

ABSTRACT

Development and Implementation of Real-time Distributed Network with the CAN Protocol

Walter D. Ford

Mentor: Ian A. Gravagne, Ph.D.

One of the most interesting applications of a new field of mathematics called dynamic equations on time scales is the modeling, analysis and design of distributed control networks. This thesis documents the development of a scalable, real-time test bed on which to test new time scale-based theories. The Controller Area Network (CAN) protocol is used as the communication backbone. A general description of CAN and reasons for its selection are included. A general purpose computational node is implemented on a desktop computer running the QNX real-time operating system. QNX development entailed writing a driver for an SJA1000-based CAN controller. Charmed Labs' Xport and the Gameboy Advance (GBA) are used for the network of embedded nodes. Development for the GBA-Xport combination involved interfacing an OpenCores.org CAN core to the Xport's bus on an FPGA in Verilog and writing a driver class. Appendices include the code and code documentation.

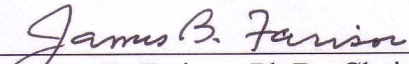
Development and Implementation of Real-time Distributed Network
with the CAN Protocol

by

Walter D. Ford, B.S.E.C.E.

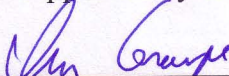
A Thesis

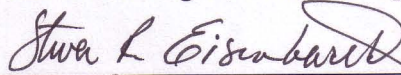
Approved by the Department of Electrical and Computer Engineering

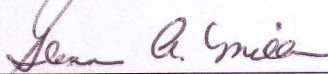

James B. Farison, Ph.D., Chairperson

Submitted to the Graduate Faculty of
Baylor University in Partial Fulfillment of the
Requirements for the Degree
of
Master of Science in Electrical and Computer Engineering

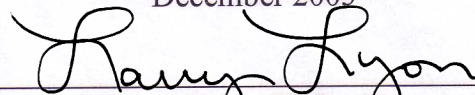
Approved by the Thesis Committee


Ian A. Gravagne, Ph.D. Chairperson


Steven R. Eisenbarth, Ph.D.


Glenn A. Miller, Ph.D.

Accepted by the Graduate School
December 2005


J. Larry Lyon, Ph.D., Dean

Copyright © 2005 by Walter D. Ford

All rights reserved

TABLE OF CONTENTS

LIST OF FIGURES	vi
ACKNOWLEDGMENTS	vii
DEDICATION	viii
CHAPTER ONE	1
Introduction	
CHAPTER TWO	4
General Description of CAN	
CHAPTER THREE	10
Technology Used in CAN Bus Implementation	
<i>QNX Real-Time Operating System</i>	10
<i>Scalable Architecture</i>	11
<i>Resource Managers</i>	12
<i>Interrupts</i>	12
<i>Arcom AIM104-CAN Controller Board</i>	14
<i>Nintendo Gameboy Advance SP</i>	15
<i>CharmedLabs Xport 2.0 and XRC</i>	15
<i>Xilinx Spartan II FPGA</i>	16
<i>Xport Robot Controller</i>	16
<i>CharmedLabs Open-Source Libraries</i>	17
<i>OpenCores.org CAN Core</i>	18
<i>Microchip MCP2551 CAN Transceiver</i>	19
CHAPTER FOUR	22
QNX Driver Development	
<i>SJA1000 Access Functions</i>	22
<i>Interrupt Handler</i>	23
<i>I/O and Connect Functions</i>	24
<i>can4qnx Resource Manager</i>	25
CHAPTER FIVE	27
Xport CAN Controller Development	
<i>Bus Description and Behaviour</i>	27
<i>GBA to OpenCores.org CAN Controller Bus Interface</i>	29
<i>CAN Core-GBA Interrupt Interface</i>	31
<i>Hardware Reset Register</i>	32
<i>CAN Transceiver</i>	33
<i>CAN Core Library Class</i>	33
<i>Conclusion</i>	35
APPENDIX A	37
Appendix Documenting can4qnx Code	
<i>QNX SJA1000 Access Functions</i>	37
<i>Interrupt Handler</i>	40
<i>I/O and Connect Functions</i>	41

<i>can4qnx Resource Manager</i>	44
<i>Miscellaneous Files and Functions</i>	45
APPENDIX B	48
Appendix Documenting CCanCore Code	
<i>CAN Core Class Access Functions</i>	48
<i>Interrupt Handler</i>	52
<i>Other Files</i>	54
APPENDIX C	55
Appendix Documenting Verilog Source	
<i>CAN Interface Inputs and Outputs</i>	55
<i>GBA Inputs and Ouptuts</i>	55
<i>CartData to address Addr</i>	55
<i>CAN Transceiver Inputs and Ouptuts</i>	55
<i>Troubleshooting Outputs</i>	56
<i>CAN Interface Module Summary</i>	56
<i>OpenCores.org CAN Core</i>	56
<i>CAN Core Reset</i>	57
<i>IRQ Interface</i>	57
<i>Bus Interface Combinational Logic</i>	57
<i>Bus Interface State Machine</i>	58
<i>Xport Primary Module</i>	58
<i>CAN Controller Module</i>	58
<i>Other Modules</i>	58
APPENDIX D	60
can4qnx Source Code	
can_close.c	60
can_defs.h	60
can_devctl.c	64
can_error.c	68
can_open.c	69
can_read.c	69
can_resmgr.c	70
can_sja1000.h	72
can_sja1000funcs.c	77
can_sysctl.c	86
can_util.c	87
can_write.c	90
can4qnx.h	91
APPENDIX E	95
Verilog Source Code	
CAN.v	95
can_bot.v	97
APPENDIX F	101
CCanCore Source Code	
ccancore.cxx	101
ccancore.h	113

gbaCAN.h	114
BIBLIOGRAPHY	120

LIST OF FIGURES

Figure 1: CAN Nodes	5
Figure 2: Differential CAN Signal.....	6
Figure 3: TTL to Differential CAN	19
Figure 4: GBA-Xport Block Diagram	20
Figure 5: CAN Node Picture.....	21
Figure 6: Resource Manager Block Diagram	26
Figure 7: Write to CAN Address.	28
Figure 8: Read from CAN Address	28
Figure 9: CAN Core Bus Interface.	30
Figure 10: CAN Interface State Diagram	31
Figure 11: Hardware Reset	33
Figure 12: CAN Transceiver Schematic	34
Figure 13: CAN Transceiver Picture	34

ACKNOWLEDGMENTS

I would like to thank several people for their involvement in this project. I especially want to thank Dr. Russell Duren without whose help the Verilog portions of this project would have been overwhelming. I would also like to thank Dr. Steven Eisenbarth for his advice, assistance, and patience in the real-time programming aspects of this project. Finally, I would like to thank Dr. Ian Gravagne for the privilege of assisting him in this research.

DEDICATION

TO

Rev. Tom Gibbs and Rev. Pete Hatton
for their sacrificial involvement in Reformed
University Fellowship at Baylor University

TO

Mr. Carol Stubbs
who taught me the value of solid, Biblical teaching

TO

My Parents
whose marriage continues to be the best picture of
Christ and the Church I have ever witnessed

CHAPTER ONE

Introduction

A recent grant to the Baylor Electrical and Computer Engineering department from the National Science Foundation (NSF) created the need for the innovation documented in this thesis. The proposed research explores the possibility of using dynamic equations on time scales to help increase the efficiency of distributed control networks [19]. This possibility would allow network busses more flexibility in message timing, thereby increasing total bus throughput. In the words of Baylor Engineering's NSF proposal, this research would explore

“how real-time distributed control systems can more effectively share available resources, predict certain types of control-related faults and instabilities, and permit designers to maximize the number of nodes that operate on the network.”

As the utilization of electromechanical sensors and actuators increases, the ability to communicate with them in real time and in rising numbers will become increasingly important.

The Controller Area Network (CAN) protocol was selected as the means for testing new time scales theories. The CAN protocol was chosen for several reasons. CAN busses use real-time message transmission, which is important when studying the timing characteristics of time scales. CAN is fast enough for most control applications. This makes CAN a practical choice for controlling physical systems in various applications. CAN is also a very flexible protocol. It is easy to add and remove CAN nodes from a bus or alter the way a CAN node behaves without disrupting

communication on the entire bus. This ability to develop and test a few CAN nodes and then scale up to dozens is convenient.

In addition to these characteristics CAN is already a widely used standard for real-time, distributed controller networks. As such, support, components, and documentation are readily available at low cost. Although CAN is an established protocol, development continues on many CAN standards and devices. Communities supporting and actively contributing to the CAN protocol are helpful in the development process. Standards also exist for evaluating bus timing and utilization. These standards will be useful in evaluating time scales effects and as benchmarks for traditional systems.

The NSF proposal outlines a specific test bed and specific testing procedures for testing time scales theories. The proposal calls for a CAN bus of 40 nodes. Desktop computers will monitor the network in real-time and also inject controlled, experimental traffic to monitor the traffic's effects on the bus. The CAN controllers developed by the author for this thesis are that test bed. Primarily, this thesis describes and documents the development and implementation of the two types of CAN controllers that will be used for this experimentation. Development of the CAN controllers for real-time desktop computers that monitor the network are presented in Chapter Four. Development of the CAN controllers for the functional nodes of the CAN bus are presented in Chapter Five. Fundamentals of the two systems used for this implementation are discussed as they relate to the CAN controllers. Characteristics of the CAN protocol that make it a desirable medium for these experiments are described in the next chapter. These CAN controllers' ability to meet the time scales testing requirements and to be implemented on

two powerful platforms makes their development and implementation an ideal subject for this thesis.

CHAPTER TWO

General Description of CAN

CAN is the predominant real-time networking solution in a variety of industrial fields. CAN was initially developed in the 1980s by Robert Bosch GmbH for automotive applications [10]. Today CAN has been recognized worldwide as a standard, and it, or a derivative of it, is used in the majority of high-end European automobiles. CAN is also an important communication protocol in industrial applications, avionics, and safety critical applications.

One interesting characteristic of the Bosch CAN specification is its omission of a communication medium. This omission allows for a variety of CAN bus media. Any medium that clearly possesses dominant and recessive states can be utilized as a CAN bus. CAN busses have been implemented on differential voltage wire pairs, single wires with a ground, and even wireless transmitters [14]. The International Standards Organization (ISO) standardized a 5V differential electrical bus for up to 1Mbit per second CAN communication in ISO11898-2 [1]. This is the most common implementation of a CAN bus.

Controller Area Networks are made up of CAN nodes that interface with the bus. Figure 1 depicts four CAN nodes connected to a CAN bus. Each of these nodes is an independent CAN controller. Every node sees every message, or frame, that is transmitted on the bus and acknowledges the message if received correctly. Several

characteristics make CAN the most suitable test bed for real-time distributed control research.

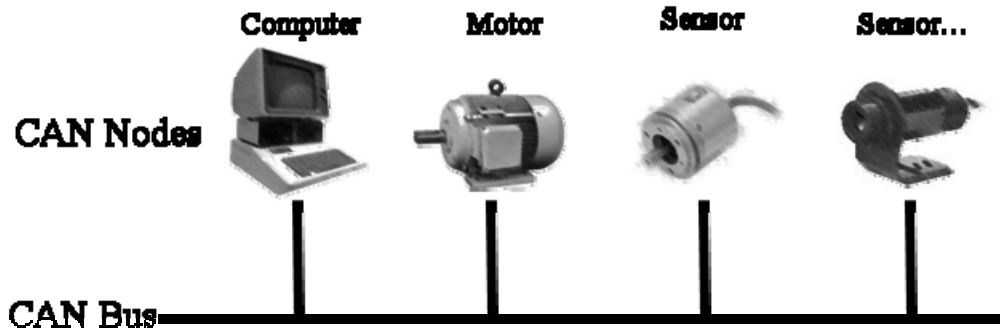


Figure 1: CAN Nodes – Multiple CAN nodes with various functionality all connect to a single CAN bus to control and analyze a physical system.

The first important characteristic of CAN that qualifies it as an appropriate medium for this research is the inherent prioritization in the network's architecture [1], [2]. Figure 2 shows the dominant and recessive states of a differential voltage CAN signal. CAN is a multi-master protocol, so any node may send a message at any time as long as the bus is free. Nodes determine whether or not the bus is free by monitoring the state of the bus at all times. If the node senses activity on the bus, it receives the incoming message. If a node senses a dominant state on the bus while it is transmitting a recessive state, it must immediately stop transmitting and process the incoming message with the higher priority. When two nodes contend for the bus, the message with the highest priority is always transmitted first. This is accomplished with a clever bitwise arbitration using the identifier of the CAN frame.

Since CAN is used in safety-critical applications, it is important that messages arrive within a specified amount of time. Guaranteed latency is also important for evaluating time scale theories because the nature of the experiments is sensitive to timing.

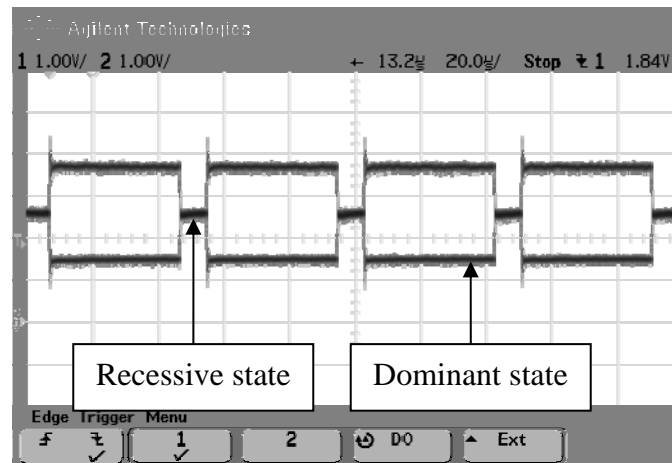


Figure 2: Differential CAN Signal – This differential CAN signal demonstrates the dominate and recessive bus states. The two wires float to 2.5V when the two bus wires are not being driven 2V apart. The top two bus wires are placed on top of one another to highlight the differential nature of the signal as it is seen on the “CAN HI” and “CAN LO” wires on the CAN bus.

The inherent prioritization of CAN frames ensures that critical messages will be sent ahead of less important messages. A critical message may be sent as long as the bus is not currently being accessed. If another node is transmitting, a critical message need only wait until the message currently on the bus has finished before gaining bus access.

The maximum latency of a critical message is only impacted by two factors. In the worst case scenario, a critical message needs to be sent just after a less critical message has been put on the bus. The critical message only needs to wait the length of one CAN message, since the critical message’s priority ensures that it will be transmitted as soon as the current message has finished. The second factor that may delay a critical message from being sent is bit stuffing. If more than five consecutive bits of the same polarity occur in a message, the node must insert one bit of opposite polarity. This “stuffed” bit contains no information and is used solely to synchronize the timing between the transmitting node and receiving nodes. Once received, the stuffed bit is

discarded. These two factors and the interframe space of a few bit times combine to give the maximum latency for a critical message.

CAN is a message-based protocol as opposed to an address-based protocol [2]. The fact that every message contains all relevant priority and data information makes CAN an ideal way to study timing characteristics of a network. A CAN network is very flexible as well. Since all nodes receive, acknowledge, and decide whether or not to process a message, it is trivial to increase the number of nodes on a CAN bus. This is important for the test bed being developed since development and testing are easier with a small number of nodes, while experimentation requires a large number of nodes. Another useful implication of a message-based network is the ability to “multicast” one message to many or all of the nodes at once. This is useful in bus reconfiguration and when it is necessary to communicate to multiple nodes.

Another defining attribute of CAN networks is their ability to address and recover from node faults and a variety of bus errors. The Bosch CAN specifications [2] outline several possible CAN errors and give actions to be taken when they are encountered. By detecting and addressing these errors it is difficult for the entire CAN bus to be brought down by one faulty node.

The CAN specification gives four methods for error checking. The first is monitoring at the transmitter. If the transmitting node is transmitting a dominant bit and monitors a recessive bit on the bus, an error has occurred. Likewise, a dominant bit on the bus while a recessive bit is being transmitted triggers an error state. Another error checking method is the Cyclic Redundancy Check (CRC). Every node computes a CRC for the message on the bus and then compares it to the CRC that has been computed by

the transmitter. If a node detects an error in the CRC, the frame is discarded by the receiving nodes and re-sent by the originating node [1]. The third method of detecting errors is the bit stuffing check. This simply generates an error if more than five consecutive bits of the same level occur. The final error check is for form errors. CAN frames have several locations that must have either a dominant or recessive bit. If a node monitors an out of place bit in one of these fields, it generates an error. These error checks and fault confinement procedures keep a CAN bus operable even in the face of a significant number of errors.

Error confinement is another important aspect of CAN error handling. When a single node encounters errors, it enters into an “error active” state. When in an error active state, a node may still participate in bus activities and acknowledge received messages until the number of errors exceeds a preset limit; then the node transitions into an “error passive” state. Nodes in an error passive state may not transmit acknowledge bits, but may continue participating in bus activities in a slightly more limited state. If errors continue to occur, the node’s drivers are switched off. If a node receives enough messages correctly, it may move out of the current error state to a lower error state. This effectively contains errors in a faulty CAN node to that node instead of allowing it to bring down the entire bus.

The speed at which CAN frames can be transmitted makes it good for most control applications. Basic CAN frames have eight or fewer data bytes. These relatively short frames and comprehensive error checking allow CAN messages to be transmitted fast enough so that they are useful to most physical systems. The Bosch CAN specification does not give specifications for the speed at which CAN messages may be

transmitted. However, the ISO 11898-2 standardization for a dual-wire differential voltage CAN bus set the maximum speed to 1Mbit per second [13]. The maximum bus length for communication at this rate is in the tens of meters and is limited by the speed of the electrical signal as it propagates down the wire and back before the node is able to monitor the correct level on the bus [14].

CAN is a communication protocol whose popularity continues to grow. The characteristics of CAN busses allow them to be used for communication between many parts of real-time systems. The ability to use CAN for real-time systems qualifies it as an excellent medium for time scales testing provided that bus diagnostics, experimentation, and monitoring can be run in real time as well.

CHAPTER THREE

Technology Used in CAN Bus Implementation

The CAN bus developed for this project has two types of nodes. The first node is controlled by a PC running the QNX real-time operating system (RTOS). This type of node is responsible for monitoring network traffic, analyzing the traffic, and generating experimental traffic. The controller developed by the author for this node type is an Arcom board that uses Philips' SJA1000 CAN controller. The second type of CAN node performs small tasks such as reporting sensor values, actuating devices, or displaying data at remote locations. These nodes are made up of Nintendo Gameboy Advance (GBA) SPs and a device named the Xport 2.0 from Charmed Labs, both of which will be discussed later.

QNX Real-Time Operating System

QNX is a real-time operating system, meaning that a thread's utilization of the processor is determined by the thread's priority and the priorities of the other threads competing for processor time. The thread with the highest priority always has access to the processor. In practice this means that critical inputs are addressed immediately and the corresponding critical applications are assured processor time as soon as it is needed instead of having to wait for processor resources [15]. QNX has been designed to run a variety of real-time applications from embedded applications with stringent memory requirements to symmetrical multi-processor machines that perform high speed, critical calculations. The emphasis QNX Software Systems Limited (QSSL) places on

development makes QNX an ideal platform for developing a real-time CAN bus analyzer. Several aspects of the QNX operating system are important in driver development.

Scalable Architecture

QNX's nucleus is its microkernel. QNX allows a user to add or remove various modules that would automatically be included in a traditional kernel [3]. This scalable architecture is what allows QNX to be so flexible. These components can be process managers, a file system, drivers, and applications. Components can be added and removed in real time at the user level. In order for different components to communicate, QNX uses a technique called message passing. Apart from direct kernel calls, which generally deal with thread scheduling, all modules interact with messages. Messages are passed between "servers" and "clients" in QNX. Clients request information from a server by sending a message to the server. Once the request has been sent, the client blocks, or waits, until it receives a message back from the server with the information it needs. In this message paradigm it is important for the clients to initiate messages to the servers because having a server that was blocking on a reply from a client would slow the system and compromise real-time execution. Most messages are sent and received without having to write specific message-sending code. The QNX POSIX libraries are already populated with functions that handle most message sending tasks; although, when writing a resource manager with POSIX functions, message passing is important.

Resource Managers

A resource manager can be described as a set of functions that allow a user access to some service or device [3]. A QNX resource manager is commonly known as a driver on other systems. When a user needs to access a device, the resource manager provides specific functions that allow the user to interact with the device at a higher level. For instance a serial port and modem will not use the same techniques for reading incoming data; however, a user can read from both devices using the *read()* command once the device has been opened. The resource manager for the modem provides a different functionality for the *read()* command than the serial port's resource manager, but the user can access them, and a host of other devices and services, with the *read()* function.

All operating systems have some form of resource manager or library for abstracting device operations. QNX is unique because its resource managers run at thread-level. QNX does not treat a resource manager any differently than it would treat a program displaying the time to the screen. Since a resource manager is not directly associated with the kernel or any other module of the operating system, it is not difficult for a user to start and stop the resource manager when needed. The ability to start and stop resource managers on the fly makes the development of resource managers much simpler than it would be otherwise.

Interrupts

Most devices operate with some type of interrupt. Interrupts allow devices to react to an external input or a condition by gaining processor access immediately at the occurrence of the event. Interrupts are obviously important to real-time systems, since real-time systems typically need to react to a host of inputs. While interrupts allow a

system to react to stimuli in a timely manner, they come with the overhead of a context switch every time there is an interrupt. QNX's microkernel architecture give it very fast context switches. On an x86 processor QNX can switch contexts in 21 instruction cycles [16]. This low cost for context switching allows QNX to handle multiple threads smoothly while continuing to respond efficiently to interrupts generated by various real-time dependent peripherals.

QNX has two methods of handling interrupts. *InterruptAttachEvent()* is typically used for devices that do not interrupt frequently and that do not require immediate attention when they interrupt. *InterruptAttachEvent()* attaches a continually running thread to an interrupt. This thread blocks on a message telling the thread that there has been an interrupt that needs to be serviced. When an interrupt occurs, the kernel masks the interrupt and sends a message to the thread to handle the interrupt. Once the thread has cleared the interrupt source, the thread unmask the interrupt, loops, and blocks on an interrupt message from the kernel again. This method of handling interrupts requires no interrupt service routine (ISR). The entire handler runs at thread, or user, level. This lessens the possibility of crashing the system if an interrupt hangs or the device malfunctions. One drawback to *InterruptAttachEvent()* is that it unblocks on every interrupt regardless of a given interrupt's relevance to the interrupts that are being checked by the interrupt handler thread.

The second method of interrupt handling in QNX is *InterruptAttach()*. *InterruptAttach()* runs an ISR for a particular interrupt and then decides whether or not to schedule a thread to run at user level. The ISR is responsible for clearing, or at least masking, the interrupt. If the device does not need urgent attention, the ISR can simply

schedule a thread to perform the task later and exit. If the device does need attention immediately, it is given the highest system priority as an ISR. The drawback to *InterruptAttach()* is that there is always some context switching involved. If frequent, unnecessary interrupts occur the system's performance may be slowed. Between these two interrupt handling methods QNX is able to respond to a variety of real-time stimuli.

Arcom AIM104-CAN Controller Board

In order to utilize the benefits of QNX, a resource manager for a CAN controller had to be written by the author. The CAN controller selected was an Arcom board originally designed for PC-104 systems [4]. Once fitted on an adapter card, this board was able to interface with the system hardware through the ISA bus. All CAN aspects of the Arcom card are handled by a Philips SJA1000 CAN controller. The developer communicates with the SJA1000 by writing to and reading from its registers. The address space of the SJA1000 is mapped to some base address in the QNX's address space. The base address is set using jumpers for the upper 8 bits of the address. The interrupt request (IRQ) line is also set using jumpers on the Arcom board. The Arcom board provides an additional register for lighting three tri-color LEDs as well as EEPROM memory that stores information about the AIM104-CAN card. (The latest documentation for the AIM104-CAN is one revision earlier than the version of the PCB used in this project. The AIM104-CAN PCB used in this system is Version 2, Issue 1. The latest documentation for the AIM104-CAN is Version 1, Issue 2. This did not pose significant problems; however, the document section "Isolated CAN Transceiver" that states the CAN transceiver requires "external powering in order to function" is no longer

correct, as the transceiver is powered from the ISA card. There are still connections for external powering, so care should be taken when attempting to boost transceiver power.)

Nintendo Gameboy Advance SP

When Nintendo developed the Gameboy Advance (GBA) it was immediately recognized as a novel, yet powerful, development target. Before the GBA was even sold, PC emulators were already being developed to test gaming applications. The GBA has two microprocessors onboard [17]. In order to support legacy games from the original Gameboy, the Nintendo engineers included the old Z80 processor on one side of the PCB. In order to run newer, more processor- and graphics-intensive games, the other side of the PCB is fitted with an ARM7TDMI 32-bit 16.78 MHz RISC processor. This processor is commonly used in a variety of technologies from cell phones to Ethernet routers. Old games used a 16 bit bus to transfer information from the game cartridge to the Z80 processor. However, new games use a 32-bit bus. In order to support legacy game cartridges, the new GBA must multiplex address and data information across the old 16-bit bus. Once the intricacies of the GBA are mastered, the GBA is a powerful embedded system. The GBA possesses a 240 x 160 pixel TFT color display, a communications port capable of several modes of operation, ten user input buttons, four channel analog sound output, and internal power supply [17]. All of these features are well documented and supported by freeware libraries, including the GNU code development tool chain.

CharmedLabs Xport 2.0 and XRC

The Xport 2.0 is a flexible and elegant solution to the problem of harnessing the power of the GBA without intensive hardware development. In the words of the Xport

developers, CharmedLabs, the Xport “turns the Game Boy Advance into a powerful embedded development system.” The Xport combines the flexibility of a Field Programmable Gate Array (FPGA) with the processing muscle and convenient I/O of the GBA. The reconfigurable FPGA allows a developer to integrate almost any hardware functionality into the Xport. The Xport also contains flash memory that the GBA boots from and 64 I/O pins that are programmable through the FPGA. CharmedLabs provides a thorough, although completely undocumented, library of functions to control many of the GBA features and all of the Xport’s functionality.

Xilinx Spartan II FPGA

The ingenuity and flexibility of the Xport lies with the FPGA. Xports are manufactured with two types of FPGAs. The version used for this project was a Xilinx Spartan II XC2S150. This FPGA has a 50 MHz clock and 150,000 gates. The XC2S150 comes with 260 I/O pins; although, only 64 are provided to the user through the Xport. CharmedLabs uses the FPGA to demultiplex the GBA’s multiplexed data and address bus. Direct communication with the GBA is abstracted so developers using the Xport have clearly defined components with which to work. The speed, number of gates, and outputs of this FPGA made the Xport an ideal target for a CAN controller implemented in the FPGA’s hardware.

Xport Robot Controller

An optional accessory to the Xport is the Xport Robot Controller (XRC), also developed by CharmedLabs [6]. The XRC, while not critical to development of a CAN controller, provides functionality for the CAN nodes. The XRC provides several data

collection options and several actuator options. These features include four motor controllers, digital I/O, analog sensors, a Bluetooth module, and infrared proximity sensors. CharmedLabs provides Verilog libraries for implementing the XRC functionality in the FPGA and software libraries, including a patent-pending method of closed loop feedback for DC motor control. These peripheral devices, while not directly related to the implementation of a CAN node, provide the functionality that a CAN node will need to be useful in a physical system.

CharmedLabs Open-Source Libraries

CharmedLabs provides the libraries necessary for a user to work with the Xport and XRC as source libraries. These libraries fall into two broad sections: Verilog source code for the FPGA and C++ libraries for the GBA. The Verilog library's most important component is the *Primary()* module. This module is responsible for decoding the GBA bus and streamlining communication from the flash to the GBA. This primary module has to be instantiated as the top module of every project so that it can access the FPGA pins that are routed to the GBA's cartridge slot. Once the bus has been decoded, developers have access to a 24 bit address bus, a 16 bit data write bus, a 16 bit data read bus, a vector of interrupts, a vector of lines to clear interrupts, a clock signal, and a reset signal. The XRC peripherals are also included as Verilog source modules. These modules can be instantiated in the same module as *Primary()* and handle the hardware needs of the motor controllers, general purpose I/O, optical encoders, and the Bluetooth module.

The second group of libraries is the C++ source code that controls the various aspects of the Xport's operation. Each peripheral has a C++ class that contains its control

methods. For instance, the infrared sensors have a class with functions like *SetXirPower()*, *GetXir()*, and *SetXirRingLength()*. One of the most useful classes is the interrupt controller class. When the interrupt controller is instantiated, all interrupts are directed to its ISR. When a user needs to use an interrupt, the user creates a class method named *Interrupt()* and registers it with the interrupts controller along with an interrupt number. If the interrupt is an external interrupt, the interrupt number corresponds to the interrupt's location in the *Primary()* module's interrupt vector. Once the interrupt is registered, whenever that IRQ line is raised, the interrupt is masked and the class's *Interrupt()* method runs to clear the interrupt. CharmedLabs also provides utility classes such as a text display class and a class for creating menus. These libraries allow the Xport and XRC's functionality to be exploited smoothly and quickly.

OpenCores.org CAN Core

The CAN functionality of the Xport comes from a CAN controller implemented in the FPGA. OpenCores.org is a website that provides open source hardware cores. A hardware core is a hardware configuration that can be implemented on an FPGA. Cores typically come in source code versions or pre-compiled into a library. OpenCores.org has only one CAN core [7]. This core was developed by Igor Mohor and is shared as source code on OpenCores.org. This core is loosely based on the SJA1000; although, this core uses active high CS, RD, and WR signals instead of active low signals like the SJA1000. The OpenCores.org CAN core is controlled by writing to and reading from addresses that can be memory mapped. This CAN core also uses an 8051 style bus. An 8051 bus shares a single 8 bit bus for address and data. The bus function is controlled using ALE, RD, and WR signals. The CAN core has interrupts that function like the

SJA1000 as well. One drawback to this open source core is the lack of documentation.

A brief functional diagram is provided on the OpenCores.org website, and the testbench gives some indication of the interface and functionality.

Microchip MCP2551 CAN Transceiver

The final piece of hardware used to outfit the Xport for CAN bus interaction is the CAN transceiver. The transceiver used is a MCP2551 from Microchip. This is a simple IC that converts the 5V differential signals from the CAN bus to a TTL receive signal for the FPGA (Figure 3). Conversely, the transceiver converts TTL output from the CAN core to a signal that operates on the CAN bus. This 8 pin IC is the last link in this developmental CAN bus.

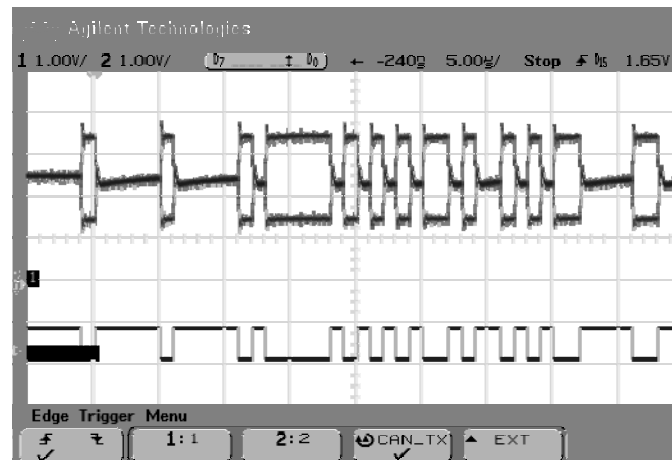


Figure 3: TTL to Differential CAN – The MCP2551 translates standard TTL levels to a 2.5V differential CAN signal. “CAN_TX” is the data input to the MCP2552.

The final configuration of these components allows the GBA-Xport combination to function as the processor and hardware for the CAN controller while the MCP2551 connects the transmit and receive lines of the OpenCores.org CAN core to the CAN bus. A block diagram of the final configuration can be seen in Figure 4, and a photograph of a complete GBA CAN node developed for the thesis is given in Figure 5.

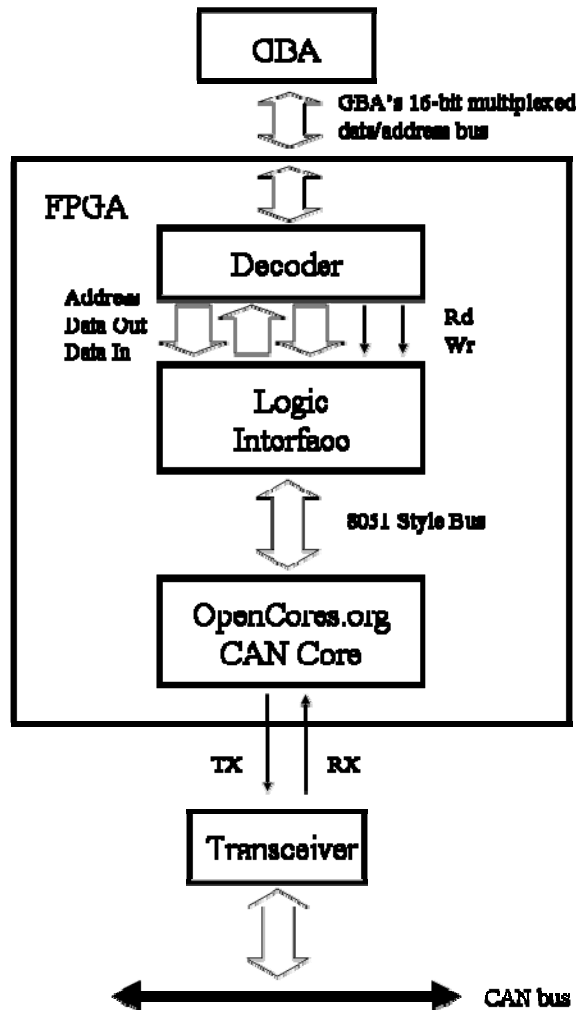


Figure 4: GBA-Xport Block Diagram – The interactions between various levels of the GBA and Xport's interface to the CAN bus are illustrated in the figure above.

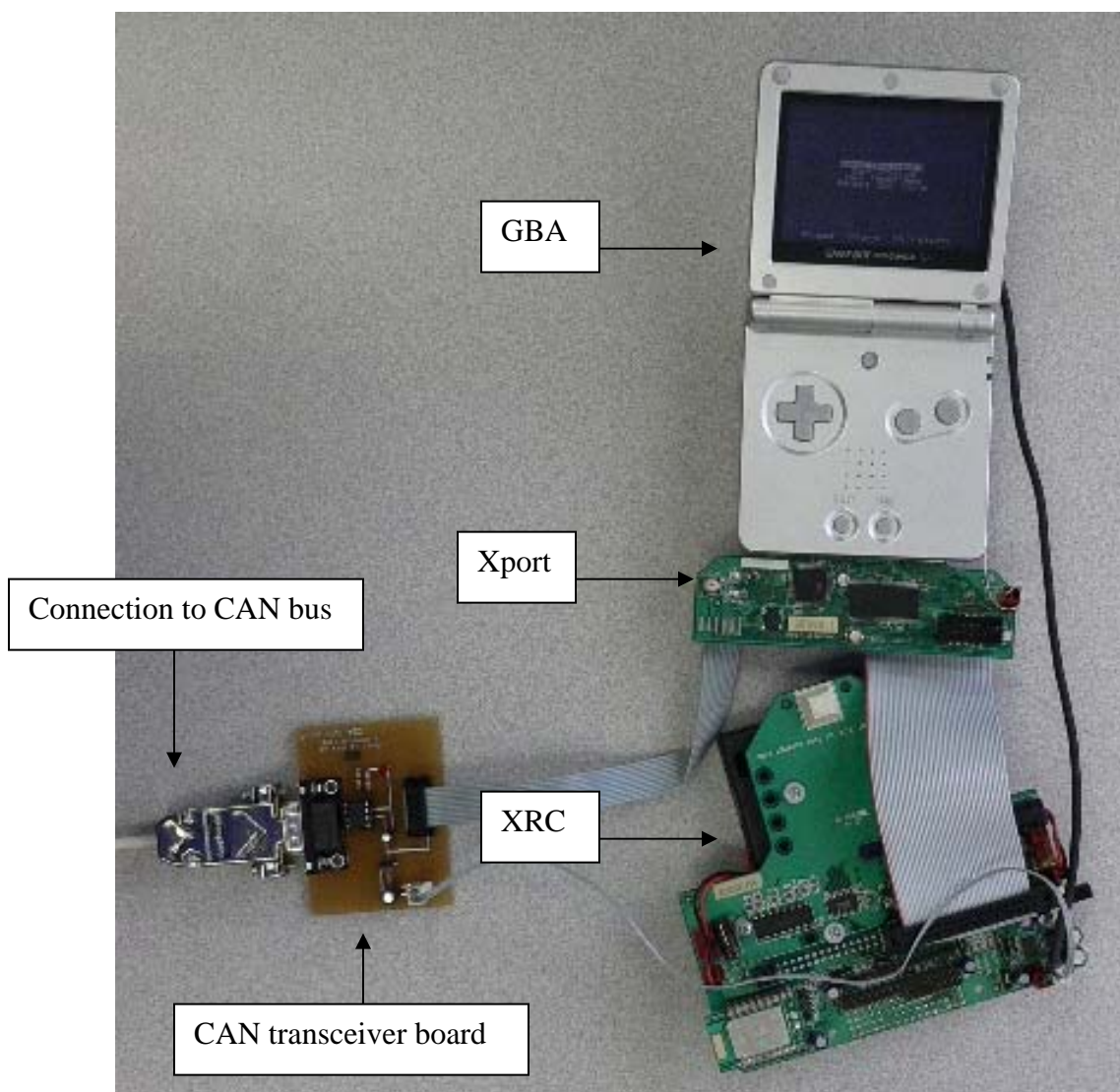


Figure 5: CAN Node Picture – Final configuration of the GBA, Xport, XRC, and CAN transceiver.

CHAPTER FOUR

QNX Driver Development

Use of the AIM104-CAN board with the QNX operating system required a resource manager to be developed. Since QNX is similar in many ways to Linux, the decision was made to develop a resource manager by modifying an existing Linux driver that supported an ISA bus interface to Philip's SJA1000. Can4linux is an open source CAN driver that supports the SJA1000. Once can4linux had been completely rewritten for QNX, the project was renamed can4qnx. The major components developed by the author are described in the following chapter. Individual functions and files written for this thesis are documented, described, and included in their entirety in Appendix A.

SJA1000 Access Functions

The lowest level driver functions access the SJA1000 directly. These functions are responsible for the direct communication with the SJA1000's registers. By packaging the register operations used to communicate with the SJA1000 into a form that is easier to deal with, operations requiring several steps can be simplified to one function call. These functions deal with everything from initializing registers of the chip to setting up timing to handling interrupts. All the device communication is done using QNX's *in8()* and *out8()* commands; although, these commands themselves are hidden by macros. In order to access registers in an intuitive manner a data type named *canregs* is defined. *Canregs* is a structure of 8-bit elements with names corresponding to the registers of the SJA1000. By typecasting the base address of the device as a pointer to a *canregs*

structure, the SJA1000's registers can be referred to by the member names in *canregs*. Bit values are defined for each register so that their purpose is more easily inferred. QNX's *usleep()* function is used to slow down I/O when a delay is necessary during device communication.

Interrupt Handler

One of the keys to receiving and transmitting CAN messages efficiently and quickly is the use of interrupts. Can4qnx uses the *InterruptAttachEvent()* method of interrupt handling. *InterruptAttachEvent()* works well for a SJA1000 based CAN controller because the interrupts that are generated do not occur frequently enough to derail real-time operation. The SJA1000 can be configured to generate interrupts on various events or not at all. In BasicCAN mode, the mode used in this project, the SJA1000 generates interrupt on wake-ups, receive FIFO overruns, errors, transmits, and receives. The relative frequency of these is low when compared to the processing speed the QNX system, so *InterruptAttachEvent()* works well.

The interrupt handler thread is simply a function written by the author called *CAN_Interrupt()* that is started from the resource manager. Since *InterruptAttachEvent()* is being used, the handler is just another thread that unblocks when an interrupt is generated. When *CAN_Interrupt()* runs, the thread registers itself with the kernel and then blocks on an interrupt. Once an interrupt is received, *CAN_Interrupt()* unblocks and checks the SJA1000's status register to determine what generated the interrupt. Reading the status register also clears the interrupt. If the status register does not show that there are any interrupts needing to be cleared, *CAN_Interrupt()* loops and blocks on the next interrupt. If the SJA1000 has generated the interrupt, it is addressed. In the case that a

received CAN message generated the interrupt, *CAN_Interrupt()* reads the appropriate registers using the SJA1000 access functions described above and stores the message in a receive queue. If a CAN message transmit generated the interrupt, *CAN_Interrupt()* checks to see if there are any messages in the transmit queue and sends the next one if there are. If the interrupt was generated by an error, *CAN_Interrupt()* increments error counters. In the case of an overrun, the overrun counter is incremented and the overrun status is cleared. After the interrupt has been cleared, *CAN_Interrupt()* unmask the interrupt so subsequent interrupts will be received.

I/O and Connect Functions

The QNX connect functions and the QNX I/O functions are the functions used when the device receives standard POSIX function calls like *read()*, *open()*, or *devctl()*. The prototypes for these functions are specified to reflect the parameters and return values that are expected by QNX. For example, *open()* is a connect function that returns a non-zero *int* value if the file is opened successfully. The *open()* function is passed a pointer to an open context block (OCB), an open message, and two pointers to shared address space. The *open()* function stores the context in the data space of the OCB. Once *open()* has been called by a program, that program has a file pointer, and I/O functions can be called. The I/O functions are passed messages from the client, the OCB initialized by *open()*, and a low-level context block. Inside the connect and I/O functions written for this resource manager the SJA1000 access functions are used to execute the commands or obtain the data that the program requires. When a program calls the *read()* function, the *read()* function checks for any new CAN messages. If there are no new messages, the *read()* function returns a zero. If there are new messages, the *read()*

message returns a pointer and the number of bytes requested by the program in a *canmsg_t* structure. Details of other I/O and connect functions written for this project can be found in Appendix A.

can4qnx Resource Manager

In QNX, the resource manager is the interface between the user and the device. The heart of a resource manager is an infinite loop that blocks on a message from a client. Once a message is received the resource manager responds to the message using various device specific connect and I/O functions.

The top level interface for a resource manager looks very similar for any device. In this CAN controller resource manager's *main()* function several configuration variables are initialized to default values. The CAN controller's make and model are set; the default baud rate is set to 125kbits per second; the acceptance code and mask's default values are set to accept all incoming message identifiers; and the value for the output control register is assigned. The resource manager must know the connect and I/O function addresses. QNX provides a default set of connect and I/O functions so that a message for a unsupported operation does not crash or hang the device. Once the connect and I/O functions have been given default values, the device specific connect and I/O functions are assigned, overwriting the default functions. These functions are now used when a user requests a POSIX function related to this device. Next, a mechanism for dispatching threads to handle device requests is implemented. Finally, the thread for handling interrupts is created, and the low-level context of the device is stored. Once these tasks are completed, the resource manager simply waits for client messages (Figure 6).

The implementation of the abstracted levels described above allows a user to interact with the Arcom AIM104-CAN in a normal, POSIX compliant manner. When a user program accesses the device in the prescribed manner, the QNX real-time system becomes a powerful CAN node.

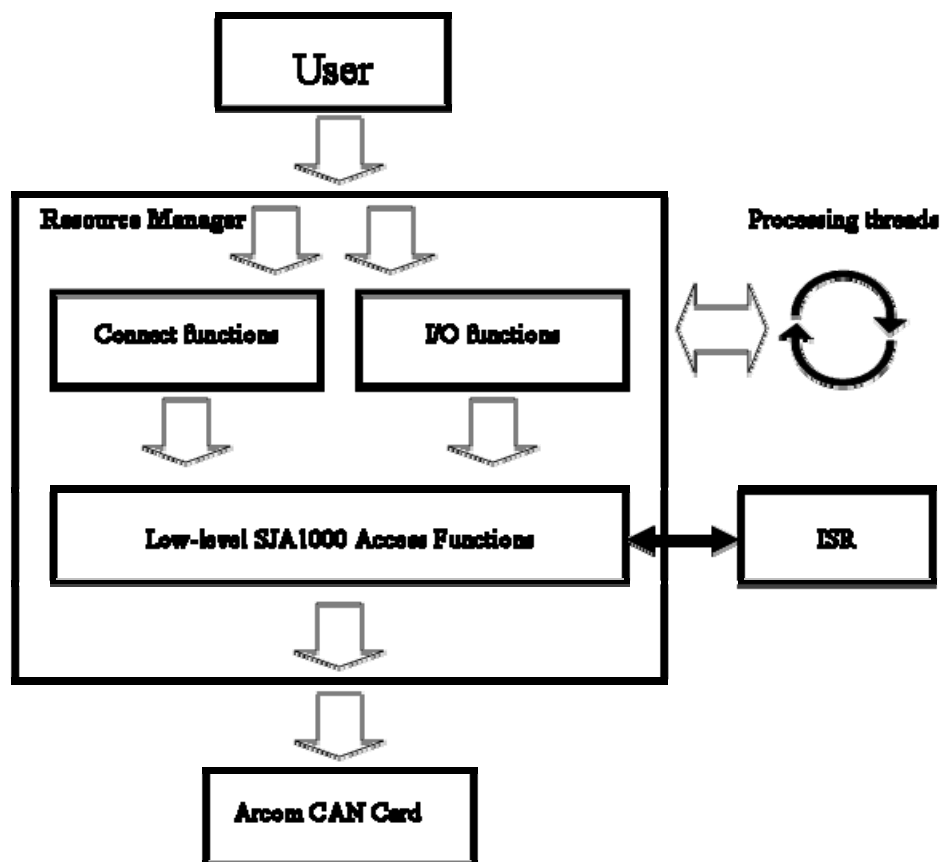


Figure 6: Resource Manager Block Diagram – This is a block diagram of the resource manager and its interaction with the user, processing threads, ISR, and Arcom CAN card.

CHAPTER FIVE

Xport CAN Controller Development

Several interfaces needed to be considered and developed for this thesis in order to assemble the various pieces of the CAN controller for the Xport. The GBA has a method of handling interrupts that is different than the CAN controller's method for generating and clearing interrupts. The GBA also uses a bus structure that does not fit simply into the CAN controller's architecture. These interfaces proved to be the most difficult aspects of this project.

Bus Description and Behavior

The most important aspect of the GBA-CAN core interaction is interfacing the GBA's address and two data buses to the CAN core's bus. The SJA1000, and hence the OpenCores.org CAN core, use an 8051 style bus. This bus format shares a single 8 bit bus for both address and data transmission. Addresses are transmitted first, followed by data. Three control signals are used: Address Latch Enable (ALE), Read (RD), and Write (WR). The address is latched into the CAN core on the falling edge of ALE. Once the address has been latched, data is driven by either the GBA or CAN core depending on whether the RD or WR strobe is raised. The GBA raises RD for the CAN core to drive data onto the bus and raises WR when the GBA is writing data to the CAN core.

The Xport presents three busses that the GBA uses to communicate: a 24 bit address bus, a 16 bit data write bus, and a 16 bit data read bus. The GBA also uses *Rd* and *Wr* lines. When the GBA reads or writes, the address becomes stable about 300ns

before the *Rd* or *Wr* line is raised. Once the address is stable, the *Wr* line is raised and the data write bus is latched on the falling edge of *Wr* [18]. When the *Rd* line is raised, data is driven onto the data read bus for the duration of the signal. Behavior of a write and read to the CAN controller's address space can be seen in Figures 7 and 8, respectively.

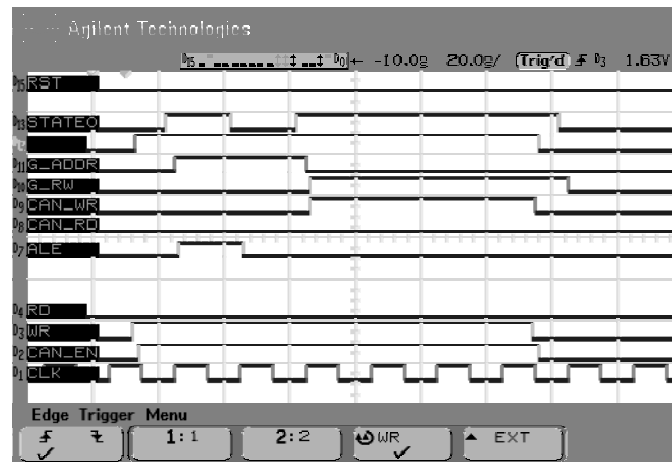


Figure 7: Write to CAN Address – This figure is an oscilloscope capture of a write to the CAN controller's address space. State changes can be seen by looking at "STATE0", the least significant bit of the current state. The state machine begins in the idle state and returns to the idle state once "WR", the GBA's *Wr* strobe goes low. "G_ADDR" is the *gateaddr* signal. "G_RW" is the *gaterw* signal.

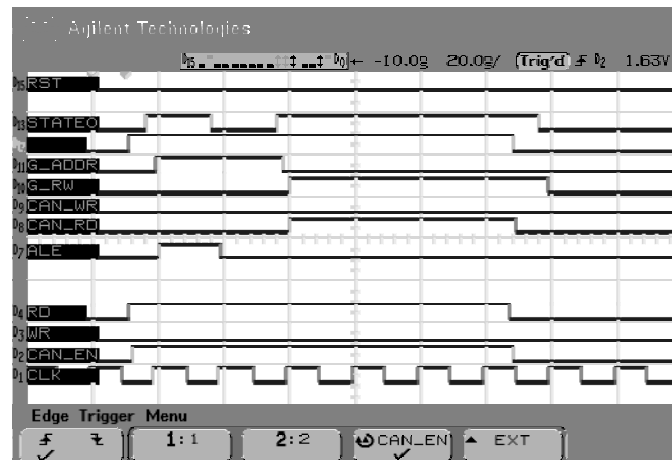


Figure 8: Read from CAN Address – This figure shows a read to the CAN controller's address space.

GBA to OpenCores.org CAN Controller Bus Interface

A minimalist approach was used by the author to streamline the state machine and the registers used by the state machine to control the CAN core (Appendix C).

Combinational logic and two control registers, *gateaddr* and *gaterw*, allow for a simple interface between the two bus structures (Figure 9). An enable signal brings the state machine in and out of its idle state. This enable signal is assigned a high using combinational logic as soon as the address is in range and a *Rd* or *Wr* strobe is raised. The RD and WR signals to the CAN core are gated straight from the GBA's *Rd* and *Wr* signals with the *gaterw* register whose value is assigned in the state machine. The 8 bits of the data read bus are always high-z unless the RD signal is high, in which case it is assigned the 8 bits of the CAN core's bus. The CAN core's bus is slightly more complicated. If the RD signal is low and *gateaddr* is high, the CAN core's bus is assigned the lower 8 address bits. When RD and *gateaddr* are low, the CAN core's bus is assigned the GBA's bus of data to be written. The CAN core's bus is assigned high-z values if RD is high, so that it can monitor the bus for the data being written by the GBA.

Since the CAN core's bus has specific timing characteristics, a state machine was implemented to interface the busses (Figure 10). Four states are used. The first state is the *idle* state. This is the initial state after a reset. During periods when there is no activity between the CAN core and GBA, the interface is in the *idle* state. All control registers are cleared in this state. The *idle* state always returns to *idle* unless the enable signal goes high indicating that the address has stabilized in the CAN controller's range. Once the address is in range, the state machine moves to the first address state, *addr1*. In *addr1* the ALE line is raised for a clock period and the lower eight bits of the address bus

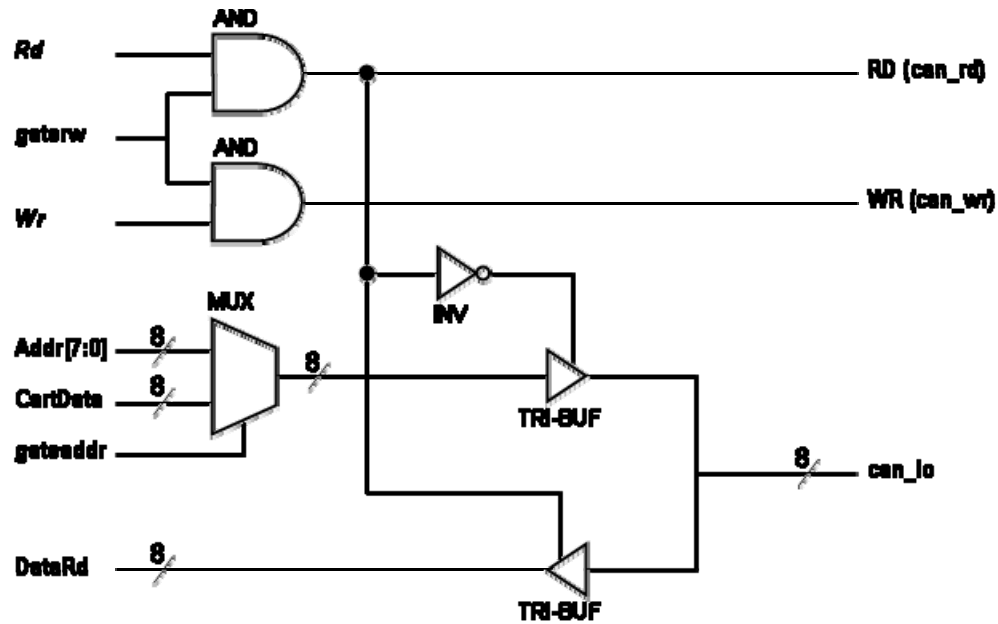


Figure 9: CAN Core Bus Interface – This is a schematic of the combinational logic that gates address and data over the 8051 style bus to the CAN core. “BUFE8” is a tri-state buffer that controls the direction of data flow on the bus. “M2_1” is a multiplexer controlled by *gateaddr* that chooses whether to send address or data to the CAN core.

proceeds to *addr2*. In *addr2* the ALE line is lowered to create a negative edge.

Lowering ALE latches the address, and the state machine moves to the *rwst*. In *rwst* *gateaddr* is lowered, *gaterw* is raised, and depending on whether the GBA has the *Rd* or *Wr* line raised, data is driven onto the CAN core’s bus from the GBA’s bus, or the GBA’s bus is set to high-z so that data can be read from the CAN core. The state machine remains in *rwst* until the *Rd* or *Wr* signal goes low. The state machine returns to the *idle* state once the enable signal goes low.

The entirety of this state machine runs during the time when *Rd* or *Wr* is high. Since the FPGA is running at 50 MHz and the GBA’s processor is only running at 16.78 MHz timing is not a problem. By waiting until one of the *Rd* or *Wr* strobes is raised, the address is guaranteed to be stable and the data is guaranteed to be ready. Having the

CAN core run almost three times as fast as the GBA also assures plenty of time for the CAN core to drive its data onto the bus during a read.

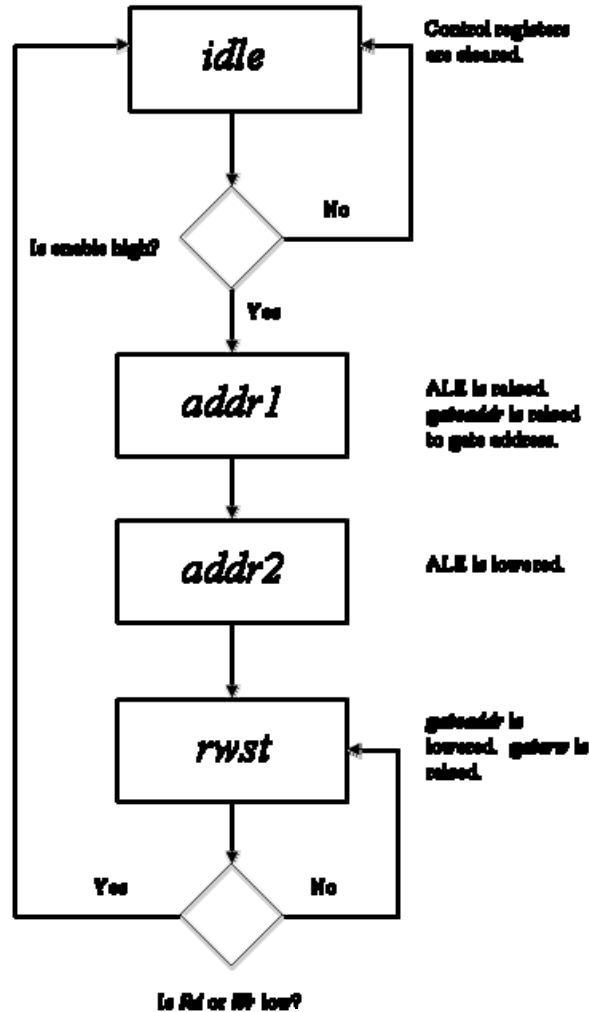


Figure 10: CAN Interface State Diagram – This is a diagram of the state machine used by the logic interface. States are shown in rectangles, and decisions are shown as diamonds.

CAN Core-GBA Interrupt Interface

Interrupts are an important part of the GBA-CAN core interface. The CAN core uses active low interrupts. These interrupts are cleared when the interrupt register of the CAN core is read. The Xport interface with the GBA expects active high interrupts that

are cleared when a corresponding IRQ clear line is raised. To reconcile these two interrupt techniques a hardware interrupt mask was developed for this project. When the CAN core generates an interrupt, it is inverted to active high, and ANDed with the interrupt mask before being sent to the GBA. The interrupt mask is set synchronously by reading the reset signal, IRQ clear line, and interrupt line from the CAN core. After a reset the interrupt mask is set to high unmasking the interrupt. If the IRQ clear line is raised, the interrupt mask is set to low. This masks the interrupt from the GBA. If the interrupt line from the CAN core goes low, indicating that the interrupt has been cleared by a read of the status register, the interrupt mask is raised on the next clock cycle. It is up to the software developer to unmask the interrupts by reading the status register, or to clear the interrupt before the IRQ clear line is raised by reading the status register if the corresponding CAN controller software discussed below is not being used.

Hardware Reset Register

The ability to reset the CAN core's hardware through software was added to the CAN core interface for this project. To the software developer this simply looks like another register following the existing CAN core registers. This register stores the last bit of the data bus into a one-bit reset register on the negative edge of Wr when the address bus points to this register. This register is tied to the reset line of the CAN core. This is useful because it allows a program to return the CAN core to a known state with the reset signal on the CAN core. Figure 11 shows a write to raise the reset line of the CAN core and another write to lower the reset line.

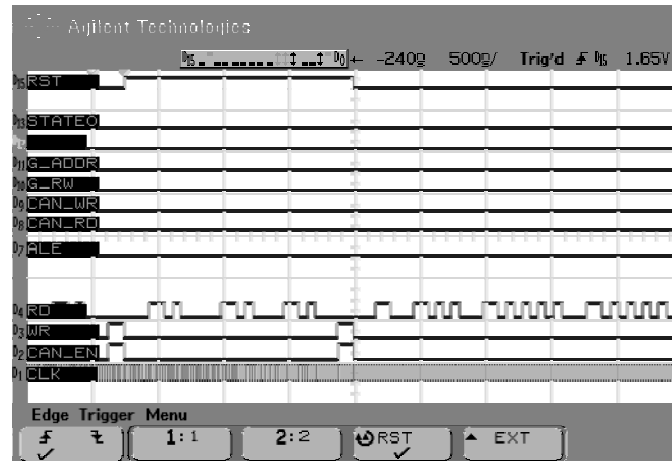


Figure 11: Hardware Reset – This figure shows writes to the reset register of the CAN interface logic. As seen in the “RST” signal, these writes raise and lower the reset line on the CAN controller.

CAN Transceiver

A custom PCB was created for the MCP2551 to connect to the Xport’s *can_wr* and *can_rd* and the CAN bus. This CAN transceiver board connects to the CAN bus using a standard DB-9 connector. This board can be terminated with a 120Ω resistor by adding a jumper. It also has an LED that lights when the bus is active and is powered from an auxiliary power output on the Xport. A schematic (Figure 12) and photograph (Figure 13) of the board can be found below.

CAN Core Library Class

To manage the OpenCores.org CAN core and the functionality of the interface logic, a controller class was written by the author. This class, *CCanCore*, is responsible for everything from low-level register manipulations, to handling interrupts, to high-level user interaction. The *CCanCore* class is written in a similar fashion to any other Xport peripheral class. The interrupt mask implemented in the CAN interface module allows

the *CCanCore* class to use the interrupt handling provided in the CharmedLabs libraries.

The ISR in the CAN controller class automatically clears and unmask the interrupt when it occurs. This class is described in detail in Appendix B.

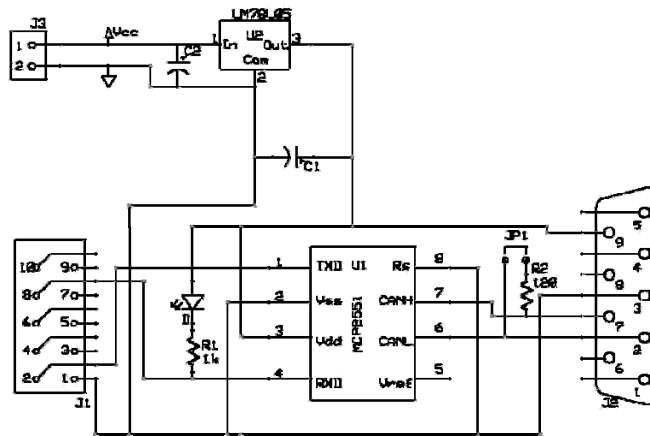


Figure 12: CAN Transceiver Schematic – This is a diagram of the custom CAN transceiver. J3 is the power connector. J2 is the DB-9 connector to the CAN bus. J1 connects to the I/O pins of the Xport's FPGA.

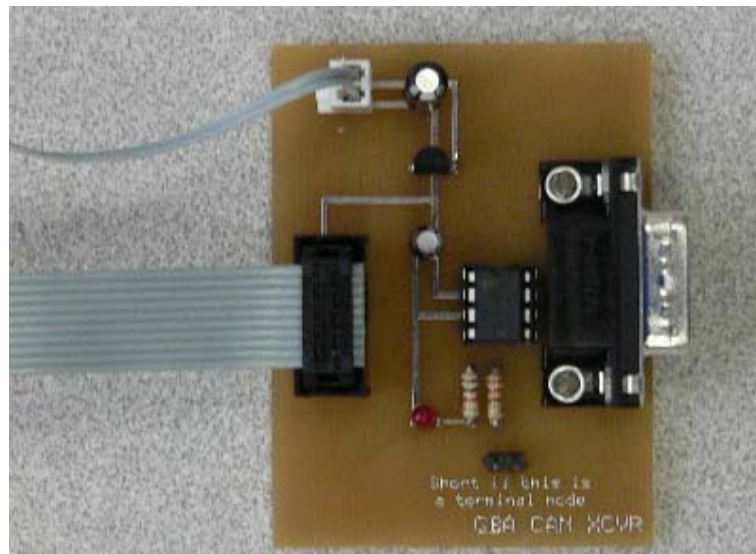


Figure 13: CAN Transceiver Picture – This is a photograph of the final CAN transceiver board.

Conclusion

Pairing the OpenCores.org CAN controller on the Xport and the Arcom AIM104-CAN board on the QNX RTOS gives a powerful real-time communication backbone for a control system. Both systems present an intuitive, well documented interface that will allow for their fast and efficient utilization. QNX's lean real-time operation paired with a CAN interface card allows it to be used as an effective node on any CAN bus, and the flexibility of the Xport combined with the convenience of the GBA's embedded system supplements the usefulness of the CAN controller implemented in the Xport's hardware. The system developed for this thesis will easily support further research in time scales and other control applications.

APPENDICES

APPENDIX A

Appendix Documenting Can4qnx Code

QNX SJA1000 Access Functions

**int CAN_GetStat(__LDDK_DEVCTL_PARAM, CanStatusPar_t *stat),
“can_sja1000funcs.c”**

CAN_GetStat() is responsible for retrieving status information from the CAN controller. The *CanStatuPar_t* is a data structure defined in “can4qnx.h”.

__LDDK_DEVCTL_PARAM is defined in “can_defs.h” and is the list of parameters for the resource manager’s *can_devctl()* I/O function. The value of the status register is printed to stderr as a decimal *int*. This function always returns an *int* value of zero after filling the data structure pointer to by *stat*.

int CAN_ChipReset(), “can_sja1000funcs.c”

CAN_ChipReset() is responsible for returning the SJA1000 to reset mode and initializing the configuration registers to values defined elsewhere. If the board is not in reset mode after ten microseconds, this function returns a “-1”. Otherwise, it initializes the clock register, mode register, interrupt enable register, and output control register. Finally, it calls the *CAN_SetTiming()* function and *CAN_SetMask()* function, passing them their respective default values. On completion a zero is returned.

int CAN_SetTiming(int baud) , “can_sja1000funcs.c”

CAN_SetTiming() configures the two timing registers on the SJA1000. The function switches on *baud*. If *baud* matches a baud that has been defined in

“can_sja1000.h”, the appropriate index is set for the *CanTiming* array that is initialized as a global in “can_sja1000funcs.c”. The indexed values are then written to the SJA1000.

If *baud* does not match a predefined value, it is treated as a two 8 bit register values. The upper 8 bits are written to the first timing register, and the lower 8 bits are written to the second timing register. In this case the two register values being used are output to stderr. This function returns zero upon completion.

int CAN_StartChip(), “can_sja1000funcs.c”

CAN_StartChip() is responsible for moving the SJA1000 from reset mode to normal operation. The *RxErr* and *TxErr* are reset; the receive buffer is released; the overrun status, if any, is cleared; and any interrupt is cleared. Finally, the interrupt enable register is initialized. Once these tasks have been completed, the mode register’s reset request bit is cleared and the function returns a zero.

int CAN_StopChip(), “can_sja1000funcs.c”

CAN_StopChip() simply sets the reset request bit in the mode register and returns zero.

int CAN_SetOMode (int arg) , “can_sja1000funcs.c”

CAN_SetOMode() writes *arg* to the output control register and returns zero. This function must be used while the SJA1000 is in reset mode.

int CAN_SetMask(unsigned int code, unsigned int mask) , “can_sja1000funcs.c”

CAN_SetMask() sets the acceptance code and acceptance mask. *code* and *mask* are treated as four 8 bit values. The most significant byte of *code* and *mask* corresponds

to the most significant 8 bits of the identifier. *CAN_SetMask()* is for use when the SJA1000 is in PeliCAN mode and only then when it is in reset mode. This function always returns zero.

int CAN_SendMessage(canmst_t *tx) , “can_sja1000funcs.c”

CAN_SendMessage() sends the CAN message pointed to by *tx*. All of *CAN_SendMessage()*'s execution takes place with interrupts disabled. If *CAN_SendMessage()* determines that the SJA1000's transmit buffer is busy, the message is enqueued in the transmit queue and a zero is returned. If the transmit buffer is free, the appropriate registers are filled with information from *tx* and the transmit request bit is set in the command register. After this, interrupts are disabled and the number of data bytes sent is returned. The return values are somewhat misleading, since zero could refer to either the number of data bytes sent or a busy transmit. A better method would be to return a “-1” or the number of items in the transmit queue as a negative number when the transmit buffer is busy.

int CAN_GetMessage(canmsg_t *rx) , “can_sja1000funcs.c”

CAN_GetMessage() is responsible for retrieving a message from the receive buffer on the SJA1000 and filling the appropriate fields of *rx*. *CAN_GetMessage()* first checks the overrun status and clears the overrun status if there has been a buffer overrun. If an overrun has occurred, a message is printed to stderr. If there is a message in the receive buffer, *rx* is filled with the relevant information. Once *rx* has been filled, *rx* is enqueued in the receive queue. If the receive queue is full, a message is printed to stderr.

CAN_GetMessage() returns the number of data bytes received or zero if there was no new message. Once again, a better system of return values would be more useful.

int CAN_VendorInit(), “can_sja1000funcs.c”

CAN_VendorInit() is responsible for memory mapping the device memory used by the Arcom AIM104-CAN to device I/O and for calling *ThreadCtl()* to give the current thread privity to map memory to device I/O. *CAN_VendorInit()* uses several global variables declared in “can_sysctl.c”. *can_dev* is of type *uintptr_t* and points to the base address of the CAN board. *Length* is declared of type *int*, but is cast to a *size_t*. *Base* is also declared as an *int* but cast to a *uint64_t*. *CAN_VendorInit()* uses the QNX function *mmap_device_io()* to map the memory. If mapping fails, a “-1” is returned; otherwise, a zero is returned.

void CAN_LoopLEDs(), “can_sja1000funcs.c”

CAN_LoopLEDs() chases the tricolor LEDs on the AIM104-CAN breakout board. LEDs chase for ten cycles.

Interrupt Handler

void *CAN_Interrupt(void *data), “can_sja1000funcs.c”

CAN_Interrupt() is the interrupt handler thread for the AIM104-CAN board. *CAN_Interrupt()* creates an *event* structure which can be used for messages, pulses, or to start a thread [3]. *ThreadCtl()* is called to give this thread permission to attach an interrupt. The final initialization task is attaching the interrupt using *InterruptAttachEvent()*. *InterruptAttachEvent()* attaches the *event* structure to interrupt *IRQ* which is defined in “can_sysctl.c”. *InterruptID* is assigned the return value from

InterruptAttachEvent(). On success *IRQ_requested* is assigned a value of one. On failure, *InterruptID* is set to “-1” and *IRQ_requested* is set to 0, which is the value it was initialized to, and the function returns a *NULL* value.

Once initialized, *CAN_Interrupt()* blocks in an infinite loop until an interrupt occurs. When an interrupt is generated, the interrupt register is read, and if the CAN board did not generate the interrupt *CAN_Interrupt()* loops and blocks on an interrupt again. If the interrupt was generated by the CAN board, each interrupt is checked and acted upon. If a receive interrupt was generated, the message is recovered and enqueued in the receive queue in a manner similar to *CAN_GetMessage()*. If a transmit interrupt generated the interrupt, *CAN_Interrupt()* checks for a message in the transmit queue and sends the next message if a message is waiting. If an error interrupt is generated, an error message is placed in the receive queue. If an overrun interrupt is generated, the overrun status is cleared and an overrun frame is placed in the receive queue. Finally, the interrupt is unmasked and the function loops to the top and blocks on another interrupt.

I/O and Connect Functions

`__LDDK_OPEN_TYPE can_open(__LDDK_OPEN_PARAM), “can_open.c”`

can_open() is the connect function that a software developer calls to use the AIM104-CAN board. *can_open()* checks to see if the device is already open. If the device is being used by another thread, an error is returned. *can_open()* calls *CAN_VendorInit()* to set up I/O communication with the CAN controller. If *CAN_VendorInit()* returns an error value, *can_open()* exits with an error. *can_open()* proceeds to initialize the receive and transmit queues. *CAN_ChipReset()* is called to initialize the CAN controller components. If *CAN_ChipReset()* returns an error,

can_open() exits with an error. *CAN_StartChip()* is called to begin CAN bus activities using the AIM104-CAN board. A message indicating that the *can_open()* function has been entered is printed to stderr. Finally, *iofunc_default_open()* is called with the original parameters and returned to the calling function to ensure that all of the correct QNX operations have taken place.

`__LDDK_CLOSE_TYPE can_close(__LDDK_CLOSE_PARAM), “can_close.c”`

can_close() is the function that is responsible for closing communication between the AIM104-CAN controller and the thread calling the *close()* function. The first operation *can_close()* performs is printing a message to stderr to announce that *can_close()* has been called. *CAN_FreeIrq()* is called to sever the interrupt request line between the CAN core and the kernel. Once the interrupts have been disconnected, *CAN_StopChip()* is called to take the AIM104-CAN offline. *munmap_device_io()* is called to free the memory mapped device I/O space that was used by the CAN board. *can_close()* calls and returns *iofunc_close_ocb_default()* to ensure that all of the QNX operations initiated in the *iofunc_default_open()* are appropriately handled.

`__LDDK_READ_TYPE can_read(__LDDK_READ_PARAM), “can_read.c”`

can_read() is a device I/O function that retrieves CAN messages from the receive queue. *can_read()* prints a message to stderr announcing that it is running before anything else occurs. *can_read()* calls *iofunc_read_verify()* to ensure that the CAN device is opened in a manner that permits reading. If *iofunc_read_verify()* returns a failure status, *can_read()* exits with an error. This resource manager does not support any special xtypes, so the new *read()* function checks for xtypes in the message and exits

with an error, if xtypes are being used. Once the precautions have been taken, *can_read()* checks to see if there are any new messages. If no new messages have been placed in the receive queue, *can_read()* replies with a message of zero bytes. If a message is waiting in the receive queue, *can_read()* replies with a message pointing to the dequeued message and sets status flags. *can_read()* returns a value to the calling function that all message replies have already been handled.

__LDDK_WRITE_TYPE can_write(__LDDK_WRITE_PARAM), “can_write.c”

can_write() is a device I/O function that is responsible for sending CAN message using the AIM104-CAN. *can_write()*'s first course of action is to print an announcement to stderr indicating that *can_write()* is being run. *iofunc_write_verify()* is called to ensure that the CAN device has been opened for writing. If the CAN device is not opened for writing, the function replies with an error message and exits. Otherwise, xtypes are checked for, and if xtypes are being used, *can_write()* exits with an error. *can_write()* obtains the message to be sent using *resmgr_msgread()*. If the message is read correctly, the requested number of bytes is stored in a *canmsg_t* structure. If an error occurs while reading the message, *can_write()* exits with an error. At this point interrupts are disabled. If no message are waiting in the transmit queue, the message is sent immediately using the *CAN_SendMessage()* function. If message are waiting to be transmitted, the current message is enqueued in the transmit queue. Interrupts are enabled; status flags are set; and *can_write()* exits with a message indicating that the write has been accomplished.

__LDDK_DEVCTL_TYPE can_devctl(__LDDK_DEVCTL_PARAM), “can_devctl.c”

can_devctl() has not been tested; although, the code needed for functionality has

been written. *can_devctl()* allows low level configuration of the AIM104-CAN to be accomplished. *can_devctl()* uses several helper functions that are also contained in “can_devctl.c”. Data structures for *can_devctl()* are defined as types in “can4qnx.h”.

**int test_devctl(resmgr_context_t *ctp, io_devctl_t *msg, RESMGR_OCB_T *ocb),
“can_resmgr.c”**

test_devctl() is a test function that calls and returns the default

iofunc_devctl_default() function that is supplied by QNX. *test_devctl()* prints a message to stderr indicating that it has run.

**int test_close_dup(resmgr_context_t *ctp, io_devctl_t *msg, RESMGR_OCB_T
*ocb), “can_resmgr.c”**

test_close_dup() is a test function that calls and returns the default

iofunc_close_dup_default() function that is supplied by QNX. *test_close_dup()* prints a message to stderr indicating that it has run.

can4qnx Resource Manager

void main(int argc, char **argv), “can_resmgr.c”

The *main()* function for the can4qnx resource manager handles all the requests for device access. The arguments passed to main are for flags appended to the executable in order to configure the device at the time the resource manager is started. No flags are implemented in the current resource manager. The resource manager’s *main()* function prints a message to stderr indicating that it has begun to run. *main()*’s functionality is given in more detail in Chapter Four of this paper.

void cleanup_module(void), “can_resmgr.c”

cleanup_module() is a function that disassociates the CAN device’s resource manager from the handling threads and path names. Since the operations in this function are synthesized when the resource manager’s thread is terminated, this function does not need to be called. In fact, the logistics of calling this function are tedious because the resource manager cannot detach itself, and once the resource manager’s thread has been terminated, no thread exists to call *cleanup_module()*.

*Miscellaneous Files and Functions***int Error(int err), “can_error.c”**

Error() is a legacy function held over from can4linux. It simply sets a global variable *Can_errno*, also declared in “can_error.c”, to *err*. *Can_errno*’s default value is zero. *Error()* returns a zero upon completion.

int Can_RequestIrq(int irq, void *handler), “can_util.c”

Can_RequestIrq() is a function that attaches *handler()* to IRQ *irq*. Since *InterruptAttachEvent()* is being used in this resource manager, this code is not run. If it were run, *interruptID* would be assigned the return value from *InterruptAttach()*. Upon success, *IRQ_requested* would be set to one, and *Can_RequestIrq()* would return a one. Upon failure, a message is printed to stderr, *IRQ_requested* is set to zero, and a zero is returned.

int Can_FreeIrq(int irq), “can_util.c”

Can_FreeIrq() calls *InterruptDetach()* to disassociate *interruptID* from *irq*.

IRQ_requested is reset to zero, and the function returns a zero.

int CanBuf_Init(canbuf_t *mq) , “can_util.c”

CanBuf_Init() is responsible for initializing the *canbuf_t* structure pointed to by *mq*. *CanBuf_Init()* sets the head, tail, and count elements of *mq* to zero and then returns a zero.

int CanBuf_InQ(canbuf_t *mq, canmsg_t *cm) , “can_util.c”

CanBuf_InQ() places *cm* into *mq* if *mq* is not full. If *mq* is full, a “-1” is returned; otherwise, *CanBuf_InQ()* returns a zero.

int CanBuf_DeQ(canbuf_t *mq, canmsg_t *cm) , “can_util.c”

CanBuf_DeQ() places the next message in *mq* into the *canmsg_t* structure pointed to by *cm*. If there are no messages in the queue, a “-1” is returned. If a message is placed in *mq*, a zero is returned.

int CanBuf_Count(canbuf_t *mq) , “can_util.c”

CanBuf_Count() simply returns the count element of *mq*.

can_util.c

can_util.c contains the functions above as well as the receive and transmit queues, *Rx_buff* and *Tx_buff*, *interruptID*, and *IRQ_requested*.

can_sysctl.c

can_sysctl.c contains several global variables. *IRQ* is an *int* that holds the interrupt used by the CAN device. *Base* is an *int* value that holds the base address of the CAN device. *Length* is an *int* value that specifies the amount of memory to map to device I/O. *Baud* is an *int* value that holds the current baud of the CAN bus. *can_dev* is a *uintptr_t* variable that holds the pointer to the device's base address. Error counters are also declared and initialized in this file.

can_defs.h

can_defs.h is a header file for can4qnx. *can_defs.h* holds function prototypes for all can4qnx functions. *can_defs.h* also defines most global variables as *extern* to cut down on the number of files that need to be included in program files. Finally, *can_defs.h* contains the macros for communicating with the CAN device. These macros are *CANin()*, *CANout()*, *CANset()*, *CANreset()*, and *CANtest()*.

can_sja1000.h

can_sja1000.h defines communication tools for use with the SJA1000 chipset. The *canregs_t* structure for register access is defined here. *can_sja1000.h* also defines bits for each register. Finally, the values for the SJA1000's timing registers are defined here by system clock frequency.

can4qnx.h

can4qnx.h defines many of the higher level CAN structures and CAN message bits. The *canmsg_t* structure is defined in this file along with flags transmitted in CAN messages. The *devctl()* structures and flags are also defined in *can4qnx.h*.

APPENDIX B

Appendix Documenting CCanCore Code

CAN Core Class Access Functions

CCanCore::CCanCore(CInterruptCont *pIntCont, unsigned long base, unsigned char vector), “ccancore.cxx”

CCanCore() is the constructor for the *CCanCore* class. When an instance of *CCanCore* is instantiated, three parameters can be passed to it. *pIntCont* is a pointer to the interrupt controller class, and must be passed to the instantiation of *CCanCore*. *base* is the base address for the CAN core, and defaults to the value of *CAN_BASE* defined in “robot2.h”. *vector* is the interrupt vector to use for CAN core interrupts. *vector* defaults to a value of 17 in the case that it is not passed to the instantiation. *CCanCore()* sets default values for the CAN core and calls the *CCanCore* method *SetInterruptCont()* to set up CAN interrupts.

CCanCore::~~CCanCore(), “ccancore.cxx”

~CCanCore() is the destructor for the *CCanCore* class. *~CCanCore()* is responsible for stopping the CAN core and unregistering the CAN interrupt with the interrupt controller.

void CCanCore::SetInterruptCont(CInterruptCont *pIntCont), “ccancore.cxx”

SetInterruptCont() is responsible for registering the *CCanCore* class with the interrupt controller and unmasking the interrupt. *SetInterruptCont()* saves the pointer to the interrupt controller in a private variable as well.

void CCanCore::GetStat(CanStatusPar_t *stat), “ccancore.cxx”

GetStat() returns various CAN core status indicators in the structure pointed to by *stat*. The *CanStatusPar_t* is defined in “gbaCAN.h”.

int CCanCore::SendMessage(canmsg_t *tx), “ccancore.cxx”

SendMessage() is the *CCanCore* method for transmitting a CAN message. *SendMessage()* reads the message from the structure pointed to by *tx* and puts it on the CAN bus, if the CAN core’s transmit buffer is empty. If the transmit buffer has a message waiting to be transmitted, *tx* is enqueued to the transmit queue. If the transmit buffer is full, the function returns the status of the transmit queue: a “-1” if the queue is full, or a zero if the message was successfully enqueued. If the message is copied to the CAN core’s transmit buffer successfully, the *SendMessage()* returns the number of data bytes copied to the transmit buffer incremented by one.

int CCanCore::GetMessage(canmsg_t *rx), “ccancore.cxx”

GetMessage() dequeues a message from the receive queue and returns the message in the *canmsg_t* structure pointed to by *rx*. *GetMessage()* returns the status of the receive queue, namely, a “-1” if the queue is empty and a zero if a message was copied to *rx*.

int CCanCore::ChipReset(), “ccancore.cxx”

ChipReset() moves the CAN controller into reset mode and initializes appropriate configuration registers to default values. If the CAN core does not respond to the reset request bit being set in the mode register, the function returns a “-1”. Otherwise, *ChipReset()* clears the status register and sets the clock register, the output control register, the timing characteristics, and the baud for the CAN bus. Note that this controller only supports BasicCAN mode. Upon completion, *ChipReset()* returns zero.

void CCanCore::StartChip(), “ccancore.cxx”

StartChip() moves the CAN controller from reset mode to normal, on bus, operation mode. Before removing the reset request bit, *StartChip()* releases the receive buffer, clears any overrun status, clears the interrupts, and specifies the events on which to be interrupted.

void CCanCore::StopChip(), “ccancore.cxx”

StopChip() sets the reset request bit in the mode register of the CAN controller.

void CCanCore::HW_reset(), “ccancore.cxx”

HW_reset() resets the CAN controller using the external reset line. Once *HW_reset()* has completed, the CAN core is in the same state it would be in following a hard reset.

void CCanCore::SetMask(uint8 code, uint8 mask), “ccancore.cxx”

SetMask() sets the acceptance code and acceptance mask registers of the CAN controller. *code* is the *uint8* value specifying the acceptance code, and *mask* is the *uint8* value specifying the acceptance mask.

void CCanCore::SetTiming(int baud) , “ccancore.cxx”

SetTiming() configures the two timing registers on the CAN core. The function switches on *baud*. If *baud* matches a baud that has been defined in “gbaCAN.h”, the appropriate index is set for the *CanTiming* array that is initialized in the same file. The indexed values are then written to the timing registers of the CAN core. If *baud* does not match a predefined value, it is treated as a two 8 bit register values. The upper 8 bits are written to the first timing register, and the lower 8 bits are written to the second timing register. *SetTiming()* also writes the CAN mode to the clock register. Recall that only BasicCAN is supported. This function returns zero upon completion.

void CCanCore::SetOMode(int arg) , “ccancore.cxx”

SetOMode() writes *arg* to the output control register.

int CCanCore::Buf_InQ(canmsg_t *cm, canbuf_t *mq), “ccancore.cxx”

Buf_InQ() is a helper function to ease the enqueueing of CAN message structures from their buffers. *Buf_InQ()* enqueues *cm* into *mq*. A zero is returned on success. If the queue is full, a “-1” is returned.

int CCanCore::Buf_DeQ(canmsg_t *cm, canbuf_t *mq), “ccancore.cxx”

Buf_DeQ() is a helper function to ease the dequeuing of CAN message structures from their buffers. *Buf_DeQ()* dequeues the next message in *mq* into the *canmsg_t* structure pointed to by *cm*. A zero is returned upon a successful dequeue. A “-1” is returned if the queue is empty.

int CCanCore::Buf_Count(canbuf_t *mq), “ccancore.cxx”

Buf_Count() returns the number of CAN messages enqueued in *mq*.

void CCanCore::DefaultInit(), “ccancore.cxx”

DefaultInit() fills the private variables of the *CCanCore* class with default configuration values that will be used in the subsequent configuration of the CAN controller. The default values used are defined in “ccancore.h” except for *CAN_BASE*, which is defined in “robot2.h”.

Interrupt Handler

Interrupt handling on the GBA and Xport is conducted using a public method *Interrupt()* that is called by the Xport’s interrupt handler. *Interrupt()* determines the source of the interrupt and calls one of several helper functions in turn. These helper functions are responsible for addressing the particular interrupt.

void CCanCore::Interrupt(unsigned char vector), “ccancore.cxx”

Interrupt() is the ISR for the CAN controller. *vector* is the interrupt vector for the CAN controller. *vector*’s value is discussed more in Appendix D. *Interrupt()* checks the

CAN core interrupt register and calls the appropriate handler function to service the interrupt.

void CCanCore::readMsgInt(), “ccancore.cxx”

readMsgInt() is the interrupt handler function for the CAN core when a receive message interrupt is generated. As such it retrieves the CAN message from the receive buffer into the receive message queue. If there has been a buffer overrun, the overrun status bit is cleared, and the overrun counter is incremented.

void CCanCore::txBufFreeInt(), “ccancore.cxx”

txBufFreeInt() is the interrupt handler function for the CAN core when the transmit buffer has just sent a message and is free. *txBufFreeInt()* checks the transmit queue to see if a message is waiting to be transmitted. If there is a message in the queue, it is dequeued and copied to the transmit buffer.

void CCanCore::errMsgInt(), “ccancore.cxx”

errMsgInt() is the interrupt handler function for error interrupts. As of now, the only functionality is to increment both the transmit and the receive error counters.

void CCanCore::wokeUpInt(), “ccancore.cxx”

wokeUpInt() is the interrupt handler function for interrupts generated on a CAN core wakeup. The only functionality of this function is to increment the wakeup interrupt counter.

void CCanCore::ovrMsgInt(), “ccancore.cxx”

ovrMsgInt() is the interrupt handler for interrupts generated when a receive buffer overrun has occurred on the CAN core’s receive buffer. The only functionality of this function is to increment the overrun interrupt counter.

Other Files

ccancore.h

“ccancore.h” is the include file for “ccancore.cxx”. It contains the class definition for the *CCanCore* class. “ccancore.h” also contains the definitions for the default initialization values. This gives the user the ability to modify startup values without having to recompile “ccancore.cxx”.

gbaCAN.h

“gbaCAN.h” contains many of the necessary structures and definitions to facilitate development on this CAN core. “gbaCAN.h” defines the type *uint8* that is used to reference registers in the CAN core. This file also defines the structure of *uint8* values used to reference registers in the CAN core. The *canmsg_t*, *canbuf_t*, and *CanStatusPar_t* structures are also created in “gbaCAN.h”. Besides creating structures used with the CAN controller, “gbaCAN.h” defines bit values for each register corresponding to the purpose of the bit. Timing register values for various oscillator frequencies are defined in this file as well and stored in the *CanTiming* array. Finally, the *CANin()*, *CANout()*, *CANset()*, *CANreset()*, and *CANtest()* macros used for register operations are defined.

APPENDIX C

Appendix Documenting Verilog Source

CAN Interface Inputs and Outputs, “CAN.v”

GBA Inputs and Outputs

input [23:0] Addr. The *CAN* module’s input *Addr* is the GBA’s address bus.

The *CAN* module receives its addressing from this bus.

input [7:0] CartData. *CartData* is a register that holds data during writes.

output [7:0] DataRd. *DataRd* is the bus that data is driven onto during reads.

input Wr. *Wr* is the Xport’s write strobe. *Wr* is high during a write of data

CartData to address *Addr*

input Rd. *Rd* is the Xport’s read strobe. *Rd* is high during a read from address space *Addr*. Data is output to *DataRd*.

input Clk. *Clk* is the FPGA’s 50 MHz clock line.

input IrqClr. *IrqClr* is the line the GBA raises to clear a cartridge interrupt.

output can_irq_out. *can_irq_out* is the IRQ line for the CAN controller.

output can_off_bus. *can_off_bus* is raised when the CAN controller is in a bus off state.

CAN Transceiver Inputs and Outputs

input can_rx. *can_rx* is the receive data line from the CAN transceiver to the CAN core.

output can_tx. *can_tx* is the transmit line from the CAN core to the CAN transceiver.

output can_clk_out. *can_clk_out* is the scaled clock output from the CAN core.

Troubleshooting Outputs

output [7:0] can_io. *can_io* is the 8051 bus input to the CAN core.

output can_rst. *can_rst* is the CAN core's reset signal. It is controlled by the CAN module with writes to register 0x20.

output can_ale. *can_ale* is the address latch enable signal to the CAN core.

output can_rd. *can_rd* is the CAN core's read strobe.

output can_wr. *can_wr* is the CAN core's write strobe.

output can_cs_en. *can_cs_en* is the chip select signal to the CAN core.

output gaterw. *gaterw* is an output of the read and write signal enable register. This signal is useful in checking state machine behavior.

output gateaddr. *gateaddr* is an output of the address signal gate enable. This signal is useful in checking state machine behavior.

output [3:0] state. *state* is an output of the current state register.

CAN Interface Module Summary, "CAN.v"

The various sections of the CAN module and a brief description of their function are listed below in the order they appear in "CAN.v".

OpenCores.org CAN Core

The OpenCores.org CAN core module named *can_top* is instantiated here. *rst_i*, the reset line, is tied to *can_rst*. *ale_i*, *rd_i*, and *wr_i* are bus control signals and are tied

to *can_ale*, *can_rd*, and *can_wr*, respectively. *port_0_io* is the bus used by *can_top*.

cs_can_i is the chip select signal for the CAN core. *clk_i* is the CAN core clock signal.

rx_i and *tx_o* are the receive and transmit signals from and to the CAN transceiver.

bus_off_on is the signal output to indicate a bus off condition in the controller. *irq_on* is the IRQ output from the CAN core. Finally, *clkout_o* is the scaled clock signal output.

CAN Core Reset

The reset signal to the CAN core is controlled here. Whenever a write to the 0x20 register of address space mapped to the CAN control is generated, the *can_rst* register receives the least significant bit of *CartData*. *can_rst* is in turn wired to the *rst_i* input on *can_top*.

IRQ Interface

A simple IRQ mask is implemented in hardware to interface the Xport and OpenCores.org CAN controller interrupt handling. The *can_irq_out* register is controlled using the *can_irq*, *can_rst*, *IrqClr*, and *can_irq_mask* signals. The methodology is described in Chapter Five of this document.

Bus Interface Combinational Logic

Some combinational logic is implemented in this section. *can_cs_en*, *can_rd*, *can_wr*, *can_io*, and *DataRd* are all given values using continuous assignment statements.

Bus Interface State Machine

The bus interface state machine employs two *always* blocks. The first is the next state logic. Next state logic simply updates the state based on the reset signal or the *next_state* register assigned in the state logic. The second section is the state logic. State logic generates control signals to gate the appropriate signals to and from the *can_io* bus. The states and signals are discussed in Chapter Five of this document.

Xport Primary Module, “can_bot.v”

The first module instantiated in this file is the Xport’s primary module. It produces and receives the busses used by the CAN controller. The interrupt vector numbering is worth mentioning here. The *vector* parameter passed to the *Register* method of the *CInterruptCont* class is determined by the interrupt request line's position in this module’s *IntStatus* bus. The least significant bit of *IntStatus* corresponds to a *vector* value of 16. Continuing, the second bit in *IntStatus* is 17 and so on until the most significant bit of *IntStatus*.

CAN Controller Module, “can_bot.v”

The *CAN* module is instantiated here. The *can_rx* and *can_tx* signal are output to unused Xport I/O pins *PB[6]* and *PB[0]*. Specific signals for the *CAN* module are enumerated in the previous sections of this Appendix.

Other Modules, “can_bot.v”

Two other modules are instantiated in “can_bot.v”. The *BemfCont4* controller is a motor controller used by the Xport. Continuous assignment statements augmenting its use are also in its section. The controller module for the optical encoders, *Encoder*, is

also instantiated here along with its continuous assignment statements. Read handling for the various modules is implemented at the bottom of the “can_bot.v” source.

APPENDIX D

can4qnx Source Code

can_close.c

```
/*
 * can_close - can4linux CAN driver module
 *
 * version 1.2
 * 3-8-2005
 */

#include <can_defs.h>
#include <stdio.h>
#include <sys/iofunc.h>
#include <sys/mman.h>

extern uintptr_t can_dev;
extern int interruptID;
extern int IRQ;

__LDDK_CLOSE_TYPE can_close ( __LDDK_CLOSE_PARAM ) {

    fprintf(stderr, "In can_close.\n");

    Can_FreeIrq(IRQ);
    CAN_StopChip();
    munmap_device_io(can_dev, Length);

#ifdef CAN_USE_FILTER
    Can_FilterCleanup();
#endif

    return(iofunc_close_ocb_default(ctp, reserved, ocb));
}
```

can_defs.h

```
/*
 * can_defs.h
 *
 * version 1.2
 * 3-8-2005
 */
#include <sys/resmgr.h>

// #define CAN_USE_FILTER
```

```

#define _BUS_TYPE "ISA-"

#define __LDDK_WRITE_TYPE      int
#define __LDDK_CLOSE_TYPE     int
#define __LDDK_READ_TYPE      int
#define __LDDK_OPEN_TYPE      int
#define __LDDK_DEVCTL_TYPE    int
#define __LDDK_SELECT_TYPE    unsigned int

#define __LDDK_SEEK_PARAM      \
    resmgr_context_t *ctp, io_lseek_t *msg, RESMGR_OCB_T *ocb
#define __LDDK_READ_PARAM      \
    resmgr_context_t *ctp, io_read_t *msg, RESMGR_OCB_T *ocb
#define __LDDK_WRITE_PARAM     \
    resmgr_context_t *ctp, io_write_t *msg, RESMGR_OCB_T *ocb
#define __LDDK_READDIR_PARAM
#define __LDDK_SELECT_PARAM
#define __LDDK_DEVCTL_PARAM    \
    resmgr_context_t *ctp, io_devctl_t *msg, RESMGR_OCB_T *ocb
#define __LDDK_MMAP_PARAM      \
    resmgr_context_t *ctp, io_mmap_t *msg, RESMGR_OCB_T *ocb
#define __LDDK_OPEN_PARAM      \
    resmgr_context_t *ctp, io_open_t *msg,      \
    RESMGR_HANDLE_T *handle, void *extra
#define __LDDK_FLUSH_PARAM
#define __LDDK_CLOSE_PARAM     \
    resmgr_context_t *ctp, void *reserved, RESMGR_OCB_T *ocb
#define __LDDK_FSYNC_PARAM     \
    resmgr_context_t *ctp, io_sync_t *msg, RESMGR_OCB_T *ocb
#define __LDDK_FASYNC_PARAM
#define __LDDK_CCHECK_PARAM    \
    kdev_t dev
#define __LDDK_REVAL_PARAM     \
    kdev_t dev

#define __CAN_TYPE__ _BUS_TYPE "PeliCAN-port I/O "

/******
extern __LDDK_READ_TYPE can_read (__LDDK_READ_PARAM);
extern __LDDK_WRITE_TYPE can_write (__LDDK_WRITE_PARAM);
extern __LDDK_SELECT_TYPE can_select ( __LDDK_SELECT_PARAM );
extern __LDDK_DEVCTL_TYPE can_devctl ( __LDDK_DEVCTL_PARAM );
extern __LDDK_OPEN_TYPE can_open ( __LDDK_OPEN_PARAM );
extern __LDDK_CLOSE_TYPE can_close (__LDDK_CLOSE_PARAM);

//      ----- Default Outc value for some known boards
//      this depends on the transceiver configuration
//
//      the port board uses optocoupler configuration as denoted
//      in the Philips application notes, so the OUTC value is 0xfa
//      <Walt>
//          normal mode => 0x02
//          Tx0 Pull Dn => 0x08
//          Tx1 Pull Up  => 0x80

#define CAN_OUTC_VAL      0x8A

```

```

#define IO_MODEL          'p'

//*****
#include <can4qnx.h>
//*****

#define MAX_BUFSIZE 200

typedef struct {
    int head;
    int tail;
    int cnt;
    canmsg_t cm[MAX_BUFSIZE];
} canbuf_t;

int CanBuf_Init(canbuf_t*);
int CanBuf_InQ(canbuf_t*, canmsg_t*);
int CanBuf_DeQ(canbuf_t*, canmsg_t*);
int CanBuf_Count(canbuf_t*);

extern canbuf_t Tx_buff;
extern canbuf_t Rx_buff;

#ifdef CAN_USE_FILTER
    #define MAX_ID_LENGTH 11
    #define MAX_ID_NUMBER (1<11)

    typedef struct {
        unsigned    use;
        unsigned    signo[3];
        struct {
            unsigned    enable    : 1;
            unsigned    timestamp : 1;
            unsigned    signal    : 2;
            canmsg_t    *rtr_response;
        } filter[MAX_ID_NUMBER];
    } msg_filter_t;

    extern msg_filter_t Rx_Filter;
#endif

extern unsigned char    *can_base;
extern unsigned int     can_range;

extern int IRQ_requested;
extern int Can_minors;           // used as IRQ dev_id *
extern int Can_IsOpen;

//*****

// ----- Global Definitions for can_dev -----
extern uintptr_t can_dev;

// ----- Global Definitions for version -----
extern char version[];

// ----- Global Definitions for Chpset -----

```

```

extern char Chipset[];

// ----- Global Definitions for IOModel -----
extern char IOModel;

// ----- Global Definitions for IRQ -----
extern int IRQ;

// ----- Global Definitions for Base -----
extern int Base;

// ----- Global Definitions for Length ----
extern int Length;

// ----- Global Definitions for Baud -----
extern int Baud;

// ----- Global Definitions for AccCode -----
extern unsigned int AccCode;

// ----- Global Definitions for AccMask -----
extern unsigned int AccMask;

// ----- Global Definitions for Timeout -----
extern int Timeout;

// ----- Global Definitions for Outc -----
extern int Outc;

// ----- Global Definitions for TxErr -----
extern int TxErr;

// ----- Global Definitions for RxErr -----
extern int RxErr;

// ----- Global Definitions for Overrun -----
extern int Overrun;

// ----- Global Definitions for dbgMask -----
extern unsigned int dbgMask;

// ----- Global Definitions for Test -----
extern int Cnt1;
extern int Cnt2;

/*****
extern int Can_errno;

/*****
// function prototypes
// can_sja1000funcs.c
/*****
extern int CAN_ChipReset();
extern int CAN_SetTiming(int);
extern int CAN_StartChip();
extern int CAN_StopChip();
extern int CAN_GetStat(__LDDK_DEVCTL_PARAM, CanStatusPar_t *);

```



```

extern int CAN_SetMask(unsigned int, unsigned int);
extern int CAN_SetOMode(int);
extern int CAN_SendMessage(canmsg_t *);
extern int CAN_GetMessage(canmsg_t *);
extern int CAN_VendorInit();
void CAN_loopLEDs();
void *CAN_Interrupt ( void *);

// util.c
extern int Can_FifoInit();
extern int Can_FilterCleanup();
extern int Can_FilterInit();
extern int Can_FilterMessage(unsigned message, unsigned enable);
extern int Can_FilterOnOff(unsigned on);
extern int Can_FilterSigNo(unsigned signo, unsigned signal);
extern int Can_FilterSignal(unsigned id, unsigned signal);
extern int Can_FilterTimestamp(unsigned message, unsigned stamp);
extern int Can_FreeIrq(int irq );
extern void Can_StartTimer(unsigned long v);
extern void Can_StopTimer(void);
extern void Can_TimerInterrupt(unsigned long unused);
extern void Can_dump();
extern void print_tty(const char *fmt, ...);

//*****
// hardware access functions or macros
//*****

// #error Intel port I/O access
// using port I/O with in8()/out8() for Intel architectures like
//   AT-CAN-MINI ISA board

#define CANout(adr,v)    out8((uintptr_t) &((canregs_t *) \
                           can_dev)->adr ,v)

#define CANin(adr)      (in8 ((uintptr_t) &((canregs_t *) \
                           can_dev)->adr  ))

#define CANset(adr,m)    out8( (uintptr_t) &((canregs_t *) \
                           can_dev)->adr, in8((uintptr_t) &((canregs_t *) \
                           can_dev)->adr) | m)

#define CANreset(adr,m) out8( (uintptr_t) &((canregs_t *) \
                           can_dev)->adr, in8((uintptr_t) &((canregs_t *) \
                           can_dev)->adr) & ~m)

#define CANTest(adr,m)  in8((uintptr_t) &((canregs_t *) \
                           can_dev)->adr  ) & m

```

can_devctl.c

```

/*
 * can_devctl - can4linux CAN driver module
 *
 * version 1.2

```

```

* 3-8-2005
*/

#include <stdio.h>
#include <sys/iosfunc.h>
#include <errno.h>
#include <stdlib.h>
#include <can_defs.h>

int can_Command(__LDDK_DEVCTL_PARAM);
int can_Send(__LDDK_DEVCTL_PARAM, canmsg_t*);
int can_Receive(__LDDK_DEVCTL_PARAM, canmsg_t*);
int can_Config(__LDDK_DEVCTL_PARAM, int, unsigned long int, unsigned
long int);

__LDDK_DEVCTL_TYPE can_devctl( __LDDK_DEVCTL_PARAM ) {
    char *argp;
    int retval = -1;
    int nbytes;
    int off;
    int doffset;

    fprintf(stderr, "in dev_ctl.\n");
    Can_errno = 0;
    nbytes = msg->i.nbytes;
    off = ocb -> offset;
    doffset = sizeof(msg -> i);

    switch(msg->i.dcmd) {
        case COMMAND:
            argp = malloc ( sizeof(Command_par_t)+1);
            if (argp == NULL) return ENOMEM;
            if (resmgr_msgread (ctp, argp, nbytes, doffset)==-1)
                return(retval);
            ((Command_par_t *) argp)->retval=
                can_Command(ctp, msg, ocb);
            ((Command_par_t *) argp)->error = Can_errno;
            memcpy ((Command_par_t *)msg->i.zero, (void *)argp,
                sizeof(Command_par_t));
            free(argp);
            break;
        case CONFIG:
            argp = malloc ( sizeof(Config_par_t) +1);
            if (argp == NULL) return ENOMEM;
            if (resmgr_msgread (ctp, (void *)argp,
                nbytes, doffset)==-1)
                return(retval);
            ((Config_par_t *) argp)->retval =
                can_Config(ctp, msg, ocb,
                    ((Config_par_t *) argp)->target,
                    ((Config_par_t *) argp)->val1,
                    ((Config_par_t *) argp)->val2 );
            ((Config_par_t *) argp)->error = Can_errno;
            memcpy ((Config_par_t *)msg->i.zero,
                (void *)argp, sizeof(Config_par_t));
            free(argp);
            break;
    }
}

```

```

case SEND:
    argp = malloc( sizeof(Send_par_t) +1);
    if (argp == NULL) return ENOMEM;
    if (resmgr_msgread (ctp, (void *)argp,
                        nbytes, doffset)==-1)
        return(retval);
    ((Send_par_t *) argp)->retval =
        can_Send(ctp, msg, ocb,
                 ((Send_par_t *) argp)->Tx );
    ((Send_par_t *) argp)->error = Can_errno;
    memcpy( (Send_par_t *) msg->i.zero, (void *)argp,
            sizeof(Send_par_t));
    free(argp);
    break;
case RECEIVE:
    argp = malloc( sizeof(Receive_par_t) +1);
    if (argp == NULL) return ENOMEM;
    if (resmgr_msgread (ctp, (void *)argp,
                        nbytes, doffset)==-1)
        return(retval);
    ((Receive_par_t *) argp)->retval =
        can_Receive(ctp, msg, ocb,
                    ((Receive_par_t *) argp)->Rx);
    ((Receive_par_t *) argp)->error = Can_errno;
    memcpy( (Receive_par_t *)msg->i.zero, (void *) argp,
            sizeof(Receive_par_t));
    free(argp);
    break;
case STATUS:
    argp = malloc( sizeof(CanStatusPar_t) +1);
    if (argp == NULL) return ENOMEM;
    ((CanStatusPar_t *) argp)->retval = CAN_GetStat(ctp,
                                                    msg, ocb, ((CanStatusPar_t *)argp));
    memcpy( (CanStatusPar_t *)msg->i.zero, (void *) argp,
            sizeof(CanStatusPar_t));
    free(argp);
    break;
case LOOPLEDS:
    CAN_loopLEDS();
    break;

#ifdef CAN_RTR_CONFIG

case CONFIGURERTR:
    argp = malloc( sizeof(CanStatusPar_t) +1);
    if (argp == NULL) return ENOMEM;
    if (resmgr_msgread (ctp, (void *)argp,
                        nbytes, doffset)==-1)
        return(retval);
    ((ConfigureRTR_par_t *) argp)->retval =
        can_ConfigureRTR(inode,
                         ((ConfigureRTR_par_t *) argp)->message,
                         ((ConfigureRTR_par_t *) argp)->Tx );
    ((ConfigureRTR_par_t *) argp)->error = Can_errno;
    memcpy( (ConfigureRTR_par_t *) msg->i.zero,
            (void *) argp, sizeof(ConfigureRTR_par_t));
    free(argp);

```

```

        break;

    #endif        // CAN_RTR_CONFIG

    default:
        return -1;
}
return 1;
}

// ioctl functions are following here

int can_Command(__LDDK_DEVCTL_PARAM )    {
    switch (msg->i.dcmd) {
        case CMD_START:
            CAN_StartChip();
            break;
        case CMD_STOP:
            CAN_StopChip();
            break;
        case CMD_RESET:
            CAN_ChipReset();
            break;
        default:
            return(-1);
    }
    return 0;
}

// is not very useful! use it if you are sure the tx queue is empty
int can_Send(__LDDK_DEVCTL_PARAM , canmsg_t *Tx)    {
    canmsg_t tx;

    if (resmgr_msgread (ctp, Tx, msg->i.nbytes, sizeof(msg->i)) == -1)
        return -1;
    memcpy((canmsg_t *) &tx, (canmsg_t *) Tx, sizeof(canmsg_t));
    return CAN_SendMessage(&tx);
}

int can_Receive(__LDDK_DEVCTL_PARAM , canmsg_t *Rx)    {
    canmsg_t rx;
    int len;

    len = CAN_GetMessage(&rx);

    if( len > 0 ){
// printf("\nrx[ ] got id 0x%x len %d\n", rx.id, rx.length, Rx);

        if (resmgr_msgread (ctp, Rx, msg->i.nbytes,
                                sizeof(msg->i)) == -1)
            return -1;
        return rx.length;
    } else { // no message available
        return 0;
    }
}

```

```

    }
}

int can_Config(__LDDK_DEVCTL_PARAM , int target, unsigned long val1,
               unsigned long val2)    {

    switch(target) {
        case CONF_ACC:
            AccMask = val1;
            AccCode = val2;
            CAN_SetMask( AccCode, AccMask);
            break;
        case CONF_ACCM:
            AccMask = val1;
            CAN_SetMask( AccCode, AccMask);
            break;
        case CONF_ACCC:
            AccCode = val1;
            CAN_SetMask( AccCode, AccMask);
            break;
        case CONF_TIMING:
            Baud = val1;
            CAN_SetTiming((int) val1);
            break;
        case CONF_OMODE:
            CAN_SetOMode( (int) val1);
            break;
#ifdef CAN_USE_FILTER
        case CONF_FILTER:
            Can_FilterOnOff( (int) val1 );
            break;
        case CONF_FENABLE:
            Can_FilterMessage( (int) val1, 1);
            break;
        case CONF_FDISABLE:
            Can_FilterMessage( (int) val1, 0);
            break;
#endif
        default:
            return(-1);
    }
    return 0;
}

```

can_error.c

```

/* Can_error
 *
 * version 1.2
 * 3-8-2005
 */
#include <can_defs.h>

int Can_errno = 0;

```

```

int Error(int err)      {
    Can_errno = err;
    return 0;
}

```

can_open.c

```

/*
 * can_open - can4linux CAN driver module
 *
 * version 1.2
 * 3-8-2005
 */

#include <stdio.h>
#include <sys/iofunc.h>
#include <errno.h>
#include <can_defs.h>

extern iofunc_attr_t attr;

__LDDK_OPEN_TYPE can_open( __LDDK_OPEN_PARAM ) {
    int lasterr;

    if (attr.count > 0)      return EMFILE;

    if( (lasterr = CAN_VendorInit()) < 0 ) {
        fprintf(stderr, "lasterr TRUE\n");
        return(ENOENT);
    }

    CanBuf_Init(&Rx_buff);
    CanBuf_Init(&Tx_buff);

    #if CAN_USE_FILTER
        Can_FilterInit();
    #endif

    if( CAN_ChipReset() < 0 ) {
        fprintf(stderr, "CAN_ChipReset < 0\n");
        return(ENOENT);
    }

    CAN_StartChip();

    fprintf(stderr, "In can_open.\n");

    return(iofunc_open_default(ctp, msg, handle, extra));
}

```

can_read.c

```

/*
 * can_read - can4linux CAN driver module
 *
 * version 1.2
 * 3-7-2005
 */

#include <stdio.h>
#include <sys/iofunc.h>
#include <errno.h>
#include <stdlib.h>
#include <can_defs.h>

extern uintptr_t candev;

__LDDK_READ_TYPE can_read( __LDDK_READ_PARAM ) {
    canmsg_t buffer;
    int status;
    int nbytes;

    fprintf(stderr, "In can_read.\n");

    if ((status = iofunc_read_verify(ctp, msg, ocb, NULL)) != EOK) {
        return (_RESMGR_ERRNO(status));
    }

    //    No special xtypes
    if ((msg->i.xtype & _IO_XTYPE_MASK) != _IO_XTYPE_NONE)
        return (EINVAL);

    nbytes = msg->i.nbytes ;

    if ( status = CanBuf_DeQ(&Rx_buff, &buffer) == -1 ) {
        MsgReply (ctp->rcvid, EOK, NULL, 0);
    } else {
        MsgReply (ctp->rcvid, nbytes, &buffer, nbytes);
        ocb->attr->flags |= IOFUNC_ATTR_ETIME |
IOFUNC_ATTR_DIRTY_TIME;
    }

    return(_RESMGR_NOREPLY);
}

```

can_resmgr.c

```

/*
 * can_core - can4linux CAN driver module
 * version 1.2
 * 3-8-2005
 *
 * Contains the code for module initialization.
 * The functions herein are never called directly by the user
 * but when the driver module is loaded into the kernel space
 * or unloaded.
 *

```

```

* The driver is highly configurable using the \b sysctl interface.
* For a description see main page.
*/

#include <stdio.h>
#include <stdlib.h>
#include <sys/iofunc.h>
#include <sys/dispatch.h>
#include <can_defs.h>

#define INTERRUPT_ATTACH      1

static resmgr_connect_funcs_t c_func;
static resmgr_io_funcs_t      io_func;
static dispatch_t             *dpp;
static int                    resmgr_id;
static iofunc_attr_t          attr;

extern __LDDK_WRITE_TYPE      can_write   (__LDDK_WRITE_PARAM);
extern __LDDK_READ_TYPE       can_read    (__LDDK_READ_PARAM);
extern __LDDK_OPEN_TYPE       can_open    (__LDDK_OPEN_PARAM);
extern __LDDK_CLOSE_TYPE      can_close   (__LDDK_CLOSE_PARAM);
extern __LDDK_DEVCTL_TYPE     can_devctl  (__LDDK_DEVCTL_PARAM);

int test_devctl(resmgr_context_t *ctp, io_devctl_t *msg, RESMGR_OCB_T
*ocb) {
    fprintf (stderr, "test_devctl\n");
    return(iofunc_devctl_default(ctp, msg, ocb));
}

int test_close_dup(resmgr_context_t *ctp, io_close_t *msg, RESMGR_OCB_T
*ocb) {
    fprintf (stderr, "test_close_dup\n");
    return(iofunc_close_dup_default(ctp, msg, ocb));
}

int main(int argc, char **argv)      {
    resmgr_context_t  *ctp;
    resmgr_attr_t      resmgr_attr;

    fprintf(stderr, "In Resource Manager.\n");

    //      initialize the variables laid down in /proc/sys/Can

//      ***** DECLARED IN can_sysctl.c *****
    IOModel      = IO_MODEL;
    Baud         = 125;
    AccCode       = AccMask = 0xffffffff;
    Timeout       = 100;
    Outc          = CAN_OUTC_VAL;
    IOModel       = '\0';

//      ***** DECLARED ABOVE *****
    IRQ_requested = 0;

    ThreadCtl(_NTO_TCTL_IO, 0);    //      give the thread I/O privity

```



```

if ((dpp = dispatch_create ()) == NULL) {
    fprintf (stderr, "%s: Unable to allocate dispatch
             context.\n", "init_module");
    return (EXIT_FAILURE);
}

iofunc_func_init (_RESMGR_CONNECT_NFUNCS, &c_func,
                  _RESMGR_IO_NFUNCS, &io_func);
iofunc_attr_init (&attr, S_IFNAM | 0666, 0, 0);
//     everyone has access to do everything to this file

//     assign io/connect function overrides here
c_func.open      = can_open;
io_func.read     = can_read;
io_func.write    = can_write;
io_func.devctl   = can_devctl;
io_func.close_ocb = can_close;
io_func.close_dup = test_close_dup;

memset (&resmgr_attr, 0, sizeof (resmgr_attr));
resmgr_attr.nparts_max = 1;
resmgr_attr.msg_max_size = 2048;

resmgr_id = resmgr_attach (dpp, &resmgr_attr,
                           "/dev/CAN", _FTYPE_ANY, 0,
                           &c_func, &io_func, &attr);
if (resmgr_id == -1) {
    fprintf(stderr, "%s: Unable to attach name. \n",
            "Resource Manager");
    return EXIT_FAILURE;
}

pthread_create(NULL, NULL, CAN_Interrupt, NULL);

ctp = resmgr_context_alloc (dpp);
while (1) {
    if ((ctp = resmgr_block (ctp)) == NULL) {
        fprintf (stderr, "unable to resmgr_block\n");
        exit (EXIT_FAILURE);
    }
    fprintf(stderr, "Resource Manager -> loop\n");
    resmgr_handler (ctp);
}

return (EXIT_SUCCESS);
}

void cleanup_module(void) {
    int retval;

    retval = resmgr_detach (dpp, resmgr_id, _RESMGR_DETACH_PATHNAME);
}

```

can_sja1000.h

```

/*
 * can_sja1000.h
 *
 * version 1.2
 * 3-8-2005
 */

typedef unsigned char uint8;

union frame {
    struct {
        uint8 canid1;
        uint8 canid2;
        uint8 canid3;
        uint8 canid4;
        uint8 canxdata[8];
    } extframe;
    struct {
        uint8 canid1;
        uint8 canid2;
        uint8 candata[8];
        uint8 dummy1;
        uint8 dummy2;
    } stdframe;
};

typedef struct canregs {
    uint8 canmode;           // 0
    uint8 cancmd;
    uint8 canstat;
    uint8 canirq;
    uint8 canirq_enable;
    uint8 reserved1;        // 5
    uint8 cantim0;
    uint8 cantim1;
    uint8 canoutc;
    uint8 cantest;
    uint8 reserved2;        // 10 (0x0A)
    uint8 arbitrationlost;  // read only
    uint8 errorcode;        // read only
    uint8 errorwarninglimit;
    uint8 rxerror;
    uint8 txerror;          // 15 (0x0F)
    uint8 frameinfo;        // (0x10)
    union frame frame;      // (0x11 to 0x1C)
    uint8 reserved3;        // (0x1D)
    uint8 canrxbufferadr;   // 30 (0x1E)
    uint8 canclk;           // 31 (0x1F)
    uint8 ledreg;           // 32 (0x20)
    uint8 eeprom;           // (0x21)
} canregs_t;

#define CAN_RANGE 0x20

//--- Mode Register ----- PelICAN -----

#define CAN_SLEEP_MODE          0x10 // Sleep Mode

```

```

#define CAN_ACC_FILT_MASK          0x08  // Acceptance Filter Mask
#define CAN_SELF_TEST_MODE        0x04  // Self test mode
#define CAN_LISTEN_ONLY_MODE      0x02  // Listen only mode
#define CAN_RESET_REQUEST         0x01  // reset mode
#define CAN_MODE_DEF CAN_ACC_FILT_MASK // Default ModeRegister Value

// bit numbers of mode register
#define CAN_SLEEP_MODE_BIT        4      // Sleep Mode
#define CAN_ACC_FILT_MASK_BIT     3      // Acceptance Filter Mask
#define CAN_SELF_TEST_MODE_BIT    2      // Self test mode
#define CAN_LISTEN_ONLY_MODE_BIT  1      // Listen only mode
#define CAN_RESET_REQUEST_BIT     0      // reset mode

//--- Interrupt enable Reg -----
#define CAN_ERROR_BUSOFF_INT_ENABLE (1<<7)
#define CAN_ARBTR_LOST_INT_ENABLE  (1<<6)
#define CAN_ERROR_PASSIVE_INT_ENABLE (1<<5)
#define CAN_WAKEUP_INT_ENABLE      (1<<4)
#define CAN_OVERRUN_INT_ENABLE     (1<<3)
#define CAN_ERROR_INT_ENABLE       (1<<2)
#define CAN_TRANSMIT_INT_ENABLE    (1<<1)
#define CAN_RECEIVE_INT_ENABLE     (1<<0)

//--- Frame information register -----
#define CAN_EFF                    0x80  // extended frame
#define CAN_SFF                    0x00  // standard frame format

//--- Command Register -----

#define CAN_GOTO_SLEEP              (1<<4)
#define CAN_CLEAR_OVERRUN_STATUS   (1<<3)
#define CAN_RELEASE_RECEIVE_BUFFER (1<<2)
#define CAN_ABORT_TRANSMISSION     (1<<1)
#define CAN_TRANSMISSION_REQUEST   (1<<0)

//--- Status Register -----

#define CAN_BUS_STATUS              (1<<7)
#define CAN_ERROR_STATUS            (1<<6)
#define CAN_TRANSMIT_STATUS        (1<<5)
#define CAN_RECEIVE_STATUS          (1<<4)
#define CAN_TRANSMISSION_COMPLETE_STATUS (1<<3)
#define CAN_TRANSMIT_BUFFER_ACCESS (1<<2)
#define CAN_DATA_OVERRUN            (1<<1)
#define CAN_RECEIVE_BUFFER_STATUS   (1<<0)

//--- Status Register -----

#define CAN_BUS_STATUS_BIT          (1<<7)
#define CAN_ERROR_STATUS_BIT        (1<<6)
#define CAN_TRANSMIT_STATUS_BIT     (1<<5)
#define CAN_RECEIVE_STATUS_BIT      (1<<4)
#define CAN_TRANSMISSION_COMPLETE_STATUS_BIT (1<<3)
#define CAN_TRANSMIT_BUFFER_ACCESS_BIT (1<<2)
#define CAN_DATA_OVERRUN_BIT        (1<<1)

```

```

#define CAN_RECEIVE_BUFFER_STATUS_BIT      (1<<0)

//--- Interrupt Register -----

#define CAN_WAKEUP_INT      (1<<4)
#define CAN_OVERRUN_INT     (1<<3)
#define CAN_ERROR_INT       (1<<2)
#define CAN_TRANSMIT_INT    (1<<1)
#define CAN_RECEIVE_INT     (1<<0)

//--- Output Control Register -----

#define CAN_OCTP1            (1<<7)
#define CAN_OCTN1            (1<<6)
#define CAN_OCPOL1          (1<<5)
#define CAN_OCTP0            (1<<4)
#define CAN_OCTN0            (1<<3)
#define CAN_OCPOL0          (1<<2)
#define CAN_OCMODE1          (1<<1)
#define CAN_OCMODE0          (1<<0)

//--- Clock Divider register -----

#define CAN_MODE_BASICCAN    (0x00)
#define CAN_MODE_PELICAN    (0xC0)
#define CAN_MODE_CLK         (0x07)      // CLK-out = Fclk
#define CAN_MODE_CLK2        (0x00)      // CLK-out = Fclk/2

//--- Remote Request -----

# define CAN_RTR              (1<<6)

// the base address register array
extern int Base;

//----- Timing values

#define CAN_SYSCLK 8

#if CAN_SYSCLK == 8
    // these timings are valid for clock 8Mhz
    #define CAN_TIM0_10K      49
    #define CAN_TIM1_10K      0x1c
    #define CAN_TIM0_20K      24
    #define CAN_TIM1_20K      0x1c
    #define CAN_TIM0_40K      0x89    // Old Bit Timing Standard of
port
    #define CAN_TIM1_40K      0xEB    // Old Bit Timing Standard of
port
    #define CAN_TIM0_50K      9
    #define CAN_TIM1_50K      0x1c
    #define CAN_TIM0_100K     4        // sp 87%, 16 abtastungen,
sjw 1
    #define CAN_TIM1_100K     0x1c
    #define CAN_TIM0_125K     3

```

```

#define CAN_TIM1_125K      0x1c
#define CAN_TIM0_250K      1
#define CAN_TIM1_250K      0x1c
#define CAN_TIM0_500K      0
#define CAN_TIM1_500K      0x1c
#define CAN_TIM0_800K      0
#define CAN_TIM1_800K      0x16
#define CAN_TIM0_1000K     0
#define CAN_TIM1_1000K     0x14

#define CAN_SYSCLK_is_ok   1
#endif

#if CAN_SYSCLK == 10
    // these timings are valid for clock 10Mhz
    // 20 Mhz crystal
#define CAN_TIM0_10K        0x31
#define CAN_TIM1_10K        0x2f
#define CAN_TIM0_20K        0x18
#define CAN_TIM1_20K        0x2f
#define CAN_TIM0_50K        0x18
#define CAN_TIM1_50K        0x05
#define CAN_TIM0_100K       0x04
#define CAN_TIM1_100K       0x2f
#define CAN_TIM0_125K       0x04
#define CAN_TIM1_125K       0x1c
#define CAN_TIM0_250K       0x04
#define CAN_TIM1_250K       0x05
#define CAN_TIM0_500K       0x00
#define CAN_TIM1_500K       0x2f
#define CAN_TIM0_800K       0x00
#define CAN_TIM1_800K       0x00
#define CAN_TIM0_1000K      0x00
#define CAN_TIM1_1000K      0x07

#define CAN_SYSCLK_is_ok    1
#endif

#if CAN_SYSCLK == 16
    // these timings are valid for clock 16Mhz
#define CAN_TIM0_10K        0x1F
#define CAN_TIM1_10K        0x7F
#define CAN_TIM0_20K        0x0F
#define CAN_TIM1_20K        0x7F
#define CAN_TIM0_50K        0x07
#define CAN_TIM1_50K        0x7A
#define CAN_TIM0_100K       0x03
#define CAN_TIM1_100K       0x7A
#define CAN_TIM0_125K       0x03
#define CAN_TIM1_125K       0x76
#define CAN_TIM0_250K       0x01
#define CAN_TIM1_250K       0x76
#define CAN_TIM0_500K       0x00
#define CAN_TIM1_500K       0x76
#define CAN_TIM0_800K       0x00

```

```

#define CAN_TIM1_800K          0x43
#define CAN_TIM0_1000K        0x00
#define CAN_TIM1_1000K        0x32

#define CAN_SYSCCLK_is_ok  1
#endif

#ifndef CAN_SYSCCLK_is_ok
    #error Please specify a valid CAN_SYSCCLK value (i.e. 8, 10, 16)
or \
    define new parameters
#endif

//      LEDs
#define OFF          0x00

#define YEL0         0x3C
#define RED0         0x3E
#define GRN0         0x3D

#define YEL1         0x33
#define GRN1         0x37
#define RED1         0x3B

#define YEL2         0x0F
#define GRN2         0x1F
#define RED2         0x2F

```

can_sja1000funcs.c

```

/* can_sja1000funcs
 *
 * version 1.2
 * 3-8-2005
 */

#include <stdio.h>
#include <sys/iosfunc.h>
#include <hw/inout.h>
#include <unistd.h>
#include <sys/mman.h>
#include <can_defs.h>
#include "can_sja1000.h"

extern uintptr_t can_dev;
extern int interruptID;

// timing values
uint8 CanTiming[10][2]={
    {CAN_TIM0_10K,CAN_TIM1_10K},
    {CAN_TIM0_20K,CAN_TIM1_20K},
    {CAN_TIM0_50K,CAN_TIM1_50K},
    {CAN_TIM0_100K, CAN_TIM1_100K},

```



```

        CAN_OVERRUN_INT_ENABLE |
        CAN_ERROR_INT_ENABLE |
        CAN_TRANSMIT_INT_ENABLE |
        CAN_RECEIVE_INT_ENABLE);

// Board specific output control
if (Outc == 0) {
    Outc = CAN_OUTC_VAL;
}
CANout(canoutc, Outc);

CAN_SetTiming(Baud);
CAN_SetMask( AccCode, AccMask );
// Can_dump();
return 0;
}

int CAN_SetTiming (int baud) {
    int i = 5;
    int custom=0;

    switch(baud) {
        case 10:  i = 0;      break;
        case 20:  i = 1;      break;
        case 50:  i = 2;      break;
        case 100: i = 3;      break;
        case 125: i = 4;      break;
        case 250: i = 5;      break;
        case 500: i = 6;      break;
        case 800: i = 7;      break;
        case 1000: i = 8;     break;
        default:  custom=1;
    }

    // select mode: Basic or Pelican
    CANout(canclk, CAN_MODE_PELICAN + CAN_MODE_CLK);
    if( custom ) {
        CANout(cantim0, (uint8) (baud >> 8) & 0xff);
        CANout(cantim1, (uint8) (baud & 0xff));
        fprintf(stderr, " custom bit timing BT0=0x%x BT1=0x%x ",
                    CANin(cantim0), CANin(cantim1));
    } else {
        CANout(cantim0, (uint8) CanTiming[i][0]);
        CANout(cantim1, (uint8) CanTiming[i][1]);
    }

    return 0;
}

int CAN_StartChip () {
    RxErr = TxErr = 0L;

    CANout(cancmd, (CAN_RELEASE_RECEIVE_BUFFER |
                    CAN_CLEAR_OVERRUN_STATUS) );

```



```

        usleep(10);

        // clear interrupts
        CANin(canirq);

        // Interrupts on Rx, TX, any Status change and data overrun
        CANset(canirq_enable, (CAN_OVERRUN_INT_ENABLE
                                + CAN_ERROR_INT_ENABLE
                                + CAN_TRANSMIT_INT_ENABLE
                                + CAN_RECEIVE_INT_ENABLE ));

        CANreset( canmode, CAN_RESET_REQUEST );
        return 0;
    }

int CAN_StopChip ()
{
    CANset(canmode, CAN_RESET_REQUEST );
    return 0;
}

// set value of the output control register
int CAN_SetOMode (int arg)
{
    CANout(canoutc, arg);

    return 0;
}

int CAN_SetMask (unsigned int code, unsigned int mask)
{
    CANout(frameinfo, (unsigned char)((unsigned int)(code >> 24)
                                         & 0xff));

    CANout(frame.extframe.canid1,
            (unsigned char)((unsigned int)(code >> 16) & 0xff));
    CANout(frame.extframe.canid2,
            (unsigned char)((unsigned int)(code >> 8) & 0xff));
    CANout(frame.extframe.canid3,
            (unsigned char)((unsigned int)(code >> 0 ) & 0xff));
    CANout(frame.extframe.canid4,
            (unsigned char)((unsigned int)(mask >> 24) & 0xff));
    CANout(frame.extframe.canxdata[0],
            (unsigned char)((unsigned int)(mask >> 16) & 0xff));
    CANout(frame.extframe.canxdata[1],
            (unsigned char)((unsigned int)(mask >> 8) & 0xff));
    CANout(frame.extframe.canxdata[2],
            (unsigned char)((unsigned int)(mask >> 0) & 0xff));

    return 0;
}

int CAN_SendMessage ( canmsg_t *tx) {
    int i = 0;
    int ext;           // message format to send
    uint8 tx2reg, stat;

```

```

InterruptDisable();

if ( ! (stat=CANin(canstat)) & CAN_TRANSMIT_BUFFER_ACCESS ) {
    CanBuf_InQ(&Tx_buff, tx);
    //      put in Tx_buff b/c chip is busy
    return 0;
}

tx->length %= 9;          // limit CAN message length to 8
ext = (tx->flags & MSG_EXT); // read message format

// fill the frame info and identifier fields
tx2reg = tx->length;
if( tx->flags & MSG_RTR)      tx2reg |= CAN_RTR;

if(ext) {
    CANout(frameinfo, CAN_EFF + tx2reg);
    CANout(frame.extframe.canid1, (uint8)(tx->id >> 21));
    CANout(frame.extframe.canid2, (uint8)(tx->id >> 13));
    CANout(frame.extframe.canid3, (uint8)(tx->id >> 5));
    CANout(frame.extframe.canid4, (uint8)(tx->id << 3) & 0xff);
} else {
    CANout(frameinfo, CAN_SFF + tx2reg);
    CANout(frame.stdframe.canid1, (uint8)((tx->id) >> 3) );
    CANout(frame.stdframe.canid2, (uint8)(tx->id << 5 )
    & 0xe0);
}

// - fill data -----
if(ext) {
    for( i=0; i < tx->length ; i++) {
        CANout( frame.extframe.canxdata[i], tx->data[i]);
        usleep(10);
    }
} else {
    for( i=0; i < tx->length ; i++) {
        CANout( frame.stdframe.candata[i], tx->data[i]);
        usleep(10);
    }
}
// - /end -----
CANout(cancmd, CAN_TRANSMISSION_REQUEST );

InterruptEnable();

return i;
}

int CAN_GetMessage ( canmsg_t *rx ) {
    uint8 dummy = 0, stat;
    int i = 0, ext, status;
    stat = CANin(canstat);

    rx->flags= 0;
    rx->length = 0;

```

```

if( stat & CAN_DATA_OVERRUN ) {
    Overrun++;
    fprintf(stderr, "CAN Rx: Overrun Status \n");
    CANout(cancmd, CAN_CLEAR_OVERRUN_STATUS );
}

if( stat & CAN_RECEIVE_BUFFER_STATUS ) {
    dummy = CANin( frameinfo );
    if(dummy & CAN_RTR ) rx->flags |= MSG_RTR;
    if(dummy & CAN_EFF ) rx->flags |= MSG_EXT;

    ext = (dummy & CAN_EFF);

    if(ext) {
        rx->id =
        ((unsigned int)(CANin(frame.extframe.canid1)) << 21)
        + ((unsigned int)(CANin(frame.extframe.canid2)) << 13)
        + ((unsigned int)(CANin(frame.extframe.canid3)) << 5)
        + ((unsigned int)(CANin(frame.extframe.canid4)) >> 3);
    } else {
        rx->id =
        ((unsigned int)(CANin(frame.stdframe.canid1 )) << 3)
        + ((unsigned int)(CANin(frame.stdframe.canid2 )) >> 5);
    }
    dummy &= 0x0F;          // strip length code
    rx->length = dummy;

    dummy %= 9;             // limit count to 8 bytes
    for( i = 0; i < dummy; i++) {
        if(ext) rx->data[i] =
            CANin(frame.extframe.canxdata[i]);
        else rx->data[i] = CANin(frame.stdframe.candata[i]);
        usleep(10);
    }

    status = CanBuf_InQ(&Rx_buff, rx);

    if( status == -1 )
        fprintf(stderr, "CAN RX: Buffer overrun\n");

    CANout(cancmd, CAN_RELEASE_RECEIVE_BUFFER );
}
return i;
}

int CAN_VendorInit (void) {

    can_dev = mmap_device_io((size_t) Length, (uint64_t) Base);
    ThreadCtl(_NTO_TCTL_IO, 0);

    if(can_dev == MAP_DEVICE_FAILED) {
        return -1;
    }

    return 0;
}

```

```

void CAN_loopLEDs()      {
    /*
     * loopLEDs chases the LES on the breakout board in three colors
     * INPUTS
     *      (none)
     * OUTPUTS
     *      (none)
     */

    int i;
    for(i=0;i<10;i++) {
        CANout(ledreg,(uint8_t) (RED2 & YEL1 & GRN0));
        usleep(1000);
        CANout(ledreg,(uint8_t) (YEL2 & GRN1 & RED0));
        usleep(1000);
        CANout(ledreg,(uint8_t) (GRN2 & RED1 & YEL0));
        usleep(1000);
    }
    CANout(ledreg,(uint8_t) (RED2 & YEL1 & GRN0));
}

void *CAN_Interrupt ( void *data)  {
    int status;
    int i;
    int ext;                // flag for extended message format
    int irqsrc, dummy;

    canbuf_t *Rx = &Rx_buff;
    canbuf_t *Tx = &Tx_buff;

    canmsg_t cm;

    struct sigevent event;

    #if CAN_USE_FILTER
        msg_filter_t *RxPass;
        unsigned int msg_id;
        RxPass = &Rx_Filter;
    #endif

    memset(&event, 0, sizeof(event));
    event.sigev_notify=SIGEV_INTR;
    ThreadCtl( _NTO_TCTL_IO, 0 );
    interruptID = InterruptAttachEvent(IRQ, &event, 0);
    if (interruptID == -1) {
        fprintf(stderr,
            "Could not attach IRQ in CAN_Interrupt.\n");
        IRQ_requested = 0;
        return NULL;
    }

    IRQ_requested = 1;

    while(1) {
        fprintf(stderr,"** In ISR **\n");
        InterruptWait(NULL,NULL);
    }
}

```

```

    irqsrc = CANin(canirq);
    if(irqsrc == 0)
        goto IRQdone_doneNothing;        // not for me

    do {

        gettimeofday(&(cm.timestamp), NULL);

        //===== receive interrupt =====

        if( irqsrc & CAN_RECEIVE_INT ) {
            dummy = CANin(frameinfo );
            if(dummy & CAN_RTR ) cm.flags |= MSG_RTR;
            if(dummy & CAN_EFF ) cm.flags |= MSG_EXT;

            ext = (dummy & CAN_EFF);

            if(ext) {
                cm.id = ((unsigned int)(CANin(frame.extframe.canid1)) << 21)
                    + ((unsigned int)(CANin(frame.extframe.canid2)) << 13)
                    + ((unsigned int)(CANin(frame.extframe.canid3)) << 5)
                    + ((unsigned int)(CANin(frame.extframe.canid4)) >> 3);
            } else {
                cm.id = ((unsigned int)(CANin(frame.stdframe.canid1 )) << 3)
                    + ((unsigned int)(CANin(frame.stdframe.canid2 )) >> 5);
            }
            // get message length
            dummy &= 0x0F;        // strip length code
            cm.length = dummy;

            dummy %= 9;        // limit count to 8 bytes
            for( i = 0; i < dummy; i++) {
                usleep(10);
                if(ext) {
                    cm.data[i] = CANin(frame.extframe.canxdata[i]);
                } else {
                    cm.data[i] = CANin(frame.stdframe.candata[i]);
                }
            }
            status = CanBuf_InQ(Rx, &cm);

            if( status == -1 )
                fprintf(stderr,
                    "CAN RX: Buffer overrun\n");

            CANout(cancmd, CAN_RELEASE_RECEIVE_BUFFER );

            if(CANin(canstat) & CAN_DATA_OVERRUN ) {
                fprintf(stderr,
                    "CAN Rx: Overrun Status \n");
                CANout(cancmd,
                    CAN_CLEAR_OVERRUN_STATUS );
            }
        }

        //===== transmit interrupt =====

```

```

if( irqsrc & CAN_TRANSMIT_INT )      {
    uint8 tx2reg;
    unsigned int msg_id;

    InterruptDisable();
        // enter critical section

    status = CanBuf_DeQ(Tx,&cm);
    if( status == -1 ) goto Tx_done;
        // nothing to be transmitted

    tx2reg = cm.length;
    if( cm.flags & MSG_RTR ) tx2reg |= CAN_RTR;

    ext = cm.flags & MSG_EXT;
    msg_id = cm.id;
    if(ext)      {
        CANout(frameinfo, CAN_EFF + tx2reg);
        CANout(frame.extframe.canid1,
            (uint8)(msg_id >> 21));
        CANout(frame.extframe.canid2,
            (uint8)(msg_id >> 13));
        CANout(frame.extframe.canid3,
            (uint8)(msg_id >> 5));
        CANout(frame.extframe.canid4,
            (uint8)(msg_id << 3) & 0xff);
    } else {
        CANout(frameinfo, CAN_SFF + tx2reg);
        CANout(frame.stdframe.canid1,
            (uint8)((msg_id) >> 3) );
        CANout(frame.stdframe.canid2,
            (uint8)(msg_id << 5 ) & 0xe0);
    }

    tx2reg &= 0x0f;    // restore length only
    if(ext)      {
        for( i=0; i < tx2reg ; i++)      {
            CANout(frame.extframe.canxdata[i],
                cm.data[i]);
        }
    } else      {
        for( i=0; i < tx2reg ; i++)      {
            CANout(frame.stdframe.candata[i],
                cm.data[i]);
        }
    }

    CANout(cancmd, CAN_TRANSMISSION_REQUEST );

    InterruptEnable();
        // leave critical section
}
Tx_done:

//===== error status =====

if( irqsrc & CAN_ERROR_INT )  {

```

```

        fprintf(stderr, "CAN: Tx err!\n");
        TxErr++;

        // insert error
        dummy = CANin(canstat);
        if(dummy & CAN_BUS_STATUS )
            cm.flags += MSG_BUSOFF;
        if(dummy & CAN_ERROR_STATUS)
            cm.flags += MSG_PASSIVE;

        cm.id = 0xFFFFFFFF;
        cm.length = 0;
        for(i=0; i<8; i++) cm.data[i] = 0;

        CanBuf_InQ(Rx,&cm);
    }

    //===== overrun interrupt=====

    if( irqsrc & CAN_OVERRUN_INT )    {
        fprintf(stderr,"CAN: overrun!\n");
        Overrun++;

        // insert error
        dummy = CANin(canstat);
        if(dummy & CAN_DATA_OVERRUN)
            cm.flags += MSG_OVR;

        cm.id = 0xFFFFFFFF;
        cm.length = 0;
        for(i=0;i<8;i++) cm.data[i] = 0;

        CANout(cancmd, CAN_CLEAR_OVERRUN_STATUS );
    }

    } while( (irqsrc = CANin(canirq)) != 0);

    IRQdone_doneNothing:
    #if CONFIG_TIME_MEASURE
        out8(0x00, 0x378);
    #endif
    InterruptUnmask(IRQ, interruptID);
}

```

can_sysctl.c

```

/*
 * can_sysctl.c
 *
 * This version of can_sysctl.c has been modified to suit
 * the needs of the QNX operating system.  Used for global
 * variables, right now.
 *
 * version 1.2

```

```

* 3-8-2005
*/

#include <can_defs.h>

#define SYSCTL_Can 1

// ----- global variables accessible through /proc/sys/Can
char version[] = ""; // IGNORE VERSION
char IOModel;
char Chipset[] = "SJA1000";
int IRQ = 5;

// dont assume a standard address, always configure,
// address == 0 means no board available
int Base = 0x300;
int Length = 0x100;
int Baud;
unsigned int AccCode;
unsigned int AccMask;
int Timeout;
uintptr_t can_dev;

// predefined value of the output control register,
// depends of TARGET set by Makefile
int Outc;
int TxErr = 0x0;
int RxErr = 0x0;
int Overrun = 0x0;

```

can_util.c

```

/*
 * can_util.c
 *
 * version 1.2
 * 3-8-2005
 */

#include <stdio.h>
#include <sys/neutrino.h>
#include <can_defs.h>

canbuf_t Tx_buff;
canbuf_t Rx_buff;

#ifdef CAN_USE_FILTER
    msg_filter_t Rx_Filter;
#endif

int interruptID;
int IRQ_requested = 0;

#ifdef INTERRUPT_ATTACH // defined in can_resmgr
    int Can_RequestIrq(int irq, void *handler) {

```



```

        fprintf(stderr, "in Can_RequestIrq...not supposed to be
here.\n");
        interruptID = InterruptAttach (irq, handler, NULL, 0, 0);
        if (interruptID == -1) {
            fprintf(stderr, "Can't attach to IRQ %d.\n", IRQ);
            IRQ_requested = 0;
            perror (NULL);
            return 0;
        } else {
            IRQ_requested = 1;
        }
        return 1;
    }
#endif

int Can_FreeIrq(int irq )    {
    InterruptDetach(interruptID);
    IRQ_requested= 0;
    return 0;
}

//-----
//  The following functions were added to simplify message handling
//-----

int CanBuf_Init(canbuf_t *mq) {
    mq->head = 0;
    mq->tail = 0;
    mq->cnt = 0;
    return 0;
}

int CanBuf_InQ(canbuf_t *mq, canmsg_t *cm)    {
    if(mq->cnt >= MAX_BUF_SIZE) return -1;

    memcpy(&(mq->cm[mq->tail]), cm, sizeof(canmsg_t));

    (mq->cnt)++;        // inc msg counter
    (mq->tail)++;       // inc msg tail ptr

    if(mq->tail >= MAX_BUF_SIZE) mq->tail = 0;

    return 0;
}

int CanBuf_DeQ(canbuf_t *mq, canmsg_t *cm)    {
    if(mq->cnt <= 0) {
        return -1; //    nothing there
        cm = NULL;
    }

    memcpy(cm, &(mq->cm[mq->head]), sizeof(canmsg_t));

    (mq->cnt)--;        // dec msg counter
    (mq->head)++;       // inc msg head pointer

```

```

        if(mq->head >= MAX_BUFSIZE) mq->head = 0;

        return 0;
    }

int CanBuf_Count(canbuf_t *mq)    {
    return mq->cnt;
}

#ifdef CAN_USE_FILTER
int Can_FilterInit()    {
    int i;

    Rx_Filter.use = 0;
    Rx_Filter.signo[0]= 0;
    Rx_Filter.signo[1]= 0;
    Rx_Filter.signo[2]= 0;

    for(i=0;i<MAX_ID_NUMBER;i++)
        Rx_Filter.filter[i].rtr_response = NULL;

    return 0;
}

int Can_FilterCleanup() {
    int i;

    for(i=0;i<MAX_ID_NUMBER;i++) {
        if( Rx_Filter.filter[i].rtr_response != NULL )
            kfree( Rx_Filter.filter[i].rtr_response);
        Rx_Filter.filter[i].rtr_response = NULL;
    }
    return 0;
}

int Can_FilterOnOff(unsigned on)    {
    Rx_Filter.use =(on!=0);
    return 0;
}

int Can_FilterMessage(unsigned message, unsigned enable)    {
    Rx_Filter.filter[message].enable =(enable!=0);
    return 0;
}

int Can_FilterTimestamp(unsigned message, unsigned stamp){
    Rx_Filter.filter[message].timestamp =(stamp!=0);
    return 0;
}

int Can_FilterSignal(unsigned id, unsigned signal){
    if( signal <= 3 )
        Rx_Filter.filter[id].signal = signal;
    return 0;
}

```

```

    int Can_FilterSigNo(unsigned signo, unsigned signal ){
        if( signal < 3 )
            Rx_Filter.signo[signal]= signo;
        return 0;
    }
#endif

#ifdef CAN_RTR_CONFIG
    int Can_ConfigRTR( unsigned message, canmsg_t *Tx )    {
        canmsg_t *tmp;

        if((tmp = kmalloc ( sizeof(canmsg_t), GFP_ATOMIC ))== NULL
)    {
            return -1;
        }
        Rx_Filter.filter[message].rtr_response = tmp;
        memcpy( Rx_Filter.filter[message].rtr_response ,
                Tx, sizeof(canmsg_t));
        return 1
    }

    int Can_UnConfigRTR( unsigned message )    {
        canmsg_t *tmp;

        if( Rx_Filter.filter[message].rtr_response != NULL ) {
            kfree(Rx_Filter.filter[message].rtr_response);
            Rx_Filter.filter[message].rtr_response = NULL;
        }
        return 1;
    }
#endif

```

can_write.c

```

/*
 * can_write - can4linux CAN driver module
 *
 * version 1.2
 * 3-8-2005
 */

#include <stdio.h>
#include <sys/iofunc.h>
#include <errno.h>
#include <stdlib.h>
#include <can_defs.h>

__LDDK_WRITE_TYPE can_write( __LDDK_WRITE_PARAM )    {
    int        status;
    int        nbytes;
    int        xtype;
    canmsg_t    buffer;

    fprintf(stderr, "In can_write\n");
}

```

```

if ((status = iofunc_write_verify(ctp, msg, ocb, NULL)) != EOK) {
    return (_RESMGR_ERRNO(status));
}

xtype = msg -> i.xtype & _IO_XTYPE_MASK;

//    No special xtypes
if ((msg->i.xtype & _IO_XTYPE_MASK) != _IO_XTYPE_NONE)    {
    return (EINVAL);
}

nbytes = msg->i.nbytes;

if(resmgr_msgread (ctp, &buffer, nbytes, sizeof(msg->i)) == -1)
{
    return(errno);
}

InterruptDisable();    // enter critical section

if (CanBuf_Count(&Tx_buff) == 0)    {
    CAN_SendMessage(&buffer);    // Send, no wait
} else {
    CanBuf_InQ(&Tx_buff, &buffer);
}

InterruptEnable();

ocb->attr->flags |= IOFUNC_ATTR_MTIME | IOFUNC_ATTR_DIRTY_TIME;

return EOK;
}

```

can4qnx.h

```

/*
 * can4linux.h - can4linux CAN driver module
 *
 * version 1.2
 * 3-8-2005
 *
 * can4linux interface definitions
 */

#include <sys/time.h>

#ifndef __CAN_H
#define __CAN_H

//----- the can message structure
#define CAN_MSG_LENGTH 8    // < maximum length of a CAN frame

```

```

#define MSG_RTR          (1<<0)          // < RTR Message
#define MSG_OVR          (1<<1)          // < CAN controller Msg overflow error
#define MSG_EXT          (1<<2)          // < extended message format
#define MSG_PASSIVE      (1<<4)          // < controller in error passive
#define MSG_BUSOFF       (1<<5)          // < controller Bus Off
#define MSG_              (1<<6)          // <
#define MSG_BOVR         (1<<7)          // < receive/transmit buffer overflow

// mask used for detecting CAN errors in the canmsg_t flags field
#define MSG_ERR_MASK      \
    (MSG_OVR + MSG_PASSIVE + MSG_BUSOFF + MSG_BOVR)

/*
 * The CAN message structure.
 * Used for all data transfers between the application and the
 * driver using read() or write().
 * QNX: replaced 'clock_t' with 'struct timeval'
 */
typedef struct {
    int flags;    // < flags, indicating or controlling
                  // special message properties
    int cob;      // < CAN object number, used in Full CAN
    unsigned long id; // < CAN message ID, 4 bytes
    struct timeval timestamp; // < time stamp for received messages
    short int length; // < number of bytes in the CAN message
    unsigned char data[CAN_MSG_LENGTH]; // < data, 0...8 bytes
} canmsg_t;

//----- DEVCTL requests
#define COMMAND          0          // < DEVCTL command request
#define CONFIG           1          // < DEVCTL configuration request
#define SEND             2          // < DEVCTL request
#define RECEIVE          3          // < DEVCTL request
#define CONFIGURERTR     4          // < DEVCTL request
#define STATUS           5          // < DEVCTL status request
#define LOOPLEDS         6          // < DEVCTL loop Arcom LEDs

//----- CAN devctl parameter types

// DEVCTL Command request parameter structure
typedef struct Command_par {
    int cmd;        // < special driver command
    int error;       // < return value
    unsigned long retval; // < return value
} Command_par_t ;

// DEVCTL Configuration request parameter structure
typedef struct Config_par {

```

```

        int target;                // < special configuration target
        unsigned long val1;        // < 1. parameter for the target
        unsigned long val2;        // < 2. parameter for the target
        int error;                 // < return value for errno
        unsigned long retval;      // < return value
    } Config_par_t ;

// DEVCTL generic CAN controller status request parameter structure
typedef struct CanStatusPar {
    unsigned int baud;             // < actual bit rate
    unsigned int status;           // < CAN controller status register
    unsigned int error_warning_limit;
                                    // < the error warning limit
    unsigned int rx_errors;        // < content of RX error counter
    unsigned int tx_errors;        // < content of TX error counter
    unsigned int error_code;       // < content of error code register
    unsigned int rx_buffer_size;   // < size of rx buffer
    unsigned int rx_buffer_used;   // < number of messages
    unsigned int tx_buffer_size;   // < size of tx buffer
    unsigned int tx_buffer_used;   // < number of messages
    unsigned long retval;          // < return value
    unsigned int type;             // < CAN controller / driver type
} CanStatusPar_t;

// DEVCTL CanStatusPar.type CAN controller hardware chips
#define CAN_TYPE_UNSPEC           0
#define CAN_TYPE_SJA1000         1
#define CAN_TYPE_FlexCAN         2
#define CAN_TYPE_TouCAN          3
#define CAN_TYPE_82527           4
#define CAN_TYPE_TwinCAN         5

// DEVCTL Send request parameter structure
typedef struct Send_par {
    canmsg_t *Tx;                 // < CAN message struct
    int error;                     // < return value for errno
    unsigned long retval;          // < return value
} Send_par_t ;

// DEVCTL Receive request parameter structure
typedef struct Receive_par {
    canmsg_t *Rx;                 // < CAN message struct
    int error;                     // < return value for errno
    unsigned long retval;          // < return value
} Receive_par_t ;

// DEVCTL ConfigureRTR request parameter structure
typedef struct ConfigureRTR_par {
    unsigned message;              // < CAN message ID
    canmsg_t *Tx;                 // < CAN message struct
    int error;                     // < return value for errno
    unsigned long retval;          // < return value
} ConfigureRTR_par_t ;

//----- DEVCTL Command subcommands

```

```
#define CMD_START 1
#define CMD_STOP 2
#define CMD_RESET 3

//----- DEVCTL Configure targets

#define CONF_ACC      0      // mask and code
#define CONF_ACCM     1      // mask only
#define CONF_ACCC     2      // code only
#define CONF_TIMING   3      // bit timing
#define CONF_OMODE     4      // output control register
#define CONF_FILTER    5
#define CONF_FENABLE   6
#define CONF_FDISABLE  7

#endif      // __CAN_H
```

APPENDIX E

Verilog Source Code

CAN.v

```
//      Walt Ford
//      CAN Controller

module CAN(Addr, DataRd, CartData, Wr, Rd, Clk, IrqClr, can_irq_out,
can_rx, can_tx, can_off_bus, can_clk_out, can_io, can_rst, can_ale,
can_rd, can_wr, gaterw, gateaddr, can_cs_en, state);

    //      troubleshooting outputs
    output      [7:0]      can_io;      //      bus to CAN core
    output      can_rst;      //      CAN core reset
    output      can_ale; // CAN core addy latch enable
    output      can_rd;      //      CAN core rd signal
    output      can_wr;      //      CAN core wr signal
    output      gaterw;      //      r/w signal gate enable
    output      gateaddr;      //      addy signal gate enable
    output      can_cs_en;      //      chip select enable
    output      [3:0]      state;      //      current state

    //      inputs and outputs for can_bot
    input      [23:0]      Addr;      //      GBA address
    input      [7:0]      CartData;      //      GBA CartData
    input      Wr;      //      Wr strobe
    input      Rd;      //      Rd strobe
    input      Clk;
    input      IrqClr;      //      clear IRQ signal
    output      [7:0]      DataRd;      //      GBA DataRd
    output      can_off_bus; //      CAN core off-bus signal
    output      can_irq_out; //      CAN core IRQ

    //      CAN inputs and outputs for the CAN transceiver
    input      can_rx;      //      rx from CAN xceiver
    output      can_tx;      //      tx to CAN xceiver
    output      can_clk_out; //      scaled clk output

    //      Wires for the Xport interface
    wire      [7:0]      DataRd;
    wire      [23:0]      Addr;
    wire      [7:0]      CartData;
    wire      Wr;
    wire      Rd;
    wire      Clk;

    //      Wires and regs for the CAN controller interface
    reg      can_ale;      //      ^
    wire      can_rd;      //      ^
```



```

wire                can_wr;        // ^
wire                can_cs_en;     // ^
wire                [7:0] can_io;   // ^
reg                can_rst;        // ^
wire                can_off_bus;    // ^
wire                can_clk_out;    // ^
wire                can_rx;         // ^
wire                can_tx;         // ^
reg                can_irq_mask;

//          for masking irq when IrqClr signal is received
wire                can_irq_out;    // ^
wire                can_irq;

//          the IRQ signal from the CAN core


//          state machine resources
reg                [3:0] state;      // ^
reg                [3:0] next_state; // next
reg                gateaddr;        // ^
reg                gaterw;          // ^


//          state names
parameter addr1     = 4'h1;         // state names
parameter addr2     = 4'h2;         //
parameter rwst      = 4'h3;         //
parameter idle      = 4'h0;         //


//          this is the instantiation of the CAN core
can_top Inst_can_top (.rst_i(can_rst), .ale_i(can_ale),
    .rd_i(can_rd), .wr_i(can_wr), .port_0_io(can_io),
    .cs_can_i(can_cs_en), .clk_i(Clk), .rx_i(can_rx),
    .tx_o(can_tx), .bus_off_on(can_off_bus), .irq_on(can_irq),
    .clkout_o(can_clk_out));


//          handle resets -
//          whenever Addr is 0x20, set can_rst to the LSB of CartData
assign RstEn = Addr==24'hffeb20;
always @ (negedge Wr) if(RstEn) can_rst = CartData[0];


//          mask the irq if there has been an IrqClr
assign can_irq_out = can_irq_mask && (~can_irq);
always @ (posedge Clk or posedge can_rst)
    if (can_rst) can_irq_mask <= 1'b1; // unmask on reset
    else if (IrqClr) can_irq_mask <= 1'b0;
        // mask on on IrqClr
    else if (can_irq) can_irq_mask <= 1'b1;
        // when the can_irq goes away, unmask
    else can_irq_mask <= can_irq_mask;
        // otherwise, keep mask latched


//          combinational logic
assign can_cs_en =
    (Addr <= 24'hffeb1f)&&(Addr >= 24'hffeb00)&&(Rd || Wr);
    //          when addy in range and Rd or Wr
assign can_rd = Rd && gaterw; // feed Rd to core when it's time
assign can_wr = Wr && gaterw; // feed Wr to core when it's time
assign can_io =
    ~can_rd ? (gateaddr ? Addr[8:0] : CartData) : 8'hzz;

```

```

// controls I/O line to the CAN core for addresses and writes
assign DataRd = can_rd ? can_io : 8'hzz;
//      controls data line out to can_bot for reads

//      next state logic
always @ (posedge Clk or posedge can_rst)
    //      async state logic reset
    if (can_rst)      state <= idle;    //      idle on reset
    else state <= next_state; // just go to the next state

always @ (state or Clk) begin
    case (state)
        addr1: begin //      get ready for a rd or wr
            gateaddr <= 1'b1; //      send addy to core
            gaterw <= 1'b0; //      hold Rd or Wr
            can_ale <= 1'b1; // raise address latch enable
            next_state <= addr2;
        end
        addr2: begin //      create latch signal for address
            can_ale <= 1'b0; // latch address to controller
            gaterw <= 1'b0; //      still hold Rd or Wr
            gateaddr <= 1'b1; //      keep sending addy
            next_state <= rwst;
        end
        rwst: begin //      wait here until ~Rd or ~Wr
            gateaddr <= 1'b0; //      hold addy
            gaterw <= 1'b1; //      send Rd or Wr
            if (~can_cs_en) next_state <= idle;
            //      we are done when Rd or Wr goes low
            else next_state <= state;
            //      wait for Rd or Wr to finish
        end
        idle: begin
            gaterw <= 1'b0; //      don't gate anything
            gateaddr <= 1'b0; //
            can_ale <= 1'b0; //
            if (can_cs_en) next_state <= addr1;
            //      when there's another Rd or Wr start again
            else next_state <= state;
            //      otherwise wait here
        end //      idle:
    endcase
end //      always @ ...
endmodule

```

can_bot.v

```

//      Walt Ford
//      CAN example

`include "primaryint.v"
`include "bemfcont4.v"

module can_bot(CartData, CartAddr, FData, FAddr, CartCs, CartRd,
               CartWr, CartIReq, FCe, FOe, FWe, PA, PB, ClkInA, ClkInB, CPData,

```

```
CPReady, CPReset, CPDir, CPStrobe, GreenLED, RedLED, RCs, Clk,
Phi);
```

```
inout      [15:0]      CartData;
input      [7:0]       CartAddr;
inout      [7:0]       FData;
output     [20:0]      FAddr;
input      CartCs;
input      CartRd;
input      CartWr;
output     CartIReq;
output     FCe;
output     FOe;
output     FWe;
inout      [30:0]      PA;
inout      [30:0]      PB;
input      ClkInA;
input      ClkInB;
inout      [3:0]       CPData;
output     CPreedy;
input      CPReset;
input      CPDir;
input      CPStrobe;
output     GreenLED;
output     RedLED;
output     RCs;
input      Clk;
input      Phi;
```

```
reg        [15:0]      DataRd;
wire       [23:0]      Addr;
wire       Rd;
wire       Wr;
wire       Reset;
wire       [13:0]      Dummy;
wire       BemfIntStatus;
wire       BemfIntReset;
wire       Btx;
```

```
//      Xport's Primary Module
PrimaryInt InstPrimaryInt(.CartData(CartData),
    .CartAddr(CartAddr), .CartCs(CartCs), .CartRd(CartRd),
    .CartWr(CartWr), .CartIReq(CartIReq), .FData(FData),
    .FAddr(FAddr), .FCe(FCe), .FOe(FOe), .FWe(FWe),
    .CPData(CPData), .CPreedy(CPreedy), .CPReset(CPReset),
    .CPDir(CPDir), .CPStrobe(CPStrobe), .GreenLED(GreenLED),
    .RedLED(RedLED), .Addr(Addr), .Rd(Rd), .Wr(Wr),
    .SecDataRd(DataRd), .Identifier(16'h8c01),
    .IntStatus({14'h0000, CanIrq/*17*/, BemfIntStatus/*16*/}),
    .IntReset({Dummy, CanIrqClr, BemfIntReset}), .Clk(Clk),
    .Reset(Reset));
//      the interrupt vector passed to
//      CInterruptCont::Register(IIInterrupt *, unsigned char
//      vector) is determined by the interrupt request line's
//      position in the above bus IntStatus. The least
//      significant bit is vector number 16; the second is
//      17; etc.
```

```

//      end Primary

//      CAN Controller
wire [7:0] CanDataRd; //      Data being read from CAN controller
wire      CanEn;
//      High when address is in CAN controller range
wire      can_tx;
//      CAN signal being transmitted by controller
wire      can_rx; //      CAN signal being received from bus
wire [3:0] state;
wire [7:0] can_io;

//      CanEn is 'hi' when address bits are in CAN Controller range
assign CanEn =
    (Addr<=24'hffeb20) && (Addr >=24'hffeb00) && (Rd || Wr);

//      send in address bits and lower 8 data bits
CAN InstCAN (.Addr(Addr), .Clk(Clk), .CartData(CartData[7:0]),
    .DataRd(CanDataRd), .Wr(Wr), .Rd(Rd), .IrqClr(CanIrqClr),
    .can_tx(can_tx), .can_rx(can_rx), .can_irq_out(CanIrq),
    .can_off_bus(can_off_bus), .can_clk_out(can_clk_out),
    .can_io(can_io), .can_rst(can_rst), .can_ale(can_ale),
    can_rd(can_rd), .can_wr(can_wr), .gaterw(gaterw),
    .gateaddr(gateaddr), .can_cs_en(can_cs_en), .state(state));

assign can_rx      = PB[6]; //      pin 8
assign PB[0]       = can_tx; //      pin 2

//      CAN TROUBLESHOOTING
assign PB[1]       = Clk; //      pin 3
assign PB[2]       = CanEn; //      pin 4
assign PB[3]       = Wr; //      pin 5
assign PB[4]       = Rd;
assign PB[5]       = can_io[0];
assign PB[7]       = can_ale;
assign PB[8]       = can_rd;
assign PB[9]       = can_wr;
assign PB[10]      = gaterw;
assign PB[11]      = gateaddr;
assign PB[12]      = can_cs_en;
assign PB[13]      = state[0];
assign PB[14]      = can_off_bus;
assign PB[15]      = can_rst;
assign PB[16]      = 1'b0;
//      end CAN TROUBLESHOOTING
//      end CAN Controller

//      Back-EMF controller
wire BemfEn;
wire [15:0] BemfDataRd;
wire [3:0] PwmOut;
wire [7:0] PwmCont;
wire BemfAdcDir;
wire BemfAdcOut;

```

```

assign PA[24] = BemfAdcDir ? BemfAdcOut : 1'bz;
assign BemfEn = Addr[23:9]==15'h7ff1; //15'hffe2;

BemfCont4 InstBemfCont (.Addr(Addr[8:0]), .DataRd(BemfDataRd),
    .DataWr(CartData), .En(BemfEn), .Rd(Rd), .Wr(Wr),
    .PwmOut(PwmOut), .PwmCont(PwmCont), .AdcIn(PA[24]),
    .AdcOut(BemfAdcOut), .AdcDir(BemfAdcDir), .AdcCs(PA[25]),
    .AdcClk(PA[27]), .IntStatus(BemfIntStatus),
    .IntReset(BemfIntReset), .Reset(Reset), .Clk(Clk)); //
    .Measure0(PA[2]), .Active0(PA[3]));

assign PA[16] = ~(PwmCont[0]&PwmOut[0]);
assign PA[17] = ~(PwmCont[1]&PwmOut[0]);

assign PA[18] = ~(PwmCont[2]&PwmOut[1]);
assign PA[19] = ~(PwmCont[3]&PwmOut[1]);

assign PA[20] = ~(PwmCont[4]&PwmOut[2]);
assign PA[21] = ~(PwmCont[5]&PwmOut[2]);

assign PA[22] = ~(PwmCont[6]&PwmOut[3]);
assign PA[23] = ~(PwmCont[7]&PwmOut[3]);
//      end Back-EMF controller

//      Optical Encoders
wire          EncEn;
wire [15:0]    EncDataRd;
assign EncEn = Addr[23:8]==16'hffea;
Encoder InstEncoder (.Addr(Addr[3:0]), .Clk(Clk), .Wr(Wr),
    .Rd(Rd),
    .A({PA[0],PA[2],PA[4],PA[6],PA[8],PA[10],PA[12],PA[14]}),
    .B({PA[1],PA[3],PA[5],PA[7],PA[9],PA[11],PA[13],PA[15]}),
    .En(EncEn), .DataRd(EncDataRd), .CartData(CartData));
//      end Optical Encoders

//      Handle data reads with Data bus mux
//      Every time address in in CAN controller range store the
//      data from InstCAN in DataRd just in case (writes are
//      handled in InstCAN)
always @ (BemfEn or BemfDataRd or EncEn or EncDataRd
    or CanEn or CanDataRd)
begin
    if (BemfEn) DataRd = BemfDataRd;
    else if (EncEn)  DataRd = EncDataRd;
    else if (CanEn)  DataRd = {8'h00, CanDataRd};
    else DataRd = 16'hxxxx;
end

// disable SDRAM if available
assign RCs = 1'b1;
endmodule

```

APPENDIX F

CCanCore Source Code**ccancore.cxx**

```

#include "ccancore.h"
#include <intcont.h>
#include <iinterrupt.h>
#include <gba.h>
#include <string.h>

// #define _VERBOSE // print messages for certain events
// **TextDisp may not work in ISR**

CCanCore::CCanCore(CInterruptCont *pIntCont, unsigned long base,
    unsigned char vector) {
/*
 * inputs:
 *     pIntCont: pointer to interrupt controller class
 *     base:     base address for CAN Core
 *     vector:   interrupt vector for CAN interrupts
 * outputs:    [none]
 *
 * Description:
 * CCanCore initializes the CAN Core with typical settings in the
 * class constructor. Also register CCanCore class with the
 * interrupt handler from CharmedLabs.
 */

#ifdef _VERBOSE
    td.Printf("*** CCanCore **\n");
#endif
    wu_int = 0;
    ovr_int = 0;
    tx_int = 0;

    HW_reset(); // reset the CAN Core controller
    StopChip();
    DefaultInit(); // initialize class variables for configuration
    SetOMode(Outc); // set output mode register
    SetMask(AccCode, AccMask); // set mask and acc code
    SetTiming(Baud); // set timing characteristics

    m_vector = vector; // set interrupt vector
    SetInterruptCont(pIntCont); // register vector with IntCont
class
}

CCanCore::~CCanCore() {
/*

```

```

*   inputs:      [none]
*   outputs:     [none]
*   Description:
*   ~CCanCore stops the can controller by putting it into reset mode
*   and then unregisters the CAN controller interrupt with CIntCont
*   when the CCanCore class is terminated.
*/
    StopChip();
    m_pIntCont->UnRegister(m_vector);
}

//   PUBLIC:

void CCanCore::SetInterruptCont(CInterruptCont *pIntCont)  {
/*
*   inputs:
*       pIntCont:  pointer to interrupt controller class
*   outputs:      [none]
*   Description:
*   SetInterruptCont saves the pointer to pIntCont, the interrupt
*   controller, in a private CCanCore variable, m_pIntCont and
*   registers the CCanCore interrupt with the interrupt controller.
*   It also unmask the interrupt.
*/
    m_pIntCont = pIntCont;  // save pIntCont location for future use
    if (m_pIntCont)  {      // if not NULL
#ifdef _VERBOSE
        td.Printf("Register: %d\n",
            m_pIntCont->Register(this, m_vector));
#else
        m_pIntCont->Register(this, m_vector); // register interrupt
#endif
        m_pIntCont->Unmask(m_vector); // enable interrupt
    }

#ifdef _VERBOSE
    td.Printf("SetIntCont\n");
#endif
}

void CCanCore::Interrupt(unsigned char vector)  {
/*
*   inputs:
*       vector:      interrupt vector for CAN interrupts
*   outputs:        [none]
*   Description:
*   Interrupt is the interrupt handler whenever a CAN interrupt is
*   generated.  Interrupt checks the canint register to determine
*   which interrupt is generated and calls the appropriate handler.
*/
    uint8 irqsrc;

#ifdef _VERBOSE
    td.Printf("Start Int\n");
#endif
}

```

```

//    enter_critical();

    irqsrc = CANin(canint);
    if (irqsrc==0x00) return;
    else if (irqsrc & CAN_RECEIVE_INT) readMsgInt();
        //    a CAN message was received
    else if (irqsrc & CAN_OVERRUN_INT) ovrMsgInt();
        //    there was an overrun in the CAN rx fifo
    else if (irqsrc & CAN_ERROR_INT) errMsgInt();
        //    error interrupt
    else if (irqsrc & CAN_TRANSMIT_INT) txBufFreeInt();
        //    the tx buffer is newly free
    else if (irqsrc & CAN_WAKEUP_INT) wokeUpInt();
        //    the CAN core was woken up from sleep mode

//    exit_critical();

#ifdef _VERBOSE
    td.Printf("Finish Int\n");
#endif

    // if we haven't emptied it enough, disable interrupt
    if (m_pIntCont->GetStatus(m_vector))
        m_pIntCont->Mask(m_vector);
}

void CCanCore::GetStat(CanStatusPar_t *stat)    {
/*
*    inputs:
*        stat: pointer to a CanStatusPar_t struct with lots of
*              status variables
*    outputs:    [none]
*    Description:
*    GetStat retrieves a number of parameters concerning the current
*    status of the CAN controller. These are stored in the struct
*    that is passed to the function.
*/
    stat->type = CAN_TYPE_OPENCORES;
        //    defined in gbaCAN.h, almost meaningless here
    stat->baud = Baud;                //    Baud being used
    stat->control = CANin(canmode); //    current operating mode config
    stat->status = CANin(canstat);    //    current status
    stat->tx_errors = TxErr;          //    number of errors
    stat->rx_errors = RxErr;          //
    stat->overrun = Overrun;          //    number of SW buffer overruns
    stat->q_overrun = q_overrun;      //    rx queue overrun counter
    stat->ovr_int = ovr_int;
    stat->tx_int = tx_int;
    stat->wu_int = wu_int;

    enter_critical(); // don't let buffers change while reading them

    stat->rx_buffer_size = MAX_BUFSIZE;        // size of rx buffer
    stat->rx_buffer_used = Buf_Count(&Rx_buff); // number of messages
    stat->tx_buffer_size = MAX_BUFSIZE;        // size of tx buffer
    stat->tx_buffer_used = Buf_Count(&Tx_buff); // number of messages

```



```

        exit_critical(); //
    }

int CCanCore::SendMessage ( canmsg_t *tx) {
/*
 *   inputs:
 *       tx:   pointer to CAN message to be transmitted
 *   outputs:
 *       -1:   transmit queue is full
 *       0:    message saved in transmit queue
 *       number of bytes written to tx buffer + 1: otherwise
 *   Description:
 *       SendMessage is a public function that allows a user transmit the
 *       CAN message contained in tx.  SendMessage enqueues the message in
 *       Tx_buff if the TX buffer is busy.
 */
    int j;
    uint8 tx2reg, stat;

#ifdef _VERBOSE
    td.Printf("SendMsg\n");
#endif

    enter_critical();

    //   if transmit buffer is full/being used
    if ( ! (stat=CANin(canstat)) & CAN_TRANSMIT_BUFFER_ACCESS ) {
        return Buf_InQ(tx, &tx_buff);
        //   send message later, return buffer status
    }

    tx->length %= 9; // limit CAN message length to 8

    // fill the frame info and identifier fields
    tx2reg = tx->length; //   get length
    if( tx->flags & MSG_RTR) tx2reg |= CAN_RTR;
                                //   set RTR if needed

    CANout(txbuf.canid1, (uint8)(tx->id >> 3) ); // tx id1 register
    CANout(txbuf.canid2, (uint8)(tx->id << 5 ) | tx2reg);
                                //   tx id2 reg

    // - fill data -----
    for( j=0; j < tx->length ; j++)
        CANout( txbuf.candata[j], tx->data[j]);

    //   transmit what's in the queue
    CANout(cancmd, CAN_TRANSMISSION_REQUEST );

    exit_critical();

    return ++j; //   return the number of bytes to be sent + 1
}

```

```

int CCanCore::GetMessage ( canmsg_t *rx ) {
/*
*   inputs:
*       rx:   pointer to space for CAN message
*   outputs:
*       -1:   no new messages
*       0:    success
*   Description:
*   GetMessage is a public function that allow a user to read a CAN
*   message if one is available in the Rx_buff queue. GetMessage
*   copies the next message in the queue to rx using the Buf_DeQ
*   helper function and returns the return value from Buf_DeQ as
*   well.
*/

    return Buf_DeQ(rx, &Rx_buff);
}

int CCanCore::ChipReset ()    {
/*
*   inputs:      [none]
*   outputs:
*       0:       success
*       -1:      could not find CAN Controller
*   Description:
*   ChipReset puts the CAN Controller in reset mode and initializes
*   several configuration registers that can only be changed in reset
*   mode.
*/

    uint8 status;

    CANout(canmode, CAN_RESET_REQUEST); //    request reset
    if( ! (CANin(canmode) & CAN_RESET_REQUEST ) ) {
                                                //    if no response

#ifdef    _VERBOSE
        td.Printf("ERROR: no board present!");
#endif

        return -1; //    no board present
    }

    status = CANin(canstat); //    get status

    // select mode: Basic or Pelican
    // ** NOTHING IN THIS CAN SOFTWARE HAS Pelican COMPATIBILITY **
    CANout(canclk, CAN_MODE_BASICCAN | CAN_MODE_CLK);
                                                //    set clock register

    // Board specific output control
    if (Outc == 0) Outc = CAN_OUTC_VAL;
    CANout(canoutc, Outc);

    SetTiming(Baud); //    set Baud
    SetMask( AccCode, AccMask ); //    set mask
    return 0; //    success
}

```

```

void CCanCore::StartChip ()    {
/*
*   inptus:      [none]
*   outputs:     [none]
*   Description:
*   StartChip conducts some basic operations that are needed to ready
*   the CAN controller for operation mode.  Once initialized
*   completely StartChip removes the reset request and the CAN
*   controller begins operating in normal mode.
*/
    RxErr = TxErr = 0L;        //      reset error stats

    CANout(cancmd, (CAN_RELEASE_RECEIVE_BUFFER |
                    CAN_CLEAR_OVERRUN_STATUS) );

    CANin(canstat);            // clear interrupts

    // Interrupts on Rx, TX, any Status change and data overrun
    CANset(canmode, Interrupts);

    CANreset( canmode, CAN_RESET_REQUEST );    //      begin normal mode

#ifdef _VERBOSE
    td.Printf("Chip Started\n");
#endif
}

void CCanCore::StopChip ()    {
/*
*   inputs:      [none]
*   outputs:     [none]
*   Description:
*   StopChip just puts the CAN controller in reset mode.
*/

#ifdef _VERBOSE
    td.Printf("Chip Stopped\n");
#endif

    CANset(canmode, CAN_RESET_REQUEST );

}

void CCanCore::HW_reset()      {
/*
*   inputs:      [none]
*   outputs:     [none]
*   Description:
*   HW_reset resets the CAN controller using the external reset.
*   This functionality is implmented in Verilog in the module that
*   instantiates the OpenCores.org CAN controller.  This reset is
*   similar to cycling the power on the SJA1000.  THIS IS NOT THE
*   SAME AS PUTTING THE CONTROLLER IN 'RESET MODE'.
*/

```

```

        CANout(reset,0x01);        //    raise reset
        CANout(reset,0x00);        //    release reset
#ifdef _VERBOSE
        td.Printf("HW_reset\n");
#endif

}

void CCanCore::SetMask (uint8 code, uint8 mask) {
/*
 *   inputs:
 *       code: the AccCode for CAN message filtering
 *       mask: the AccMask for CAN message filtering
 *   outputs:    [none]
 *   Description:
 *   SetMask sets the AccCode and AccMask registers in the CAN
 *   Controller with the corresponding values passed to it.
 */
    CANout(acccode, code);
    CANout(accmask, mask);

#ifdef _VERBOSE
    td.Printf("Set Mask\n");
#endif
}

void CCanCore::SetTiming (int baud) {
/*
 *   inputs:
 *       baud: the baud to use
 *   outputs:    [none]
 *   Description:
 *   SetTiming sets the Timing0 and Timing 1 registers for the CAN
 *   Core to transmit and receive CAN messages.  If baud doesn't match
 *   one of the default baud rates, the first byte of the int is put
 *   into Timing Register 0 and the next 8 bits are put in Timing
 *   Register 1.
 */
    int i = 5; //    default
    int custom = 0;

    switch(baud)    {
        //    choose corresponding index
        case 10:    i = 0;    break;
        case 20:    i = 1;    break;
        case 50:    i = 2;    break;
        case 100:   i = 3;    break;
        case 125:   i = 4;    break;
        case 250:   i = 5;    break;
        case 500:   i = 6;    break;
        case 800:   i = 7;    break;
        case 1000:  i = 8;    break;
        default:    custom = 1; // if not listed, it's custom
    }

    // select mode: Basic or PeliCAN:

```

```

// ** NOTHING IN THIS CAN SOFTWARE HAS Pelican COMPATIBILITY **
CANout(canclk, CAN_MODE_BASICCAN + CAN_MODE_CLK);
if( custom ) {
    CANout(cantim0, (uint8) (baud >> 8));
    //      best guess at setting custom baud
    CANout(cantim1, (uint8) baud);
#ifdef _VERBOSE
    td.Printf(" custom bit timing BT0=0x%x
              BT1=0x%x\n",CANin(cantim0),CANin(cantim1));
#endif
} else {
    CANout(cantim0, (uint8) CanTiming[i][0]);
    //      use tables defined elsewhere
    CANout(cantim1, (uint8) CanTiming[i][1]);
}

#ifdef _VERBOSE
    td.Printf("Timing Set\n");
#endif
}

void CCanCore::SetOMode (int arg)  {
/*
 *   inputs:
 *       arg:  values to write into Output Control register
 *   outputs:  [none]
 *   Description:
 *   SetOMode sets the Output Control register with the supplied
 *   value.
 */
    CANout(canoutc, arg);

#ifdef _VERBOSE
    td.Printf("Oh Mode Set\n");
#endif
}

//      PRIVATE:
void CCanCore::readMsgInt()  {
/*
 *   inputs:      [none]
 *   outputs:      [none]
 *   Description:
 *   readMsgInt is the interrupt handler when a RX Interrupt is
 *   generated.  It reads the CAN message from the RX FIFO and
 *   enqueues is in the Rx_buff variable.
 */
    uint8 dummy=0, stat=0;
    int j=0;
    canmsg_t msg;

    msg.flags = 0;
    msg.length = 0;

```

```

    stat = CANin(canstat);

    if( stat & CAN_DATA_OVERRUN ) {
        //      if there has been a FIFO overrun
        Overrun++; //      increment overrun counter
        CANout(cancmd, CAN_CLEAR_OVERRUN_STATUS );

#ifdef _VERBOSE
        td.Printf("CAN RX buffer overrun\n");
        td.Printf("\t%d\n",status);
#endif
    }

    //      if there's a new CAN message
    if( stat & CAN_RECEIVE_BUFFER_STATUS ) {
        dummy = CANin( rxbuf.canid2 );
        if(dummy & CAN_RTR ) msg.flags |= MSG_RTR;
        //      set RTR flag
        msg.id = ((uint8)(CANin(rxbuf.canid1 )) << 3)
        //      get CAN ID
        + ((uint8)(CANin(rxbuf.canid2 )) >> 5);

        dummy &= 0x0F; // strip length code
        msg.length = dummy;

#ifdef _VERBOSE
        if (msg.length > 8)
            td.Printf("*** LENGTH IS %-2x\n",msg.length);
#endif

        dummy %= 9; // limit length to 8 bytes
        //      read in CAN data
        for( j = 0; j < dummy; j++)
            msg.data[j] = CANin(rxbuf.candata[j]);

        CANout(cancmd, CAN_RELEASE_RECEIVE_BUFFER );

        if (Buf_InQ(&msg, &Rx_buff)==-1) {
            //      enqueue received message in Rx_buff queue
            q_overrun++; //      increment queue overrun counter
#ifdef _VERBOSE
            td.Printf("CAN RX Queue Overrun\n");
#endif
        }
    }
}

void CCanCore::txBufFreeInt() {
/*
*   inputs:      [none]
*   outputs:     [none]
*   Description:
*   txBufFree is the interrupt handler when a TX Buffer Free
*   interrupt is received.  txBufFree checks the queue of messages

```

```

*   waiting to be transmitted and if there is a message waiting it
*   dequeues the message and sends the message.
*/
    uint8 tx2reg;
    canmsg_t cm;
    int j, status;

    tx_int++;

#ifdef _VERBOSE
    td.Printf("TX Buff Free\n");
#endif

    status = Buf_DeQ(&cm, &tx_buff);    //    get TX buffer status
    if( status == -1 ) return; // nothing to be transmitted

    tx2reg = cm.length; //    get length of CAN message to be sent
    if( cm.flags & MSG_RTR ) tx2reg |= CAN_RTR;
                                //    set RTR bit if needed

    CANout(txbuf.canid1, (uint8)(cm.id >> 3) );
                                //    fill ID register 1
    CANout(txbuf.canid2, (uint8)(cm.id << 5) | tx2reg); // 2

    // - fill data -----
    for( j=0; j < cm.length ; j++)
        CANout( txbuf.candata[j], cm.data[j]);

    CANout(cancmd, CAN_TRANSMISSION_REQUEST );
                                //    request to transmit
}

void CCanCore::errMsgInt()    {
/*
*   inputs:    [none]
*   outputs:   [none]
*   Description:
*   errMsgInt is the interrupt handler when a error interrupt is
*   received. Right now it doesn't do anything except print a
*   message if _VERBOSE is defined and increment both TxErr and RxErr.
*/

#ifdef _VERBOSE
    td.Printf("*** ERROR INTERRUPT **\n");
#endif
    TxErr++;
    RxErr++;
}

void CCanCore::wokeUpInt()    {
/*
*   inputs:    [none]
*   outputs:   [none]
*   wokeUpInt is the interrupt handler when the controller is woken
*   up from sleep mode. Right now it doesn't do anything except

```

```

*   print a message if _VERBOSE is defined.
*/

    wu_int++;
#ifdef _VERBOSE
    td.Printf("*** WAKE UP INTERRUPT **\n");
#endif
}

void CCanCore::ovrMsgInt()    {
/*
*   inputs:      [none]
*   outputs:     [none]
*   ovrMsgInt is the interrupt handler when a rx buffer overrun
*   interrupt is received. Right now it doesn't do anything except
*   print a message if _VERBOSE is defined.
*/

    ovr_int++;
#ifdef _VERBOSE
    td.Printf("*** BUFFER OVERRUN **\n");
#endif
}

int CCanCore::Buf_InQ(canmsg_t *cm, canbuf_t *mq)    {
/*
*   inputs:
*       cm:   pointer to CAN message
*       mq:   pointer to CAN message queue
*   outputs:
*       0:    success
*       -1:   buffer overrun
*   Description:
*   Buf_InQ is a helper function to manage the Rx and Tx queues.
*   Buf_InQ simply enqueues the message pointed to by cm into the
*   queue specified by mq.
*/
    if(mq->cnt >= MAX_BUFSIZE) return -1;    //    buffer is full

    //    copy message into next available location
    memcpy(&(mq->cm[mq->tail]), cm, sizeof(canmsg_t));

    (mq->cnt)++;    // inc msg counter
    (mq->tail)++;    // inc msg tail ptr

    //    if reached end of queue, loop to front
    if(mq->tail >= MAX_BUFSIZE) mq->tail = 0;

    return 0;    //    success
}

int CCanCore::Buf_DeQ(canmsg_t *cm, canbuf_t *mq)    {
/*
*   inputs:

```



```

*          mq:   pointer to CAN message queue
*          cm:   pointer to CAN message
*  outputs:
*          -1:   queue empty
*          0:    success
*  Description:
*  Buf_DeQ is a helper function to manage the Rx and Tx queues.
*  Buf_DeQ simply dequeues the next available CAN message from the
*  queue pointed to by mq and stores it into the CAN message pointed
*  to by cm.
*/
    if(mq->cnt <= 0)  return -1;  //    nothing there

    //    copy message from queue to canmsg_t pointer
    memcpy(cm, &(mq->cm[mq->head]), sizeof(canmsg_t));

    (mq->cnt)--;        // dec msg counter
    (mq->head)++;       // inc msg head pointer

    //    if reached end of queue, loop to front
    if(mq->head >= MAX_BUFSIZE) mq->head = 0;

    return 0;  //    success
}

int CCanCore::Buf_Count(canbuf_t *mq)    {
/*
*  inputs:
*          mq:   pointer to CAN message queue
*  outputs:
*          number of CAN messages in given queue
*  Description:
*  Buf_Count is a helper function for managing Tx and Rx queues.
*  Buf_Count simply returns the number of messages in the CAN
*  message queue pointed to by mq.
*/
    return mq->cnt;
}

void CCanCore::DefaultInit () {
/*
*  inputs:      [none]
*  outputs:     [none]
*  Description:
*  DefaultInit initializes several class variables with default
*  configuration values that will be used in starting up and
*  operating the CAN controller.  Values are defined in ccancore.h
*  except where noted.
*/
    Baud   = CAN_DEFAULT_BAUD;
    AccCode = AccMask = CAN_DEFAULT_ACC;
    Timeout = CAN_DEFAULT_TIMEOUT;
    Outc = CAN_OUTC_VAL;
    base = CAN_BASE;  //    defined in robot2.h
    Interrupts = CAN_DEFAULT_INTS;
}

```

```

        Overrun = 0;          //      set number of overruns to zero
    }

```

ccancore.h

```

#include "gbaCAN.h"
#include <gba.h>
#include "robot2.h"
#include <textdisp.h>
#include <xport.h>
#include <iinterrupt.h>
#include <intcont.h>

#ifndef _CAN_H
#define _CAN_H

#define CAN_DEFAULT_INTS CAN_BASIC_INT_RX_EN | CAN_BASIC_INT_TX_EN \
    | CAN_BASIC_INT_ERR_EN | CAN_BASIC_INT_OVR_EN
#define CAN_DEFAULT_BAUD      1000
#define CAN_DEFAULT_ACC       0xffff
#define CAN_DEFAULT_TIMEOUT   100
#define CAN_OUTC_VAL          0x8A
#define enter_critical()      GBA_REG_IME = 0x00
#define exit_critical()       GBA_REG_IME = 0x01

class CCanCore : public IInterrupt {
public:
    CCanCore(CInterruptCont *pIntCont, unsigned long base=CAN_BASE,
        unsigned char vector=17);
    //      ^=initialize chip, set default values, register interrupt

    virtual ~CCanCore();          //      stop CAN core, unregister interrupt
    void SetInterruptCont(CInterruptCont *pIntCont);
        //      register interrupt w/ Rick's IntCont class
    virtual void Interrupt(unsigned char vector);
    void GetStat(CanStatusPar_t *);
    int SendMessage ( canmsg_t *);
        //      if busy, enqueues, otherwise sends msg
    int GetMessage ( canmsg_t *);
        //      return message from rx message queue if available
    int ChipReset ();             //      set clk, mode, outc, timing, mask
    void StartChip ();
        //      clear status, clear ints, set ints, start core
    void StopChip ();             //      set reset request
    void HW_reset();
        //      disables and enables hardware in the same function
    void SetMask (uint8, uint8);  //      set AccCode and AccMask
    void SetTiming (int);         //      set tim1 and tim0 regs
    void SetOMode (int);          //      sets outc

private:
    //      CAN controller functions
    void readMsgInt();            //      reads msg from rx fifo on interrupt

```

```

void txBufFreeInt();
    // tx message from Tx_buff on tx buffer free interrupt
void errMsgInt(); // ISR for error interrupt (nothing)
void wokeUpInt(); // ISR for wake up interrupt (nothing)
void ovrMsgInt(); // ISR for buffer overrun interrupt (nothing)
int Buf_InQ(canmsg_t *, canbuf_t *);
    // helper function for buffer enqueues
int Buf_DeQ(canmsg_t *, canbuf_t *);
    // helper function for buffer dequeues
int Buf_Count(canbuf_t *);
    // returns number of messages in buffer
void DefaultInit ();
    // initializes private variables to default values

// CAN controller variables
unsigned long base; // holds CAN base
canbuf_t Rx_buff; // rx queue
canbuf_t Tx_buff; // tx queue
int Outc; // values for Output Control register
unsigned int TxErr; // tx error counter
unsigned int RxErr; // rx error counter
unsigned int Overrun; // rx buffer overrun counter
unsigned int q_overrun; // rx queue overrun counter
unsigned int Baud; // baud value
unsigned int Timeout; // timeout value
uint8 Interrupts; // value for control register (interrupts)
unsigned int AccCode; // value for AccCode register
unsigned int AccMask; // value for AccMask register
canbuf_t tx_buff; // queue for messages to be transmitted
canbuf_t rx_buff; // queue for messages to be received

int wu_int;
int tx_int;
int ovr_int;

// xport variables
CInterruptCont *m_pIntCont; //pointer to interrupt controller
unsigned char m_vector; // interrupt vector
CTextDisp td; // for text display
};

#endif

```

gbaCAN.h

```

/*
 * gbaCAN.h
 * Walt Ford
 * 9-8-2005
 */

typedef volatile unsigned short uint8;

typedef struct canregs {

```

```

uint8 canmode;           // 0
uint8 cancmd;
uint8 canstat;
uint8 canint;
uint8 acccode;
uint8 accmask;          // 5
uint8 cantim0;
uint8 cantim1;
uint8 canoutc;
uint8 cantest;
struct {
    uint8 canid1;        // 10 (0x0A)
    uint8 canid2;
    uint8 candata[8];    // 12-19 (0x0C-0x13)
} txbuf;
struct {
    uint8 canid1;        // 20 (0x14)
    uint8 canid2;
    uint8 candata[8];    // 22-29 (0x16-0x1D)
} rxbuf;
uint8 reserved;         // 30 (0x1E)
uint8 canclk;           // 31 (0x1F)
uint8 reset;            // 32 (0x20)
} canregs_t;

typedef struct CanStatusPar {
    int baud;            // < actual bit rate
    uint8 control;        // < CAN controller control register
    uint8 status;         // < CAN controller status register
    unsigned int rx_errors; // < content of RX error counter
    unsigned int tx_errors; // < content of TX error counter
    unsigned int overrun;  // < buffer overrun count
    unsigned int q_overrun; // < rx queue overrun count
    unsigned int rx_buffer_size; // < size of rx buffer
    unsigned int rx_buffer_used; // < number of messages
    unsigned int tx_buffer_size; // < size of tx buffer
    unsigned int tx_buffer_used; // < number of messages
    unsigned int type;      // < CAN controller / driver type
    unsigned int wu_int;
    unsigned int ovr_int;
    unsigned int tx_int;
} CanStatusPar_t;

#define CAN_RANGE 0x20
#define MAX_BUFSIZE 0x20
#define CAN_TYPE_OPENCORES 0x2

//--- Mode Register ----- BasicCAN -----
#define CAN_BASIC_INT_RX_EN 0x02
#define CAN_BASIC_INT_TX_EN 0x04
#define CAN_BASIC_INT_ERR_EN 0x08
#define CAN_BASIC_INT_OVR_EN 0x10
#define CAN_RESET_REQUEST 0x01 // reset mode

// bit numbers of mode register
#define CAN_SLEEP_MODE_BIT 4 // Sleep Mode

```

```

#define CAN_ACC_FILT_MASK_BIT      3      // Acceptance Filter Mask
#define CAN_SELF_TEST_MODE_BIT    2      // Self test mode
#define CAN_LISTEN_ONLY_MODE_BIT  1      // Listen only mode
#define CAN_RESET_REQUEST_BIT     0      // reset mode

```

```

//--- Interrupt enable Reg -----

```

```

#define CAN_ERROR_BUSOFF_INT_ENABLE (1<<7)
#define CAN_ARBTR_LOST_INT_ENABLE  (1<<6)
#define CAN_ERROR_PASSIVE_INT_ENABLE (1<<5)
#define CAN_WAKEUP_INT_ENABLE      (1<<4)
#define CAN_OVERRUN_INT_ENABLE     (1<<3)
#define CAN_ERROR_INT_ENABLE       (1<<2)
#define CAN_TRANSMIT_INT_ENABLE    (1<<1)
#define CAN_RECEIVE_INT_ENABLE     (1<<0)

```

```

//--- Frame information register -----

```

```

#define CAN_EFF      0x80 // extended frame
#define CAN_SFF      0x00 // standard frame format

```

```

//--- Command Register -----

```

```

#define CAN_GOTO_SLEEP      (1<<4) // this can be strange
#define CAN_CLEAR_OVERRUN_STATUS (1<<3)
#define CAN_RELEASE_RECEIVE_BUFFER (1<<2)
#define CAN_ABORT_TRANSMISSION (1<<1)
#define CAN_TRANSMISSION_REQUEST (1<<0)

```

```

//--- Status Register -----

```

```

#define CAN_BUS_STATUS (1<<7)
#define CAN_ERROR_STATUS (1<<6)
#define CAN_TRANSMIT_STATUS (1<<5)
#define CAN_RECEIVE_STATUS (1<<4)
#define CAN_TRANSMISSION_COMPLETE_STATUS (1<<3)
#define CAN_TRANSMIT_BUFFER_ACCESS (1<<2)
#define CAN_DATA_OVERRUN (1<<1)
#define CAN_RECEIVE_BUFFER_STATUS (1<<0)

```

```

//--- Interrupt Register -----

```

```

#define CAN_WAKEUP_INT (1<<4)
#define CAN_OVERRUN_INT (1<<3)
#define CAN_ERROR_INT (1<<2)
#define CAN_TRANSMIT_INT (1<<1)
#define CAN_RECEIVE_INT (1<<0)

```

```

//--- Output Control Register -----

```

```

#define CAN_OCTP1 (1<<7)
#define CAN_OCTN1 (1<<6)
#define CAN_OCPOL1 (1<<5)
#define CAN_OCTP0 (1<<4)
#define CAN_OCTN0 (1<<3)
#define CAN_OCPOL0 (1<<2)
#define CAN_OCMODE1 (1<<1)
#define CAN_OCMODE0 (1<<0)

```

```

//--- Clock Divider register -----

```

```

#define CAN_MODE_BASICCAN (0x00)

```

```

#define CAN_MODE_PELICAN      (0xC0)
#define CAN_MODE_CLK          (0x07)      // CLK-out = Fclk
#define CAN_MODE_CLK2         (0x00)      // CLK-out = Fclk/2

//--- Remote Request -----
#define      CAN_RTR              0x10

//----- Timing values
#define CAN_SYSCLK 50

#if CAN_SYSCLK == 50
    // these timings are valid for clock 50Mhz
    #define CAN_TIM0_10K      0x00
        // setting not available at this frequency
    #define CAN_TIM1_10K      0x00
        // setting not available at this frequency
    #define CAN_TIM0_20K      0x31
    #define CAN_TIM1_20K      0x7f
    #define CAN_TIM0_40K      0x18
    #define CAN_TIM1_40K      0x7f
    #define CAN_TIM0_50K      0x13
    #define CAN_TIM1_50K      0xff
    #define CAN_TIM0_100K     0x09
    #define CAN_TIM1_100K     0x7f
    #define CAN_TIM0_125K     0x07
    #define CAN_TIM1_125K     0x7f
    #define CAN_TIM0_250K     0x03
    #define CAN_TIM1_250K     0x7f
    #define CAN_TIM0_500K     0x01
    #define CAN_TIM1_500K     0x7f
    #define CAN_TIM0_800K     0x00
        // setting not available at this frequency
    #define CAN_TIM1_800K     0x00
        // setting not available at this frequency
    #define CAN_TIM0_1000K    0x00
    #define CAN_TIM1_1000K    0x7f

    #define CAN_SYSCLK_is_ok      1
#endif

#if CAN_SYSCLK == 8
    // these timings are valid for clock 8Mhz
    #define CAN_TIM0_10K      49
    #define CAN_TIM1_10K      0x1c
    #define CAN_TIM0_20K      24
    #define CAN_TIM1_20K      0x1c
    #define CAN_TIM0_40K      0x89 // Old Bit Timing Standard of port
    #define CAN_TIM1_40K      0xEB // Old Bit Timing Standard of port
    #define CAN_TIM0_50K      9
    #define CAN_TIM1_50K      0x1c
    #define CAN_TIM0_100K     4 // sp 87%, 16 abtastungen, sjw 1
    #define CAN_TIM1_100K     0x1c
    #define CAN_TIM0_125K     3
    #define CAN_TIM1_125K     0x1c
    #define CAN_TIM0_250K     1
    #define CAN_TIM1_250K     0x1c
    #define CAN_TIM0_500K     0

```

```

#define CAN_TIM1_500K    0x1c
#define CAN_TIM0_800K    0
#define CAN_TIM1_800K    0x16
#define CAN_TIM0_1000K   0
#define CAN_TIM1_1000K   0x14

#define CAN_SYSCCLK_is_ok    1
#endif

#if CAN_SYSCCLK == 10
    // these timings are valid for clock 10Mhz
    // 20 Mhz crystal
#define CAN_TIM0_10K    0x31
#define CAN_TIM1_10K    0x2f
#define CAN_TIM0_20K    0x18
#define CAN_TIM1_20K    0x2f
#define CAN_TIM0_50K    0x18
#define CAN_TIM1_50K    0x05
#define CAN_TIM0_100K   0x04
#define CAN_TIM1_100K   0x2f
#define CAN_TIM0_125K   0x04
#define CAN_TIM1_125K   0x1c
#define CAN_TIM0_250K   0x04
#define CAN_TIM1_250K   0x05
#define CAN_TIM0_500K   0x00
#define CAN_TIM1_500K   0x2f
#define CAN_TIM0_800K   0x00
#define CAN_TIM1_800K   0x00
#define CAN_TIM0_1000K  0x00
#define CAN_TIM1_1000K  0x07

#define CAN_SYSCCLK_is_ok    1
#endif

#if CAN_SYSCCLK == 16
    // these timings are valid for clock 16Mhz
#define CAN_TIM0_10K    0x1F
#define CAN_TIM1_10K    0x7F
#define CAN_TIM0_20K    0x0F
#define CAN_TIM1_20K    0x7F
#define CAN_TIM0_50K    0x07
#define CAN_TIM1_50K    0x7A
#define CAN_TIM0_100K   0x03
#define CAN_TIM1_100K   0x7A
#define CAN_TIM0_125K   0x03
#define CAN_TIM1_125K   0x76
#define CAN_TIM0_250K   0x01
#define CAN_TIM1_250K   0x76
#define CAN_TIM0_500K   0x00
#define CAN_TIM1_500K   0x76
#define CAN_TIM0_800K   0x00
#define CAN_TIM1_800K   0x43
#define CAN_TIM0_1000K  0x00
#define CAN_TIM1_1000K  0x32

```

```

        #define CAN_SYSClk_is_ok 1
    #endif

    // timing values
    uint8 CanTiming[10][2]={
        {CAN_TIM0_10K,CAN_TIM1_10K},
        {CAN_TIM0_20K,CAN_TIM1_20K},
        {CAN_TIM0_50K,CAN_TIM1_50K},
        {CAN_TIM0_100K, CAN_TIM1_100K},
        {CAN_TIM0_125K, CAN_TIM1_125K},
        {CAN_TIM0_250K, CAN_TIM1_250K},
        {CAN_TIM0_500K, CAN_TIM1_500K},
        {CAN_TIM0_800K, CAN_TIM1_800K},
        {CAN_TIM0_1000K,CAN_TIM1_1000K}
    };

    #ifndef CAN_SYSClk_is_ok
        #error Please specify a valid CAN_SYSClk value (i.e. 8, 10, 16, \
            50) or define new parameters
    #endif

    #define CAN_MSG_LENGTH 8          // < maximum length of a CAN frame
    #define MSG_RTR          (1<<0)  // < RTR Message

    typedef struct {
        int flags;          // < flags, indicating or controlling
                           //      special message properties
        int cob;            // < CAN object number, used in Full CAN
        unsigned long id;   // < CAN message ID, 4 bytes
        short int length;   // < number of bytes in the CAN message
        unsigned char data[CAN_MSG_LENGTH]; // < data, 0...8 bytes
    } canmsg_t;

    typedef struct {
        int head;
        int tail;
        int cnt;
        canmsg_t cm[MAX_BUFSIZE];
    } canbuf_t;

    #define CANout(adr,v)    (((canregs_t *)base)->adr = v
    #define CANin(adr)       (((canregs_t *)base)->adr) & 0x00ff
    #define CANset(adr,m)    (((canregs_t *)base)->adr = \
                           (((canregs_t *)base)->adr) | m) & 0x00ff
    #define CANreset(adr,m) (((canregs_t *)base)->adr = \
                           ((canregs_t *)base)->adr & (0x0000 + ~m)
    #define CANTest(adr,m)  (((canregs_t *)base)->adr & m

```


BIBLIOGRAPHY

1. Pazul, Keith. "Controller Area Network (CAN) Basics". Application Note AN713, Microchip Technology, 1999.
2. *CAN Specification, Version 2.0*. BOSCH. Robert Bosch GmbH, 1991.
3. Krten, Rob. *Getting Started with QNX Neutrino 2: A Guide for Realtime Programmers*. PARSE Software Devices, 2001.
4. "AIM104-CAN". Arcom Control Systems. Data Sheet. Arcom Control Systems Ltd, 1999.
5. "Spartan II 2.5V FPGA Family: Complete Data Sheet". Xilinx, Inc., 2004.
6. "Xport Robot Controller". Charmed Labs LLC. 30 September 2005 <<http://www.charmedlabs.com>>.
7. "CAN Protocol Controller: Overview". OpenCore.org. 29 September 2005 <<http://www.opencores.org/projects.cgi/web/can/overview>>.
8. "can4linux – CAN network device driver". can4linux. <<http://www.port.de/software/can4linux/index.html>>.
9. Gravagne, Ian; et. al. "Real-Time Distributed Control Networks: Dynamic Bandwidth Allocation via Adaptive Sampling." NSF Proposal. Baylor University, 2003.
10. Weiss, Giselle. "Little Robotic Autos That Can," *IEEE Spectrum*, vol 39, (7) pp. 24 – 25, 2002.
11. Ferreira, J.; Pedreiras, P.; Almeida, L.; Fonseca, J.A. "The FTT-CAN protocol for flexibility in safety-critical systems," *IEEE Micro*, vol 22, (4) pp. 46 – 55, 2002.
12. Cena, G.; Valenzano, A., "An improved CAN fieldbus for industrial applications," *IEEE Transactions on Industrial Electronics*, vol 44, (4) pp. 553 – 564, 1997.
13. CiA. *CAN physical layer*. 20 September 2005 <<http://www.can-cia.org/can/physical-layer/index.html>>.
14. Kvaser. 20 September 2005 <<http://www.kvaser.com/index.htm>>.

15. “Real-time operating system.” *Wikipedia: The Free Encyclopedia*. 28 September 2005 <http://en.wikipedia.org/wiki/Real-time_operating_system>.
16. “Writing an Interrupt Handler.” QNX.com, 25 September 2005
<http://www.qnx.com/developers/docs/qnx_6.1_docs/neutrino/prog/inthandler.html>.
17. Korth, Martin. “GBATEK.” <<http://www.work.de/nocash/gbatek.htm>>.
18. LeGrand, Rich. “FPGA Rd timing.” 28 September 2005
<<http://www.charmedlabs.com>> discussion board. Posted: Wed Oct 27, 2004.
19. Bohner, Martin; Peterson, Allan. *Dynamic Equations on Time Scales: An Introduction with Applications*. Boston: Birkhäuser, 2001.