2010

A NOVEL MESSAGE ROUTING LAYER FOR THE COMMUNICATION MANAGEMENT OF DISTRIBUTED EMBEDDED SYSTEMS

Darren Jacob Brown
University of Kentucky, darrenjacobbrown@gmail.com

ABSTRACT OF THESIS

A NOVEL MESSAGE ROUTING LAYER FOR THE COMMUNICATION MANAGEMENT OF DISTRIBUTED EMBEDDED SYSTEMS

Fault tolerant and distributed embedded systems are research areas that have the interest of such entities as NASA, the Department of Defense, and various other government agencies, corporations, and universities. Taking a system and designing it to work in the presence of faults is appealing to these entities as it inherently increases the reliability of the deployed system. There are a few different fault tolerant techniques that can be implemented in a system design to handle faults as they occur. One such technique is the reconfiguration of a portion of the system to a redundant resource. This is a difficult task to manage within a distributed embedded system because of the distributed, directly addressed data producer and consumer dependencies that exist in common network infrastructures. It is the goal of this thesis work to develop a novel message routing layer for the communication management of distributed embedded systems that reduces the complexity of this problem. The resulting product of this thesis provides a robust approach to the design, implementation, integration, and deployment of a distributed embedded system.

KEYWORDS: Message Routing, Broadcast Network, Distributed Embedded Systems, Communication Management Mechanism, Fault Tolerance

Darren Jacob Brown

December 3, 2010

A NOVEL MESSAGE ROUTING LAYER FOR THE COMMUNICATION MANAGEMENT OF DISTRIBUTED EMBEDDED SYSTEMS

By

Darren Jacob Brown

James E. Lumpp, Jr., Ph.D.

Director of Thesis

Stephen D. Gedney, Ph.D.

Director of Graduate Studies

December 3, 2010

RULES FOR THE USE OF THESES

Unpublished theses submitted for the Master's degree and deposited in the University of Kentucky Library are as a rule open for inspection, but are to be used only with due regard to the rights of the author. Bibliographical references may be noted, but quotations or summaries of parts may be published only with permission of the author, and with the usual scholarly acknowledgments.

Extensive copying or publication of the thesis in whole or in part also requires the consent of the Dean of the Graduate School of the University of Kentucky.

A library that borrows this thesis for use by its patrons is expected to secure the signature of each user.

Name Date

_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____

Thesis

Darren Jacob Brown

Graduate School

University of Kentucky

2010

A NOVEL MESSAGE ROUTING LAYER FOR THE COMMUNICATION MANAGEMENT OF DISTRIBUTED EMBEDDED SYSTEMS

THESIS

A Thesis submitted in partial fulfillment of the requirements
for the degree of Masters of Science in Electrical Engineering
in the College of Engineering at the University of Kentucky

By

Darren Jacob Brown

Lexington, Kentucky

Director: Dr. James. E. Lumpp, Jr., Associate Professor of
Electrical and Computer Engineering

Lexington, Kentucky

2010

To my loving wife and family …

ACKNOWLEDGEMENTS

The following thesis, while an individual work, benefited from the insight, patience, experiences, and guidance from several people. I would like to thank my Thesis Chair, Dr. James Lumpp, for giving me the opportunity to pursue my masters degree, and for providing me with the tools and opportunity to start a career in the aerospace industry. If it had not been for the experiences provided, lessons learned, and mentorship given by Dr. Lumpp during my time in the IDEA lab, I would not be the engineer and leader that I am. Next, I would like to thank the other members of my thesis committee, Dr. William Smith and Dr. Bruce Walcott. Dr. Smith as both an advisor on the BIG BLUE project and professor impacted my undergraduate and graduate education. Dr. Walcott as a very involved mentor during my undergraduate experience through programs like FESC, B3 tutoring, KNEED, and Tau Beta Pi, indirectly guided me through my EE undergraduate experience. It was those experiences that lead me to continue my Masters education at the University of Kentucky. For all these reasons, I am forever thankful for all the contributions each member made toward my career, education, and thesis.

Beyond the mentorship and guidance mentioned above, the support of my entire family helped me through my masters and my complete collegiate career. However, this thesis would not have been possible if it had not been for my wife, Anne-Walker Brown. The love, patience, sacrifices, and support she has given me during this process is beyond belief. I truly owe the success of this thesis to her.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

1.1 Background

Fault tolerant and distributed embedded systems are research areas that have the interest of such entities as NASA, the Department of Defense, and various other government agencies, corporations, and universities. The reason for this is that embedded systems are everywhere; from the video cards in our PCs to the cell phones in our pockets. Distributed embedded systems also include much larger scale systems like the electronics in our vehicles and control systems in such things as unmanned aerial vehicles (UAVs) and satellites. It is often the case in these expensive, large scale systems that dependability is an important system requirement. Dependability is often achieved in one of two ways. The first method is by hardening the physical circuit components and circuit designs used in the system. This is done by using various expensive hardening materials, as well as shielding techniques to decrease the likelihood of a failure; unfortunately, this method increases cost significantly and still leaves the system with a potential, although not as likely, single point of failure. The second method- which has become a huge area of research itself- is to incorporate fault tolerant mechanisms and design techniques into the system. The basic concept of this method is the realization that faults are sometimes inevitable, and thus the system should be designed in such a way to handle a fault when it occurs. However, traditional fault tolerant methods- like the reconfiguration of a subsystem to a redundant backup or the graceful degradation of a subsystem in the presence of a fault- present a difficult problem for distributed embedded system development. It will be shown in the coming sections that this problem stems from the necessity of a reliable system to have the ability to dynamically manage and reestablish network data producer/consumer dependencies in order to preserve functionality in the presence of faults.

This need for the dynamic management and reestablishment of data dependencies has an affect on each stage of the system's life cycle. From the conceptual design and requirements of the system, to the implementation of each subsystem, to the integration

of all the subsystems, to the deployment and use of the system, and even on to the maintenance of the system- the management and guaranteeing of data delivery between subsystems is crucial for system functionality. This is true especially in the state of a system as it reacts to faults. Thus, it is a goal of this thesis work to present a system solution for the management of data dependencies in a fault-tolerant distributed embedded system. This is accomplished through the design and implementation of a unique Message Routing Layer (MeRL), as well as a unique real-time operating system support infrastructure called IDEAnix.

1.1.1 Fault Tolerant Embedded Systems

Fault tolerance can be described as a system's ability to in some way react in an expected- thereby tolerant- manner in the presence of faults. This reaction can be classified by looking at two basic properties of a distributed system which are the safety and liveness of the system. Safety, as a property of a distributed system, refers to both the well being of the system and its environment, as well as the systems awareness and preservation of state and system data. In other words, a system is considered safe in the literal since if at no time does it cause harm to itself or anything around it. Likewise, a system is classified as safe if when a fault occurs the system does not propagate the fault due to improper state or data. The term liveness refers to the functionality of the system. If the system still runs and is not in a stopped or shutdown state- even if in a reduced or faulty capacity- the system is considered live.

Table 1: The Four Forms of Fault Tolerance **(1)**

	Live	**Not Live**
Safe	Fault Masking	Fail Safe
Not Safe	Non-masking	None

It is the combinational state of system safety and liveness which results in the presence of faults that define the four major forms of fault tolerance. These four forms are system fault masking, non-masking, fail safe, and none(1). The first- system fault masking- is the form of fault tolerance that is one of the most active forms of fault tolerant research (1). This in fact, describes a system that both guarantees system liveness and safety. Meaning that the system in the presence of faults will continue to function nominally without causing any harm to the system. This result of preserving total system liveness and safety of the system is complete fault masking. The second form of fault tolerance is non-masking. This refers to the system preserving liveness but at the cost of system safety. Thus, the fault is not masked and system operation is effected by the presence of faults. An example of this form of fault tolerance includes graceful degradation. This is due to the fact that the system is often directed from a functionally nominal state to a dedicated fault handling mode. In this mode the effects of the faults are not completely masked from the system, resulting in reduced functionality. The third form of fault tolerance is known commonly as fail-safe. This is the form of fault tolerance where the expected behavior of the system in the presence of a fault is for the system to transition to a safe state and then shutdown. In doing this system safety is ensured, but operational liveness is not maintained. Finally, the fourth form of fault tolerance is for the system to fail by ignoring system liveness, and in doing so the safety of the system is not preserved. For this case, the system is not tolerant to a fault, and is thus the null case of the fault tolerant domain. Hence, this form of fault tolerance is not typically a desired state of the system, and should not be consider a viable solution for a fault tolerant system.

1.1.2 Distributed Embedded Systems

Distributed embedded systems, as the name implies, are embedded systems in which the functional load of the system is divided among several processing units. A real world example of this is the control systems in a vehicle such as the ABS breaks, the cruise control system, and the sensor (O2, engine temp, etc.) monitoring. For these systems to work several independent processors can be utilized to collect and process data. These processors are commonly tied together via a bused network such as the

Controller Area Network (CAN) bus which enables them to collaboratively perform the system application of running the vehicle nominally.

Distributed embedded systems are often modeled based upon their application. The application of any distributed system can be modeled as a set of intercommunicating processes that work together (2). Thus, this implies, as Felix Gartner states in the "Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments", "Distributed systems can be modeled as a finite set of processes that communicate by sending messages from a fixed message alphabet through a communication subsystem" (1). This process model of distributed embedded systems is the model that is utilized by this thesis work.

This thesis work aims to define a methodology for the implementation of a distributed system, by making the task of distributing work over the processes of the system smooth and seamless. In doing so, each step- from the concept and design through the integration of all the subsystems- becomes simpler. This is done through the development of a novel design and implementation of a communication subsystem that binds each subsystem of the distributed system together with a predetermined "fixed message alphabet".

By modeling a distributed embedded system using the process model, it is easy to further describe distributed embedded systems as being either synchronous or asynchronous. These two categories of distributed embedded systems are based on timing constraints. If all processes of the distributed embedded system operate within predefined timing boundaries, then the system is said to be synchronous. This means that in synchronous distributed embedded systems all processing and message delivery times must be bounded(1). On the other hand, if the system is allowed to operate without being bound to strict timing constraints then the system is said to be asynchronous(1).

It is important to define these categories of embedded systems because the systems classification (asynchronous or synchronous) sets the requirements for the mechanisms within the system. For example, if an embedded operating system (OS) is utilized in the implementation of the system, by knowing whether the system is

synchronous or asynchronous, a designer can then easily select an OS that has the appropriate mechanisms to meet deadlines if required. Likewise, if known timing constraints must be imposed on message delivery times, then a designer must select a network that supports quantifiable delivery times. It will be shown in the coming chapters that the design of this thesis work is applicable to both classifications of distributed embedded systems.

1.1.3 Embedded Networks

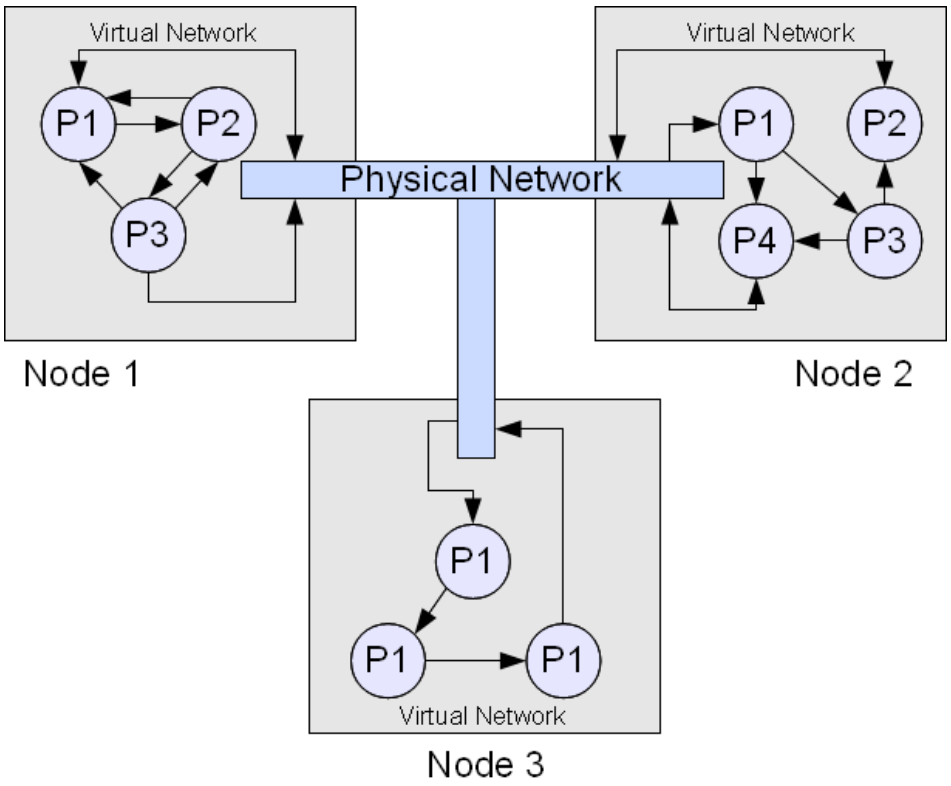


Figure 1: Example Multi-Node Network with both Virtual and Physical Networks

Embedded networks are the heart of every distributed embedded system. Without a network there would be no means of data transfer; thus, creating a division of labor within a distributed system would nearly become an impossible feat. With this in mind,

one can actually define a network as the means by which two or more interconnected nodes communicate(2). If the term node were further defined as a process- as in the process model of distributed embedded systems - then the network can be defined as the means by which interconnected processes communicate. This network definition encompasses the scope of two groupings of networks. These two groupings include physical and virtual networks. A physical network implies the physical connection which provides a data pathway between two or more tangible nodes such as multiple embedded controllers. This physical type of network is commonly identified by the communication protocol (CAN, Ethernet, SPI, I2C, RS485, etc.) it adheres to. A virtual network, on the other hand, is a network which is created inside processors as a means of passing data between internal processes. This type of network is created with the use of operating system level structures such as pipes, queues, and mailboxes.

Table 2: OSI Model- Seven Layer Description(3)

OSI Model			
	Data unit	**Layer**	**Function**
Host layers	Data	7. Application	Network process to application
		6. Presentation	Data representation and encryption
		5. Session	Interhost communication
	Segment	4. Transport	End-to-end connections and reliability
Media layers	Packet	3. Network	Path determination and logical addressing
	Frame	2. Data Link	Physical addressing (MAC & LLC)
	Bit	1. Physical	Media, signal and binary transmission

A common way of describing networks is with the use of the Open Systems Interconnection Reference Model (OSI model). The OSI model was created in the 1970s as, “a layered description for communications and computer network protocol design”(3). As seen in Table 2 there are seven layers to the OSI model. The first layer is the Physical Layer. The Physical Layer defines the electrical details and characteristics of a communication protocol such as voltage levels, wiring standards, etc. The second layer is the Data Link Layer which provides a means of data transfer, and thus acts as the media access control. The third layer is the Network Layer which controls the logic for data

packet delivery and routing. The next layer is the Transport Layer. This fourth layer is responsible for the packetization of application data segments into protocol acceptable datagrams to be passed on to the network layer. The fifth layer is the Session Layer which is responsible for managing (opening, closing, etc.) a transmission session between communicating nodes. The sixth layer is the Presentation Layer. It is in this layer that data is formatted to be used in the application layer; likewise, it is in this layer that data being transmitted from the application layer is formatted for transmission. Finally, the seventh and last layer of the OSI Model is the application layer. The responsibility of this layer is to act as the high level software/firmware application's interface for data transmission.

The product of this thesis work is a novel embedded message routing system. When looking at the OSI model this message routing system was designed to aide with the roles of the network layer through the application layer (layers 3-7). The resulting message routing system- through the development of specialized application program interfaces (APIs)- abstracts the segregation of the mentioned physical and virtual networks, as well. It will be shown in later chapters that with the inclusion of this thesis work into an embedded system design, the network truly becomes a system by which messages can be routed between processes completely independent of the location or address of where those processes may physically reside.

1.1.4 Embedded Operating Systems

Embedded operating systems are specialized operating systems targeted for embedded computing platforms. Like traditional computing operating systems, embedded operating systems have many of the same resources and data management structures like semaphores, queues, and mailboxes. They also often have the common functionality of supporting multitasking; thus, incorporate runtime task management schedulers. The distinction, therefore, comes with the intended platform resources available to each. Traditional operating systems are designed dependent on a lot of hardware resources being available. Take for instance Microsoft Windows XP

Professional that requires a minimum of 128MB of RAM, 1.5 GB of free diskspace, a Super VGA (800x600) resolution or higher video adapter and monitor, CD-Rom drive, and a user keyboard and mouse (4). Compare those requirements to that of an embedded operating system like microC/OS- II. MicroC/OS-II has been successfully ported to several cores including the Intel 8051 core (5), and run on such microcontroller platforms as the Silicon Labs C8051F040 (6)(7). The Silicon Labs C8051F040 has only 4KB of RAM, 64KB of Flash, and a common operating clock frequency of 25MHz(8). As one can see embedded operating systems are targeted to processors and microcontrollers that have limited resources and are integrated into embedded computing systems such as the controller in a dishwasher, the avionics suite in a UAV (6)(7), or even the computing platform that controls the anti-lock braking system in a car(9).

Every embedded system has unique needs and demands that may or may not warrant the use of an embedded operating system. There are two main classifications for the applications that do profit from the inclusion of an embedded operating system. These classifications reflect the dependency and importance of a system meeting deadlines. Thus, those applications that demand the tasks of the system to meet real deadlines utilize the classification of embedded operating systems referred to as real-time operating systems (RTOS). Therefore, all other embedded operating systems which do not put importance on meeting deadlines fall into the second classification known as non-real time operating systems.

The classification of embedded operating systems as real-time encompasses the requirements of a majority of embedded applications. One can conclude this, as embedded systems are often designed around having fast response times due to the common nature of embedded applications. Therefore, the term real-time has two sub-classifications which are hard real-time and soft real-time. A hard real-time operating system has the requirement of providing an absolute guarantee that critical tasks will be completed at or before their deadline, and thus includes schedulers that will satisfy this requirement. On the other hand, a soft real-time operating system- although still placing an emphasis on all tasks meeting their deadline- typically only provides that the most critical task will have the highest active priority. This results in all other tasks being

placed in the background until the most critical task completes(9). Thus, in soft real-time systems it is generally expected for a few deadlines to be missed in acceptable situations(10).

The discussion of real-time versus non-real time and the corresponding elaboration of hard and soft real-time leads to further defining the grouping of embedded operating systems based on their style of task scheduling. Although there have been many published algorithms for the scheduling of tasks within a multitasking operating system(11), there are two groups that all scheduling algorithms can be classified into. These groups are preemptive and cooperative (non-preemptive). In a preemptive scheduling scheme tasks are assigned a priority level. If a task of lower priority is running when a task of higher priority becomes available for scheduling, the lower priority task is halted by the scheduler that sequentially allows the higher priority task to actively run. This is not true of cooperative task scheduling. With a cooperative scheduler, once a task is allowed to run it maintains control until it completes or releases its control.

As can be seen, there are many attributes that affect the selection of an embedded operating systems for various embedded applications. Each attribute has its own advantages and disadvantages; however, there are common advantages for the use of any embedded operating system. From a systems perspective, the use of an embedded operating system is advantageous because it enables modularity. An embedded application can be divided into several modular tasks that can be independently written and then integrated together. The ability to associate tasks into a modular design, and subsequently have a division of labor allows for the system to be efficiently completed in less time.

Adding modularity into any design is generally considered an advantageous design element. However, often times the delicate process of integration of the modular elements becomes quite cumbersome. This is especially true for a modular design of a multitasking embedded application within an embedded operating system. The difficulty comes with the dependencies formed between tasks due to the need of data exchange and the corresponding producer and consumer relationships formed. Therefore, to simplify

the process of integration within an embedded operating system a level of independence is needed between all tasks which are apart of a producer and consumer relationship.

This thesis work aims at developing an operating system level middle layer solution to create this much needed task independence. The creation of this middle layer will result in a systems solution to simplify the integration of modular tasks within an embedded system application.

1.2 Motivation

Over the course of the past several years the University of Kentucky has been involved in the development of unique Unmanned Aerial Systems (UAS). At the heart of the control research and embedded architectural designs for these UAS platforms has been the Intelligent Distributed Embedded Architectures (IDEA) Lab (12). Among the many projects the IDEA lab has had involvement in include cross disciplinary engineering projects as the Baseline Inflatable-wing Glider Balloon Launched Unmanned Experiment (BIG BLUE)(13), as well as unique electronic suites and embedded system designs for the Association for Unmanned Vehicle Systems International (AUVSI) UAV competitions(6)(7). The UAV research sparked by the collaborative work on several UAV designs inspired the IDEA lab's own work on a novel UAV design referred to as the Ready UAV.

In all designs, some collaborative others independent, the IDEA lab has always emphasized the research and development of dependable systems. This emphasis on dependability resulted in the research of achieving dependability through fault tolerant designs. As a result of this the ARDEA Framework was created(14)(15).

It has been a conglomeration of various aspects from each of these projects, and the experience from each that has driven the need for this thesis work.

1.2.1 BIG BLUE

The University of Kentucky's BIG BLUE project was a NASA funded endeavor to research UAV technologies for Mars exploration. This project consisted of several phases that encompassed many years of collaborative efforts between mechanical and electrical teams. Through out each phase of the BIG BLUE mission, high altitude balloons were utilized to take experimental payloads to an altitude where the atmospheric conditions were similar to that of Mars. The first phase, BIG BLUE I consisted primarily of defining and proofing the experimental concept of inflatable wing technology. It was this phase of BIG BLUE that the first high altitude balloon launch was conducted. The payload for this mission was the first set of inflatable, UV curable wings, which when exposed to UV light became permanently rigid structures. The second phase, BIG BLUE II, focused more on the avionics backbone and the development of an instrumentation suite needed to control flight, which was the basis of a custom autopilot design. The third phase, BIG BLUE III encompassed two different missions. The first was a flight mission based around verifying a new mechanical design for the wing technology of a Mars Exploratory UAV. This new wing design- unlike the first two missions which were inflatable, permanently rigidizible wings cured by UV exposure- while still inflatable achieved its rigid structure by maintaining a certain internal wing-bladder pressure. The avionics portion of this mission was focused on the mission controller, execution sequencer, instrumentation, telemetry processing, and data transmitting. The instrumentation was designed around capturing mission critical data to monitor every aspect of the wings including digital imagery of the inflation process at altitude. The second mission of BIG BLUE III was focused on refining the autonomous control of a UAV design fashioned around the design of the inflatable wings used in the high altitude phase of this mission. To accomplish this goal a novel UAV design was completed to compete in the AUVSI PAX RIVER competition(6). The mission of this competition consisted of building a fully functional UAV capable of autonomous flight, GPS waypoint navigation, target reconnaissance, as well as the ability to adjust to changing mission requirements. Thus, the electrical designs for this mission included an avionics suite focused on the mission requirements of autonomous flight, GPS navigation, and

ground target recognition. Finally, the last two phases of the BIG BLUE project (BIG BLUE IV and V) were focused on designing a smaller lighter structure to support the inflatable wings, as well as, utilizing a commercial autopilot not only as the principle flight controller, but as the mission controller and sequencer.

Every stage of the BIG BLUE mission was a success unto itself. Many great things were accomplished and learned in each phase. However, it was in the BIG BLUE III phase of development that the need for this thesis work became apparent. In BIG BLUE III the firmware design utilized for the first time an embedded real time operating system. The operating system concept of tasks allowed for the clear division of subsystem development and their integration onto a single microprocessor. However, traditional methods of data delivery were used between tasks which made both the addition of new data fields during development and the integration of all the individual subsystems quite a cumbersome task. It was during this firmware driven subsystem integration that the need to streamline the integration process by abstracting direct communication links between individual tasks became very clear. This concept of communication link abstraction is one of the guiding principles of this thesis work and will be discussed in detail in later chapters.

1.2.2 AUVSI UAV Student Competitions

The organization AUVSI has developed a series of competitions to inspire the minds of young engineers in the area of unmanned system development and control(16). To that end, the University of Kentucky has taken part in the 3rd, 4th, and 5th annual AUVSI UAV Student Competitions(6)(7)(17). The overall mission requirements for these competitions included developing a UAV platform that was capable of autonomous flight, GPS navigation, and ground target recognition.

The first University of Kentucky entry (3rd Annual AUVSI Student UAV Competition) to this competition was the Airborne Intelligent Research Craft for Autonomous Technology (AIRCAT) UAV (6). The AIRCAT design and

implementation was apart of the second phase of the BIG BLUE III project. The AIRCAT (Figure 2) was a completely custom UAV design. The mechanical structure was a fiberglass design that was based around the profile of the inflatable wings used in the high altitude testing phase of BIG BLUE III. The avionics suite for the AIRCAT, as seen in Figure 3, included a mission controller which controlled all integrated interfaces. These interfaces included telemetry gathering, commanding of still camera imaging, data overlay of present GPS location on the live video feed, sensor telemetry gathering, etc. Other avionics included the MicroPilot MP2028 autopilot for autonomous control and waypoint navigation, infrared and visible light digital cameras for high resolution image captures of the ground target area, Amateur Television (ATV) transmitter for live video streaming of the ground target area, various servo motors for control surface actuation, and a battery pack centric power supply design to power all electrical systems.

Figure 2: The University of Kentucky AIRCAT UAV

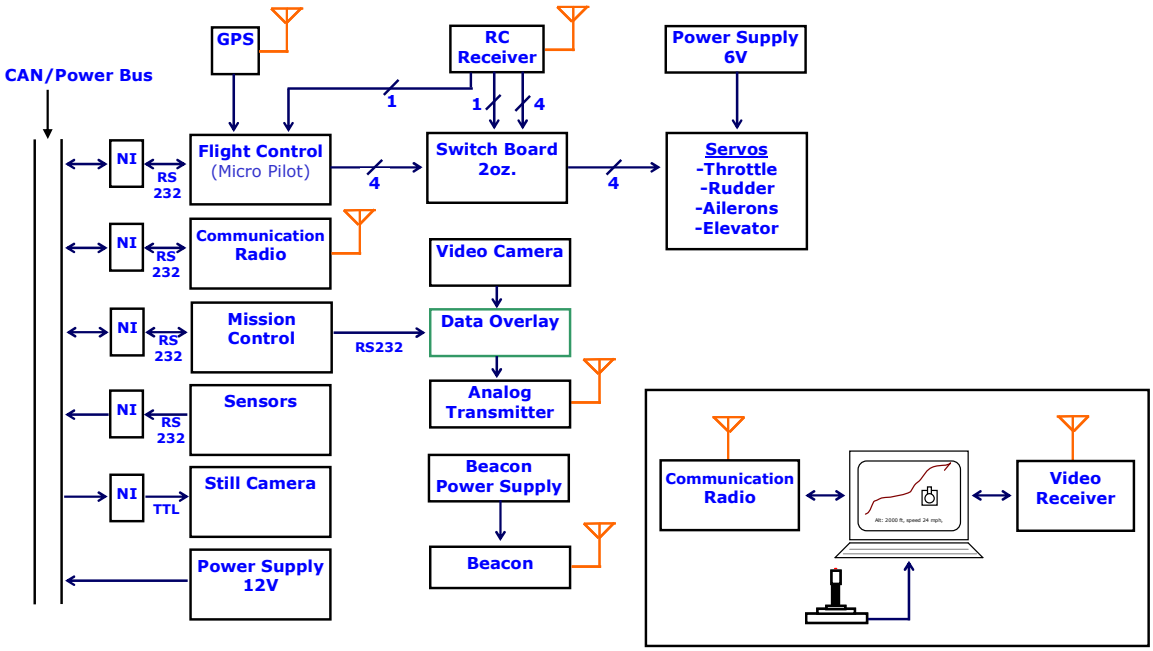


Figure 3: Avionics Suite for the AIRCAT

In a continuing effort to develop UAV technologies, the Aerial Robotics Team at the University of Kentucky developed the Southern Komfort UAV which was entered in the 4th Annual AUVSI STUDENT UAV Competition (7). The Southern Komfort, as seen in Figure 4, utilized a standard R/C model airframe with a custom avionics suite (Figure 5) that integrated the COTS Piccolo Autopilot to handle autonomous flight stability and GPS waypoint navigation. This UAS platform also included in its ground station support a state-of-the-art imaging software that would process a series of still images based upon their centered GPS location. The resulting output from the image processing was a composite aerial view and accurate mapping of the surveyed area.

Finally, the last entry into the 5th Annual AUVSI UAV Competition by the University of Kentucky was the Southern Komfort UAV with additional payloads and advanced sensor selections. This entry built upon the advances made in the previous years design for the camera stabilization and live video feeds. Also, an additional GPS sensor (Lassen iQ) was selected to complement the GPS data from the Piccolo autopilot. Due to its much faster sampling frequency, the instantaneous GPS data was far more accurate in assisting to locate mission critical targets captured in the live video feeds.

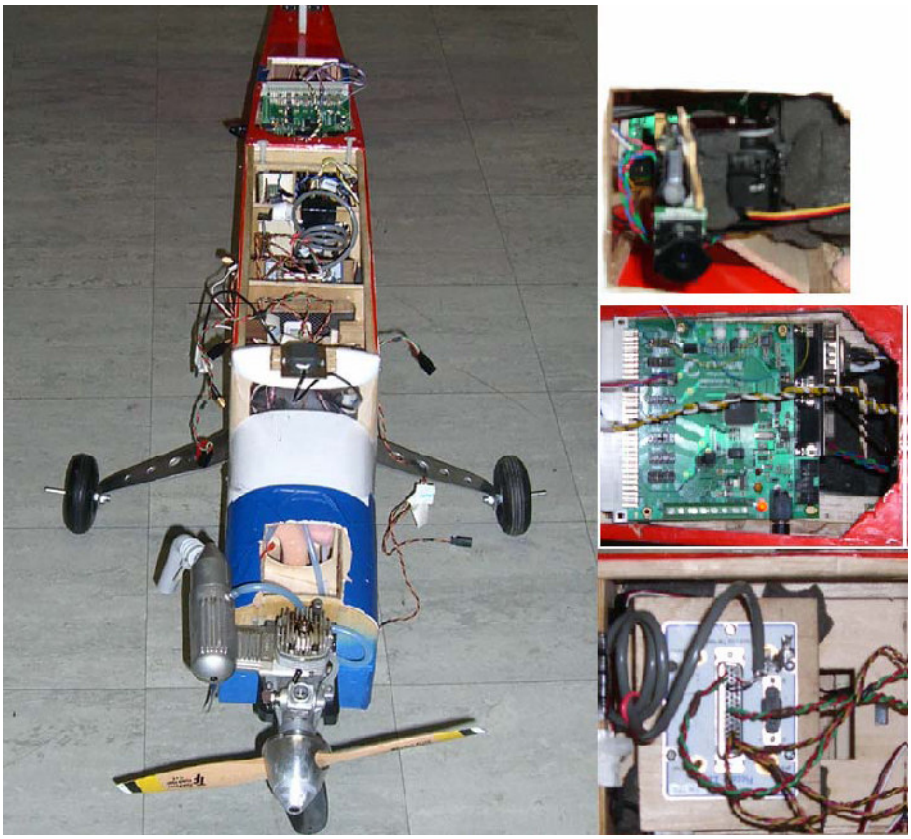


Figure 4: The University of Kentucky's Southern Komfort UAV

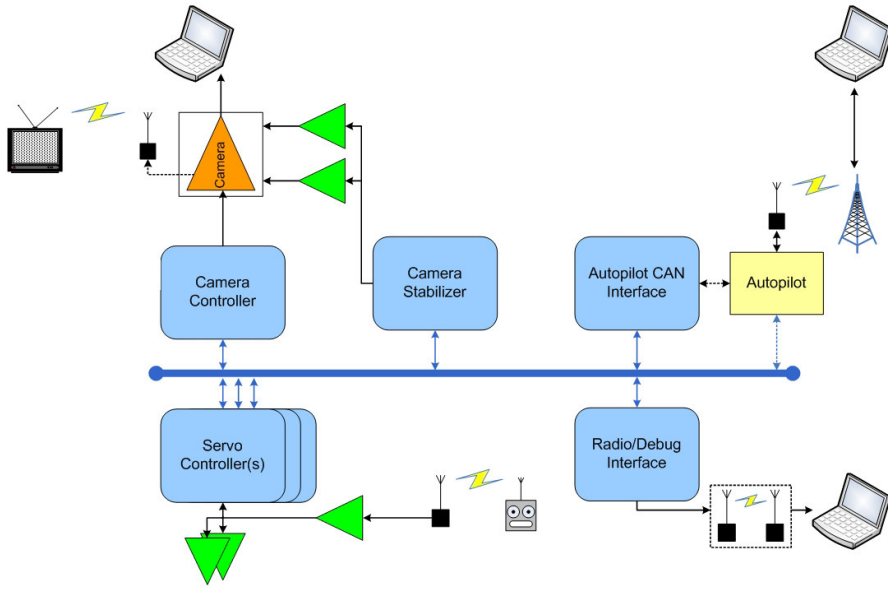


Figure 5: Avionics System Design for the Southern Komfort UAV

Each of these entries into the AUVSI competition made important contributions to the UAV technology development work at the University of Kentucky. These projects had common functionality and guiding design principles that through their lessons learned, helped to further define the need and mandate requirements for this thesis work. An example of this is that of the independent subsystem development. Each UAV design had a set of subsystems that either produced data to be used by other subsystems, or themselves were consumers of data from other subsystems. This created an infrastructure of producer and consumer dependencies that was integrated by not just a physical network (ex: connection between autopilot and mission controller processor), but by a virtual network of independent subsystem controllers, as well. These subsystem controllers were implemented as inter-communicating tasks integrated onto a single processor with a real-time operating system (ex: autopilot telemetry monitoring task and video overlay command task). These independent subsystem designs being network centric with cross system producer/consumer dependencies, showed the need for a data handling mechanism that would assist in both the development and integration process. It was apparent that development time spent on establishing and managing direct communication links between producers and consumers of many different data points was into itself a very cumbersome task. This became even clearer as the need to establish new data dependencies would arise during development. It was also apparent that the complexity of integrating several independently developed tasks could have been simplified. This simplification could have been done had the role of producer and consumer of data been abstracted such that there no longer existed a need to establish direct communication links.

The development of these UAV projects demonstrated the need for several factors to exist in an embedded system design. If these factors are present it would not only make development easier, but make the integration process much simpler and more reliable. The first key factor is the need for a data handling mechanism to abstract direct data communication links between data producers and consumers from independent subsystem developers. The second factor is the need for this data handler to not only handle data over physical networks such as CAN, but also handle networks where the communication of data existed over a virtual network between tasks both on the same

processor and on different processors. Finally, the last key factor these UAV projects brought to life is the need for a data handling mechanism to be designed in such a way to be a systems engineering tool. This mechanism should by its very nature force the definition of all data to be generated by the system to be defined in the first stage of design. In doing so, the mechanism should make the identification of all data fields readily available to all subsystems. Also, if during the implementation of the system a new data field should be introduced- the mechanism should be easily adaptable to make an addition without causing any intrusion into the development of a subsystem. By doing this the integration process should then result in an easy conglomeration of all subsystems without any major additions needed.

1.2.3 ARDEA Framework and Run-Time Environment

The automatically reconfigurable distributed embedded architectures (ARDEA) framework facilitates the fault tolerant design and implementation of real-time distributed embedded systems (18). To accomplish this, the ARDEA framework utilizes a novel graphical software specification system that uses dependency graphs to model and map an embedded system's resources(14). As a result, this mapping specifies the interaction of software modules within the system. In Figure 6 there is an example ARDEA dependency graph which shows a control system configuration with multiple sensor inputs that flow into a software module. The software module interprets its input and correspondingly drives the system outputs.

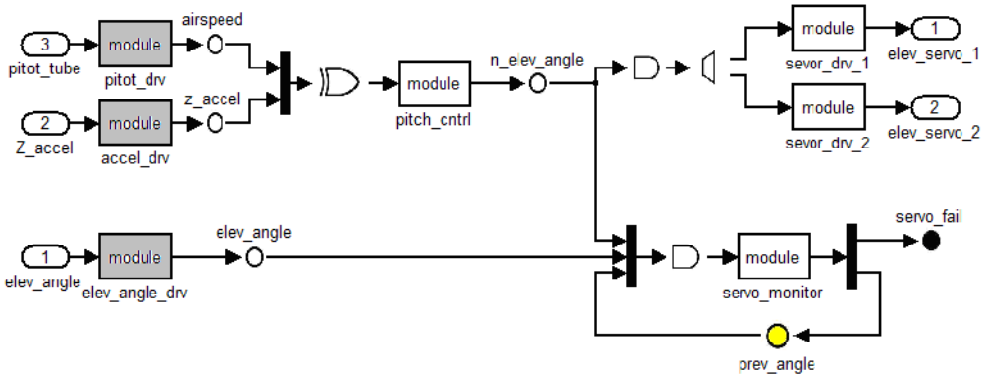


Figure 6: Example ARDEA Dependency Graph (18)

As a distributed embedded system framework, ARDEA consists of a few key components. These major components as seen in Figure 7 include a system manager, processing elements, sensors and actuators, as well as a communication subsystem that allows for the passing of data between each subsystem(14). At the bottom of Figure 7 are the sensors and actuators which are the inputs and outputs that connect the system processing elements to the real world. Next are the processing elements which are the computing platforms where the software modules reside within the ARDEA framework. Observing the status of these processing elements and the availability of other system resources is the ARDEA system manager. The system manager's main role is to monitor the system for faults and failures. If a failure were to occur the system manager uses its knowledge of the system configuration via predefined dependency graphs to reallocate the unaffected and available resources within the system. This system reconfiguration requires the processing elements to have the ability to accept the scheduling and rescheduling of affected software modules. Thus, to enable such a reconfiguration requires both the presences of an embedded operating system running on every processing element, as well as a message routing system. The message routing system is needed to ensure that data is correctly sent from its producer to the appropriate consumers. It is important to note that the location of the producers and consumers can change dynamically in the presences of faults. Thus, it is requirement that the message routing layer of the system must handle the passing of data between producers and consumers independent of network location.

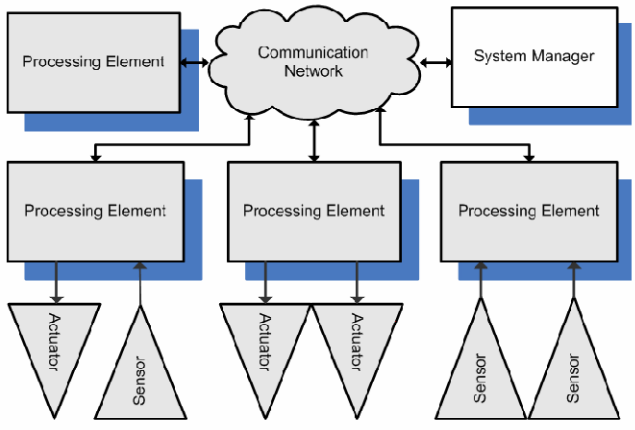


Figure 7: Components of an ARDEA Supported Distributed Embedded System (14)

It will be shown in the coming chapters that the product of this thesis work is a system which will fulfill the message routing requirements mandated by both ARDEA and similar fault tolerant architectures.

1.2.4 Ready UAV

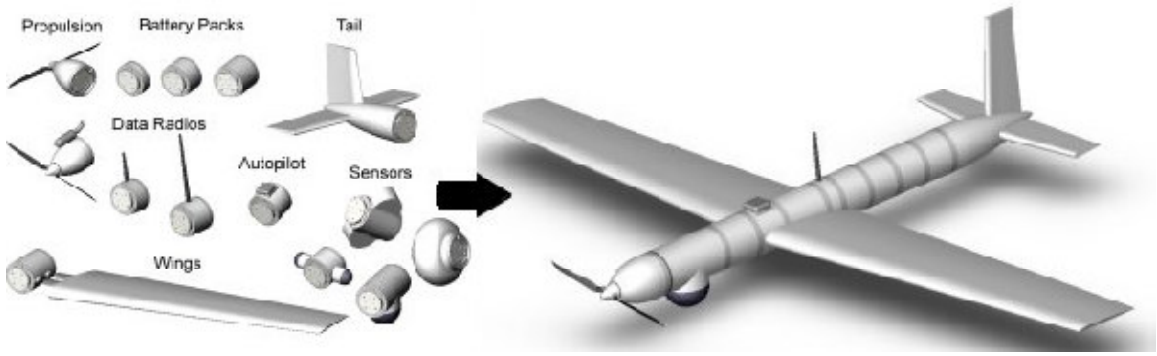


Figure 8: The Ready UAV Concept and Example Break-down of Modules

The Ready UAV is a novel modular UAV design conceived by the IDEA lab. Its uniqueness comes from its adaptability to be configured in different ways in order to directly support various mission critical tasks. Examples of such mission tasks include aerial surveillance and/or environmental testing. The basic concept for this UAV design as seen in Figure 8, stems from the idea that the plane is made up of several mechanically independent modules that can be joined together to form a light airframe, which is easily deployable. From a systems perspective, each module is its own smart device, meaning that each module is either a sensor (ex: carbon dioxide, radiation dosimeter, etc.) or actuator (motor, servo controller, high resolution camera unit, etc.) with its own processing interface. All modules interface together not just mechanically, but over a fault tolerant network like CAN, that supports reconfiguration with the aid of an embedded application interface, such as that of this thesis work. Much work has been done on an embedded architecture to support the dynamic flight reconfiguration known as ARDEA. ARDEA with its system manager and method of heartbeats becomes the

enabling force behind the ability to have various flight configurations without causing any system complications. With both the ARDEA architecture and a message routing mechanism- a product of this thesis- built on top of a reliable broadcast network, the embedded system design for the Ready UAV is complete.

1.3 Problem Statement

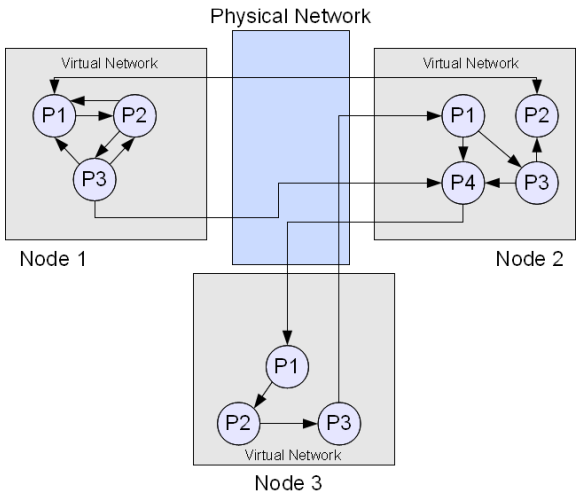


Figure 9: Example Data Dependencies over both Virtual and Physical Networks

The implementation of large scale complex distributed embedded systems is often broken into several smaller subsystems. This allows specialized teams of engineers to focus on and perfect the design of each portion of the much larger system. Once all subsystems are completed, they are integrated together to build a single, common functioning system. It is this final task of integration over a network that makes the design and implementation stages of each subsystem complex. This complexity is due to the fact that each subsystem often either generates data to be used, or itself relies upon the data generated by other subsystems. These data producer and consumer dependencies present a very important system level design constraint of creating and managing data pathways among all the subsystems to guarantee data delivery. This creates an interesting problem for large-scale distributed embedded systems such as unmanned

aerial vehicles (UAVs), airplanes, satellites, etc. All of these systems have the additional requirement of being reliable. These systems often use redundant and reconfigurable architectural designs to achieve their reliability. In such design architectures when a hardware or software fault occurs, functionality is often shifted to a redundant backup which then assumes the primary role as the new producer or consumer of data. In this scenario, all communication dependencies must be reestablished dynamically. An example of network dependencies is visualized in Figure 9, with subsystem processes being integrated both by physical network connections (ex. CAN or Ethernet), as well as by virtual networks based on operating system level mechanisms as mailboxes, queues, or pipes. It can be deduced from the network concept visualized in Figure 9, that reestablishing traditional direct point to point communication pathways can be very difficult, especially in a dynamic reconfiguration mode. Thus, a need exists to create an extensible, system-level mechanism to simplify the management of data dependencies between producers and consumers both on a physical network, as well as a virtual network.

In this section, an introductory definition and requirements for such a methodology and resulting mechanism are discussed.

1.3.1 Requirements

As with all designs, there exist many requirements both on the system and implementation level that must be imposed on the design of a unique methodology. From the systems perspective, one can reference the definition of systems engineering. As stated in NASA's Systems Engineering Handbook(19), "systems engineering is a robust approach to the design, creation, and operation of systems." This definition encompasses the goal of the data management mechanism created as a product of this thesis work. It is the goal- by the very nature of its design- for this data management mechanism to be a robust approach to reduce the design complexity, ease the implementation, simplify the integration, and make the system operation more reliable for all distributed embedded systems that incorporate it.

Given this definition and direction, there are many system level requirements that must be fulfilled. First among many is the requirement to make the integration process simple, quick, and easy. As a means to simplify the integration process and aid in the design and implementation phases of the system development, direct data dependencies must be avoided by abstracting them into their roles as producers and consumers. No longer should a dependent consumer subsystem have a preset knowledge of the locale (network address, subnet, pipe reference, mailbox structure, etc.) of the producer of its data. With this abstraction as a requirement, comes the need to have all data easily identifiable by all systems on the same network. This easy identification will be further defined and shown in the coming chapters to assist in the fulfillment of making the system reliable through its support of reconfiguration. This requirement of reliability support mandates that the process of establishing and reestablishing data dependencies be simple and itself a reliable process. Finally, the last system requirement that must be imposed is that if change were to occur during the implementation stage, the system should be able to unobtrusively add or remove data relations without the mandate of substantial design change.

Imposing these system level requirements, presents the need to impose concrete requirements on the implementation of the data management mechanism, as well. Being that the data management mechanism is in reality a communications layer that manages the abstraction of data dependencies; a communication infrastructure must be created. Thus, the first major requirement is the existence of a network. As mentioned previously, a network can be defined as a mechanism by which data is passed from a producer of data to a consumer of data. This definition allows the term network to refer to both the internal routing of data on a single processor by means of operating system level mechanisms (pipes, mailboxes, queues, etc.), and the physical embodiment of wires and associated protocols (CAN, Ethernet, etc.). With internal networks on a single processor commonly relying upon operating system level data structures, a requirement must be imposed that an embedded operating system must exist. Finally, the data management and routing mechanism must be extensible and scalable from large scale distributed networks with multiple processors down to a small internal network on a single processor.

With all of these requirements imposed, a successful communications layer to support data management and routing will be created.

1.4 Thesis Overview

This thesis work, as referenced throughout this introduction chapter, was completed with the intention of creating a system methodology for the communication management of data dependent tasks within distributed embedded systems. The requirement for this methodology to work for safety critical applications drove the need for the support of reliable design, and the application of this thesis work to meet the communication needs of fault tolerant embedded architectures. Thus, the flow of this written thesis leads into the description of embedded system faults, and the various aspects associated with fault tolerant embedded systems in Chapter 2. Next in Chapter 3, topics of communication networks as they relate to this thesis work are described. These network topics include broadcast and multicast messaging, node versus message oriented protocols, and the selection of a network protocol for fault tolerant applications. Following these topic chapters is chapter 4, "The Design of a Message Routing Layer". This chapter covers the design of the main product of this thesis, a message routing layer that abstracts direct data dependencies of inter-task communication over both a physical and virtual network. Next, in Chapter 5 "IDEAnix" the operating system selection and support infrastructure needed for the implementation and integration of this thesis's unique message routing layer is described. Then in Chapter 6, the concepts of this thesis work are related and shown to be beneficial- from a systems perspective- for the managing of an embedded system through its project lifecycle. Next, in Chapter 7 details are given about a test implementation of both the operating system infrastructure and the messaging routing layer of this thesis, targeted for a simple UAV architecture. Finally, Chapter 8 is the conclusion which summarizes this thesis work.

2 FAULT TOLERANT EMBEDDED SYSTEMS

Fault tolerance can be described as a system's ability to continue to be operational in the presence of system failures. It is the nature and intention of this thesis work to be applicable to systems that are designed to achieve such fault tolerance. As such, this chapter serves to explain the various aspects of fault tolerant systems. Throughout this chapter fault tolerance is further defined by describing its individual elements. Thus, the first section entails descriptions of common faults found in distributed embedded systems. This leads into the following sections that detail fault detection methods and fault handling techniques. Finally, the last section focuses on understanding the characteristics of a fault tolerant embedded system and discusses two models of fault tolerant systems based on timing constraints.

2.1 Embedded System Faults

To understand and implicitly define a fault beyond the simple definition of being the causing agent of an error, one must look at the scope of which a fault may occur. For this intended purpose, a fault is restricted to being the causing agent of errors within a distributed embedded system. As it is known, distributed embedded systems consist of many parts which often include several sensors and actuators attached to several processing elements such as FPGAs, microprocessors, DSPs, and various other computing platforms. These processing elements are interconnected and work together to perform an overseeing function governed by the independent algorithms implemented in software modules written for each processing element. It can be logically deduced that each portion of an embedded system- input and output devices, processing elements, software modules- are all prone to various types of faults. Therefore, embedded system faults can be classified into four major categories: input device faults, output device faults, processing element faults, and software module faults(14).

Though faults may fall into classifications based on their origins, faults have no affect on the system unused. For example, radiation may cause a permanent memory cell

latch up, if this memory cell is never accessed then the fault results in no system error or failure. Thus, the fault goes unnoticed and causes no harm to the system. For this reason, it then can be argued that this defect in memory although a fault was not a system fault. It is for this reason that faults in process modeled embedded systems are formally defined based on system state change(1). This is based on the fact that systems can be observed as having two main classes of transition events. These transition events are normal system operation, and fault occurrence. In the absence of faults the system works normally transitioning from one operational state to the next; however, in the presence of faults the system will transition into an undesired error state. Therefore, faults in embedded systems can be formally defined as the undesired system triggers that result in a transition into an error state. It is in the unwanted but often accounted for error state that faults can be detected and potentially corrected.

2.2 Fault Handling

To claim that a system is fault tolerant, the system in fact has to be able to handle a fault as it occurs. This fault handling process consists of the two phases of fault tolerance which are fault detection and correction(1)(20). This section elaborates on these two key phases.

2.2.1 Fault Detection

The first phase in achieving a fault tolerant environment is detecting faults that may occur. This is often a cumbersome task due to the fact that faults are unexpected events, and difficult to plan for. It is for this reason that when a requirement of fault tolerance is imposed on a system, the system should first be analyzed for likely fault occurrences often referred to as fault classes(21). For example, if the system being designed is an attitude control sensor board for a satellite to be placed in a low earth orbit (LEO), one possible fault scenario is that of radiation effects (single event upsets (SEU), single event latchups (SEL), etc.). If radiation effect faults are to be tolerated by the

system then the system designer would then have a narrowed classification of faults for the system to target for detection and potentially correction.

Once all fault classes are determined for the fault tolerant system, the system designer then has to determine the appropriate technique to detect the faults as they occur. There are numerous fault detection techniques that have been defined over the many years that fault tolerant systems have been designed. It is often the case that the implementations of these techniques are focused on the specific fault occurrences they aim at detecting. However, many of these techniques can be categorized into technique classes. Four common detection technique classes are n-version redundancy and voting, state estimation and monitoring, system feedback monitoring, as well as software wrapping and monitoring(14).

N-version redundancy and voting is a technique that uses n identical mechanisms (n-sensors, n-software modules, etc.) that are expected to result in the same output. The outputs are compared and then the majority result is assumed correct, with the minority outliers assumed to have resulted by some fault. This majority rule has long been accepted as an accurate means of fault detection. An example of n-version redundancy and voting is the well known Byzantine Generals' problem (22).

State estimation and monitoring is a technique where the state of a system (variable values, scheduled task selection, active processors, etc.) can be predicted and compared within some bounds to the actual current state. For further clarification, consider the real world example of a UAV GPS speed output (14). If the UAV has been instrumented with an airspeed sensor and an IMU, an estimate of the actual speed of the UAV can be calculated to within some tolerance. This value can be compared against the value given from the GPS sensor which may be the primary input to a speed control algorithm. If the monitored value from the GPS sensor is not within the tolerance of the speed estimate then the GPS speed fault was detected.

System feedback monitoring is a fault detection technique that relies upon the monitoring of a response mechanism that is given for some input, request, or action. There are several applications for this type of fault detection for use in both network

interfaces, as well as actuator interfaces. In fact, some communication protocols have this feedback monitoring technique built in. Take for consideration RS232's parity bit. A parity check of the transmission bits is preformed before transmission, and upon receipt the user can monitor the parity bit to ensure all bits were transmitted properly. The TCP protocol also has it built in where data is guaranteed delivered with a system of ACK checks and retransmissions requests. Other examples include actuators such as motors or pressure transducers that are paired with an analog voltage respectively indicating position or pressure. Thus, if the system commands a certain position or pressure to be set and the monitored feedback voltage does not indicate the proper adjusted value, then obviously a fault occurred. It is through the monitoring of feedback that these types of hardware and network faults can be detected.

Software wrapping and monitoring this technique is similar to state estimation and monitoring, but refers explicitly to the monitoring of the input and output of software algorithms that are compared against predefined rules (14). If the software modules deviate from their expected behavior (e.g. output is out of expected range) then a fault is known to have occurred.

2.2.2 Fault Correction

As outlined in the "Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments" published in the ACM Computing Surveys(1), system safety can be preserved through fault detection, but without fault correction system liveness cannot. That is to say that if a fault were to be detected, it could be handled in such a way as to trigger or transmit an alert stating it has an error, or even potentially acting on its own to shut the system down in a safe manner. This handling based on detection does not make any claim to sustain functionality. Thus, as the term correction implies, if it is the goal of the system to function in the presence of faults- potentially even at a reduced capacity- the system must have the ability to handle faults in such away as to keep the system alive and functioning in some nominal state. This leads to the fact that appropriate resources must be available in the system when a fault occurs to support

the affected functionality. Therefore, it can be said that redundancy is necessary for a system to guarantee both safety and liveness, and thus be truly fault tolerant (1) (23).

There are many forms and uses of redundancy that enable a system to be fault tolerant. The selection of the appropriate technique is often governed by many constraints including system safety, environment safety, cost restrictions, and functional requirements. A few of these fault tolerant redundancy techniques include N-version redundancy with voting, redundant estimation, and redundant resource reallocation.

N-version redundancy and voting is a redundancy scheme that is both a detection and correction mechanism. Faults are detected as they occur by the system utilizing n identical mechanisms which are expected to yield similar results. If a dissimilar result is yielded then the majority result is assumed correct. Thus the system then knows which of the n redundant mechanisms faulted, while simultaneously preserving system functionality/correctness by utilizing the correct result from the majority vote. This methodology has been proven to handle transient faults through the Byzantine fault model(22). A common implementation of n-version redundancy and voting is where n is set to three, and the voting result is based on a "best 2 out of 3" ideology. This common implementation is known as Triple Modular Redundancy (TMR).

The redundant estimation technique refers to the use of system estimators as a redundant backup to potentially faulty inputs or functional results of a system. This can be integrated into the system where estimations can be performed. Thus, as an input or result of some kind is given, the system can use the estimation to check its validity. If the result or input is found to be faulty (out of anticipated range) and is still needed for the system to operate properly, then the system may then elect to use the estimation. Using this technique will result in the fault being masked and the system liveness being preserved.

Redundant resource reallocation is a form of fault tolerant correction, where in the case of a fault the system may reallocate its unused resources to ensure the system still operates in a correct, live manner. There are two major forms of resource reallocation. The first is a dedicated physical one to one resource redundancy. This is where there

exists an exact unused duplicate of a resource that is ready to be activated in the case of a fault. A common example of this is redundant hard drives setup in a raid configuration. One hard drive is actively being read from and written to, while a second hard drive is present as an inactive duplicate. If the primary drive fails, then the secondary drive is available to be used without any loss of data. The second form of redundant resource reallocation is where resources exist in the system but may be shared by other subsystems. A prime example of this form of resource reallocation is that of flash memory. If a single page of flash on a single flash chip is found to be bad (latch ups, burn outs, etc…) and is no longer writable, the flash chip may be large enough to have unused flash pages that could then be used. In this example, there were not multiple redundant flash chips; however, there were redundant flash pages within the same flash chip. Thus, the resource was redundant, but not dedicated.

2.3 Fault Tolerant System Models

Fault tolerant systems, having been a wide area of research for decades, have been modeled in hundreds of different ways for hundreds of different systems. However, to create a model for a fault tolerant environment; one must know the system for which fault tolerance is desired. This thesis work targets distributed embedded systems. As mentioned in the introduction, distributed embedded systems can be defined and modeled as a collaborative set of processes which are enabled to communicate in an independent manner. There are two system models that are often used to describe these types of process oriented distributed embedded systems. These models are the asynchronous and synchronous system models. An asynchronous system is one in which processors and processes communicate by sending messages to one another with some arbitrary delay. Likewise, in asynchronous systems functional response times are also assumed arbitrary. On the other hand, in synchronous systems time is of the essences. The definition of synchronous in relation to computing, as stated on Webster's Online Dictionary, is that of "occurring at the same time or at the same rate or with a regular or predictable time relationship or sequence" (24). Thus, within synchronous systems, known time

constraints are imposed on inter-process/ inter-processor communications, as well as system response times(1).

In fault tolerant systems, it is difficult to achieve a completely synchronous system. This is due to the arbitrary timing introduced into the system as faults occur. Different faults require different detection and correction methodologies that justifiably utilize varying amounts of computational time based upon the type and severity of the system fault or faults. However, if no assumptions about timing are assumed including process run times of a system- as is the case in an asynchronous system- it has been shown impossible to detect process state transitional failures(25). Without the ability to detect a process failure it is impossible for fault tolerance to be achieved(1). So then it logically follows that fault tolerant system models, despite the arbitrary timing issues involved in detection and correction, must make some synchronous-like assumptions. These assumptions include things such as the existence of upper-bounds on process runtimes, and timing constraints on delays between message deliveries (1). If these assumptions of bounded time are assumed to exist, but the bounds themselves remain arbitrary and thereby unknown; then a uniform asynchronous model is still valid for fault tolerant embedded systems(26).

It is also possible to achieve a uniform synchronous model for distributed embedded systems, despite the delays associated with fault detection and correction. As stated previously, for a distributed system to be fault tolerant it must have the ability to detect such things as process failures. To detect process failures, there must be bounds imposed on process run times and periodic message delivery delays. Likewise, it is logically inferred that when the system state changes to a known fault mode, if all times for recovery are known and all running events occur within a schedulable time frame, then the system can be said to be running in a valid synchronous mode. Although the response time of the faulted action may no longer be held to its real-time value, the active state of the system adheres to all imposed time constraints. Therefore, if the process run-times, periodic message delays, and the times associated with the detection and correction of faults can be quantified to a known time constraint, then the system can be said to be uniformly synchronous. Thus, such a valid synchronous system model exists.

It is important to note that embedded systems regardless of fault tolerant capabilities are often modeled based upon process communication and system response timing (asynchronous and synchronous). Thus, for systems that require fault tolerance in a definable time constrained manner the network and message passing methodology become central in determining whether the system can be modeled as synchronous or asynchronous. Therefore, the selection of a network and message passing mechanism becomes the enabler for how fault tolerance can be achieved. This is a central idea and one of the governing reasons for this thesis work.

3 THE NETWORK

The network is an integral part of a fault tolerant distributed embedded system. As seen in chapter 2, for a system to support fault tolerance there must be redundancy (27) (28) (2)(1). Without a network the activation, support, and general management of a redundant feature such as a redundant processing unit would be nearly impossible. In distributed fault tolerant architectures like ARDEA, the network is also vital due to the architectural support for software module or process reallocation. This is where, in the presence of a faulty processing node, a key software module may be moved or instantiated on a different processing element (18) (15)(14). This proves to be difficult to implement with traditional network support alone. With a traditional network, if a process were in a producer/consumer relationship with another process, and were forced to be instantiated on another node several things would have to be reestablished. For example, the communication pathways or network connections would have to be reestablished with such things as its new network address, as well as the potential establishment of dedicated connected ports or channels on the new node. It is this tracking, mapping, and general dependencies needed in producer and consumer relationships that make the design and implementation of a fault tolerant distributed embedded system a difficult task (2). Thus, for a network infrastructure to support reconfiguration and thereby support fault tolerant design- it must allow for the routing of data properly from a producer to its corresponding consumers regardless of their physical location within the network.

It will be shown in this chapter that the support of a fault tolerant infrastructure by a network can be accomplished through the selection and novel design of various attribute levels of the OSI model. This thesis takes advantage of the fact that the lower levels of the OSI model are easily integrated into the design of a reconfigurable network infrastructure through the proper selection of existing network protocols. Important selection criteria such as the need for multicast/broadcast support and message oriented data transmissions will be discussed in the following sections.

3.1 Broadcast/Multicast Networks

Broadcast/ multitask networks are data delivery methods that ensure data may be sent to multiple receivers. Specifically, a broadcast network message is sent from a single node to all nodes on the network; where as, a multicast network message is sent from one node to a selective group of nodes within the network. Conceptually, they both can be thought of as a one to many messaging system, where broadcast explicitly defines many as all. The use of these messaging schemas has been stated as inherently supporting consistent message delivery(28). This can be logically reasoned as true by observing the messaging scenario of a physical bus broadcast protocol such as CAN. In such a messaging environment, a message that is placed on the bus is observed at the physical layer by all nodes. Electrically, a message on a physical bus is the same over the entire system. Hence, messages sent over a physical bus can be observed as the same message by all nodes on the bus. Therefore, from a physical layer perspective, all messages read off the bus at the same time are the same and consistent. Utilizing a broadcast messaging system's inherently consistent messaging takes away the possibility of attributing faults to the network for an inconsistent delivered message seen by multiple nodes. Such a messaging system truly provides the foundation for a distributed fault tolerant embedded system(28).

3.2 Communication Protocol Classification

Communication protocols can be classified by several different types of attributes. An example of a classifiable attribute is the protocol's message delivery methodology. Choosing this attribute for classification allows protocols to be grouped into such detailed categories as broadcasting, multicasting, or point-to-point. However, for the purpose of this thesis the claim is made that all communication protocols can be classified as belonging to one of two unique and slightly broader groups. The two classification groups are focused not on the delivery style, but on the messaging style. Protocols- using this messaging style classification- are grouped by whether the message sending within

the protocol is node oriented or message oriented. Both classifications will be discussed in detail in the following sections.

3.2.1 Node Oriented Protocols

Node oriented protocols are communication protocols that allow message transmissions to occur between nodes based on a node identifier. Although many nodes may exist on the network, the type of message sent and the corresponding data are only visible to the specific nodes directly identified as the intended receivers. Therefore, node oriented messaging protocols require an addressing schema to be embedded into each message. The intended address, being unique, is the network identifier that determines the allowed receivers of data. The advantage of this style of network comes in the safety and security of limiting a transmitted message's receivers. This forces the producer of data to have a knowledge of every intended consumer of its data. While this is a reasonable expectation, it often complicates the integration and debugging phase of a complex networked distributed system due to the intricate producer and consumer relationships that are dependent on network addresses. Examples of node oriented protocols which are seen in Figure 10 include I2C and the internet protocol TCP.

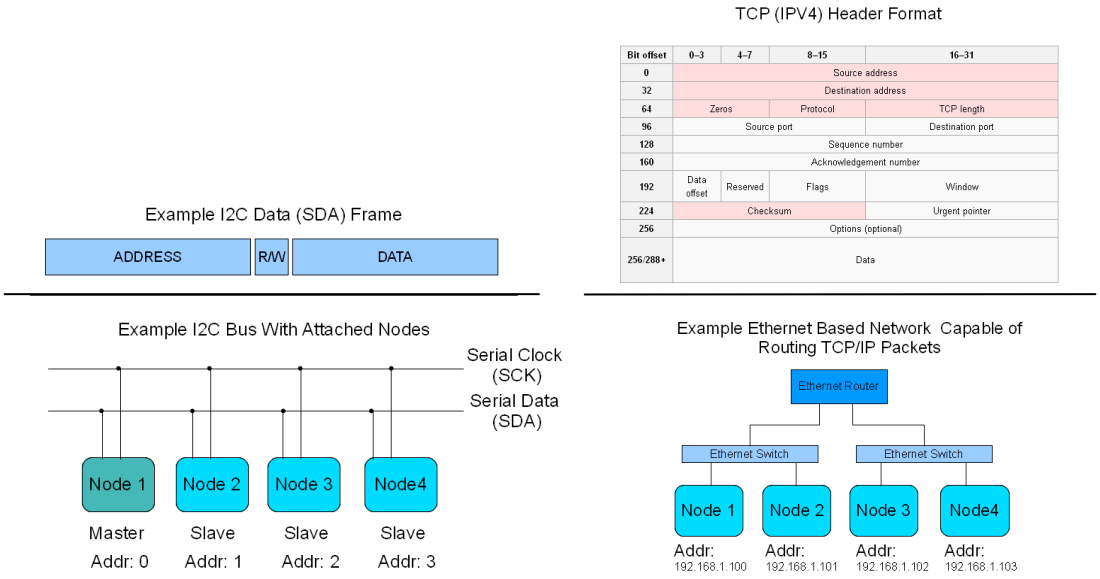


Figure 10: Data Frame & Network Configuration of I2C and TCP(29) Network

3.2.2 Message Oriented Protocols

Message oriented protocols are communication protocols that allow message transmissions to occur between nodes based on a message identifier. A message identifier is a descriptive label that is directly associated with the data being transmitted. Thus, when integrated into a broadcast bused network all nodes on the network may see the message identifier and then select whether the message should be processed or not. This allows multiple receivers to see needed data without the producer of the data having to perform several consecutive transmissions to individual nodes; thereby, reducing the potential total network traffic. Message oriented protocols also eases the configuration and integration phases of design by not forcing the tracking of multiple dependent consumer addresses. No tracking of consumer and producer addressing is required due to messages being transmitted and correspondingly accepted based upon message ids not network addresses.

There are many reasons that message oriented protocols are advantageous for fault tolerant distributed embedded systems. Among these reasons is the fundamental support that message oriented protocols provide for the independence of software modules. This independence means that each software module doesn't have to be associated with a physical location for intersystem communication. This location independence provides a key element of the base infrastructure that fault tolerant, reconfiguration supporting architectures such as ARDEA need. That key element is the ability for software modules and processes to be moved to redundant processing elements in the network as faults occur.

3.3 Network Protocol Selection for Fault Tolerant Applications

The base of any fault tolerant distributed embedded system is the network that connects each individual processing element. As there are many standard network protocols in existence, it is only responsible of an engineer to utilize the tools that are

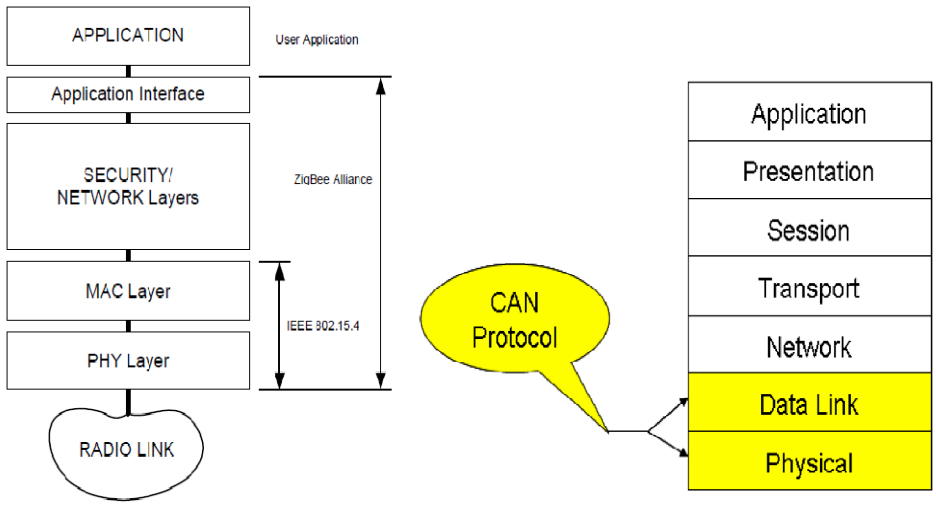
available, and thus save development time. However, not all protocols are suitable as a backbone for a fault tolerant distributed embedded system. As outlined in the previous sections, there are two major attributes that are vital for easing the development and integration of such systems. These attributes are broadcast support and message oriented data communication. It is the combination of these attributes that provide the needed infrastructure for the routing of data properly from a producer to its corresponding consumers regardless of either's physical location within the network. Hence, a network with these attributes creates the base infrastructure for a fault tolerant system.

An example of a fault tolerant supportive physical, hardwired bus protocol is CAN. There are many papers that site CAN as being the proper choice for both a real-time bus protocol (2)(30), and a robust fault tolerant supportive bus protocol (31). One key advantage of CAN for fault tolerant applications is its extensive protocol level error checking. This protocol level error checking ensures that message transmissions over the network are reliable and correct (31) (32). Another advantage of CAN is the robustness built into its physical layer. In the physical layer, CAN requires the use of differential 2-wire communication. This approach masks the effects of EMI noise, which adds to the reliability of data transmissions in noisy, harsh, common fault injecting environments. With reliability provided by extensive data error checking and reliable transmissions in faulty environments, CAN and similar protocols are appropriate selections as a fault tolerant supportive network.

Like the wired bus protocol CAN, the wireless protocol Zigbee is a viable solution as a fault tolerant supportive network. As a wireless mesh-networking standard, nodes within a Zigbee network have the ability to communicate in a one to many fashion. In a fully connected configuration a single node's messages may be seen by all other nodes in the network in a true broadcast fashion, as well. An advantage of using a wireless mesh network like Zigbee comes in the fact that the network has the ability to "self-heal" in the presence of a node failure(33). This means in a multi-node network when a node fails, messages will still be reliably delivered. Beyond data delivery reliability, extensive research has been conducted to guarantee that Zigbee operation is reliable in noisy environments such as operating environments where multiple similar

frequency band communication protocols (wireless Ethernet (802.11), Bluetooth, etc) are being used (34). With both the guarantee of reliable data delivery and wireless transmissions in faulty environments, Zigbee is a valid wireless protocol solution for the base of a fault tolerant network.

A common method of developing and comparing network protocols is by building and evaluating each of the seven layers of the OSI model. The development of a complete fault tolerant network is no different. Therefore, to fulfill the lower levels of the OSI model, as stated earlier in this section, the integration of existing networks are appropriate. Using the hardwired bus CAN as the base protocol of a fault tolerant network fulfills the requirements of both the physical layer (layer 1) and the data link layer (layer 2) of the OSI Model(35). The wireless protocol Zigbee utilizes the IEEE 802.15.4 standard as its means of fulfilling both the physical (layer 1) and data link (layer 2) layers. Per the Zigbee Alliance, the remaining levels of the OSI model within this protocol can be lumped into two levels the security/network layers and then a simple application interface layer (36). The perspective OSI model layer allocation of both protocols can be seen in Figure 11.



A B

Figure 11: Mapping of Zigbee [A](36) and CAN [B](35) to the OSI Model.

Regardless of the network protocol selected as the foundation of a fault tolerant supportive network infrastructure, additional work is needed to provide extensibility and usability within a fault tolerant architecture. This additional work will fulfill the remaining layers of the OSI model, by encompassing application message management, internal routing, and network access via a network interface driver and application interface middle layer, as seen in Figure 12. It is the addition of these layers that provide for the routing of data properly from a producer to its corresponding consumers independent of either's physical location within the network. This independence is the vital feature that gives a fault tolerant architecture the ability to reconfigure itself in the presence of faults.

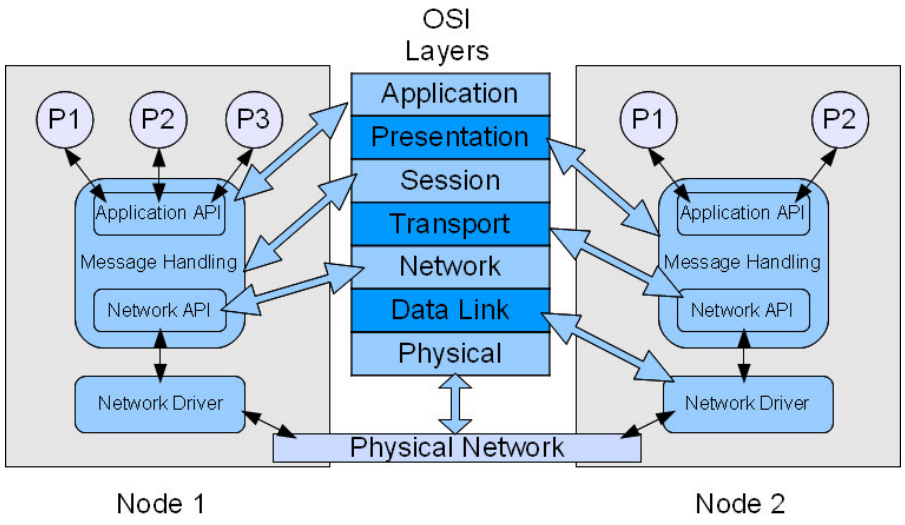


Figure 12: OSI Layer Mapping of a Fault Tolerant Supportive Network System

4 THE DESIGN OF A MESSAGE ROUTING LAYER

As stated in chapter 1, the goal of this thesis work is to develop an extensible, system-level solution for the management of producer and consumer data dependencies in fault-tolerant distributed embedded systems. Achieving this goal will result in rapid, reliable, embedded system design and development. To accomplish this goal MeRL was developed as a unique approach to data dependency management over both a virtual and physical network. MeRL's unique approach comes from it being a layer of abstraction for network data dependencies. This is visualized in Figure 13. In Figure 13A we see an example of typical network producer and consumer relationships, and the dependencies that the communication system has on location and fixed predetermined communication pathways. It can be seen that due to these dependencies it is a difficult process to initiate a reconfiguration in this common communication framework. Therefore, a system with similar inter-task dependencies does not inherently possess the ability to dynamically manage and reestablish network data producer/consumer dependencies. These abilities, which are needed for a system reconfiguration, are vital for the system to preserve functionality in the presence of faults. However, as seen in Figure 13B with the addition of MeRL, the system gains inter-task independence. This inter-task independence creates a system environment where tasks are not bound to network locations, and can be moved or reinstantiated in other locations within the system. Thus, with the integration of MeRL, a distributed embedded system's communication infrastructure can support reconfiguration. By supporting reconfiguration, the embedded system design can employ a fault tolerant design; hence, making it a reliable, fault-tolerant system.

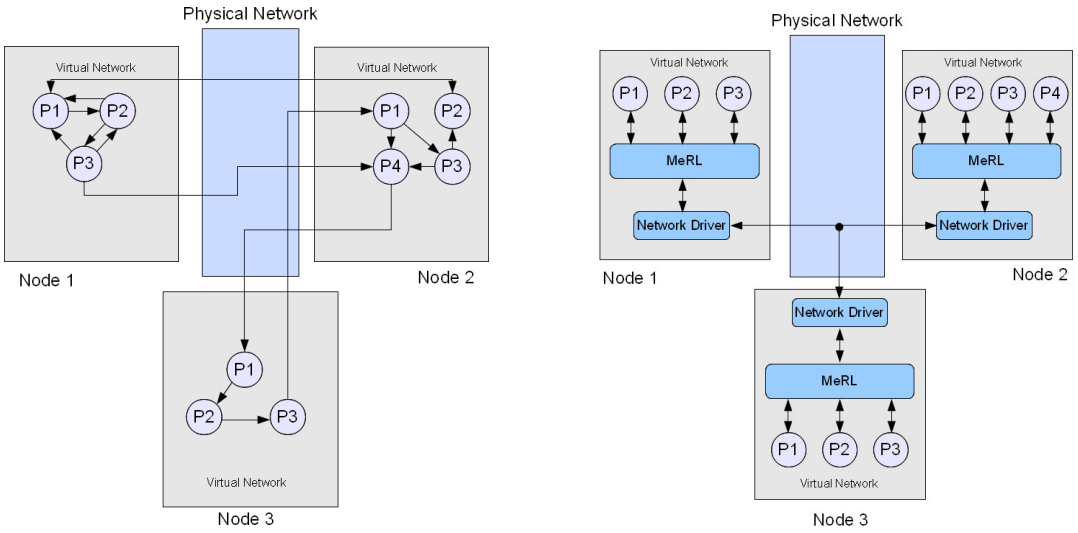


Figure 13: Example Network Dependencies Without MeRL [A] and With MeRL [B]

In this chapter, the design of MeRL which creates the framework for inter-task independence is detailed. First in Section 4.1 the justification for the design of MeRL and the requirements that MeRL adheres to are presented. Then Section 4.2 highlights the high-level design, as well as the intricate mechanics of data flow among distributed tasks through MeRL. Next section 4.3 highlights the two application programming interfaces (APIs) the MeRL infrastructure incorporates. It will be shown in this section how the implementation of both APIs (Application Layer and Network Layer) allows MeRL to be easily integrated into an embedded system design. With the completion of this chapter, the governing principles of MeRL's design will be understood.

4.1 MeRL's Design Requirements

The design and implementation of MeRL was inspired in part by the need for dependable modular design in a real time embedded system. In such University of Kentucky projects as BIG BLUE and several UAV designs for the student AUVSI UAV competitions (13)(7)(6)- the observation was made that the implementation of complex embedded systems is completed faster when the system is divided into several modular

and distributed function blocks, worked on independently, and then integrated together to form a completed system. However, the integration process was complicated by the use of typical producer and consumer relationships similar to those pictured in Figure 13.A. Through these experiences it became clear that if the system were divided into individual tasks and implemented independently, then the easiest way to integrate the tasks would be to keep them all independent of each other despite data dependencies created by their roles as producers and consumers. This was a difficult concept at first, as there are always dependencies based upon the need for data generated and utilized by tasks in the relationship roles of producers and consumers of data. To solve this problem there was only one solution. The solution was to force this independence by creating a single layer for all tasks to communicate through as seen in Figure 13.B. This layer would be responsible for routing all data within the distributed embedded system, hence eliminating any direct point to point dependencies. Thus, the concept of MeRL began.

In the early conceptual stages of the design of MeRL, another benefit for developing inter-task independence was brought to the foreground. This benefit was the inherent support an embedded system could have for system reconfiguration with MeRL. It was conceptualized that reconfiguration would be supported if all communicating tasks were to be kept independent from one another on both a single processor design, as well as a networked multi-processor design. By adding a physical network and the potential to distribute tasks among multiple processors the system would also have physical redundancy to rely upon in the presence of faults. With this concept of reconfiguration support of a multi-processor design being a guiding principle of MeRL, MeRL had the potential to become a tool that could be incorporated into fault-tolerant embedded system designs. This not only solidified the requirement that the system could not allow traditional direct communication dependencies between two tasks, but it also meant that tasks communicating over a network could not be dependent upon location.

MeRL, being fully conceptualized, had several requirements that the design needed to fulfill. The first was that the inclusion of MeRL would help to streamline the integration of several tasks. The second was that the inclusion of MeRL would be a tool for the implementation of a fault tolerant distributed embedded system design. Both of

these system level requirements are met by the way MeRL provides for inter-task independence. With this inter-task independence comes a third requirement that MeRL will support the routing of data between tasks on the same processor as well as tasks located on independent processors. Thereby, making the communication methodology between all tasks the same which results in location independent messaging between tasks.

4.2 The Design of MeRL

Having the conceptual requirements for MeRL in place the physical design of the message routing layer followed. This design, as seen in Figure 14, fulfills each system requirement that was highlighted in the previous section. These requirements are fulfilled with the way the data flows into, through, and out of the MeRL layer. Figure 13.B shows the placement of MeRL within an embedded system design. All processing nodes consist of individual tasks that reside above MeRL. Therefore, all internal and external inter-task communication is handled by passing messages to MeRL. If the system requires inter-task communication over multiple processors, a network layer may be integrated below the MeRL level. Thus, MeRL is a middle layer that handles all the passing of data within the system.

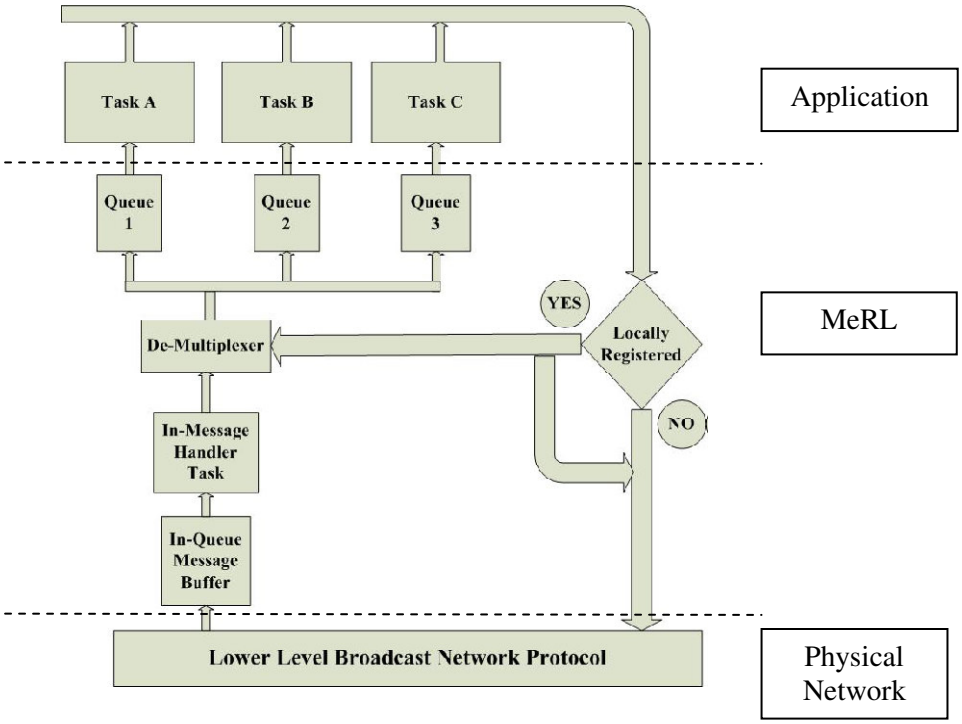


Figure 14: Top Level Block Diagram for the Design of MeRL (18)

The internal data flow structure of MeRL is seen in Figure 14. All data originates from the tasks of the system. Once data is created, it is sent from the task to MeRL. To fulfill the requirement that internal data is to be routed within the same processor, the first stage within MeRL is a check to see if any local task may want access to the data. This is done through a registration process. With MeRL, in the design phase all data to be generated within the system is associated with a predetermined id. To gain access to this data, tasks which need this data simply register to receive the desired data id. Hence, if a local task is registered to receive data that is generated by another task on the same processor, this data is routed directly to the corresponding task's incoming message queue. After this local registration check is performed on the data, if physical network support has been enabled the message is then broadcast with its associated id over the physical network. This allows any listening processor node to check if any residing task has registered to receive this data message. If one or more tasks have registered to receive the data message the data is placed into an incoming message queue used as a first in first out (FIFO) buffer of incoming data. Once data has been placed into the

incoming message queue an incoming message handler task is signaled to handle the message. In this task, the message id is used to parse through the local task data registration, and the message is placed into the appropriate task input buffer queues. It is then the responsibility of the author of each task to handle the input data in FIFO order. With the appropriate tasks receiving data the dataflow cycle is completed.

4.3 MeRL Integration

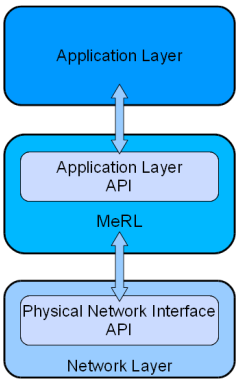


Figure 15: Visualization of MeRL and API Placements

The implementation of the mechanics of MeRL is a middle layer design which resides below the application layer and if present above a physical network layer as seen in Figure 15. Therefore, to interface to both levels the design of MeRL utilizes a strict Application Programming Interface (API) to interface with each adjacent layer. Both APIs are discussed in the following sections.

4.3.1 MeRL Application Layer API

```
ERROR_TYPE   init_taskqing (void);
ERROR_TYPE   init_messaging (void);
ERROR_TYPE   reg_msg (MSG_ID_TYPE id ) ;
ERROR_TYPE   unreg_msg ( MSG_ID_TYPE id );
ERROR_TYPE   unreg_all (void);
ERROR_TYPE   send_msg ( MSG_ID_TYPE id , MSG_TYPE payload ) REENTRANT;
ERROR_TYPE   send_local (MSG_ID_TYPE id, MSG_TYPE payload) REENTRANT;
ERROR_TYPE   get_msg ( MSG_ID_TYPE *id, MSG_TYPE *payload ) REENTRANT;
ERROR_TYPE   accept_msg ( MSG_ID_TYPE *id, MSG_TYPE *payload ) REENTRANT;
ERROR_TYPE   num_waiting_q_msgs (INTEGER_TYPE *num_msgs) REENTRANT;
```

Figure 16: MeRL to Application Layer API Function Prototypes

The most prevalent API to a task developer is the API which unites the task or application layer to MeRL. This API can be seen in Figure 16 written in function prototype form. The first two API functions shown are the init_taskqing (void) and init_messaging (void) functions. These two functions as their names imply are the initialization functions for the MeRL system. The function init_taskqing is utilized to incorporate the native operating system mechanisms needed to initialize all the queue input buffers of the system. These queue input buffers include each task's input queue as well as the receiving buffer queue for the network interface when the network interface is present. The function init_messaging is a function which is called at the beginning of each task to dynamically associate a created input buffer queue with the calling task. This function effectively removes the task developer from any operating system setup needed to interface to MeRL for message reception. The next API function call listed is the reg_msg (MSG_ID_TYPE id) function. The reg_msg (MSG_ID_TYPE id) function is called from within a task to request that the task receive all messages associated with the passed id. It logically follows that a developer may desire to unregister an id that it is

associated with; thus, the function unreg_msg (MSG_ID_TYPE id) is provided in this API. Likewise, if a developer wishes to disassociate itself from all data traffic at one time- the unreg_all (void) function can be utilized as it unregisters the calling task from all previously associated message ids.

The remaining listed functions in the MeRL to Application Layer API are the heart of MeRL and are used for data message handling. Being the primary data handling functions, these final few functions should be implemented as reentrant code to provide for operational safety. This means the code of the function can not be self-modifying during its execution(37). With adherence to a reentrant implementation these functions can be safely called and executed simultaneously from multiple tasks.

The first listed reentrant function is the send_msg(MSG_ID_TYPE id, MSG_TYPE payload) function. This function takes as arguments the message id of the data to be transmitted, as well as the packaged data message payload. The send_msg function checks for local registration, passes the payload to any local task registered for the reception of the data id, and then passes the payload to the network layer to broadcast the id and payload over the network. The next function send_local (MSG_ID_TYPE id , MSG_TYPE payload) gives the task author the ability to send data only to local tasks registered to receive the message payload. The next listed API function is get_msg(MSG_ID_TYPE *id, MSG_TYPE *payload). The purpose of the get_msg function is to suspend the calling task until a message is pushed into the calling task's input buffer. Once data is received, the function processes the data by pushing the first message and id into the referenced payload and id variables. The next listed API function is the accept_msg(MSG_ID_TYPE *id, MSG_TYPE *payload) function. Like get_msg, this function is used to retrieve data from the task input message buffer; however, accept_msg differs in that it is non-blocking. The accept_msg function simply checks to see if any new data has been pushed into the input buffer. If no new data exists the function simply returns a value associated with "no new message received". However, if data is in the input buffer then accept_msg processes the data by pushing the first message and id into the referenced payload and id variables. Finally, the last function call in the MeRL to Application Layer API is the num_waiting_q_msgs

(INTEGER_TYPE *num_msgs) function. The num_waiting_q_msgs is a simple function that looks at the input queue of the calling task and counts the number of queued messages. This value is then loaded into the reference variable passed through the num_msgs pointer argument.

It is important to note that unless otherwise stated the return value of each function is any error that may have occurred during the execution of the function.

4.3.2 MeRL Network Interface API

```
ERROR_TYPE    init_network ( void );

ERROR_TYPE    reg_pkt ( MSG_ID_TYPE msg_id );

ERROR_TYPE    unreg_pkt ( MSG_ID_TYPE msg_id );

ERROR_TYPE    get_pkt ( MSG_ID_TYPE *msg_id_ptr , MSG_TYPE *payload_ptr );

ERROR_TYPE    send_pkt ( MSG_ID_TYPE msg_id , MSG_TYPE payload );

ERROR_TYPE    msg_receive ( void ) REENTRANT;
```

Figure 17: Network Interface to MeRL API

If the decision is made by the system designer to implement a distributed architecture with multiple processing nodes then a physical network layer will be present. To manage the interprocessor communication, MeRL can be added. To integrate a network layer beneath MeRL the system network driver must simply be wrapped into the API functions listed in Figure 17.

The first MeRL to Network Interface API function listed is the init_network(void) function. This function simply contains all the initialization calls required to enable the network. This function is called independently of the MeRL API by the implementer, when a network is known to be incorporated. The next function listed is reg_pkt(MSG_ID_TYPE msg_id). A reg_pkt function implementation should

incorporate the code necessary to enable the network reception of a data payload associated with the passed message id. The reg_pkt network function is wrapped into the reg_msg function from the MeRL to Application Layer API. The next function in the API is unreg_pkt (MSG_ID_TYPE msg_id). The unreg_pkt simply disassociates the processing node's reception of the passed message id's associated payload message from the network. This function is incorporated into the MeRL to Application Layer API function unreg_msg(MSG_ID_TYPE msg_id). The next function listed is get_pkt(MSG_ID_TYPE *msg_id_ptr , MSG_TYPE *payload_ptr). This function is typically used in a stand alone mode of operation independent of the network driver. Its purpose is to load the latest received message id and payload packet received over the network into the corresponding variables referenced in the function call. The next listed function in Figure 17 is send_pkt (MSG_ID_TYPE msg_id , MSG_TYPE payload). The send_pkt function is responsible for placing the passed payload and message id onto the network. This function is wrapped in the send_msg function of the MeRL to Application Layer API. Finally, the last API function is msg_receive (void). The msg_receive function is responsible for moving received data from the network to the MeRL incoming data buffer queue. This function is most effectively called from the network ISR which is triggered upon the receipt of incoming data. However, if the network peripheral does not provide an ISR this function maybe called within a network handling task.

5 IDEAnix

Embedded systems often require the use of an embedded operating system. Whether it is to utilize implemented data structures or to have system support for task scheduling- incorporating an embedded operating system into a design inherently aids in the development process. It is for this reason IDEAnix was created.

IDEAnix is a custom build of the standalone RTOS microC/OS-II. IDEAnix utilizes microC/OS-II's efficiently designed data structures and scheduler to incorporate an implementation of MeRL for inter-task communication. Also included in IDEAnix are peripheral drivers for the targeted Silicon Labs C8051 microcontrollers including a custom built CAN driver (38) implemented to the API specifications found in section 4.3.2 for network communication.

The following sections of this chapter detail the selections and features included through the development of IDEAnix. Section 5.1 lists the OS features that were the discriminators in the selection of microC/OS-II as the base operating system for IDEAnix. In Section 5.2, the focus is the main development of the pre-compiler macros written for dynamic resource management within IDEAnix. Finally, Section 5.3 overviews the IDEAnix included peripheral drivers for the targeted c8051f04x Silicon Labs microcontroller platform.

5.1 Operating System Selection for the Integration of MeRL

The proper selection of an embedded operating system can be a daunting task. There are many factors that play into the selection. One of the most crucial factors is the intended application that the operating system is to support. In other words, does the operating system include the appropriate data structures, resource management constructs, hardware interface methods, and schedulers that the intended application requires? Another key factor that plays a role in the selection of an operating system is the resource constraints that an operating system requires. In embedded applications the

microcontroller/microprocessor is often selected based upon the physical hardware interface requirements of the system it is to control. For example, if the system being built is a sensor interface module that must interface to I2C temperature sensors and analog output pressure transducers, then the microcontroller selected should have the necessary peripherals to support these common sensor interfaces. It is typical of microcontrollers with such interfaces to have limited RAM and flash memory space. Therefore, the operating system selected for such an application should be selected to match the available resource constraints imposed by the targeted system's microprocessor/microcontroller. One additional requirement for selecting an operating system is based on proven reliability. There exist certifications to validate the reliability of operating systems and software implementations. These certifications may be obtained with the demonstration of reliability through various simulations or viable real-world application deployment and testing. Whether through certification or proven coding practices, the stability of the operating system is crucial. It is crucial because the stability of the selected operating system determines the stability of the application that utilizes it.

All these mentioned factors influenced the selection of which operating system would be utilized for IDEAnix. The intended purpose for IDEAnix was to provide a package of an existing operating system with an efficient implementation of MeRL. To demonstrate the utility of IDEAnix with MeRL, a UAV design was proposed as the target platform. The UAV design included a modular avionics architecture designed around a CAN bus network. Having to support a CAN network the 8051 cored C8051F04x series microcontroller from Silicon Labs(8) was selected as the target hardware for the development of IDEAnix. Having these guidelines in place the operating system selection was narrowed. The embedded operating system had to fit on a microcontroller with only 64kB of flash while leaving enough space for user code. With this in mind the embedded RTOS microC/OS-II went to the top of the selection list. This RTOS had already been ported for use with the 8051 core(5), and had a scalable footprint of 5K to 24Kbytes(39). This restriction was well within the resource limitation of the C8051f040 microcontroller. Not only had microC/OS-II been ported to the 8051 and met the resource constraints of the C8051f040, but it also had passed the aerospace industry

standard for use in DO-178B applications (39)(40). This means that microC/OS-II has been certified for use in safety critical systems where human life is at risk(40). As for its support of an implementation of MeRL, microC/OS-II included implementations of datastructures such as queues for task input buffers, and semaphores for resource management and event triggers. It also included constructs for operating system calls from system ISRs which would allow for direct loading of the network input queue buffer. Beyond those features, the task scheduler of microC/OS-II was implemented as a real-time priority based scheduling routine. This was perfect for MeRL- as network data comes in, it can be routed with the highest priority to ensure data delivery is handled as fast as possible.

As can be seen, microC/OS-II was the perfect operating system to be wrapped into IDEAnix. It offered all the necessary support for an efficient implementation of MeRL. Also, it had already been ported to the 8051 processing core used by the targeted microcontroller platform, and could fit within the resource limitations imposed by the hardware. Finally, microC/OS-II had been certified for use in DO-178B aerospace applications, which ensures that it is a stable and reliable operating system.

5.2 The Development of IDEAnix

The development of IDEAnix was centered on integrating a resource efficient implementation of MeRL with microC/OS-II on a resource limited Silicon Labs c8051f04x microcontroller. The approach taken for handling resource management was to only compile the necessary code blocks required by the application, adjust RAM based array sizes, and limit the inclusion of unused variables. To accomplish selective code compiling and resource throttling, pre-compiler macros were utilized. To control the macros a list of macro variables were defined that a system developer can set to throttle both the code included and the RAM being allocated for MeRL use. These macro variables are seen in Figure 18.

```
#define NETCOMM_EN      1
#define TASKCOMM_EN     1

#define MAX_ID_VAL      25
#define MAX_TASK_PRIO   10

#define SIZE_of_Q       10
#define NUM_TASKs       7

#define TASK_STK_SIZE   512
```

Figure 18: The Set of Variables Used to Control Resource Utilization

IDEAnix packaged MeRL with microC/OS-II which provided a developer with a tool for inter-task communication management. It was important that this integration not limit the end developer and needlessly use code space if unutilized. Thus, although MeRL is a very powerful tool and can be used for any size project; it is possible that a developer may not need such a powerful communication management tool for all applications. As a result, the defined variable TASKCOMM_EN is included as a boolean to signify whether the source code for MeRL should be compiled (TASKCOMM_EN = 1) or not (TASKCOMM_EN = 0). Likewise, MeRL provides a mechanism for managing inter-processor communication via a physical network implementation. In this build of IDEAnix, the physical network interface included is a CAN driver. Once again, IDEAnix was designed to be functional for several end applications, and it was seen that not all applications would require a physical network interface. Therefore, the defined variable NETCOMM_EN is used to include (NETCOMM_EN =1) and remove (NETCOMM_EN = 0) the physical network interface code at compile time. It is important to note that MeRL can still be included and function for same processer inter-task communication without the physical network being included (TASKCOMM_EN = 1 ; NETCOMM_EN = 0).

Just as TASKCOMM_EN and NETCOMM_EN were included to control code space utilization, additional defined variables were designed into the pre-compiler macro to control MeRL's RAM utilization. These variables were MAX_ID_VAL, MAX_TASK_PRIO, SIZE_of_Q, NUM_TASKs, and TASK_STK_SIZE. The variable MAX_ID_VAL holds the maximum payload identifier value. This is the largest

identifier value that is assigned to a data message within the systems messaging schema. This variable is used to set the size of an array that tracks which tasks have requested particular id data messages. Although this array is not anticipated to require a large amount of memory, it is important to realize that its size is directly proportional to the number of message ids used for message identification. The next defined variable is MAX_TASK_PRIO. This variable holds the maximum priority value assigned to a task. It is used in two separate pre-compiler macro conditionals. The first macro was created to limit the number of task message queues created to only the exact number required. Its second role is as a conditional check to ensure that the maximum number of priorities allowed is not exceeded. If a developer attempts to exceed this threshold a compile-time error is given to alert the developer. The next variable is SIZE_of_Q. This variable is used to allocate the number of elements that will be created for each tasks input message buffer queue. Next, the variable NUM_TASKs is used to hold the number of tasks used within the application. This variable is used to allocate the appropriate number of task stacks. It is also used in a macro to perform a safety check to ensure that the amount of RAM being allocated for use as task stacks does not exceed the hardware limitation. This macro will signal a compile-time warning to the developer. The final defined variable, TASK_STK_SIZE, is used to define a common task stack size. In truth, for the most efficient application implementation the size of each task's stack should be determined on a task by task basis. In such a scenario this variable should be assigned the maximum stack size allocated for a single task. Then, TASK_STK_SIZE would still function in the pre-compiler conditional used to serve as a warning for the possibility of the over utilization of RAM.

The implementation and integration of these defined variables can be found in APPENDIX: Demonstration UAV Source Code.

5.3 IDEAnix Driver Support of Silicon Labs c8051f04x

The targeted processing platform for this build of IDEAnix was the Silicon Labs c8051f04x CAN enabled microcontrollers. Therefore, to support microC/OS-II and to

provide the end developer with serial interfaces, IDEAnix includes a set of standard C8051f04x peripheral drivers. This set of standard peripheral drivers includes system clock, I/O crossbar assignment, timer, and serial UART initializations.

```
#define UART0_BAUDRATE 9600
#define UART1_BAUDRATE 9600
#define SYSCLK         24500000

void SYSCLK_Init   (void);
void Port_IO_Init  (void);
void TimerInitiate (void);

void init_network (void);

void UART0_init (void);
void UART1_init (void);

void init_all (void);
```

Figure 19: IDEAnix Peripheral Driver Support for Silicon Labs C8051f04x

The function prototypes of the standard driver initializations and adjustable defined variables are seen in Figure 19. The first function listed is SYSCLK_Init (). This function utilizes the internal oscillator of the targeted Silicon Labs C8051f041 microcontroller as its clock generation source. It sets the selectable oscillator divider scale to 1 which ensures the system clock rate is set to 24.5 MHz. This set system clock rate is stored in the defined variable SYSCLK for global use in internal time related calculations. If a design were to incorporate an external crystal, then this function should be modified to accommodate the selection of an external oscillator source. The next initialization function listed is Port_IO_Init (). This function is used to initialize the settable input/output (I/O) crossbar of the Silicon Labs C8051f041 by assigning peripheral I/O to the pins of the microcontroller as outlined in the C8051f041 datasheet (41). Port_IO_Init() sets the I/O crossbar to ensure functional pin compatibility with the Tiny Interface Module hardware design(42). The IDEAnix default crossbar configuration and pin assignment is seen in Figure 20. If IDEAnix is to be used on a C8051f041 microcontroller incorporated with another PCB layout, this function's register settings should be reviewed to ensure the I/O pin assignment is compatible. The next

function shown in Figure 19 is TimerInitiate (). TimerInitiate() initializes internal timer 0 to a clock rate of 100 Hz which is a period of 10 ms. This function enables the timer 0 interrupt service routine (ISR) to be triggered at this set period. Timer 0's interrupt is associated with the microC/OS-II system clock ISR- OSTickISR(). Thus, OSTickISR() is entered at a rate of 100 Hz, and is responsible for calling OSTimeTick. Because of this association the microC/OS-II OS tick rate occurs every 10 ms. The remaining functions are microcontroller specific peripheral driver initializations. The init_network() function initializes the physical network as outlined in section 4.3.2. Targeted for the c8051f041 the IDEAnix included init_network() implements the initialization for the CAN peripheral. The following two functions – UART0_init() and UART1_init()- are the initialization functions for the c8051f041 UART serial peripherals. These two functions use the values set in the defined variables UART0_BAUDRATE and UART1_BAUDRATE to dynamically set the baudrate of the serial port peripherals. Finally, the last initialization function is init_all(). This function is used to initialize all the application peripherals by making calls to all the initialization functions. If desired, developers can also place their additional driver initializations in this function to centralize the location of all peripheral initializations in their project.

```
P0.0  -  TX0  (UART0)
P0.1  -  RX0  (UART0)
P0.2  -  SCK  (SPI0)
P0.3  -  MISO (SPI0)
P0.4  -  MOSI (SPI0)
P0.5  -  NSS  (SPI0)
P0.6  -  SDA  (SMBus)
P0.7  -  SCL  (SMBus)

P1.0  -  TX1  (UART1)
P1.1  -  RX1  (UART1)
```

Figure 20: IDEAnix Default Silicon Labs C8051F04x Crossbar Configuration

It is important to note that not all peripheral initializations and support drivers were included in IDEAnix for the targeted Silicon Labs c8051f04x microcontroller. The peripheral drivers that were included, were packaged to provide a stable known microcontroller system clock, support the embedded RTOS internal tick counter, and provide the developer with UARTs to support application debugging and simple serial interactions. It was seen that the other peripherals implementations could not be

generalized to fit a broad application domain. Therefore, the remaining initializations and drivers were left for implementation by the end developer to support their unique end application requirements.

6 A UNIQUE EMBEDDED SYSTEM DESIGN PROCESS

There are two basic and very different approaches when it comes to the implementation of a system. The first is commonly known as the brute force method. This method is a "get it done" approach. Little attention is given to project planning or design, the system is simply built. Issues such as design flaws, integration difficulties, and system bugs are handled as they arise. On the other hand, there is the second method which is known as the systems approach. The systems approach is a method where every aspect of the system is thoroughly planned. Each phase of the project's life cycle is detailed before its beginning. This helps in avoiding costly delays in subsystem implementation, system integration, system testing, and system deployment. With effort placed in the planning of each phase, the systems approach proves to be a more robust and efficient option when compared to the alternative brute force method.

This chapter consists of two sections. The first section is an elaboration on the systems approach and subsequent project lifecycle. The following section then outlines how this thesis work and the resulting MeRL, if integrated into an embedded system, can benefit several stages of an embedded system's lifecycle.

6.1 Systems Engineering - System Lifecycle

As stated in the introduction of this chapter, the systems approach of facilitating a project is a robust and efficient method of handling the implementation and deployment of a system. In fact, the systems approach has been adopted and implemented by several different industries including military suppliers, consumer product manufacturers, and even the automotive industry. There are dedicated departments within the corporations of these industries that work continuously on ensuring the systems approach is followed for every project undertaken. Because of this, there is an industry need for individuals who understand how to apply the systems approach. Due to the industry need, many

universities have begun to offer degree programs in this field of study. This resulting field of study and its application has become known as systems engineering.

One of the key elements of systems engineering is the breaking down of a project into manageable phases. The connection and sequence of these phases can be represented in what is known as the system lifecycle. The quantity of phases, respective titles, and scope vary depending on which agency or corporation defines the system lifecycle to make it unique to their entity (43)(44) (45)(46). Despite these variances, the objectives of the system life cycle can be represented in eight independent phases that all systems experience as visualized in Figure 21. These phases start by taking a system concept and defining the system requirements, then moving on through the other stages of system design, development, production/construction, distribution, operation, maintenance, and finally system phase-out/obsolescence.

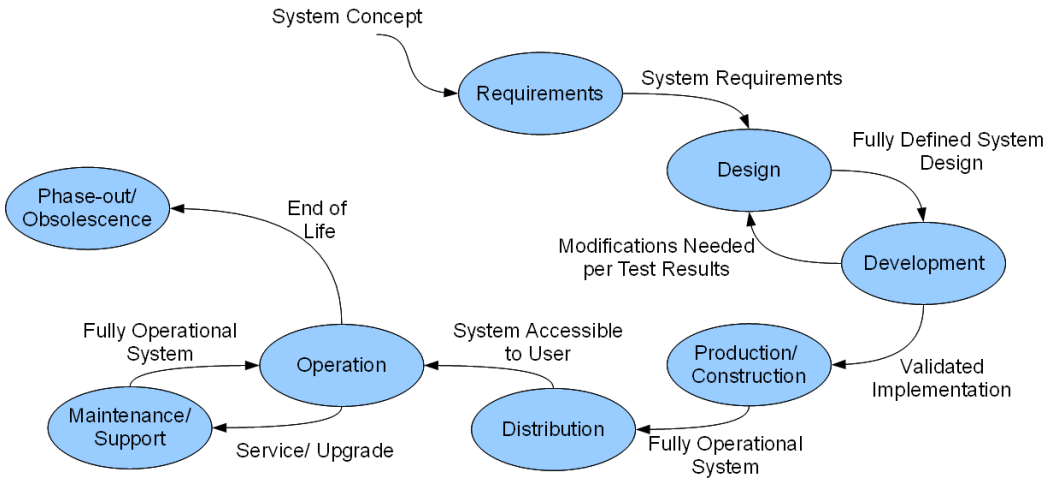


Figure 21: System Lifecycle Diagram

The definition of the system requirements is the first phase of the system lifecycle. System requirements are defined by the intended purpose and application of the system to be implemented. Thus, it is in this stage that the concept and vision of what the system needs to accomplish and how it must perform is fully defined. In some cases,

capturing the requirements of how a system is to perform is lumped into the design stage. However, for the purposes of this thesis, the requirements phase of the system lifecycle is seen to be an important independent stage.

The second stage is the system design phase. It is in this stage that a conceptual method to achieve the functionality mandated by the requirements is defined. All the corresponding inputs, outputs, connections, and general dependencies are decided in this stage. A result of this stage is the creation of a well planned high level design of the system. This allows engineers to work through and plan all the details of each of the subsequent stages of the system lifecycle.

The next stage of the system lifecycle is the development phase. In the development phase the concepts captured in the design phase start to become reality. It is in this phase that each portion of the system is implemented. Detailed schematics of hardware are completed, software/firmware is written, and mechanical drawings are made. Care is taken to ensure that all requirements are adhered to. At this point it may be appropriate for a protoptye to be developed and tested to ensure all development decisions are functional and that each requirement was met. If any issue occurs, the design is reviewed, changes are made, and the development process begins again implementing the appropriate modifications. At the end of the development cycle, each subsystem is finalized and carefully reviewed before the next phase is started.

Following the requirements, design, and development phases comes the production/construction phase. As the name implies, it is in this phase that the system is materialized for production. All subsystems are constructed and integrated to build the system as a whole. Once completed, the system is once again thoroughly tested to ensure that all imposed requirements were fulfilled, and thus the system functions as anticipated. Once all specification testing is completed on the constructed test units and all anomalies are addressed, the system is then placed into full production. The production portion of this phase is when the finished product is finally constructed. All requirements have been verified and all issues have been worked through in an organized fashion resulting in the manufacturing of a quality system.

With a quality system built, it then goes into the distribution phase. It is in this phase that the completed system is delivered to the end user. The end user with product in hand then begins the operation phase of the system lifecycle. This refers to the period of time the system is used by the end user for its intended purpose.

During the course of the system's operation phase, the system may begin to malfunction, need attention due to general wear and tear, or simple improvements may become desired by the user. This begins the maintenance phase for the system. During this phase it is common for both general upkeep and repairs to be performed. So the parts of the system designed to be replaceable are replaced. Also, problems that occur during operation are addressed. This usually results in updates being released or product recalls being issued. Thus, the maintenance phase can be either short or very long based upon the application domain of the system.

Once the system has been in the operation phase and in some instances cycling in and out of the maintenance phase a conscious decision is made whether the system should go into the final phase of the system lifecycle. The final phase of the system lifecycle is system obsolescence/phase-out. There are several reasons why a system would be phased out. For one, as time goes by a system's functionality may simply no longer be needed. In other situations, the system may become outdated as newer technologies are made available. This can be seen in the trend in modern computing and consumer products, as there are always newer, better, and more competitive systems that are being placed into production and distribution. Regardless the scenario, as in life all good things- even technical systems- must come to an end.

6.2 A Systems Approach with MeRL

MeRL, through its unique design, is a tool that is more than just an embedded system data message routing mechanism. By selecting to utilize MeRL in an embedded system design, the system designer will have guiding rules and design principles to aid in

the design and development of a communication layer. As a result, these rules and design principles aid in achieving the goals of several stages of a system's lifecycle.

The decision to use MeRL starts in the first stage of the system's lifecycle. It is in this requirements capture phase for the system that the selection of MeRL either becomes a viable option or not. If requirements for fault tolerance, system modularity, and ease of integration are mandated- then the use of MeRL is a viable option to include in the design.

In the design phase the requirements for the system are reviewed, and a decision is made of whether to utilize the features of MeRL. If MeRL is selected, then the system designer now has a guide for data dependency management over a communication infrastructure. The conceptual network for inter-task and inter-processor communication is now bound to the constraints of MeRL, which aids with the design phase mapping of direct producer and consumer relationships within the network. The producer and consumer dependency mapping becomes cumbersome as new data producers and consumers are added into the system. This is especially difficult when redundant elements are added in the scenario of developing a fault tolerant system where the dependency mapping can change dynamically during operation. It is known that MeRL reduces the complexity of direct producer and consumer dependencies by abstracting them into their individual roles. This abstraction is done by assigning all data fields an identifier. Therefore, in the design phase, the system designer must design the identifier schema so that all data to be produced within the system can be tracked by assigning each data field an identifier. Assigning the data identifiers in the design phase will provide developers with a reference of ids for all data produced within the system. Hence, MeRL inherently provides a method of tracking produced data within the system, and enforces a design principle of identifying all data that will be created and shared within the system.

As in the design phase, a developer during the development phase does not have to track where data produced must go or where to go to retrieve necessary data. A developer with MeRL simply registers for the data it must receive or associates the data produced with the appropriate identifier. In an embedded coding environment, this is simply done with the inclusion of a common list of defined constants associated with data

that is shared with all developers. It is common during the development of a system that the need for more data arises. With MeRL, it is easy to make these additions by simply adding a new identifier to the list of existing identifiers that are shared among all developers. It is also important to note at this stage, that even if the system has been divided into subsystems by individual processes, and a resulting producer and consumer of data are placed on the same processor MeRL will properly handle the data routing internal to the processor. Thus, no additional inter-task communication data structures (pipes, mailboxes, queues, etc.) need to be created or managed.

After the development phase comes the construction/production phase. The most critical portion of this phase is the integration of all the individual subsystems. This is where the utilization of MeRL in the design and development phases is most helpful. Due to the way that MeRL abstracts the producer and consumer relationships, the integration of the completed subsystems over a network is seamless. Having each consumer node register for the data it needs and having the producer simply associate all of its output data with the corresponding predefined identifiers simplifies the integration process. Since there are no dependencies of data based on addresses, subnets, etc- each node or subsystem can simply be plugged into the network and begin operating. From a communications standpoint with the selection of an appropriate physical network packaged with MeRL- the only major system integration concern is the correct association of identifiers with data. This is easily guaranteed with the inclusion of a shared predefined list of data identifiers among developers.

The next phase MeRL assists in the system lifecycle is the operation phase. MeRL, as mentioned in previous chapters, supports reliable system operation by its being a network interface and data routing system that supports reconfiguration. This means that fault tolerant systems implemented via a reconfigurable architecture such as ARDEA that include MeRL have the ability to properly route data in the presence of faults. MeRL also assists with the communication needs of systems that have a modular plug and play environment. Since direct producer and consumer dependencies within MeRL are masked, nodes can be easily added and removed without any major integration effort during the system's phase of operation.

Finally, the last phase of the system lifecycle that the integration of MeRL may assist is the maintenance phase. Key elements of the maintenance phase include system upgrades and subsystem replacements. MeRL is a tool that can assist the process of either of those elements as they pertain to the processing nodes within the system. If there is a need to add or replace a node that either produces or consumes data within the system, it is simple to do with MeRL. Just as in the integration phase, the new or replacement node can simply plug into the network and begin operating. The routing of data is then handled by MeRL. Likewise, upgrades are often necessary. An example is where a completely new data field is needed within the system. With MeRL the addition can be added into the shared data field identifier list, deployed, and the new producers and consumers within the network can then begin to transmit data with the new identifier as well as register to receive it.

As can be seen in this section, MeRL is a powerful tool that will provide design principles and guidelines in the design phase, make the communication development process easier, support reliable fault tolerant operation, and be the support needed for quick easy communication maintenance needs.

7 IMPLEMENTATION AND TESTING

The design and implementation of MeRL was based on the need to efficiently handle inter-task communication in modular distributed embedded systems. This need to have a data routing mechanism incorporated into the design of an embedded system became very apparent in the development of several UAV designs. As such, a UAV system became the intended development test bed for a deployment of an IDEAnix based distributed embedded system with MeRL handling all inter-task communication.

Therefore, in this chapter details of both the implementation of IDEAnix and MeRL utilized in the UAV test system are discussed. Starting in section 7.1, which discusses the implementation of IDEAnix from the perspective of the development involved in incorporating a port of microC/OS-II. Next, in section 7.2 the implementation of MeRL for the targeted Silicon Labs c8051f04x microcontroller is detailed. Then in section 7.3 details of the development of a modular UAV intended as the targeted system for the first deployment of MeRL and IDEAnix are discussed. Section 7.4 describes aspects of the predictable and timing associated with MeRL. Finally, in section 7.5 the results of the first deployment of IDEAnix and MeRL are reviewed.

7.1 IDEAnix Implementation

As discussed in chapter 5 “IDEAnix”, IDEAnix was developed to provide a package of an existing RTOS with an implementation of MeRL. To support the deployment and testing of such an implementation, the Silicon Labs C8051f04x microcontroller family was selected and the necessary driver support for this targeted processing platform was also included.

The embedded RTOS selected for the base of IDEAnix, as justified in section 5.1, was microC/OS-II. As the intended processing core was the 8051 integrated into the Silicon Labs C8051f04x microcontroller family, a pre-ported version of microC/OS-II

was utilized. The port utilized in IDEAnix was microC/OS-II version 2.00 originally ported by John X. Liu, and then modified and released by Junmin Zheng targeted for the 8051 processing core and formatted for development in the KEIL C51 V6 IDE(5).

Although utilizing this port for the most part addressed the porting instructions listed in chapter 13 of Jean Labrosse's uC/OS-II book(47), the task support and system functions specific to the targeted hardware had to be implemented. This included inserting the specific pointers (e.g. pointers to the interrupt enable register 'EA') where needed, as well as allocating a system timer to run the operating system clock. For IDEAnix, timer 0 of the Silicon Labs C8051f04x was reserved and implemented as an ISR running at a 10ms time interval. The uC/OS-II OSTickISR() was associated with the interrupt vector for timer 0 and modified to meet C8051f04x specific interrupt handling requirements. By making this associations and setting timer 0 to interrupt in a 10 ms period the operating system tick rate happens at in 10ms set interval, and thus all operating system background operations occur at a 100 Hz frequency. These simple additions and modifications are all included in the uC/OS-II port files (OS_CPU.C, OS_CPU.H, OS_CPU_A.ASM, OS_KCDEF.h) which are documented in APPENDIX: Demonstration UAV Source Code.

It is also of note that during the integration of the ENDURA physical CAN network driver (38) a problem- despite stated compliance (47) - with the implementation of operating system calls within an ISR execution block was discovered. The problem was traced to the default assembly function (OSIntCtxSw found in OS_CPU_A.asm) for handling interrupt context switches. The issue was simply that the default method for making a context switch to an operating system function within an ISR was incorrect. The problem was resolved after careful review of how uC/OS-II handled the context switching during normal task operation. It was seen that the same method was appropriate for making a context switch from an ISR, and thus the assembly for this process was duplicated in the assembly function OSIntCtxSw. The modifications made to the assembly function OSIntCtxSW in file OS_CPU_A.asm which can be seen in APPENDIX: Demonstration UAV Source Code.

7.2 MeRL Implementation

MeRL was designed to be a system level methodology and extensible mechanism for the management of data producer and data consumer dependencies over both a virtual and physical network. As such, the implementation of MeRL – adhering to the requirements and design described in chapter 4- includes a configuration file. The creation of a configuration file was mainly to provide a method of resource management to aid in the deployment of MeRL on various microcontroller and microprocessor platforms. The configuration file provides access to variables that adjust the sizing of the structures and pre-compiler macro variables that are used to manage the code blocks utilized by MeRL. Thus, the deployment of MeRL becomes scalable to accommodate the smallest to the largest design.

A key factor in the implementation and deployment of MeRL for distributed embedded systems is the integration of a physical broadcast based network. The network driver must adhere to the MeRL to Network API as specified in section 4.3.2. As such, the networking bus protocol CAN was selected for this implementation as it is both a broadcast protocol and includes an inherent message identification mechanism (48). Specifically, for the deployment of this implementation of MeRL the CAN driver ENDURA was designed and implemented(38). This specialized CAN driver, targeted for the Silicon Labs integrated CAN module of the C8051f04x, handles direct data delivery via an ISR. To optimize the data delivery directly from the CAN network to MeRL for task delivery a function called load_up_buff (unsigned int id, unsigned long payload) was created. This function was designed to incorporate the operating system calls needed to directly load the queue based input data buffer of MeRL with data. This function was incorporated into the ENDURA MsgReceive() function that is incorporated directly into the CAN interrupt service routine for fast CAN message loading. The implementation of load_up_buff, as well as the complete source code for the implementation of MeRL can be seen in APPENDIX: Demonstration UAV Source Code.

7.3 Modular UAV Design

The IDEA lab at the University of Kentucky has designed several UAV/ UAS platforms (6)(7) (13)(17). During each development cycle a few minor issues and potential improvements were noted. Among these issues included EMI that affected servo motor actuation and control. This issue became apparent in early designs due to the radio placement being near the central servo power lines, and often times induced servo jitter. Another big issue was the time and complexity required for the integration of several independently developed modules such as a central mission controller, serial data radio interface, ATV controller, digital camera actuators, and camera gimbal controllers. The time and frustration of integrating these subsystems led to the solution of incorporating modular design.

The approach of modular design as a solution encompassed not only the MeRL centered firmware code design, but also the physical design of the system. To address EMI issues within the UAV, it was deduced that actuation and control lines should be kept as short as possible. To keep these lines as short as possible it was reasoned that individual controllers could be paired directly beside of each servo. Having multiple controllers placed throughout the UAV required the integration of a communication bus to support the required inter-processor communication with MeRL. Taking into consideration the inherent EMI present in a small form factor UAV, and the requirement of a broadcast based network in support of MeRL- CAN was utilized. CAN inherently is immune to minor EMI affects due to its differential signal wiring. So effectively, by localizing the controllers next to the servos, minimizing the length of the servo control lines, and utilizing an EMI tolerant communication bus- this modular UAV design became virtually immune to the common EMI effects observed over several UAV design implementations.

The controllers designed to be interfaced directly to the servo controllers for this UAV design were called the Tiny Interface Modules (T.I.Ms) (42) (18). The T.I.M hardware design included several servo control line interfaces, as well as the hardware support for CAN, and a RS232 compliant serial interface all controlled by the c8051f041

microcontroller by Silicon Labs. Another key feature was the daughter card support incorporated for wireless communication via a XBEE communication module. This design of the T.I.Ms was intended to be flexible for its inclusion into several active IDEA Lab projects.

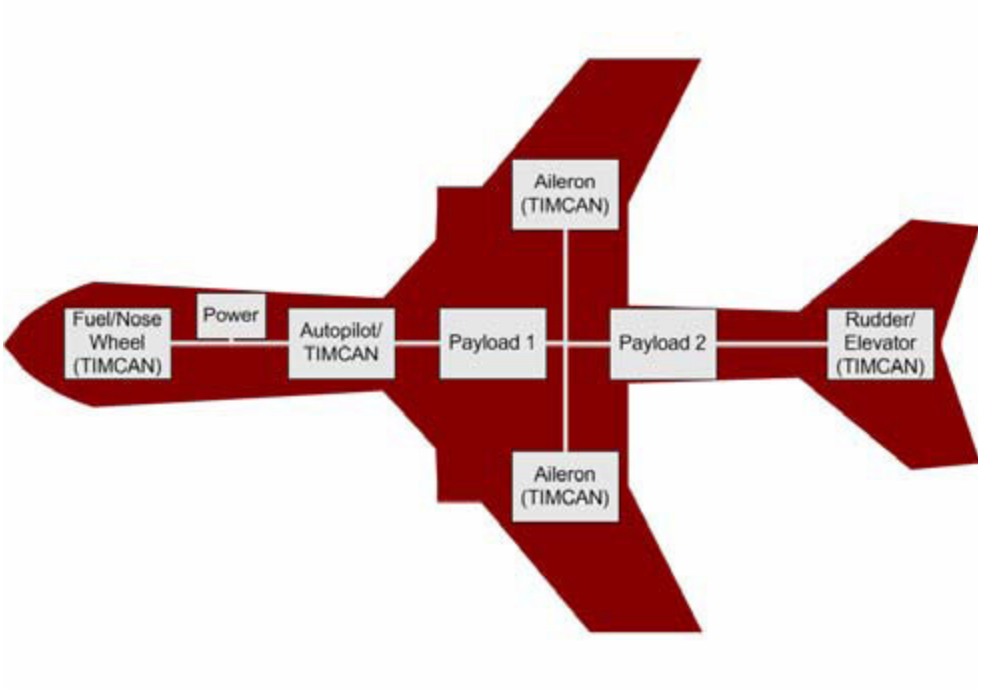


Figure 22: A Simple Modular Distributed UAV Architecture (18)

To test and demonstrate this modular UAV design with MeRL, a simple distributed architecture was implemented. This architecture, as seen in Figure 22, included direct controller interfaces for all the airplane control surface servos via T.I.Ms, commanded by a Cloudcap Piccolo autopilot paired with a T.I.M. and interlinked via a CAN bus. The communication link from the UAV to ground control software was handled by the internal radio interface built into the Piccolo autopilot. The internal operation was straight forward. From a systems perspective, each control surface servo was assigned a specific message id. The data producer, in this case, was the monitoring T.I.M connected to the autopilots servo outputs. This T.I.M's only function was to capture the pulse width associated with each servo output channel, and then package and transmit the captured information onto the CAN bus via MeRL ensuring appropriate

message id assignment. Then each dedicated servo interface T.I.M. - as the data consumer- would actively be waiting for the CAN data packet with the appropriate message id that it had registered for. Once it received its registered id, it would then translate the data packet back into the pulse size and output the signal to the servo. A picture of the actual UAV showing the T.I.M. integration is seen in Figure 23.

Figure 23: Demonstration UAV Showing T.I.M. Placements

This UAV was not only a simple demonstrator for the integration of MeRL and IDEAnix, but was intended to demonstrate the modularity that MeRL can give a system. The integration of additional payloads would be as simple as assigning the appropriate ids and ensuring this information was present on the bus. There is no unnecessary overhead associated with addressing. This is why MeRL is the perfect middleware for handling interprocess and interprocessor communication. With this flexibility it can be seen how an architecture with MeRL is flexible and can be easily adapted for any additional payload. This is why this simple UAV design also demonstrates the feasibility of the novel Ready UAV design as described in section 1.2.4.

7.4 Timing

The timing associated with any system is important. This is true for distributed embedded systems as well. It is well known that research in the field of timing has been ongoing(2)(11)(28)(49) (50). The understanding of how fast a system can respond and whether the response time can be reasonably bound to perform under expected conditions is classified as real-time system analysis. A system- as discussed in relation to the classification of embedded operating systems in section 1.1.4- can itself be classified based upon the timing associated with the response of an event. If system operation is restricted to meet set deadlines and if in failing to do so the system errors; the system timing is said to be hard real-time. However, if the system is bound to operate with set deadlines, but the missing of a deadline, although not desired, is accepted the system timing is said to be soft real-time. Finally, if the system has no imposed timing constraints- the system is classified as a non-real-time system.

Regardless of the semantics associated with the classification of real-time systems, the distinction of a real-time system comes with the constraint that a system response must occur within a set period of time. It is of note that real-time in no way implies an "instantaneous" response, but instead a response that is bound and thus predictable. This distinction is significant for safety critical applications which mandate predictability (2). This infers that real-time safety critical systems may be designed as a distributed system, as long as the linking communication layer latency is predictable by being accounted for within the timing of a system response. Because of this there has been extensive work in the development of communication protocols that support message scheduling through set time intervals such as SAFEbus, SPIDER, and Time-Triggered Protocol (2). Despite the reliability of these protocols, there are occasions that a distributed real-time system lends itself to a more dynamic communication protocol in which message passing becomes event-driven. An event-driven dynamic protocol can improve the response times to a sporadic system event. For this reason, there are several dynamic communication protocols- like CAN- that have been utilized in real-time safety critical applications such as those in the automotive industry.

One of the intended platforms for MeRL is real-time, safety-critical, distributed embedded systems. As such, it is important that the inclusion of MeRL not affect the predictability of a system's response. This means that the timing associated with MeRL's internal operation must be quantifiable. Taking an analytical approach, equations can be formulated for the transmission latency through MeRL, internal message reception latency through MeRL, and the external message reception through MeRL.

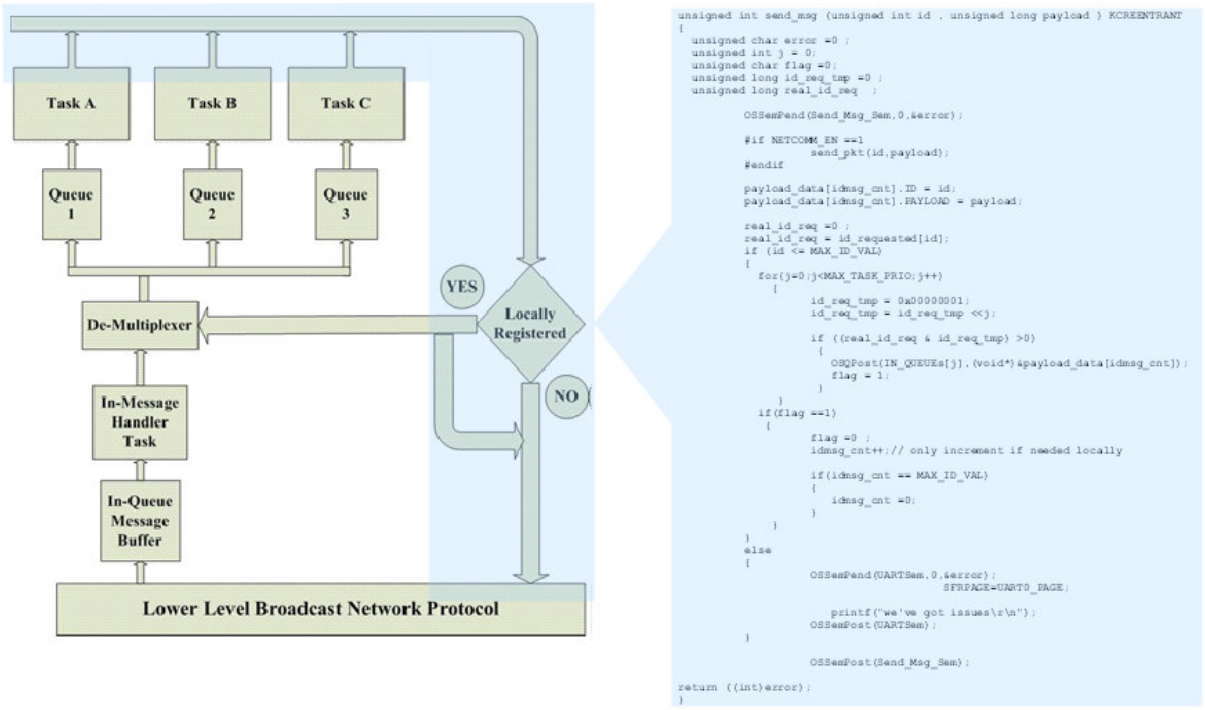


Figure 24: MeRL Block Diagram Highlighted with Message Passing Code

Referencing Figure 24, the transmission latency or the duration of time it takes for a message to be sent from a task to being electrically on the utilized physical network can be expressed as:

Transmission Latency = (Network Driver Tx Latency) + (Physical Layer Network Latency)

Likewise, the external reception latency or the duration of time it takes for a message to get to the intended receiving task via the physical network can be represented as:

External Reception Latency = (Node1 Transmission Latency) + (Node 2 Physical Layer Network Latency) + (Node2 Network Driver Rx Latency) + (Node2 Incoming Message Buffer Queue Load Latency) + (Node2 Message Handling Task Latency)

Finally, the internal message reception or the duration of time it takes for a message to be sent from one task (message producer) to another task (message consumer) implemented on the same processor varies. It varies depending on which send function (send_msg () or send_local() as described in Section 4.3.1) is used. The duration for each can be represented as follows:

send_msg (id , payload):

Internal Message Reception Latency = (Network Driver Tx Latency) + (Incoming Message Buffer Queue Load Latency) + (Message Handling Task Latency)

send_local (id , payload):

Internal Message Reception Latency = (Incoming Message Buffer Queue Load Latency) + (Message Handling Task Latency)

So as can be seen, the timing associated with data routing through MeRL can be quantified. However, the bounding and predictability of MeRL routing of data is dependent on Operating System mechanisms utilized within MeRL such as the queue and task latency. This implies that MeRL's predictability is subject to the predictability of the structures it is implemented with. Thus, if MeRL is implemented with structures within a certified real-time operating system, then that implementation of MeRL can be certified for use in a real-time system. Likewise, if the inter-processor network communication bus operating beneath MeRL- if present- has been certified as real-time, then the inter-processor communication utilizing MeRL is acceptable for use in a physically distributed real-time system.

For these reasons, it can be seen that with the addition of MeRL to a distributed embedded system the communication infrastructure remains quantifiable and predictable. Thus MeRL can be utilized in both hard and soft real-time safety critical applications.

7.5 Experimental Results

The implementation of MeRL for the simple UAV architecture as described in section 7.3 was completed using uC/OS-II wrapped in IDEAnix. The physical network for inter-processor communication was selected to be CAN. As described in section 7.4, the use of MeRL for distributed, safety-critical, real-time applications is governed by its implementation. This is why the selection of the RTOS uC/OS-II was critical, because its implementation had already been certified DO-178B(39). Having the DO-178B certification means it is certified for use in safety-critical applications. Thus, implying an implementation of MeRL with this certified OS would therefore be compliant for use in such systems, as well. A similar thought process also went into the selection of the physical network incorporated beneath MeRL. This physical layer was selected to be CAN. This was done for many reasons including the fact that CAN is a dynamic broadcast protocol which has been utilized by several real world, real-time, safety critical applications. Thus, in selecting the implementation of MeRL to be done with a real-time operating system certified for use in safety critical applications, and integrated with a broadcast network that has been utilized in several real-world, real-time, safety critical applications- this implementation of MeRL is inherently acceptable to be utilized in real-time, safety-critical applications.

To characterize this implementation, experiments were run utilizing the deployment of MeRL and IDEAnix in the distributed UAV design described in section 7.3. The experiments were performed using GPIO lines of the Silicon Labs c8051f041 on T.I.Ms to capture timing attributes. The first experiment was done to capture the minimum delivery time for messages sent via MeRL between two nodes on this physical network. In this experiment, the producer sent a single payload packet through MeRL over the CAN network to a producer node that was registered to receive the message id.

The transmission time was captured by setting a GPIO line low prior to calling the send_msg () function within the experiment transmission task, and on the receiving node a GPIO line was set high after the experiment receiving task had received data. This duration from data transmission to reception was found to be roughly 640 microseconds as shown in the oscilloscope screen capture seen in Figure 25.

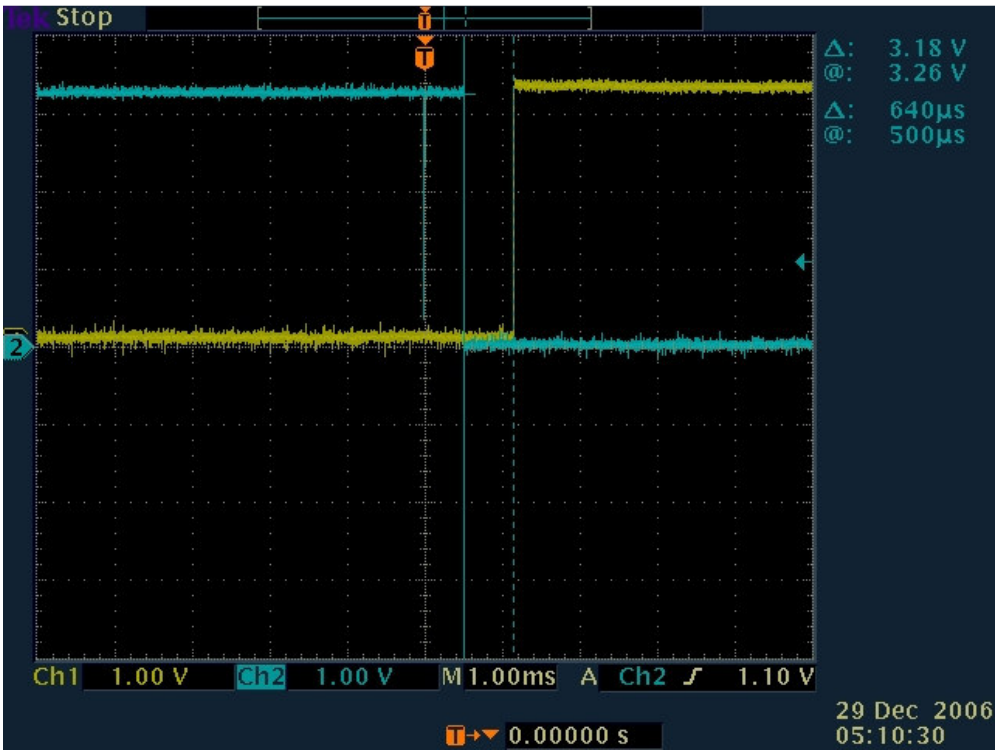


Figure 25: Transmission of a Data Packet via MeRL to a Receiving Node

The second experiment was designed to show the minimum processing time it takes for a message to be transmitted through MeRL on the same processor. Once again to capture this time a GPIO line was used. The GPIO line was set high before the send_msg() function was called and then set low once the send_msg() function returned. The minimum processing time a message transmission takes was found to be 497.2 microseconds. This result is shown in the oscilloscope screen capture seen in Figure 26.

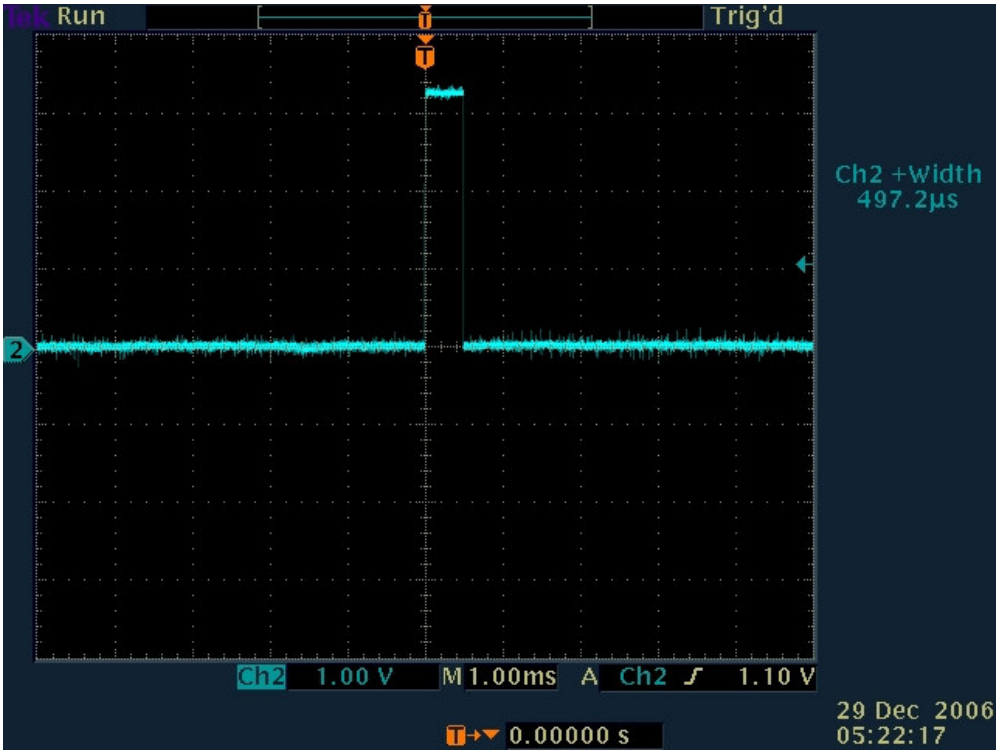


Figure 26: Time Taken to Execute the send_msg ( ) Function in MeRL

It is important to note that the timing associated with these experiments represent a minimum time for this specific implementation and deployment of MeRL. It can be logically determined that generically associating timing performance of MeRL is impossible, as timing is dependent on so many variables such as the number of nodes in the network, the number of internal tasks registered for data, the speed of the network, the operating processor's clock speed, etc. Beyond those specific variables, MeRL is only a mechanism or method of data routing, and its performance- in truth- is subject to its own implementation such as the operating characteristics of the RTOS used. This is why the results of the experiments performed serve as only benchmarks of this specific implementation.

8 CONCLUSIONS

The purpose of this thesis work was to develop an extensible mechanism to simplify the system-level management of data dependencies and subsequently data routing within a reliable distributed embedded system. The detailed descriptions and background information in support of the fulfillment of this purpose are present throughout this thesis document and result in the development of a novel messaging routing layer- MeRL. Chapter 1 described the background and motivation that resulted in the need for the development of MeRL. Chapter 2 surveyed the different aspects of fault tolerant embedded systems, as it is a requirement that MeRL support the communication needs for such systems. Chapter 3 highlighted information about the type of network protocol needed to support fault tolerant applications such as the reliable inter-processor communication MeRL is designed for. Chapter 4 discusses the requirements imposed on MeRL and the resulting high level design solution. Chapter 5 discusses the operating system selection required for the development and integration of MeRL, as well as the general IDEAnix infrastructure that was created to support the implementation of MeRL discussed in this thesis. Chapter 6 reviews the system lifecycle and describes how aspects of the system lifecycle benefit from the inclusion of MeRL. Finally, chapter 7 describes the implementation of MeRL and IDEAnix targeted for a simple UAV application, and reviews this specific implementation's timing results.

Within this thesis document requirements were imposed to create a mechanism that encompassed a robust approach to the design, creation, and operation of any incorporating embedded system. The result was the development of MeRL- a data message routing layer. One of the basic goals set for MeRL was that it must provide a means of data tracking within the embedded system. This was accomplished by MeRL mandating the association of all data within the system to have a data identifier. Another set of goals for MeRL was to reduce design complexity, ease implementation, and simplify integration. These goals were all accomplished with the way that MeRL decouples data producer and consumer direct dependencies. Instead of following a traditional location based methodology MeRL implements an id based system in which

all nodes have visibility to all data within the network. In this MeRL environment, producer and consumer addresses do not have to be tracked which simplifies the design and implementation. Since all consumers and producers are location independent, integration becomes as simple as connecting individual nodes to the properly selected communications bus. The last major goal for MeRL was that it had to support reliable design. It was shown that MeRL supports reliable design through both its clear requirements for reliable network integration and its inherent reconfiguration support mandated by such fault tolerant architectures as ARDEA.

With this summary it is clear that the intent and goals of this thesis work are fulfilled.

APPENDIX: Demonstration UAV Source Code

uC/OS-II Source Code:

All uC/OS-II RTOS source code- with the exception of OS_CPU_A.ASM- used in the development of the Demonstration UAV is located via download from:

http://micrium.com/page/downloads/ports/intel

Confirmed as of March 27, 2010

The source code files found in the following portions of this Appendix are files which have been customized for the target platform or are the result of independent development effort, as referenced throughout the text, for the purpose of this thesis work.

The source code is sectioned by the incorporating file as it was included into a project for the demonstration of this thesis work.

uC/OS-II Port Files:

OS_CPU.H:

```
/*
*********************************************************************************************************
*********************
* uC/OS-II
* The Real-Time Kernel
*
* (c) Copyright 1992-1998, Jean J. Labrosse, Plantation, FL
* All Rights Reserved
*
* Keil C51 6.20c specific code
* LARGE MEMORY MODEL
*
* File : OS_CPU.H
* By : Jean J. Labrosse
*
* Ported by: John X. Liu (johnxliu@163.com)
* Now it used by Junmin Zheng (zhengjunm@263.net)for porting uC/OS-II 2.00 to 8051
* Target platform: Keil C51 V6
*********************************************************************************************************
*********************
*/

#ifndef __OS_CPU_H
#define __OS_CPU_H

#ifdef OS_CPU_GLOBALS
#define OS_CPU_EXT
#else
#define OS_CPU_EXT extern
```

```
#endif

/*
*********************************************************************************************************
*********************
* DATA TYPES
* (Compiler Specific)
*********************************************************************************************************
*********************
*/

typedef unsigned char BOOLEAN;

typedef unsigned char INT8U;
/* Unsigned 8 bit quantity */
typedef signed char INT8S;
/* Signed 8 bit quantity */
typedef unsigned int INT16U;
/* Unsigned 16 bit quantity */
typedef signed int INT16S;
/* Signed 16 bit quantity */
typedef unsigned long INT32U;
/* Unsigned 32 bit quantity */
typedef signed long INT32S;
/* Signed 32 bit quantity */
typedef float FP32;
/* Single precision floating point */

typedef unsigned char OS_STK;
/* Each stack entry is 8-bit wide */
typedef unsigned char OS_CPU_SR;
/* Define size of CPU status register (PSW = 8 bits) */

#define BYTE INT8S /* Define data types for backward compatibility ... */
#define UBYTE INT8U /* ... to uC/OS V1.xx. Not actually needed for ... */
#define WORD INT16S /* ... uC/OS-II. */
#define UWORD INT16U
#define LONG INT32S
#define ULONG INT32U

/*
*********************************************************************************************************
*********************
* Keil C51 on generic 8051-based microcontroller
*********************************************************************************************************
*********************
*/

#define OS_CRITICAL_METHOD 2

#if OS_CRITICAL_METHOD == 1
#define OS_ENTER_CRITICAL() EA=0 /* Disable interrupts */
#define OS_EXIT_CRITICAL() EA=1 /* Enable interrupts */
#endif
```

```
#if OS_CRITICAL_METHOD == 2


/* As an undocumented keyword of keil c. __asm is supported in Keil C v6.20.
. No other means to define assemble language code in a macro, I have to use it here. If your compiler does
not support __asm, use method 1 or 3 then. */


/* A2 AF MOV C, EA*/
/* C2 AF CLR EA */
/* C0 D0 PUSH PSW */


#define OS_ENTER_CRITICAL() __asm DB 0A2H, 0AFH, 0C2H, 0AFH, 0C0H, 0D0H

/* D0 D0 POP PSW */
/* 92 AF MOV EA, C */


#define OS_EXIT_CRITICAL() __asm DB 0D0H, 0D0H, 092H, 0AFH
#endif

#if OS_CRITICAL_METHOD == 3
#define OS_ENTER_CRITICAL() (cpu_sr = EA, EA=0) /* Disable interrupts*/
#define OS_EXIT_CRITICAL() (EA=cpu_sr) /* Enable interrupts */
#endif


#define OS_STK_GROWTH 1 /* Stack grows from HIGH to LOW memory on for large mode */
#define OS_TASK_SW() OSCtxSw()

//#define OS_ISR_PROTO_EXT 1
void OSCtxSw(void) KCREENTRANT;


#endif
```

OS_CPU_A.ASM

```
;
; Ported date: MAY 29, 2002
; By: Junmin Zheng, China, (zhengjunm@263.net)
; Target platform: Keil C51 V6.20, V6.21
; Revision:
; Now uC/OS V2.00 has been supported by this port.
; What I port is very similar to that of 80x86 in form.
; Task switch functions are writen with assemble language.
; The sequence of registers to save to hardware stack now has been re-
arranged to conform to what Keil C does.
; A decision was made that

NAME OS_CPU_A_ASM

;#include "reg52.h"

; ?C_XBP is the simulated external stack pointer in large mode, and I use four
; bytes ?C_XBP to ?C_XBP+3 in this module ,so users can not use them in his functions.

PUBLIC ?C_XBP
PUBLIC _?OSIntCtxSw
PUBLIC Stack
PUBLIC _?OSCtxSw
PUBLIC _?OSStartHighRdy

EXTRN  CODE(_?OSTaskSwHook)
EXTRN  XDATA(OSRunning)
EXTRN  XDATA(OSTCBHighRdy)
EXTRN  XDATA(OSPrioHighRdy)
EXTRN  XDATA(OSPrioCur)
EXTRN  XDATA(OSTCBCur)

; LoadStack MACRO is the assembly code to load the highest priority
; task context
LoadStack MACRO
LOCAL LOOPLS
MOV  DPTR,#OSTCBHighRdy
INC  DPTR
MOVX A,@DPTR
MOV  ?C_XBP+2,A
INC  DPTR
MOVX A,@DPTR
MOV  ?C_XBP+3,A
MOV  DPH,?C_XBP+2
MOV  DPL,?C_XBP+3
INC  DPTR
MOVX A,@DPTR
MOV  ?C_XBP,A
INC  DPTR
MOVX A,@DPTR
MOV  ?C_XBP+1,A
MOV  DPH,?C_XBP
MOV  DPL,?C_XBP+1
```

```
MOVX A,@DPTR
MOV  SP,A
INC  DPTR
MOV  R0,SP
LOOPLS:
MOVX A,@DPTR
MOV  @R0,A
INC  DPTR
DEC  R0
CJNE R0,#Stack-1,LOOPLS
MOV  ?C_XBP,DPH
MOV  ?C_XBP+1,DPL
ENDM

; SaveStack MACRO is the assembly code to save the current priority
; task context
SaveStack MACRO
LOCAL LOOPSS1,LOOPSS2
MOV DPH,?C_XBP
MOV DPL,?C_XBP+1
MOV A, SP
CLR C
SUBB A, #Stack-1
MOV R0, A
INC A
CLR C
XCH A, DPL
SUBB  A, DPL
JNC LOOPSS1
DEC DPH
LOOPSS1:
MOV DPL,A
MOV ?C_XBP,  DPH
MOV ?C_XBP+1, DPL
MOV A,SP
MOVX @DPTR, A
LOOPSS2:
INC DPTR
POP ACC
MOVX @DPTR, A
DJNZ R0, LOOPSS2
MOV  DPTR,#OSTCBCur
INC  DPTR
MOVX A,@DPTR
MOV  ?C_XBP+2,A
INC  DPTR
MOVX A,@DPTR
MOV  ?C_XBP+3,A
MOV  DPH,?C_XBP+2
MOV  DPL,?C_XBP+3
INC  DPTR
MOV  A,?C_XBP
MOVX @DPTR,A
INC  DPTR
MOV  A,?C_XBP+1
MOVX @DPTR,A
```

```
ENDM

; LoadTCB MACRO is the assembly code to make OSTCBCur point to OSTCBHighRdy
LoadTCB MACRO
MOV  DPTR,#OSTCBHighRdy
INC  DPTR
MOVX A,@DPTR
MOV  ?C_XBP+2,A
INC  DPTR
MOVX A,@DPTR
MOV  ?C_XBP+3,A
MOV  DPTR,#OSTCBCur
INC  DPTR
MOV  A,?C_XBP+2
MOVX @DPTR,A
INC  DPTR
MOV  A,?C_XBP+3
MOVX @DPTR,A
ENDM

; LoadTCB MACRO is the assembly code to make OSPrioCur equal to OSPrioHighRdy
LoadPrio MACRO
MOV  DPTR,#OSPrioHighRdy
MOVX A,@DPTR
MOV  DPTR,#OSPrioCur
MOVX @DPTR,A
ENDM

; The PUSHA now emulates the pushing sequence what Keil C does.
PUSHA MACRO
IRP REG, <ACC, B, DPH, DPL, PSW, 0, 1, 2, 3, 4, 5, 6, 7>
PUSH REG
ENDM
ENDM

POPA MACRO
IRP REG, <7, 6, 5, 4, 3, 2, 1, 0, PSW, DPL, DPH, B, ACC>
POP REG
ENDM
ENDM


; Declare the external stack pointer by ourself.
DT?C_XBP SEGMENT DATA
RSEG DT?C_XBP
?C_XBP:
DS 4
; Declare a label 'Stack' in the hardware stack segment so that we know where it begins.
?STACK SEGMENT IDATA
RSEG ?STACK
Stack:

; Task level context switch entry point, which is intended to be called by task gracefully.

PR?OSCtxSw  SEGMENT CODE
RSEG PR?OSCtxSw
```

```
_?OSCtxSw:
PUSHA
PUSH IE
SaveStack
LCALL _?OSTaskSwHook
LoadTCB
LoadPrio
LoadStack
POP  ACC
RLC  A
MOV  EA,C
POPA
RET


; Interrupt level context switch entry point, which is intended to be called by task gracefully.
PR?OSIntCtxSw  SEGMENT CODE
RSEG PR?OSIntCtxSw
_?OSIntCtxSw:
PUSHA
PUSH IE
SaveStack
LCALL _?OSTaskSwHook
LoadTCB
LoadPrio
LoadStack
POP  ACC
RLC  A
MOV  EA,C
POPA
RETI



;The first Task level context switch entry point.
PR?OSStartHighRdy  SEGMENT CODE
RSEG PR?OSStartHighRdy
_?OSStartHighRdy:
LCALL _?OSTaskSwHook

MOV  DPTR,#OSRunning
MOV  A,#01H
MOVX @DPTR,A
LoadStack
POP  ACC
RLC  A
MOV  EA,C
POPA
RET
END
```

OS_CPU_C.C:

```
/*
*********************************************************************************************************
*******************
* uC/OS-II
*  The Real-Time Kernel
*
* (c) Copyright 1992-1998, Jean J. Labrosse, Plantation, FL
* All Rights Reserved
*
*
* 80x86/80x88 Specific code
* LARGE MEMORY MODEL
*
* File : OS_CPU_C.C
* By  : Jean J. Labrosse
*
*
* Ported date: MAY 29, 2002
* By:  Junmin Zheng, China, (zhengjunm@263.net)
* Target platform: Keil C51 V6.20
*
*********************************************************************************************************
*******************
*/

#define OS_CPU_GLOBALS
#include "..\OS_config\includes.h"
#include "..\OS_port\os_kcdef.h"

#include "..\ideanix\ideanix_init.h"

/*
*********************************************************************************************************
*******************
* INITIALIZE A TASK'S STACK
*
* Description: This function is called by either OSTaskCreate() or OSTaskCreateExt() to initialize the
*  stack frame of the task being created. This function is highly processor specific.
*
* Arguments : task  is a pointer to the task code
*
*  pdata is a pointer to a user supplied data area that will be passed to the task
*  when the task first executes.
*
*  ptos  is a pointer to the top of stack. It is assumed that 'ptos' points to
*  a 'free' entry on the task stack. If OS_STK_GROWTH is set to 1 then
*  'ptos' will contain the HIGHEST valid address of the stack. Similarly, if
*  OS_STK_GROWTH is set to 0, the 'ptos' will contains the LOWEST valid address
*  of the stack.
*
*  opt specifies options that can be used to alter the behavior of OSTaskStkInit().
*  (see uCOS_II.H for OS_TASK_OPT_???).
*
* Returns : Always returns the location of the new top-of-stack' once the processor registers have
*  been placed on the stack in the proper order.
```

```
*
* Note(s) : Interrupts are enabled when your task starts executing. You can change this by setting the
*  PSW to 0x0002 instead. In this case, interrupts would be disabled upon task startup. The
*  application code would be responsible for enabling interrupts at the beginning of the task
*  code. You will need to modify OSTaskIdle() and OSTaskStat() so that they enable
*  interrupts. Failure to do this will make your system crash!
*********************************************************************************************
*******************
*/

/* The stack variable points to the start pointer in hardware stack and is defined in OS_CPU_A */
extern idata unsigned char Stack[1];

OS_STK *OSTaskStkInit (void (*task)(void *pd) KCREENTRANT, void * vd, OS_STK *ptos, INT16U o
pt) KCREENTRANT
{
   INT8U * stk;
   opt = opt; /* 'opt' is not used, prevent warning  */
   stk = (INT8U *) ptos; /* Load stack pointer  */



   stk -= sizeof(INT16U);  /* The value should be loaded to PC */
   *(INT16U*)stk = (INT16U) task;  /* next time when this task is running */

   stk -= sizeof(INT16U);  /* The value should be loaded to PC */
   *(INT16U*)stk = (INT16U) task;  /* next time when this task is running */

   /* Following is the registers pushed into hardware stack */
   *--stk = 'A'; /* ACC */
   *--stk = 'B'; /* B  */
   *--stk = 'L'; /* DPL */
   *--stk = 'H'; /* DPH */
   *--stk = PSW; /* PSW */
   *--stk = 0; /* R0 */

   stk -= sizeof(void *);  /* Keil C uses R1,R2,R3 to pass the */
   *(void**)stk = vd;  /* arguments of functions.  */

   *--stk = 4; /* R4 */
   *--stk = 5; /* R5 */
   *--stk = 6; /* R6 */
   *--stk = 7; /* R7 */

   *--stk = 0x80; /* IE, EA is enabled */
   /*
   Next is calculating the hardware stack pointer.
   */
   *--stk = (INT8U) Stack-1 /* Initial value when main was called */
   +1 /* IE */
   +8 /* R0-R7, eight registers was saved  */
   +5 /* PSW, ACC, B, DPH, DPL, five registers */
   +sizeof(INT16U)  /* The PC value to be loaded  */
   +sizeof(INT16U)  /* The PC value to be loaded  */
   ;
```

```
    return ((void *)stk);
}



/* OSTickISR can be writen in c language now, so it is more easy for user to write code for their own */
void OSTickISR(void) KCREENTRANT interrupt 1
{
    #pragma ASM
    PUSH IE
    #pragma ENDASM


    OSIntEnter();
    SFRPAGE=TIMER01_PAGE;

    OSTimeTick();


    // TH0 = 0xb0;
    // TL0 = 0x3f;

    if (SYSCLK/UART1_BAUDRATE/2/256 < 1)
    {
        TH0 = 0xb0;
        TL0 = 0x3f;
        // T1M = 1; SCA1:0 = xx PURE SYSCLK
    }
    else if (SYSCLK/UART1_BAUDRATE/2/256 < 4)
    {
        TH0 = 0x10;
        TL0 = 0xbd;
        // T1M = 0; SCA1:0 = 01 SYSCLK/4
    }
    else if (SYSCLK/UART1_BAUDRATE/2/256 < 12)
    {
        TH0 = 0xb0;
        TL0 = 0x3f;
        // T1M = 0; SCA1:0 = 00 SYSCLK/12
    }
    else
    {
        TH0 = 0xEC;
        TL0 = 0x0F;
        // T1M = 0; SCA1:0 = 10 SYSCLK/48
    }
    OSIntExit();

    #pragma ASM
    POP IE
    #pragma ENDASM
}

/* If you want to write ISRs for your own, just do as OSTickISR() */

/*$PAGE*/
```

```
/*
*********************************************************************************************************
*********************
* OS INITIALIZATION HOOK
*  (BEGINNING)
*
* Description: This function is called by OSInit() at the beginning of OSInit().
*
* Arguments : none
*
* Note(s) : 1) Interrupts should be disabled during this call.
*********************************************************************************************************
*********************
*/
#if OS_CPU_HOOKS_EN > 0 && OS_VERSION > 203
void OSInitHookBegin (void) KCREENTRANT
{
}
#endif

/*
*********************************************************************************************************
*********************
* OS INITIALIZATION HOOK
* (END)
*
* Description: This function is called by OSInit() at the end of OSInit().
*
* Arguments : none
*
* Note(s) : 1) Interrupts should be disabled during this call.
*********************************************************************************************************
*********************
*/
#if OS_CPU_HOOKS_EN > 0 && OS_VERSION > 203
void OSInitHookEnd (void) KCREENTRANT
{
}
#endif

/*$PAGE*/
/*
*********************************************************************************************************
*********************
* TASK CREATION HOOK
*
* Description: This function is called when a task is created.
*
* Arguments : ptcb  is a pointer to the task control block of the task being created.
*
* Note(s) : 1) Interrupts are disabled during this call.
*********************************************************************************************************
*********************
```

```
*/
#if OS_CPU_HOOKS_EN > 0
void OSTaskCreateHook (OS_TCB *ptcb) KCREENTRANT
{
   ptcb = ptcb; /* Prevent compiler warning  */
}
#endif


/*
*********************************************************************************************************
* TASK DELETION HOOK
*
* Description: This function is called when a task is deleted.
*
* Arguments : ptcb  is a pointer to the task control block of the task being deleted.
*
* Note(s) : 1) Interrupts are disabled during this call.
*********************************************************************************************************
*/
#if OS_CPU_HOOKS_EN > 0
void OSTaskDelHook (OS_TCB *ptcb) KCREENTRANT
{
   ptcb = ptcb; /* Prevent compiler warning  */
}
#endif

/*
*********************************************************************************************************
*  IDLE TASK HOOK
*
* Description: This function is called by the idle task. This hook has been added to allow you to do
*  such things as STOP the CPU to conserve power.
*
* Arguments : none
*
* Note(s) : 1) Interrupts are enabled during this call.
*********************************************************************************************************
*/
#if OS_CPU_HOOKS_EN > 0 && OS_VERSION >= 251
void OSTaskIdleHook (void) KCREENTRANT
{
}
#endif

/*
*********************************************************************************************************
* STATISTIC TASK HOOK
*
* Description: This function is called every second by uC/OS-II's statistics task. This allows your
*  application to add functionality to the statistics task.
```

```
*
* Arguments : none
*********************************************************************************************************
*/

#if OS_CPU_HOOKS_EN > 0
void OSTaskStatHook (void) KCREENTRANT
{
}
#endif

/*$PAGE*/
/*

/*
*********************************************************************************************************
* TASK SWITCH HOOK
*
* Description: This function is called when a task switch is performed. This allows you to perform other
*  operations during a context switch.
*
* Arguments : none
*
* Note(s) : 1) Interrupts are disabled during this call.
*  2) It is assumed that the global pointer 'OSTCBHighRdy' points to the TCB of the task that
* will be 'switched in' (i.e. the highest priority task) and, 'OSTCBCur' points to the
* task being switched out (i.e. the preempted task).
*********************************************************************************************************
*/
#if OS_CPU_HOOKS_EN > 0
void OSTaskSwHook (void) KCREENTRANT
{
}
#endif

/*
*********************************************************************************************************
* OSTCBInit() HOOK
*
* Description: This function is called by OS_TCBInit() after setting up most of the TCB.
*
* Arguments : ptcb is a pointer to the TCB of the task being created.
*
* Note(s) : 1) Interrupts may or may not be ENABLED during this call.
*********************************************************************************************************
*/
#if OS_CPU_HOOKS_EN > 0 && OS_VERSION > 203
void OSTCBInitHook (OS_TCB *ptcb) KCREENTRANT
{
   ptcb = ptcb; /* Prevent Compiler warning */
}
```

```
#endif


/*
*********************************************************************************************************
*******************
* TICK HOOK
*
* Description: This function is called every tick.
*
* Arguments : none
*
* Note(s) : 1) Interrupts may or may not be ENABLED during this call.
*********************************************************************************************************
*******************
*/
#if OS_CPU_HOOKS_EN > 0
void OSTimeTickHook (void) KCREENTRANT
{
   char SFRPAGE_SAVE = SFRPAGE; // Save Current SFR page
   SFRPAGE = CONFIG_PAGE;  // set SFR page
   SFRPAGE = SFRPAGE_SAVE;  // Restore SFR page
}
#endif
```

OS_KCDEF.H

```
/*
*********************************************************************
* uC/OS-II port for Keil C
* By John X. Liu (johnxliu@163.com)
* Now it used by Junmin Zheng (zhengjunm@263.net)for porting uC/OS-II 2.00 to 8051
*********************************************************************
*/

/*
This file is used to resume the definition of the keil c's keywords
which were used as identifiers in uC/OS source files. If your programs need
use this keywords, you must include this head file in.
*/

#ifndef __OS_KCDEF_H
#define __OS_KCDEF_H

#undef data
#undef pdata
#undef xdata
#undef idata
#endif
```

uC/OS-II Configuration Files:

INCLUDES.h

```
/*
*********************************************************************************************************
*                                              uC/OS-II
*                                        The Real-Time Kernel
*
*                  (c) Copyright 1992-1998, Jean J. Labrosse, Plantation, FL
*                                           All Rights Reserved
*
*                                           MASTER INCLUDE FILE
*********************************************************************************************************
*/

/*
*********************************************************************************************************
* Keil C port.
* By:  Junmin Zheng (zhengjunm@263.net)
*********************************************************************************************************
*/

#ifndef __INCLUDES__
#define __INCLUDES__


#include "..\ideanix\c8051f040.h"

#include   <stdio.h>
#include   <string.h>
#include   <ctype.h>
#include   <stdlib.h>
#include   <setjmp.h>

/* These macros are intended to be used to re-declare the uC/OS-II functions to be reentrant */
#define TASK_REENTRANT                  large reentrant
#define KCREENTRANT                     large reentrant

#include   "..\OS_port\os_cpu.h"
#include   "..\OS_config\os_cfg.h"

/* These Keil keywords are used in uC/OS-II source files. So the next macro definitions are needed
*/
#define data       ucos51_data
#define pdata      ucos51_pdata
#define xdata      ucos51_xdata

#include   "..\OS_source\ucos_ii.h"

#endif
```

OS_CFG.h

```
/*
*********************************************************************************************************
*********************
* uC/OS-II
* The Real-Time Kernel
*
* (c) Copyright 1992-2001, Jean J. Labrosse, Weston, FL
* All Rights Reserved
*
*  uC/OS-II Configuration File for V2.00
*
* File : OS_CFG.H
* By  : Jean J. Labrosse
*********************************************************************************************************
*********************
*/

/*
*********************************************************************************************************
*********************
*  uC/OS-II CONFIGURATION
*********************************************************************************************************
*********************
*/

#define OS_MAX_EVENTS  40 /* Max. number of event control blocks in your application ... */
/* ... MUST be >=2 */
#define OS_MAX_FLAGS  2 /* Max. number of Event Flag Groups in your application ... */
/* ... MUST be >= 2 */
#define OS_MAX_MEM_PART 2 /* Max. number of memory partitions ...  */
/* ... MUST be >=2 */
#define OS_MAX_QS 30 /* Max. number of queue control blocks in your application ... */
/* ... MUST be >=2 */
#define OS_MAX_TASKS 18 /* Max. number of tasks in your application ... */
/* ... MUST be >= 2  */

#define OS_LOWEST_PRIO 24//22  /* Defines the lowest priority that can be assigned ... */
/* ... MUST NEVER be higher than 63!  */

#define OS_TASK_IDLE_STK_SIZE 128 /* Idle task stack size (# of OS_STK wide entries)  */

#define OS_TASK_STAT_EN 0 /* Enable (1) or Disable(0) the statistics task */
#define OS_TASK_STAT_STK_SIZE 64 /* Statistics task stack size (# of OS_STK wide entries) */

#define OS_ARG_CHK_EN  1 /* Enable (1) or Disable (0) argument checking  */
#define OS_CPU_HOOKS_EN 1 /* uC/OS-II hooks are found in the processor port files */


#define OS_FLAG_EN 0 /* Enable (1) or Disable (0) code generation for EVENT FLAGS */
#define OS_FLAG_WAIT_CLR_EN  0 /* Include code for Wait on Clear EVENT FLAGS */
#define OS_FLAG_ACCEPT_EN 0 /* Include code for OSFlagAccept() */
#define OS_FLAG_DEL_EN 0 /* Include code for OSFlagDel()  */
#define OS_FLAG_QUERY_EN  0 /* Include code for OSFlagQuery() */
```

```
/* ---------------------- MESSAGE MAILBOXES ---------------------- */
#define OS_MBOX_EN 0//1 /* Enable (1) or Disable (0) code generation for MAILBOXES  */
#define OS_MBOX_ACCEPT_EN 0 /* Include code for OSMboxAccept() */
#define OS_MBOX_DEL_EN 0 /* Include code for OSMboxDel()  */
#define OS_MBOX_POST_EN 0//1 /* Include code for OSMboxPost()  */
#define OS_MBOX_POST_OPT_EN  0 /* Include code for OSMboxPostOpt()  */
#define OS_MBOX_QUERY_EN  0 /* Include code for OSMboxQuery() */


/* ---------------------- MEMORY MANAGEMENT ---------------------- */
#define OS_MEM_EN 0 /* Enable (1) or Disable (0) code generation for MEMORY MANAGER */
#define OS_MEM_QUERY_EN 0 /* Include code for OSMemQuery()  */

/* ---------------- MUTUAL EXCLUSION SEMAPHORES ---------------- */
#define OS_MUTEX_EN 0 /* Enable (1) or Disable (0) code generation for MUTEX  */
#define OS_MUTEX_ACCEPT_EN 0 /* Include code for OSMutexAccept()  */
#define OS_MUTEX_DEL_EN 0 /* Include code for OSMutexDel()  */
#define OS_MUTEX_QUERY_EN 0 /* Include code for OSMutexQuery() */


/* ---------------------- MESSAGE QUEUES ---------------------- */
#define OS_Q_EN 1 /* Enable (1) or Disable (0) code generation for QUEUES */
#define OS_Q_ACCEPT_EN 1 /* Include code for OSQAccept()  */
#define OS_Q_DEL_EN 0 /* Include code for OSQDel() */
#define OS_Q_FLUSH_EN  0 /* Include code for OSQFlush() */
#define OS_Q_POST_EN  1 /* Include code for OSQPost() */
#define OS_Q_POST_FRONT_EN 0 /* Include code for OSQPostFront() */
#define OS_Q_POST_OPT_EN  0 /* Include code for OSQPostOpt()  */
#define OS_Q_QUERY_EN  1 /* Include code for OSQQuery() */


/* ------------------------ SEMAPHORES ------------------------ */
#define OS_SEM_EN 1 /* Enable (1) or Disable (0) code generation for SEMAPHORES */
#define OS_SEM_ACCEPT_EN  0 /* Include code for OSSemAccept()  */
#define OS_SEM_DEL_EN  0 /* Include code for OSSemDel() */
#define OS_SEM_QUERY_EN 0 /* Include code for OSSemQuery()  */

/* ---------------------- TASK MANAGEMENT ---------------------- */
#define OS_TASK_CHANGE_PRIO_EN 0 /* Include code for OSTaskChangePrio()  */
#define OS_TASK_CREATE_EN 1 /* Include code for OSTaskCreate() */
#define OS_TASK_CREATE_EXT_EN 1 /* Include code for OSTaskCreateExt() */
#define OS_TASK_DEL_EN 0 /* Include code for OSTaskDel()  */
#define OS_TASK_SUSPEND_EN 0 /* Include code for OSTaskSuspend() and OSTaskResume()  */
#define OS_TASK_QUERY_EN  1 /* Include code for OSTaskQuery() */


#define OS_TIME_DLY_HMSM_EN  0 /* Include code for OSTimeDlyHMSM()  */
#define OS_TIME_DLY_RESUME_EN 0 /* Include code for OSTimeDlyResume() */
#define OS_TIME_GET_SET_EN 0 /* Include code for OSTimeGet() and OSTimeSet()  */


/* ---------------------- MISCELLANEOUS ---------------------- */
#define OS_SCHED_LOCK_EN  0 /* Include code for OSSchedLock() and OSSchedUnlock()  */


#define OS_TICKS_PER_SEC 100  /* This is not a variable rate it is actually based upon what the timer
```

controlling the interation of OS_TICKs is set to and it happens to be set at
at rate such that a tick happens ever .010 seconds.
Set the number of ticks in one second */

typedef INT16U OS_FLAGS; /* Date type for event flag bits (8, 16 or 32 bits) */

ENDURA (CAN Network) Files:

CAN_HEADER.h

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// CAN_HEADER.h
//
//Written and Controlled by:  Nithyananda Siva Jeganathan
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

#ifndef _CAN_HEADER_H
#define _CAN_HEADER_H
#include "..\main\OS_main.h"
#include "..\ideanix\ideanix_config.h"
#if NETCOMM_EN == 1
////////////////////////////////////////////////////////////////////////////
// Control Registers
////////////////////////////////////////////////////////////////////////////
#define CANCTRL 0x00 //Control Register
#define CANSTAT 0x01 //Status register
#define ERRCNT  0x02 //Error Counter Register
#define BITREG  0x03 //Bit Timing Register
#define INTREG  0x04 //Interrupt Low Byte Register
#define CANTSTR 0x05 //Test register
#define BRPEXT  0x06 //BRP Extension Register
////////////////////////////////////////////////////////////////////////////
//IF1 Interface Registers
////////////////////////////////////////////////////////////////////////////
#define IF1CMDRQST 0x08 //IF1 Command Rest  Register
#define IF1CMDMSK  0x09 //IF1 Command Mask  Register
#define IF1MSK1 0x0A //IF1 Mask1  Register
#define IF1MSK2 0x0B //IF1 Mask2  Register
#define IF1ARB1 0x0C //IF1 Arbitration 1 Register
#define IF1ARB2 0x0D //IF1 Arbitration 2 Register
#define IF1MSGC 0x0E //IF1 Message Control  Register
#define IF1DATA1 0x0F //IF1 Data A1 Register
#define IF1DATA2 0x10 //IF1 Data A2 Register
#define IF1DATB1 0x11 //IF1 Data B1 Register
#define IF1DATB2 0x12 //IF1 Data B2 Register
////////////////////////////////////////////////////////////////////////////
//IF2 Interface Registers
////////////////////////////////////////////////////////////////////////////
#define IF2CMDRQST 0x20 //IF2 Command Rest  Register
#define IF2CMDMSK  0x21 //IF2 Command Mask  Register
#define IF2MSK1 0x22 //IF2 Mask1  Register
#define IF2MSK2 0x23 //IF2 Mask2  Register
#define IF2ARB1 0x24 //IF2 Arbitration 1 Register
#define IF2ARB2 0x25 //IF2 Arbitration 2 Register
#define IF2MSGC 0x26 //IF2 Message Control  Register
#define IF2DATA1 0x27 //IF2 Data A1 Register
#define IF2DATA2 0x28 //IF2 Data A2 Register
#define IF2DATB1 0x29 //IF2 Data B1 Register
#define IF2DATB2 0x2A //IF2 Data B2 Register
////////////////////////////////////////////////////////////////////////////
//Message Handler Registers
////////////////////////////////////////////////////////////////////////////
```

```
#define TRANSREQ1  0x40 //Transmission Rest1 Register
#define TRANSREQ2  0x41 //Transmission Rest2 Register
#define NEWDAT1 0x48 //New Data 1 Register
#define NEWDAT2 0x49 //New Data 2 Register
#define INTPEND1 0x50 //Interrupt Pending 1  Register
#define INTPEND2 0x51 //Interrupt Pending 2  Register
#define MSGVAL1 0x58 //Message Valid 1  Register
#define MSGVAL2 0x59 //Message Valid 2  Register
////////////////////////////////////////////////////////////////////////////
//UART
////////////////////////////////////////////////////////////////////////////
#define BAUDRATE 9600  // Baud rate of UART in bps
#define SYSCLK  24500000  // Internal oscillator frequency in Hz
#define SAMPLE_RATE 50000  // Sample frequency in Hz
#define INT_DEC  256 // integrate and decimate ratio


sfr16 CAN0DAT = 0xD8;

#define MSG_OBJ_31  0x0F

#define MSG_OBJ_32  0xF0

// Using Message Obj 31 and 32 for sending of packets and rest 1 to 30 for RX
#define MAX_MSG_OBJS_RX 29


#define MAX_PAYLOAD_LENGTH  4

typedef unsigned int UINT16;
typedef unsigned long UINT32;
typedef unsigned char UINT8;
typedef UINT16 CAN_ID_TYPE;
typedef UINT32 PAYLOAD_TYPE;

//CAN SEMAPHORE
extern OS_EVENT *CANSem;


//////////////////////function prototypes/////////////////////////////////////

/*
* CAN main is the entry point for the application to run the CAN
*/
void can_main (void *arg) KCREENTRANT;

/*
* Init_CAN: Initializes the CAN bus and clears the MsgObjects
* and Starts the CAN bus
*/

UINT8 init_network( void );

/*
* Reg_pkt: Registers a particular Message ID for receiving of packets
*/
```

```
UINT8 reg_pkt ( CAN_ID_TYPE can_id );
/*
* Unreg_pkt: Unregisters a particular Message ID from being received
*/
UINT8 unreg_pkt ( CAN_ID_TYPE can_id );

/*
* get_pkt: Returns the last received message for the Message ID
*/

UINT8 get_pkt ( CAN_ID_TYPE *can_id_ptr, PAYLOAD_TYPE *payload_ptr );

/*
* MsgSend: Sends a message on the CAN Bus for a particular ID
*/

UINT8 send_pkt (CAN_ID_TYPE can_id , PAYLOAD_TYPE payload);
/*
* MsgReceive: This function is used by ISR for moving data from MsgObj to Message RAM
*/
void MsgReceive(void) KCREENTRANT;


/*
* Packet_rx: This is the entry point to the function for a new task that will receive the packet
* after the ISR has woken up
*/
void packet_rx(void* arg) KCREENTRANT;

void err_tsk(void *arg) KCREENTRANT;

#endif // NETCOMM_EN
#endif //_CAN_HEADER_H
```

CAN_HEADER_INTERNAL.h

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// CAN_HEADER_INTERNAL.h
//
//Written and Controlled by:  Nithyananda Siva Jeganathan
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
#ifndef __CAN_INTERNAL_H
#define __CAN_INTERNAL_H

//////////////////////////Internal APIs used by CAN //////////
unsigned int GetObjNum(void);

unsigned int Search_ObjNum(unsigned int );

void clear_msg_objects (void);

void start_CAN (void);

void config_IO (void);

void pkt_process(void);

typedef enum
{
   CAN_DRIVER_ERROR = 200,
   CAN_MSGOBJS_FULL = 201 ,
   CAN_MSGID_NOTFOUND = 202,
   CAN_MSGDATA_NOTFOUND = 203,
   CAN_DUPLICATE_ID_REG = 204,
   CAN_UNREG_ID_ERR = 205,
   CAN_DRIVER_NOERROR = 1
}error_no;

#endif
```

CAN_LIB.c

```
//////////////////////////////////////////////////////////////////////
// CAN_LIB.c
//
//Written and Controlled by:  Nithyananda Siva Jeganathan
//////////////////////////////////////////////////////////////////////


#include <stdio.h>
#include "..\ideanix\c8051f040.h"
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include "..\OS_config\includes.h"
#include "..\OS_port\os_kcdef.h"

#include "can_header.h"
#include "can_header_internal.h"
#include "..\ideanix\ideanix_msg_handler.h"
#include "..\ideanix\ideanix_config.h"

#if NETCOMM_EN == 1

OS_EVENT *CANSem;

///////////////Global variables and FLAGS ////////////////
unsigned int status_can;

int ObjArray[32];
unsigned long DataArray[32];
unsigned long ObjNumVar;
unsigned char send_flag=MSG_OBJ_31;

unsigned char msgValidReg[4];
unsigned char newDataReg[4];
unsigned char temp_reg[4]={0,0,0,0};

static unsigned long buffStatus=0;
static unsigned int regStatus=0;
unsigned long MsgStatus=0;
unsigned long tempStatus=0;
unsigned long tempStatus1=0;
unsigned int msgc=0;
unsigned int msg_id=0;
unsigned long msg_data =0;

int dummy=0;

// for global CAN use
unsigned int can_rx_flag =0;
```

```
//////////////////////////////////////////////////////////////////////////
//////////////////////////Application Functions////////////////////////////
//////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////
///////////////// Get Object Number based on Message ID/////////////////////
//////////////////////////////////////////////////////////////////////////

unsigned int GetObjNum()
{

   unsigned int ObjNum=0,n=0;

   for (n=0; n< MAX_MSG_OBJS_RX; n++)
   {
      if (((unsigned long)(ObjNumVar & (((unsigned long)0x1) << n)) >>n) ==0)
      {
         ObjNumVar |= (((unsigned long)0x1)<< n);
         ObjNum = n+1;
         return ObjNum;
      }
   }
   return CAN_MSGOBJS_FULL;
}

//////////////////////////////////////////////////////////////////////////
////////////////////// Search a specific Message Objects ////////////////////////////////
//////////////////////////////////////////////////////////////////////////

unsigned int Search_ObjNum(unsigned int MsgId)
{
   unsigned int i;
   for (i=0;i<MAX_MSG_OBJS_RX;i++)
   {
      if ((((unsigned long)(ObjNumVar & (((unsigned long)1) << i))) >>i) ==1)//checking if the msgObj is
valid or not?
      {
         if(ObjArray[i] == MsgId)
         {
            return i+1;
         }
      }

   }
   return CAN_MSGID_NOTFOUND;
}

//////////////////////////////////////////////////////////////////////////
////////////////////// Clear Message Objects ////////////////////////////////
//////////////////////////////////////////////////////////////////////////
void clear_msg_objects (void)
{
   unsigned int i;
   SFRPAGE = CAN0_PAGE;
   CAN0ADR = IF1CMDMSK; // Point to Command Mask Register 1
   CAN0DATL = 0xFF; // Set direction to WRITE all IF registers to Msg Obj
```

```
   for (i=0;i<32;i++)
   {
      CAN0ADR = IF1CMDRQST; // Write blank (reset) IF registers to each msg obj
      CAN0DATL = (i+1);
   }

}

//////////////////////////////////////////////////////////////////////////
////////////////// Destroy a Recieve object ////////////////////////////////
//////////////////////////////////////////////////////////////////////////

UINT8 unreg_pkt ( CAN_ID_TYPE can_id )
{
   unsigned int ObjNum,ArbId;
   unsigned char i=0;

   ObjNum=Search_ObjNum(can_id);
   if (ObjNum == CAN_MSGID_NOTFOUND)
   {
      return CAN_UNREG_ID_ERR;
   }
   ObjNumVar ^= (((unsigned long)0x1)<<(ObjNum-1));
   ObjArray[ObjNum-1] = 0;
   DataArray[ObjNum-1] = 0;

   SFRPAGE = CAN0_PAGE;
   CAN0ADR = IF1CMDMSK; // Point to Command Mask 1
   CAN0DAT = 0x00B8; // Set to WRITE, and clear MsgObject

   CAN0ADR = IF1ARB1; // Point to arbitration1 register
   CAN0DAT = 0x0000; // Set arbitration1 ID to "0"
   ArbId = 0x0000; // Set the lower register to 0 & MsgVal to 0
   CAN0DAT = ArbId; // Arb2 high byte:Clear MsgVal bit

   CAN0DAT = 0x0000; // Msg Cntrl: is cleared
   CAN0ADR = IF1CMDRQST; // Point to Command Request reg.
   CAN0DATL = ObjNum; // Move the Object Numbe

   return (UINT8)CAN_DRIVER_NOERROR;
}

//////////////////////////////////////////////////////////////////////////
////////////////////////// Get Message Routines////////////////////////////////
//////////////////////////////////////////////////////////////////////////

UINT8 get_pkt ( CAN_ID_TYPE *can_id_ptr, PAYLOAD_TYPE *payload_ptr)
{
   unsigned long value =0;
   unsigned char i=0;

   value = ObjNumVar;
   msgc =0;

   if (value == 0)
   {
```

```
      *payload_ptr = 0; // Load it into the pointer
      *can_id_ptr = 0; // populating the ID

      return (UINT8)0;
   }

   i=0; // Just to reinforce!
   while( i < MAX_MSG_OBJS_RX)
   {
      if( ((MsgStatus & ((UINT32)0x1<<i)) >>i) !=0)
      {
         *payload_ptr= DataArray[i]; // Get the payload from the buffer
         *can_id_ptr = ObjArray[i]; // populating the ID
         DataArray[i] = 0; // Clearing the data!
         MsgStatus ^= ((UINT32)0x1<<i);
         return (UINT8)1;
      }
      i++;
   }


   *payload_ptr = 0; // Get the payload from the buffer
   *can_id_ptr = 0; // populating the ID

   return (UINT8)0;
}

//////////////////////////////////////////////////////////////////////////////
////////////////// Create a Recieve object ///////////////////////////////
//////////////////////////////////////////////////////////////////////////////

UINT8 reg_pkt ( CAN_ID_TYPE can_id )
{
   unsigned int ObjNum=0,MskId=0,i=0;


   for(i=0;i<MAX_MSG_OBJS_RX;i++)
   {
      if(ObjArray[i] == can_id)
      {
         return (UINT8)CAN_DUPLICATE_ID_REG;
      }
   }
   ObjNum = GetObjNum();

   if(ObjNum == CAN_MSGOBJS_FULL)
   {
      return (UINT8)ObjNum;
   }
   else
   {
      SFRPAGE = CAN0_PAGE;
      CAN0ADR = IF1CMDMSK; // Point to Command Mask 1
      CAN0DAT = 0x00B8; // Set to WRITE, and alter all Msg Obj except ID MASK

      CAN0ADR = IF1ARB1; // Point to arbitration1 register
```

```
      CAN0DAT = 0x0000; // Set arbitration1 ID to "0"
      /*
      Here we are adding the message ID for arbitration and also setting msgvalid bit to 1
      */
      MskId = ((unsigned int)0x8000|(can_id<<2));
      CAN0DAT = MskId ; // Arb2 high byte:Set MsgVal bit, no extended ID,

      CAN0ADR = IF1MSGC;
      CAN0DAT = 0x0480; // Set UMask as Zero, RXIE enabled

      CAN0ADR = IF1CMDRQST; // Point to Command Request reg.
      CAN0DATL = ObjNum; // Select Msg Obj passed into function parameter list
   }  // --initiates write to Msg Obj

   ObjArray[(ObjNum-1)] = can_id; // Storing info on which msgObj has which msgId configured!

   return (UINT8)CAN_DRIVER_NOERROR;
}


////////////////////////////////////////////////////////////////////////////////
////////////////// Send a message over the CAN bus//////////////////////////////
////////////////////////////////////////////////////////////////////////////////

UINT8 send_pkt (CAN_ID_TYPE can_id , PAYLOAD_TYPE payload)
{
   long Ctrl_Reg;
   char n=1;
   unsigned char i=0;
   unsigned char ObjNum=0;
   unsigned char dtosend[8];
   unsigned long longdata;

   unsigned long MsgLen = MAX_PAYLOAD_LENGTH;
   longdata = payload;

   #if 0
   dtosend[0]=(longdata & 0xff);
   dtosend[1]=((longdata & 0xff00)>> 8);
   dtosend[2]=((longdata & 0xff0000) >> 16);
   dtosend[3]=((longdata & 0xff000000)>> 24);
   #endif

   if(send_flag == MSG_OBJ_31)
   {
      ObjNum = 0x1F;
      send_flag = MSG_OBJ_32;
   }
   else
   {
      ObjNum = 0x20;
      send_flag = MSG_OBJ_31;
   }

   for(i=0;i<MsgLen;i++)
   {
```

```
      dtosend[i] = ((longdata & ((unsigned long)0xff << i*8)) >>(i*8));
   }

   SFRPAGE = CAN0_PAGE;
   CAN0ADR = IF2CMDMSK; // Point to Command Mask2
   CAN0DAT = 0x00B2; // Set to WRITE, & alter all Msg Obj except ID MASK bits

   CAN0ADR = IF2ARB1; // Point to arbitration2 register
   CAN0DAT = 0x0000; // Autoincrement to Arb2 high byte:
   can_id = can_id << 2; // place destination in arbitration field
   can_id = can_id | 0xA000;
   CAN0DAT = can_id;

   Ctrl_Reg = 0x0080; // Msg Cntrl: remote frame function not enabled
   Ctrl_Reg = Ctrl_Reg | MsgLen; // set DLC (Data Length Code)
   CAN0DAT = Ctrl_Reg;

   CAN0ADR = IF2CMDRQST; // Point to Command Request reg.
   CAN0DAT = ObjNum; // Select Msg Obj passed into function parameter list

   SFRPAGE = CAN0_PAGE; // IF1 already set up for TX
   CAN0ADR = IF2CMDMSK; // Point to Command Mask 2
   CAN0DAT = 0x0087; // Config to WRITE to CAN RAM, write data bytes,set TXrqst/NewDat, Clr IntP
nd

   CAN0ADR = IF2DATA1;  // Point to 1st byte of Data Field
   for (n=0 ; n < MsgLen ; n=n+2)
   {
      CAN0DATH = dtosend[n+1];
      CAN0DATL = dtosend[n]; //DATA to send on CAN
   }
   CAN0ADR = IF2CMDRQST; // Point to Command Request Reg.
   CAN0DATL = ObjNum; // Move new data for TX to ObjNum

   return (UINT8) CAN_DRIVER_NOERROR;
}

//////////////////////////////////////////////////////////////////////////////
////////////////////////// START CAN ////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////

void start_CAN (void)
{

   // Enable CAN interrupts in CIP-51
   EIE2 |= 0x20;
   SFRPAGE = CAN0_PAGE;
   CAN0CN |= 0x41;  // Configuration Change Enable CCE and INIT
   CAN0ADR = BITREG  ; // Point to Bit Timing register
   CAN0DAT = 0x6FC0; // see above

   CAN0ADR = IF1CMDMSK; // Point to Command Mask 1
   CAN0DATL = 0x1F;  // Config for RX : READ CAN RAM, read data bytes,

   // RX-IF2 operation may interrupt TX-IF1 operation
   CAN0ADR = IF2CMDMSK; // Point to Command Mask 2
```

```
   CAN0DAT = 0x0087; // Config for TX : WRITE to CAN RAM, write data bytes,
   // set TXrqst/NewDat, clr IntPnd
   CAN0CN |= 0x06;  // Global Int. Enable IE and SIE
   CAN0CN &= ~0x41;  // Clear CCE and INIT bits, starts CAN state machine
}


//////////////////////////////////////////////////////////////////////////////////////
////////////////////////// Configure the IO ////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////////

void config_IO (void)
{

   SFRPAGE = CONFIG_PAGE; //Port SFR's on Configuration page
   XBR3 = 0x80; // Configure CAN TX pin (CTX) as push-pull digital output
   XBR2 |= 0x40; // Enable Crossbar/low ports

}

//////////////////////////////////////////////////////////////////////////////
////////////////////////INIT CAN ////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////

UINT8 init_network(void)
{
   unsigned char i =0;
   config_IO(); //configure Port I/O


   // Clear CAN RAM
   clear_msg_objects();

   //Function call to start CAN
   start_CAN();
   ObjNumVar=0x00000000;

   for(i=0;i<32;i++)
   {
      ObjArray[i]=0;
      DataArray[i]=0;
   }

   return (UINT8) CAN_DRIVER_NOERROR;
}

void MsgReceive(void) KCREENTRANT
{
   unsigned int MsgNum=0,ArbId=0;
   unsigned char MsgData[4],i=0,j=0;
   unsigned char tempObj=0;
   unsigned char newDataReg[4];
   unsigned int temp=0,Msglen=0,n=0;
   unsigned long recvdata=0,mdata=0;
   unsigned char err1=0;
```

```
SFRPAGE = CAN0_PAGE;
CAN0ADR = NEWDAT1;
// Read NewData registers that indicate new messages after acceptance filtering
newDataReg[0] = CAN0DATL;
newDataReg[1] = CAN0DATH;

SFRPAGE = CAN0_PAGE;
CAN0ADR = NEWDAT2;
newDataReg[2] = CAN0DATL;
newDataReg[3] = CAN0DATH;

for(i=0;i<4;i++)
{
   n=0;
   tempObj = newDataReg[i];
   while ( ( ( tempObj & ((UINT8)0xFF)) != (UINT8)0x0) && (n<8) ) //temp here
   {

      if ((tempObj & (UINT8)0x1) == 0x1) // temp here
      {
         MsgNum = (UINT8)((i*8)+n+1);

         SFRPAGE = CAN0_PAGE;
         CAN0ADR = IF1CMDMSK;
         CAN0DAT = 0x007F;

         SFRPAGE = CAN0_PAGE;
         CAN0ADR = IF1CMDRQST;
         CAN0DAT = MsgNum;

         SFRPAGE = CAN0_PAGE;
         CAN0ADR = IF1MSGC; // Point to message control register
         Msglen = CAN0DATL; // read DLC (data length code) of recieved message
         Msglen = Msglen & 0x0f; // making sure only the 4 bits of DLC are saved

         SFRPAGE = CAN0_PAGE;
         CAN0ADR = IF1ARB2; // Point to arbitration2 register
         ArbId = CAN0DAT;  // Get the arbitration information or MsgID
         ArbId = ((UINT16)ArbId >>2) &0x7FF;  // Right shift to get the correct info

         CAN0ADR = IF1DATA1; // Point to 1st byte of Data Field
         for (j=0;j<2; j++) ///// loop 'Length' times
         {
            temp = CAN0DAT;  //copying data from message object
            mdata |= (((unsigned long)temp) << (j*16));
         }

         DataArray[(MsgNum)-1] = mdata;
         ObjArray[(MsgNum)-1] = ArbId;
         msg_id = ArbId;
         msg_data = mdata;
         load_up_buff(ArbId,mdata);

      }//if

      n =n+1;
```

```
            tempObj =(UINT8) (newDataReg[i] >>(UINT8)n);

        }//while

    } //for
}


/*
* Packet_rx: This is the entry point to the function for a new task that will receive the packet
* after the ISR has woken up
*/

void packet_rx(void* arg) KCREENTRANT
{
    unsigned char error=0;
    unsigned char err1 =0;
    unsigned char retval =0;
    unsigned char i=0;

    for (i=1;i<MAX_MSG_OBJS_RX;i++)
    {
        err1 = reg_pkt(i);
    }


    CANSem = OSSemCreate(0);
    SFRPAGE = CAN0_PAGE;
    while(1)
    {
        OSSemPend(CANSem,0,&error);
        MsgReceive();

        SFRPAGE = UART0_PAGE;

        SFRPAGE = CAN0_PAGE;
        msg_id =0;
        msg_data=0;
    }
    arg = arg; // This removes the warning and is never executed
}

////////////////////////////////////////////////////////////////////////////
//////////////////// Interrupt Service Routine /////////////////////
////////////////////////////////////////////////////////////////////////////

void CAN_ISR (void) interrupt 19
{
    EA=0;
    OSIntEnter();
    status_can = CAN0STA;

    if ((status_can&0x10) != 0)
    {
        // RxOk is set, interrupt caused by reception
        CAN0STA = (CAN0STA&0xE8);  // Reset RxOk, set LEC to NoChange
```

```
        MsgReceive();
    }
    else if ((status_can&0x08) != 0)
    {  // TxOk is set, interrupt caused by transmision
        CAN0STA = (CAN0STA&0xF0);  // Reset TxOk, set LEC to NoChange
    }
    else if (((status_can&0x0007) != 0)&&((status_can&0x0007) != 7))
    { // Error interrupt, LEC changed
        /* error handling */
        CAN0STA = CAN0STA|0x07;  // Set LEC to NoChange
    }
    EA=1;
    OSIntExit();
}
//////////////////////////////////////////////////////////////////////////////

#endif
```

Ideanix/ MeRL Files:

IDEANIX_INIT.h

```
/*****************************************************************************************
*       Filename:       IDEANIX_INIT.h
*       Version:        V1.0
*       Author:         Darren Brown
*                       © Copyright 2006, University of Kentucky
*                       All Rights Reserved
*
*****************************************************************************************
*/

#ifndef ideanix_init_h

#define ideanix_init_h

#define UART0_BAUDRATE 9600
#define UART1_BAUDRATE 9600
#define SYSCLK      24500000        // Internal oscillator frequency in Hz


void init_all(void);
void SYSCLK_Init (void);
void Port_IO_Init(void);
void UART0_init(void);
void UART1_init(void);
void TimerInitiate(void);


unsigned char hdr_config(unsigned char pca_in, unsigned char pca_out, unsigned char gpio_in, unsigned
char gpio_out,unsigned char push_pull  );

void GPIO_init(unsigned char gpio_in,unsigned char gpio_out,unsigned char pca_tot, unsigned
push_pull);//returns first avalible output pin



#endif
```

IDEANIX_INIT.c

```
/*****************************************************************************
*       Filename:       IDEANIX_INIT.c
*       Version:        V1.0
*       Author:         Darren Brown
*                       © Copyright 2006, University of Kentucky
*                       All Rights Reserved
*
*****************************************************************************
*/

#include "../OS_config/INCLUDES.H"
#include "ideanix_config.h"
#include "ideanix_init.h"
#include "stdio.h"
#include "ideanix_task_msg.h"
#include "ideanix_msg_handler.h"
#include "../NETWORK/can_header.h"
#include "ideanix_servo.h"
unsigned char free_port_pin;//returns pin number of first free port  if 8 then it is p2.0

void init_all(void)
{
        WDTCN = 0xde;           // disable watchdog timer
        WDTCN = 0xad;

        Port_IO_Init();
        SYSCLK_Init ();

        TimerInitiate();

        UART0_init();
        UART1_init();


        #if PCA_ENABLE == 1
                PCA_Init ();
        #endif
        #if NETCOMM_EN == 1
                init_network();
        #endif

        #if TASKCOMM_EN == 1
                init_taskqing ();
        #endif
}

void Port_IO_Init(void)
{

        // This assigns the pins as follows
   // P0.0  -  TX0 (UART0), Open-Drain, Digital
   // P0.1  -  RX0 (UART0), Open-Drain, Digital
   // P0.2  -  SCK  (SPI0), Open-Drain, Digital
```

```
    // P0.3  -  MISO (SPI0), Open-Drain, Digital
    // P0.4  -  MOSI (SPI0), Open-Drain, Digital
    // P0.5  -  NSS  (SPI0), Open-Drain, Digital
    // P0.6  -  SDA (SMBus), Open-Drain, Digital
    // P0.7  -  SCL (SMBus), Open-Drain, Digital

    // P1.0  -  TX1 (UART1), Open-Drain, Digital
    // P1.1  -  RX1 (UART1), Open-Drain, Digital

    char SFRPAGE_SAVE = SFRPAGE; // Save Current SFR page
    SFRPAGE   = CONFIG_PAGE;

    XBR0 = 0x37; // set to route PCA CHANNELS OUT TO CROSS BAR
    XBR1 = 0x00; // XBAR1: Initial Reset Value
    XBR2 = 0x44; // XBAR2: Initial Reset Value
    XBR3 = 0x00;    // XBAR3: Initial Reset Value

    SFRPAGE= SFRPAGE_SAVE;
}

void SYSCLK_Init (void)
{
   char SFRPAGE_SAVE = SFRPAGE;       // Save Current SFR page

    SFRPAGE = 0x0F;
            OSCXCN = 0x00;          // EXTERNAL Oscillator Control Register
            CLKSEL = 0x00;          // Oscillator Clock Selector
            OSCICN = 0x83;          // Internal Oscillator Control Register
                                    //SYSCLK derived from Internal Oscillator divided by 1.

   SFRPAGE = SFRPAGE_SAVE;  // Restore SFR page
}



//UART0 is configured using timer2

void UART0_init(void)
{
   char SFRPAGE_SAVE = SFRPAGE;       // Save Current SFR page
   unsigned int tmp =0;
   unsigned char high  =0 ;
   unsigned char low =0;
   unsigned long tmp2=0;

   SFRPAGE=UART0_PAGE;

   TMR2CF = 0x08;  // Timer 2 Configuration

    tmp2 = ((unsigned long)UART0_BAUDRATE)<<4;
    tmp = 65535-(SYSCLK/tmp2);
    high = (tmp >> 8) & 0x00ff;
    low = (tmp & 0x00ff);

    RCAP2L = low;  // Timer 2 Reload Register Low Byte
    RCAP2H = high;
```

```
   TMR2L = 0x00;     // Timer 2 Low Byte
   TMR2H = 0x00;     // Timer 2 High Byte
   TMR2CN = 0x04;  // Timer 2 CONTROL

   SFRPAGE = 0x00;
   SADEN0 = 0x00;    // Serial 0 Slave Address Enable
   SADDR0 = 0x00;    // Serial 0 Slave Address Register
   SSTA0 |= 0x15;       // UART0 Status and Clock Selection Register
   SCON0 |= 0x50;       // Serial Port Control Register
   PCON |= 0x00;        // Power Control Register

  TI0 = 1;

  ES0 = 1; // enable UART0 interrupts

  SFRPAGE = SFRPAGE_SAVE;

}



// UART1 configured using timer 1

void UART1_init(void)
{
  char SFRPAGE_SAVE = SFRPAGE;        // Save Current SFR page

  SFRPAGE = UART1_PAGE;
  SCON1   = 0x10;                  // SCON1: mode 0, 8-bit UART, enable RX

  SFRPAGE = TIMER01_PAGE;


  TMOD   &= ~0xF0;
  TMOD   |=  0x20;                 // TMOD: timer 1, mode 2, 8-bit reload


  if (SYSCLK/UART1_BAUDRATE/2/256 < 1)
 {
    TH1 = -(SYSCLK/UART1_BAUDRATE/2);
    CKCON |= 0x10;                 // T1M = 1; SCA1:0 = xx
  }
  else if (SYSCLK/UART1_BAUDRATE/2/256 < 4)
 {
    TH1 = -(SYSCLK/UART1_BAUDRATE/2/4);
    CKCON &= ~0x13;                // Clear all T1 related bits
    CKCON |=  0x01;                // T1M = 0; SCA1:0 = 01
  }
 else if (SYSCLK/UART1_BAUDRATE/2/256 < 12)
{
    TH1 = -(SYSCLK/UART1_BAUDRATE/2/12);
    CKCON &= ~0x13;                // T1M = 0; SCA1:0 = 00
 }
```

```
 else
{
    TH1 = -(SYSCLK/UART1_BAUDRATE/2/48);
    CKCON &= ~0x13;                // Clear all T1 related bits
    CKCON |=  0x02;                // T1M = 0; SCA1:0 = 10
  }

  TL1 = TH1;                       // initialize Timer1
  TR1 = 1;                         // start Timer1
  SFRPAGE = UART1_PAGE;
  TI1 = 1;                         // Indicate TX1 ready

  EIE2 = EIE2 | 0x40;
  SFRPAGE = SFRPAGE_SAVE;          // Restore SFR page

}

//-----------------------------------------------------------------------------
// TimerInitiate()
//-----------------------------------------------------------------------------
// timer0 for OS ticks.
//

void TimerInitiate(void)
{
  char SFRPAGE_SAVE = SFRPAGE;       // Save Current SFR page

  SFRPAGE = TIMER01_PAGE;

          TCON   |= 0x00;
          TMOD   |= 0x01;            // TMOD: timer0, mode 1

          CKCON &= 0xF4;             //sysclk/12
          TH0     =  0xb0;
          TL0     =  0x3f;

  if (SYSCLK/UART1_BAUDRATE/2/256 < 1)
  {
          CKCON &= 0xF4;             //sysclk/12
          TH0     =  0xb0;
          TL0     =  0x3f;
  }
  else if (SYSCLK/UART1_BAUDRATE/2/256 < 4)
  {
          CKCON &= 0xF5;//sysclk/4
          CKCON  |= 0x01;//sysclk/4
          TH0      =  0x10;
          TL0      =  0xbd;
  }
  else if (SYSCLK/UART1_BAUDRATE/2/256 < 12)
  {
          CKCON &= 0xF4;//sysclk/12
          TH0     =  0xb0;
          TL0     =  0x3f;
  }
```

```
  else
  {
          CKCON&= 0xF6;//sysclk/12
          CKCON|= 0x02;
          TH0     =  0xEC;
          TL0     =  0x0F;
  }


  ET0   = 1;                    // enable timer zero overflow interrupt
  TR0   = 1;                    // start Timer0

  SFRPAGE = SFRPAGE_SAVE;           // Restore SFR page
}
```

IDEANIX_ISR.h

```
/*********************************************************************************************
*       Filename:       IDEANIX_ISR.h
*       Version:        V1.0
*       Author:         Darren Brown
*                       © Copyright 2006, University of Kentucky
*                       All Rights Reserved
*
*********************************************************************************************
*/

#ifndef isr_h
#define isr_h


        void UART0_ISR (void)  ;
        void UART1_ISR (void)  ;

        void PCA_isr(void);

        char putchar (char c);
        char getchar (void) ;


#endif
```

IDEANIX_ISR.c

```
/*********************************************************************************************
*       Filename:       IDEANIX_ISR.c
*       Version:        V1.0
*       Author:         Darren Brown
*                       © Copyright 2006, University of Kentucky
*                       All Rights Reserved
*
*********************************************************************************************
*/

#include <stdio.h>
#include "ideanix_config.h"
#include "ideanix_init.h"
#include "ideanix_isr.h"
#include "..\OS_config\INCLUDES.H"

 bit RI_0 = 0;
 bit RI_1 = 0;
 bit TI_0 = 0;
 bit TI_1 = 0;


void UART0_ISR (void)  interrupt 4
{
        unsigned char SFRPAGE_SAVE  = SFRPAGE;

        OSIntEnter();

   SFRPAGE=UART0_PAGE;

        if (RI0 == 1)
        {
                RI_0 = 1;
                RI0 = 0;                    // clear RX complete indicator
        }
        if( TI0 ==1 )
        {
                TI_0 = 1;
                TI0 = 0;
        }

        SFRPAGE=SFRPAGE_SAVE;

   OSIntExit();

}


void UART1_ISR (void)interrupt 20
{
        unsigned char SFRPAGE_SAVE = SFRPAGE;
        OSIntEnter();
        SFRPAGE=UART1_PAGE;
```

```
            if (RI1 == 1)
            {
                RI_1 = 1;               // handle receive function
                RI1 = 0;                // clear RX complete indicator
            }
            if( TI1 ==1 )
            {
                TI_1 = 1;
                TI1 = 0;
            }
SFRPAGE = SFRPAGE_SAVE ;
   OSIntExit();
}


//---------------------------------------------------------------------------------------
#define XON  0x11
#define XOFF 0x13

char putchar (char c)
{
    if (SFRPAGE == UART1_PAGE)
     {
            if (RI1)
            {
               if (SBUF1 == XOFF)
               {
                        do
                        {
                               RI1 = 0;
                               while (!RI1)
                               {;}
                         }
                         while (SBUF1 != XON)
                         {;}
                          RI1 = 0;
                }
            }
             while (!TI_1)
            {;}
             TI_1 = 0;
             return (SBUF1 = c);
      }
     else
     {
            if (RI0)
            {
               if (SBUF0 == XOFF)
                {
                            do
                            {
                                    RI0 = 0;
                                    while (!RI0)
                                    {;}
                            }
```

```
                        while (SBUF0 != XON)
                         {;}
                        RI0 = 0;
                    }
                }
               while (!TI_0);
               TI_0 = 0;
                return (SBUF0 = c);
        }
}


char _getkey (void)
{
  char c;
            if (SFRPAGE == UART1_PAGE)
            {
                        while(!RI_1)
                        {;}
                         c= SBUF1;
                           RI_1 = 0;
                        return (c);
            }
            else
            {
                        while (!RI_0)
                        {;}
                        c = SBUF0;
                                    RI_0 = 0;
                        return (c);
            }

}

char getchar (void)
{
  char c;
            if (SFRPAGE == UART1_PAGE)
            {
                        while(!RI_1)
                        {;}
                         c= SBUF1;
                           RI_1 = 0;
                        return (c);
            }
            else
            {
                        while (!RI_0)
                        {;}
                        c = SBUF0;
                                    RI_0 = 0;
                        return (c);
            }

}
```

IDEANIX_MSG_HANDLER.h

```
/*****************************************************************************
*       Filename:       IDEANIX_MSG_HANDLER.h
*       Version:        V1.0
*       Author:         Darren Brown
*                       © Copyright 2006, University of Kentucky
*                       All Rights Reserved
*
*****************************************************************************
*/

#ifndef ideanix_msg_handler_h
#define ideanix_msg_handler_h

#include "ideanix_config.h"

#if NETCOMM_EN  > 0

#define SIZE_of_DRIVER_Q 10


extern void *DRIVER_BUFF_Q[SIZE_of_DRIVER_Q];


void network_router (void *dataa) KCREENTRANT;

void load_up_buff (unsigned int id, unsigned long payload) ;


#endif
#endif
```

IDEANIX_MSG_HANDLER.c

```
/*********************************************************************************************
*       Filename:       IDEANIX_MSG_HANDLER.c
*       Version:        V1.0
*       Author:         Darren Brown
*                       © Copyright 2006, University of Kentucky
*                       All Rights Reserved
*
*********************************************************************************************
*/


#include "../OS_config/INCLUDES.H"
#include "ideanix_config.h"
#include "ideanix_init.h"
#include "ideanix_task_msg.h"
#include "ideanix_msg_handler.h"
#include "../main/OS_main.h"


#if NETCOMM_EN  > 0 && TASKCOMM_EN > 0

struct MSG_packet rx_buff[SIZE_of_DRIVER_Q];


OS_EVENT                *DRIVER_BUFF;
void *DRIVER_BUFF_Q  [SIZE_of_DRIVER_Q];

void network_router (void *dataa) KCREENTRANT
{
   void *msg;
   unsigned int id_local =0;
   unsigned long payload_local=0;
   unsigned char error =0 ;
   struct MSG_packet *msg_val;
   unsigned char SFRPAGE_save = SFRPAGE;

        DRIVER_BUFF = OSQCreate(&DRIVER_BUFF_Q[0], SIZE_of_DRIVER_Q);

        while(1)
        {
            msg = (void *)OSQPend(DRIVER_BUFF,0,&error);

            if(msg != (void*)0)
            {
                msg_val = (struct MSG_packet *)msg;
                id_local = msg_val->ID;
                payload_local= msg_val->PAYLOAD;
                send_local (id_local, payload_local);
            }

        }
 }
```

```
void load_up_buff (unsigned int  id, unsigned long payload)
{
        unsigned char error =0 ;
        unsigned char SFRPAGE_save = SFRPAGE;
        static unsigned char buff_cnt = 0;

        if(id <= MAX_ID_VAL )
        {
                if(     id_requested[id] != 0)
                {
                        rx_buff[buff_cnt].ID = id;
                        rx_buff[buff_cnt].PAYLOAD = payload;

                        OSQPost(DRIVER_BUFF, (void *)&rx_buff[buff_cnt]);
                        buff_cnt++;

                        if(buff_cnt == SIZE_of_DRIVER_Q)
                        {
                                buff_cnt = 0;
                        }
                }
         }
        SFRPAGE =SFRPAGE_save;

}

#endif
```

IDEANIX_TASK_MSG.h

```
/*********************************************************************************************
*       Filename:       IDEANIX_TASK_MSG.h
*       Version:        V1.0
*       Author:         Darren Brown
*                       © Copyright 2006, University of Kentucky
*                       All Rights Reserved
*
*********************************************************************************************
*/



#ifndef ideanix_user_CAN_h
#define ideanix_user_CAN_h
#include "ideanix_config.h"
#if TASKCOMM_EN > 0

extern OS_EVENT         *IN_QUEUEs[MAX_TASK_PRIO];

extern unsigned long    id_requested[MAX_ID_VAL];

extern unsigned int idmsg_cnt;
extern unsigned int snd_local_idmsg_cnt;

#define  NULL_val 0

struct MSG_packet {
                unsigned int ID;
                unsigned long PAYLOAD;
};


void init_taskqing (void);
        //init_taskqing is a function called from an initialization function from
        //main which initialized all existing queues to known zero'd value

unsigned char init_messaging (void );
        //init_messaging ()  has to be called at the beginning of each user task
        //it is what actually creates a message recieving queue for the task
        //it returns 1 meaning queue has been allocated successfully and
         //meaning queue has not been created

unsigned int send_msg ( unsigned int id , unsigned long payload ) KCREENTRANT;
        //send_msg successfully passes the payload to any local task who has
        //registered to recieve data from the specific id as well as calls the low
         //level driver function send_pkt to broadcast the id and payload over
        //the low level comm bus
        //Error codes returned are those reported from the low level driver

unsigned char send_local (unsigned int id, unsigned long payload) KCREENTRANT;
        //Away to send an id and payload to just local tasks
```

```
unsigned int reg_msg ( unsigned int id ) ;
        //reg_msg must be called from the task to register for the reception
        //of a payload sent with the specified id this function handles
        //association of ids with the task queues
        //returns include MLE_NO_Q which means no queue exists to handle
        //association or error codes from low level driver


unsigned int unreg_msg ( unsigned int id );
        // De-allocates space in hardware for this can_id and stops messages of this type from being placed
        //into the tasks' incoming buffer.
        // Returns error codes from low level driver


void unreg_all (void);
        //unregisters all ids reguardless of which task calls it

unsigned int get_msg ( unsigned int *id, unsigned long *payload ) KCREENTRANT;
        // Places the first can_id and payload in the queue into the memory
        //space refered to by the pointers.
        // get_msg pends meaning waits forever until a message was passed
        //into the task's queue
        //returns OS error codes assosicated with the queue pending process

unsigned int accept_msg ( unsigned int *id, unsigned long *payload ) KCREENTRANT;
        //does not wait for a message to be placed into the queue simply
        //checks to see if there was a message in the queue if there is
        //grabs the id and payload passes these values back through id and
        //payload if not leaves the values in id and payload as that of the
        //variables passed into the function
        //returns 1 if message was waiting and 0 if no message was waiting

unsigned char num_waiting_q_msgs (unsigned int *num_msgs) KCREENTRANT ;
        // the return value of num_waiting_q_msgs is any OS error that may
        //have occured when calling OSQQuery()
        //it also places the number of msgs waiting in the task's queue into the
        //variable passed into num_msgs
#endif
#endif
```

IDEANIX_TASK_MSG.c

```
/*********************************************************************************************
*       Filename:       IDEANIX_TASK_MSG.c
*       Version:        V1.0
*       Author:         Darren Brown
*                       © Copyright 2006, University of Kentucky
*                       All Rights Reserved
*
*********************************************************************************************
*/

#include "..\OS_config\includes.h"
#include "..\OS_port\os_kcdef.h"
#include "ideanix_msg_handler.h"
#include "ideanix_task_msg.h"
#include "ideanix_config.h"
#include "../main/OS_main.h"
#include "../NETWORK/can_header.h"

#if TASKCOMM_EN > 0
OS_EVENT                *IN_QUEUEs[MAX_TASK_PRIO];
volatile unsigned long   id_requested[MAX_ID_VAL];


//init_taskqing is a function called from an initialization function from
//main which initialized all existing queues to known zero'd value

void init_taskqing (void)
{
    unsigned char i ;
    unsigned int  j ;
        for(j=0;j<MAX_TASK_PRIO; j++)
        {
                IN_QUEUEs[j] = (OS_EVENT *) NULL_val;
        }
        for(i=0;i<MAX_ID_VAL;i++)
        {
                        id_requested[i]=0;
                        idmsg_cnt=0;
        }
}



//init_messaging ()  has to be called at the beginning of each user task
//it is what actually creates a message recieving queue for the task
//a return of 1 means messaging is enabled

unsigned char init_messaging (void ) //reentrant
{
    unsigned char error =0 ;
    unsigned char TASK_ID;   // this holds the value of the calling task's priority at the end of this function
    unsigned char SFRPAGE_save = SFRPAGE;
    OS_TCB Task_Data;
```

```
    OSTaskQuery(OS_PRIO_SELF, &Task_Data);
    TASK_ID = Task_Data.OSTCBPrio;

     if(TASK_ID <= MAX_TASK_PRIO)
         {
              if( create_q_space (TASK_ID) ==1 )
                  {
                      SFRPAGE=SFRPAGE_save;
                      return 1;
                  }
              else
                  {
                      SFRPAGE=SFRPAGE_save;
                      return Q_SPACE_NOT_CREATED;
                  }
          }
         else
          {
              return INVALID_TASK_PRIO;
          }
}

struct MSG_packet  payload_data[MAX_ID_VAL];
unsigned int idmsg_cnt;      // this is initialized in the function init_taskqing

//send_msg successfully passes the payload to any local task who has
//registered to recieve data from the specific id as well as calls the low
//level driver function send_pkt to broadcast the id and payload over
//the low level comm bus
//Error codes returned are those reported from the low level driver or OS_errors occured during
//SEMAPHORE implementation

unsigned int send_msg (unsigned int id , unsigned long payload ) KCREENTRANT
{
    unsigned char error =0 ;
    unsigned int j = 0;
    unsigned char flag =0;
    unsigned long id_req_tmp =0 ;
    unsigned long real_id_req  ;

    OSSemPend(Send_Msg_Sem,0,&error);

        #if NETCOMM_EN ==1
                send_pkt(id,payload);
        #endif

        // This reserves the space for the QUEUE to operate
        payload_data[idmsg_cnt].ID = id;
        payload_data[idmsg_cnt].PAYLOAD = payload;

        real_id_req =0 ;
        real_id_req = id_requested[id];
```

```
        if (id <= MAX_ID_VAL)
        {
            for(j=0;j<MAX_TASK_PRIO;j++)
                {
                    id_req_tmp = 0x00000001;
                    id_req_tmp = id_req_tmp <<j;

                   if ((real_id_req & id_req_tmp) >0)
                    {
                        OSQPost(IN_QUEUEs[j], (void *)&payload_data[idmsg_cnt]);
                        flag = 1;
                     }
                }
             if(flag ==1)
                {
                    flag =0 ;
                    idmsg_cnt++;   // only increment if a local task needs to grab the data

                     if(idmsg_cnt == MAX_ID_VAL)
                     {
                        idmsg_cnt =0;
                     }
                }
        }
        else
        {
                OSSemPost(Send_Msg_Sem);
                return INVALID_ID_VAL;
        }

        OSSemPost(Send_Msg_Sem);

    return ((int)error);
}


unsigned char send_local (unsigned int id, unsigned long payload) KCREENTRANT
{
    unsigned char error =0 ;
    unsigned int j = 0;
    unsigned char flag =0;
    unsigned long id_req_tmp =0 ;

        OSSemPend(Send_Msg_Sem,0,&error);

        // This reserves the space for the QUEUE to operate
        payload_data[idmsg_cnt].ID = id;
        payload_data[idmsg_cnt].PAYLOAD = payload;

        if (id <= MAX_ID_VAL)
        {
                for(j=0;j<MAX_TASK_PRIO;j++)
                {
                    id_req_tmp = 0x00000001;
                    id_req_tmp = id_req_tmp <<j;
```

```
                    if ((id_requested[id] & id_req_tmp) >0)
                    {
                        OSQPost(IN_QUEUEs[j], (void *)&payload_data[idmsg_cnt]);
                        flag = 1;
                    }
                }

                if(flag ==1)
                {
                    flag =0 ;
                    idmsg_cnt++;// only increment if a local task needs to grab the data  )
                    if(idmsg_cnt == MAX_ID_VAL)
                    {
                        idmsg_cnt =0;
                    }
                }
            }
        else
        {
            OSSemPost(Send_Msg_Sem);
            Return ( INVALID_ID_VAL);
        }

        OSSemPost(Send_Msg_Sem);
        return 0;
}

 //reg_msg must be called from the task to register for the reception
//of a payload sent with the specified id this function handles
//association of ids with the task queues
//returns include MLE_NO_Q which means no queue exists to handle
//association or error codes from low level driver

unsigned int reg_msg (unsigned int id )
{
    unsigned char err =0 ;
    unsigned char error = 0;    // capture errors from the low level messaging/can functions
    unsigned char TASK_ID; // this holds the value of the calling task's priority at the end of this function
    unsigned char cnt = 0;
    unsigned long tmp = 0x00000000;

        OS_TCB Task_Data;
        OSTaskQuery(OS_PRIO_SELF, &Task_Data);
        TASK_ID = Task_Data.OSTCBPrio;

          #if       NETCOMM_EN == 1
                    err =  reg_pkt ( id );
          #endif


         if (id <= MAX_ID_VAL)
        {
            if(TASK_ID <= MAX_TASK_PRIO)
            {
                // in init_queueing () IN_QUEUEs is initialized to a NULL_val which means if it still
                //equals NULL_val no QUEUE has been created for this task
```

```
                    if (IN_QUEUEs[TASK_ID] != (OS_EVENT *) NULL_val)
                    {
                       tmp = 0x00000001;
                       tmp= tmp<<TASK_ID;
                       id_requested[id] = id_requested[id] | tmp;
                       error = 0;
                     }
                    else
                    {
                       error=1; // NO QUEUE exsists
                    }
                 }
                 else
                {
                       error = INVALID_TASK_PRIO;              // set error = to invalid task priority;
                 }
            }
            else
            {
                       error = INVALID_ID_VAL;
            }

            return error;
}


// De-allocates space in hardware for this can_id and stops messages of this type from being placed into the
//tasks' incoming buffer.
// Returns error codes from low level driver

unsigned int unreg_msg (unsigned int id )
{
     unsigned char error =0;     // capture errors from the low level messaging/can functions
     unsigned char TASK_ID; // this holds the value of the calling task's priority at the end of this function
     unsigned char cnt =0;
     unsigned long tmp =0;

            OS_TCB Task_Data;
            OSTaskQuery(OS_PRIO_SELF, &Task_Data);
            TASK_ID = Task_Data.OSTCBPrio;

              #if       NETCOMM_EN == 1
                          unreg_pkt ( id );
              #endif

               if(id <= MAX_ID_VAL)
               {
                       tmp = 0x00000001;
                       tmp = tmp<<TASK_ID;
                       tmp = ~tmp;
                       id_requested[id] = id_requested[id] & tmp;
                }
               else
               {
                       return (INVALID_ID_VAL);
                }
```

```
        if(error > 0)
        {
                return 0;
        }

   return 1;
}


//unregisters all ids reguardless of which task calls it
void unreg_all (void)
{
        unsigned int i =0 ;
        unsigned int j =0 ;
        for(i=0;i<MAX_ID_VAL;i++)
        {
                #if     NETCOMM_EN == 1
                                unreg_pkt ( i );
                #endif

                for(j=0;j<MAX_TASK_PRIO;j++)
                {
                        id_requested[i]=0;
                }
        }
}


// Places the first can_id and payload in the queue into the memory
//space refered to by the pointers.
// get_msg pends meaning waits forever until a message was passed
//into the task's queue
//returns OS error codes assosicated with the queue pending process

unsigned int get_msg (unsigned int *id, unsigned long *payload ) KCREENTRANT
{
  void *msg;
  unsigned char sem_error =0;
  unsigned char error =0;

  unsigned char TASK_ID;
  struct MSG_packet *msg_val;
  unsigned int id_local =0;
  unsigned long payload_local=0;

        OS_TCB Task_Data;
        OSTaskQuery(OS_PRIO_SELF, &Task_Data);
        TASK_ID = Task_Data.OSTCBPrio;

   msg = (void *)OSQPend(IN_QUEUEs[TASK_ID],0,&error);

        if(msg != (void*)0)
        {
                msg_val = (struct MSG_packet *)msg;
                id_local = msg_val->ID;
                payload_local= msg_val->PAYLOAD;
```

```
                *id      = id_local;
                *payload = payload_local;
                        OSSemPost(Get_Msg_Sem);
                        return 1;
        }
        else
        {
                OSSemPost(Get_Msg_Sem);
                return 0;
        }

        OSSemPost(Get_Msg_Sem);
   return 0;
}



//does not wait for a message to be placed into the queue simply
//checks to see if there was a message in the queue if there is
//grabs the id and payload passes these values back through id and
//payload if not leaves the values in id and payload as that of the
//variables passed into the function
//returns 1 if message was waiting and 0 if no message was waiting

unsigned int accept_msg ( unsigned int *id, unsigned long *payload ) KCREENTRANT
{
  void *msg;
  unsigned char sem_error =0;
  unsigned char error =0;

  unsigned char TASK_ID;
  struct MSG_packet *msg_val;
  unsigned int id_local ;
  unsigned long payload_local;

        OS_TCB Task_Data;
        OSSemPend(Accept_Msg_Sem,0,&sem_error);
        OSTaskQuery(OS_PRIO_SELF, &Task_Data);
        TASK_ID = Task_Data.OSTCBPrio;

   msg = (void *)OSQAccept(IN_QUEUEs[TASK_ID]);

        if(msg != (void*)0)
        {
                msg_val = (struct MSG_packet *)msg;
                id_local = msg_val->ID;
                payload_local= msg_val->PAYLOAD;

                *id      = id_local;
                *payload = payload_local;
                        OSSemPost(Accept_Msg_Sem);
                        return 1;
        }
```

```
        else
        {
                OSSemPost(Accept_Msg_Sem);
                return 0;
        }

}

// the return value of this function is any error that may have occured when calling OSQQuery()
//it also places the number of msgs waiting in the task's queue into the variable passed into num_msgs

unsigned char num_waiting_q_msgs (unsigned int *num_msgs) KCREENTRANT
{

        unsigned char TASK_ID =0 ;
        unsigned char error =0 ;
        OS_Q_DATA q_info;
        OS_TCB Task_Data;

        OSTaskQuery(OS_PRIO_SELF, &Task_Data);
        TASK_ID = Task_Data.OSTCBPrio;

        error = OSQQuery(IN_QUEUEs[TASK_ID], &q_info);
        if(error == OS_NO_ERR)
        {
                *num_msgs = q_info.OSNMsgs;
        }

    return error;
}

#endif
```

IDEANIX_CONFIG.h

```
/*****************************************************************************
*       Filename:       IDEANIX_CONFIG.h
*       Version:        V1.0
*       Author:         Darren Brown
*                       © Copyright 2006, University of Kentucky
*                       All Rights Reserved
*
*****************************************************************************
*/



#ifndef IDEANIX_config_h
#define IDEANIX_config_h


unsigned char create_q_space (unsigned char TASK_ID);


#define NETCOMM_EN          1
#define TASKCOMM_EN         1
#define MAX_ID_VAL          25
#define MAX_TASK_PRIO       10   //this value should be equvilanent to the highest priority you set
                                 //for your tasks remember the highest priority supported for
                                 //message routing is 15

#define NUM_TASKs 7
#define SIZE_of_Q 10

//PCA ENABLE
        #define PCA_ENABLE              1

//PCA OUTPUT SELECTIONS
        #define PCA_OUT_ENABLE      0

        #define  PCA_OUT_CH0 0
        #define  PCA_OUT_CH1 0
        #define  PCA_OUT_CH2 0
        #define  PCA_OUT_CH3 0
        #define  PCA_OUT_CH4 0
        #define  PCA_OUT_CH5 0

//PCA CAPTURE SELECTIONS
        #define PCA_CAPTURE_ENABLE 1

        #define  PCA_CAPT_CH0       1
        #define  PCA_CAPT_CH1       1
        #define  PCA_CAPT_CH2       1
        #define  PCA_CAPT_CH3       1
        #define  PCA_CAPT_CH4       1
        #define  PCA_CAPT_CH5       1
```

```
//OS TASKS INFORMATION
#define  TASK_STK_SIZE            512     /* Size of each task's stacks (# of WORDs)        */
#define  STK_SIZE                 512     /* Size of each task's stacks (# of WORDs)        */


//ERROR CODES
#define INVALID_TASK_PRIO   175
#define Q_SPACE_NOT_CREATED   176
#define INVALID_ID_VAL 177
#define NO_QUEUE 178

#define PCA_OUT_CHANNEL_INVALID  180
#define PCA_CAPT_CHANNEL_INVALID  181

#endif
```

IDEANIX_CONFIG.c

```
/*********************************************************************************************
*       Filename:       IDEANIX_CONFIG.c
*       Version:        V1.0
*       Author:         Darren Brown
*                       © Copyright 2006, University of Kentucky
*                       All Rights Reserved
*
*********************************************************************************************
*/



#include "..\OS_config\includes.h"
#include "ideanix_task_msg.h"

#if  TASKCOMM_EN > 0
/// What follows is the allocation of queue arrays to reserve space for the queue in program memory
// this macro will create a number of queue arrays up to the number of TASKs

#if MAX_TASK_PRIO > 0
        void *Q0[SIZE_of_Q];
#endif

#if MAX_TASK_PRIO > 1
        void *Q1[SIZE_of_Q];
#endif

#if MAX_TASK_PRIO > 2
        void *Q2[SIZE_of_Q];
#endif

#if MAX_TASK_PRIO > 3
        void *Q3[SIZE_of_Q];
#endif

#if MAX_TASK_PRIO > 4
        void *Q4[SIZE_of_Q];
#endif

#if MAX_TASK_PRIO > 5
        void *Q5[SIZE_of_Q];
#endif

#if MAX_TASK_PRIO > 6
        void *Q6[SIZE_of_Q];
#endif

#if MAX_TASK_PRIO > 7
        void *Q7[SIZE_of_Q];
#endif

#if MAX_TASK_PRIO > 8
        void *Q8[SIZE_of_Q];
```

```
#endif

#if MAX_TASK_PRIO > 9
        void *Q9[SIZE_of_Q];
#endif

#if MAX_TASK_PRIO > 10
        void *Q10[SIZE_of_Q];
#endif

#if MAX_TASK_PRIO > 11
        void *Q11[SIZE_of_Q];
#endif

#if MAX_TASK_PRIO > 12
        void *Q12[SIZE_of_Q];
#endif

#if MAX_TASK_PRIO > 13
        void *Q13[SIZE_of_Q];
#endif

#if MAX_TASK_PRIO > 14
        void *Q14[SIZE_of_Q];
#endif


#if MAX_TASK_PRIO > 15
        void *Q15[SIZE_of_Q];
#endif


#if MAX_TASK_PRIO > 16
        void *Q16[SIZE_of_Q];
#endif


#if MAX_TASK_PRIO > 17
        void *Q17[SIZE_of_Q];
#endif


#if MAX_TASK_PRIO > 18
        void *Q18[SIZE_of_Q];
#endif


#if MAX_TASK_PRIO > 19
        void *Q19[SIZE_of_Q];
#endif


#if MAX_TASK_PRIO > 20
        void *Q20[SIZE_of_Q];
#endif
```

```
#if MAX_TASK_PRIO > 21
        void *Q21[SIZE_of_Q];
#endif

#if MAX_TASK_PRIO > 22
        void *Q22[SIZE_of_Q];
#endif

#if MAX_TASK_PRIO > 23
        void *Q23[SIZE_of_Q];
#endif

#if MAX_TASK_PRIO > 24
        void *Q24[SIZE_of_Q];
#endif

#if MAX_TASK_PRIO > 25
        void *Q25[SIZE_of_Q];
#endif



unsigned char create_q_space (unsigned char TASK_ID)
{
        switch (TASK_ID)
        {
                #if MAX_TASK_PRIO > 0
                    case 0:
                        IN_QUEUEs[TASK_ID] = OSQCreate(&Q0[0], SIZE_of_Q );
                        break;
                #endif

                #if MAX_TASK_PRIO > 1
                    case 1:
                        IN_QUEUEs[TASK_ID] = OSQCreate(&Q1[0], SIZE_of_Q );
                        break;
                #endif

                #if MAX_TASK_PRIO > 2
                    case 2:
                       IN_QUEUEs[TASK_ID] = OSQCreate(&Q2[0], SIZE_of_Q );
                       break;
                #endif

                #if MAX_TASK_PRIO > 3
                    case 3:
                        IN_QUEUEs[TASK_ID] = OSQCreate(&Q3[0], SIZE_of_Q );
                        break;
                #endif

                #if MAX_TASK_PRIO > 4
                    case 4:
                        IN_QUEUEs[TASK_ID] = OSQCreate(&Q4[0], SIZE_of_Q );
                        break;
                #endif
```

```
#if MAX_TASK_PRIO > 5
    case 5:
            IN_QUEUEs[TASK_ID] = OSQCreate(&Q5[0], SIZE_of_Q );
            break;
#endif

#if MAX_TASK_PRIO > 6
    case 6:
            IN_QUEUEs[TASK_ID] = OSQCreate(&Q6[0], SIZE_of_Q );
            break;
#endif

#if MAX_TASK_PRIO > 7
    case 7:
            IN_QUEUEs[TASK_ID] = OSQCreate(&Q7[0], SIZE_of_Q );
            break;
#endif

#if MAX_TASK_PRIO > 8
    case 8:
            IN_QUEUEs[TASK_ID] = OSQCreate(&Q8[0], SIZE_of_Q );
            break;
#endif

#if MAX_TASK_PRIO > 9
    case 9:
            IN_QUEUEs[TASK_ID] = OSQCreate(&Q9[0], SIZE_of_Q );
            break;
#endif

#if MAX_TASK_PRIO > 10
    case 10:
             IN_QUEUEs[TASK_ID] = OSQCreate(&Q10[0], SIZE_of_Q );
             break;
#endif

#if MAX_TASK_PRIO > 11
    case 11:
             IN_QUEUEs[TASK_ID] = OSQCreate(&Q11[0], SIZE_of_Q );
             break;
#endif

#if MAX_TASK_PRIO > 12
    case 12:
             IN_QUEUEs[TASK_ID] = OSQCreate(&Q12[0], SIZE_of_Q );
             break;
#endif

#if MAX_TASK_PRIO > 13
   case 13:
            IN_QUEUEs[TASK_ID] = OSQCreate(&Q13[0], SIZE_of_Q );
            break;
#endif
```

```
#if MAX_TASK_PRIO > 14
    case 14:
            IN_QUEUEs[TASK_ID] = OSQCreate(&Q14[0], SIZE_of_Q );
            break;
#endif


#if MAX_TASK_PRIO > 15
    case 15:
            IN_QUEUEs[TASK_ID] = OSQCreate(&Q15[0], SIZE_of_Q );
            break;
#endif

#if MAX_TASK_PRIO > 16
    case 16:
            IN_QUEUEs[TASK_ID] = OSQCreate(&Q16[0], SIZE_of_Q );
            break;
#endif

#if MAX_TASK_PRIO > 17
     case 17:
             IN_QUEUEs[TASK_ID] = OSQCreate(&Q17[0], SIZE_of_Q );
              break;
#endif

#if MAX_TASK_PRIO > 18
     case 18:
             IN_QUEUEs[TASK_ID] = OSQCreate(&Q18[0], SIZE_of_Q );
             break;
#endif

#if MAX_TASK_PRIO > 19
     case 19:
            IN_QUEUEs[TASK_ID] = OSQCreate(&Q19[0], SIZE_of_Q );
            break;
#endif

#if MAX_TASK_PRIO > 20
     case 20:
             IN_QUEUEs[TASK_ID] = OSQCreate(&Q20[0], SIZE_of_Q );
             break;
#endif

#if MAX_TASK_PRIO > 21
     case 21:
             IN_QUEUEs[TASK_ID] = OSQCreate(&Q20[0], SIZE_of_Q );
             break;
#endif

#if MAX_TASK_PRIO > 22
     case 22:
             IN_QUEUEs[TASK_ID] = OSQCreate(&Q20[0], SIZE_of_Q );
             break;
#endif
```

```
                #if MAX_TASK_PRIO > 23
                    case 23:
                            IN_QUEUEs[TASK_ID] = OSQCreate(&Q20[0], SIZE_of_Q );
                            break;
                #endif

                #if MAX_TASK_PRIO > 24
                    case 24:
                            IN_QUEUEs[TASK_ID] = OSQCreate(&Q20[0], SIZE_of_Q );
                            break;
                #endif

                #if MAX_TASK_PRIO > 25
                    case 25:
                            IN_QUEUEs[TASK_ID] = OSQCreate(&Q20[0], SIZE_of_Q );
                            break;
                #endif

                    default:
                            return 0;
                            break;
                }
        return 1;
}
#endif // end of #if TASKCOMM_EN

//ERRORs

#if MAX_TASK_PRIO > 15
        #error  "MAX_TASK_PRIO is set to a value larger than IDEANIX intertask communication can
                support (15 is max)"
#endif

#if TASKCOMM_EN == 0 && NETCOMM_EN ==1
        #error "YOU CAN NOT ENABLE NETCOMM_EN without ENABLING TASKCOMM_EN"
#endif

//PCA init ERRORS
        #if PCA_ENABLE > 0
            #if PCA_OUT_ENABLE > 0 && PCA_CAPTURE_ENABLE > 0
                #if PCA_OUT_CH0 == 1 && PCA_CAPT_CH0 == 1
                    #error "CAN NOT ENABLE CH0 FOR BOTH output and capture"
                #endif
                #if PCA_OUT_CH1 == 1 && PCA_CAPT_CH1 == 1
                    #error "CAN NOT ENABLE CH1 FOR BOTH output and capture"
                #endif
                #if PCA_OUT_CH2 == 1 && PCA_CAPT_CH2 == 1
                    #error "CAN NOT ENABLE CH2 FOR BOTH output and capture"
                #endif
                #if PCA_OUT_CH3 == 1 && PCA_CAPT_CH3 == 1
                    #error "CAN NOT ENABLE CH3 FOR BOTH output and capture"
                #endif
                #if PCA_OUT_CH4 == 1 && PCA_CAPT_CH4 == 1
                    #error "CAN NOT ENABLE CH4 FOR BOTH output and capture"
                #endif
```

```
        #if PCA_OUT_CH5 == 1 && PCA_CAPT_CH5 == 1
            #error "CAN NOT ENABLE CH5 FOR BOTH output and capture"
        #endif
    #endif
#endif
```

IDEANIX_SERVO.h

```
/*****************************************************************************
*       Filename:       IDEANIX_SERVO.h
*       Version:        V1.0
*       Author:         Darren Brown
*                       © Copyright 2006, University of Kentucky
*                       All Rights Reserved
*
*****************************************************************************
*/

#ifndef ideanix_servo_h
#define ideanix_servo_h

#if PCA_ENABLE > 0


    extern unsigned int  PWM0_HIGH; // Number of PCA clocks for waveform to be high
    extern unsigned int PWM0_LOW;  // Number of PCA clocks for  waveform to be low
                                    // duty cycle = PWM_HIGH / (PWM_HIGH + PWM_LOW)
    extern unsigned int  PWM1_HIGH; // Number of PCA clocks for waveform to be high
    extern unsigned int PWM1_LOW;  // Number of PCA clocks for  waveform to be low
                                    // duty cycle = PWM_HIGH / (PWM_HIGH + PWM_LOW)
    extern unsigned int  PWM2_HIGH; // Number of PCA clocks for waveform to be high
    extern unsigned int PWM2_LOW;  // Number of PCA clocks for  waveform to be low
                                    // duty cycle = PWM_HIGH / (PWM_HIGH + PWM_LOW)
    extern unsigned int  PWM3_HIGH; // Number of PCA clocks for waveform to be high
    extern unsigned int PWM3_LOW;  // Number of PCA clocks for  waveform to be low
                                    // duty cycle = PWM_HIGH / (PWM_HIGH + PWM_LOW)
    extern unsigned int  PWM4_HIGH; // Number of PCA clocks for waveform to be high
    extern unsigned int PWM4_LOW;   // Number of PCA clocks for  waveform to be low
                                     // duty cycle = PWM_HIGH / (PWM_HIGH + PWM_LOW)
    extern unsigned int  PWM5_HIGH; // Number of PCA clocks for waveform to be high
    extern unsigned int PWM5_LOW;   // Number of PCA clocks for  waveform to be low
                                     // duty cycle = PWM_HIGH / (PWM_HIGH + PWM_LOW)


    void PCA_Init(void);
    void PCA_ISR (void);
    void set_pwm (unsigned int freq, float duty, unsigned char channel);
    void set_high_low(unsigned int high, unsigned int low, unsigned char channel);

    #if PCA_OUT_ENABLE > 0
        //use set_output to set the output on a selected PCA channel after recieving a bit packed
        // high and low time through a long payload over CAN
        unsigned char set_output (unsigned long payload, unsigned char channel);
    #endif


    #if PCA_CAPTURE_ENABLE > 0
        unsigned char get_capture (unsigned long *ch_data, unsigned char channel);
    #endif

#endif //end of PCA_ENABLE macro
```

```
#endif //end of ifndef
```

IDEANIX_SERVO.c

```
/*****************************************************************************
*       Filename:       IDEANIX_SERVO.c
*       Version:        V1.0
*       Author:         Darren Brown
*                       © Copyright 2006, University of Kentucky
*                       All Rights Reserved
*
*****************************************************************************
*/

#include "..\OS_config\includes.h"
#include "ideanix_config.h"
#include "ideanix_servo.h"
#include "ideanix_init.h"
#if PCA_ENABLE > 0

#define  PWM_START  0x0bf6

unsigned int  PWM0_HIGH = 0x0bf6;          // Number of PCA clocks for waveform to be high
unsigned int PWM0_LOW = 0x9f81-0x0bf6; // Number of PCA clocks for  waveform to be low
                                                // duty cycle = PWM_HIGH / (PWM_HIGH + PWM_LOW)
unsigned int  PWM1_HIGH = 0x0bf6;          // Number of PCA clocks for waveform to be high
unsigned int PWM1_LOW = 0x9f81-0x0bf6; // Number of PCA clocks for  waveform to be low
                                                // duty cycle = PWM_HIGH / (PWM_HIGH + PWM_LOW)
unsigned int  PWM2_HIGH = 0x0bf6;          // Number of PCA clocks for waveform to be high
unsigned int PWM2_LOW = 0x9f81-0x0bf6; // Number of PCA clocks for  waveform to be low
                                                // duty cycle = PWM_HIGH / (PWM_HIGH + PWM_LOW)
unsigned int  PWM3_HIGH = 0x0bf6;          // Number of PCA clocks for waveform to be high
unsigned int PWM3_LOW = 0x9f81-0x0bf6; // Number of PCA clocks for  waveform to be low
                                                // duty cycle = PWM_HIGH / (PWM_HIGH + PWM_LOW)
unsigned int  PWM4_HIGH = 0x0bf6;          // Number of PCA clocks for waveform to be high
unsigned int PWM4_LOW = 0x9f81-0x0bf6; // Number of PCA clocks for  waveform to be low
                                                // duty cycle = PWM_HIGH / (PWM_HIGH + PWM_LOW)
unsigned int  PWM5_HIGH = 0x0bf6;          // Number of PCA clocks for waveform to be high
unsigned int PWM5_LOW = 0x9f81-0x0bf6; // Number of PCA clocks for  waveform to be low
                                                // duty cycle = PWM_HIGH / (PWM_HIGH + PWM_LOW)

void PCA_Init (void)
{
//PWM SETUP
        unsigned char sfrpage_save = SFRPAGE;

        SFRPAGE = PCA0_PAGE;
        PCA0MD = 0x00; // disable CF interrupt
                        // PCA time base = SYSCLK / 4

        #if PCA_OUT_ENABLE
                //enable PCA as high speed output (PWM with controllable frequency)

                #if PCA_OUT_CH0 > 0
                        set_pwm (50, 7.5, 0);
                        PCA0CPL0 = (0xff & PWM0_HIGH); // initialize PCA compare value
                        PCA0CPH0 = (0xff & (PWM0_HIGH >> 8));
```

```
                PCA0CPM0 = 0x4d;// CCM0 in High Speed output mode
        #endif

        #if PCA_OUT_CH1 > 0
                set_pwm (50, 7.5, 1);
                PCA0CPL1 = (0xff & PWM1_HIGH); // initialize PCA compare value
                PCA0CPH1 = (0xff & (PWM1_HIGH >> 8));
                PCA0CPM1 = 0x4d;//in High Speed output mode
        #endif

        #if PCA_OUT_CH2 > 0
                set_pwm (50, 7.5, 2);
                PCA0CPL2 = (0xff & PWM2_HIGH); // initialize PCA compare value
                PCA0CPH2 = (0xff & (PWM2_HIGH >> 8));
                PCA0CPM2 = 0x4d; // High Speed output mode
        #endif

        #if PCA_OUT_CH3 > 0
                set_pwm (50, 7.5, 3);
                PCA0CPL3 = (0xff & PWM3_HIGH); // initialize PCA compare value
                PCA0CPH3 = (0xff & (PWM3_HIGH >> 8));
                PCA0CPM3 = 0x4d;// in High Speed output mode
        #endif

        #if PCA_OUT_CH4 > 0
                set_pwm (50, 7.5, 4);
                PCA0CPL4 = (0xff & PWM4_HIGH); // initialize PCA compare value
                PCA0CPH4 = (0xff & (PWM4_HIGH >> 8));
                PCA0CPM4 = 0x4d;//  in High Speed output mode
        #endif

        #if PCA_OUT_CH5 > 0
                set_pwm (50, 7.5, 5);
                PCA0CPL5 = (0xff & PWM5_HIGH); // initialize PCA compare value
                PCA0CPH5 = (0xff & (PWM5_HIGH >> 8));
                PCA0CPM5 = 0x4d;//  in High Speed output mode
        #endif
#endif

#if PCA_CAPTURE_ENABLE > 0
        #if     PCA_CAPT_CH0 > 0
                PCA0CPM0 = 0x21;// 0b00100001 = 0x21  set for positive edge detection
        #endif
        #if     PCA_CAPT_CH1 > 0
                PCA0CPM1 = 0x21;// 0b00100001 = 0x21  set for positive edge detection
        #endif
        #if     PCA_CAPT_CH2 > 0
                PCA0CPM2 = 0x21;// 0b00100001 = 0x21  set for positive edge detection
        #endif
        #if     PCA_CAPT_CH3 > 0
                PCA0CPM3 = 0x21;// 0b00100001 = 0x21  set for positive edge detection
        #endif
        #if     PCA_CAPT_CH4 > 0
                PCA0CPM4 = 0x21;// 0b00100001 = 0x21  set for positive edge detection
        #endif
```

```
                #if     PCA_CAPT_CH5 > 0
                        PCA0CPM5 = 0x21;// 0b00100001 = 0x21  set for positive edge detection
                #endif
        #endif

// all PCA initialization
        EIE1 |= 0x08;     // enable PCA interrupt
        PCA0CN = 0x40;  // enable PCA counter


}

//valid for period from 1us to 32ms  1MHz to 31.25 Hz
//duty is % in whole numbers 0 to 100

void set_pwm (unsigned int freq, float duty, unsigned char channel)
{
 double seconds;
 unsigned int period;
 unsigned char SFRPAGE_save = SFRPAGE;

        SFRPAGE = PCA0_PAGE;
  seconds = (double)((double)1/(double)freq);
  period = (unsigned int)(seconds*((double)SYSCLK/(double)12));

  switch(channel)
        {
                case 0:
                          PWM0_HIGH=(unsigned int)((double)duty*((double)period/(double)100));
                          PWM0_LOW = period - PWM0_HIGH;
                        break;
                case 1:
                          PWM1_HIGH=(unsigned int)((double)duty*((double)period/(double)100));
                          PWM1_LOW = period - PWM1_HIGH;
                        break;
                case 2:
                          PWM2_HIGH=(unsigned int)((double)duty*((double)period/(double)100));
                          PWM2_LOW = period - PWM2_HIGH;
                        break;
                case 3:
                          PWM3_HIGH=(unsigned int)((double)duty*((double)period/(double)100));
                          PWM3_LOW = period - PWM3_HIGH;
                        break;
                case 4:
                          PWM4_HIGH=(unsigned int)((double)duty*((double)period/(double)100));
                          PWM4_LOW = period - PWM4_HIGH;
                        break;
                case 5:
                          PWM5_HIGH=(unsigned int)((double)duty*((double)period/(double)100));
                          PWM5_LOW = period - PWM5_HIGH;
                        break;
        }
        SFRPAGE = SFRPAGE_save ;

}
```

```
void set_high_low(unsigned int high, unsigned int low, unsigned char channel)
{
 double seconds;
 unsigned int period;
 unsigned char SFRPAGE_save = SFRPAGE;

  switch(channel)
        {
                case 0:
                        PWM0_HIGH=high;
                        PWM0_LOW =low;
                      break;
                case 1:
                        PWM1_HIGH=high;
                        PWM1_LOW =low;
                      break;
                case 2:
                        PWM2_HIGH=high;
                        PWM2_LOW =low;
                      break;
                case 3:
                        PWM3_HIGH=high;
                        PWM3_LOW =low;
                      break;
                case 4:
                        PWM4_HIGH=high;
                        PWM4_LOW =low;
                      break;
                case 5:
                        PWM5_HIGH=high;
                        PWM5_LOW =low;
                      break;
        }
        SFRPAGE = SFRPAGE_save ;

}


#if PCA_OUT_ENABLE >0

        unsigned char flag0  =0 ;
        unsigned char flag1  =0 ;
        unsigned char flag2  =0 ;
        unsigned char flag3  =0 ;
        unsigned char flag4  =0 ;
        unsigned char flag5  =0 ;


#endif

#if PCA_CAPTURE_ENABLE > 0

unsigned int CAPTURE_P0 =0 ; // this will hold the time it took from the positive edge detection to the
                                //next edge
```

```
unsigned int CAPTURE_N0 =0 ; // this will hold the time it took from the falling edge to the next edge
                              //which will be a rising

unsigned int CAPTURE_P1 =0 ; // this will hold the time it took from the positive edge detection to the
                              //next edge
unsigned int CAPTURE_N1 =0 ; // this will hold the time it took from the falling edge to the next edge
                              //which will be a rising

unsigned int CAPTURE_P2 =0 ; // this will hold the time it took from the positive edge detection to the
                             //next edge
unsigned int CAPTURE_N2 =0 ; // this will hold the time it took from the falling edge to the next edge
                              //which will be a rising

unsigned int CAPTURE_P3 =0 ; // this will hold the time it took from the positive edge detection to the
                             //next edge
unsigned int CAPTURE_N3 =0 ; // this will hold the time it took from the falling edge to the next edge
                              // which will be a rising

unsigned int CAPTURE_P4 =0 ; // this will hold the time it took from the positive edge detection to the
                             //next edge
unsigned int CAPTURE_N4 =0 ; // this will hold the time it took from the falling edge to the next edge
                              //which will be a rising

unsigned int CAPTURE_P5 =0 ; // this will hold the time it took from the positive edge detection to the
                             //next edge
unsigned int CAPTURE_N5 =0 ; // this will hold the time it took from the falling edge to the next edge
                              //which will be a rising

unsigned int DUTY0 = 10 ;
unsigned char find_period0 = 0;


unsigned int DUTY1 = 10 ;
unsigned char find_period1 = 0;

unsigned int DUTY2 = 10 ;
unsigned char find_period2 = 0;

unsigned int DUTY3 = 10 ;
unsigned char find_period3 = 0;

unsigned int DUTY4 = 10 ;
unsigned char find_period4 = 0;

unsigned int DUTY5 = 10 ;
unsigned char find_period5 = 0;

unsigned char pedge0 =1; //sets the positive edge flag to 1
unsigned int PERIOD0;

unsigned char pedge1 =1; //sets the positive edge flag to 1
unsigned int PERIOD1;

unsigned char pedge2 =1; //sets the positive edge flag to 1
unsigned int PERIOD2;
```

```
unsigned char pedge3 =1; //sets the positive edge flag to 1
unsigned int PERIOD3;

unsigned char pedge4 =1; //sets the positive edge flag to 1
unsigned int PERIOD4;

unsigned char pedge5 =1; //sets the positive edge flag to 1
unsigned int PERIOD5;

unsigned int HIGH_TIME_0 =0x0bf6 ;
unsigned int LOW_TIME_0 = 0x9f81-0x0bf6;

unsigned int HIGH_TIME_1 =0x0bf6 ;
unsigned int LOW_TIME_1 =0x9f81-0x0bf6;

unsigned int HIGH_TIME_2 =0x0bf6 ;
unsigned int LOW_TIME_2 =0x9f81-0x0bf6;

unsigned int HIGH_TIME_3 =0x0bf6 ;
unsigned int LOW_TIME_3 =0x9f81-0x0bf6;

unsigned int HIGH_TIME_4 =0x0bf6 ;
unsigned int LOW_TIME_4 =0x9f81-0x0bf6;

unsigned int HIGH_TIME_5 =0x0bf6;
unsigned int LOW_TIME_5 =0x9f81-0x0bf6;

#endif

void PCA_ISR (void) interrupt 9
{
   unsigned int temp; // holding value for 16-bit math
   unsigned char SFRPAGE_save = SFRPAGE;

        SFRPAGE = PCA0_PAGE;

        if (CCF0)
        {
                CCF0 = 0; // clear compare indicator
                #if PCA_OUT_CH0 > 0 && PCA_OUT_ENABLE >0
                        if (flag0)//(PWM0_OUT)
                        {       // process rising edge
                                // update compare match for next falling edge
                                flag0 =0;
                                temp = (PCA0CPH0 << 8) | PCA0CPL0; // get current compare value
                                temp += PWM0_HIGH; // add appropriate offset
                                PCA0CPL0 = (0xff & temp); // replace compare value
                                PCA0CPH0 = (0xff & (temp >> 8));
                        }
                        else
                        {       // process falling edge
                                // update compare match for next rising edge
                                flag0 =1;
                                temp = (PCA0CPH0 << 8) | PCA0CPL0; // get current compare value
                                temp += PWM0_LOW; // add appropriate offset
                                PCA0CPL0 = (0xff & temp); // replace compare value
```

```
                PCA0CPH0 = (0xff & (temp >> 8));
            }
        #elif PCA_CAPT_CH0 > 0 && PCA_CAPTURE_ENABLE > 0
            if(pedge0)
            {
                pedge0=0;//now look for falling edge
                if(find_period0 == 1)
                {   temp =((PCA0CPH0 << 8) | PCA0CPL0);
                   PERIOD0 = temp - CAPTURE_P0;
                   HIGH_TIME_0=DUTY0;
                   LOW_TIME_0=(PERIOD0-DUTY0);
                   CAPTURE_P0 =temp ; // get current compare value
                   find_period0=0;
                }
                else
                {
                   // get current compare value
                    CAPTURE_P0 = (PCA0CPH0 << 8) | PCA0CPL0;
                    find_period0 =1;
                }

                PCA0CPM0 = 0x11;
            }
            else
            {
                    // get current compare value
                    CAPTURE_N0 = (PCA0CPH0 << 8) | PCA0CPL0;
                    DUTY0 =  CAPTURE_N0 - CAPTURE_P0;
                    pedge0=1;//now look for rising/positive edge
                    PCA0CPM0 = 0x21;
            }
            #endif
}
else if (CCF1)
{
        CCF1 = 0; // clear compare indicator
        #if PCA_OUT_CH1 >0 && PCA_OUT_ENABLE >0
            if (flag1)//(PWM0_OUT)
            {       // process rising edge
                // update compare match for next falling edge
                flag1 =0;
                temp = (PCA0CPH1 << 8) | PCA0CPL1; // get current compare value
                temp += PWM1_HIGH; // add appropriate offset
                PCA0CPL1 = (0xff & temp); // replace compare value
                PCA0CPH1 = (0xff & (temp >> 8));
            }
            else
            {       // process falling edge
                // update compare match for next rising edge
                flag1 =1;
                temp = (PCA0CPH1 << 8) | PCA0CPL1; // get current compare value
                temp += PWM1_LOW; // add appropriate offset
                PCA0CPL1 = (0xff & temp); // replace compare value
                PCA0CPH1 = (0xff & (temp >> 8));
            }
        #elif PCA_CAPT_CH1 > 0 && PCA_CAPTURE_ENABLE > 0
```

```
                    if(pedge1)
                    {
                            pedge1=0;//now look for falling edge
                            if(find_period1 == 1)
                            {   temp =((PCA0CPH1 << 8) | PCA0CPL1);
                               PERIOD1 = temp - CAPTURE_P1;
                               HIGH_TIME_1=DUTY1;
                               LOW_TIME_1=(PERIOD1-DUTY1);
                               CAPTURE_P1 =temp ; // get current compare value
                               find_period1=0;
                            }
                            else
                            {
                               // get current compare value
                                CAPTURE_P1 = (PCA0CPH1 << 8) | PCA0CPL1;
                                find_period1 =1;
                            }


                            PCA0CPM1 = 0x11;
                    }
                    else
                    {
                            // get current compare value
                            CAPTURE_N1 = (PCA0CPH1 << 8) | PCA0CPL1;
                            DUTY1 =  CAPTURE_N1 - CAPTURE_P1;
                            pedge1=1;//now look for rising/positive edge
                            PCA0CPM1 = 0x21;
                    }
            #endif

}
else if (CCF2)
{
            CCF2 = 0;
            #if PCA_OUT_CH2 >0 && PCA_OUT_ENABLE >0
                    if (flag2)//(PWM0_OUT)
                    {       // process rising edge
                            // update compare match for next falling edge
                            flag2 =0;
                            temp = (PCA0CPH2 << 8) | PCA0CPL2; // get current compare value
                            temp += PWM2_HIGH; // add appropriate offset
                            PCA0CPL2 = (0xff & temp); // replace compare value
                            PCA0CPH2 = (0xff & (temp >> 8));
                    }
                    else
                    {       // process falling edge
                            // update compare match for next rising edge
                            flag2 =1;
                            temp = (PCA0CPH2 << 8) | PCA0CPL2; // get current compare value
                            temp += PWM2_LOW; // add appropriate offset
                            PCA0CPL2 = (0xff & temp); // replace compare value
                            PCA0CPH2 = (0xff & (temp >> 8));
                    }
```

```
        #elif PCA_CAPT_CH2 > 0 && PCA_CAPTURE_ENABLE > 0
                if(pedge2)
                {   pedge2=0;//now look for falling edge
                    if(find_period2 == 1)
                     {   temp =((PCA0CPH2 << 8) | PCA0CPL2);
                         PERIOD2 = temp - CAPTURE_P2;
                         HIGH_TIME_2=DUTY2;
                         LOW_TIME_2=(PERIOD2-DUTY2);
                         CAPTURE_P2 =temp ; // get current compare value
                         find_period2=0;
                      }
                    else
                     {
                           // get current compare value
                           CAPTURE_P2 = (PCA0CPH2 << 8) | PCA0CPL2;
                           find_period2 =1;
                      }

                     PCA0CPM2 = 0x11;
                }
                else
                {    // get current compare value
                     CAPTURE_N2 = (PCA0CPH2 << 8) | PCA0CPL2;
                     DUTY2 =  CAPTURE_N2 - CAPTURE_P2;
                     pedge2=1;//now look for rising/positive edge
                     PCA0CPM2 = 0x21;
                }
        #endif
}
else if (CCF3)
{
        CCF3 = 0;
        #if PCA_OUT_CH3>0 && PCA_OUT_ENABLE >0
                if (flag3)//(PWM0_OUT)
                {
                    // process rising edge
                    // update compare match for next falling edge
                    flag3 =0;
                    temp = (PCA0CPH3 << 8) | PCA0CPL3; // get current compare value
                    temp += PWM3_HIGH; // add appropriate offset
                    PCA0CPL3 = (0xff & temp); // replace compare value
                    PCA0CPH3 = (0xff & (temp >> 8));
                }
                else
                {          // process falling edge
                           // update compare match for next rising edge
                     flag3 =1;
                     temp = (PCA0CPH3 << 8) | PCA0CPL3; // get current compare value
                     temp += PWM3_LOW; // add appropriate offset
                     PCA0CPL3 = (0xff & temp); // replace compare value
                     PCA0CPH3 = (0xff & (temp >> 8));
                }
        #elif PCA_CAPT_CH3 > 0 && PCA_CAPTURE_ENABLE > 0

                if(pedge3)
                {   pedge3=0;//now look for falling edge
```

```
                        if(find_period3 == 1)
                        {   temp =((PCA0CPH3 << 8) | PCA0CPL3);
                            PERIOD3 = temp - CAPTURE_P3;
                            HIGH_TIME_3=DUTY3;
                            LOW_TIME_3=(PERIOD3-DUTY3);
                            CAPTURE_P3 =temp ; // get current compare value
                            find_period3=0;
                        }
                        else
                        {       // get current compare value
                              CAPTURE_P3 = (PCA0CPH3 << 8) | PCA0CPL3;
                              find_period3 =1;
                         }

                          PCA0CPM3 = 0x11;
                    }
                    else
                    {           // get current compare value
                                CAPTURE_N3 = (PCA0CPH3 << 8) | PCA0CPL3;
                                DUTY3 =  CAPTURE_N3 - CAPTURE_P3;
                                pedge3=1;//now look for rising/positive edge
                                PCA0CPM3 = 0x21;
                    }
          #endif
}
else if (CCF4)
{
          CCF4 = 0;
          #if PCA_OUT_CH4 > 0  && PCA_OUT_ENABLE >0
                    if (flag4)//(PWM0_OUT)
                    {           // process rising edge
                                // update compare match for next falling edge
                        flag4 =0;
                        temp = (PCA0CPH4 << 8) | PCA0CPL4; // get current compare value
                        temp += PWM4_HIGH; // add appropriate offset
                        PCA0CPL4 = (0xff & temp); // replace compare value
                        PCA0CPH4 = (0xff & (temp >> 8));
                    }
                    else
                    {           // process falling edge
                                // update compare match for next rising edge
                         flag4 =1;
                         temp = (PCA0CPH4 << 8) | PCA0CPL4; // get current compare value
                         temp += PWM4_LOW; // add appropriate offset
                         PCA0CPL4 = (0xff & temp); // replace compare value
                         PCA0CPH4 = (0xff & (temp >> 8));
                    }

          #elif PCA_CAPT_CH4 > 0 && PCA_CAPTURE_ENABLE > 0
                    if(pedge4)
                    {   pedge4=0;//now look for falling edge
                                if(find_period4 == 1)
                                {   temp =((PCA0CPH4 << 8) | PCA0CPL4);
                                   PERIOD4 = temp - CAPTURE_P4;
                                   HIGH_TIME_4=DUTY4;
                                   LOW_TIME_4=(PERIOD4-DUTY4);
```

```
                                CAPTURE_P4 =temp ; // get current compare value
                                find_period4=0;
                               }
                               else
                               {
                                 // get current compare value
                                 CAPTURE_P4 = (PCA0CPH4 << 8) | PCA0CPL4;
                                 find_period4 =1;
                                 }


                                 PCA0CPM4 = 0x11;
                        }
                        else
                        {       // get current compare value
                            CAPTURE_N4 = (PCA0CPH4 << 8) | PCA0CPL4;
                            DUTY4 =  CAPTURE_N4 - CAPTURE_P4;
                            pedge4=1;//now look for rising/positive edge
                            PCA0CPM4 = 0x21;
                        }

            #endif
}
else if (CCF5)
{
            CCF5 = 0;
            #if PCA_OUT_CH5 > 0  && PCA_OUT_ENABLE >0
                        if (flag5)//(PWM0_OUT)
                        {       // process rising edge
                                // update compare match for next falling edge
                                flag5 =0;
                                temp = (PCA0CPH5 << 8) | PCA0CPL5; // get current compare value
                                temp += PWM5_HIGH; // add appropriate offset
                                PCA0CPL5 = (0xff & temp); // replace compare value
                                PCA0CPH5 = (0xff & (temp >> 8));
                        }
                        else
                        {       // process falling edge
                                // update compare match for next rising edge
                                flag5 =1;
                                temp = (PCA0CPH5 << 8) | PCA0CPL5; // get current compare value
                                temp += PWM5_LOW; // add appropriate offset
                                PCA0CPL5 = (0xff & temp); // replace compare value
                                PCA0CPH5 = (0xff & (temp >> 8));
                        }

            #elif PCA_CAPT_CH5 > 0  && PCA_CAPTURE_ENABLE > 0
                        if(pedge5)
                        {       pedge5=0;//now look for falling edge
                                if(find_period5 == 1)
                                {       temp =((PCA0CPH5 << 8) | PCA0CPL5);
                                        PERIOD5 = temp - CAPTURE_P5;
                                        HIGH_TIME_5=DUTY5;
                                        LOW_TIME_5=(PERIOD5-DUTY5);
                                        CAPTURE_P5 =temp ; // get current compare value
                                        find_period5=0;
```

```
                                        }
                                        else
                                        {       // get current compare value
                                                CAPTURE_P5 = (PCA0CPH5 << 8) | PCA0CPL5;
                                                find_period5 =1;
                                        }
                                        PCA0CPM5 = 0x11;
                                }
                                else
                                {       // get current compare value
                                        CAPTURE_N5 = (PCA0CPH5 << 8) | PCA0CPL5;
                                        DUTY5 =  CAPTURE_N5 - CAPTURE_P5;
                                        pedge5=1;//now look for rising/positive edge
                                        PCA0CPM5 = 0x21;
                                }
                        #endif
                }
                else if (CF)
                {
                        CF = 0;
                }
                 SFRPAGE= SFRPAGE_save;
}

#if PCA_OUT_ENABLE > 0

unsigned char set_output (unsigned long payload, unsigned char channel)
{

                unsigned int ch0_high_time=0;
                unsigned int ch0_low_time =0;

                unsigned int ch1_high_time=0;
                unsigned int ch1_low_time =0;

                unsigned int ch2_high_time =0;
                unsigned int ch2_low_time =0;

                unsigned int ch3_high_time =0;
                unsigned int ch3_low_time =0;

                unsigned int ch4_high_time=0;
                unsigned int ch4_low_time =0;

                unsigned int ch5_high_time=0;
                unsigned int ch5_low_time =0;

                switch(channel)
                {
                        case 0:
                                ch0_high_time = (unsigned int)((payload&0xffff0000)>>16);
                                ch0_low_time = (unsigned int)((payload&0x0000ffff));
                                set_high_low(ch0_high_time, ch0_low_time,0);
                                break;
                        case 1:
                                 ch1_high_time =(unsigned int)((payload&0xffff0000)>>16);
```

```
                    ch1_low_time =(unsigned int)((payload&0x0000ffff));
                    set_high_low(ch1_high_time,ch1_low_time ,1);
                    break;
            case 2:
                    ch2_high_time = (unsigned int)((payload&0xffff0000)>>16);
                    ch2_low_time = (unsigned int)((payload&0x0000ffff));
                    set_high_low(ch2_high_time, ch2_low_time,2);
                    break;
            case 3:
                    ch3_high_time =(unsigned int)((payload&0xffff0000)>>16);
                    ch3_low_time =(unsigned int)((payload&0x0000ffff));
                    set_high_low( ch3_high_time,ch3_low_time,3);
                    break;
            case 4:
                    ch4_high_time = (unsigned int)((payload&0xffff0000)>>16);
                    ch4_low_time = (unsigned int)((payload&0x0000ffff));
                    set_high_low( ch4_high_time,ch4_low_time,4);
                    break;
            case 5:
                    ch5_high_time =(unsigned int)((payload&0xffff0000)>>16);
                    ch5_low_time =(unsigned int)((payload&0x0000ffff));
                    set_high_low(ch5_high_time ,ch5_low_time ,5);
                    break;
            default:
                    return PCA_OUT_CHANNEL_INVALID;
                    break;
        }
        return 0; //no error;
}

#endif


#if PCA_CAPTURE_ENABLE > 0

unsigned char get_capture (unsigned long *ch_data, unsigned char channel)
{
        switch (channel)
        {
            case 0:
                    *ch_data = (((unsigned long)HIGH_TIME_0) <<16) | ((unsigned
                    long)LOW_TIME_0);
                    break;
            case 1:
                    *ch_data = (((unsigned long)HIGH_TIME_1) <<16) | ((unsigned
                    long)LOW_TIME_1);
                    break;
            case 2:
                    *ch_data = (((unsigned long)HIGH_TIME_2) <<16) | ((unsigned
                    long)LOW_TIME_2);
                    break;
            case 3:
                    *ch_data = (((unsigned long)HIGH_TIME_3) <<16) | ((unsigned
                    long)LOW_TIME_3);
                    break;
            case 4:
```

```
                        *ch_data = (((unsigned long)HIGH_TIME_4) <<16) | ((unsigned
                        long)LOW_TIME_4);
                        break;
                case 5:
                        *ch_data = (((unsigned long)HIGH_TIME_5) <<16) | ((unsigned
                        long)LOW_TIME_5);
                        break;
                default:
                        return PCA_CAPT_CHANNEL_INVALID;
                        break;
        }
        return 0; // no error;
}
#endif

#endif //end of PCA_ENABLE macro
```

Project Source:

OS_MAIN.h

```
/*********************************************************************************************
*       Filename:       OS_MAIN.h
*       Version:        V1.0
*       Author:         Darren Brown
*                       © Copyright 2006, University of Kentucky
*                       All Rights Reserved
*
*********************************************************************************************
*/

#ifndef  OS_main_h
#define OS_main_h



#include "..\OS_config\includes.h"
#include "..\OS_port\os_kcdef.h"



// Semaphores

extern OS_EVENT      *UARTSem;
extern OS_EVENT      *Send_Msg_Sem;
extern OS_EVENT      *Get_Msg_Sem;
extern OS_EVENT      *Accept_Msg_Sem;


#endif  //OS_main_h
```

OS_MAIN.c

```
/*********************************************************************************************
*       Project:        Darren Brown's IDEAnix with MeRL
*       Filename:       OS_MAIN.c
*       Version:        V1.0
*       Author:         Darren Brown
*                       © Copyright 2006, University of Kentucky
*                       All Rights Reserved
*
*********************************************************************************************
*/


#include "OS_main.h"
#include "..\ideanix\ideanix_init.h"
#include "..\ideanix\ideanix_config.h"
#include "..\ideanix\ideanix_isr.h"
#include "..\ideanix\ideanix_msg_handler.h"
#include "..\ideanix\ideanix_task_msg.h"
#include "..\ideanix\ideanix_servo.h"


/*

*********************************************************************************************
*                                   VARIABLES
*********************************************************************************************
*/

OS_STK          TaskStk[NUM_TASKs][TASK_STK_SIZE];     /* Tasks stacks */


/*********************************************************************************************
*                                   OS EVENTS
*********************************************************************************************/

// Semaphores

OS_EVENT        *UARTSem;
OS_EVENT        *Send_Msg_Sem;
OS_EVENT        *Get_Msg_Sem;
OS_EVENT        *Accept_Msg_Sem;

void *null_ptr;

/*********************************************************************************************
*                                   FUNCTION PROTOTYPES
*********************************************************************************************
*/

void  TaskFirst_task(void *dataa) KCREENTRANT;       // Function prototypes of Startup task
```

```
/*********************************************************************************************
*                                      MAIN
*********************************************************************************************
*/


void main (void)

{

  char SFRPAGE_SAVE = SFRPAGE; // Save Current SFR page
  unsigned char error_ck =0 ;


   EA = 0;                                                    // disable global interrupts


WDTCN = 0xde;                    // disable watchdog timer
WDTCN = 0xad;


init_all ();

OSInit();   /* Initialize uC/OS-II*/


//  The highest priority task is the TASK WHICH STARTS OFF ALL THE CHILDREN TASK
//  It is important to note as is stated on page of 484 of the uC OS II book under the OSTaskCreate()
// section ONE SHOULD NOT USE task priorities 0,1,2,3, OS_LOWEST_PRIO-3, OS_LOWEST_PRIO -
//2,  OS_LOWEST_PRIO-1, OS_LOWEST_PRIO because they are reserved for use by uC/OS-II.
//This leaves  you with up to 56 application task at most.  IT IS NOT A GOOD IDEA TO USE ALL 56
 // tasks because of scheduling and starvation issues. Because of the recommendations found within the uC
//OS II reference book- 4 is the highest priority task allowable

    EA = 1;     // enable global interrupts


//create Semaphores

            UARTSem  = OSSemCreate(1);    // create a UART semaphore

            //used by MERL

            #if TASKCOMM_EN == 1
                Send_Msg_Sem  = OSSemCreate(1);
                Get_Msg_Sem = OSSemCreate(1);
                Accept_Msg_Sem = OSSemCreate(1);
            #endif

            #if NETCOMM_EN ==1
                 error_ck=   OSTaskCreate(network_router, (void *) 0,&TaskStk[0][TASK_STK_SIZE], 4);
             #endif

            error_ck=   OSTaskCreate(TaskFirst_task, (void *) 0,&TaskStk[1][TASK_STK_SIZE], 5);
```

```
        SFRPAGE=UART0_PAGE;

        printf("#### STARTING OS ######.\r\n");

  // start OS

  OSStart();  // Start multitasking

}


/*********************************************************************************************
*                                   First (STARTUP) TASK
*********************************************************************************************
*/


void TaskFirst_task (void *dataa) KCREENTRANT

{

        char SFRPAGE_SAVE = SFRPAGE; // Save Current SFR page
        unsigned int id;
        unsigned long payload;
        unsigned long ch_data;
        unsigned char error;


        dataa = dataa;

        OSSemPend(UARTSem,0,&error);

                SFRPAGE=UART0_PAGE;

        printf("#########   The OS version %u is up!, All processes started
############.\r\n",OSVersion());

        OSSemPost(UARTSem);

        if( init_messaging () ==1 )
        {
                OSSemPend(UARTSem,0,&error);
                SFRPAGE=UART0_PAGE;
                        printf("messaging is enabled \r\n");
                OSSemPost(UARTSem);
        }
        else
        {
                OSSemPend(UARTSem,0,&error);
                SFRPAGE=UART0_PAGE;
                        printf("Messaging is NOT enabled \r\n");
                OSSemPost(UARTSem);
        }
```

```
        #if PCA_OUT_ENABLE == 1
        reg_msg (1);
        reg_msg (2);
        reg_msg (3);
        reg_msg (4);
        reg_msg (5);
        reg_msg (6);

#endif

        for(;;)
          {
                #if PCA_OUT_ENABLE == 1   &&  PCA_ENABLE == 1
                        get_msg (&id, &payload );

                        switch(id)
                        {
                                case 1:
                                        set_output(payload,0);
                                        break;
                                case 2:
                                        set_output(payload,1);
                                        break;
                                case 3:
                                        set_output(payload,2);
                                        break;
                                case 4:
                                        set_output(payload,3);
                                        break;
                                case 5:
                                        set_output(payload,4);
                                        break;
                                case 6:
                                        set_output(payload,5);
                                        break;
                        }


                #endif


              #if PCA_CAPTURE_ENABLE == 1 &&  PCA_ENABLE == 1
                        #if PCA_CAPT_CH0 == 1
                                get_capture (&ch_data, 0);
                                send_msg (1 , ch_data );
                        #endif

                        #if PCA_CAPT_CH1 == 1
                                get_capture (&ch_data, 1);
                                send_msg (2 , ch_data);
                        #endif

                        #if PCA_CAPT_CH2 == 1
                                get_capture (&ch_data, 2);
```

```
                send_msg (3 , ch_data );
            #endif

            #if PCA_CAPT_CH3 == 1

                get_capture (&ch_data, 3);
                send_msg (4 , ch_data);
            #endif


            #if PCA_CAPT_CH4 == 1
                get_capture (&ch_data, 4);
                send_msg (5 , ch_data );
            #endif


            #if PCA_CAPT_CH5 == 1
                get_capture (&ch_data, 5);
                send_msg (6 , ch_data);
            #endif


            OSTimeDly(5);

        #endif

    }


} //TaskFirst_task
```

REFERENCES

1. **Gartner, Felix C.** Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments. *ACM Computing Surveys, Vol. 31, No. 1* . March 1999.

2. **Paul Pop, Petru Eles, Zebo Peng, and Traian Pop.** Analysis and Optimization of Distributed Real-Time Embedded Systems. s.l. : ACM Inc., July 2006. Vol. 11, No. 3, pp. 593-625.

3. The OSI Model . *Wikipedia.* [Online] March 26, 2010. [Cited: January 12, 2008.] http://en.wikipedia.org/w/index.php?title=OSI_model&oldid=298710907.

4. Windows XP Professional System Requirements. *Microsoft.* [Online] Microsoft, August 24, 2001. [Cited: January 12, 2008.] http://www.microsoft.com/windowsxp/sysreqs/pro.mspx.

5. Micrium uC/OS-II Ports - Intel. *A Micrium Web site.* [Online] Micrium, 2010. [Cited: March 27, 2010.] http://micrium.com/page/downloads/ports/intel.

6. **Darren Brown, Tim Arrowsmith, Alicia Mylin, Robert Koontz Adam Fox, Neal Phelps, Daniel.** "AIRCAT: Airborne Intelligent Research Craft for Autonomous Technology". *AUVSI Student.* 2005.

7. **Darren Brown, Craig Collins, Dale McClure, Ala Aldin Meriden, Phillip Profitt, Matt Smith, and.** "University of Kentucky Aerial Robotics Team: 2006 AUVSI Student UAV Competition Design.". *2006 AUVSI Student UAV Competition Paper.* 2006.

8. **Laboratories, Silicon.** C8051F040 Datashort. *Silicon Laboratories Products - http://www.silabs.com.* s.l. : Silicon Laboratories, May 5, 2005.

9. **Silberschatz, Abraham, Gagne, Greg and Galvin, Peter Baer.** Chapter 19: Real-Time Systems. *Operating System Concepts.* 7th Edition. Danvers, MA : John Wiley & Sons, Inc., 2005, 19, p. 696.

10. **Liu, Jane W.S.** Chapter 2: Hard versus Soft Real-Time Systems. *Real-Time Systems.* Upper Saddle River, New Jersey : Prentice-Hall, Inc., 2000, 2, p. 30.

11. —. *Real-Time Systems.* Upper Saddle River, New Jersey : Prentice-hall Inc. , 2000. 0-13-099651-3.

12. Intelligent Embedded Dependable Architectures Lab. [Online] University of Kentucky, 2005-2006. [Cited: March 27, 2010.] http://www.engr.uky.edu/idea.

13. **A. Simpson, O. Rawashdeh, S. Smith, J. Jacob, W. Smith, and J. Lumpp.** "BIG BLUE: A High-Altitude UAV Demonstrator of Mars Airplane Technology". Big Sky, Montana : IEEE Aerospace, March 2005. IEEEAC paper #1436.

14. **O. Rawashdeh, D. Feinauer, C. Harr, G. Chandler, D. Jackson, A. Groves, and J. Lumpp.** "A Dynamically Reconfiguring Avionics Architecture for UAVs". *AIAA Infotech@Aerospace Conference.* Arlington, Virginia, Virginia : AIAA, September 2005. AIAA-2005-7050.

15. **Rawashdeh, Osamah A.** ARDEA: A Reconfigurable Architecture for Fault Tolerant Distributed Embedded Systems. *Ph.D. Dissertation.* Lexington , KY : Department of Electrical and Computer Engineering, University of Kentucky, December 2005.

16. **AUVSI.** AUVSI Student Competitions. *AUVSI Student Competitions.* [Online] Association for Unmanned Vehicle Systems International. [Cited: March 27, 2010.] http://www.auvsi.org/AUVSI/AUVSI/Events/AUVSIStudentCompetitions/Default.aspx.

17. **Dale McClure, Nate Rhodes, Shadab Ambat, Yehya Bekheet.** University of Kentucky Aerial Robotics Team: 2007 AUVSI Student UAV Competition Design. *2007 AUVSI Student UAV Competition Paper.* Lexington, KY : AUVSI, 2007.

18. *A Reliable Reconfigurable Bus for Light Unmanned Aircraft.* **Brown, D.J., et al.** May 2007. AIAA Infotech@Aerospace Conference. Paper# AIAA-2007-2959.

19. **National Aeronautics and Space Administration.** NASA Systems Engineering Handbook. June 1995. SP-610S.

20. *"Guest Editors' Introduction: Fault-Tolerant Embedded Systems".* **Avresky, Dimiter R., et al.** 5, s.l. : IEEE, Sept/Oct 2001, IEEE Micro, Vol. 21, pp. 12-15. 0272-1732/01.

21. *"Component Based Design of Multitolerant Systems".* **Arora, Anish and Kulkarni, Sandeep S.** 1, s.l. : IEEE, January 1998, IEEE Transactions on Software Engineering, Vol. 24, pp. 63-78. 10.1109/32.663998.

22. *"The Byzantine Generals Problem".* **Lamport, Leslie, Shostak, Robert and Pease, Marshall.** 3, s.l. : ACM, July 1982, ACM Transactions on Programming Languages and Systems, Vol. 4, pp. 382-401. 0164-0925/82/0700-0382.

23. *Understanding Fault-Tolerant Distributed Systems.* **Cristian, Flavin.** 2, New York, NY : ACM, February 1991, Communications of the ACM, Vol. 34. 0001-0782 .

24. Definition of Synchronous. *Webster's Online Dictionary.* [Online] Philip M. Parker, 2010. [Cited: March 27, 2010.] http://www.websters-online-dictionary.org/definition/synchronous.

25. **Chandy, K. Mani and Misra, Jayadev.** How Processes Learn. *Annual ACM Symposium on Principles of Distributed Computing.* Proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Computing, 1985, pp. 204-214.

26. *Consensus in the Presence of Partial Synchrony.* **Dwork, Cynthia, Lynch, Nancy and Stockmeyer, Larry.** 2, s.l. : ACM, April 1988, Journal of the ACM, Vol. 35, pp. 288-323. 0004-5411.

27. **Provan, Gregory and Yi-Liang, Chen.** Model-Based Fault-Tolerant Control Reconfiguration for General Network Topologies. *IEEE Micro.* Los Alamitos, CA : IEEE Computer Society Press, September 2001. Vol. 21, 5, pp. 64-76. 0272-1732 .

28. **Rushby, John.** A Comparison of Bus Architectures for Safety-Critical Embedded Systems. *Computer Science Laboratory Technical Report.* Menlo Park, CA : SRI International, June 2002.

29. Transmission Control Protocol. *A Wikipedia Website.* [Online] Wikipedia.org. [Cited: March 27, 2010.] http://en.wikipedia.org/wiki/Transmission_Control_Protocol.

30. **Koopman, Phil, Hendrey, Geoff and Tran, Eushiuan.** Embedded Network Performance & Robustness. [Online] [Cited: March 27, 2010.] http://www.ece.cmu.edu/~koopman/networks/embedded_network_brochure.pdf.

31. *"A Columbus' Egg Idea for CAN Media Redundancy".* **Rufino, Jose, Verissimo, Paulo and Arroz, Guilherme.** Madison, Wisconsin : s.n., 1999. Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing. pp. 286-293. 0-7695-0213-X.

32. **Intel Corporation.** "Chapter 5: Error Handling Means Reliability". *CAN Overview for the Intel 88C0196EC Embedded Microcontroller- Application Note.* s.l. : Intel Corporation, March 2004. 301706-001.

33. Mesh Networking. *Wikipedia.org.* [Online] April 2, 2010. [Cited: April 3, 2010.] http://en.wikipedia.org/wiki/Mesh_networking.

34. **ZigBee Alliance.** ZigBee and Wireless Radio Frequency Coexistence. *ZigBee White Paper.* s.l. : ZigBee Alliance - www.zigbee.org, June 2007.

35. **Intel Corporation.** "Chapter 3: Where Does CAN Fit In?". *CAN Overview For the Intel 88C0196EC Embedded Microcontroller- Application Note.* s.l. : Intel Corporation, March 2004. 301706-001.

36. **Freescale Semiconductor, Inc.** "Relating the Zigbee Network and the IEEE 802.15.4 Models to the OSI Communication Model". Rev. 0.0, 09/2007 *IEEE 802.15.4 / Zigbee Software Selector Guide - Application Note.* s.l. : Freescale Semiconductor, September 2007. AN3403.

37. **Silberschatz, Abraham, Gagne, Greg and Galvin, Peter.** Chapter 8: Main Memory. *Operating System Concepts.* Hoboken : John Wiley & Sons, Inc., 2005, p. 296.

38. **Jeganathan, Nithyananda Siva.** "A Controller Area Network Layer For Reconfigurable Embedded Systems". *University of Kentucky College of Engineering Thesis.* Lexington, KY : University of Kentucky, 2007.

39. **Micrium.** "uC/OS-II Real-Time Operating System". *Micrium Datasheet for uC/OS-II.* s.l. : Micrium, May 2009.

40. **Labrosse, Jean J.** Preface. *MicroC/OS-II The Real-Time Kernel.* 2nd Edition. San Francisco : CMP Books, 2002, Preface, p. XV.

41. **Silicon Laboratories.** C8051F040/1/2/3/4/5/6/7 Mixed Signal ISP Flash MCU Family Datasheet. *http://www.silabs.com/products/mcu/mixed-signalmcu/Pages/C8051F04x.aspx.* [PDF]. s.l. : Silicon Labs, December 2005. Rev. 1.5.

42. **Arrowsmith, Timothy W.** A NETWORK PROCESSING NODE FOR LIGHT UNMANNED AIRCRAFT. s.l. : University of Kentucky, 2007.

43. **U.S. House of Representatives.** Systems Development Life-Cycle Policy. *http://www.house.gov/cao-opp/ref-docs/SDLCPOL.pdf.* [PDF]. s.l. : U.S. House of Representatives, March 24, 1999.

44. **Department of the Air Force - Software Technology Support Center.** Guidelines for Successful Acquisition and Management of Software-Intensive Systems: Weapon Systems Command and Control Systems Management Information Systems. Version 3.0 *http://www.stsc.hill.af.mil/resources/tech_docs/.* [PDF]. s.l. : United States Air Force, May 2000.

45. **Services, Centers for Medicare & Medicaid.** Selecting a Development Approach. *http://www.cms.gov/SystemLifecycleFramework/Downloads/SelectingDevelopmentApproach.pdf.* [PDF]. s.l. : U.S. Department of Health & Human Services , March 27, 2008.

46. **Fabrycky, Blanchard and.** *Systems Engineering and Analysis.* Fourth Edition. s.l. : Prentice Hall, 2006.

47. **Labrosse, Jean J.** *MicroC/OS-II The Real-Time Kernel.* [ed.] Robert Ward. 2nd Edition. San Francisco : CMP Books, 2002. 1-57820-103-9.

48. **Bosch, Robert GmbH.** *CAN Specification Part A and B.* Version 2.0. Stuttgart, Germany : s.n., 1991.

49. *Time, Clocks, and the Ordering of Events in a Distributed System.* **Lamport, Leslie.** No. 1, s.l. : ACM, March 1999, ACM Computing Surveys, Vol. Vol. 31.

50. *Response Time Analysis of Asynchronous Real-Time Systems.* **Bernat, Guillem.** s.l. : Kluwer Academic Publishers, 2003, Real-Time Systems, Vol. 25, pp. 131-156.

VITA

Darren Jacob Brown

Personal

Birthplace: Bowling Green, KY
Date of Birth: June 22, 1983

Education

Bachelor of Science (B.S.) in Electrical Engineering, December 2005
Department of Electrical and Computer Engineering
University of Kentucky, Lexington, KY, USA

Bachelor of Science (B.S.) in Computer Science, December 2005
Department of Computer Science
University of Kentucky, Lexington, KY, USA

Work Experience

Bigelow Aerospace LLC, Las Vegas, NV
- Deputy Avionics Hardware Manager
  12/09 – Present
- Interim Avionics Hardware Manager
  06/09 – 12/09
- Lead Embedded Systems Engineer
  01/07 – 06/09

Bigelow Aerospace Advanced Space Studies, LLC, Las Vegas, NV
- Project Manager
  06/09 – 09/09
- Electrical Engineer
  01/09 – 06/09

Intelligent Dependable Embedded Architectures (IDEA) Laboratory
University of Kentucky, Lexington, KY
- Research Assistant
  01/05 – 12/06

Precision Systems and Instrumentation, LLC, Lexington, KY
- Embedded Systems / Electrical Engineer
  01/05 – 12/06

customKYnetics, Inc., Versailles, KY
- Electrical Engineer – Co-op
  01/04 – 8/04

Ashland Inc., Lexington, KY
- Domain Administrator – Co-op
  05/03 – 08/03

Honors

University Scholars Program
College of Engineering Academic Scholarship
University of Kentucky Merit Scholarship
Dean's List
Tau Beta Pi – Engineering Honor Society
Eta Kappa Nu – Electrical Engineering Honor Society
Upsilon Phi Epsilon – Computer Science Honor Society

Publications

Darren Brown, Tim Arrowsmith, Alicia Mylin, Robert Koontz Adam Fox, Neal Phelps, Daniel. "AIRCAT: Airborne Intelligent Research Craft for Autonomous Technology". AUVSI Student UAV Competition Paper. 2005.

Darren Brown, Craig Collins, Dale McClure, Ala Aldin Meriden, Phillip Profitt, Matt Smith. "University of Kentucky Aerial Robotics Team: 2006 AUVSI Student UAV Competition Design.". 2006 AUVSI Student UAV Competition Paper. 2006.

Darren Brown, Tim Arrowsmith, O.A. Rawashdeh, James E. Lumpp Jr. "A Reliable Reconfigurable Bus for Light Unmanned Aircraft." May 2007. AIAA Infotech@Aerospace Conference. Paper# AIAA-2007-2959.