2^k halves, and the hypothesis holds.

Lastly, the level $k+1$ tasks must be scheduled. Since each level $k+1$ task needs to run once every 2^{k+1} slots, we can pick one slot in our scheduling block of 2^{k+1} slots for each to run in. If, for each level $k+1$ task we need to register, we pick the emptiest slot, and ties can be resolve randomly, these tasks will be spread out such that there will never be a difference greater than one task between the fullest and emptiest slots. Thus, the hypothesis holds for $P(k)$, and it also holds for $P(k+1)$.

Therefore for all $n>0$, using this exponential scheduling algorithm, the maximum difference between the fullest and the emptiest slot will always be no greater than 1. This is an optimal evenly distributed schedule.

APPENDIX II:

LOOP TIMING ANALYSIS OF RAW ETHERNET

Adapted From a Course Paper By

Daniel Thompson

Troy Walker

December 11, 2002

By

Michael Baxter

LIST OF FIGURES

ABSTRACT

By restricting network nodes to one controller and multiple devices that only speak when spoken to, we removed the need for collision detection and retransmission. In this way, we created a deterministic platform on which many devices could be controlled. This testing seeks to quantify performance gains realized by eliminating the un-needed Ethernet communication overhead.

EXPERIMENTAL IMPLEMENTATION

Our experimental implementation is a simple client/server program. One 866Mhz Pentium PC acted as the server to initiate the communication while another duplicate PC acted as a client to return the communication. Both machines were running RedHat Linux version 7.3 with no windowing system running. The client and server exchanged packets containing a 10- to 20-byte payload as quickly as they could. Two implementations were built. One used standard Ethernet and one used, raw Ethernet protocol. In the raw Ethernet implementation the minimum packet size was set to 62 bytes by the network interface hardware.

The standard Ethernet implementation used standard TCP sockets. The raw Ethernet implementation used the Packet Capture Libraries, libpcap available at http://www.tcpdump.org/, and libnet available at http://www.packetfactory.net/Projects/Libnet/

RESULTS

Both executables were compiled with “g++ -O3 -Wall <sources>”. The raw-Ethernet implementation was statically linked to libpcap and libnet. We ran the programs between PCs named 460two and 460three through eth2, which we hooked together with a 100BaseT hub. We monitored the packets on the network using a laptop which was also plugged into the hub. The TCP implementation used ftime for timing; the raw-Ethernet implementation used the timestamps on captured packets. Here are the raw results for 10,000 and 100,000 round trips:

10,000 Round Trips	
TCP Implementation (s)	Raw-Ethernet Implementation (ms)
1151	840.04
1144	840.04
1151	840.04
1142	840.04
1141	840.04
1144	840.04
1170	840.04
1145	840.04
1153	840.04
1144	840.04
Average: 1148.500	Average: 840.037

Figure 1*: Timing for 10,000 loops*

TCP Implementation (s)	Raw-Ethernet Implementation (ms)
11.51	8.42
11.43	8.39
11.42	8.38
11.48	8.68
11.6	8.4
11.45	8.37
11.4	8.41
11.49	8.41
Average: 11.473	Average: 8.433

Figure 2*: Timing for 100,000 loops*

The on-the-wire packet sizes were 98 for the TCP implementation and 62 bytes for the raw-Ethernet implementation. Our experiments showed an average RTT of 114 microseconds for the TCP implementation and 84 microseconds for raw Ethernet.

Since we were operating over a 100 Mbps Ethernet network and we have eliminated the possibility of collisions, we expected close to full utilization. 60bytes / 42 us *

1000000 us/s * 8 bits/byte = 11,428,571 bits/second = 11.4 Mbps. We are only utilizing just over 10% of the network. This primarily represents the computing speed limit of each of the PCs involved in the test.

One thing we were unable to do in our experiments was get rid of the minimum packet size. To do so, we would have to control the Ethernet card on a hardware level. If we were able to, we could reduce the packet size from 62 to 28 which, potentially, a marked increase in transmission rate because of reduced communication overhead.

CONCLUSION

With the reduced packet overhead achieved by using the raw-Ethernet implementation rather than TCP, we did achieve a significant performance gain. Also, the bandwidth of 100baseT Ethernet exceeds the communication processing bandwidth of 866Mhz PCs.

APPENDIX III:

QNX IMPLEMENTATION OF

VMAC NETWORK COMMUNICATION STACK

By

Michael Baxter

LIST OF FIGURES

INTRODUCTION

The host software runs in a process space of QNX called io-net. This process space contains all the threading and processes to support network communication. The VMAC system introduces a process into this space to handle the different responsibilities of the VMAC host. Additionally, these threads spawn the device driver and the UI manager processes which are external to io-net. The relationship of these processes is shown in Figure 1.

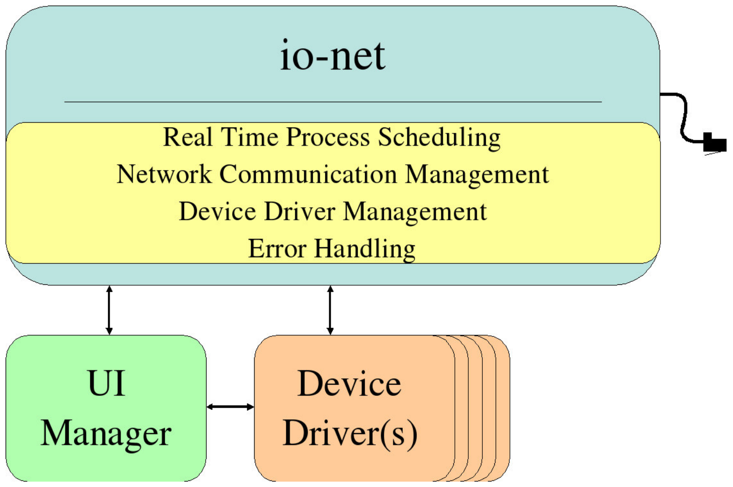


Figure 1: VMAC process spaces are separated for greater reliability

The VMAC process running in io-net configures, runs and maintains the VMAC system. The communication function move information up and down the network communications stack. The scheduler builds the device driver service schedule based on the 2^n algorithm. The device driver management maintains the registry of connected devices from which the service schedule is built. Error handling is done by enforcing timing deadlines.

QNX IO-NET ARCHITECTURE

Io-net is a traditional server/client architecture. The network interface card (NIC) drivers and all other network related resources register with io-net. Once all resources are registered, a user can register with io-net and request a particular network stack and NIC to be used for a communication. This is show in Figure 2. In this illustration, the network driver has registered with io-net as a NIC. The TCP/IP stack and QNET stacks are registered as different stack implementations.

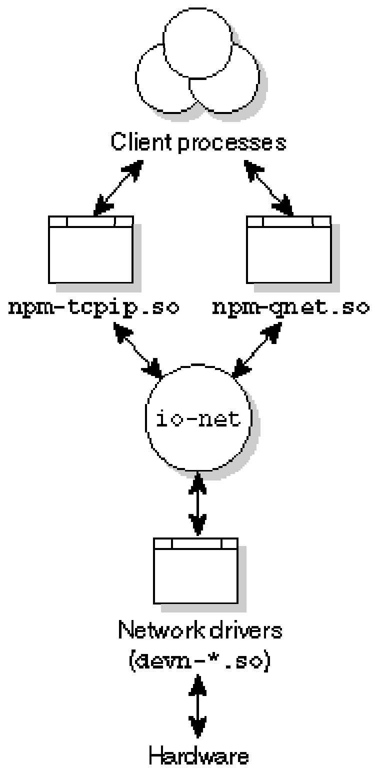


***Figure 2**: QNX io-net client/server relationship*

Either stack can generate packets that will be sent by a NIC connected to io-net. If there is more than one NIC, the user can decide which combination of stacks and NICs should be used.

VMAC IO-NET ARCHITECTURE

VMAC uses this infrastructure to implement the simplified Ethernet communication described in Chapter 4. This communication requires a stack implementation register with io-net that simply passes packets unmodified to the proper NIC. This structure can place packets on the network that were generated by the highest-level user. This structure is shown in Figure 3.

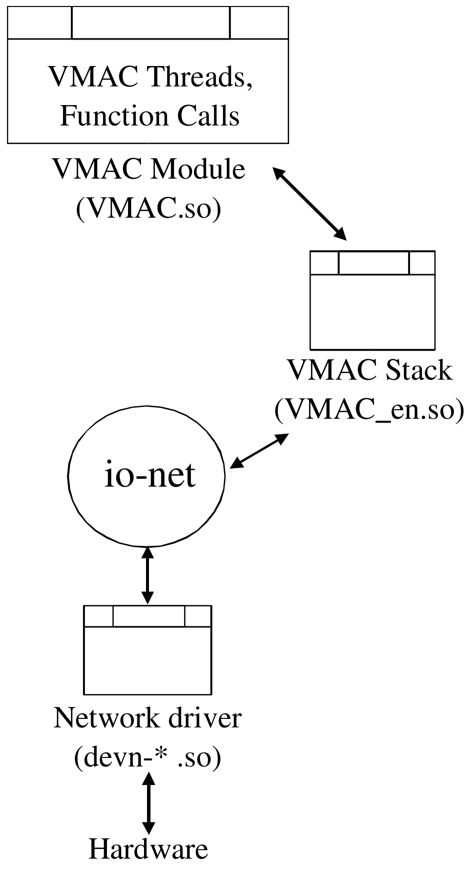


***Figure 3**: VMAC implementation in QNX*

VMAC PACKET STACK

On system startup, the VMAC stack registers with io-net as a packet converter. Its function is to pass packets between the NIC and VMAC Module without modifying the packet. This passing is done both for incoming packets as well as outgoing packets.

The reason for having a converter is that QNX io-net does not have an open API that processes or threads can use to directly access its services. As of version 6.2.1 io-net must have a converter that connects directly to the io-net server to provide this access.

APPENDIX IV:

COST ANALYSIS OF THE VMAC SYSTEM

By

Michael Baxter

LIST OF TABLES

The following tables present a hardware cost analysis of the VMAC system and a comparison to top-of-the-line commercially available motion control hardware.

***Table 1**: VMAC demonstration system cost*

Dell Computer	2	$2,200
Interface Cards	3	$1,500
Network Hub	2	$35
Network Wire	100 feet	$20
UI Computer	1	$800
Misc Hardware	1	$100
Total System Cost:		$4,655

Cost Savings In Control Application

Part	Quantity	Cost
Dish Washer Main Controller	1	$275
Clothes Washer Main Controller	1	$410
Total Unit Savings:		$685
Total Resulting Cost:		$3,970

***Table 2**: Allen Bradley equivalent control system*

ControlLogix	1	$6,000
Kintex 6000	2	$12,000
Dell Computer	1	$1,100
Misc Hardware	1	$100
Total System Cost:		$19,200

The cost information for the Allen Bradley control system are only approximate as the salesman could not give exact values accept for specific control applications.

***Table 3**: VMAC servo motor cost*

Amplifier	1	$250
Interface Circuitry	1	$75
Total Servo Cost:		$600

APPENDIX V:

IMPLEMENTATION SOFTWARE WRITTEN FOR

THE QNX OPERATING SYSTEM

By

Michael Baxter

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

VMAC SOFTWARE DOCUMENTATION INTRODUCTION

This documentation provides a brief explanation of how to work with the VMAC system as a whole as well as details of the implementation software. The implementation software is explained by file, which explanation encompasses memory structure (i.e. C++ class or C struct, etc.), function and variables.

The real time operating system chosen for the VMAC host controller heavily influenced the implementation software. The detailed explanation of conventions and techniques specific to the QNX operating system is left to the public documentation available on the QNX web site (http://www.qnx.com). There are references provided to the information on the QNX website that are current as of the publication date of this document.

Working with the VMAC System

This section describes how to start and stop the vmac system. The process for getting the VMAC system up and running is as follows:

- `Turn on the UI computer and the VMAC host computer`
- `Log into the UI computer as root and start X windows (startx)`
- `ssh into the VMAC host as root (as of Aug 04 the IP address is: 10.8.117.215)`
- `Load the VMAC protocol stack by running the vmac script (vmac)`
- `On the UI computer navigate to ~/fugalh/ui/state_mgr/ and start the state manager (./state_mgr)`
- `On the UI computer navigate to ~/fugalh/ui/net_mgr/ and start the UI net manager listening to ethernet card 1 (./net_mgr eth1)`

Once the VMAC protocol stack is loaded the host (QNX) system is running. The other two processes started on the UI web server facilitate the UI system. The state manager keeps track of the UI state while the UI net manager enforces the rules for talking on the control network by the UI computer.

The UI system allows anyone on the BYU local network to see the user interface page of this system. BYU maintains a firewall that does not allow anyone external to BYU to interrogate internal computers. To enable public access to the pages being served by the UI server a tunnel must be set up through BYU's firewall. For the April 2004 demonstration Hans Fugal set up a computer to form the tunnel. This computer connected to another computer outside of BYU that acted as the server available to the world.

Working with QNX

For development in QNX, some of the handy commands for development in the QNX system include the following:

- `io-net -drtl -pvmac`
  To load just the VMAC stack.
- `sloginfo -w -h`
  To view the vmac debug output written to system log info. Run this in its own terminal.
- `vmac`
  This is a scrip located in /root to start the vmac processes running
- `novmac`
  This is the nice way to end all the VMAC processes that are running
- `slay -q io-net`
  This is a way to kill just the individual `io-net` process. Say no to the first

process, with the lower process IDs (PIDs). The lower PIDs represent the regular NIC diver and TCP/IP stack. The higher PID(s) represent the VMAC stuff.

- `slay XPhoton`

  Handy way to kill the windowing system if you do something that causes it to hang.

The QNX directory structure is strange, but for a reason. See http://www.qnx.com/developer/docs/momentics621_docs/neutrino/prog/make_convent.html. The directory structure levels are as follows:

- `vmac/`
- `vmac/src/`
- `vmac/src/npm/` is the PROJECT level
- `vmac/src/npm/vmac/` is the SECTION level
- `vmac/src/npm/vmac/dll/` is the VARIANT level

A '`make install`' (or even just '`make`' when in `vmac/`) will install the `.so(s)` into `vmac/scratch/x86/lib/dll`.

See also http://www.qnx.com/developer/docs/momentics621_docs/ddk_en/network/writing.html

VMAC SOFTWARE REFERENCE

The files for this project are grouped according to function as follows:

- Configuration management
- User Interface
- Device Driver

The following sections describe the files germane to these functionalities.

Configuration/Management software

The configuration management software provides the interfaces needed to manage the configuration of the VMAC control system. This includes device discovery and setup of new drivers, scheduling all traffic on the control network, and forwarding all received communication to the proper recipient. The device discovery is explained in greater detail in the section of this appendix by that title.

Configuration/Management software

The files of the configuration management software and a short description of each is shown in Table 1.

vmac/src/npm/vmac/down.c	Everything relating to transmitting information down the protocol stack, ie from process to NIC driver

vmac/src/npm/vmac/registry.cc	This contains the functionality to: Add a task to the registry Remove a task from the registry Reprioritize a task in the registry
vmac/src/npm/vmac/registry.h	Registry header
vmac/src/npm/vmac/scheduler.cc	This is the algorithm that forms the schedule for communication with peripheral devices as well as execution of the device control loops
vmac/src/npm/vmac/scheduler.h	Header for the scheduler. Declares functionality to build communication schedule.
vmac/src/npm/vmac/task.h	Schedulable task template used to point to each device device driver
vmac/src/npm/vmac/taskman.cc	Manages all communication with device drivers
vmac/src/npm/vmac/taskman.h	Task manager header file.
vmac/src/npm/vmac/up.c	Everything relating to communication moving up the (receiving) the protocol stack
vmac/src/npm/vmac/vmac.c	Main coordination file for the VMAC system
vmac/src/npm/vmac/vmac.h	Vmac network producer module (npm) that gets compiled into the .so to be loaded into io-net
vmac/src/npm/vmac/vmac_en.c	Vmac converter: From ethernet to VMAC and vice versa
vmac/src/npm/vmac/vmac_en.h	Vmac_en converter header file

Table 1: Configuration management files

The following is a more detailed explanation of each file.

VMAC/SRC/NPM/VMAC/DOWN.C

Everything relating to transmitting information down the protocol stack.

Author:
Hans Fugal

Definition in file down.c.

```
#include <netinet/in.h>
#include <sys/neutrino.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/syspage.h>
#include "vmac.h"
#include "vmac_driver.h"
#include "util.h"
#include "scheduler.h"
#include "registry.h"
```

Include dependency graph for down.c is shown in Figure 1.

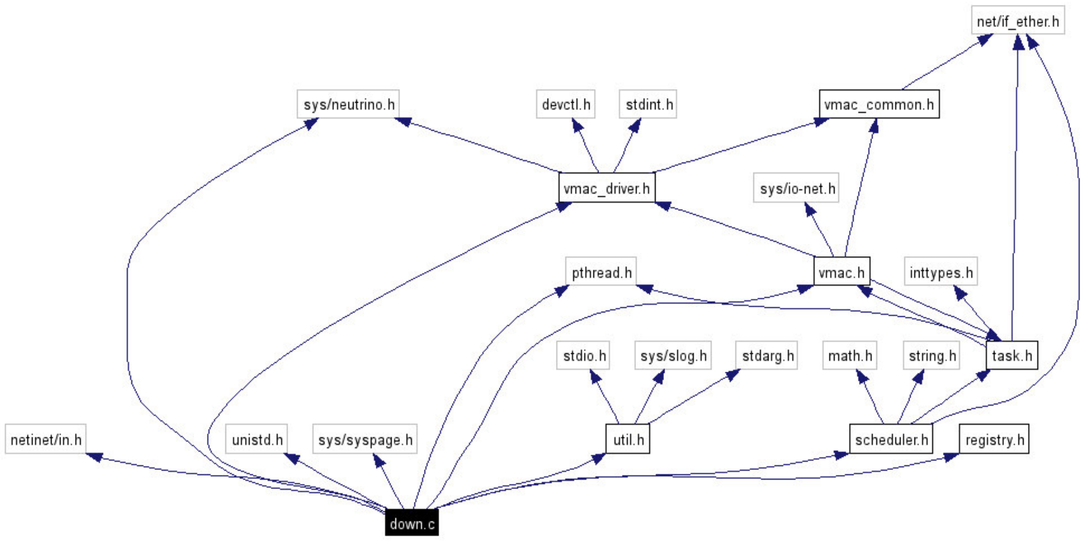


Figure 1: Down.c dependency graph

Functions

void * tx_thread (void *func_hdl)
The Tx thread, AKA the control thread.

Variables

task * cur_task
Task currently being serviced.

Table 2: Functions for Down.c

vmac/src/npm/vmac/registry.cc

This file contains the functionality to:

- Add a task to the registry
- Remove a task from the registry
- Reprioritize a task in the registry

The registry is stored as a sorted STL set of Task structs. Each of these structs keeps the information needed to access the device driver.

Author:

Michael Baxter <mike.baxter@byu.edu>

Definition in file registry.cc.

```
#include "scheduler.h"
#include <pthread.h>
#include <sys/neutrino.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <unistd.h>
#include <list>
#include "util.h"
#include "task.h"
#include "registry.h"
#include "vmac_driver.h"
#include "vmac.h"
```

Include dependency graph for registry.cc in Figure 2.

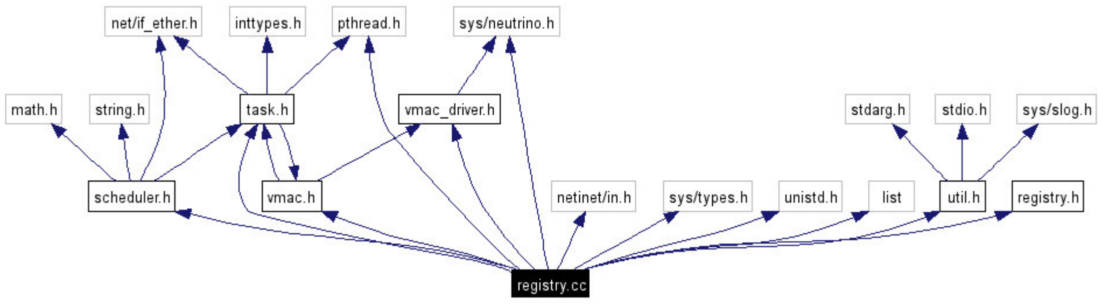


Figure 2: Dependence graph for registry.cc

Functions

task *	create_task (int id) *pick an element of tasks to be pointed to in id_task_map, insert it into the map, and return it.*

Variables

int	rcoid *registration thread connection id*
int	rchid *Registration thread channel id.*
std::map< int, task * >	id_task_map *id-to-task map*
int	discovery_task_id *The discovery task's task id.*
int	ucoid *UIM connection id.*
int	next_id = 1 *The next id. Initialized to 1 because id 0 is an invalid id.*
Registry	registry *lazy singleton*

Table 3: Functions implmented in registry.cc

vmac/src/npm/vmac/registry.h

Registry header.

This class contains the functionality to:

- Add a task to the registry
- Remove a task from the registry
- Reprioritize a task in the registry
- Write the registry to non-volatile media
- Build the registry from information stored on the non-volatile media

The registry is stored in 15 linked lists of task structs. Each of these structs keeps the information needed to access the device driver and each linked list represents a priority.

Author:

Michael Baxter <mike.baxter@byu.edu>
Hans Fugal

Definition in file registry.h.

This graph shows which files directly or indirectly include this file:

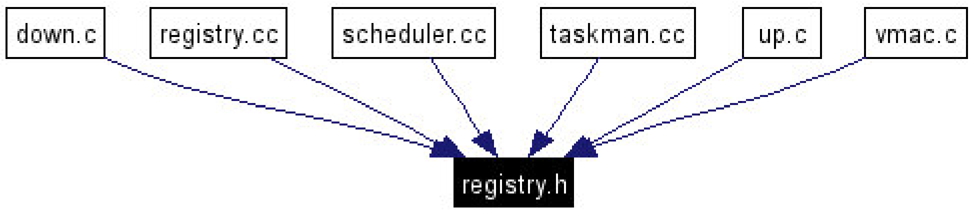


Figure 3: Dependency graph for registry.h

FUNCTIONS

void * reg_thread (void *)
registration thread function

Variables

int rcoid

registration thread connection id

int rchid

Registration thread channel id.

int discovery_task_id

The discovery task's task id.

Table 4: Functions declared in registry.h

vmac/src/npm/vmac/scheduler.cc

This is the algorithm that will form the Schedule for the the communication with and execution of the device control loops in the VMAC system.

The structure of this algorithm is to use 2 linked lists to keep track of where to put Devices. The first linked list contains pointers to positions for the next device. The second contains the pointers of where Devices have previously been placed.

The proceedure is to start placing Devices in the location pointed to by the first pointer in the first linked list. After placing a device then the pointer is moved into the second linked list.

For example, if the Devices to be Scheduled are: N = 0 A1 N = 1 B1,B2 N = 2 C1,C2,C3 N = 3 D1 then A1 is placed in the first location pointed to by the first pointere in list k. Starting in the first location A1 is placed in succeeding locations until all locations have A1, and each pointer is copied to list K+1.

Next the pointers for the two liked lists are swapped so that K+1 is now K and k is incremented to k = 1. B1 is placed in the first location, and then every 2^N. These pointers are moved to the now empty list K+1. B2 is placed and the remaining pointers are coped to K+1. Another pointer swap and K is filled again (k = 2). C1 is placed in the first pointer, C2 is placed, C3 is placed...

To keep track of where the nodes of the linked list are there is another array of pointers to the nodes of this list that are always in the right order. So the process becomes this:

- Return the pointer and index from the first element in the linked list to get the offset into the slot array
- Use this pointer to find the first location in the schedule array
- Use the index into the slot array as a starting point for the 2^n offseting rule
- Move through the slot array according to the 2^n rule at each element in the slot describing where to place something, use the pointer stored at that element to obtain the node's pointer to the schedule array.
- Move the node from the k list to the k_1 list

The Schedule is full when 10 swaps have taken place of the k and k+1 lists. This is defined by k = 10.

Author:

Michael Baxter <mike.baxter@byu.edu>

Definition in file scheduler.cc.

```
#include "scheduler.h"
#include "task.h"
#include "registry.h"
#include "util.h"
```

Include dependency graph for scheduler.cc shown in Figure 4:

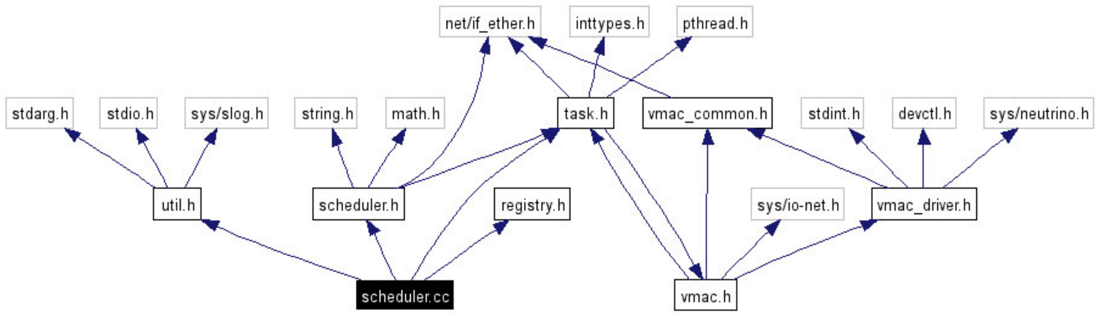


Figure 4: Dependency graph for scheduler.cc

FUNCTIONS

task *	next_task (void) *next task (extern "C" function)*

VARIABLES

Scheduler	scheduler *The Scheduler instance (lazy-singleton).*

Table 5: Functions for scheduler.cc

VMAC/SRC/NPM/VMAC/SCHEDULER.H

Common header for the scheduler.

Author:

Michael Baxter <michael.baxter@byu.edu>

Definition in file scheduler.h.

#include <math.h>

#include <string.h>

#include <net/if_ether.h>

#include "task.h"

Include dependency graph for scheduler.h shown in Figure 5:

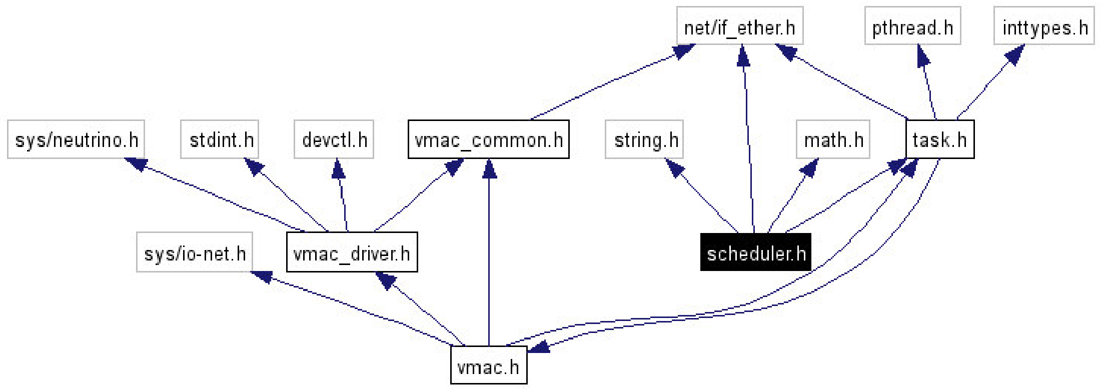


Figure 5: Dependency graph for scheduler.h

Figure 6 shows which files directly or indirectly include this file:

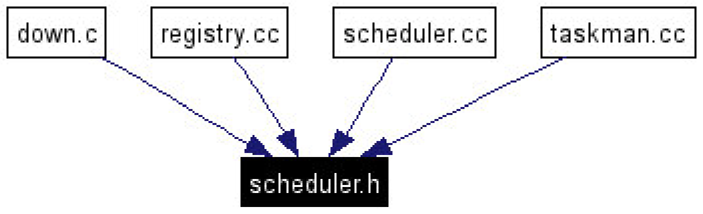


Figure 6: Files that refer to scheduler.h

DEFINES

#define N 3

Lowest priority level (reverse 0-based count).

#define POW2(x) (1 << (x))

2^x

#define SCHED_SIZE POW2(N)

2^N

#define	SLOT_SIZE 2
	number of slices in a slot
#define	SLOT_PERIOD .1e6
	Slot period (in microseconds).
#define	SLICE_PERIOD (SLOT_PERIOD/SLOT_SIZE)
	The period of one slice = slot period / number of slices per slot.
FUNCTIONS	
task *	next_task (void)
	next task (extern "C" function)

Table 6: Macros and Functions of scheduler.h

VMAC/SRC/NPM/VMAC/TASK.H

Schedulable task template.

Author:

Hans Fugal

Michael Baxter

Definition in file task.h.

```
#include <inttypes.h>
#include <net/if_ether.h>
#include <pthread.h>
#include "vmac.h"
```

Include dependency graph for task.h:

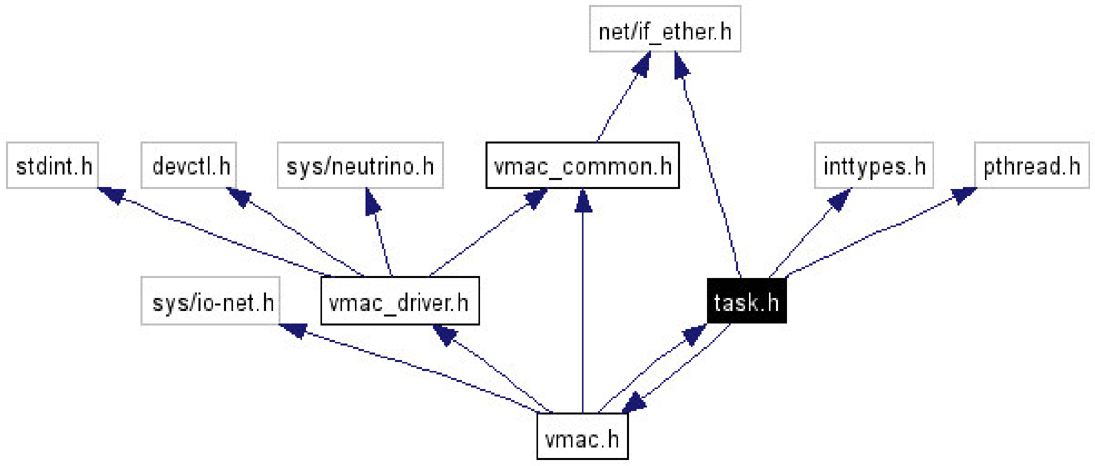


Figure 7: Dependency graph for task.h

This graph shows which files directly or indirectly include this file:

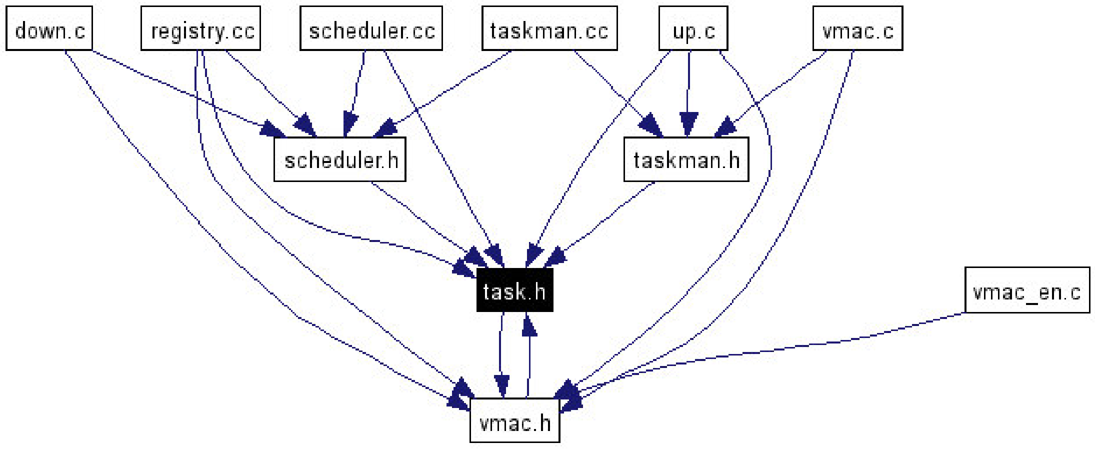


Figure 8: Files that include task.h

CLASSES

struct task
All the goodies it takes to make a usefull task

VARIABLES

task * cur_task
Task currently being serviced.

Table 7: Classes and variables of task.h

VMAC/SRC/NPM/VMAC/TASKMAN.CC

The task manager is the heart of communication with drivers.

It is a thread in an event loop. It receives messages from drivers, the UIM, and the other npm threads, then takes the appropriate action.

(No, it's not related to Windows' task manager. ;-)

Author:
Hans Fugal

Definition in file taskman.cc.

```
#include <string.h>
#include <deque>
#include <stddef.h>
#include <process.h>
#include <sched.h>
#include <assert.h>
#include <string>
#include <map>
#include <sstream>
#include <errno.h>
#include "taskman.h"
#include "util.h"
```

```
#include "registry.h"
#include "scheduler.h"
```

Include dependency graph for taskman.cc shown in Figure 9:

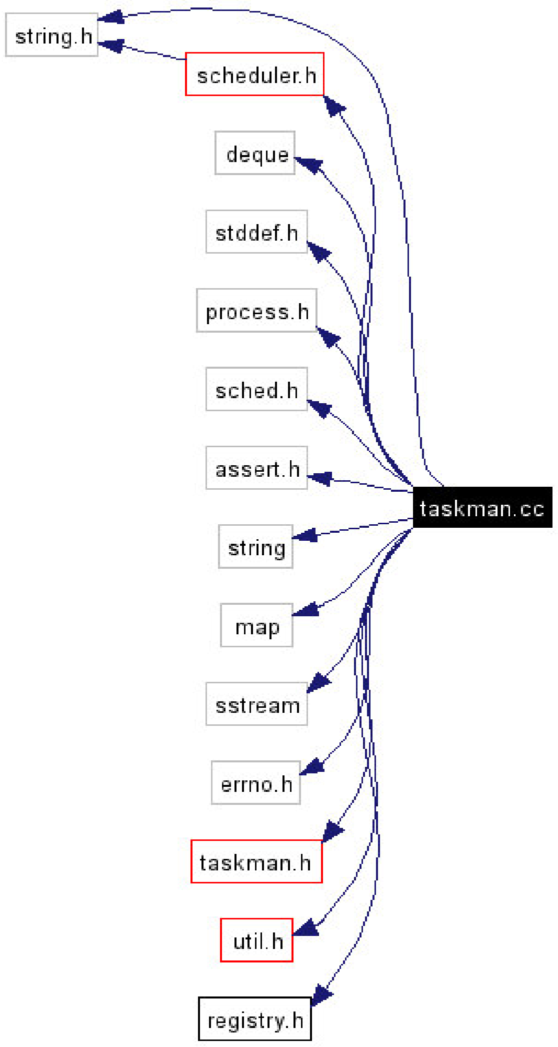


Figure 9: Dependency graph for taskman.cc

FUNCTIONS

void *	task_thread (void *arg)
	task manager thread function
void	empty_rx (int id)
	If we got an empty upwards packet and would like to forgive.

Table 8: Functions implemented in taskman.cc

VMAC/SRC/NPM/VMAC/TASKMAN.H

Task manager header file.

(No, it's not related to Windows' task manager. ;-)

Author:
Hans Fugal

Definition in file taskman.h.

#include "task.h"

Include dependency graph for taskman.h:

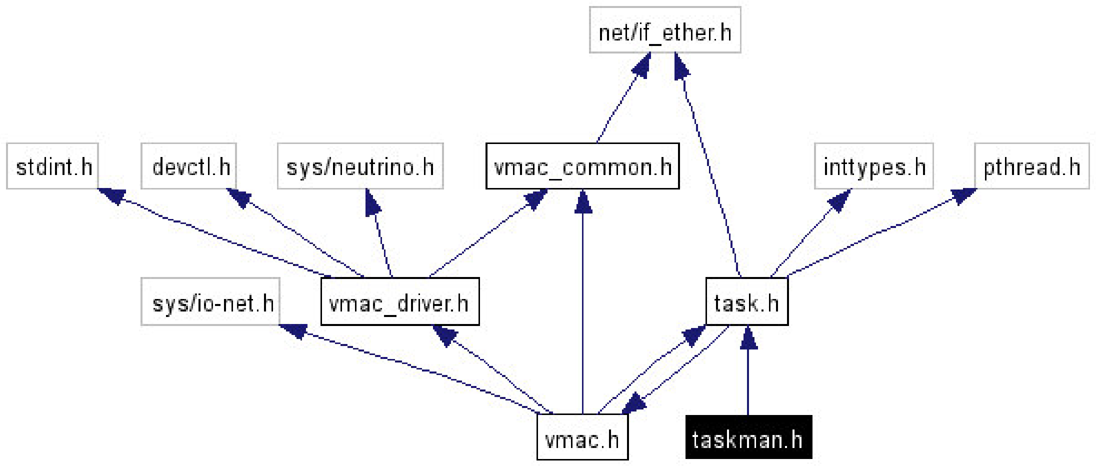


Figure 10: Dependency graph for taskman.h

This graph shows which files directly or indirectly include this file:

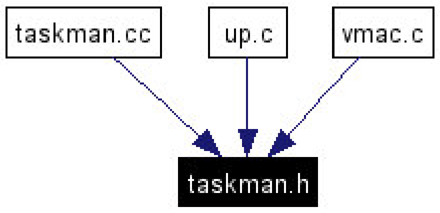


Figure 11: Files that include taskman.h

FUNCTIONS	
void *	task_thread (void *arg)
	task manager thread function
void	empty_rx (int id)
	We got an empty upwards packet and would like to forgive.

Table 9: Functions declared in taskman.h

vmac/src/npm/vmac/up.c

Everything relating to passing information up the protocol stack (receiving network data).

Author:

Hans Fugal

Definition in file up.c.

```
#include <sys/syspage.h>
#include <assert.h>
#include <netinet/in.h>
#include <sys/neutrino.h>
#include <pthread.h>
#include <unistd.h>
#include <semaphore.h>
#include <stdlib.h>
#include "vmac.h"
#include "vmac_driver.h"
#include "util.h"
#include "registry.h"
#include "taskman.h"
#include "task.h"
```

Include dependency graph for up.c shown in Figure 12:

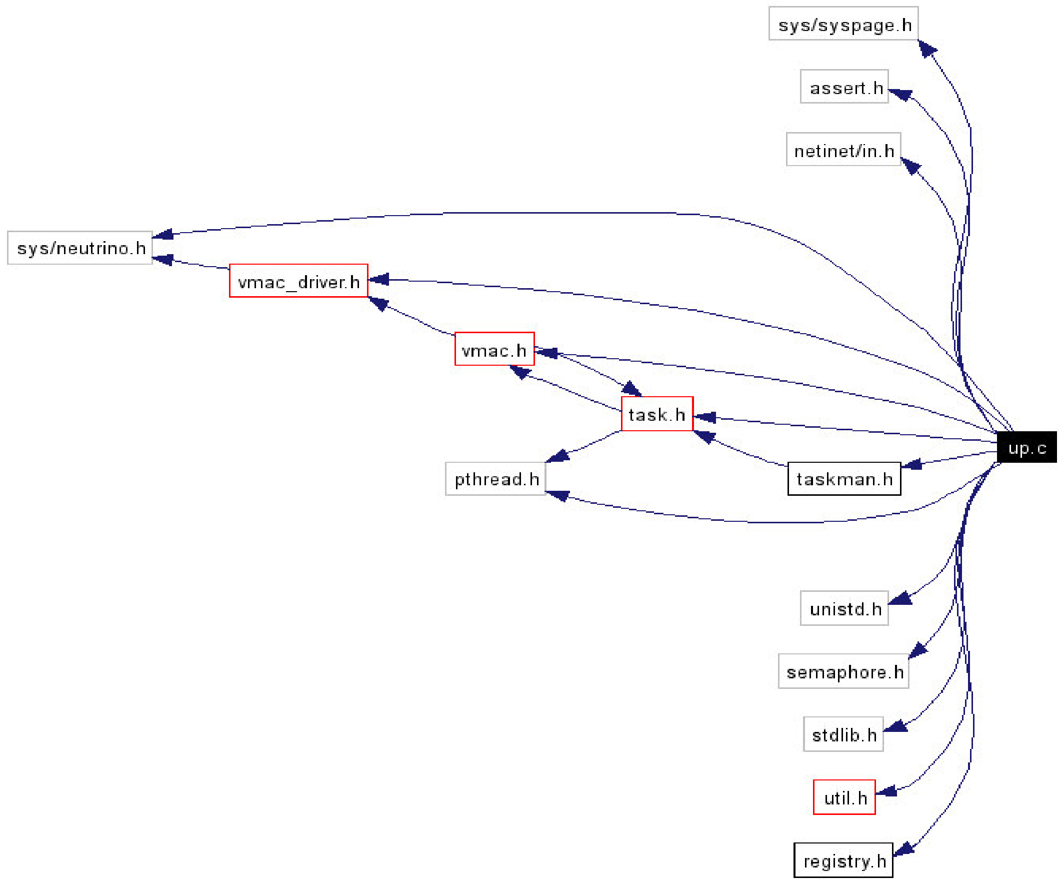


Figure 12: Dependency graph for up.c

Functions

int vmac_rx_up (npkt_t *npkt, void *func_hdl, int off, int framlen_sub, uint16_t cell, uint16_t endpoint, uint16_t iface)
http://www.qnx.com/developer/docs/momentics621_docs/ddk_en/network/io_net_registrant_funcs_t.html

Table 10: Function implemented in up.c

vmac/src/npm/vmac/vmac.c

Main coordination file for the VMAC system.

Author:

Hans Fugal

Definition in file vmac.c.

```
#include <errno.h>
#include <stdlib.h>
#include <pthread.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/io-net.h>
#include <net/if.h>
#include <netinet/in.h>
#include <sys/neutrino.h>
#include <sys/syspage.h>
#include <assert.h>
#include <sys/dcmd_io-net.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include "vmac.h"
#include "vmac_en.h"
#include "util.h"
#include "vmac_driver.h"
#include "registry.h"
#include "taskman.h"
```

Include dependency graph for vmac.c shown in Figure 13:

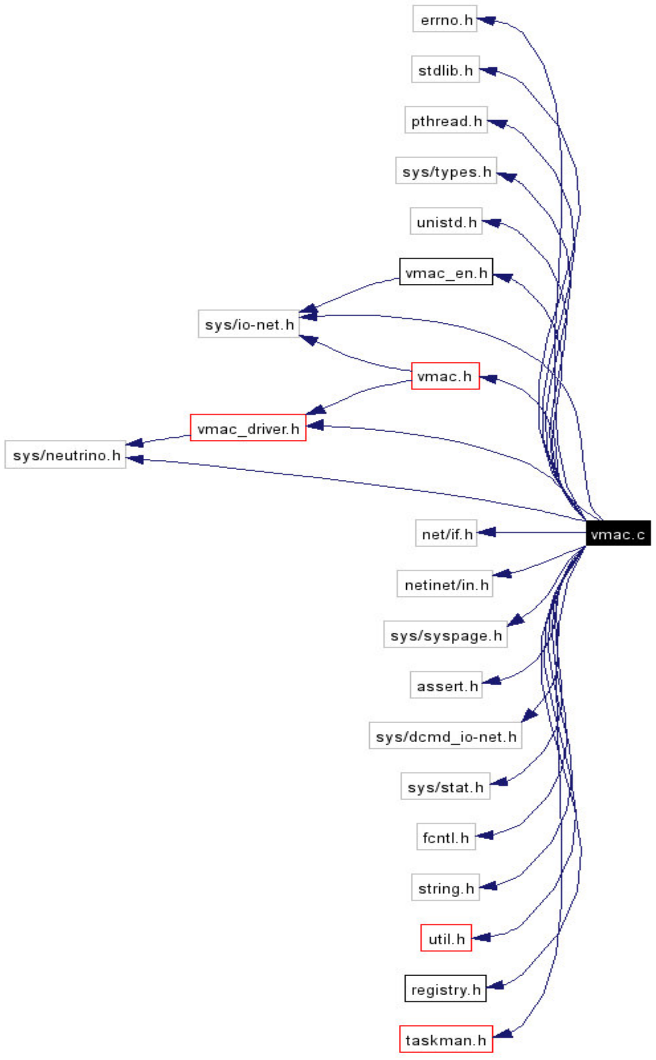


Figure 13: Dependency graph for vmac.c

Functions

int vmac_shutdown (void *dll_hdl)
master shutdown

int vmac_register (vmac_hdl *hdl)
vmac's io-net registration function

nt vmac_rx_down (npkt_t *npkt, void *func_hdl)
http://www.qnx.com/developer/docs/momentics621_docs/ddk_en/network/io_net_registrant_funcs_t.html

int vmac_tx_done (npkt_t *npkt, void *done_hdl, void *func_hdl)

		http://www.qnx.com/developer/docs/momentics621_docs/ddk_en/network/io_net_registrant_funcs_t.html
int	vmac_shutdown1 (int registrant_hdl, void *func_hdl)	*http://www.qnx.com/developer/docs/momentics621_docs/ddk_en/network/io_net_registrant_funcs_t.html*
int	vmac_shutdown2 (int registrant_hdl, void *func_hdl)	*http://www.qnx.com/developer/docs/momentics621_docs/ddk_en/network/io_net_registrant_funcs_t.html*
int	vmac_dl_advert (int reg_hdl, void *func_hdl)	*http://www.qnx.com/developer/docs/momentics621_docs/ddk_en/network/io_net_registrant_funcs_t.html*
int	vmac_devctl (void *registrant_hdl, int dcmd, void *data, size_t size, union _io_net_dcmd_ret_cred *ret)	*devctl handler.*
int	vmac_flush (int registrant_hdl, void *func_hdl)	*http://www.qnx.com/developer/docs/momentics621_docs/ddk_en/network/io_net_registrant_funcs_t.html*
VARIABLES		
int	done = 0	*are we done?*
io_net_dll_entry_t	io_net_dll_entry	*Entry for a shared object to be loaded by io-net.*
io_net_registrant_funcs_t	vmac_funcs	*Functions in your driver that io-net can call.*
io_net_registrant_t	vmac_registrant	*http://www.qnx.com/developer/docs/momentics621_docs/ddk_en/network/io_net_registrant_funcs_t.html*

Table 11: Functions and variables implemented in vmac.c

VMAC/SRC/NPM/VMAC/VMAC.H

vmac network producer module (npm).

This is a down producer, with an up-type of NULL and a down-type of vmac.

Author:

Hans Fugal

Definition in file vmac.h.

```
#include <sys/io-net.h>
#include "vmac_common.h"
#include "vmac_driver.h"
#include "task.h"
```

Include dependency graph for vmac.h shown in Figure 14:

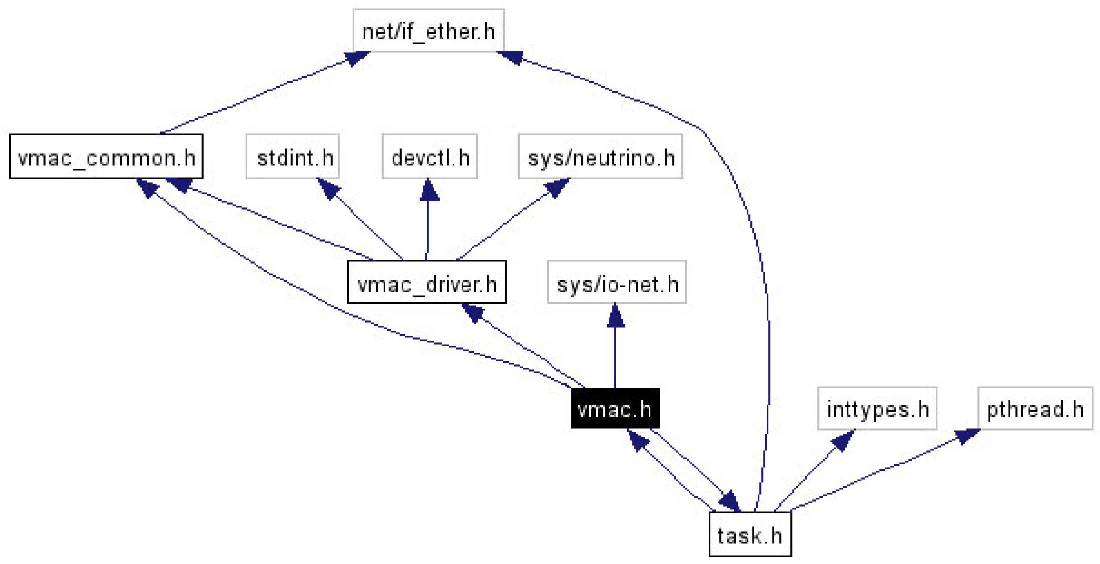


Figure 14: Dependency graph for vmac.h

This graph shows which files directly or indirectly include this file:

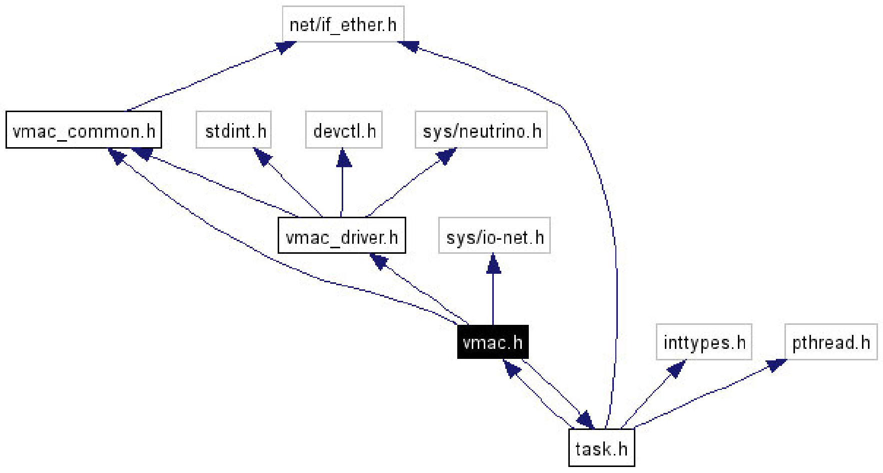


Figure 15: Files that include vmac.h

Classes

struct vmac_hdl

vmac instance handle.

This context is passed as the func_hdl parameter to our registrant functions by io-net.

Definition at line 32 of file vmac.h.

Public Attributes

*void * dll_hdl*

This may not be necessary.

int reg_hdl

The registration handle by which io-net knows us.

uint16_t cell

The cell we reside in.

*io_net_self_t * ion*

io-net function pointers

uint16_t endpoint

The endpoint associated with this instance.

Defines

#define ETHER_ADDR_LEN 6
the length of a mac address

FUNCTIONS

int vmac_register (vmac_hdl *hdl)
vmac's io-net registration function

int vmac_rx_up (npkt_t *npkt, void *func_hdl, int off, int framlen_sub, uint16_t cell, uint16_t endpoint, uint16_t iface)
http://www.qnx.com/developer/docs/momentics621_docs/ddk_en/network/io_net_registrant_funcs_t.html

int vmac_rx_down (npkt_t *npkt, void *func_hdl)
http://www.qnx.com/developer/docs/momentics621_docs/ddk_en/network/io_net_registrant_funcs_t.html

int vmac_tx_done (npkt_t *npkt, void *done_hdl, void *func_hdl)
http://www.qnx.com/developer/docs/momentics621_docs/ddk_en/network/io_net_registrant_funcs_t.html

int vmac_shutdown1 (int registrant_hdl, void *func_hdl)
http://www.qnx.com/developer/docs/momentics621_docs/ddk_en/network/io_net_registrant_funcs_t.html

int vmac_shutdown2 (int registrant_hdl, void *func_hdl)
http://www.qnx.com/developer/docs/momentics621_docs/ddk_en/network/io_net_registrant_funcs_t.html

int vmac_dl_advert (int registrant_hdl, void *func_hdl)
http://www.qnx.com/developer/docs/momentics621_docs/ddk_en/network/io_net_registrant_funcs_t.html

int vmac_flush (int registrant_hdl, void *func_hdl)
http://www.qnx.com/developer/docs/momentics621_docs/ddk_en/network/io_net_registrant_funcs_t.html

int vmac_devctl (void *registrant_hdl, int dcmd, void *data, size_t size, union _io_net_dcmd_ret_cred *ret)
devctl handler.

void * tx_thread (void *)
The Tx thread, AKA the control thread.

VARIABLES

int done
are we done?

Table 12: Classes, macros, functions and varibles declared in vmac.h

Vmac converter: From ethernet to VMAC and vice versa.

Author:

Hans Fugal

Definition in file vmac_en.c.

```
#include <errno.h>
#include <stdlib.h>
#include <sys/io-net.h>
#include <net/if_types.h>
#include <net/if.h>
#include <netinet/in.h>
#include "vmac_common.h"
#include "vmac_driver.h"
#include "vmac_en.h"
#include "util.h"
#include "vmac.h"
```

Include dependency graph for vmac_en.c shown in Figure 16:

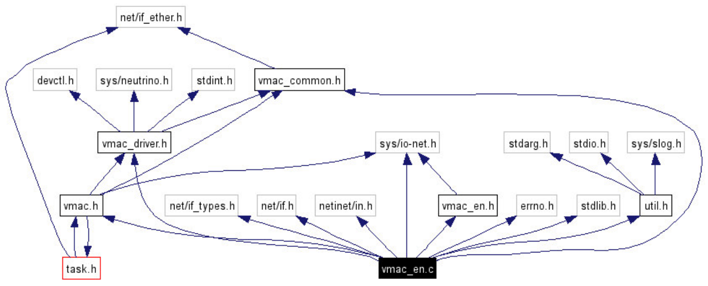


Figure 16: Dependency graph for vmac_en.c

FUNCTIONS

int	vmac_en_register (vmac_en_hdl *hdl) *io-net registration function for vmac_en*
int	vmac_en_rx_up (npkt_t *npkt, void *func_hdl, int off, int framelen_sub, uint16_t cell, uint16_t endpoint, uint16_t iface) *http://www.qnx.com/developer/docs/momentics621_docs/ddk_en/network/io_net_registrant_funcs_t.html*
int	vmac_en_rx_down (npkt_t *npkt, void *func_hdl) *fill in the ethernet source header.*
int	vmac_en_tx_done (npkt_t *npkt, void *done_hdl, void *func_hdl) *http://www.qnx.com/developer/docs/momentics621_docs/ddk_en/network/io_net_registrant_funcs_t.html*
int	vmac_en_shutdown1 (int registrant_hdl, void *func_hdl) *http://www.qnx.com/developer/docs/momentics621_docs/ddk_en/network/io_net_registrant_funcs_t.html*
int	vmac_en_shutdown2 (int registrant_hdl, void *func_hdl) *http://www.qnx.com/developer/docs/momentics621_docs/ddk_en/network/io_net_registrant_funcs_t.html*
int	vmac_en_dl_advert (int reg_hdl, void *func_hdl) *http://www.qnx.com/developer/docs/momentics621_docs/ddk_en/network/io_net_registrant_funcs_t.html*
int	vmac_en_flush (int registrant_hdl, void *func_hdl) *http://www.qnx.com/developer/docs/momentics621_docs/ddk_en/network/io_net_registrant_funcs_t.html*

VARIABLES

io_net_registrant_funcs_t	vmac_en_funcs *Functions in your driver that io-net can call.*
io_net_registrant_t	vmac_en_registrant *http://www.qnx.com/developer/docs/momentics621_docs/ddk_en/network/io_net_registrant_funcs_t.html*

Table 13: Functions and variables implemented in vmac_en.c

VMAC/SRC/NPM/VMAC/VMAC_EN.H

vmac_en converter.

It converts from en to vmac and vice versa. At present it does not add any header information (that is, whatever it gets in rx_down() will be what goes out on the wire)

Author:
Hans Fugal

Definition in file vmac_en.h.

#include <sys/io-net.h>

Include dependency graph for vmac_en.h:

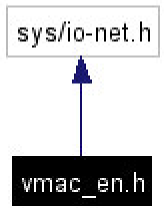


Figure 17: Dependency graph for vmac_en.h

This graph shows which files directly or indirectly include this file:

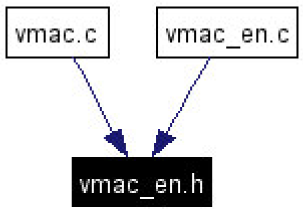


Figure 18: Files that include vmac_en.h

Classes

struct

vmac_en_hdl

vmac_en instance handle.

This context is passed as the func_hdl parameter to our registrant functions by io-net.

Definition at line 17 of file vmac_en.h.

Public Attributes

*void * dll_hdl*

This may not be necessary.

int reg_hdl

The registration handle by which io-net knows us.

uint16_t cell

The cell we reside in.

*io_net_self_t * ion*

io-net function pointers

uint16_t endpoint

The endpoint associated with this instance.

uint8_t src_addr [6]

MAC address.

struct

eth_hdr_t

ethernet header. Definition at line 28 of file vmac_en.h.

Public Attributes

uint8_t dst [6]

Destination address.

uint8_t src [6]

Source address.

uint16_t type

type (we use this as the length, per 802.3)

Functions

int	vmac_en_register (vmac_en_hdl *hdl) *io-net registration function for vmac_en*
int	vmac_en_rx_up (npkt_t *npkt, void *func_hdl, int off, int framlen_sub, uint16_t cell, uint16_t endpoint, uint16_t iface) *http://www.qnx.com/developer/docs/momentics621_docs/ddk_en/network/io_net_registrant_funcs_t.html*
int	vmac_en_rx_down (npkt_t *npkt, void *func_hdl) *fill in the ethernet source header.*
int	vmac_en_tx_done (npkt_t *npkt, void *done_hdl, void *func_hdl) *http://www.qnx.com/developer/docs/momentics621_docs/ddk_en/network/io_net_registrant_funcs_t.html*
int	vmac_en_shutdown1 (int registrant_hdl, void *func_hdl) *http://www.qnx.com/developer/docs/momentics621_docs/ddk_en/network/io_net_registrant_funcs_t.html*
int	vmac_en_shutdown2 (int registrant_hdl, void *func_hdl) *http://www.qnx.com/developer/docs/momentics621_docs/ddk_en/network/io_net_registrant_funcs_t.html*
int	vmac_en_dl_advert (int registrant_hdl, void *func_hdl) *http://www.qnx.com/developer/docs/momentics621_docs/ddk_en/network/io_net_registrant_funcs_t.html*
int	vmac_en_flush (int registrant_hdl, void *func_hdl) *http://www.qnx.com/developer/docs/momentics621_docs/ddk_en/network/io_net_registrant_funcs_t.html*

Table 14: Classes and functions declared in vmac_en.h

User Interface manager

The user interface manager's purpose is to ensure that all UI web servers connected to control network receive the same UI information and that any new UI web servers are updated with all available information.

The UIM is designed to look like a device driver. It contains many similary elements and uses similar interfaces as a device driver. When the system starts the UIM is launched in the same manner as a device driver. The UIM is scheduled for service in the same fassion as a driver. When there is user input or user information to be displayed the UIM handles this driver while waiting for the next communication from the configuration management software. In this way the UIM is exactly like a device driver.

The UIMkeeps a detailed mapping between device driver IDs and UI server IDs. In this way it is able to track which web server has received what information. The result of this system is that all UI Web Servers are allways current with the state of all devices being controlled. This is true for all usage situations, ie when a peripheral device is connected or removed, when a driver has an update, or when there has been user input given through one of the web servers.

The files of the user interface manager software and a short description of each is shown in Table 15.

vmac/src/driver/uim/comm_director.cc	Forulates network packets for transmission
vmac/src/driver/uim/comm_director.h	Declaration of the packet former for the UIM
vmac/src/driver/uim/ui_driver.cc	The executable main UIM file that makes it all happen
vmac/src/driver/uim/ui_driver.h	Header file for the UIM
vmac/src/driver/uim/uim_mapper.cc	Map between driver id and UI server ids
vmac/src/driver/uim/uim_mapper.h	Contains mapping from drivers to UI server IDs
vmac/src/driver/uim/uim_master.cc	Keeps track of all UI servers
vmac/src/driver/uim/uim_master.h	Header for keeping track of UI servers

Table 15: Files required for the User Interface Manager

The following is a description of each file required for the the UIM.

VMAC/SRC/DRIVER/UIM/COMM_DIRECTOR.CC

Forulates network packets for transmission.

Author:
Michael Brailsford
created: Mon Aug 11 17:11:26 MDT 2003
copyright: 2003 Michael Brailsford
Revision: 1.24

Definition in file comm_director.cc.

#include "comm_director.h"

Include dependency graph for comm_director.cc:

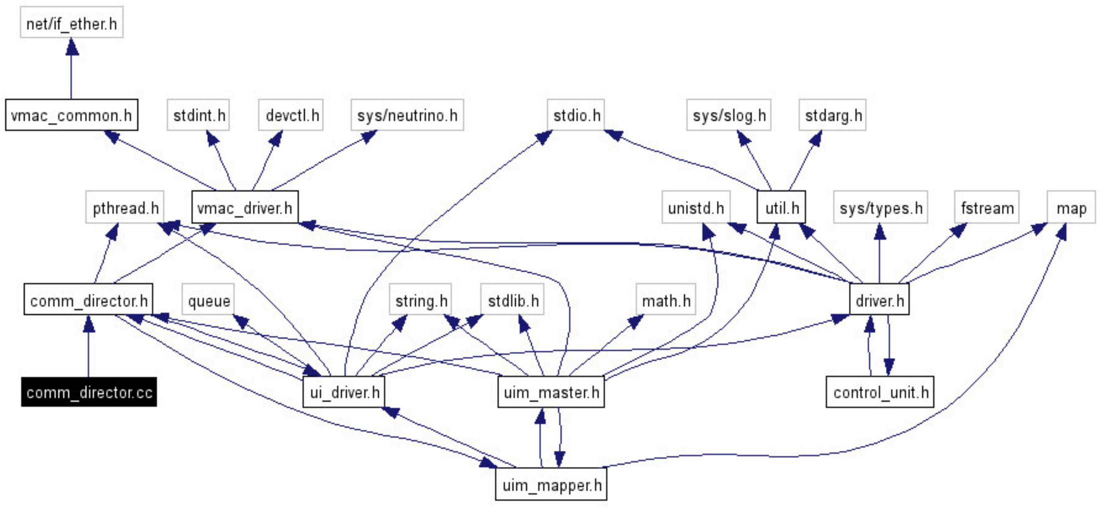


Figure 19: Dependency graph for comm_director.cc

VMAC/SRC/DRIVER/UIM/COMM_DIRECTOR.H

Declaration of the packet former for the UIM.

Author:
    Michael Brailsford
    created: Fri Aug 08 23:27:31 MDT 2003
    copyright: (c) 2003 Michael Brailsford
    Revision: 1.14

Definition in file comm_director.h.

#include <pthread.h>

#include "vmac_driver.h"

#include "uim_mapper.h"

#include "ui_driver.h"

Include dependency graph for comm_director.h:

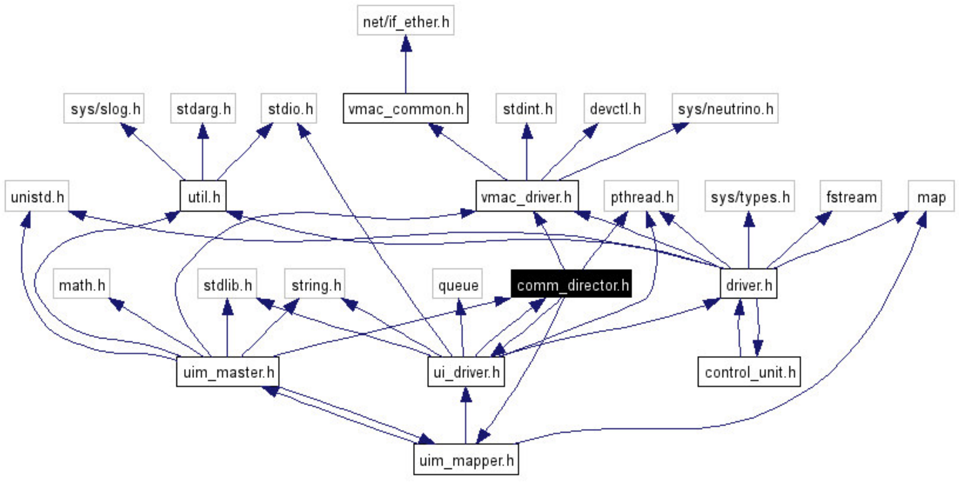


Figure 20: Dependency graph for comm_director.h

This graph shows which files directly or indirectly include this file:

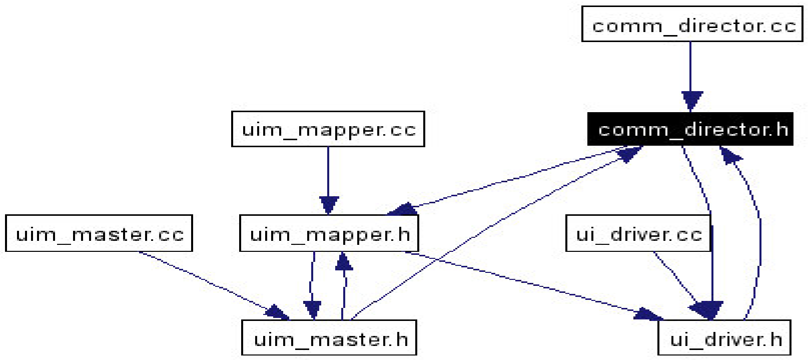


Figure 21: Files that include comm_director.h

Classes

class CommDirector

This class is the Mediator (see Mediator Pattern) between the UiDriver objects and the device drivers.

It provides a simple mechanism for normal communication to and from the

UiDriver objects to the device drivers. Following is a list of messages that a UI device driver can send to an device driver. Anythong not on this list is an error.

- *UIM_DRV_INIT expects a simple empty reply in response.*
- *UIM_DRV_REPORT expects a DRV_UIM_REPORT in response. When the response comes back then it is the responsiblity of the CommDirector to make sure that each UI gets the update in its buffer.*
- *UIM_DRV_RECORD expects a simple empty reply in response. Following is a list of the all the response the the CommDirector expects from a device driver. Anything that is not in this list is an error.*
    - *DRV_UIM_REPORT should come in response to a UIM_DRV_REPORT message sent to the driver.*

Definition at line 39 of file comm_director.h.

Public Member Functions

~CommDirector ()

Default destructor.

*void talk_to (pid_t pid, vmac_msg_t *vmsg)*

Sends a message to the device driver with the process id of pid.

void listen_for_updates ()

Listen on the update notification channel for updates from a device driver.

void listen ()

Listen on the update notification channel, and forward all updates to the UiDriver objects that are registered in the system.

void stop_listening ()

Make the CommDirector stop listening for updates on the update notification channel.

*void get_all_templates (UiDriver *drv)*

This returns all XML templates concatenated into one long buffer.

*void add_data_to_all_uis (uint8_t *data, int len, pid_t pid=-1)*

Add the data to all the UIs' buffers.

*bool send_init (struct driver_info *dinfo, pid_t drv_pid)*

Send the driver initialization sequence to the driver with pid.

Static Public Member Functions

*CommDirector * get_instance ()*

Returns the singleton instance of the CommDirector class.

DEFINES

#define MAX_DRIVERS 50

This is an aribtrary upper limit on the number of devices that can be in the VMAC system.

Table 16: Classes and macros for comm_director.h

VMAC/SRC/DRIVER/UIM/UI_DRIVER.CC

The executable main UIM file that makes it all happen.

Author:

Michael Brailsford
created: Wed Aug 06 15:28:06 MDT 2003
copyright: 2003 Michael Brailsford
Revision: 1.28

Definition in file ui_driver.cc.

#include "ui_driver.h"

Include dependency graph for ui_driver.cc:

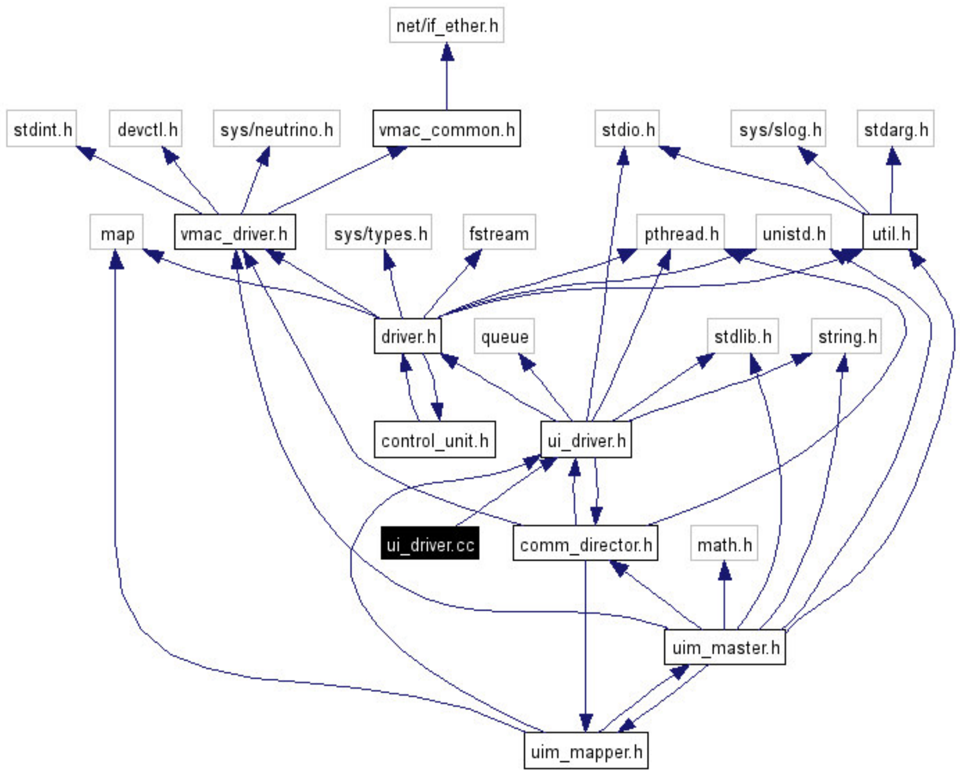


Figure 22: Dependency graph for ui_driver.cc

VMAC/SRC/DRIVER/UIM/UI_DRIVER.H

Header file for the UIM.

Author:
    Michael Brailsford
    created: Wed Aug 06 14:24:33 MDT 2003
    copyright: (c) 2003 Michael Brailsford
    Revision: 1.18

Definition in file ui_driver.h.

```
#include <stdio.h>
#include <string.h>
```

```
#include <stdlib.h>
#include <queue>
#include <pthread.h>
#include "driver.h"
#include "comm_director.h"
```

Include dependency graph for ui_driver.h:

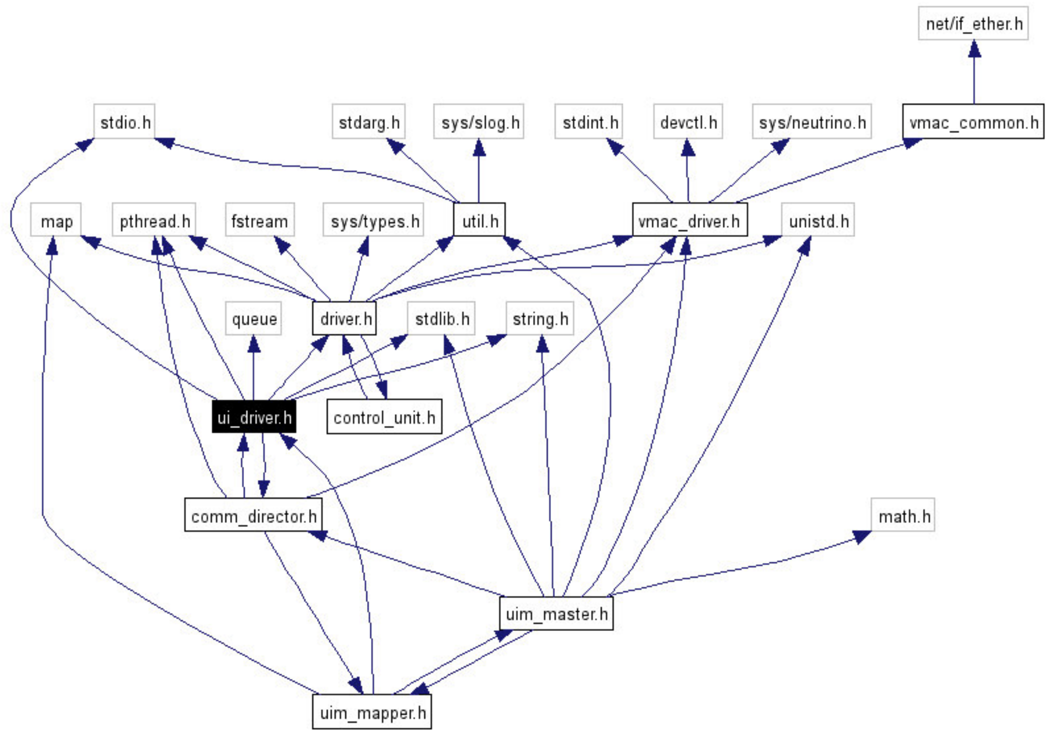


Figure 23: Dependency graph for ui_driver.h

This graph shows which files directly or indirectly include this file:

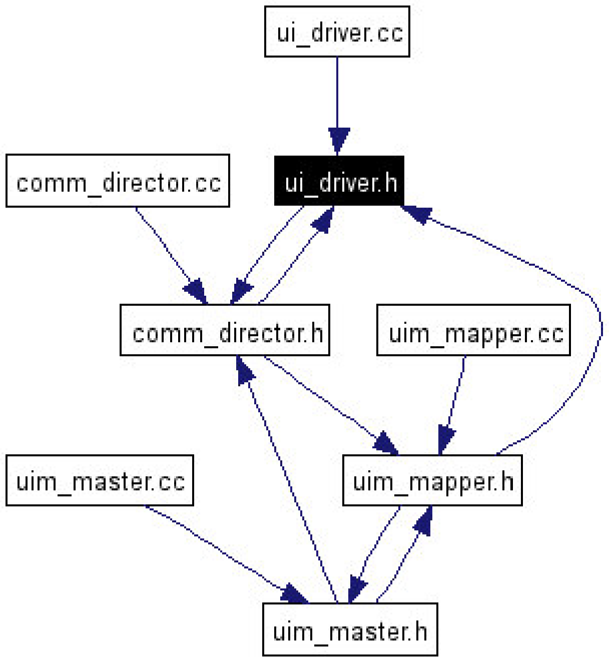


Figure 24: Files that include ui_driver.h

CLASSES

struct user_update

This is the format of the updates from a UI device. Definition at line 35 of file ui_driver.h.

Public Attributes

opcode_t op

This is the opcode, not to be confused with the opcodes from vmac_driver.h.

pid_t pid

This is the pid of the driver that is to receive the update.

uint8_t data [MAX_UPDATE_SIZE]

This is the payload of the update.

struct netpkt_t

This is a struct to be used in the queue of MTU sized network packets. Definition at line 47 of file ui_driver.h.

Public Member Functions

netpkt_t ()

Simple default constructor to initialize the structs members to reasonable values.

class UiDriver

This class is a UI driver.

Public Attributes

uint8_t data [VMAC_MTU]

This is the data that will fit into one VMAC network packet.

int len

This is the length of data.

DEFINES

#define RABBIT_INIT_LEN VMAC_MTU

This is the length of the initial rabbit initialization packet.

#define RABBIT_TIMEOUT_OPCODE 0x0001

This is the timeout opcode for the Rabbit.

#define RABBIT_TIMEOUT 5000

This is the timeout value for the rabbit board. It is measured in milliseconds.

Table 17: Classes and macros for ui_driver.h

VMAC/SRC/DRIVER/UIM/UIM_MAPPER.CC

Map between driver id and UI server IDs.

Author:

Michael Brailsford
created : Sat Aug 09 18:11:52 MDT 2003
copyright : 2003
Revision: 1.8

Definition in file uim_mapper.cc.

#include "uim_mapper.h"

Include dependency graph for uim_mapper.cc:

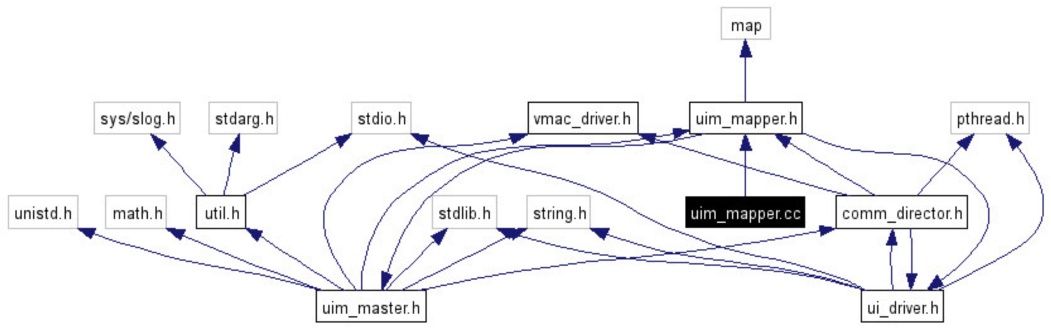


Figure 25: Dependency graph for ui_mapper.cc

VMAC/SRC/DRIVER/UIM/UIM_MAPPER.H

Contains mapping from drivers to UI server IDs.

Author:
Michael Brailsford
created: Sat Aug 09 18:04:31 MDT 2003
copyright: (c) 2003 Michael Brailsford
Revision: 1.7

Definition in file uim_mapper.h.

```
#include <map>
#include "uim_master.h"
#include "ui_driver.h"
```

Include dependency graph for uim_mapper.h shown in Figure 26:

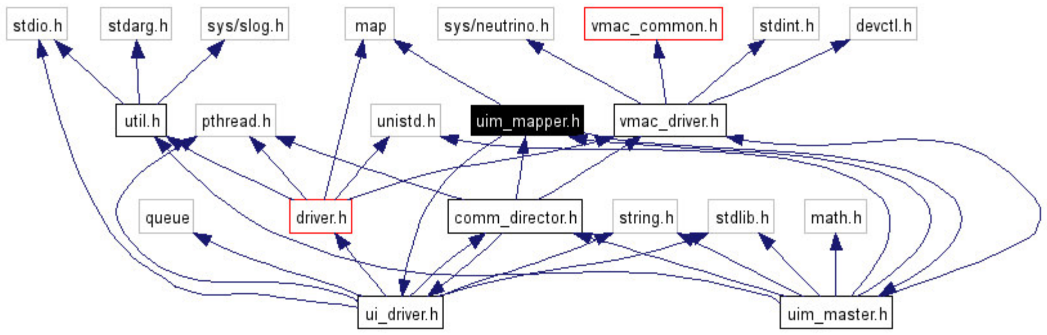


Figure 26: Dependency graph for ui_mapper.h

This graph shows which files directly or indirectly include this file:

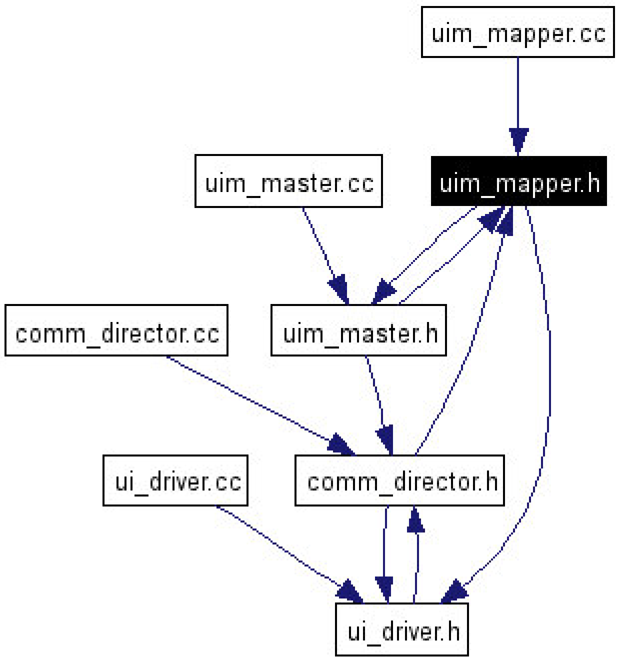


Figure 27: Files that directly include ui_mapper.h

CLASSES

class UimMapper

Class where the mapping all happens.

DEFINES

#define ui_drv_map map<int, UiDriver *>

Map the UI device ID from npm to the UiDriver object which maintains information about the UI.

#define driver_chan_map map<pid_t, struct driver_info *>

Map the from device driver pids to device driver channels.

Table 18: Classes and macros for ui_mapper.h

VMAC/SRC/DRIVER/UIM/UIM_MASTER.CC

Keeps track of all UI servers.

Author:

Michael Brailsford
created : Mon Aug 11 23:17:24 MDT 2003
copyright : 2003 Michael Brailsford
Revision: 1.19

Definition in file uim_master.cc.

#include "uim_master.h"

Include dependency graph for uim_master.cc:

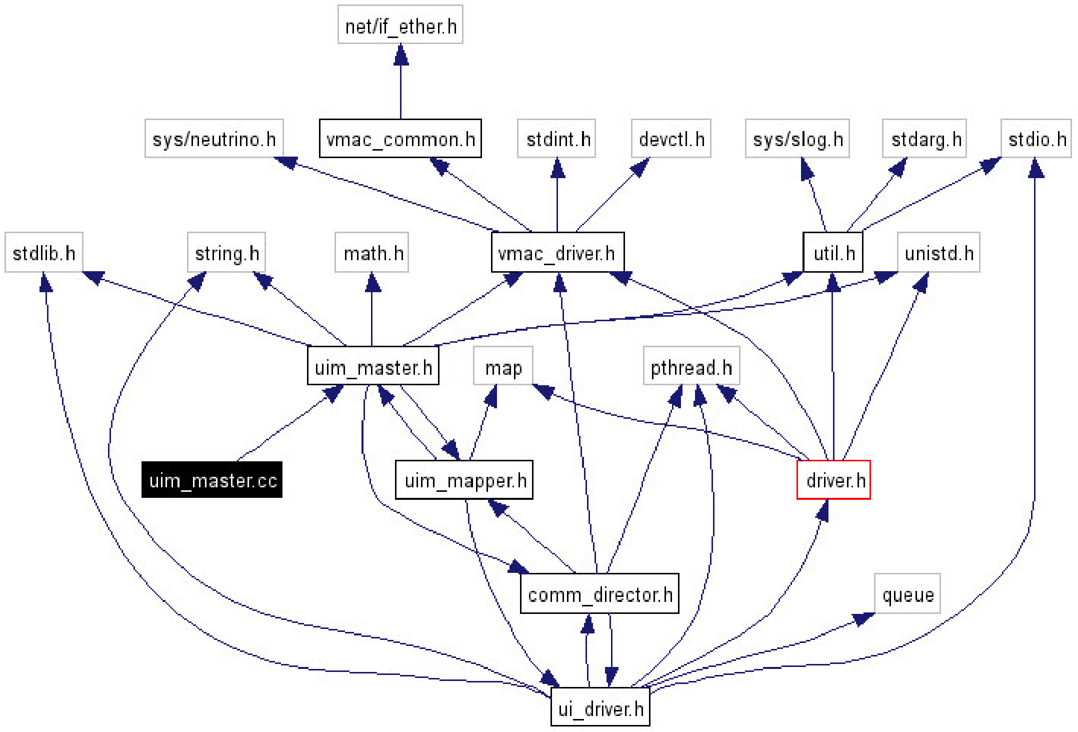


Figure 28: Dependency graph for ui_master.cc

FUNCTIONS

void *	run_uim (void *uim_master)
	Instance of UIM.

Table 19: Functions for ui_master.cc

VMAC/SRC/DRIVER/UIM/UIM_MASTER.H

Header for keeping track of UI servers. This is the main file where all things are coordinated together. This can be thought of as the master library file that supports the executable.

Author:

Michael Brailsford
created: Mon Aug 11 22:57:27 MDT 2003

copyright: (c) 2003 Michael Brailsford
Revision: 1.14

Definition in file uim_master.h.

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <math.h>

#include "util.h"

#include "comm_director.h"

#include "uim_mapper.h"

#include "vmac_driver.h"

Include dependency graph for uim_master.h:

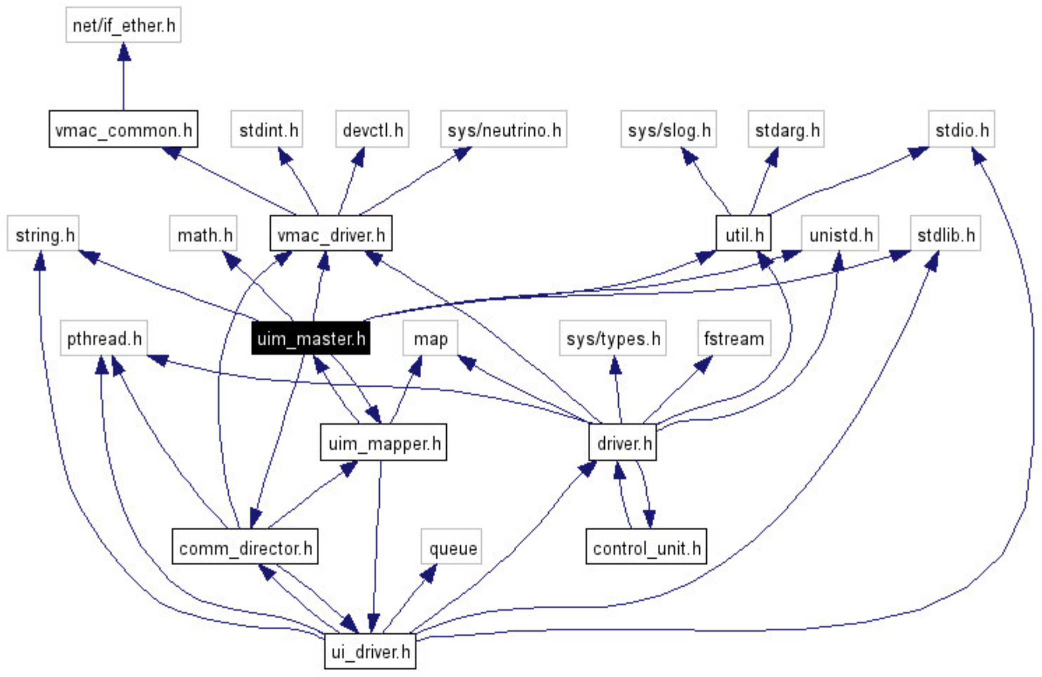


Figure 29: Dependence graph for ui_master.h

This graph shows which files directly or indirectly include this file:

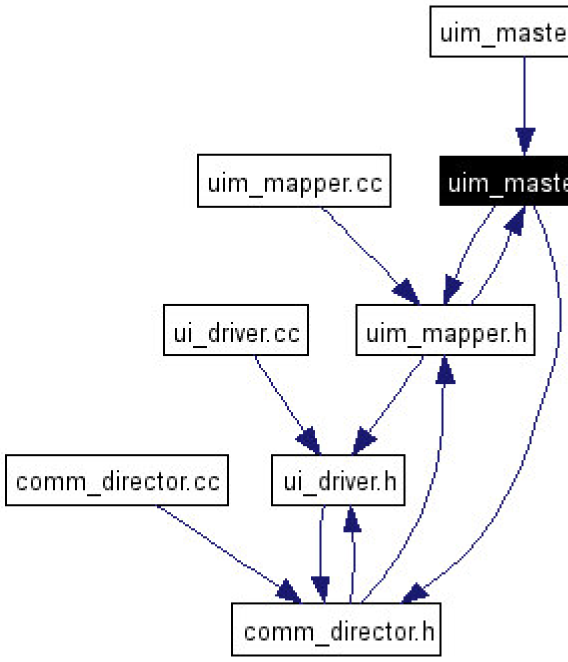


Figure 30: Files that include ui_master.h

CLASSES

struct driver_comm_hdr

This is the data sent to a UI device when a device driver has an update of any sort.

Public Attributes

pid_t pid

This is the pid of the device driver.

opcode_t opcode

This is the opcode that the UI device will understand.

int len

This is the length of the packet

struct driver_info

This struct contains information releated to a device driver. Specifically it stores the channel id used for communication with the device driver and it also stores a copy of the device driver's XML template for future use. Definition at line 47 of file uim_master.h.

Public Member Functions

driver_info ()

Default constructor to initialize the driver_info struct to the proper values.

driver_info (int coid)

Constructor to initialize the connection_id to the coid passed in.

Public Attributes

int connection_id

This is the connection_id for communications with this device driver.

int len

This is the length of the driver's xml_template.

bool sent_template

This lets us know if the driver has sent us its XML template yet.

uint8_t xml_template [MAX_XML_TEMPLATE_SIZE]

This is the actual XML template for the driver.

class UimMaster

This is the master control of the UIM. It is really not much more than a Facade pattern. It provides the interface to the UIM that the NPM will use. It hides the complexity of the entire UIM. The UimMaster provides all the registration and deregistration functionality to the UIM. The UimMaster will create a new UiDriver for each NPM_UIM_REG_DRIVER pulse sent from the NPM to the UimMaster. It will also register a Device Driver with the mapper for each NPM_UIM_REG_DRIVER message recieved. It will also likewise deregister a device driver for each NPM_UIM_DEREG_DRIVER message recieved.

The UimMaster is a Reciever in the Command pattern. Th NPM is the Invoker. After system initialization *the UimMaster listens on the channel with the NPM for the NPM_UIM_EVENT_PULSE which is a command from the NPM to query it for the data that is waiting. In the reply of the message the the UimMaster sends in response to the NPM_UIM_EVENT_PULSE will be one of the opcodes that the UimMaster understands. The UimMaster initially begins with no UI devices, nor device drivers. Definition at line 97 of file uim_master.h.*

Public Member Functions

UimMaster ()

Default constructor.This is the first time that the mapper and comm are instantiated, so this class will also be the one in charge of deallocating them.

~UimMaster ()

Default destructor.

void register_ui (int id)

This registers a new UI driver and places it in the mapper. register_ui() allocates all memory for a UiDriver object, it is the responsibility of deregister_ui() to deallocate the memory. id is the NPM assigned id for the UI device.

void deregister_ui (int id)

This deregisters a UI driver from the mapper.

void register_device_driver (pid_t pid, int chid)

Register a new device driver with the UIM and place all information in the mapper. pid is the pid of the driver. chid is the open channel of communication that the driver has created to communicate with the UIM. This method must call ConnectAttach on this chid to send messages to the driver.

void run ()

This is used to begin execution of the UimMaster control.

void deregister_device_driver (pid_t pid)

Deregister a the device driver specified with the pid.

	void shutdown_uim () *Shutdown the UIM.*
DEFINES	
#define	UIM_DEBUG *A debug flag to conditionally compile somethings that only make sense while debugging.*
#define	UIM_UI_ID 0 *UIM ID for the UIs.*
VARIABLES	
driver_comm_hdr	packed *This is the data sent to a UI device when a device driver has an update of any sort.*

Table 20: Classes, macros and variables defined in ui_master.h

DEVICE DRIVERS

The device drivers are designed to be independent executable files. When a peripheral device responds to a new device quiry the configuration manager launches the new driver. The driver establishes the needed communication channels with the configuration manager as well as the UIM. After this initial setup the driver execution blocks on receipt of a feedback packet from a peripheral device. When one is received the driver then services the feedback. This may include sending a control value to the configuration manger to be sent to the device as well as sending a status update to the UIM for display on associated UI pages.

The following sections provide additional explanation on working with device drivers followed by source file references.

When a vmac device is attached to the vmac network, the network protocol module (npm) either knows about it or does not. A device may be in either of these states at any time. Initially the device will be unkown, and will need to make the transition into the known state by responding to a discovery packet. Generally, once the device is known it will remain in that state until network connectivity is broken. However, it is possible to transition into the unkown state, in which case the device will not receive communicae from the npm. When the device has detected that it is in the unkown state (e.g. via a watchdog timer), it should then respond to a discovery packet in order to make the transition into the known state again.

The npm will periodically send a discovery packet to the broadcast address. The first two data bytes will be 0xff and 0x01. The first byte (0xff) indicates that this is a vmac network system packet; it does not come from the device driver, but rather from the vmac npm itself. The second byte (0x01) is the opcode for the discovery packet. A device should only respond to a discovery packet when it wishes to transition from the unkown state to the known state.

```
0000  FF FF FF FF FF FF 00 40 05 7C 8E FB 00 02 FF 01   .......@.|......
```

The response to a discovery packet is addressed to the host's MAC address, as usual, and contains four pieces of data: 0xff, 0x10, the name of the driver to be executed, a four-byte major id, and a four-byte minor id. The major id is unique among devices using a specific driver, the minor id is unique among devices associated with the major id. For example, given two mills with four motors that use the same driver name "mill", the first motor on the second mill might have major id 1 and minor id 0, while the third motor on the first mill might have major id 0 and minor id 2. (endianness does not matter for these ids, so long as they are unique) User interfaces to be managed by the vmac user interface manager should specify "UI" as the driver name, and need not specify major and minor ids (if you do they will be ignored). For example:

```
0000  00 40 05 7C 8E FB 00 40 05 7C 8E FE 00 04 FF 10   .@.|...@.|......
0010  55 49 00                                          UI.
```

A 'foo' device would respond like this, assuming that the driver executable is also named 'foo', and the device has major id 42 and minor id 12:

```
0000  00 40 05 7C 8E FB 00 40 05 7C 8E FE 00 04 FF 10   .@.|...@.|......
```

```
0010  66 6F 6F 00 00 0C 00 2A                    FOO.....
```

The npm will not send a special packet acknowledging receipt of the discovery response. If it is successfully received, you will know because you will begin receiving transmissions from your device driver (assuming it works, of course).

This all the device discovery documentation

Developing a Device Driver

See the documentation for the Driver and ControlUnit classes, and the example implementation: IOBoardDriver and IOBoardControlUnit.

Do the following to develop a new device driver:

- Subclass Driver
    - Define a constructor that calls Driver::read_template_file() and Driver::add_control_unit()
- Subclass ControlUnit
    - Implement
        - ControlUnit::reset()
        - ControlUnit::control_loop()
        - ControlUnit::process_update()
- Write a main routine similar to the following:

INT MAIN(INT ARGC, CHAR **ARGV)

```
{
LOG_DEBUG2("*** MY DRIVER STARTING");
PTHREAD_T T;
MYDRIVER *DRIVER = NEW MYDRIVER(ATOI
(DEVICE_ID_ARG));
// WITH ONLY ONE CONTROL UNIT, THERE'S NOT MUCH
POINT TO USING THREADS, BUT
// IF YOU HAVE MORE THERE WILL BE.
PTHREAD_CREATE(&T, NULL, START_DRIVER, DRIVER);

PTHREAD_JOIN(T, NULL);

LOG_DEBUG2("*** MY DRIVER TERMINATING");

DELETE DRIVER;

DRIVER = NULL;
}
```

File Reference

For the demonstration there were three drivers written. Included in this documentation are the series of files used for the drivers and their documentation. The driver files and a short description of each is shown in Table 21.

vmac/src/driver/control_unit.cc	Template to contain device-specific control code
vmac/src/driver/control_unit.h	Template header for device-specific control code
vmac/src/driver/driver.cc	Driver template that contains all accept the device control unit
vmac/src/driver/driver.h	Driver template header
vmac/src/driver/ClothesWasher/ClothesWasher.cc	Clotheswasher control coordination
vmac/src/driver/ClothesWasher/ClothesWasher.h	Clotheswasher header declaring the control unit
vmac/src/driver/ClothesWasher/PIDControl.cc	Implementation of the PIDControl loop class
vmac/src/driver/ClothesWasher/PIDControl.h	Definition of the PIDControl loop class
vmac/src/driver/ClothesWasher/TrajectoryGenerator.cc	Implementation of the TrajectoryGenerator class
vmac/src/driver/ClothesWasher/TrajectoryGenerator.h	Definition of the TrajectoryGenerator class
vmac/src/driver/dishwasher/dishwasher.cc	Dishwasher implementation file with control unit
vmac/src/driver/dishwasher/dishwasher.h	Dishwasher header file

Table 21: Device driver files for the VMAC system

VMAC/SRC/DRIVER/CONTROL_UNIT.CC

Template to contain device-specific control code.

Author:
Michael Brailsford
created : Thu Oct 16 12:26:42 UTC 2003
copyright : 2003 Michael Brailsford
Revision: 1.11

Definition in file control_unit.cc.

```
#include "control_unit.h"
```

Include dependency graph for control_unit.cc:

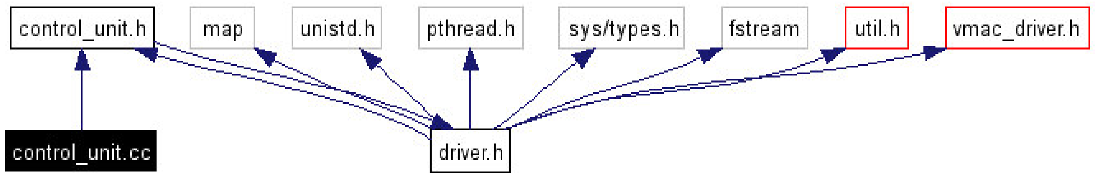


Figure 31: Dependency graph for control_unit.cc

FUNCTIONS

void
 * start_me (void *control_unit)
 Function used to start the ControlUnit's active operation in a seperate thread.

VMAC/SRC/DRIVER/CONTROL_UNIT.H

Template header for device-specific control code.

Author:

Michael Brailsford
created : Thu Oct 16 12:00:56 UTC 2003
copyright : (c) 2003 Michael Brailsford
Revision: 1.12

Definition in file control_unit.h.

This is the base class for all control units within a composite driver.

A control unit functions as the driver for a specific piece of hardware in the VMAC system, but it also coordinates its operation with the master controller of the larger machine of which is/may be a part. This provide the means by which a large machine with several components motors may be driven in the VMAC system while providing a single clean interface. A control unit communicates directly with the NPM to control its device. It communicates with the master driver object to send information to the UI devices.

The process of registering a control unit is as follows.

1. The device responds to the NPM with its Major and Minor device IDs.
2. The NPM check to see if the Major device is already running.
    - If it is not running, the NPM spawns it.
3. The NPM then sends a registration event pulse to the Major device controller which corresponds to the Major ID.
4. The master device controller then sends the NPM a DRV_NPM_GET_COMMAND message
5. The NPM responds to the DRV_NPM_GET_COMMAND message with NPM_DRV_REG_CU message, with the Minor ID in the message.
6. The master controller looks up which ControlUnit object corresponds to the Minor ID passed in the NPM_DRV_REG_CU message.
7. The master controller then calls ControlUnit->run(int) for the ControlUnit which corresponds to the Minor ID.
8. The master controller then increments its count of active ControlUnits.
9. The ControlUnit then sends the NPM the DRV_NPM_REG message with all information the NPM needs to schedule the device for service.
    - The NPM needs only ControlUnit->devid and ControlUnit->priority from ControlUnits.
10. The ControlUnit then enters normal operation.

ControlUnits do not get deregistered in the normal sense. They simply get reset and await another call to run(). This allows a device in the VMAC system to be plugged in and out at any time and still function properly. It should be noted that there are two cases of this when a ControlUnit can be removed from the VMAC system.

1. The device being unplugged is the only/last device being serviced by the master control.
2. The device being unplugged is not the only/last device being serviced by the master control.

The process of ControlUnit deregistration is as follows.

1. The device goes AWOL (gets unplugged, is too slow to respond, etc...)
2. The NPM will respond with an error the next time the corresponding ControlUnit sends a message to it.
3. The ControlUnit will call Driver->inactivate_cu() which will decrement the count of active ControlUnits.
4. The ControlUnit will reset itself to a state in which it is waiting for the run (int) method to be called again.
5. If the count of active ControlUnits in the Master controller reaches zero it will send the NPM the DRV_NPM_DEREG message.
    - The Master controller will then terminate its execution.
6. If the count of active ControlUnits in the Master controller is not zero, the Master continues execution.

A few points of consideration.

- Since a ControlUnit cannot function without the NPM, any error in communications with the NPM will result in the ControlUnit initializing deregistration.

#include "driver.h"

Include dependency graph for control_unit.h shown in Figure 32:

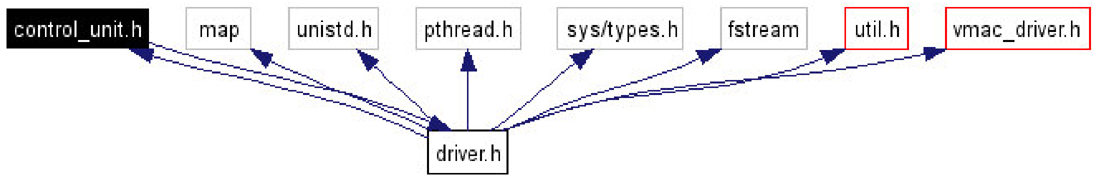


Figure 32: Dependency graph for control_unit.h

This graph shows which files directly or indirectly include this file:

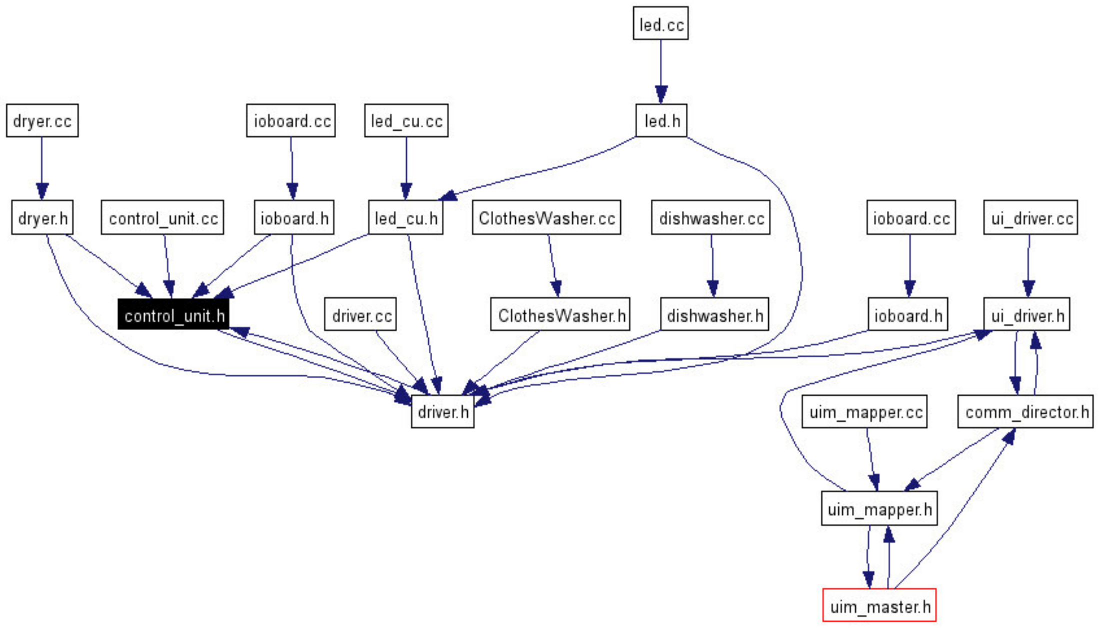


Figure 33: Files that include control_unit.h

NAMESPACES

namespace **std**

Classes

class	ControlUnit *This is the base class for all control units within a composite driver.*

Defines

#define	DEFAULT_PRIORITY 0 *This is the default priority with which ControlUnits will register with the NPM.*

Functions

void *	start_me (void *cu) *Function used to start the ControlUnit's active operation in a seperate thread.*

Table 22: Memory structure of control_unit.h

The following is a description of the member functions of the ControlUnit class.

Public Member Functions

	ControlUnit (Driver *master) *Constructor.*
virtual	~ControlUnit () *Default destructor.*
void	run (int devid) *Run the ControlUnit's control loop in a seperate thread.*
virtual void	control_loop ()=0 *This is the control loop for the ControlUnit.*
virtual void	operator= (const ControlUnit &cu) *This is required by the STL's map.*
virtual void	reset ()=0 *Resets the ControlUnit to the initial state waiting for void run(int) to be called.*
virtual void	process_update (struct set_point *sp)=0 *This processes the set_point passed in.*
int	get_devid () *Get the device id for this ControlUnit.*

Protected Member Functions

void	rabbit_init (short timeout) *Prepares the rabbit initialization packet.*

Protected Attributes

Driver *	master *This is the master controller of the machine.*
int	npm_chid *This is the channel id for the NPM.*
int	npm_coid *This is the connection id over which we will be communicating with the NPM.*
int	devid *communications with the NPM.*
int	priority *This is the scheduling priority of this ControlUnit.*
uint16_t	timeout *This is the timeout value. This value is in milliseconds.*
cu_npm_reg_msg	dev_reg *This is the dev_reg message that needs to be sent to the NPM when run () is called.*
cu_npm_txrx_msg	dev_txrx *This is the dev_txrx message that is sent to the NPM during the course of normal operation.*

Table 23: Member functions of the ControlUnit class

vmac/src/driver/driver.cc

Driver template that contains all accept the device control unit.

Author:

Michael Brailsford
created : Thu Oct 09 11:09:15 MDT 2003
copyright : 2003 Michael Brailsford
Revision: 1.20

Definition in file driver.cc.

#include "driver.h"

Include dependency graph for driver.cc:

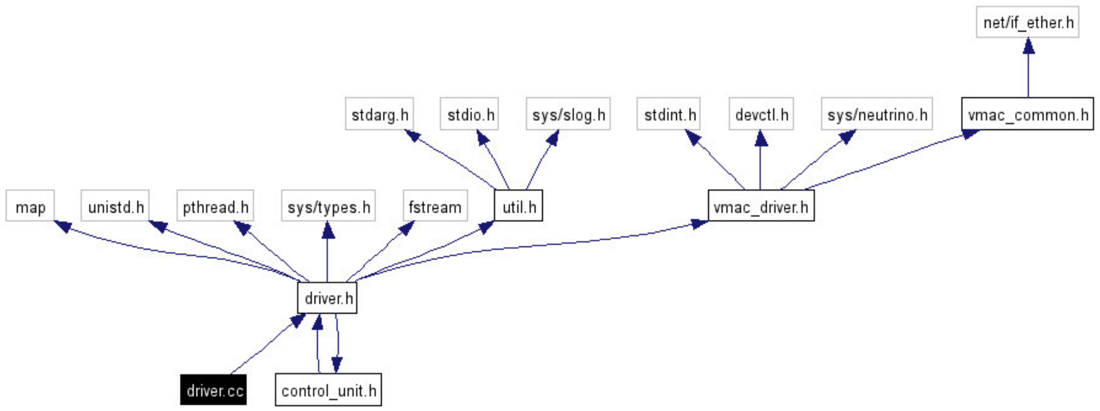


Figure 34: Dependency graph for driver.cc

FUNCTIONS

void * start_driver (void *driver)

This function's responsiblity is to simply call driver->run().

void * start_uim_control_loop (void *driver)

The responsibility of this function is to call driver->uim_control_loop().

VMAC/SRC/DRIVER/DRIVER.H

Driver template header.

Contains structure needed for a driver.

Author:

Michael Brailsford
created : Thu Oct 09 10:47:03 MDT 2003
copyright : (c) 2003 Michael Brailsford
Revision: 1.18

Definition in file driver.h.

This is the main Driver base class.

All drivers should have this class as their parent class. Any modifications to the functionality will be done in the subclasses of this class.

This should be a fairly complete Base class. All that will be required will be to override those methods that require special handling. Below is outlined each method and what it does, so that someone else can make sense of the Driver architecture, and write one themselves. The following list contains the purpose of the method, its current implementation and some possible reasons to override it in a child class.

- start_control_unit(ControlUnit *, int) -- This starts the single ControlUnit passed in.
    - The current implementation is to simply start the ControlUnit passed in with no dependency checking of any kind.
    - If a ControlUnit's functionality requires that another ControlUnit already be in operation, then this will need to be overridden to provide that checking.

- shutdown() -- This shutsdown the entire system this Driver manages.
    - The current implementation is to iterate over the control_units map and stop each ControlUnit.
    - If there is a required shutdown sequence this will need to be overridden.

- run() -- This is the main entry point for the Driver.
    - The current implementation should be sufficient for all cases. It intializes all channels and sig_event structures required for proper Driver function.

    - I don't see why this might need to be overridden, but it can be done. Most perceived needs to override this method are better handled in overriding registration_control_loop() and uim_control_loop().

- registration_control_loop() -- The control loop that sits and waits for registration events from the NPM
    - The current implementation should be sufficient for most cases.
    - If there are ControlUnit dependencies or start up sequences then this will need to be overridden to enforce them. Typically if a dependency can be handled in start_control_unit() than it should be. This is for things that are not easily handled by start_control_unit().

- uim_control_loop() -- The control loop that communicates with the UIM.
    - The current implementation should suffice for most if not all Drivers.
    - There may be some necesary UIM/Driver coupling.

- deregister_control_unit(int) -- Does cleanup of a running ControlUnit.
    - The current implementation simply sets the thread_id in the control_units map to CU_IDLE_ID, with no dependency checking, or shutting down dependant ControlUnits. This is called from the ControlUnit itself as it is about to reset itself to a state in which it can be started again, so not much cleanup needs to occur.
    - If there are ControlUnit dependencies then this will need to be overridden to enforce those dependencies and ensure a valid state for the device.

How This Driver Gets Initiated:

- When the NPM receives notice that a new device driver needs to be registered it will spawn a process which instantiates an instance of this Driver class or one of its subclasses.
- The NPM will then send a pulse to the new Driver instance.
- The new Driver object will respond with a msg with opcode DRV_NPM_GET_COMMAND.
- The NPM will then reply with NPM_DRV_REG and the associated data for the Driver object to register a corresponding control unit.
    - This will be the case for any Driver no matter how many control units it needs. So the case where a single motor comprises an entire machine, the NPM will start up the Driver and then register that one control unit.

```
#include <map>
#include <unistd.h>
#include <pthread.h>
#include <sys/types.h>
#include <fstream>
#include "util.h"
#include "vmac_driver.h"
#include "control_unit.h"
```

Include dependency graph for driver.h:

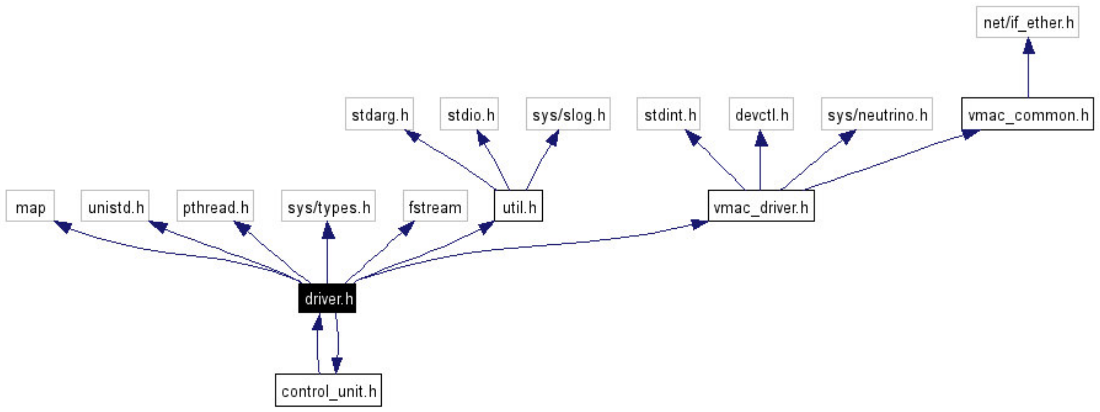


Figure 35: Dependence graph for driver.h

This graph shows which files directly or indirectly include this file:

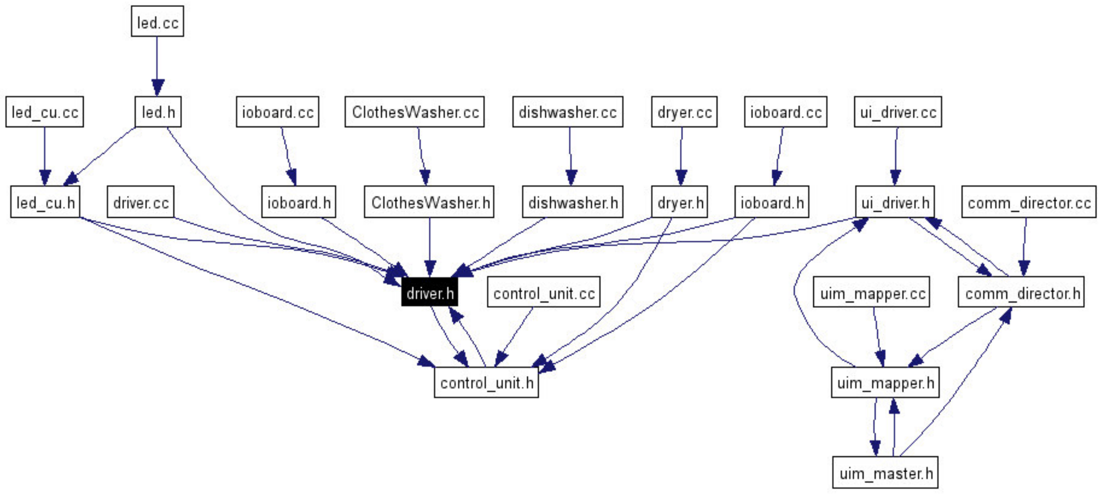


Figure 36: Files including driver.h

The memory structure of driver.h is shown next.

CLASSES

struct set_point

This is the 'update' that a device needs to send to a UI device. It is also the format of the data that the UI will use to send updates to the driver. Definition at line 57 of file driver.h.

Public Attributes

opcode_t op

This is the opcode to inform the UI/ControlUnit what to do with this set_point.

int minor_id

This is the minor id of the device within the Master device.

uint8_t id

This is the id of the data point within the device.

class Driver

This is the main Driver base class. All drivers should have this class as their parent class. Any modifications to the functionality will be done in the subclasses of this class.

This should be a fairly complete Base class. All that will be required will be to override those methods that require special handling. Below is outlined each method and what it does, so that someone else can make sense of the Driver architecture, and write one themselves. The following list contains the purpose of the method, its current implementation and some possible reasons to override it in a child class.

- *start_control_unit(ControlUnit *, int) -- This starts the single ControlUnit passed in.*
  - *The current implementation is to simply start the ControlUnit passed in with no dependency checking of any kind.*
  - *If a ControlUnit's functionality requires that another ControlUnit already be in operation, then this will need to be overridden to provide that checking.*
- *shutdown() -- This shutsdown the entire system this Driver manages.*
  - *The current implementation is to iterate over the control_units map and stop each ControlUnit.*
  - *If there is a required shutdown sequence this will need to be overridden.*
- *run() -- This is the main entry point for the Driver.*
  - *The current implementation should be sufficient for all cases. It intializes all channels and sig_event structures required for proper Driver function.*

    - *I don't see why this might need to be overridden, but it can be done. Most perceived needs to override this method are better handled in overriding registration_control_loop() and uim_control_loop().*
- *registration_control_loop() -- The control loop that sits and waits for registration events from the NPM*
    - *The current implementation should be sufficient for most cases.*
    - *If there are ControlUnit dependencies or start up sequences then this will need to be overridden to enforce them. Typically if a dependency can be handled in start_control_unit() than it should be. This is for things that are not easily handled by start_control_unit().*
- *uim_control_loop() -- The control loop that communicates with the UIM.*
    - *The current implementation should suffice for most if not all Drivers.*
    - *There may be some necesary UIM/Driver coupling.*
- *deregister_control_unit(int) -- Does cleanup of a running ControlUnit.*
    - *The current implementation simply sets the thread_id in the control_units map to CU_IDLE_ID, with no dependency checking, or shutting down dependant ControlUnits. This is called from the ControlUnit itself as it is about to reset itself to a state in which it can be started again, so not much cleanup needs to occur.*
    - *If there are ControlUnit dependencies then this will need to be overridden to enforce those dependencies and ensure a valid state for the device.*

How This Driver Gets Initiated:

- *When the NPM receives notice that a new device driver needs to be registered it will spawn a process which instantiates an instance of this Driver class or one of its subclasses.*
- *The NPM will then send a pulse to the new Driver instance.*
- *The new Driver object will respond with a msg with opcode DRV_NPM_GET_COMMAND.*
- *The NPM will then reply with NPM_DRV_REG and the associated data for the Driver object to register a corresponding control unit.*
    - *This will be the case for any Driver no matter how many control units it needs. So the case where a single motor comprises an entire machine, the NPM will start up the Driver and then register that one control unit.*

Definition at line 151 of file driver.h.

Public Member Functions

Driver (int id)

Default constructor.

virtual ~Driver ()

Virtual destructor to properly destroy memory.

virtual void run ()

This begins execution of the driver.

virtual void registration_control_loop ()

This is the method which listens for a pulse over the NPM channel to notify the driver of a registration/deregistration event.

virtual void uim_control_loop ()

This is the method which listens to the UIM and sends updates and receives input to/from the user interfaces through the UIM.

virtual void deregister_control_unit (int minid)

This is not called in response to a message from the NPM, This is called by a control unit when it goes away.

int get_npm_coid ()

ControlUnits need rapid access to this member.

void deactivate_cu ()

Decrements the count of active ControlUnits.

pid_t get_my_pid ()

Grab the master's pid.

pid_t get_unc_coid ()

Return the Update Notification Connection ID.

Protected Member Functions

*void read_template_file (char *fname)*

Reads the XML template file from disk.

virtual void shutdown ()

Shutsdown the driver and all it associated ControlUnits.

*void add_control_unit (int devid, ControlUnit *cu)*

Adds a ControlUnit to the map of ControlUnits with the id of devid.

virtual void start_control_unit (int devid, int minor_id)

Start the ControlUnit with minor id of minid.

void activate_cu ()

Increments the count of active ControlUnits.

Protected Attributes

int id

This is the id that the NPM passes on the command line.

pthread_t my_t

This is the thread id of the driver.

pid_t my_pid

This is the pid of this instance of the Driver, it is required for all communication.

int npm_coid

The connection id for sending messages to the NPM.

int uim_chid

This is the channel id for communication with the UIM.

int unc_coid

The connection id of the Update Notification Channel (UNC), on which update notification pulses will be sent.

CU_MAP control_units

This is a map of ControlUnit objects which represent a driver for a motor in the

SEND_BUF_MAP send_buffer

This is the buffer of set points to send to the UIM.

int active_cus

The number of currently active ControlUnits.

int event_chid

This is the channel id that we are creating to listen for the registration event from the NPM.

int event_coid

This is the cconnection id that we are creating to listen for the registration_ev or shutdown_ev events from the NPM.

uint8_t ui_data [MAX_XML_TEMPLATE_SIZE]

This is the storage area for all the XML template.

DEFINES

#define DEVICE_ID_ARG argv[1]

This is the command line argument that is the device id of the master device.

#define CU_IDLE_ID 0

This is a thread ID that lets us know that a ControlUnit has no associated thread, and it is therefore not running.

#define CU_MAP map<int, pair<pthread_t, ControlUnit *> >

This is a convenience macro to make using this map easier.

#define SEND_BUF_MAP map<int, struct set_point>

This a convenience macro to make using this map easier.

ENUMERATIONS

enum { INT_TYPE, LONG_TYPE, FLOAT_TYPE }

This is an enumeration to describe the type of data in a set_point.

Enumeration values:

INT_TYPE	*The type in a set_point is an int.*
LONG_TYPE	*The type in a set_point is an long.*
FLOAT_TYPE	*The type in a set_point is an float.*

FUNCTIONS

void * start_driver (void *driver)

This function's responsiblity is to simply call driver->run().

void * start_uim_control_loop (void *driver)

The responsibility of this function is to call driver->uim_control_loop().

VARIABLES

set_point packed

This is the 'update' that a device needs to send to a UI device.

Table 24: Memory structure of driver.h

The following is more detailed documentation of the driver class contained within driver.h.

Public Member Functions

	Driver (int id)
	Default constructor.
virtual	~Driver ()
	Virtual destructor to properly destroy memory.
virtual void	run ()
	This begins execution of the driver.
virtual void	registration_control_loop ()
	This is the method which listens for a pulse over the NPM channel to notify the driver of a registration/deregistration event.
virtual void	uim_control_loop ()
	This is the method which listens to the UIM and sends updates and receives input to/from the user interfaces through the UIM.
virtual void	deregister_control_unit (int minid)
	This is not called in response to a message from the NPM, This is called by a control unit when it goes away.
int	get_npm_coid ()
	ControlUnits need rapid access to this member.
void	deactivate_cu ()
	Decrements the count of active ControlUnits.
pid_t	get_my_pid ()
	Grab the master's pid.
pid_t	get_unc_coid ()
	Return the Update Notification Connection ID.

Protected Member Functions

void	read_template_file (char *fname)
	Reads the XML template file from disk.
virtual void	shutdown ()
	Shutsdown the driver and all it associated ControlUnits.
void	add_control_unit (int devid, ControlUnit *cu)
	Adds a ControlUnit to the map of ControlUnits with the id of devid.
virtual void	start_control_unit (int devid, int minor_id)
	Start the ControlUnit with minor id of minid.
void	activate_cu ()
	Increments the count of active ControlUnits.

Protected Attributes

int	id
	This is the id that the NPM passes on the command line.

pthread_t	my_t *This is the thread id of the driver.*
pid_t	my_pid *This is the pid of this instance of the Driver, it is required for all communication.*
int	npm_coid *The connection id for sending messages to the NPM.*
int	uim_chid *This is the channel id for communication with the UIM.*
int	unc_coid *The connection id of the Update Notification Channel (UNC), on which update notification pulses will be sent.*
CU_MAP	control_units *This is a map of ControlUnit objects which represent a driver for a motor in the driver.*
SEND_BUF_MA P	send_buffer *This is the buffer of set points to send to the UIM.*
int	active_cus *The number of currently active ControlUnits.*
int	event_chid *This is the channel id that we are creating to listen for the registration event from the NPM.*
int	event_coid *This is the cconnection id that we are creating to listen for the registration_ev or shutdown_ev events from the NPM.*
uint8_t	ui_data [MAX_XML_TEMPLATE_SIZE] *This is the storage area for all the XML template.*

VMAC/SRC/DRIVER/CLOTHESWASHER/CLOTHESWASHER.CC

Clotheswasher control coordination.

The details of the control system are found in PIDControl.h

Author:

Tyler Davis Based on the ioboard code written by Hans Fugal Used to drive the Maytag Neptune clothes washer

Definition in file ClothesWasher.cc.

#include "ClothesWasher.h"

#include "rabbit.h"

#include <netinet/in.h>

#include "PIDControl.h"

Include dependency graph for ClothesWasher.cc:

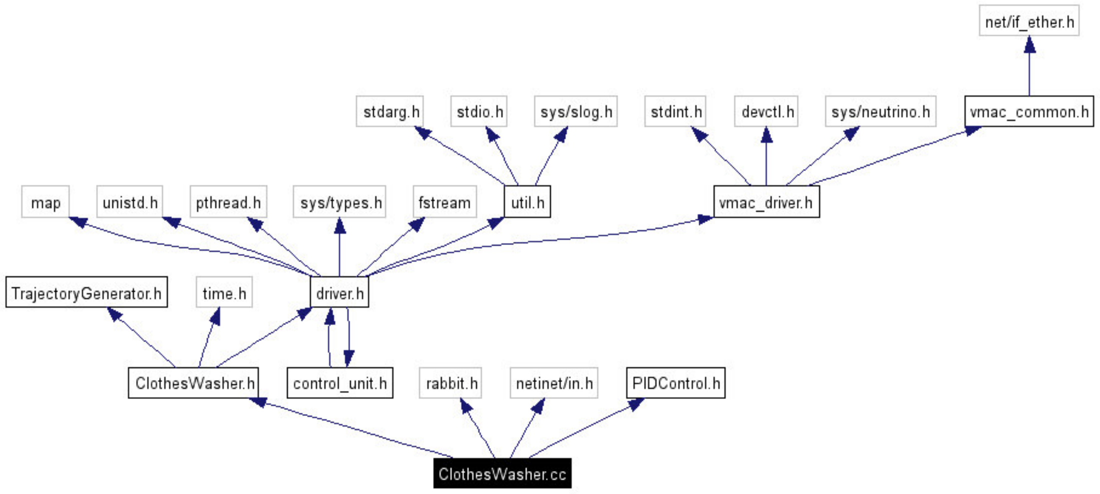


Figure 37: Dependency graph for ClothesWasher.cc

VMAC/SRC/DRIVER/CLOTHESWASHER/CLOTHESWASHER.H

Clotheswasher header declaring the control unit.

Author:
Hans Fugal
Tyler Davis

Definition in file ClothesWasher.h.

#include "driver.h"

#include "TrajectoryGenerator.h"

#include <time.h>

Include dependency graph for ClothesWasher.h:

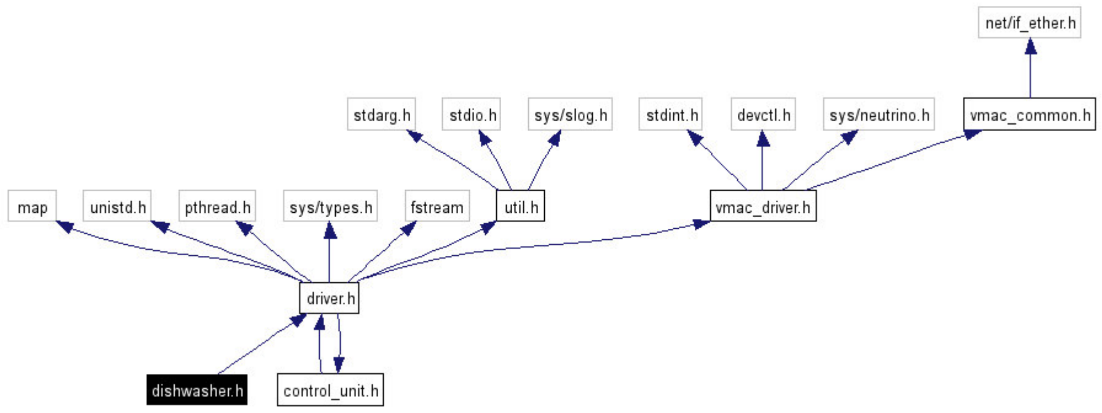


Figure 38: Dependency graph for ClothesWasher.h

This graph shows which files directly or indirectly include this file:

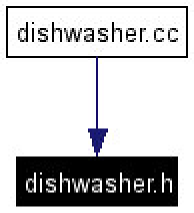


Figure 39: Files that include ClothesWasher.h

Classes

class ClothesWasherDriver
ioboard driver

Class member documentation is as follows:

Public Member Functions

	ClothesWasherDriver (int id) *Constructor.*
void	control_thread () *This control loop is called by run().*

PROTECTED MEMBER FUNCTIONS

virtual void	ui_recv (set_point *sp) *This "update" comes from the UI.*

Table 25: Memory structure of ClothesWasher.h

VMAC/SRC/DRIVER/CLOTHESWASHER/PIDCONTROL.CC

Implementation of the PIDControl loop class.

PID control law used for motor speed control in washingmachine

Author:
Tyler Davis

Definition in file PIDControl.cc.

#include "PIDControl.h"

Include dependency graph for PIDControl.cc:

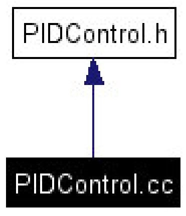


Figure 40: Dependency graph for PIDControl.cc

VMAC/SRC/DRIVER/CLOTHESWASHER/PIDCONTROL.H

Definition of the PIDControl loop class.

PID control law used for motor speed control in washingmachine

Author:
    Tyler Davis

Definition in file PIDControl.h.

This graph shows which files directly or indirectly include this file:

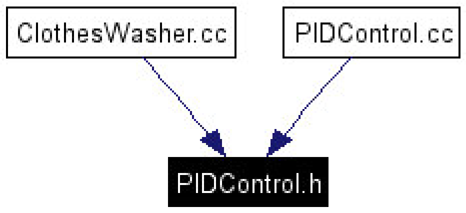


Figure 41: Dependency graph for PIDControl.h

CLASSES

class CPIDControl

Class used for PID control law.

Public Member Functions

CPIDControl (double P, double I, double D, double IntegralMax)

Accessor function.

double GetNextOutput (double DesiredState, double CurrentInput, double Seconds)

Implementation of control law.

Members of the CPIDControl class are explained below.

Public Member Functions

	CPIDControl (double P, double I, double D, double IntegralMax) *Accessor function.*
double	GetNextOutput (double DesiredState, double CurrentInput, double Seconds) *Implementation of control law.*

Table 26: Memory structure of PIDControl.h

vmac/src/driver/ClothesWasher/TrajectoryGenerator.cc

Implementation of the TrajectoryGenerator class.

Used constant acceleration trajectories

Author:
Tyler Davis

Definition in file TrajectoryGenerator.cc.

#include "TrajectoryGenerator.h"

#include <math.h>

Include dependency graph for TrajectoryGenerator.cc:

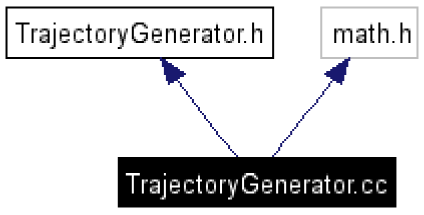


Figure 42: Dependency graph for TrajectoryGenerator.cc

VMAC/SRC/DRIVER/CLOTHESWASHER/TRAJECTORYGENERATOR.H

Definition of the TrajectoryGenerator class.

Used constant accel trajectories

Author:
Tyler Davis

Definition in file TrajectoryGenerator.h.

This graph shows which files directly or indirectly include this file:

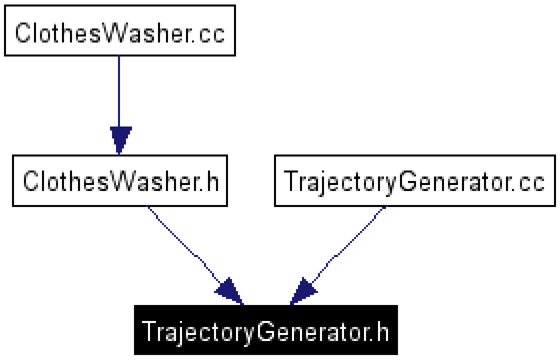


Figure 43: Files that include TrajectoryGenerator.h

CLASSES

class CTrajectoryGenerator

Trajectory generator class. Develops step function in accel, trapezoid in velocity.
Definition at line 10 of file TrajectoryGenerator.h:

Public Member Functions

CTrajectoryGenerator ()

Sets initial conditions.

double GetNextVelocity (double Seconds)

Determines the velocity given a time to change from one to another.

Table 27: Memory structure of TrajectoryGenerator.h

vmac/src/driver/dishwasher/dishwasher.cc

Dishwasher implementation file with control unit.

Author:

Hans Fugal

Definition in file dishwasher.cc.

#include "dishwasher.h"

#include "rabbit.h"

#include <netinet/in.h>

#include <time.h>

Include dependency graph for dishwasher.cc Figure 44:

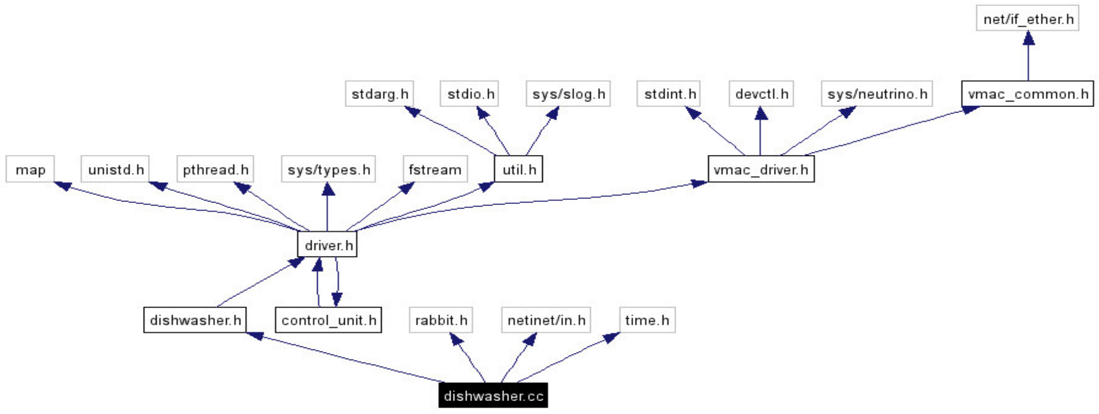


Figure 44: Dependency graph for dishwasher.cc

VMAC/SRC/DRIVER/DISHWASHER/DISHWASHER.H

dishwasher header file that declares the control unit.

Author:

Hans Fugal

Tyler Davis

Definition in file dishwasher.h.

#include "driver.h"

Include dependency graph for dishwasher.h:

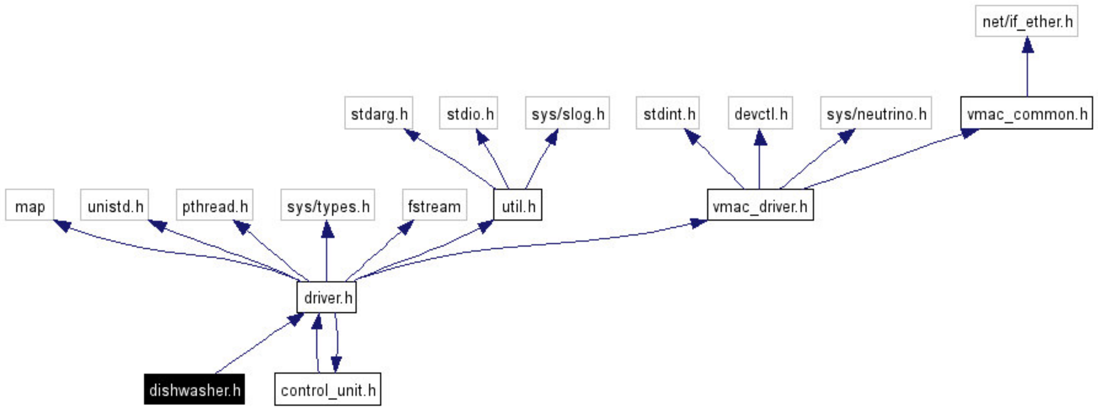


Figure 45: Dependency graph for dishwasher.h

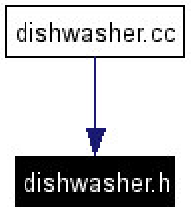


Figure 46: Files that include dishwasher.h

This graph shows which files directly or indirectly include this file:

Classes

class	DishwasherDriver ioboard driver

Definition at line 18 of file dishwasher.h.

Public Member Functions

	DishwasherDriver (int id) *Constructor.*
void	control_thread () *This control loop is called by run().* *It calculates what to send, sends it, gets the reply from the device, then interprets it; over and over until an error is received from the npm, in which case it calls reset() and breaks out of the loop and returns.*

Protected Member Functions

virtual void	ui_recv (set_point *sp) *This "update" comes from the UI.* *In this driver we will simply change the internal state, which will then be reflected the next time the control loop sends to the device.*

Table 28: Memory structure declared in dishwasher.h